EECS 151/251A FPGA Spring 2021
Lab 2:
Sequential Circuits, Simulation, Button Parser

Prof. John Wawrzynek
TAs: Sean Huang, Tan Nguyen
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_sp21`.

Before you begin this lab, please review our lecture and discussion slides on Verilog. Another useful resource is Verilog Primer Slides. You should understand some fundamental concepts of hardware design in Verilog, such as combinational logic and sequential logic, in order to complete this lab. We also suggest that you take some time to look through these documents that will help you better understand some Verilog constructs.

1. wire_vs_reg.pdf - The differences between wire and reg nets and when to use each of them.
2. always_at_blocks.pdf - Understanding the differences between the two types of always @ blocks and what they synthesize to.

Also, please note that we adopt the "No register inference" policy in this semester (refer to slide 18, Lec 4). Therefore, you should not write any sequential always blocks in your code. You should not need to use non-blocking assignments or *reg* nets for registers. Instead, **please instantiate a REGISTER* module from the lib/EECS151.v file from our lab Github repository whenever you want to declare a register**. This only applies to Verilog design file, but not necessarily to testbench code as we will see later.

The previous lab walked you through how to create and manage an FPGA project in Vivado Design suite. In this lab, we cover Simulation using Vivado Design suite. The first few labs are meant to be long and tedious. Don't be intimidated! As you will see, being a good hardware designer does not mean just writing Verilog or any HDL code; you also need to have a firm understanding on the design tools to productively generate a working hardware implementation for your FPGA.

We also learn how to design a Debouncer Circuit which can reliably sample button presses to provide control signals for our sequential digital circuits. Read the "What is Switch Bounce" section about why we need a debouncer circuit. Read the "Digital Switch Debouncing" section about the implementation of a debouncer circuit.

2 Structural coding vs. Behavioral coding

In this section, you will design a 3-bit adder using either structural or behavioral approach in Verilog.

2.1 Build a Structural 3-bit Adder

What do we mean by "structural approach"? It builds a circuit using only Boolean logic gates and multiplexers. It's similar to how you are drawing your design in a schematic editor, but we use Verilog instead. To help you with this task, please refer to the 'Code Generation with for-generate loops' slide in the Verilog Primer Slides (slide 35).

1. Create a new project (or reuse existing project). Add the following Verilog files to your project: `lab2/src/full_adder.v`, `lab2/src/structural_adder.v`, `lab2/src/z1top_adder.v`. Also add the constraint file to your project: `lab2/consrr/z1top.xdc`. For your own convenience, the pin assigments for the buttons, switches, and LEDs have been done. Notice their relevance to the top-level Verilog module `z1top_adder`.

2. Open `full_adder.v`; fill in the logic to produce the full one-bit adder outputs from the inputs.

3. Open `structural_adder.v` and construct a ripple carry 3-bit adder using the full one-bit adder cells you designed earlier. Use 'for-generate loop' construct to make your code compact.

4. Finally, inspect the `z1top_adder.v` top-level module and see how your structural adder is instantiated and hooked up to the top-level signals. For now, just look at the `user_adder` instance of your structural adder. Note how the adder operands and result are constructed from the input/output signals of the top-level Verilog module, and in turn the buttons and switches and LEDs on the FPGA.

5. Set `z1top_adder.v` as the top-level module by right-clicking `z1top_adder` in Sources → Design Sources and click 'Set as Top'. You can skip this step if it is already the top-level module.

6. Run bitstream generation as usual.

7. Program the FPGA and test out the design. Try different combinations of buttons and switches. Make sure you get correct results from your adder via the LEDs.

2.2 Inspection of Structural Adder

2.2.1 Schematics and FPGA Layout

Take a look at how the Verilog you wrote mapped to the primitive components on the FPGA. Three levels of schematic are generated for you once you've run the pipeline. In the *Flow Navigator*, you can view *Schematic*s under

1. *RTL Analysis → Open Elaborated Design*

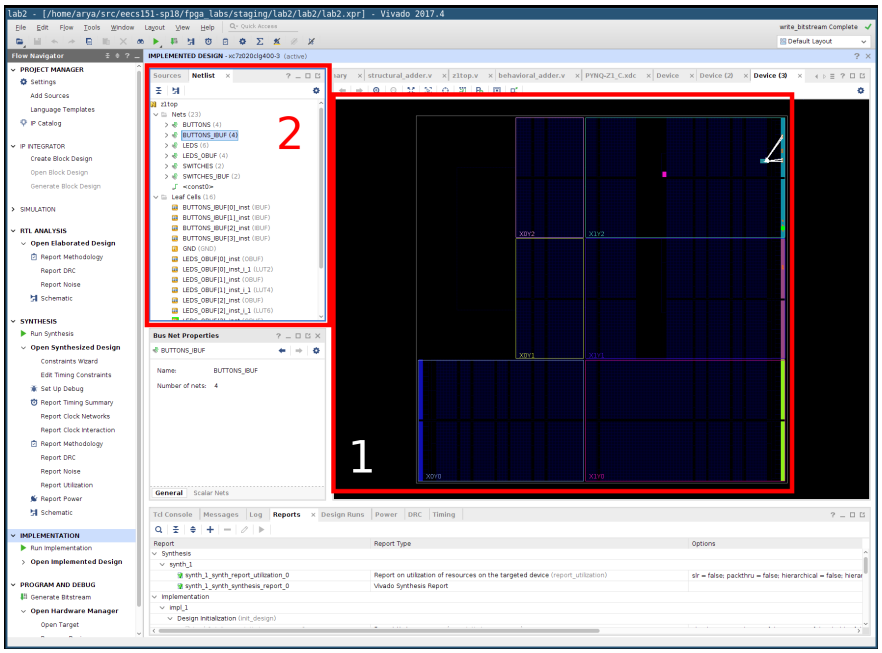
2. *Synthesis → Open Synthesized Design*

