ASIC FPGA Prototyping - Final Report

Jared Botte

Jared Botte

Dr. Mark Johnson

Spring 2022

1 Credit Hour

1 Overview

This report will discuss the work performed over the course of the Spring 2022 semester for ECE 49600 - ASIC FPGA Prototyping. I will discuss the deliverables created, future work, and hardware limitations encountered.

2 Summary

The primary goal of this independent study was for me to get more exposure to FPGA and Digital Logic development. To do this I took the labs completed in ECE 33700, expanded upon them, and created lab experiments to bring those labs onto a physical FPGA development board. The Altera DE2-115 FPGA development board was used as the hardware platform for these labs, as it is used in ECE 43700 and contains many useful peripherals. The ultimate goal of these labs is to integrate them as part as ECE 337 to further reinforce the topics learned in that class.

3 Deliverables

All of the work produced for this independent study are located in two GitHub repositories.

jaredbotte/asic-prototyping-tools contains the scripts I created to automate compilation and programming of the DE2-115 development board. They are based on the ECE 437 scripts, and should work in any standard linux distribution.

jaredbotte/asic-prototyping-labs contains the bulk of my deliverables, which includes lab documents (in both markdown and PDF format), solution documents (in markdown) and my design code solutions for the lab. It also contains some additional work that never made its way into labs, but may be useful if this work is continued in the future.

4 Preparation

The first 2 - 3 weeks of my independent study was primarily focused on obtaining the hardware, installing and configuring the software, and documenting the setup so that it could be reproduced. This took longer than I had anticipated, as I had no background in Intel's FPGA workflow so I did not understand the steps needed to program the development board. I also had to reverse engineer the scripts used by ECE 437 that were written in perl, which I had no prior experience with.

Once I was able to determine the steps needed to program the board and define the pins, I created python scripts and a makefile to facilitate the compilation process. Lastly, I documented this process as best as I could in the `setup.md` document, so that the setup can be reproduced.

5 Labs

5.1 Adder Error Detector

This lab, designed to be completed first, is a soft introduction to the DE2-115 development board and the procedure for programming the board. The sensor error detector and flexible adder are taken from ECE 337 and put onto the board. This introduces the students to the buttons, switches, LEDs, and 7-segment displays on the development board.

5.2 Counter Shift Register

This lab, designed to be completed second, introduces students to the clock available on the development board, as well as clock dividers, counters, and shift registers. A module is created to show hexadecimal values on the 7-segment displays, and this is used to show the current value of the counters. In this lab, serial-to-parallel shift registers are also explored.

5.3 Custom Game Controller

This lab, designed to be completed third, is one of my personal favorite labs. It uses physical and implemented shift registers to create a custom game controller that operates on the same principles as the SNES and NES systems. This lab is the first lab that requires quite a bit of thinking and a strong understanding of counters, state machines, and shift registers. This lab combines all the knowledge from the previous labs into a cool practical implementation.

5.4 UART Receiver

This lab, designed to be completed fourth, takes the UART lab completed in ECE 337 into the physical domain. In this lab, a module is created to display alphanumeric characters on the 7-segment displays. The displays are linked together and connected via an RS-232 cable, which allows communicating with the dev board over a terminal on the host machine. This allows displaying received messages on the board.

The UART lab was originally going to print the received characters to the LCD display on the dev board, however there is no way to easily interact with the screen on the DE2-115, so the use of the LCD screen was scrapped.

5.5 Random Number Generator

This lab was initially created as part of the custom game controller series, when I had intended to make labs that created a game. Unfortunately, for reasons discussed later on in this document, a game was never implemented. This lab is too short to be a standalone lab, although is not too far from becoming part of a larger lab.

This lab focuses on implementing a Linear-Feedback Shift Register to create Pseudo-Random numbers. This can be used with a game implementation to provide some form of somewhat random numbers. LFSRs are also ideally implemented in hardware designs, so I believe they are a cool topic to introduce to students.

5.6 Audio FIR Filter

This lab is designed to be completed last, and explores using SPI to communicate with an external device. Using SPI, we send audio samples to the dev board, where the FIR filter implemented in ECE 337 is used to modify the audio. It is then sent back over SPI so that the controller can replay it.

The original plan for a FIR filter lab was to use the 24-bit audio Codec that's on the DE2-115 development board. However, due to complexities discussed later in this lab, that idea had to be scrapped. Instead, the STM32 development board used in ECE 362 was used as an SPI controller and handles the ADC and DAC.

5.7 VGA

As part of the custom game controller series, I had planned to use the VGA peripheral on the screen to implement a game, playable with the custom game controller. Basic interfacing with the screen was completed and can be found in the `vga-base-files` folder, but creating a frame buffer and more complex graphics would involve interfacing with the SDRAM on the board. This was outside the scope of the independent study and therefore was not implemented.

VGA may make for an interesting lab on its own as students would need to use clock division and the design isn't very complex, but understanding VGA is something that would need to be taught.

6 Problems Encountered and Limitations

As mentioned above, many of these labs were going to utilize some of the different peripherals on the DE2-115 development board. Examples of this are the FIR filter lab which was going to use the audio jacks and the UART lab which was going to use the LCD screen.

The problem with this however is that the IP cores provided to interface with the peripherals are all meant to be interfaced with an Avalon bus. The Avalon bus is Intel's internal FPGA bus that is used in designs that implement microprocessors. It is essentially the Intel FPGA version of an APB bus.

Unfortunately, this means that to interface with many of the components on the board, we would need to implement a processor, which is outside the scope of what these labs aim to do. It's theoretically possible to write IP cores to interface with each of the peripherals, although that still comes with its own complexities.

7 Conclusion

I hope that these labs are fun, interesting, and help students apply the knowledge learned in ECE 337. I believe that providing real world examples, and showing real world implementations of the topics we learn in ECE is critical in aiding students understanding.

8 Appendix

Altera DE2-115 Setup on Ubuntu 20.04

Jared Botte

1 Overview

This setup document aims to guide you through the setup and initial configuration of an Ubuntu 20.04 system with the Altera DE2-115 development board. Setup on Windows will be similar, but with different steps to set environment variables.

2 Download the necessary files

To begin, open a terminal window and download the ASIC prototyping tools and scripts:

```
cd ~
git clone git@github.com:jaredbotte/asic-prototyping-tools.git tools
```

2.1 Set the TOOLSDIR environment variable

The provided scripts will attempt to look for each other as needed by referencing the `$TOOLSDIR` environment variable. It is important that this gets set and points to the right place.

```
vim ~/.bashrc
```

Note that this file starts with the name of your command line interpreter. For `zsh` use `zshrc`. For `csh` use `cshrc`.

Add the following line to the top of your `rc` file:

```
export TOOLSDIR=/home/<user>/tools
```

Where `<user>` is replaced by your username.

3 Installation of Quartus Prime

3.1 Download and run the installer

Go to https://fpgasoftware.intel.com and select the most recent "lite" version of Quartus Prime to download. Download the main installer under the "Combined Files" tab.

Extract the downloaded file by right clicking and hitting extract here, and open up a new terminal window. We'll use the terminal window to start the installation.

```
cd ~/Downloads
./setup.sh
```

If you are unable to find the file or it is not able to be run, first ensure that you are in the correct directory by running `ls` and looking for the `setup.sh` file. If necessary, `cd` into the folder containing the setup file. If you are in the correct folder and it's unable to run, try running `chmod +x setup.sh` before running `./setup.sh` again.

Follow the steps in the setup wizard and ensure that the installation directory is set to `/home/<user>/intelFPGA_lite/21.1` where `<user>` is your username and `21.1` is the version of Quartus Prime you are installing. Also ensure that you install all the tools and support equipment.

3.2 Add Quartus Prime command line tools to your path variable

The provided scripts require access to the command line tools for Quartus programs, so we'll need to add them to our `PATH` variable. Once again, open your `rc` folder and add the following lines to the beginning:

```
export PATH=$PATH:/home/<user>/intelFPGA_lite/21.1/quartus/bin/
export PATH=$PATH:/home/<user>/intelFPGA_lite/21.1/questa_fse/bin/
```

Now for the changes to take effect, run `source ~/.bashrc`

4 Obtain a QuestaSim license through Intel

You may or may not need to obtain a license through Intel for QuestaSim. I was unable to talk to the board until I did, and it will allow you to use `vsim` so it is nice to have either way.

First, go to Intel's Self-Service Licensing Center and create an account. Once you've recieved an email setting up your account, login to the website again.

On the menu bar below "Intel FPGA Self-Service Licensing Center", click the "Sign up for Evaluation or Free Licenses" tab. Select "Questa*-Intel FPGA Starter Edition SW-QUESTA". Select the edit icon under the "# of Seats" column and type in 1. Agree to the terms of use, and click "Get License"

Next you will be asked to either create a new computer, or add to an existing one. Click "+New Computer".

A prompt will now show up asking for information about your computer. First, give your computer a name. Then, select "FIXED" for the license type.

