Name: Ari Porad

Collaborators: None

Homework 5

The git repository hws/5 directory is the basis for this homework. Space for answers intentionally left out - include these in your single pdf in the submission zip file. Also this grew beyond last year's version based on a lot of recent feedback. Let me know if this is on the right track!

Reading

- ***The verilog cheatsheet was just updated to include a ton of extra tips. Re-read it!***
- Chapter 5: 5.2 to 5.2.4, 5.4, 5.5
  - Optional: 5.3 Number Systems - Highly encouraged for anyone interested in scientific computing!
  - Next Time: 5.2, 5.6
- Context: "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information" G A Miller [pdf]
- Did you know that there is a comprehensive gtkwave manual? See Q4.

1. Context Beyond CompArch - Psychology/Information Theory

Read "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information" G A Miller. This is one of the most cited papers in the field of psychology, but it has tremendous implications to appropriate design of digital systems.

a) Describe at least one situation in which you had to use or measure more bits than were useful to a problem.

Digits of Pi in Elementary School! Memorizing digits (like 20+) of Pi was such a big thing back then, but actually I could use 22/7 (or, heck, 3) for everything I'll ever do and almost certainly be totally fine.

b) Have you ever had a situation where you didn't use enough bits?

Uhh… yes? I mean there have definitely been situations where I didn't fully appreciate the nuance or detail of a situation/question.

Specific example: I have almost no sense of smell. I would subjectively estimate that my nose has 10±2 bits of precision (relative to all scents in the world, ie. I think I can only smell 256-4096 things).

c) How many bits did you use to answer the previous question?

See above—I'd guestimate 10±2.

d) In your own words, what is the difference between a bit and a chunk?

A bit is the ability to make a yes-or-no decision. It's an amount of information. A chunk is a grouping of symbols/options/whatevers that are re-coded as a single symbol, like how my phone number is "twoohsix" "ninefouroh" "foureightsixsix" instead of "two" "oh" "six" "nine" "four" "oh" "four" "eight" "six" "six". A chunk of four decimal digits has the same number of symbols as four binary digits, but far more information.

e) How are the generalizations from this paper still applicable half a century later? Alternatively, what do you feel no longer applies?

I don't know that I think time plays a big factor here, since the underlying system (human perception) hasn't changed all that much… There are some things I think this paper missed, but I think all of them (*maybe* with one exception) should have been obvious at the time.

Specifically, I think the paper really missed two big ideas: First, I think a human brain stores a binary digit and a decimal digit in the same way. I'm surprised that the author was surprised that you can remember a roughly equal number of binary digits and decimal digits. Second, they slightly hinted at but didn't really address the notion of semantic memory: we don't remember words as words, we remember words as concepts, and remember those concepts nonlinearly based on interest, etc. It seems immensely obvious that you'd be able to differentiate between, for example, ten different animals than ten meaningless tones. (Admittedly, this framing is perhaps influenced by having some understanding of artificial intelligence and semantic understanding in a computational context.) See also: [Correct Horse Battery Staple](#).

f) (optional) This is a paper I feel everyone should read[1], do you have an equivalent must-read scientific article?

2. Combinational Review: ALU Part Two

Use only structural combinational logic (no flops, no ifs/elses, etc.). That means use `always_comb` statements with only `~&|^?` operators for these modules. A working adder is included in the homework folder.
- Implement a `compare_eq` module that compares two n-bit numbers and outputs high if they are equal, low if they are not.

---

[1] I have sent this to family, friends, and colleagues. Amazingly I'm still on some of those group chats.

- Implement a compare_lt (less than "<" operator) that can support up to 32-bit inputs, that outputs high if a is less than b.
- Make sure the Makefile can still run the test (include any of the submodules in the target!).
- Augment the test_comparators.sv example with more test cases to give you confidence that you have implemented the two comparisons correctly.
- Descriptions **and** schematics in the top level PDF that show your approach to the two comparators.
- Bonus: Use the cheatsheet to load in test vectors from a memh/memb file. Update make test* and submission accordingly. *You'll get a guided version of this next homework.*

Confidence/Skills Check

This should feel straightforward at the block diagram level, but possibly still tricky in terms of execution in SystemVerilog. Contact an instructor ASAP if you don't know how to start, reach out to peers after ~15 min of (re)reading if you feel you can't sketch a block diagram, then follow up with more instructor time if you still feel shaky. Only start writing HDL after you have a block diagram that is at least vetted by a peer! Reach out to instructors after ~5-10 min of debugging HDL, Makefile, verilator, iverilog, or gtkwave issues.