3. *Implementation → Open Implemented Design*

The first two will give you a fairly straightforward hierarchical block-level view of your design. You will find your circuit by drilling down into the `user_adder` module (that's the name you gave the instantiation of `structural_adder` in `z1top_adder.v`). Check to see that your structural adder module is hooked up properly and looks sane. It's ok if the wires don't appear to be connected, just hover your mouse over the endpoints on the schematic and ensure that the connections are as you expect. Take note of the primitive blocks used in your circuit.

In the RTL Analysis (1) you are viewing a visualization of the topology your RTL describes. At this point, logic elaboration is very abstract: you'll notice that your logic is expressed in terms of the logic gates you described (XOR, AND, etc). Any logic you describe in RTL is included, even if it's disconnected. In the Synthesis schematic (2) this logic has been elaborated further into what look like FPGA elements, but still at higher layer of abstraction, and with some unused signals still present. In the final schematic of the three, Implementation (3), the schematic now shows which of the elements in your nominated chip are actually targeted. Superfluous logic has been elided from the design.

Finally, you also look at how your circuit was placed and laid out on the FPGA. Once you've run the pipeline, open *Implemented Design*, click on the *Window* menu, and select *Device*. You'll be presented with a layout of the FPGA package as in box 1 in the figure below. It'll be hard to see with a small design, but the logic elements you've ended up using with your design will be highlighted. You can highlight your own nets in the diagram to make it easier to find them by selecting a net or signal from the Netlist pane (*Window* → *Netlist*; see box 2).



Explore your design and look for the modules that you wrote. If you scroll down in the `Netlist Window` you should see various components of your logic. Some elements are mapped to LUTs: somewhere buried in their properties is the type of slice. See if you can find out which nets have been assigned to LUTs, and how they are connected. Go ahead and explore several SLICELs that implement the structural adder to see how they are connected to each other and the outputs of your circuit.

2.3 Build a Behavioral 3-bit Adder

Check out `behavioral_adder.v`. It has already been filled with the appropriate logic for you. Notice how behavioral Verilog allows you to describe the function of a circuit rather than the topology or implementation. Modify your top-level module in `z1top_adder.v` to use the behavioral adder instead of the structural adder. Generate and load the bitstream to the FPGA. Verify if the behavioral adder works correctly.

Behavioral coding style allows designers to build their designs using arithmetic operators, com-

parators, or software-like constructs such as `if-else`, `case` statements (to generate muxes), which greatly improves productivity.

2.4 Inspection of Behavioral Adder Schematics and FPGA Layout

Go through the same steps as you did for inspecting the structural adder. View the schematics at successive levels of logic elaboration and how FPGA components are connected.

2.5 Questions

1. Record and note down any differences you see between both types of adders in the schematics and the FPGA layouts (from *RTL Elaborated Design* to *Implemented Design*).

3 Build Your First Sequential Digital Circuit

In this section, you will design a 4-bit wrap-around counter that increments every one second. The counter value is shown on the LEDS 0-3 of the PYNQ board.

3.1 Clock Sources

Look at the PYNQ Reference Manual. Read Section 11 about the clock sources available on the PYNQ. We are using the 125 MHz clock from the Ethernet PHY IC on the PYNQ board that connects to pin H16 of the FPGA chip.

Look at the `lab2/src/z1top_counter.v` top-level module and its `CLK_125MHZ_FPGA` input.

```
module z1top_counter (
    input CLK_125MHZ_FPGA,
    ...
);
```

We can access the clock signal from our Verilog top-level module and can propagate this clock signal to any submodules that may need it.

Next, open the constraint file `lab2/constr/z1top.xdc`, **add the following TCL commands to it**

```
set_property -dict { PACKAGE_PIN H16 IOSTANDARD LVCMOS33 } [get_ports { CLK_125MHZ_FPGA }];
create_clock -add -name CLK_125MHZ_FPGA -period 8.00 -waveform {0 4} [get_ports { CLK_125MHZ_FPGA }];
```

The first command assigns our clock input signal to H16, just as specified in the PYNQ-Z1 Reference Manual. The second command informs Vivado that our input clock signal has a period of 8 ns (125 MHz) which matches the actual clock frequency from pin H16. This is the target clock constraint (or often referred as frequency constraint) of your design. Vivado will try to optimize your design to meet this constraint, and the post-routing timing analysis will report whether it is the case. Please do not forget to set a clock constraint when you design a sequential circuit.

