Summer 2017 Summary

Zachary Stier

August 2017

Contents

1 VHDL **2**
- 1.1 File layout and VHDL basics . . . 2
- 1.2 Nuances, intricacies and frequent sources of errors . . . 4

2 Quartus II **7**
- 2.1 Files and programs for running Quartus . . . 7
- 2.2 Pin assignments and DE2-115 input/output . . . 7
- 2.3 Temporary upload . . . 8
- 2.4 Permanent upload . . . 8
- 2.5 Quartus miscellany . . . 9

3 Optical alignment **10**
- 3.1 Polarizer angles and the laser . . . 10
- 3.2 Optical component combinations . . . 10
- 3.3 Alignment procedure for when the signal/idler beam is firing directly at the sensor . . . 12
- 3.4 Alignment procedure for when the signal/idler beam reflects before meeting the sensor . . . 12

4 Useful FPGA programs **14**
- 4.1 Rolling rate (`roll.qsf`) . . . 14
- 4.2 Single-output AND gate (`SOAndGate.qsf`) . . . 15
- 4.3 10-second counter (`tensec.qsf`) . . . 15
- 4.4 LCD display example (`lcd_example.qsf`) . . . 15

5 Results **16**

1 VHDL

1.1 File layout and VHDL basics

The VHDL files that I have written all have the following format:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL; —essential libraries

entity designName is
    port (
    inputVariables      : in  std_logic;
    outputVariables     : out std_logic
    );
end designName;

architecture A of designName is
    type : globalVariable;
    type : globalVariable := presetValue;
begin
    process(relevantInputs)
        type : localVariable;
        type : localVariable := presetValue;
    begin
        —your code here
    end process;

    —other processes as necessary
end A;
```

(For some reason LaTeX autocorrects `--` to — in these code snippets, but know that in your text editor you should comment using two hyphens.)

The architecture will be implemented by the software (in our case, Quartus II) as a physical circuit on the chip that will run so long as the board is turned on; as such, it is as if the program is constantly refreshing. Each "refresh" is known as a *delta cycle*, which the compiler deftly navigates and weaves together the instructions for your variables. Each delta cycle is on the order of 5ns, which is why the board can handle input pulses as low as 10ns (which is the length of a pulse from the SPCMs in Room 474).

The first thing to know about VHDL is that it is case *in*sensitive and whitespace-blind; I use capitalization and whitespace formatting only for readability purposes (and out of habit).

When writing VHDL in an environment such as Quartus, the "design name" – i.e., the name of the project – must follow the keyword `entity` and must appear in the other two locations indicated in the above snippet.

The primary data structure classifications are input, output, signal, variable,[1] and constant. When altering a value, inputs and outputs are treated as

[1] Technically `variable` and `shared variable` are distinct, but the latter is global and exclusively used before beginning the architecture and the former is local and exclusively used in a process.

signals, which use `<=`, as in `a <= b;` to set the value of signal `a` to that of `b`. Variables use `:=`, as in `a := b;` for the analogous purpose. Constants, as their name suggests, cannot be altered.

The primary data structures are `std_logic`, `std_logic_vector`, and `integer`. Logic types are either true or false; in VHDL they are represented as `'1'` and `'0'`. It is possible to evaluate logic functions of these structures, such as `and`, `or`, `not` and `xor`.[2]

Here is an example of most of these concepts in action:

```
entity sampleLogic is
    port (
    a,b      : in  std_logic;
    c        : in  std_logic; --you can have inputs on many lines
    f        : out std_logic;
    g        : out std_logic  --same for outputs
    );
end sampleLogic;

architecture A of sampleLogic is
    constant         speed : integer := 299792458;
    signal           s     : std_logic;
    shared variable count : integer := 0;
begin
    process(a,b,c)
        variable localSig : std_logic;
    begin
        localSig := a or b;
        f <= localSig and not c; --goes to the output pin for f
        s <= localSig and not c; --this (and f) is just (a|b).(!c)
    end process;