Under the computer type tab, select "NIC ID". You'll need to obtain this ID from your computer to allow the license to work. It can be obtained by running `ip addr show` and looking for the first set of hexidecimal digits right after `link/ether`. Put this into the computer ID and click "Generate License". You should eventually recieve an email containing the license file.

Copy this license file into your `intelFPGA_lite/21.1/questa_fse` directory and once again open your `.rc` file. Add the following line:

```
export LM_LICENSE_FILE=/home/<user>/intelFPGA_lite/21.1/questa_fse/LR-*****_License.dat
```

The "****" should be replaced by the license number in the name of the file. Don't forget to run `source ~/.basrc` after adding these lines.

Setup should now be complete.

ASIC FPGA Prototyping - Sensor Error Detector, 1-, and 8-bit Adders

Jared Botte

1 Overview

In this lab, we will be implementing the Sensor Error Detector, 1-, and 8-bit adders on the DE2-115 development board. This lab will assume that these assignments have already been completed, and will walk through the steps of implementing these programs on a development board for testing.

2 Setup

To begin, open a terminal window, create a new directory, and move into it. Also, setup the directory structure and copy over the base makefile.

```
mkdir adder-error-detector
cd adder-error-detector
mkdir -p source include
cp $TOOLSDIR/makefile .
```

This will be the standard setup for most labs.

3 Demonstrating a sensor error detector

Let's demonstrate the sensor error detector we created. First, choose any of your designs and copy into your new source directory. Rename it `sensor.sv`.

Next, we'll need to modify our file to use inputs on the board. The options that come to mind when looking at the board are the buttons (or keys as they're labeled) and the switches. We'll also need to choose an output. Luckily, there are tons of LEDs to choose from.

3.1 Change the inputs to switches 3:0

To change the sensors to be switch input, we'll need to change the input declaration at the top of our module.

Change it to look like this instead:

```
input logic [3:0]SW,
```

Hopefully the transition here is clear; we've set the input type to `logic` instead of `wire`, and told it to use switches 3, 2, 1, and 0 as input.

Now, change all references in your modules to use `SW[x]` in place of `sensors[x]`.

3.2 Change the output to LEDG8

Now that we've changed the input to use the switches, we'll need to find a way to display the output. Let's output the error on green led 8 (between the hexadecimal displays). Although it's only a one bit output, we must set it to go from 8:8 in order to isolate LEDG8. Replace the output line with:

```
output logic [8:8]LEDG
```

Lastly, replace `error` with `LEDG[8]`. This will now light up LEDG8 when an error has been detected.

3.3 Program the design to the board

To program your design to the board, return to the terminal window. In your main directory for the lab (i.e., not in the `source` directory, but the parent directory), run `make sensor`. This will compile the design, create the necessary files to map the physical pins on the board to your design, and upload the design to the development board.

For more make rules, run `make help`

3.4 Try it out!

Now that you can test your sensor error detector using the switches and leds, try messing around with the development board some more.

3.4.1 Challenge 1: Use the buttons (keys) instead of the switches

Refer to the DE2-115 documentation (Specifically chapter 4) to determine the names of the keys and their functionality.

Question: Why does the error detector appear to have "flipped" behavior when using the buttons?

3.4.2 Challenge 2: Use a red LED to show the error

Now try using LEDR[8] as your error output. Also try using LEDR[0].

Question: What happens if you initialize more than one red LED by saying `output logic [8:0]LEDR`? What happens to all the LEDs that are initialized? What about the uninitialized ones?

4 1-bit adder

Next we will move onto our 1-bit adder program.

Copy your 1-bit adder into the `source` folder and make a copy of it called `adder_1bit_mapped.sv`. Use this mapped version for this part of the lab (more on why later)

Open `adder_1bit_mapped.sv` in an editor, and replace your inputs and outputs with switches and LEDs respectively. When finished, run `make adder_1bit_mapped` to test out your design.

Question: Did you use 3 input statements and 2 output statements? If you're using sequential Switches/LEDs, do you need 3/2 statements respectively? How can you condense these into one statement each? Try it out!

5 8-bit adder using n-bit adder template

Now that the 1-bit adder is working, let's add in our 8-bit full adder. remember that the 8-bit adder uses the n-bit template adder, so you'll need to copy that file over too.

5.1 Why did we need to make a copy of the 1-bit adder earlier?

Our template requires the 1-bit adder to have inputs with names `a`, `b`, and `carry_in`, and outputs with names `carry_out` and `sum`. On top of that, if we were to map our 1-bit adder inputs directly to switches on the board, all our adders would have the same inputs and not be connected properly! Therefore, we created a separate file to map to our board, and leave the generic one for later use. In this case, we're using a wrapper file to create the 8-bit adder, so we'll map our inputs and outputs there.

5.2 Setup the 8-bit adder wrapper to use switches and LEDs

Try to modify the 8-bit adder file to take input from SW[7:0] (input a), SW[15:8] (input b), SW[16] (carry_in) and to put the output on LEDG[7:0] (sum) and LEDG[8] (overflow). Once you've done this, try running it and testing that it works as expected.

5.3 Why does Quartus Prime give us the error "this block requires a name"?

Chances are if you've tried to compile this that you've gotten an error on the for loop of the n-bit adder module. The error states that "this block requires a name". This is something specific to Quartus Prime (as opposed to Quartus II used in ECE 337). To fix this, simply add a name for the generate loop after the `begin` statement like so:

```
genvar i;

generate
for(i = 0; i < BIT_WIDTH; i = i + 1)
    begin: adder_generator
        <adder logic here>
    end
endgenerate
```

5.4 Try it out!

5.4.1 Challenge 1: Display the state of the switches on the red LEDs above them

Try using the red LEDs above the switches to show their current state. Your implementation should be 2 lines of code. 1 port declaration and one `assign` statement.

5.4.2 Challenge 2: Nicer displays

Although counting in binary is a favorite pastime of us computer engineering students, we have 7-segment displays on our development board! Let's use them to make our life a bit easier so we can verify that our adder works as expected. Set the leftmost displays HEX7/HEX6 to map to SW15-SW12/SW11-SW8 respectively. Map the middle displays to the other switches, and map the rightmost three displays to the carry out and sum values.

Note: It may be useful to use a 2d array to store and access the hexadecimal character values for the seven segment displays. Also note that the seven segment displays are active low.

Hint: How can you control an output value using an input value using a hardware implementation?

ASIC FPGA Prototyping - Counters, "1101" Detector, and Serial-to-Parallel Shift Register

Jared Botte

1 Overview

In this lab, we will explore state machines, counters, and shift registers.

2 Setup

To begin, open a terminal window, create a new directory, and move into it. Also, setup the directory structure and copy over the base makefile.

```
mkdir counter-shift-register
cd counter-shift-register
mkdir -p source include
cp $TOOLSDIR/makefile .
```

This will be the standard setup for most labs.

3 New make target

I've added in a new make target that will just compile a design. This is a good way to check that your design compiles before moving on and creating wrapper files. To use this target, run:

```
make %_compile
```

Where `%` is the module name you'd like to compile. This will also compile any modules referenced by the one provided.

4 "1101" Detector

In the first part of the lab, we will be using a state machine to make a "1101" detector. We will provide the device with a stream of 0's and 1's and it should light up any time "1101" is detected in the stream.

4.1 Create a general "1101" detector

Create a new module `detector_1101` with the following parameters:

Name	Direction	Width	Description
clk	input	1 bit	clock signal
n_rst	input	1 bit	reset signal
data	input	1 bit	data stream
state	output	3 bits	the current state
out	output	1 bit	true (1) if "1101" detected in stream

4.2 Create a mapped version

Next, create a module `detector_1101_mapped` that maps the general detector to the FPGA I/O. Use pushbuttons for the `clk` and `n_rst` signals, a switch for the `data` signal, and green LEDs for the `out` and `state` signals.

Confirm that your "1101" detector works as expected.

5 Counters

The rest of this lab will focus on counters. We will begin by having the counter be controlled by a push button, then move on to controlling it from the clock on the DE2-115 development board. To begin, copy over your implementation of the `flex_counter` from ECE 337. This should already be a fully functional n-bit counter, so we'll only need to create a wrapper file.

5.1 Create a 7-segment display module

Since we'll be using the 7-segment displays in this and many future labs, let's create an easy-to-use module called `hex_display` to deal with mapping a 4-bit value to a hex display.

Module `hex_display`:

Name	Direction	Width	Description
value	input	4-bits	The value to display on the screen
disp	output	7-bits	The value sent to the 7-segment display

5.2 Show a 4-bit counter on a 7-segment display

For this part of the lab, choose push-buttons (keys) to act as the clock, reset, and clear signals and display the output of the counter on one of the 7-segment displays. Set the rollover value to `4'hf`, and `count_enable` to `1'b1`. Name your wrapper file `counter_4bit.sv`.

Hint: If your counter seems to be stuck at 0, remember how the push-buttons are connected to the FPGA.

Question: What is the difference in behavior between the `clear` button and the `n_rst` button? What happens when you press them?