3.2 Build a 4-bit Counter

Your circuit receives an input clock signal with a clock period of 8 ns (125 MHz). How many cycles of this clock signal are equivalent to one second? Note that

$$Time(sec) = Clock\ Period \times Number\ of\ cycles$$

You will need to wait for this amount of clock cycles before incrementing your LED counter. You may use an additional counter to keep track of this.

You are required to use the library of registers that we provide you (`lib/EECS151.v`). Please instantiate a REGISTER* module whenever you want to build a register. Please do *not* write any sequential always block (e.g., `always @(posedge clock)`) or non-blocking assignment in your code.

1. Create a new project (or reuse existing project). Add the following Verilog file to your project: `lab2/src/z1top_counter.v`.
2. Add the EECS151 library of registers to your project `lib/EECS151.v`.
3. Open `z1top_counter.v`; fill in the necessary logic to implement the counter. Some initial code has been provided to help you get started.
4. Set `z1top_counter.v` as the top-level module by right-clicking `z1top_counter` in Sources → Design Sources and click 'Set as Top'. You can skip this step if it is already the top-level module.
5. Generate the bitstream, program the FPGA and test it.
6. Check the timing report (where is it? Check lab 1 again). Does it indicate that your circuit meets the target timing constraint?

4 Simulation

At this point, you have designed basic digital circuits and verified them on the PYNQ-Z1 board using the LEDs, buttons, and switches. However, that testing approach only works for small designs that can fit to the IO capability provided by our target platform. Imagine you want to create a 100-bit adder or counter; the board clearly does not have enough IO for you to test your circuit. A more empirical method of verifying whether your circuit meets the functional specification is RTL simulation (or functional/behavioral simulation). To do that, we need a software simulator. There are many simulators out there including free, open-source ones as well as commercial products. In the past labs, we used ModelSim (from Mentor Graphics) or VCS (from Synopsys). They are available in our instructional machines. They generally require licenses if you want to use them outside of the lab machines. In this lab, we use Vivado simulator (xsim) as it is a part of the Vivado Design suite. If you have a Vivado WebPACK installed in your laptop, you should be able to use xsim too. You might also be interested to learn about other open-source simulators such as Verilator or Icarus Verilog (IVerilog). xsim is recommended for the FPGA labs, but feel free to take advantage of any simulation software you like or are familiar with.

Besides a simulator software, you will need to write your own testbench in Verilog. The testbench instantiates the module you want to test (say, your 100-bit adder, or we often refer to it as "DUT" – design-under-test), and provides stimuli (e.g., test cases) to your DUT. Your job is to check if the output of your DUT matches your expectation. You can either print out the output values or dump a waveform file that can be opened by a waveform viewer software. xsim has its own waveform viewer, so you don't have to worry about installing another tool. If you are looking for free tools, GTKWave is decent and able to support the output waveform generated by Verilator/IVerilog/VCS.

Simulation plays a huge role in digital design. You probably don't want to waste time on the synthesis and implementation tools only to generate a buggy hardware implementation. Simulate your design as you go, and only when you are sure that it meets your expectation, you can hand it

to the synthesis tool.

Let's get some practice from the lab exercises.

5 Simulating Your 32-bit Structural Adder

Reuse the structural adder code you did previously. You need to parameterize your adder so that it can support arbitrary bitwidth. Update the files `lab2/src/full_adder.v`, `lab2/src/structural_adder.v` with your own code. A testbench code has been provided to you.

Create a new Vivado project (or reuse existing project). Add the following files to your project: `lab2/src/structural_adder.v`, `lab2/src/full_adder.v`. Add the testbench file to your project (make sure to select *Add or create simulation sources* when you add it): `lab2/sim/structural_adder_tb.v`. Let's take a look at the testbench code.

```
`timescale 1ns/1ns
```

The timescale declaration needs to be at the top of every testbench file.

The first argument to the timescale declaration is the simulation step time. It defines the granularity of discrete time units in which the simulation advances. In this case, we have defined the simulation step time to be one nanosecond, so we can advance the simulation time by as little as 1ns at a time.

The second argument to the timescale declaration is the simulation resolution. In our example it is also 1ns. The resolution allows the simulator to model transient behavior of your circuit in between simulation time steps. For this lab, we aren't modeling any gate delays, so the resolution can safely equal the step time.

```
module structural_adder_tb();
```

Next, we have a Verilog module `structural_adder_tb` that does not have any input or output signal, as it is not meant for being synthesized to a bitstream! Simulation is a totally separate step from what we have learned in the previous lab which we have a tool flow from *Synthesis* → *Implementation* → *Generate Bitstream*; the testbench code does not necessarily go through the flow.

```
reg clock;
initial clock = 0;
always #(4) clock <= ~clock;
```

This is the clock generation code. The clock signal needs to be generated in our testbench so it can be fed to the DUT. The initial statement sets the value of the clock net to 0 at the very start of the simulation. The next line toggles the clock signal every 4ns, i.e. half period of 125 MHz clock.

```
reg [N-1:0] operand1, operand2;
wire [N:0]  adder_output;
```

Here are the inputs and output of our `structural_adder`. Notice that the inputs to the `structural_adder` are declared as `reg` type nets and the outputs are declared as `wire` type net. This is because we will be driving the inputs in our testbench inside an `initial` block and we will be reading the output.

