EECS 151/251A FPGA Lab 4: Memories, IO Circuits, FSMs

Prof. Borivoje Nikolic and Prof. Sophia Shao
TAs: Cem Yalcin, Rebekah Zhao, Ryan Kaveh, Vighnesh Iyer
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Before You Start This Lab

Run `git pull` in `fpga_labs_fa19`.

Review these documents that will help you better understand some concepts we will be covering.

1. verilog_fsm.pdf - constructing FSMs in Verilog.
2. Debouncer Circuit

   Read the "What is Switch Bounce" section to get idea of why we need a debouncer circuit. Read the "Digital Switch Debouncing" section to get a general overview of the circuit, its parts, and their functions.

2 Lab Overview

In this lab, we will

- use a ROM to store a melody and play it on the FPGA using the `tone_generator`
- build input conditioning circuits that make control signals from physical input devices, like the buttons and switches we've been using up to now, more reliable
- verify the conditioning circuits are working correctly using the FPGA LEDs
- use synchronous resets to reset our circuits to a known initial state
- create a basic FSM in the `music_streamer` that uses the buttons to change states and alter the music playback
- (optionally) extend our `music_streamer` into a sequencer.

3 Using an Asynchronous ROM to Build the Music Streamer

An asynchronous memory is a memory block that isn't governed by a clock. In this lab, we will use a Python script to generate a ROM block in Verilog.

A ROM is a read-only memory. This data can be accessed by supplying an address to the ROM; after some time, the ROM will output the data stored at that address. A memory block in general can contain as many addresses in which to store data as you desire. Every address should contain the same amount of data (bits). The number of addresses is called the **depth** of the memory, while the number of bits stored per address is called the **width** of the memory.

The synthesizer takes the Verilog you write and converts it into a low-level netlist of the structures are actually used on the FPGA. Our Verilog **describes** the functionality of some digital circuit and the synthesizer **infers** what primitives implement the functional description. In this section, we will examine the Verilog that allows the synthesizer to infer a ROM. This is a minimal example of a ROM in Verilog: (depth of 8 entries/addresses, width of 12 bits)

```
module rom (input [2:0] address, output reg [11:0] data);
  always @(*) begin
    case(address)
      3'd0: data = 12'h000;
      3'd1: data = 12'hFFF;
      3'd2: data = 12'hACD;
      3'd3: data = 12'h122;
      3'd4: data = 12'h347;
      3'd5: data = 12'h93A;
      3'd6: data = 12'h0AF;
      3'd7: data = 12'hC2B;
    endcase
  end
endmodule
```

To power our `tone_generator`, we will be using a ROM that is X entries/addresses deep and 24 bits wide. The ROM will contain tones that the `tone_generator` will play. You can choose the depth of your ROM based on the length of the sequence of tones you want to play.

We've provided you with a few scripts that can generate a ROM from either a file with it's contents or even from sheet music. Run these commands from `lab3/`.

```
python scripts/musicxml_parser.py musicxml/Twinkle_Twinkle_Little_Star.mxl music.txt
python scripts/rom_generator.py music.txt ./lab3.srcs/sources_1/new/rom.v 1024 24
```

The first script will parse a MusicXML file and turn it into a list of `tone_switch_periods` for each of the notes for a piece of sheet music. The second script will take that list and turn it into a ROM that's 1024 entries deep with a width of 24 bits.

Take a look at `music.txt` and `src/rom.v`. You can download your own music in MusicXML format from here (`https://musescore.org/`) and run it through the same parser; it should ideally only have one part to work properly. You can also directly edit the `music.txt` file to customize the contents of the ROM as you wish.

4 Design of the music_streamer

Open up the `music_streamer.v` file. You will need to modify this module to contain an instance of the ROM you created earlier and logic to address the ROM sequentially to play notes. The `music_streamer` will play each note in the ROM for a predefined amount of time by sending it to the `tone_generator`.

We will play each note for 1/25th of a second. Calculate what that is in terms of 125Mhz clock cycles.

Now let's begin the design of the `music_streamer` itself. Instantiate your ROM in the `music_streamer` and connect the ROM's `address` and `data` ports to wire or reg nets that you create in your module. The `last_address` port outputs the last address in the ROM (depth).

