

# Homework 6

*The git repository hws/6 directory is the basis for this homework. Space for answers intentionally left out - include these in your single pdf in the submission zip file.*

## Reading

- Reread:
  - H&H 3.5: Timing, Metastability, except 3.5.6
    - Definitely read sections 3 and 4 of this guide (the rest is great too, read it if you have time!) [Synchronization and Metastability, Steve Golson, Synopsys Users Group Conference 2014](#)
  - H&H 5.2.5: Shifters and Rotators
  - H&H 5.5: Memory Arrays
- New:
  - H&H 6.1 to 6.4
  - H&H 7.1-7.2

## 0. Spiral 2 Feedback

Please fill out this [form](#) to give me feedback on how the last spiral through the course went.

## 1. Metastability and Designing for Failure

### (a) Reasonable MTBFs (Mean Times Between/Before Failure)

For each of the following systems, pick an MTBF (specify your time unit!) that you think makes sense as a design requirement for the entire system. Note that for systems with constant probability of failure rates ~66% of units will have failed at least once before the MTBF has elapsed, and that in the case of a metastability failure a full system reset (power cycling) is required. Write a sentence explaining why you chose your number for the application.

Toddler Toy Piano (ages 2 to 4):

Something on the order of 5 years (ie. substantially longer than one toddler would own the toy). A failure isn't a huge problem (assuming the failure is like an annoying noise, not a fire), but probably diminishes our brand's reputation. If our toy brand cares a lot about quality, then maybe closer to 10-15 years to account for siblings and safety margin.

Industrial Robot Arm:

I think this depends a *lot* on what the arm is doing, and how bad it is if a failure occurs. For something safety-critical, perhaps several hundred years? If its an arm used for painting cars, then maybe closer to a dozen years, since power-cycling isn't a huge issue but has the potential to cost time and cause logjams.

Vehicle ADAS (Automated Driver Assistance System):

Who cares! Metastability probably isn't going to be the thing that kills you.

More seriously: Let's say every car needs to be refueled at least once every ~3 hours, and assume that refueling/recharging fully restarts the ADAS system (which, IMO, is probably a good idea). If a failure means that the driver needs to take control, then I'd say a MTBF on the order of a dozen years should be sufficient to ensure the majority of people never encounter an issue. If the failure mode is a crash, then perhaps on the order of several hundred years is better for safety.

### (b) Case Study: Toy Piano

Our toy piano has 24 different digital asynchronous inputs (keys) that an eager toddler can hit very quickly. Using Equations (24) and (25) from the Synchronization and Metastability guide (page 22), and the following device parameters, determine the fastest clock speed  $f_c$  that lets the **full system** meet the MTBF you found in part (a). You can assume the following device parameters:

- $\tau = 175 \text{ ps}$
- $T_o = 225 \text{ ps}$
- $f_d = 10 \text{ Hz}$  (these are very eager toddlers)
- $t_R = t_{\text{setup}} - T_c$ , where  $T_c = 1/f_d$  and  $t_{\text{setup}} = 200 \text{ ps}$

This problem isn't supposed to be about annoying algebra, so there is a python script in the homework folder that shows you how to sweep some frequencies on a log scale to see what works (there's no analytical solution for this, so you have to guess and check).

$T_c = 250\text{MHz}$  gives us a system MTBF of 6.3 years, which feels roughly appropriate to me and is a nice round number. If you want to stick to exactly the 5 years I said before, then  $T_c = 252.5\text{MHz}$  gives you a system MTBF of 5 years exactly.

## 2. Combinational Review: A complete 32-bit ALU

The “thinky” element in our custom RISC-V CPU is the component that can do just about any math we need for reasonable computation. This is called an Arithmetic Logic Unit, or ALU. You should have all the building blocks for this module.

***Use only structural combinational logic (no flops, no ifs/elses, etc.).*** That means use always\_comb statements with only `~&` | `^?` operators for these modules. Working adders and muxes are included in the stub folder. There are many opportunities to be clever with submodule re-use, but remember that it is better to have a working large or slow ALU than a very fast but incomplete one. The only new features that shouldn't be in prior examples/solutions are:

- Implement an SLL (shift left logical) operation that shifts input a to the left by b bits. The result should be padded with zeros.
- Implement an SRL (shift right logical) operation that shifts input a to the right by b bits. The result should be padded with zeros.
- Implement an SRA (shift right logical) operation that shifts input a to the right by b bits. This result should be “sign extended” - that means that you need to copy the most significant bit.
- **STRETCH** - add correct overflow logic so that the ALU reports if an operation resulted in a 32 bit overflow. Only do this if you've finished the rest of the assignment.

You should augment the test\_alu.sv example with more test cases to give you confidence that you have implemented the different operations correctly. Include descriptions **and** schematics in the top level PDF that show your approach to the shifters.