A parameterized 32-bit structural adder is instantiated in the testbench module. This is the "design-under-test" module. We assign the input and output signals to our DUT.

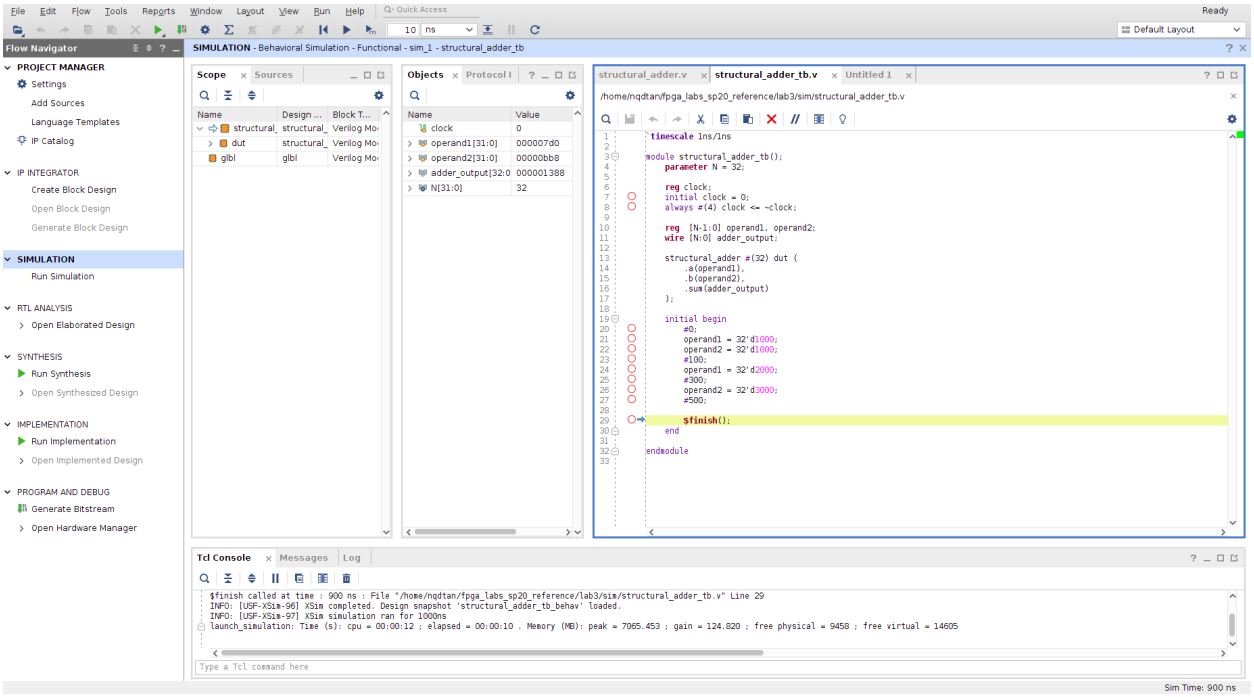
```
// Note: this assumes that you have the parameter N in your structural_adder code
// (the bitwidth of your operands)
module structural_adder #(.N(32)) dut(
  .a(operand1),
  .b(operand2),
  .sum(adder_output)
);
```

You could instantiate multiple modules for testing in a single testbench, but it is generally recommended to have a separate, small testbench for each DUT to not overload yourself with many debug information to keep track of, and also to catch easier bugs. Once you are done with making sure individual modules work properly, you can go up to the next design hierarchy to test their integration. We will see that in later lab exercises and project.