Next, write the RTL that will increment the address supplied to the ROM every **1/25th of a second**. The data coming out of the ROM should be fed to the `tone` output. The ROM's address input should go from 0 to the depth of the ROM and should then loop around back to 0. You don't have a reset signal, so define the initial state of any registers in your design for simulation purposes. Also hook up the `rom_address` output to the ROM address currently being accessed.

Now that you have implemented `music_streamer`, create an instance of it in the module `z1top.v`. Use the instance name `streamer` to match the expected name in the `.do` file. Instantiate a `tone_generator` and wire `SWITCHES[1]` to `output_enable`, `CLK_125MHZ_FPGA` to `clk`, and `aud_pwm` to `square_wave_out`. Assign `aud_sd` to 1. Connect the `tone` output of the `music_streamer` to the `tone_switch_period` input of the `tone_generator`. Connect the `music_streamer`'s `clk` input to the global clock signal. Finally, connect its `rom_address` output to the `LEDS`s by routing the top 6 bits of address.

5 Testbench Techniques

There are several testbenches included in this lab for your synchronizer, edge detector, shift register, debouncer, and music streamer that introduce you to some useful Verilog testbench constructs.

- `@(posedge <signal>)` and `@(negedge <signal>)` - These are a different type of delay statement from what you have seen before. `#10` would advance the simulation by 10 timesteps. These commands will advance the simulation until the `<signal>` rises or falls.

  For example:

  ```
  @(posedge signal);
  @(posedge signal);
  ```

  Simulation time will advance until we have seen two rising edges of `signal`.

- `repeat` - it acts like a `for` loop but without an increment variable

  For example:

  ```
  repeat (2) @(negedge clk);
  ```

```
repeat (10) begin
    @(posedge clk);
end
```

The simulation will advance until we have seen 2 falling clock edges and will then advance further until we have seen 10 rising clock edges.

- `$display` - acts as a print statement. Similar to languages like C, if you want to print out a wire, reg, integer, etc... value in your testbench, you will need to format the string. It works like `printf()` in C.

  For example:

```
$display("Wire x in decimal is %d", x);
$display("Wire x in binary is %b", x);
```

- `tasks` - tasks are subroutines where you can group and organize some commands rather than haphazardly putting them everywhere. They can take inputs and assign outputs.

```
task wait_for_n_clocks();
input [7:0] num_edges;
begin
    repeat (num_edges) @(posedge clk);
end
endtask
```

- `fork/join` - Allows you to execute testbench code in parallel. You create a fork block with the keyword `fork` and end the block with the keyword `join`.

  For example:

```
fork
    begin
        task1();
    end
    begin
        $display("Another thread");
        task2();
    end
join
```

  Multiple threads of execution are created by putting multiple begin/end blocks in the fork-join block. In this example, thread 1 runs `task1()`, while thread 2 first `$display`s some text then runs `task2()`. The threads operate in parallel.

- Hierarchical Paths - you can access signals inside an instantiated module for debugging purposes. This can be helpful in some cases where you want to look at an internal signal but don't want to create another output port just for debug.

For example:

```
tone_generator tone_gen ();
$display("Signal inside my tone_generator instance, clock_counter: %b",
↪  tone_gen.clock_counter);
```

6 Simulating the music_streamer

To simulate your `music_streamer` open up the `lab3/src/music_streamer_testbench.v`. In contrast to the `tone_generator_testbench` where the `tone_generator` was instantiated in isolation, in this testbench we are instantiating our entire top-level design, `z1top`. This testbench is referred to as a system-level testbench, which tests our entire design using top-level I/O, in contrast to the `tone_generator_testbench` which is a block-level testbench. This is similar to the difference between unit and integration tests in software development.

You can see that this testbench just runs a simulation for 2 seconds and then exits. You might have to modify the `music_streamer_testbench.do` file to match the name of your module instances in `z1top.v`.

To execute the testbench, run `make CASES=tests/music_streamer_testbench.do` in `lab3/sim`. This may take several minutes to complete. You may have to run `make clean` before running `make` if ModelSim has cached build artifacts.

Inspect your waveform to make sure you get what you expect. Verify that there are no undefined signals (red lines, x) Then run the Python script to generate a `.wav` file of your simulation results and listen to your `music_streamer`. It should sound like the first few seconds of the song that was loaded on the ROM.