    process(s) --you can use a variable to control a process!
    begin
        if (rising_edge(s)) then
            count := count + 1;
            if (count = 5) then
                count := 0;
            end if;

            if (count = 0) then
                g <= '1';
            elsif (count = 2) then
                g <= '1';
            else
                g <= '0';
            end if; --this if block sets g high if f has been sent
    high 0 or 2 times mod 5
        end if;
    end process;
end A;
```

Inputs and outputs may also be `std_logic_vectors`; this can make the

[2] Sometimes the Quartus VHDL compiler won't let me use these logic functions, in those cases the best solution is often to rewrite using explicit values, i.e. `(a or b) and not c` becomes `((a = '1') or (b = '1')) and (c = '0')`.

code cleaner and requires no extra work when doing the pin assignments in Quartus. Logic operations on two (identically-sized) vectors will perform the operation done bitwise. Consider the following snippet for an example of vector declaration and slicing:

```
    process
        variable vec   : std_logic_vector(4 downto 0) := "10110";
        variable vec31 : std_logic_vector(2 downto 0);
    begin
        vec31 := vec(3 downto 1); --vec31 = "011"
    end process;
```

1.2 Nuances, intricacies and frequent sources of errors

One simple error that can cause trouble is that the final output in the entity port *does not have a semicolon after* `std_logic`: the semicolon is included in the `);` on the next line. The compiler will catch this mistake, but knowing about it in advance can be a time-saver.

One common error that Quartus' compiler throws and which causes compilation to fail has to do with "resolving multiple drivers." This occurs when a value is altered in more than one process. For example,

```
entity driverFail is
    port (
    a,b     : in  std_logic;
    f       : out std_logic;
    );
end driverFail;

architecture A of driverFail is
    shared variable count : integer := 0;
begin
    process(a)
    begin
        if (rising_edge(a)) then
            count := count+1;
        end if;
    end process;

    process(b)
    begin
        if (rising_edge(b)) then
            count := count+1;
        end if;
    end process;

    process(a,b)
    begin
        if (count >= 5) then
            count := count-5;
        end if;
        if (count = 0) then
            f <= '1';
        else
            f <= '0';
```

```
        end if;
    end process;
end A;
```

will fail to compile, since `count` is altered in three processes. However,

```
architecture A of driverWin is
    shared variable count : integer := 0;
    signal          s     : std_logic;
begin
    process(a,b)
    begin
        s <= a or b;
    end process;

    process(a,b)
    begin
        if (rising_edge(s)) then
            if (a = '1' and b = '1') then
                count := count+2;
            else
                count := count+1;
            end if;

            if (count >= 5) then
                count := count-5;
            end if;

            if (count = 0) then
                f <= '1';
            else
                f <= '0';
            end if;
        end if;
    end process;
end A;
```

accomplishes the desired task, and in a single process.

Oddly enough, VHDL will not compile even if a variable is modified in a "safe" manner in multiple processes. For example,

```
    process(a)
    begin
        if (a = '1') then
            f <= '1';
        end if;
    end process;

    process(a,b)
    begin
        if (a = '0' and b = '1') then
            f <= '1';
        elsif (a = '0' and b = '0') then
            f <= '0';
        end if;
    end process;
```

does not compile despite each of the conditions to modify `f` being mutually exclusive.

VHDL is also very picky about clocks. For example,

```
architecture A of clockFail1 is
    shared variable count : integer := 0;
begin
    process(a,b)
    begin
        if (rising_edge(a)) then
            count := count+1;
        end if;
        if (rising_edge(b)) then
            count := count+1;
        end if;
    end process;

    process(a,b)
    begin
        if (count >= 5) then
            count := count-5;
        end if;
        if (count = 0) then
            f <= '1';
        else
            f <= '0';
        end if;
    end process;
end A;
```

fails because VHDL will not let a single process' action depend on the state of two different clocks. (This issue is resolved by the `driverWin` approach.)