5.3 Try it out!

Now that we've got our 4-bit counter working, lets try counting higher!

5.3.1 Challenge 1: Show an 8-bit counter on 2 7-segment displays

Create a new wrapper module, `counter_8bit` that creates and 8-bit flex counter, and shows the result on 2 7-segment displays.

5.4 Connect the counter to the system clock

Hitting a push button to verify all our values work takes a lot of clicks (well, 255 to be specific). Let's connect the counter to the system clock so that it counts automatically. To do this, let's copy the 8-bit counter into a new module called `auto_counter`. Let's also change the clock push button to control `count_enable` instead. Once done, run it on the development board.

5.5 Slowing things down

You probably noticed the displays both show eights. This is because the counter counts fast. Very fast. 50 MHz fast. Try hitting the `count_en` button and seeing different values. Let's try slowing things down a bit. The `flex_counter` module can also be used as a clock divider.

Question: What do we have to divide our clock input by to get a 20 Hz clock?

Question: How can we divide our clock using the flex counter? How many bits wide does the counter need to be to get the clock to 20 Hz? What should the rollover value be set to?

Once you've answered the above questions, try implementing it. Also make the counter a 16-bit counter that uses 4 7-segment displays

6 Serial-to-Parallel Shift Register

For the final part of this lab, we will create a Serial-to-Parallel shift register. Once we've created this shift register, we'll use it both to make a different "1101" detector and as a means of communication with a game controller.

6.1 1-bit Serial-to-Parallel Shift Register

Just like our n-bit adder implementation, we will implement an n-bit shift register by starting with a 1-bit shift register. This is a super simple device, and is just a flip flop. Create a new module `stp_1bit` to implement it. Have it reset to a `1'b1`.

6.2 n-bit Serial-to-Parallel Shift Register

Next create a module called `stp_nbit` which uses a parameter `BIT_WIDTH` and a generate block to create an n-bit shift register using the 1-bit shift register previously made. You should do this the same way you created the n-bit adder in ECE 337.

6.3 1101 Detector without State Machines

Now that we have our n-bit StP Shift register out of the way, let's try using a 4-bit shift register to detect a "1101" input. Create a new module `stp_4bit` and map it the same way you mapped the previous state detector. Instead of the state being displayed on the LEDs, display the parallel output of the shift register.

Think about how you can detect a "1101" using the parallel output of the shift register, and write a statement in your module to detect this and display it on a green LED.

ASIC FPGA Prototyping - Custom Game Controller

Jared Botte

1 Overview

In this lab, we will put together some of the building blocks we have learned to create our very own game controller! We will use the same principles that the NES, SNES, and Virtual Boy used to communicate with their controllers.

2 Required materials

For this lab the following materials will be needed:

- Breadboard (Large one will do but a small one is ideal)
- Buttons (any size/type will work)
- 10k Resistors
- 5 Jumper cables with at least one female end (Can be obtained from the ECE shop)
- 74HC165 or comparable parallel-to-serial shift register (Can be obtained from the ECE shop)

3 Setup

To begin, open a terminal window, create a new directory, and move into it. Also, setup the directory structure and copy over the base makefile.

```
mkdir custom-game-controller
cd custom-game-controller
mkdir source
cp $TOOLSDIR/makefile .
```

This will be the standard setup for most labs.

Also, copy over your `flex_counter` and `flex_stp_sr` modules from previous labs.

4 Basic Game Controller

The principles learned so far in these labs can be used to talk to old game controllers, or even a custom game controller that we can make ourselves! In this lab, we'll make a custom game controller and use our Serial-to-Parallel Shift Register implementation to talk to it! The concepts of this section are exactly how NES, SNES, and Virtual Boy Controllers work, and it's super simple! I'll be referring to the SNES the most since I have a reproduction controller on hand, but all three controllers/consoles work the same way and are actually compatible[1].

4.1 How the SNES Controller works

As alluded to above, the SNES controller is essentially just a 16-bit Parallel-to-Serial IC that has it's signals controlled by the console. The console "polls" the controller at a rate of 60 Hz. The console "polls" the controller by doing the following:

1. Send a 12µs wide latch pulse.

[1] Well, almost. The NES controller has 8 buttons while the SNES controller has 12. This means that you can only use the first 8 buttons of the SNES controller if hooked up to a NES. Likewise, hooking an NES controller up to an SNES means you'll be missing some buttons. Also, they each use different connectors (although they contain the same signals).

- On the SNES, this is a positive pulse.
- This puts the state of the buttons into the shift register

2. 6µs after the falling edge of the latch pulse, 16 clock pulses are sent.
   - The period of these pulses is 12µs
   - They have a 50% duty cycle
3. On the falling edge of each clock pulse, the data line is recorded.
4. This is mapped back to the buttons, and is used by the system for input.

A couple things to note:

- The buttons on the SNES controller are active low.
- Since the SNES controller only has 12 buttons, the last 4 pulses are always high

I disassembled my reproduction SNES controller, with the intention of determining the chip that converts the button inputs to a serial stream, but the chip is unfortunately covered, so you can't see it. Luckily the internet has plenty of pictures of real SNES controllers such as this one from ifixit:

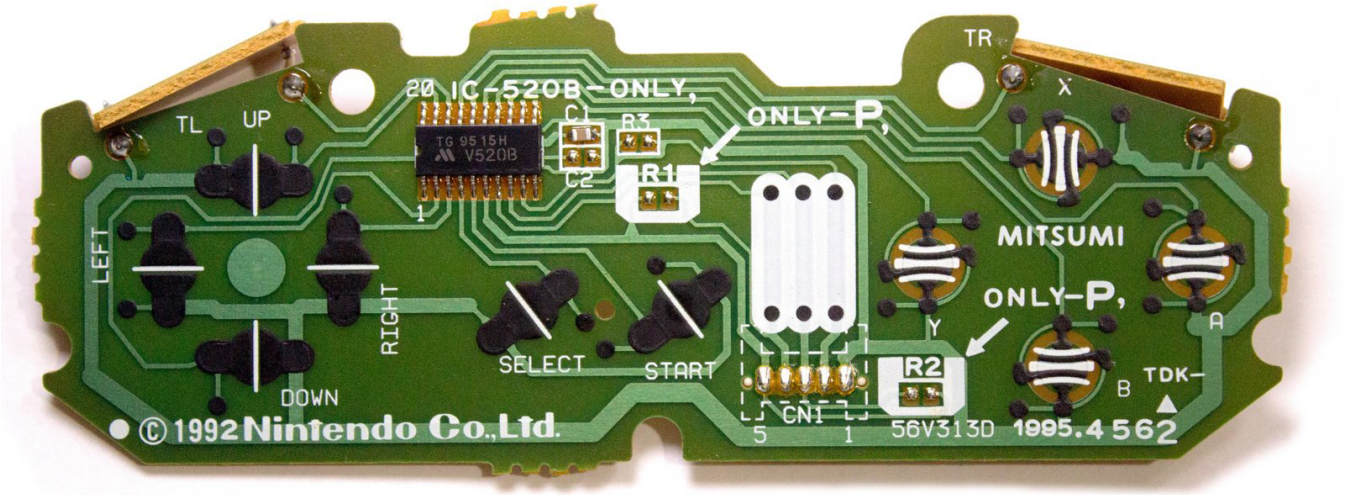


Figure 1: The SNES controller PCB. Source: ifixit.com

As you can see, it uses the 520B IC. Unfortunately, I was unable to find a datasheet for that specific chip anywhere online, but plenty of people vouch that it's a 16-bit parallel-to-serial shift register. Given the protocol that this controller uses and the traces on the controller's PCB, we can see that this makes sense.

4.2 What is the 74HC165 and what do we need it for?

The SN74HC165 is an 8-bit Parallel-to-Serial (PtS) Shift Register.

It works by allowing the `SH/~LD` pin to determine the current operation of the chip. When `SH/~LD` is high, the values in the shift register are shifted out. It's important to note that the shifted value out starts from pin 'H' not 'A'. The value on `SER` determines what value gets shifted in. For this lab, we'll pull it low. When the `SH/~LD` line goes low, the values currently asserted on lines A-H are loaded into the shift register.

The following table describes the connections to the 74HC165.

signal	pin name
latch	`SH/~LD`
pulse	`CLK`
data	`~QH`
+3.3v	`VCC`
GND	`GND`

You should take a look at the data sheet (linked above) for this IC, since we will need to understand how it works to talk to it. Think about the difference between the signals used for the SNES controller and the signals we'll use for our controller.

A couple of differences between the SNES controller and our custom controller are listed below:

- On the SNES, the latch pulse is high. Since our IC will shift when the latch signal is high and load when the signal is low, our latch pulse needs to be a negative pulse.
- We will wire our controller to have active high buttons.
  - However, we will use the inverted output on the '165, so the buttons will appear to be active low
  - This is also because the inverted output is on the same side of the IC as the other signals we need. This will make wiring neater.
- The '165 only has 8 inputs. Therefore, we'll only send 8 clock pulses to our controller.