7 Verify your Code Works For Rest Notes

In simulation, you can often catch bugs that would be difficult or impossible to catch by running your circuit on the FPGA. You should verify that if your ROM contains an entry that is zero (i.e. generate a 0Hz wave), that the `tone_generator` holds the `square_wave_out` output at either 1 or 0 with no oscillation. Verify this in simulation, and prove the correct functionality during checkoff.

8 Try it on the FPGA!

Now try your `music_streamer` on the FPGA. You should expect the output to be the same as in simulation. The `SWITCHES[1]` switch should still work to disable the output of the `tone_generator`. Show your final results, simulation, and the working design on the FPGA to the TA for checkoff.

9 Synchronizer, Debouncer

9.1 Synchronizer

In Verilog (RTL), digital signals are either 0's or 1's. In a digital circuit, a 0 or 1 corresponds to a low or high voltage. If the circuit is well designed and timed (fully synchronous), we only have to worry about the low and high voltage states, but in this lab we will be dealing with asynchronous signals.

The signals coming from the push buttons and slide switches on the PYNQ-Z1 board don't have an associated clock signal. For the push-buttons in particular, when those signals are put through a register, the hold or setup time constraints of that register may be violated. This may put that register into a **metastable** state.

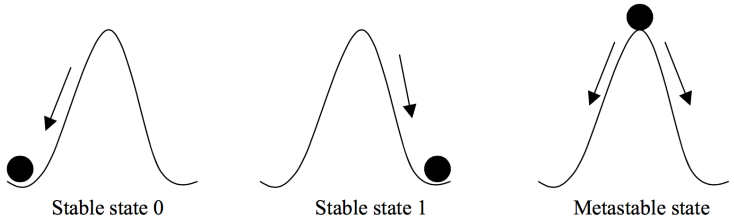


Figure 1: The 'ball on a hill' metaphor for metastability. If a register's timing constraints are violated, its output voltage oscillates and after some time unpredictably settles to a stable state.

In a fully synchronous circuit, the timing tools will determine the fastest clock frequency under which the setup time constraints are all respected and the routing tools will ensure that any hold time constraints are handled. Introducing an asynchronous signal that isn't changing with respect to a clock signal can cause a register to go into a metastable state. This is undesirable since this will cause a 'mid-rail' voltage to propagate to other logic elements and can cause catastrophic timing violations that the tools never saw coming.

We will implement a synchronizer circuit that will safely bring an asynchronous signal into a synchronous circuit. The synchronizer needs to have a very small probability of allowing metastability to propagate into our synchronous circuit.

This synchronizer circuit we want you to implement for this lab is relatively simple. For synchronizing one bit, it is a pair of flip-flops connected serially. This circuit synchronizes an asynchronous signal (not related to any clock) coming into the FPGA. We will be using our synchronizer circuit to bring any asynchronous off-FPGA signals into the clock domain of our FPGA design.

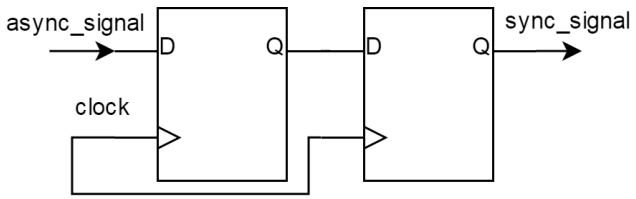


Figure 2: 1-bit 2 Flip-Flop Synchronizer

Edit the `synchronizer.v` (in your project sources) file to implement the two flip-flop synchronizer. This module is parameterized by a `width` parameter which indicates the number of one-bit signals to synchronize.

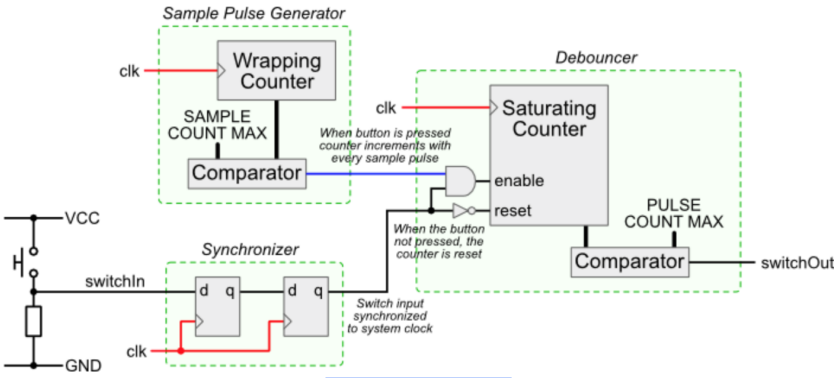
9.1.1 Testing in Simulation

The testbenches to be run are stored in `lab4/sim/tests`. Each `.do` file in this directory is run when you run `make` in the `lab4/sim` directory. If you only want to run one testbench, you can rename all the other `.do` files in this directory to have a different file extension. Alternatively, you can leave the file extensions alone, and specify the test you want to run like this:

```
cd lab4/sim
make CASES="tests/sync_testbench.do"
```