```
architecture A of clockFail2 is
    shared variable count : integer := 0;
begin
    process(a,b)
    begin
        if (rising_edge(a)) then
            f <= '1';
        elsif (b = '1') then
            f <= '0';
        end if;
    end process;
end A;
```

This fails to compile because VHDL will not let a variable "hold its value outside the clock edge" – i.e., it is unhappy with an `elsif` on the heels of an `if rising_edge`.

There is of course more powerful functionality to be had with VHDL, such as incorporating many files into a single Quartus design or using port maps for more complicated individual files, but I managed to accomplish everything I needed to this summer with just the means outlined here.

2 Quartus II

In this section I will explain how to do some essential tasks in Quartus and will give some pointers that hopefully will save some time. (Here "file_name" is the placeholder name for the pertinent file in the given situation.)

2.1 Files and programs for running Quartus

I had substantial trouble installing any version of Quartus II, and was entirely unsuccessful with downloading it. Unfortunately, the Windows version is a 64 bit EXE file and Wine for Unix systems is a 32 bit Windows emulator; in addition, for some reason the installer was uncooperative with the Dell machine in Room 380. Version 10.0 is currently installed on that computer, as well as the Dell laptop in Room 474.[3] If you would like to install it on another machine, I found success with the CD that came in the DE2-115 box. You will also need to install the USB Blaster driver (this is already installed on the two aforementioned machines); click here for directions. (For Windows 10 the steps are slightly different but are virtually identical to the first set of directions.)

2.2 Pin assignments and DE2-115 input/output

The theory behind programming the board is that each (`std_logic`[4]) input/output variable in your VHDL file corresponds uniquely to a physical location on the DE2-115. In order for Quartus to teach the chip how to route that information, the chip contains a grid of *pins*, each of which corresponds uniquely to a location on the board; by identifying each variable to a pin, that variable gets mapped directly to the physical position.

I: The 18 **switches** on the bottom of the board

I: The internal **50MHz clock**

I/O: The 22 **I/O pins** (6 on the left and 16 on the right of the board). The pin arrays have 2 columns; the left column is entirely ground, except for the bottom-rightmost pin in the left array.[5] The I/O pins are each meant to receive and output 3.3V.

O: **LEDs** above the row of switches

O: **7-segment displays**. Each segment is controlled by a different pin, making their pin planning cumbersome.

O: **LCD display**, whose control requires dark magic (see `lcd_example.vhd`)

[3] To log on to that computer, the PIN is 1746.

[4] This distinction is made only because it is possible to read in a `std_logic_vector` as input to a VHDL file.

[5] I actually couldn't decipher what this pin is supposed to do, but it briefly shut the board down when I accidentally touched a free wire to it.

O: **RS-232 port**, whose use I did not have a chance to fully explore.