4.3 Tips for building your controller

It's now time to build your custom controller! You are free to build your controller however you like, but here are some requirements/tips:

- You can have a maximum of 8 buttons (unless you daisy-chain '165 chips)
- You should have a minimum of 6 buttons
- The buttons should be wired active high (as mentioned above)
- Each button should have a pull down resistor on the signal side
- All unused inputs on the shift register should be pulled low
- The `SER` and `CLK INH` pins must be pulled low
- Don't forget to connect your power rails together
- Neat wiring is critical for debugging and mapping

4.4 Button mapping

If it isn't clear already, we'll be using our Serial-to-Parallel shift register implementation to keep track of the buttons being pressed. We'll poll the controller and take in the data one bit at a time. To do this, you'll need to know which buttons are mapped to which input of the '165 shift register. You should create a picture with your buttons labeled and a table that matches these labels to the bit of the StP shift register. My custom controller is shown as an example below:

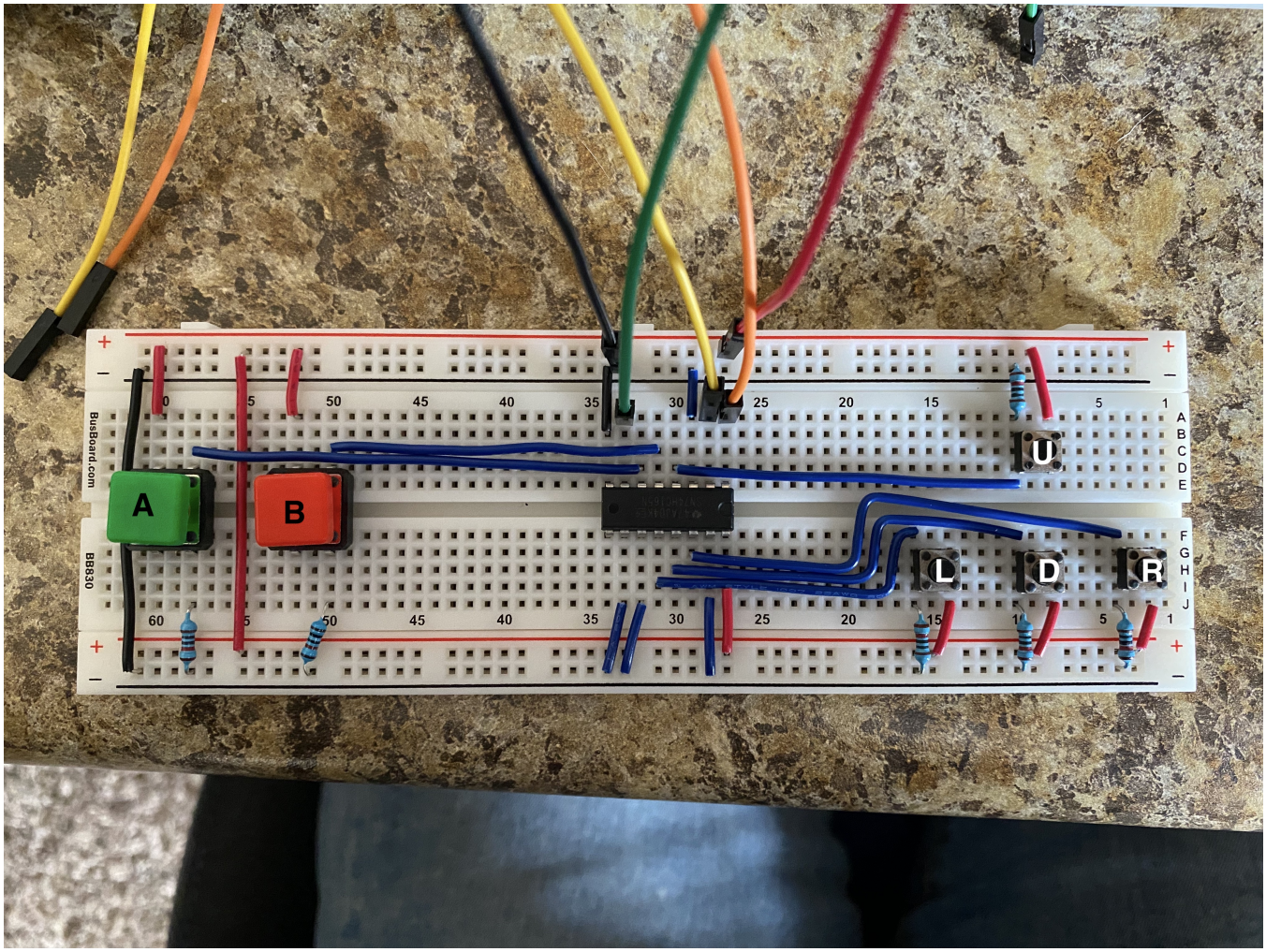


Figure 2: My Custom Controller

Bit #	Button Mapping
0	low (unused)
1	L
2	D
3	R
4	low (unused)
5	U
6	B
7	A

5 Putting it all together

In the final part of this lab we will be using our shift register and counter implementations to communicate with our custom game controller. We won't be using the controller to do anything too crazy, just light up the LEDs for now, but we will use the controller in later labs to do some cool things! You should try experimenting with the development board and controller to do something cool.

5.1 Physical Connections

We have 5 total wires that will connect between our controller and the development board. One wire is power, one wire is ground, the remaining three wires are the signals: clk (pulse), latch, and data. We'll connect these wires to GPIO pins on the development board. Connect latch to GPIO[0], clk/pulse to GPIO[1], and data to GPIO[2]. On the expansion header, pins 29 and 30 are +3.3v and GND respectively. Connect them to your controller as well. Be sure to check the polarity before plugging it into your controller!

5.2 `custom_controller` module

To start, create a new module `custom_controller` with the following parameters. Keep the signal names general for now, as we'll create a wrapper to map this to our actual controller later. This will allow us to specify multiple controllers in the future.

Name	Direction	Width	Description
clk	input	1 bit	clock signal. Assume it's a 50 MHz clock
n_rst	input	1 bit	reset signal
data	input	1 bit	input data signal
latch	output	1 bit	generated latch signal
pulse	output	1 bit	generated pulse signal
buttons	output	8 bits	the current states of the buttons

5.2.1 Signals

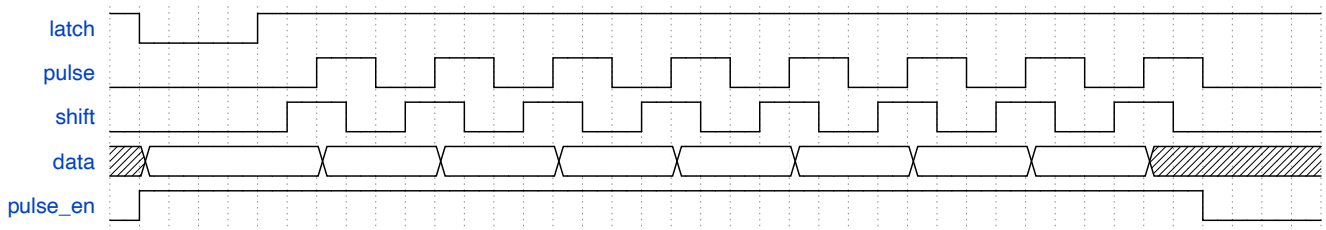


Figure 3: The controller signals

It is important to note that the `latch`, `pulse`, and `data` signals are the only ones visible to the controller. The `shift` and `pulse_en` signals are internal to the implementation and are shown to help with your implementation. The signals are all described in more detail below.

- The `latch` signal is a negative pulse that is 12µs wide and has a frequency of 60Hz
- The `pulse` signal will control the shift register on the controller itself. It has a period of 12µs
- The `shift` signal will control the StP shift register, and occurs 3µs before the `pulse` signal
  - This is so that we can get the data bit in a stable state
  - We can't just use the negative edge of `pulse` because the first bit is asserted on load
- The `data` signal comes from the controller and is high for not pressed, low for pressed
- The `pulse_en` signal enables the `pulse` and `shift` signals

5.2.2 Additional Specifications

- Assume the input `clk` is 50 MHz
- `buttons` should not update until after all 8 data pulses have been received
  - I.E. We should not see the shifting as the buttons come in

5.3 `mapped_controller` module

This module should be pretty straightforward, and will connect the GPIO pins on the dev board to your `custom_controller` implementation.

name	direction	description
CLOCK_50	input	a 50 MHz input clock from the development board
[0:0] KEY	input	KEY[0] will act as the reset button
[7:0] LEDG	output	the green LEDs will show the button states
[2:0] GPIO	inout	GPIO will be declared as inout since some signals are inputs and some are outputs

5.4 How to Implement the `custom_controller` and `mapped_controller` Modules

To begin, create the `mapped_controller` module and connect the signals appropriately. This way, you are able to test your generated signals as you go along.

5.4.1 Create the 60Hz `latch` Signal

The first challenge is to create a 60Hz `latch` signal that follows the provided specifications. Once completed, connect GPIO[0] to an oscilloscope and confirm the signal acts as intended.