3. Design Challenge/Lab Prep

Implement the following using **synchronous** and **combinational** logic (i.e. only `always_ff`, `always_comb` allowed). You can either go behavioral (ifs, elses, ==, <,>, etc) or use the structural modules from the prior question/labs. Using `generates`/`for` loops is **not allowed**. For each module **you must include a schematic sketch in your top level PDF for full credit!**
   ● [straightforward] pulse_generator.sv
   ● [getting a little tricky] triangle_generator.sv
   ● [putting it all together] pwm.sv

Specifications for the modules are in the provided stub files. This is a lot to do, but if you manage time well **and ask questions early** you'll find that (a) all of these modules use the same tricks of combining counters, comparators, and registers (with a simple 1-2bit FSM), and (b) each module leverages techniques from the previous one to do more (this doesn't mean you should re-use the previous module, more that the same ideas apply).

Confidence/Skills Check

Getting to the block diagram level for each of these modules is the hardest part. I recommend ~15 min. trying to come up with a block diagram solo, ~15 min. working with peers (please cite them), and then getting instructor help before proceeding. Once you have a block diagram (labeled with bus widths, good wire names, etc.) don't spend more than ~5 min. or so stalled on syntax/verilog issues. Get verilator linting in your editor asap, it will save you a ton of time! See the next question

4. Tool Usage - gtkwave/Makefile practice

A poor engineer blames their tools. A garbage engineer doesn't even use them correctly[2].Good engineers RTFM (Read The Full[3] Manual).

I've been noticing a strange reluctance to use gtkwave to analyze results. Printing ($display-ing) is barely sufficient for combination logic; sequential demands more. All the tools have easily searchable manuals, this question is about forcing you to RTFM. The homework folder includes an example of using a Makefile with a `waves_*` target that you can reverse engineer, but to get full credit you should have added Makefile targets `waves_pulse_generator`, `waves_triangle_generator`, `waves_pwm` that call your saved gtkw configurations - when the instructors call make waves_* we should get the same waveform (with comments!).2
For special fun - right click a waveform and select the analog option for the triangle wave generator.

---

[2] The unofficial ending to that joke is that a professor/engineer just destroys the tool, making more work for the technicians. I was gonna make you guess which institution taught me this, but the answer is all of them. Literally everywhere I've worked or interned or contracted at. And it applies to software too! Listen to the techs, they know more than you ever will, and appreciate the vice versa.
[3] I know, the F doesn't stand for Full.

== comparator

Let a, b = one bit

a XOR b = 1 iff a and b differ

∴ A XNOR b = 1 iff a and b match

 — equals

If a, b have size N, then a XOR b has len N, w/ each bit of output corresponding to that bit matching.

The whole thing matches if every bit of the XNOR output is 1. This is an AND:

 equals

< Comparator

We want: $a < b$ → a, b have len N, big-endian, signed 2's complement

$$= a - b < 0$$

$$= \text{is negative}(a-b)$$

$$= \text{sign bit is high }(a-b)$$

$$= (a-b)[N-1]$$

$$= (a + (-b))[N-1]$$

$$\quad\quad\quad -b = \bar{b} + 1$$

$$= (a + \bar{b} + 1)[N-1]$$

$$c_{in} = 1$$

a $\overset{N}{/}$ ... $c_{in}=1$ ... + ... $\overset{N}{/}$ ... N-1th bit out

b $\overset{N}{/}$ NOT $\overset{N}{/}$ ... N-2...0 bits discard
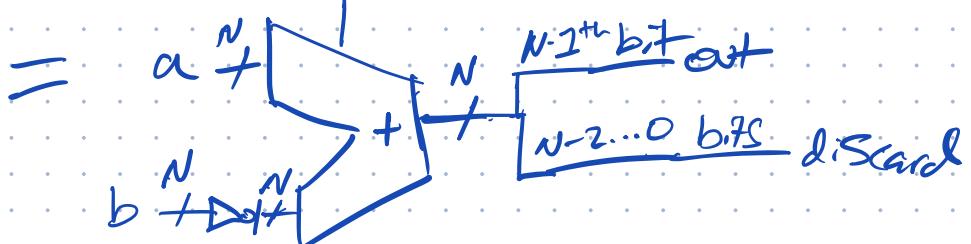
Caveat!: Will overflow & produce nonsense if a, b are large & opposite signs! (If same sign, one will be negated so the result will be closer to 0.) Next page →
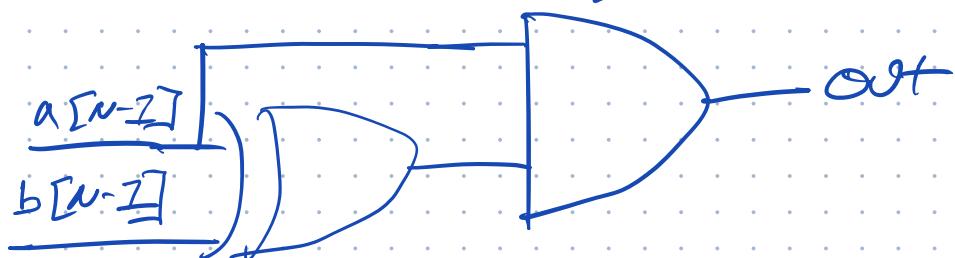
Solution: If signs differ, easy to compute answer manually:

Truth Table:

Sign bits

a[N-1]	b[N-1]	Out
0	0	N/A
1	1	N/A
0 (POS)	1 (Neg)	0
1	0	1
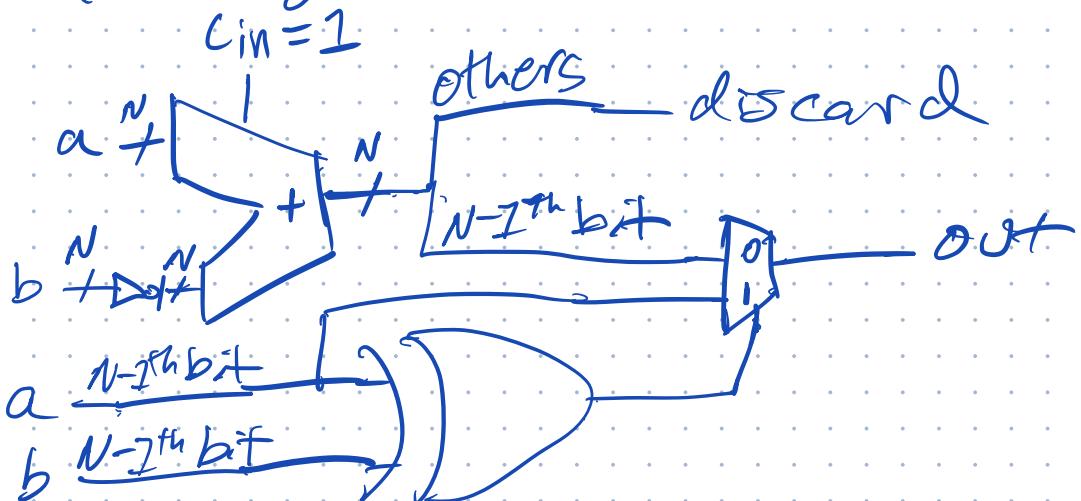
So: If $a[N-1] \mathrel{!=} b[N-1]$ (XOR!), then answer is $a[N-1]$

All together:

$c_{in} = 1$
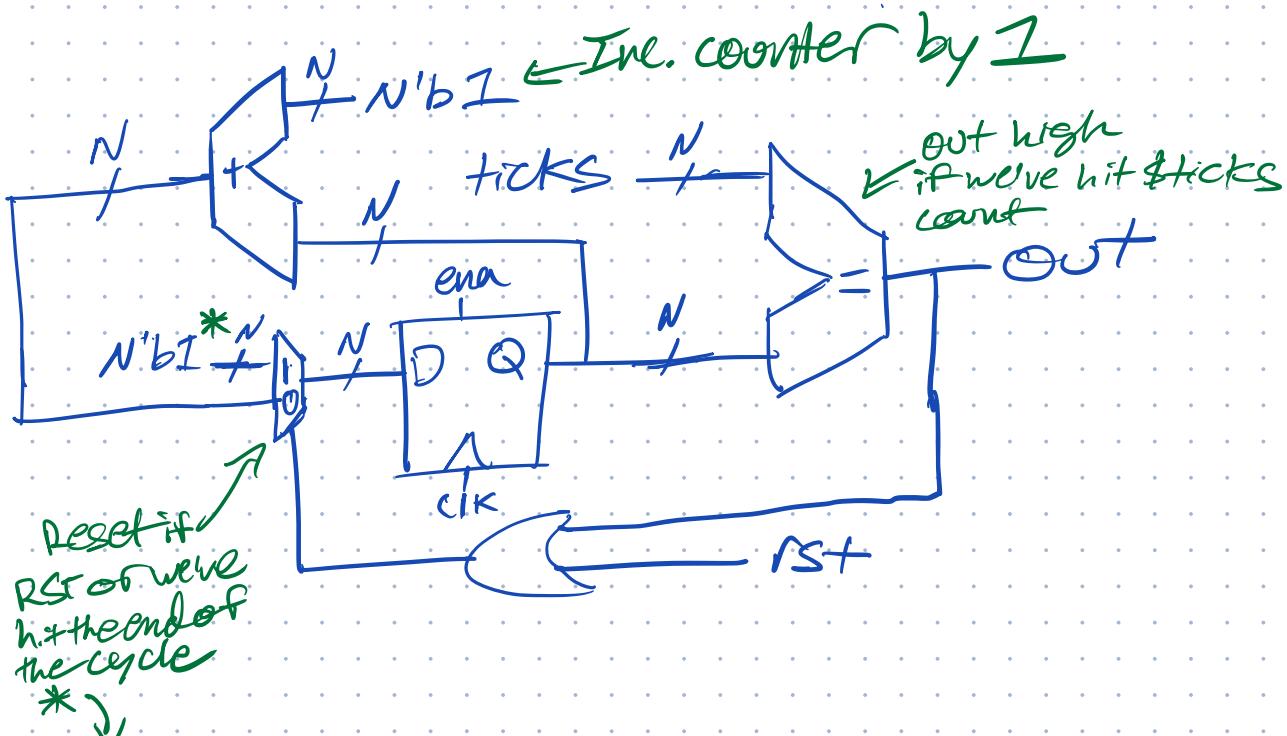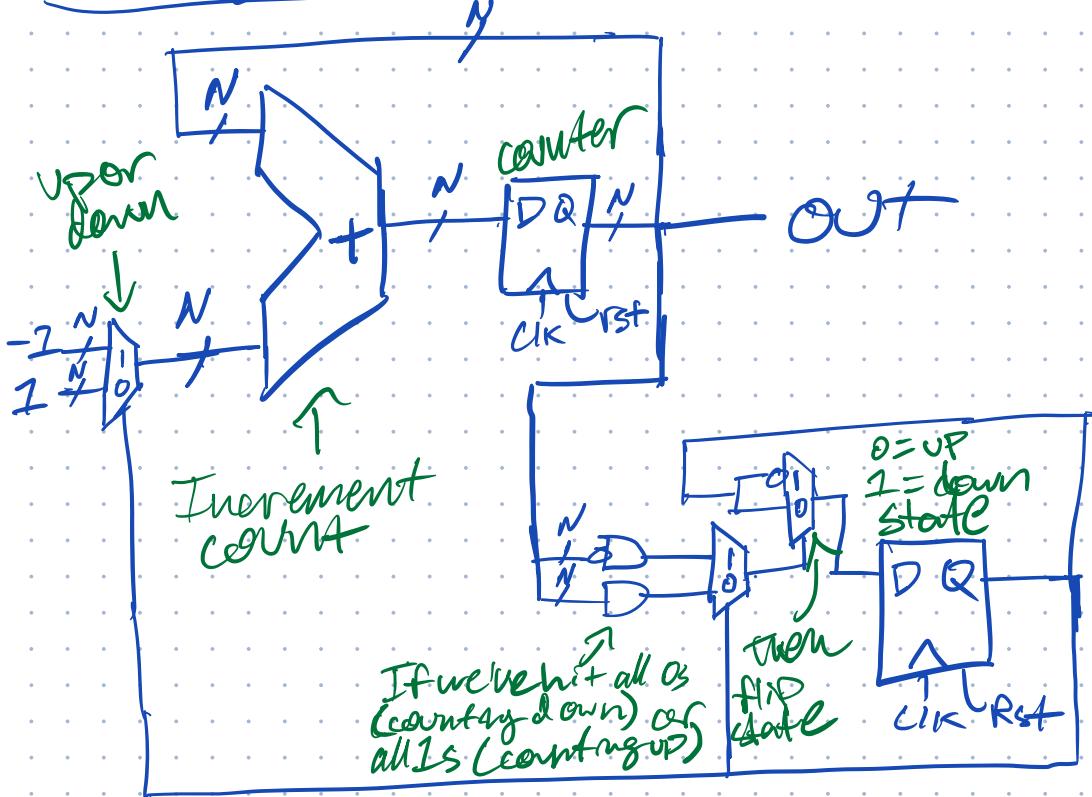
Pulse Generator

Out is high every $ticks ticks

Inc. counter by 1

out high if we've hit $ticks count

OUT

ticks

ena

D Q

CLK

N'b1 * N

Reset if RST or we've hit the end of the cycle *

rst

The spec is slightly unclear here, I'm reading it as the output should be high for 1/ticks clock cycles (ie. low for ticks-1 cycles then high for 1). It might be low for ticks cycles then high for 1, in which case this should be 0. (Same below)

Triangle Generator

Up or down

-1

1

Increment count

counter

D Q

clk rst

OUT

If we've hit all 0s (counting down) or all 1s (counting up)

then flip state

0 = up
1 = down
State

D Q

clk Rst

PWM

Increment counter
↓

N'b1

Go high for first
$duty - 1$ ticks of cycle, or
always if $duty$ is
all 1s

duty

ena RST

D Q

CLK counter

No updates
unless
step = 1

Step

output off
if ena = 0

ena

out