To use the input/output components, click here to consult the DE2-115 User Manual. The pages of interest are Chapter 4, pages 29-69. In Quartus, enter the Pin Planner and for each variable set the pin number of the appropriate part (e.g. if the variable `a` should represent the state of the rightmost switch, SW0, in the Pin Planner set `a`'s pin to PIN_AB28).

2.3 Temporary upload

By default, the RUN/PROG switch on the board should be set to RUN (as it should be for the present purposes).

1. Make sure the design has successfully been compiled (including with a complete pin assignment).
2. Open the Programmer window (third button from the right on the icon bar in Quartus). (The Dell laptop in Room 474 often gives an error popup when opening the Programmer; clicking "Okay" usually makes it disappear and does not stop one from continuing.)
3. Make sure the Mode is set to JTAG (this is the default unless you had saved the .cdf[6] as Active Serial previously).
4. If file_name.sof does not appear in the main portion of the window, add it using "Add File" and check the box under "Program/Configure." If "Hardware Setup..." does not read "USB-Blaster" then make sure the board is turned on and plugged into your computer before clicking on "Hardware Setup...", double-clicking the USB Blaster in the new window, and clicking "Close."
5. Click "Start;" the upload should complete within 10 seconds.

Do not turn off the board for as long as you want to use the design just uploaded; it will remain until a new one is uploaded or the board loses power.

2.4 Permanent upload

1. Make sure the design has successfully been compiled (including with a complete pin assignment).
2. Go to "File > Convert Programming Files..." For the DE2-115, in the Configuration Device menu I have found that EPCS64 and EPCS128 work, but a different specification may be necessary for a different board (click here for more information about EPCS and Altera boards). Click on the text in the largest box in the window and select "Add File..." on the right; choose file_name.sof and click "Generate." Once that completes (it should be almost instantaneous) you can close the window.

[6]The .cdf file is the Programmer setup file that Quartus will load as the default when you open the Programmer each time.

3. Usually the RUN/PROG switch on the board is set to RUN; flip it to PROG.

4. Open the Programmer window.

5. Make sure the Mode is set to Active Serial.

6. If file_name.pof does not appear in the main portion of the window, add it using "Add File" and check the box under "Program/Configure." If "Hardware Setup..." does not read "USB-Blaster" then make sure the board is turned on and plugged into your computer before clicking on "Hardware Setup...", double-clicking the USB Blaster in the new window, and clicking "Close."

7. Click "Start;" in my experience the upload typically takes between two and five minutes to complete.

8. Though probably unnecessary, to be sure that the upload was successful I always switch the RUN/PROG switch back to RUN, turn the board off, and turn it back on again.

The design just uploaded is now the default for when the board starts up.

2.5 Quartus miscellany

Since all the tasks we need the FPGA to perform have reasonably large margins of error, there are lots of built-in precision tools that Quartus wants to use that we just don't need. This includes .sdc and Timing Analysis files that Quartus mentions in the Critical Warning tab upon compilation; I have been fine just ignoring these complaints.