Question: How many 50MHz clock cycles are in 1 60Hz cycle?

A proper `latch` signal can be seen in the images below.

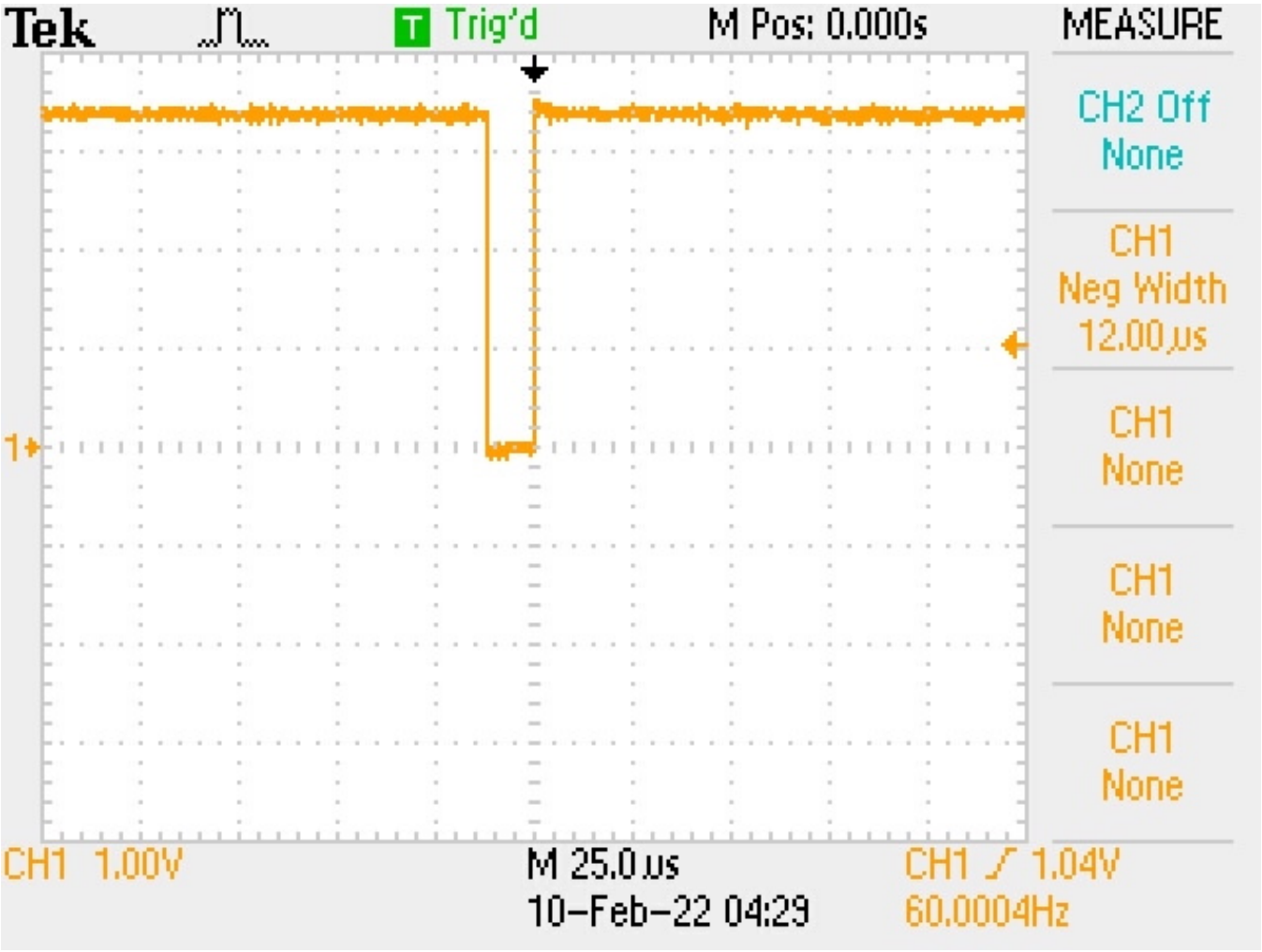


Figure 4: The latch signal up close

The latch signal up close. You can see it is exactly 12µs wide.

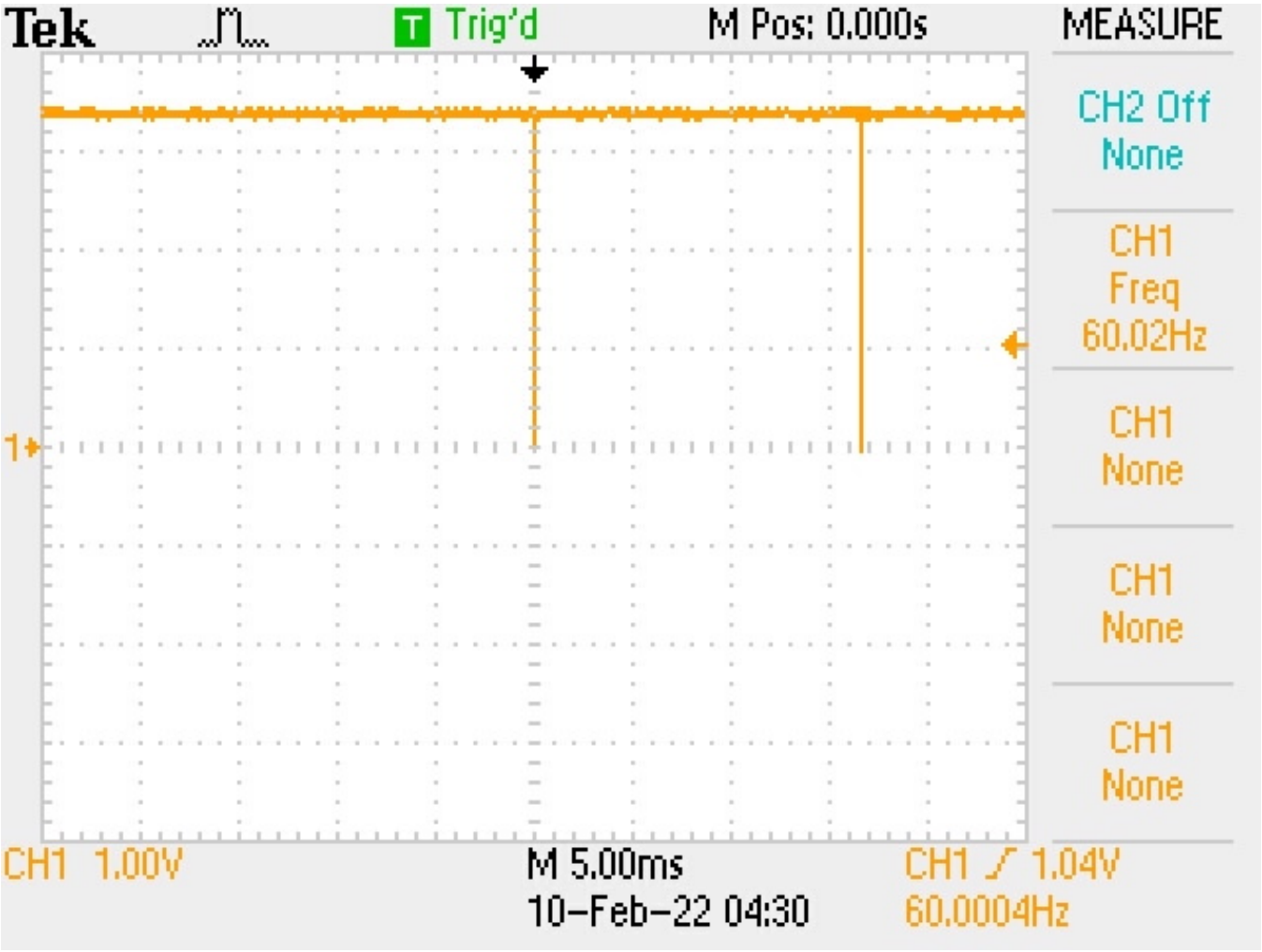


Figure 5: Zoomed out latch signals

Multiple latch signals. You can see that they are occurring at a frequency of 60Hz

5.4.2 Create the `pulse` Signal

Now that you have a proper `latch` signal, create the `pulse` signal, which should start right after the `latch` signal goes low.

Question: How can we use our existing signal to align the `pulse` signal with the `latch` signal? How can we stop the `pulse` signal after 8 pulses?

Proper `latch` and `pulse` signals can be seen together below.