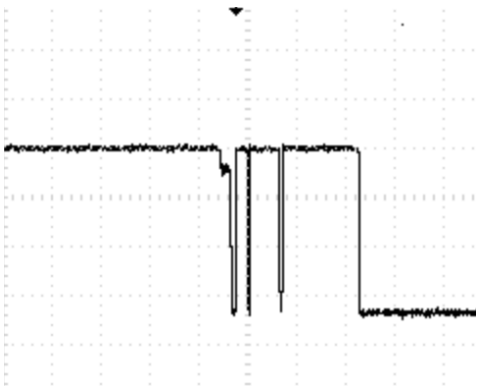
We have provided a testbench for your synchronizer called `sync_testbench` in `sync_testbench.v` (in the project source files). Take a look at the code for this testbench and run it; **the testbench shouldn't print any failure messages and you should inspect the waveform before you move on.** For details on the constructs/techniques/syntax used in this testbench, refer to Section 5: Testbench Techniques in this lab.

9.2 Debouncer and Edge Detector



Recall this graphic from the prelab debouncer reading. It is an overview of the debouncer circuit which includes the synchronizer circuit.

For this lab, the debouncer circuit will take a button's glitchy digital input and output a clean signal indicating a single button press. The reason we need a circuit for this can be seen in the figure below.



When we press the button, the signal doesn't behave like a perfect step function. Instead the button signal is glitchy due to mechanical "bounce". A debouncer turns this waveform, which shows a single button press, into a clean signal with a single voltage transition.

Take a look at `debouncer.v` in your project sources. This is a parameterized debouncer which can debounce `width` signals at a time. Your debouncer receives a vector of synchronized 1-bit signals and it outputs a debounced version of those signals. The other parameters reference the constants used in the circuit from the prelab reading.

The debouncer consists of:

1. **Sample Pulse Generator** - Tells our saturating counter when to sample the input signal. It should output a 1, every `sample_count_max` clock cycles. By default `sample_count_max` is set to 25000.

2. **Saturating Counter** - This is a counter that counts up to `pulse_count_max`. The saturating counter should increment by one if both the sample pulse and the input signal are high at a clock edge. At any clock edge, if the input signal is 0, the saturating counter should be reset to 0. Once the saturating counter reaches `pulse_count_max`, it should hold that value indefinitely until the input signal falls to 0, upon which the saturating counter should be reset to 0. The `debounced_signal` of your debouncer should be an equality check between the saturating counter and `pulse_count_max`.

You should use the same sample pulse generator for all input signals into your `debouncer`, but you should have a separate saturating counter per input signal. You will likely need to use a 2D reg in Verilog to create the saturating counters. You will also likely need to use generate-for.

Here is an **example** of creating a 2D array:

```
reg [7:0] arr [3:0]; // 4 X 8 bit array
arr[0]; // First byte from arr (8 bits)
arr[1][2]; // Third bit of 2nd byte from arr (1 bit)
```

And here is an **example** of using a generate-for loop:

```
genvar i;
generate
    for (i = 0; i < width; i = i + 1) begin:LOOP_NAME
```

```
        always @ (posedge clk) begin
            // Insert synchronous Verilog here
        end
    end
endgenerate
```

Implement the debouncer (in `debouncer.v`).

9.2.1 Edge Detector

The debouncer will act to *smooth-out* the button press signal. It is then followed up with an edge detector that can take the low-to-high transition of the debouncer output and use it to generate a 1 clock period wide pulse that the rest of our digital design can use.

Create a variable-width edge detector in `edge_detector.v`.

9.2.2 Testing in Simulation

We've provided a testbench to test your debouncer and edge detector circuits in `debouncer_testbench.v` and `edge_detector_testbench.v`. Run the testbench, make sure it passes, and inspect the waveforms before FPGA testing. Make sure there are no undefined (red line) signals.