3 Optical alignment

When I arrived in June I found the optical breadboard partially set up for the Grangier experiment. The portion that sends the laser beam through a polarizer and the BBO crystal was already aligned, so I did not touch that end of the table all that much; I did, however, manage to find ways to calibrate the other end, outlined below.

3.1 Polarizer angles and the laser

The laser beam passes through a polarizer and the BBO crystal before reaching the detectors. The relative angle between those polarizers determines the angle of the signal/idler beams above the table via the formula

$$\frac{\cos^2\theta_m}{n_o^2} + \frac{\sin^2\theta_m}{n_e^2} = \frac{1}{n_o^2 - \sin^2\theta_L}, \tag{1}$$

where θ_L is the "lab angle," θ_m is the relative rotation of the polarizers, $n_o \approx 1.66$ and $n_e \approx 1.57$.[7] When solving this for specific values of θ_m or θ_L I found Wolfram|Alpha to be sufficient.

Unfortunately, in the setup on Room 474's optical breadboard, the reference angles for the two polarizers are different, so it appears to be impossible to precisely measure the rotational difference. All data collected was with the empirically accurate rotations of the polarizer set to its reference mark of 0° and the BBO rotated so that it reads 322°.

One other thing to keep in mind is that the laser strength decays at an approximately linear rate (~10%/hour), but I did not have a chance to let it run long enough to determine if it would stabilize at a nonzero power or not.

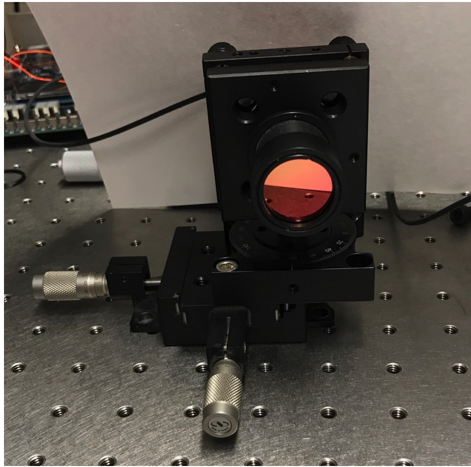
3.2 Optical component combinations

In order to mount the sensors so that they are coplanar with the BBO and laser beam, and so that that plane is parallel to the tabletop, there are three primary combinations of parts that can be used.

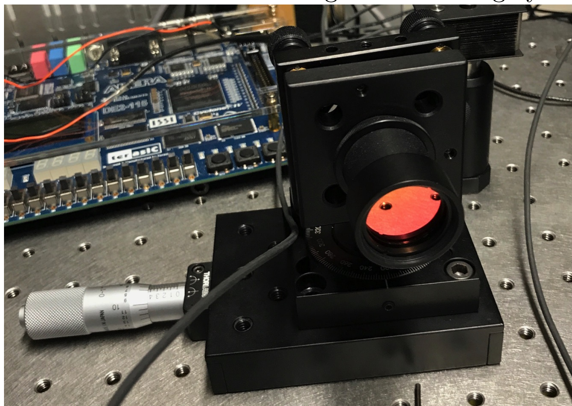
1. 2D trans. mount + rot. stage + custom cylinder[8] + sensor mount

[7] n_o and n_e are calculated for $\lambda = 810$ nm and 405 nm, respectively, using (B5) in Galvez et al.; (1) is derived from (B3) and (B4), their statement of Snell's law as $\sin\theta_L = n_s \sin\theta_c$, setting $n_s = n_o$, their statement that $\widetilde{n}(\theta_m) = n_o \cos\theta_c$, and algebraic manipulation (squaring and summing) to eliminate θ_c.

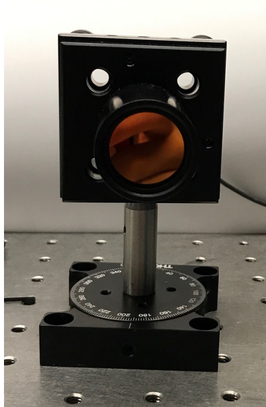
[8] The custom cylinder was built in the shop and is recognizable via its lack of holes on its curved surface. It is 0.5" tall and has a 0.5" diameter.



2. 1D trans. mount + rot. stage + 0.75"-long cylinder + sensor mount



3. rot. stage + 1.5"-long cylinder + sensor mount



3.3 Alignment procedure for when the signal/idler beam is firing directly at the sensor

1. Make sure that all devices are turned off.
2. Position the sensor/mount where you would like it to be on the optical breadboard.[9]
3. Unscrew the bandpass filter from the sensor.
4. Unscrew the optical cable from the SPCM and screw that end into into the red handheld laser.
5. Rotate the stage until the laser fires near the crystal. (Ideally at least a sliver of the beam will hit the crystal.)
6. Fix the rotating stage.
7. Use the screws on the back of the sensor's mount to nudge the beam onto the crystal. The laser should now fire dead-center right through the crystal.
8. Return the bandpass filter to its place on the sensor. Unscrew the optical cable from the handheld laser and return it to the SPCM.