### Confidence/Skills Check

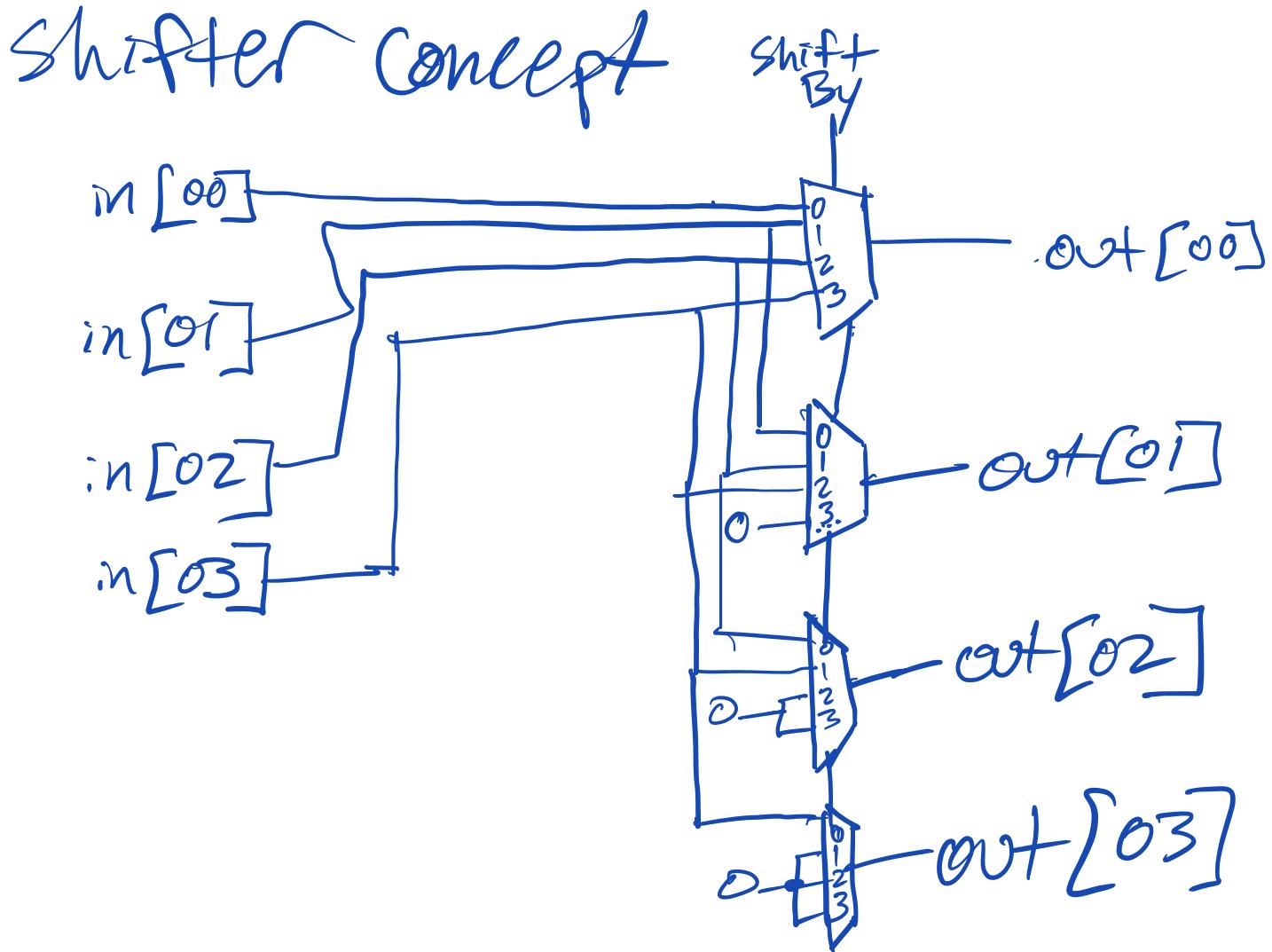
With the hints from the reading this should be starting to feel straightforward in theory but a little tricky in execution (there are a lot of wires to deal with in 32 bit systems). See the solution muxes for examples of using python to auto generate long connection lists.

# Shifters

I conceptualized the shifters as being a set of muxes--one mux for each bit of the output--where the select is the number of bits to shift and the inputs are the appropriately shifted bits (ie. the mux for SRL bit 3 has in00 = in[3], in01 = in[4], etc.).

However, I didn't want to do the ugly thing of using Python to generate 32 separate inputs for each mux. Instead, I built a new type of mux (called a "one bit mux", although perhaps that's not quite the right word), which takes all of its inputs as one port (of length 32) but can only output one bit.

This made it easy to use a generate statement for all the muxes. Each one was given a sliced version of the input, relying on the fact that verilog pads short port values with 0s. SRA has to specially front-pad with the sign bit. For SLL, I had to specifically reverse the input since `in[0:31]` isn't valid in Verilog. I think the whole system is fully parametric for  $N \neq 32$ , but I haven't tested that.



### 3. Sequential Review - Register File

Implement a 32-bit register file using **structural synchronous** and **combinational** logic (i.e. only `always_ff`, `always_comb` with basic ops `~&|?` allowed). A working `register.sv` file is provided, but you will need to add the other modules together to create this file. Hint, there are some combinational modules not included in the folder that you might need for this one.

#### Confidence/Skills Check

This is more about understanding how a register file is supposed to work and getting better at wiring large systems together - the underlying circuitry is simple once you understand how a register file is supposed to work.