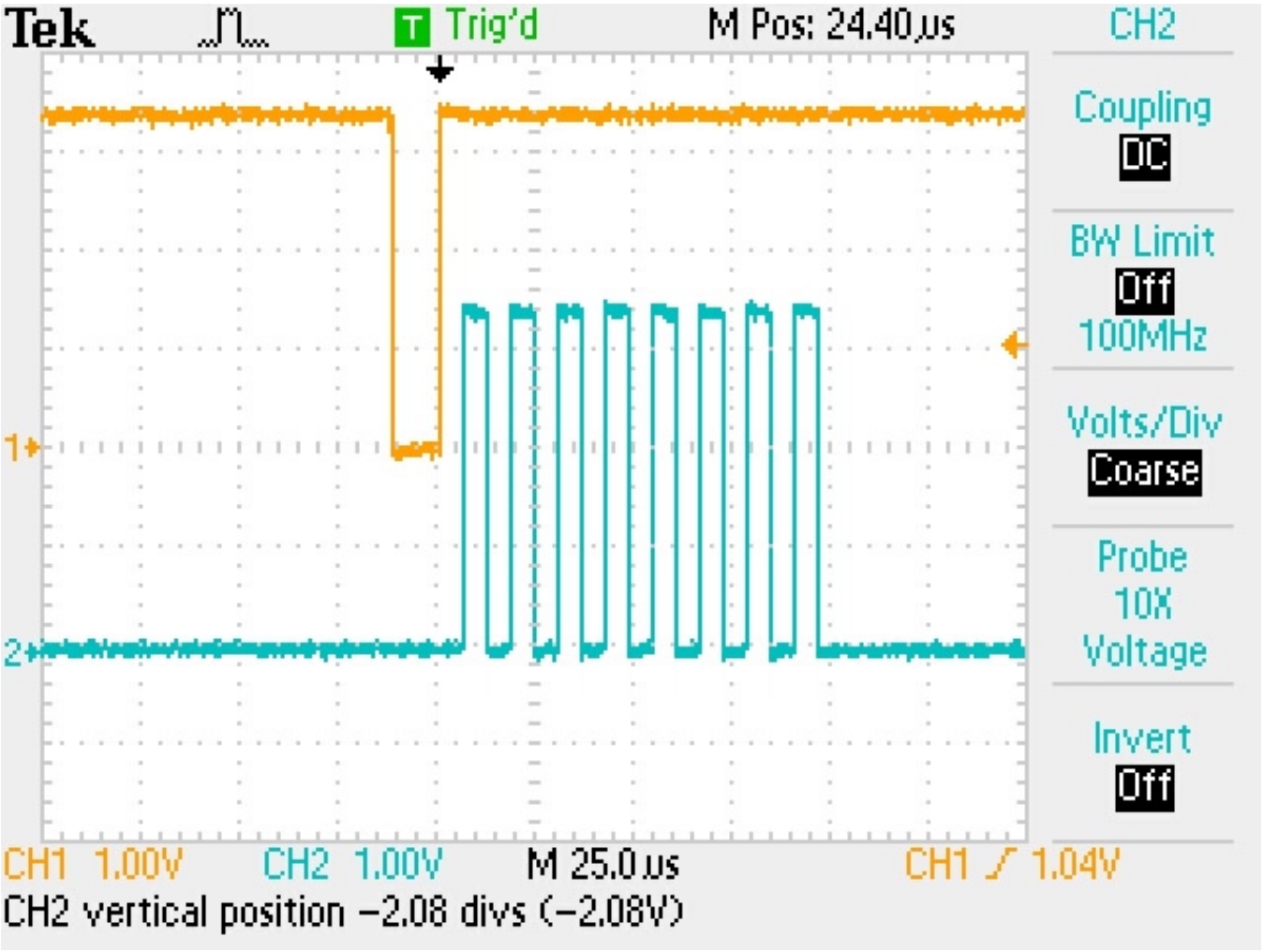


Figure 6: The latch and pulse signals together

5.4.3 Create the `shift` Signal

This signal should be identical to the `pulse` signal, but shifted forward by 3µs. For testing, output it on GPIO[2] and compare it to the `pulse` signal. The two signals together can be seen in the image below.

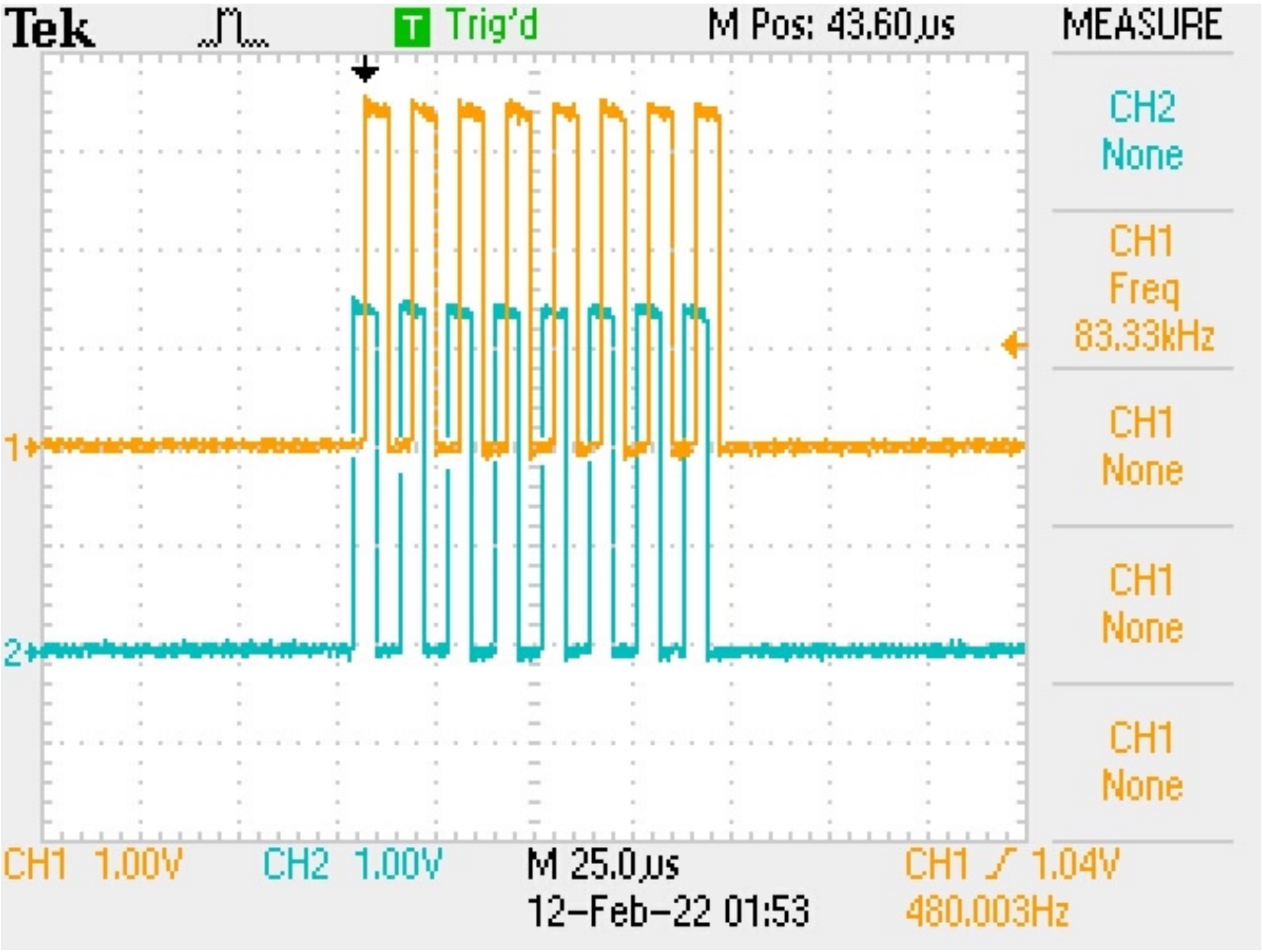


Figure 7: The pulse and shift signals shown together

5.4.4 Buffer the `buttons` Output

Remember that we want to update the `buttons` output only after all the buttons have been received. Otherwise, out `buttons` output would constantly be changing and it would be impossible to determine which buttons are being pressed. To ensure no issues, chose a point sometime after the pulses have stopped to load output of the StP shift register into `buttons`.

You'll know if this is working if your LEDs are either completely on or completely off (I.E. not dimmed when the corresponding button is pressed)

5.5 Test it out!

At this point, your controller interface should be complete. Try testing it out and seeing your button presses turn off the corresponding LEDs!

5.6 Advice

Don't overcomplicate things! This can seem like a monumental task unless you break it into small pieces

ASIC FPGA Prototyping - Random Number Generation

Jared Botte

1 Overview

In this lab, we will be implementing a Linear-Feedback Shift Register (LFSR), which will allow us to create a Pseudo Random Number Generator (PRNG).

2 Setup

Create a new directory and copy over the `makefile`.

```
mkdir random-number-generator
cd random-number-generator
mkdir source
cp $TOOLSDIR/makefile .
```

3 What is a Linear-Feedback Shift Register?

A Linear-Feedback Shift Register (LFSR) is a shift register in which the shifted in bit is determined by bits currently in the shift register. The specific bits in the shift register that are used to determine the shifted in bit are called taps. These taps are combined using XOR operations.

The selection of these taps is critical to create a LFSR with a maximal period (number of bits before repeating its sequence).