9. Make sure the FPGA is set to the 1-second reset program (`roll.qsf`; this is the current default file as of 8/11). Turn off the lights, turn on the power supply to the laser and sensors, and rotate the crystal until you notice the rate peaking. (The laser output will most likely be decaying during this time, but the variance is high enough that there's a few degrees of error anyway.)

3.4 Alignment procedure for when the signal/idler beam reflects before meeting the sensor

1. Make sure that all devices are turned off.
2. Position the sensor/mount approximately where you would like it to be on the optical breadboard.[10] (The sensor will end up at most a millimeter or two from where you initially place it.)
3. Position the optically-mounted laser behind the BBO crystal, and adjust it until it fires through the crystal along the path that the signal/idler beam will take.[11]

[9] This method only requires the mount to have a rotating stage.

[10] This method requires the mount to have both a rotating stage as well as one dimension of translation.

[11] Once this is aligned, I usually clamp down the laser's mount's base, but this is probably optional.

4. Position the beam splitter along the signal/idler path, approximately at the foot of the perpendicular from the sensor onto that line, and rotate it until the main deflected beam appears near the center of the bandpass filter.

5. Unscrew the bandpass filter from the sensor. Adjust the translational stage until the deflected beam hits the sensor. (The sensor will be red if and only if this is happening, since the handheld laser should be turned off.)

6. Turn off the laser behind the BBO and turn on the handheld laser.

7. Rotate the stage until the laser fires near the crystal. (Ideally at least a sliver of the beam will hit the crystal.)

8. Fix the rotating stage.

9. Use the screws on the back of the sensor's mount to nudge the beam onto the crystal. The laser should now fire dead-center right through the crystal.

10. Turn off the handheld laser and turn the laser behind the crystal, just to make sure that is still aligned as well. It might be slightly off, so in that case you may need to repeat these steps (but the adjustments will be much slighter).

11. Turn off both lasers. Return the bandpass filter to its place on the sensor. Unscrew the optical cable from the handheld laser and return it to the SPCM.

12. Make sure the FPGA is set to the 1-second reset program (`roll.qsf`; this is the current default file as of 8/11). Turn off the lights, turn on the power supply to the laser and sensors, and rotate the crystal until you notice the rate peaking. (The laser output will most likely be decaying during this time, but the variance is high enough that there's a few degrees of error anyway.)

4 Useful FPGA programs

This section contains the function of each part of the board for some of the FPGA programs that I've written; hopefully this will be a timesaver (so you don't have to spend as much time parsing the code). Each of these is in a folder on the desktop of the laptop in Room 474.

4.1 Rolling rate (`roll.qsf`)

Inputs:

A: EX_IO[6]

B: EX_IO[5]

C: EX_IO[4]

use A? SW[17]

use B? SW[16]

use C? SW[15]

flip A? SW[4]

flip B? SW[3]

flip C? SW[2]

Reset: SW[1]

Pause: SW[0]