If you are seeing issues where certain registers are red lines (X's), make sure you give them an initial state. For a 2D reg initialization, use the following initialization code in `debouncer.v`:

```
integer k;
initial begin
    for (k = 0; k < width; k = k + 1) begin
        saturating_counter[k] = 0;
    end
end
```

The debouncer testbench has 2 tests:

1. Verifies that if a glitchy signal initially bounces and then stays high for **less** than the saturation time, that the debouncer output never goes high.
2. Verifies that if a glitchy signal initially bounces and then stays high for **more** than the saturation time, that the debouncer goes high and stays high until the glitchy signal goes low.

The edge detector testbench tests 2 scenarios, when the `signal_in` is a pulse 10 clock cycles wide and a pulse 1 clock cycle wide and verifies that the `edge_detect_pulse` output goes high twice, both times with a width of 1 clock cycle.

9.2.3 Testing on the FPGA

We have created a top level module called `debouncer_fpga_test` that will create a 6-bit register and will use button presses to add and subtract from it. This module will use both your `debouncer.v` and `edge_detector.v`.

Pressing BTN1 or BTN2 will cause the register to incrment by 1. Pressing BTN0 will cause the register to decrement by 1. BTN3 resets the register to 0. The LEDs will show the current value of the register.

In your Vivado project, set `debouncer_fpga_test` as the top module and run the build flow, then program the board. Make sure that you **investigate any warnings for synthesis**. You can ignore some: the top-level module has unused input/output ports to avoid having to change the XDC file. Make sure you fix any other warnings you find so that the your debouncer will work as expected on the FPGA.

You will discover when playing with your debouncer that the buttons have a way that they like being pressed to minimize bounce; get a good feel for them.

10 Synchronous Resets In Design and Simulation

Begin by copying your `tone_generator` and `music_streamer` from lab 3. **Do not change the port declaration of the lab 4 skeleton files, only copy over your implementation**.

Now that we have a debouncer that can give us a pulse for a press of a button, we have a way of explicitly resetting our circuits! You will recall that in the previous lab, we set the initial value of registers as below so that our simulation would have defined signals.

```
reg [23:0] clock_counter = 0;
```

or

```
initial clock_counter = 0;
```

Tying one of the push buttons to a reset signal, we can now do this instead.

```
always @ (posedge clk) begin
    if (rst) begin
        clock_counter <= 24'd0;
    end
    else begin
        clock_counter <= clock_counter + 24'd1;
    end
end
```

Unlike what we did before, this Verilog is synthesizable for all deployment targets, FPGAs, ASICs, and CPLDs alike. Go ahead and modify your `tone_generator` and `music_streamer` to use the provided `RESET` signal to get your registers to a sensible initial state.

After doing this, run the `tone_generator_testbench` again using `make` in the `lab4/sim/` directory (or just use Vivado's simulator). View the waveform using the simulator and see how we used a reset in the testbench to bring all the registers to a defined state without specifying a default value.

11 Music Streamer Tempo Control

Let's use the new user inputs we now have access to. You will recall that your `music_streamer` by default chooses to play each tone in the ROM for 1/25th of a second. Extend the functionality of the `music_streamer` so that one input increases the tempo, one input decreases it, and one resets it to a default value. If you have the hex keypad working, you can map any of the keys to these functions, but make a note of it. If not, use the push-buttons left on the Pynq-Z1 (remembering that BTN3 is now reserved as a global reset signal).