There are many videos/articles about LFSRs online. You are encouraged to do some external research and learn about them. Here is a good article, and here is a good video.

4 How do we use it to get a Pseudo Random Number?

To get a random number, we can use the last n bits of the LFSR as our random number. A 24-bit LFSR has a maximal period of 2^24 - 1 = 16,777,215. This means that our random numbers will not repeat in a sequence for over 16 Million cycles. That should be enough for any application we are going to implement.

It is important to note that the output is deterministic meaning that if you know the taps and a value, you can easily determine the next numbers. There are also mathematical ways to reverse the operation and derive the taps given enough values. For this reason, it's not a secure method of generating random numbers.

Lastly, as you probably can figure out, the starting value of our LFSR is very important. Since the order of the random numbers is a simple mathematical function, starting from the same value every time will give you the same random sequence of numbers. For this reason, we must provide a starting number, or seed, to randomize the starting point in the sequence each time. In our case, we can simply use a 24-bit counter which will be connected to our 50MHz clock. This means that we'll cycle through our possible values 3 times per second, and we'll use user input to determine the seed.

5 Implementing an LFSR

To implement a PRNG, we must first create our LFSR. This design will be a combination of the StP and PtS shift registers, as we need the ability to load in a value (the seed) and get the current output value.

Create a module `lfsr` with the following ports:

name	direction	width	description
clk	input	1 bit	clock
n_rst	input	1 bit	active low reset
shift_enable	input	1 bit	signal to enable shifting
load_enable	input	1 bit	signal to load in a seed
seed	input	24 bits	seed input values
value	output	24 bits	current value of the LFSR

Note: If using an XOR LFSR, the design may never be seeded to start with an all 0 value. Doing this will cause the LFSR to be stuck in an infinite loop and never produce anything except zeros.

The taps in this design should be bits 23, 22, 21, and 16. These four bits should be XORed together to generate the next MSB shifted in.

Also note that when using an LFSR, you won't want to use the whole value for your random number, you'll want to use a subset of the generated number to get your value. You'll also want to shift multiple times between values to add some extra randomness.

An example of this is Tetris on the NES. Your next Tetromino (Tetris game piece) is determined by an LFSR, which is shifted each frame. Therefore, the frame at which your piece is placed at the bottom of the board determines what your next piece will be.

6 Generating Pseudo-Random Numbers

Now we'll create a PRNG module that uses the LFSR to give us a random number.

ASIC FPGA Prototyping - UART Receiver

Jared Botte

1 Overview

In this lab, we will be using the UART receiver designed in ECE 337 and the 7-segment displays to interact with a terminal.

2 Required Materials

It will be necessary to have a USB to RS-232 cable to connect your computer to the DE2-115 development board.

I recommend this one for $9 on amazon

3 Setup

To begin, open a terminal window, create a new directory, and move into it. Also, setup the directory structure and copy over the base makefile.

```
mkdir uart-receiver
cd uart-receiver
mkdir source
cp $TOOLSDIR/makefile .
```

In addition to the above setup, copy over all the files from your UART implementation.

4 Alphanumeric 7-Segment Display Module

First, create a new 7-segment display module called `alpha_display` that will display alphanumeric characters on our 7-segment display.

`alpha_display.sv`:

name	direction	width	description
ascii	input	8 bits	the 8-bit ascii value to interpret
disp	output	7 bits	the 7-segment display to output on

Since 7 segment displays have 7 segments which can be on or off, we can theoretically display all 128 characters defined by the original ASCII standard (2^7 = 128). However, many of these characters are unrecognizable (think x, m, and w) and thus wouldn't make much practical sense to implement.

Instead, we'll implement the characters that display nicely, and have a catch-all case to display nothing for all the other characters.

Your design should meet the following requirements:

- Be purely combinational (no clock, no reset)
- Must implement characters 0-9 and A-Z except K, M, V, W, X, Z
- Both the upper and lowercase versions of each character should map to the same result
  - I.E 'B' and 'b' should both map to lowercase 'b' on the display

  - Try to distinguish between letters and numbers (0 and o)
- Add any additional characters you would like

Some resources on 7-segment display characters and ASCII codes:

- wikichip
- pfnicholls

Once completed, be sure to test your design by writing "hello" on the 7 segment displays.

5 Modifications to the UART module

As it currently stands, our UART module requires a baud rate of 400Mb/s. For reference, the highest standardized baud rate is under 1Mb/s. Additionally, we will be using the DE2's 50MHz clock instead of a 400MHz clock. This gives us a new baud rate of 5Mbps, although we want to use something more standard.

You should modify your UART design to have a baud rate of 115200.

5.0.1 Question: What do you need to divide the clock by to get a baud rate of 115200?

6 Setting up RS-232 communications

At this point in the lab it is necessary to setup RS-232 communications so that we can utilize the USB to UART cable. Follow the instructions in the video to get your cable working with your computer.

https://www.youtube.com/watch?v=YNfvNvqnYgM

7 Receiving a single character

Now that our UART module can receive at a standard rate and our USB to UART connection is setup, you should create a module called `single_char_receive` that connects the UART instance with an alphanumeric display.

You should use the following peripherals on the board:

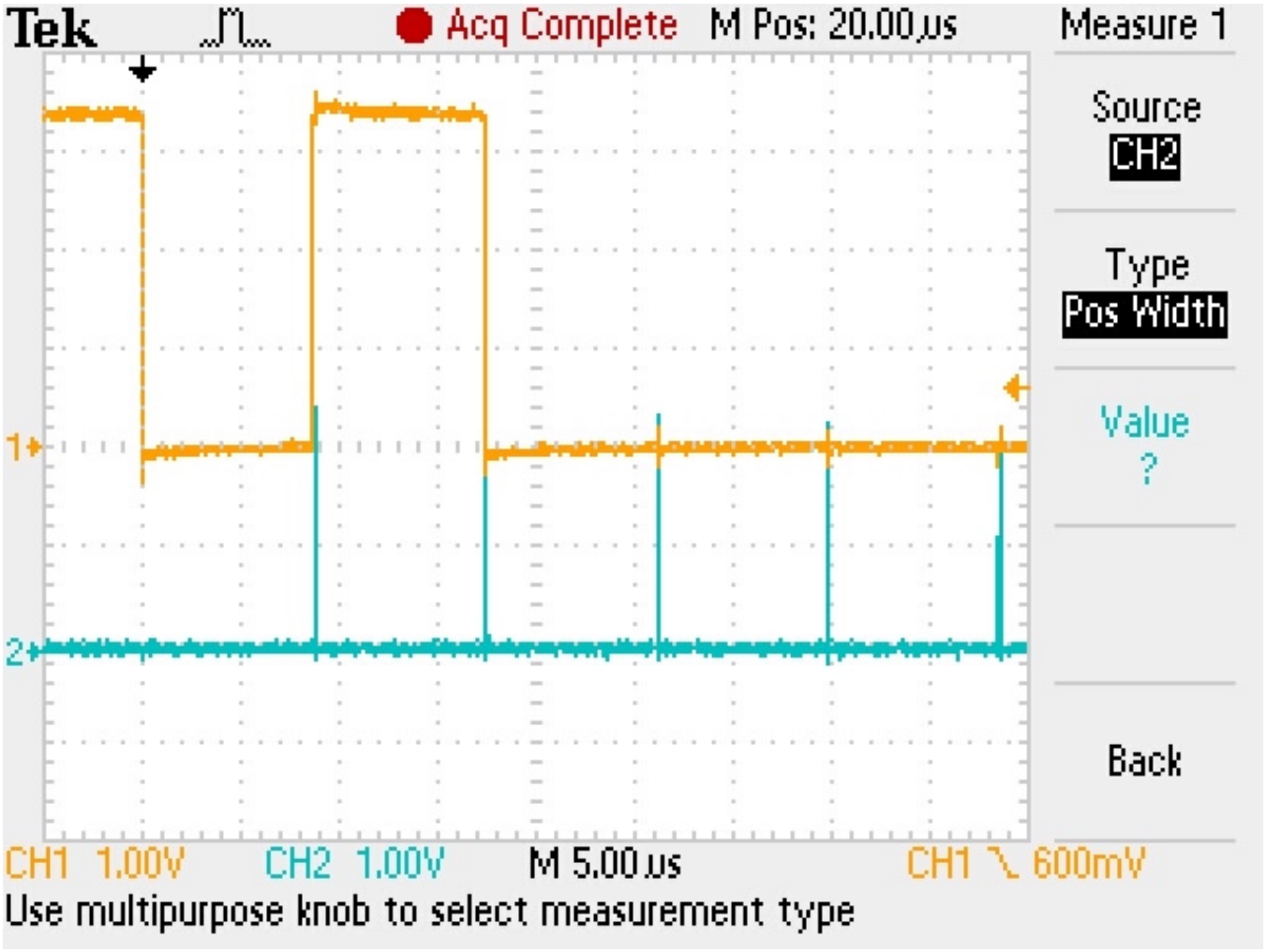
- 50 MHz Clock
- One Key (button) for active low reset
- All four UART signals
  - Pull CTS low
  - Pull TXD high
- 2 red LEDs to indicate framing and overrun errors
- One 7-segment display

Once you've created your module, try sending characters to it!