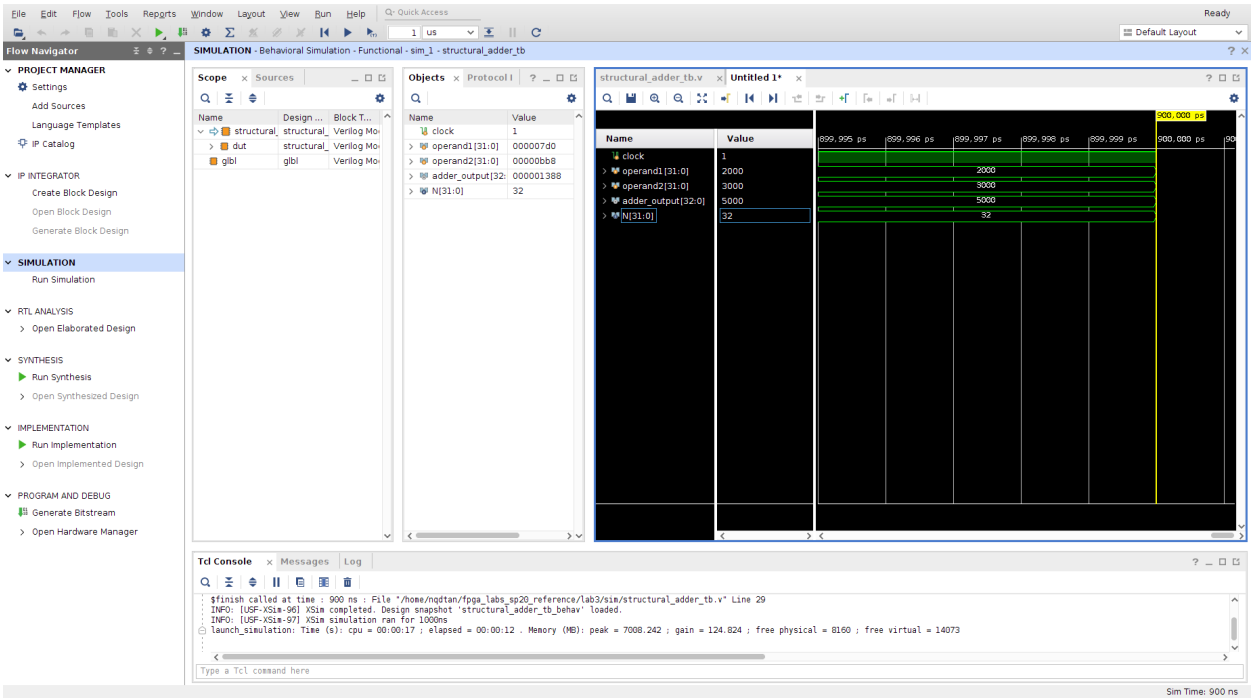
```
initial begin
    #0;                  // at the start of simulation
    operand1 = 32'd1000; // (1)
    operand2 = 32'd1000; // (2)
    #100;                // forward time by 100ns
    operand1 = 32'd2000; // (3)
    #300;                // forward time by 300ns
    operand2 = 32'd3000; // (4)
    #500;                // forward time by 500ns

    $finish();
end
```

In this initial block, you assign some values to the inputs of your DUT. Think of those assignments as events as they happen at specific timestamps. For example, (1) and (2) happen at the start of the simulation, (3) happens at 100ns from the start, and (4) happens at 400ns from the start. The statement `#X;` tells the simulator to advance the simulation by `X` $\times$ simulation step time. The keyword `$finish()` tells the simulator to exit the simulation. In our example, the simulation finishes after 900ns.

Let's run our first simulation to understand what it means. In the *Flow Navigator*, click *Simulation* $\rightarrow$ *Run Simulation*. Make sure to choose *Run Behavioral Simulation*. Vivado will compile your Verilog code and launch the simulator. By default, Vivado simulation automatically runs until it hits `$finish()`. You will see that it stops at the statement `$finish();` in the testbench.



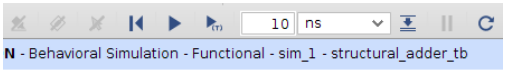
Go to the waveform tab (*Untitled 1*) to observe the waveform. Check the timings of those assignments as well as the values of `adder_output` signal to see if they make sense to you. The simulation should stop at 900ns. By default, the multi-bit signals such as `operand1`, `operand2`, `adder_output` are displayed in hex format. You can convert them to decimal format by selecting those signals, then right click → *Radix* → *Unsigned Decimal*. You should expect to see something as follows.



Spend some time to explore the functions of the buttons in the waveform window. Things like *Next Transition* or *Previous Transition* along with *Markers* are helpful when you want to track

the transitions of your signals. You can do `Ctrl +` your mouse wheel to zoom in/out part of the waveform. Click *Zoom Fit* to see the entire waveform from the start to the end.

Next, we explore the functions of the simulation buttons (right under Vivado top menu bar). Click *Restart* button (or `Ctrl + Shift + F5`) to restart your simulation. You will see that all the signals are in 'X' state (uninitialized). Click *Run All* button (or `F3`) to run the simulation. This will run the simulation until finish. The tool also provides an option to run the simulation for some amount of time that you want (click *Run for ...* or `Shift + F2`). There's also a debug breakpoint capability which we won't cover here.



Try changing a few things in the initial block of your testbench. After making changes to the code, you can relaunch the simulation by clicking the *Relaunch simulation* button.

Please note, sometimes Vivado simulator may stop before it actually reaches the `$finish();` statement in your testbench. That's because by default the tool only runs the simulation for a certain duration (which you can change), so don't be confused. In this situation, you can click the *Run All* ("Play") button (or press `F3`) to continue. Another thing to observe is the progress bar ("Current time") at the bottom of the Vivado program when running simulation. If your simulation runs for a while and does not appear to terminate as you wish, you should stop it, otherwise the program will continues dumping the simulated data to disk and your system may run out of space! A good practice when creating a testbench is to have some code that checks for timeout to abort the simulation. You will see some examples of that in the next section of the lab when we cover the Input Conditioning circuits.