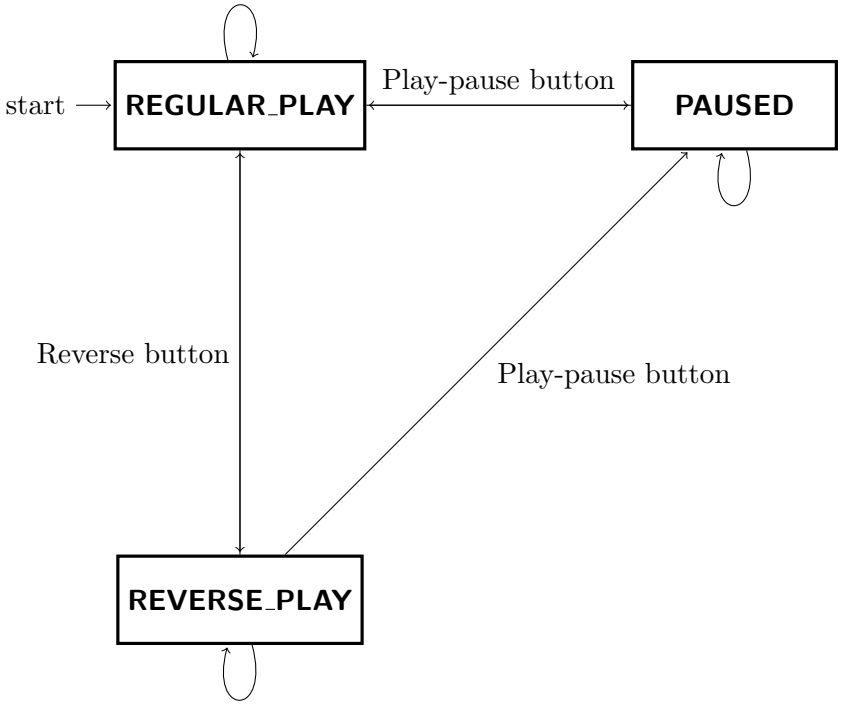
You can implement this by using a register to hold the number of clock cycles per note. Instead of this number being hardcoded in Verilog to represent $\frac{1}{25}$th of a second, you can change it at runtime. One input should add or subtract a fixed number from this register, which should alter the time each tone is played. You get to choose this number; find something reasonable.

Try this out on the FPGA and verify that you have control of your `music_streamer`'s tempo using the inputs. You should be able to speed up and slow down the music you are playing.

12 Music Streamer FSM

Now, you will implement a simple FSM in the `music_streamer`.

The FSM will have 3 states: `PAUSED`, `REGULAR_PLAY`, `REVERSE_PLAY`. Here is the state transition diagram:



1. Your initial state should be `REGULAR_PLAY`.
2. Pressing one button should transition you into the `PAUSED` state from either the `REGULAR_PLAY` or `REVERSE_PLAY` states. Pressing the same button while in the `PAUSED` state should transition the FSM to the `REGULAR_PLAY` state.
3. In the `PAUSED` state, your ROM address should be held steady at its value before the transition into `PAUSED` and no sound should come out of the speaker. After leaving the `PAUSED` state your ROM address should begin incrementing again from where it left off and the speaker should play the tones.
4. You can toggle between the `REGULAR_PLAY` and `REVERSE_PLAY` states by using a second input button. In the `REVERSE_PLAY` state you should decrement your ROM address by 1 rather than incrementing it by 1 every X clock cycles as defined by your tempo.
5. If you don't press any buttons, the FSM shouldn't transition to another state.

Your `music_streamer` takes in user button inputs that it can use to transition states. If you have a working keypad you will have plenty of inputs to choose from, and you should select new inputs such that the tempo control from the previous section still works (again, remember to note your scheme). If not, you have to be creative: use an edge detector circuit with the slide switches to convert their actuation into a momentary signal (using the switch position as state would be too easy).

Also, drive the LEDs such that they show the current state. An example:

LED	**Value**
LEDS[5]	`current_state` == `REGULAR_PLAY`
LEDS[4]	`current_state` == `PAUSED`
LEDS[3]	`current_state` == `REVERSE_PLAY`
LEDS[2:0]	0

You can run the testbench in `music_streamer_testbench.v` to test out your state machine. Take a look at the code to see what it does and inspect your waveform to check that your FSM is performing correctly. Verify that you don't have any unexpected synthesis warnings.

Finally, program the FPGA with your `music_streamer`, and verify its functionality.

13 Checkoff

1. Prove that if the ROM contains an entry for a `tone_switch_period` of 0, that the square wave doesn't oscillate.
2. Show the working `music_streamer` on the FPGA.

Ackowlegement

This lab is the result of the work of many EECS151/251 GSIs over the years including:

- Sp12: James Parker, Daiwei Li, Shaoyi Cheng
- Sp13: Shaoyi Cheng, Vincent Lee
- Fa14: Simon Scott, Ian Juch
- Fa15: James Martin
- Fa16: Vighnesh Iyer
- Fa17: George Alexandrov, Vighnesh Iyer, Nathan Narevsky
- Sp18: Arya Reais-Parsi, Taehwan Kim
- Fa18: Ali Moin, George Alexandrov, Andy Zhou