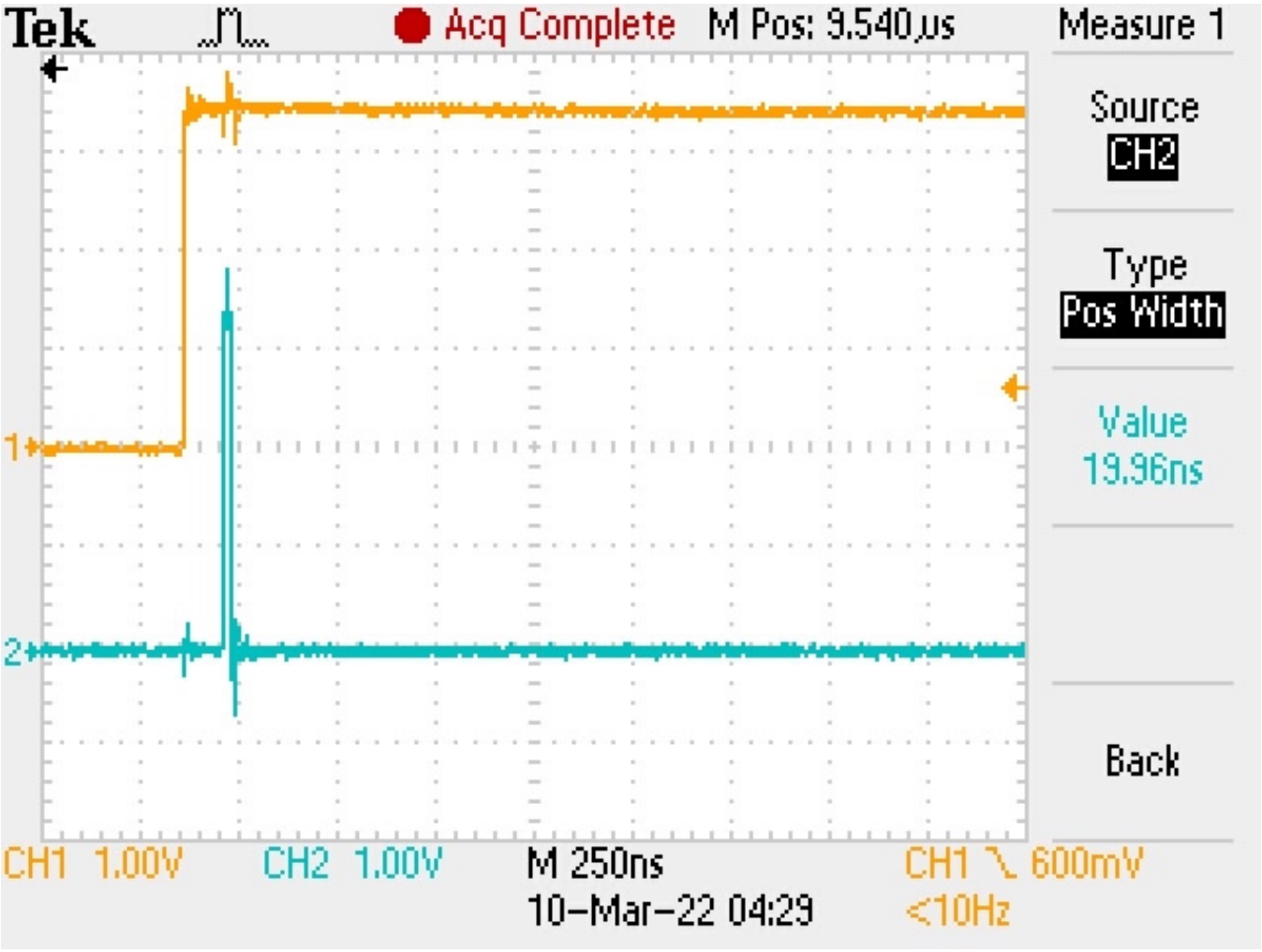
You'll notice that you occasionally get framing errors (actually, quite often). This is because the two clock cycle wait state is not adequate enough for our use. We're measuring as the signals are still settling, whereas we should be aiming to measure in the middle of our bit.

What we currently have:

The signal sampled after a 2-clock wait state

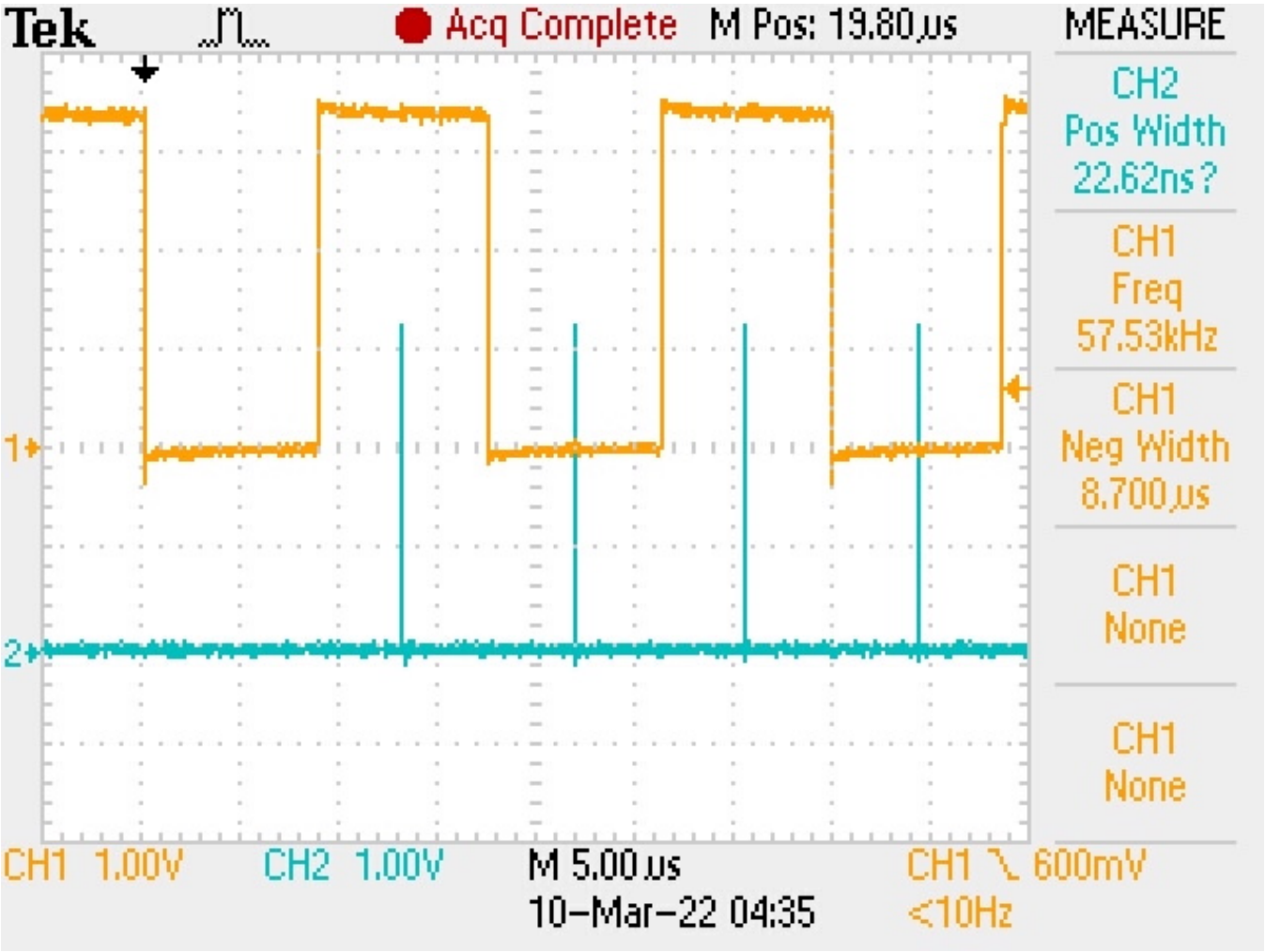


Zoomed in version of early sampling



What we want:

The signal sampled in the middle of the bit



Further modify your UART design to use a timer in the wait state so that the signal samples occur in the middle of the bit.

8 Receiving multiple characters

Now that you've fixed the framing errors and have a single character receive system working. Expand your implementation to keep multiple characters by chaining the 7-segment displays together. Try spelling out words and phrases. Place this implementation in a module named `multi_char_receive`.

9 How to debug when things don't work

The best and really only efficient way to debug issues with your UART signals are to route them to GPIO pins and view them on an oscilloscope. This will allow you to see where your signal is being measured, the UART signal coming from the terminal, and your clock signals that control your UART implementation. It's very similar to viewing the simulated waves, except these waves are real!