You can also save your waveform (`Ctrl + S`) before exiting Simulation or Vivado to retain the configuration (such as Decimal radix) for next time you open it up.

5.1 Questions

1. The adder output is updated immediately with the inputs. This is because our adder circuit is a combinational circuit. Modify the structural adder so that the adder output is updated at every clock edge. Remember to use a REGISTER module from the EECS151.v library. Don't forget to add it to your project! Run the simulation again to see if your `adder_output` signal now is synchronous with the clock signal (remember to also wire the clock signal to your DUT). Screenshot the waveform and submit it to the lab report (pick a region of the waveform which the output transition w.r.t clock signal can be observed clearly).

2. Instead of manually setting the values of `operand1` and `operand2` in the initial block, modify the testbench to provide a test vector of 10 values for each of the operands (`operand1` and `operand2`). You can define arbitrary values for your test vectors. A test vector can be an array of wires or regs (or integer) with pre-defined values. Drive each value in the test vectors to your DUT per clock cycle. You can write some sequential always block(s) for this task. Submit the modified testbench and a screenshot of your waveform to the lab report.

   Here is an *example* of creating a 2D array of *reg* (works for *wires* too):

```
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)
```

Note that you can also have multiple `initial` blocks in your testbench code. The initial blocks run concurrently.

If you are unsure or confused about any of the steps above, don't hesitate to discuss with a TA.

6 Additional Testbench Techniques

- `@(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. It is a great debug method if you are not a fan of waveform.

  For example:

  ```
  $display("Wire x in decimal is %d", x);
  $display("Wire x in binary is %b", x);
  // You can even print out the current simulation time
  $display("Time %d", $time);
  ```

- `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:

```
$display("counter2_val: %d", dut.counter2_val);
$display("counter2_reg_q: %d", dut.counter2_reg.q);
```

As a general tip, you don't have to create separate projects for different Vivado simulations. You can add all the source files and testbench files to a single project. You need to set which one is the top-level module for *Simulation*. In the *Sources* pane, under *Simulation Sources*, right click on the testbench file of the DUT you want to test and choose *Set as top*.

7 Input Conditioning Circuits

To safely use the button signals on the PYNQ board with respect to clock signal of our circuits, we have to design the input conditioning circuits to handle metastability and button bounce. Check the prelab reading on Debouncer circuit 1 if you have not done so.

7.1 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.

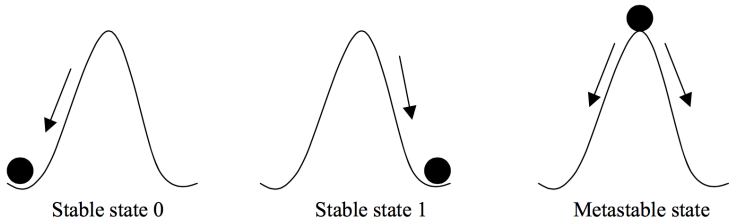


Figure 1: The 'ball on a hill' metaphor for metastability. If a register's timing constraints are violated, its output voltage oscillates and after some time unpredictably settles to a stable state.

The signals coming from the push buttons and slide switches on the PYNQ board don't have an associated clock (asynchronous). When the button signals are put through a register, its hold or setup time may be violated. This may put that register into a *metastable* state (Figure 1).

An asynchronous signal could violate timing constraints, and cause a 'mid-rail' voltage from a register to propagate to other logic elements. This 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 for this lab is relatively simple (Figure 2). For synchronizing one bit, it is a pair of flip-flops connected serially.

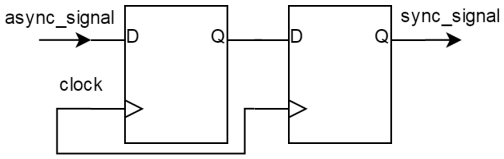


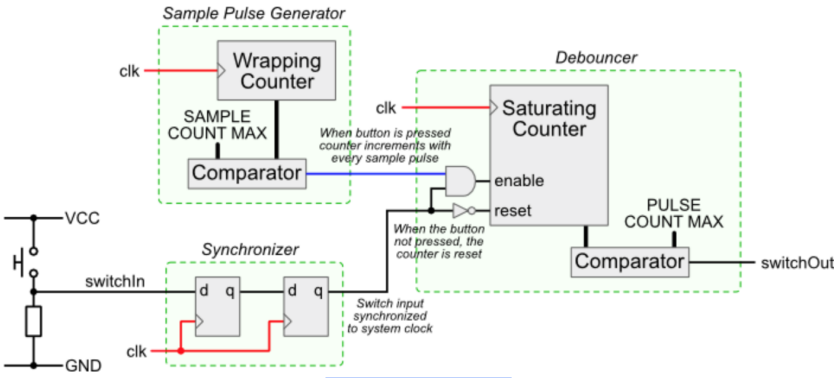
Figure 2: 1-bit 2 Flip-Flop Synchronizer

Edit `lab2/src/synchronizer.v` to implement the two flip-flop synchronizer. This module is parameterized by a `width` parameter which controls the number of one-bit signals to synchronize.

7.1.1 Synchronizer Simulation

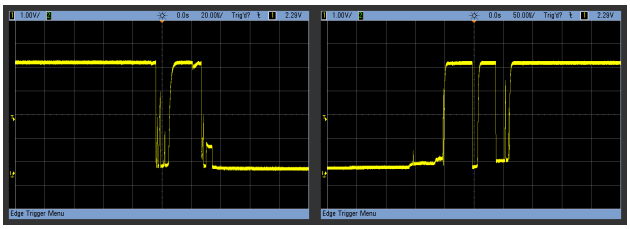
A testbench is provided in `lab2/sim/sync_tb.v`. Run it using Vivado simulation.

7.2 Debouncer



Recall this graphic from the prelab debouncer reading.

The debouncer circuit takes a button's glitchy digital signal and outputs 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 or depress a button, the signal doesn't behave like a perfect step function. Instead the button signal is glitchy due to mechanical "bounce". If we naively used the button signal directly there would be many spurious "button presses".

Look at `lab2/src/debouncer.v`. This is a parameterized debouncer which can debounce `width` signals at a time. 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_CNT_MAX` clock cycles. By default `SAMPLE_CNT_MAX` is set to 25000.

2. **Saturating Counter** - This is a counter that counts up to `PULSE_CNT_MAX`. If the sample pulse is high at a clock edge, increment the counter if the input signal is also high, else reset the counter to 0. Once the saturating counter reaches `PULSE_CNT_MAX`, it should hold that value indefinitely until the sampled input signal becomes 0. The `debounced_signal` of your debouncer should be an equality check between the saturating counter and `PULSE_CNT_MAX`.

You can use the same sample pulse generator for all input signals into your `debouncer`, but you should have a separate saturating counter per input signal.

7.2.1 Debouncer Simulation

A testbench has been provided in `lab2/sim/debouncer_tb.v`. Make sure you understand what the testbench is doing. Run it as usual.

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.

Pay attention to the printouts in the *TCL Console* to see if tests are passed or failed.

7.3 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 cycle wide pulse that the rest of our digital design can use.

Create a parameterized-width edge detector in `lab2/src/edge_detector.v`.

7.3.1 Edge Detector Simulation

A testbench is provided in `edge_detector_testbench.v`. Run as usual.

The edge detector testbench tests 2 scenarios, when the `signal_in[0]` transitions from low-to-high, and so does `signal_in[1]` a few cycles after. At each time, the corresponding bit of `edge_detect_pulse` goes high at the next clock edge and stays high for one cycle.

7.4 Button Parser on the FPGA

Now that we have tested all the modules required to produce a clean, stable button press signal. It's time to put everything together. In this section, we design a counter that uses the buttons on the PYNQ board to provide some functionalities.

7.4.1 Counter with with Button Parser

Look at `src/button_parser.v` which combines the synchronizer, debouncer, and edge detector in a chain. Create a Vivado project (or reuse existing project). Add these files

`lab2/src/synchronizer.v`, `lab2/src/debouncer.v`, `lab2/src/edge_detector.v`,

`lab2/src/button_parser.v`, `lab2/src/z1top_mode_counter.v`, `lib/EECS151.v`,

and `lab2/constraints/pynq-z1.xdc`. Set `z1top_mode_counter` as the top-level module. Generate a bitstream and test this circuit on the PYNQ board. Check that each button performs the right action as written in `z1top_mode_counter.v`.

7.4.2 Play and Pause your Counter

Right now, your `z1top_mode_counter.v` is in a static mode; you can only increment or decrement the counter value for each button press. Modify the code such that when you press BTN2, your

counter starts incrementing every one second (running mode), and pauses when you press BTN2 again (static mode). It is up to you if you may or may not want to update your counter with BTN0 or BTN1 when your counter is in running mode.

We'd like suggest that you think about the structure of your design ahead of coding. The debouncer circuit figure above is a great example. Remember that Verilog coding is all about describing your hardware circuit! Sketch a block diagram of your mode counter circuit in terms of the `button_parser` block and some register blocks from `lib/EECS151.v` as you see fit. Label the relevant input and output signals. Feel free to use any logic gates, MUXes, or adders. Don't worry about listing all the details. Submit your sketched diagram in your report.

8 Lab Deliverables

8.1 Lab Checkoff (due: 11.00AM, Feb 10th, 2020)

To checkoff for this lab, have these things ready to show the TA:

1. Demonstrate a working mode counter with button parser. Show that you can toggle the modes of your counter with BTN2.

8.2 Lab Report (due: 11.59PM, Feb 10th, 2020)

Please submit a short report to Gradescope with the following items.

1. Your answers to the questions 2.5 and 5.1. In addition, please also submit the block diagram of your mode counter.

A Local Dev Setup

Besides Vivado, we'll cover installing free tools for simulation for each OS here so you can have more options to do your lab/homework exercises.

A.1 Linux/OSX

1. Add vivado to your `$PATH` by adding this in your `.bashrc`:

   ```
   export PATH="/opt/Xilinx/Vivado/2019.1/bin:$PATH"
   ```

2. Install Icarus Verilog
   - Linux: `sudo apt install iverilog`
   - OSX: `brew install icarus-verilog`
3. Install Verilator
   - Linux: `sudo apt install verilator`
4. Install gtkwave
   - Linux: `sudo apt install gtkwave`
   - OSX: Download the app

In a terminal you should be able to successfully run `vivado`, `iverilog`, `gtkwave`.

A.2 Windows

Windows packages for Icarus are available. You should be able to download the appropriate .exe for your machine. Gtkwave can be installed along with Icarus (just make sure you check the box for it). Lastly, here is a link to install Cygwin. When you install Cygwin, you need to check the boxes for make and git as shown in Fig. 3, and also Python 3.6.

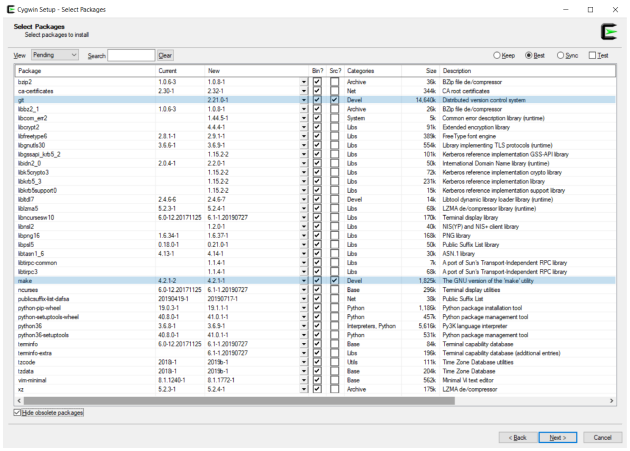


Figure 3: Check make and git with your cygwin install!

After installing Icarus, gtkwave, and cygwin, you'll need add Vivado, Icarus, and gtkwave to your Windows PATH with the below steps.

- Go the windows control panel and find the "Edit the system variables menu". Alternatively, if you just search for "system variables", the menu should pop up.
- Click the "Environment Variables" button near the window's bottom right corner (Fig. 4).
- Double click the "Path" variable in the user variables dialog (Fig. 5).
- Click the next empty row and paste the path of the appropriate program (Fig. 6). Hit okay and you're done!

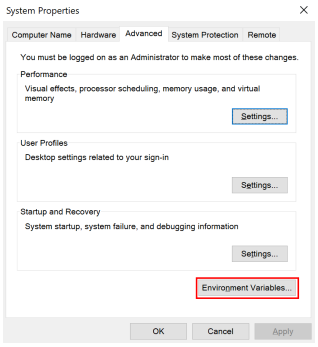


Figure 4: The System variables dialogue.

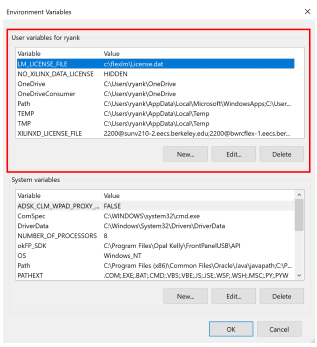


Figure 5: Environment variables dialogue.

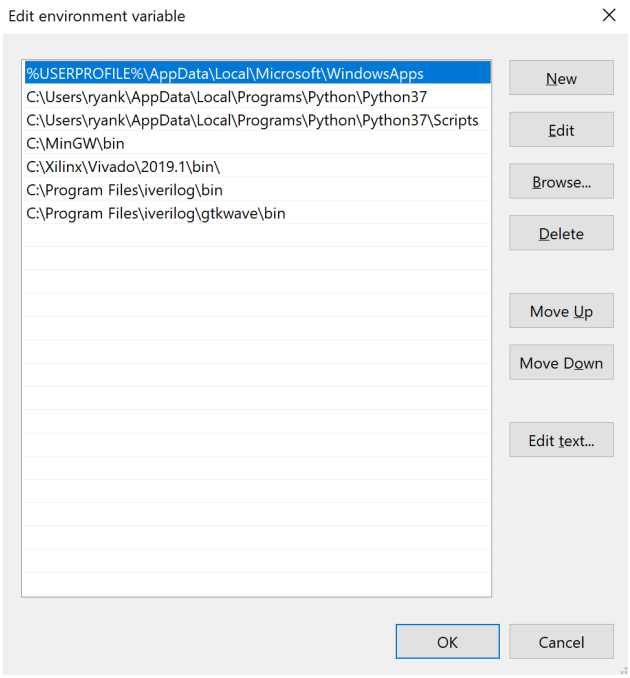


Figure 6: Adding programs to your PATH.

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
- Sp19: Christopher Yarp, Arya Reais-Parsi
- Fa19: Cem Yalcin, Rebekah Zhao, Ryan Kaveh, Vighnesh Iyer
- Sp20: Tan Nguyen
- Fa20: Charles Hong, Kareem Ahmad, Zhenghan Lin