Formally, the program counts instances of `(useA and (A xor flipA)) and (useB and (B xor flipB)) and (useC and (C xor flipC))` but it's better to think of it as the "use" switches determining which variables to include, and then the "flip" switches negating the corresponding variable if that switch is high. So, (useA,useB,useC,flipA,flipB,flipC)=(1,0,1,0,0,1) will cause the board to count instances of the function `A.(!C)`.

Outputs:

count: 7-segment display

pulse: EX_IO[1]

The output pulse fires whenever the board detects a coincidence. The 7-segment display resets every second and displays the number of coincidences it detected in the previous second.

4.2 Single-output AND gate (SOAndGate.qsf)

Inputs: identical to roll.qsf

Outputs: identical to roll.qsf

The 7-segment display updates in real time and does not reset.

4.3 10-second counter (tensec.qsf)

Inputs: identical to roll.qsf

Outputs: identical to roll.qsf

The 7-segment display updates in real time and pauses after exactly 10 seconds.

4.4 LCD display example (lcd_example.qsf)

Inputs: none

Outputs: counts seconds on the LCD display

This program is perhaps most useful as a template for programs to incorporate use of the LCD display; the best explanation I can offer for how to control that display is to examine lcd_example.vhd.

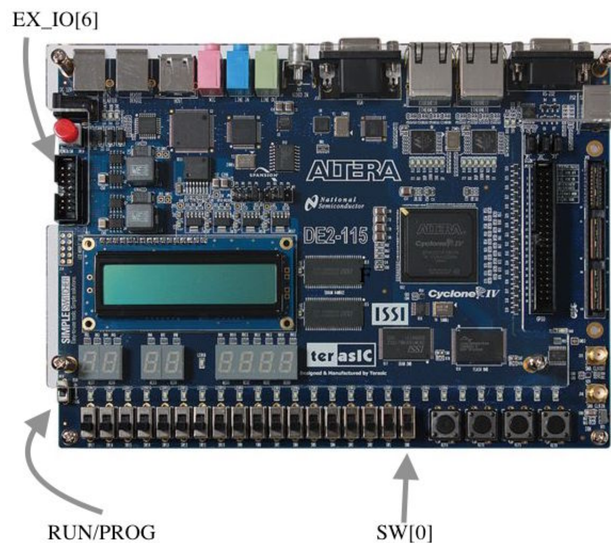


Figure 1: Some useful reference locations on the board.

5 Results

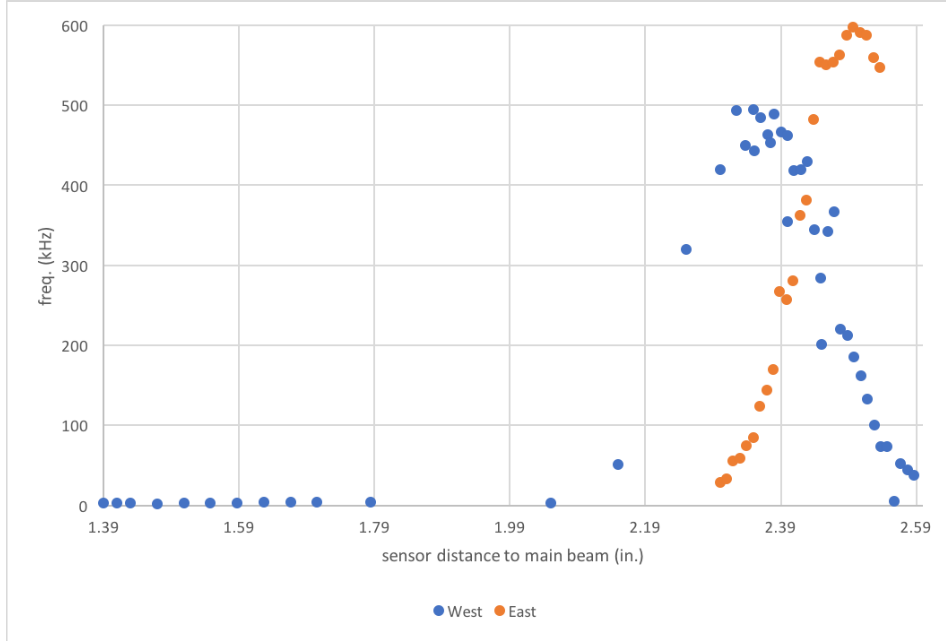


Figure 2: Graph of how single-count frequency changes in distance to the main beam. "East" and "West" refer to the sides of the table (the door is on the Eastern half of the room). Each sensor's mount's southernmost screw was placed 34" down the table from the BBO. The maximal values were reached with sensor A on the 1D stage set to 1 with its closest screw to the beam 2" away, and sensor B on the 2D stage set to 5 with its screws 5" away, getting 495 and 598 kHz, respectively;* the coincidence rate was approximately 60 kHz.

These values are the raw FPGA outputs – they do not account for dark counts or SPCM efficiency. The formula for computing the exact value is

$$R \approx 2(R' - D),$$

where R is the actual photon rate, R' is the value measured using the FPGA, and D is the individual SPCM's dark count rate, which is between 1.4 and 2 kHz for each SPCM in Room 474, so when the setup is properly aligned D is negligible. The factor of two comes from dividing by SPCM efficiency, which is approximately 50% on 810 nm photons.[12] Noise is also negligible for coincidence counts; using resolving time of 21 ± 2 ns, dark counts should account for up to 60 Hz of the measured coincidences, 0.1% of the value measured by the FPGA.

[12] Click here for related manufacturer information.

y	R_A	R_B
1.39	3.2	
1.41	3.2	
1.43	3.2	
1.47	1.9	
1.51	3.5	
1.55	3.4	
1.59	3.4	
1.63	3.6	
1.67	3.6	
1.70	3.7	
1.78	4.4	
2.05	2.5	
2.15	51	
2.25	320	
2.30	420	29
2.31		33
2.32		56
2.33	493	59
2.34	450	75
2.35	495	85
2.36	485	124
2.37	463	144
2.38	489	170
2.39	467	267
2.40	462	257
2.41	418	281
2.42	420	362
2.43	430	382
2.44	345	482
2.45	284	554
2.46	342	551
2.47	367	554
2.48	220	563
2.49	212	587
2.50	185	598
2.51	162	591
2.52	133	588
2.53	101	560
2.54	74	547
2.55	74	
2.56	5.6	
2.57	52	
2.58	44	
2.59	38	

Figure 3: Source data for Figure 1. y represents distance from the sensor to the main beam, and R_A and R_B are the rates detected by each sensor, measured in kHz.

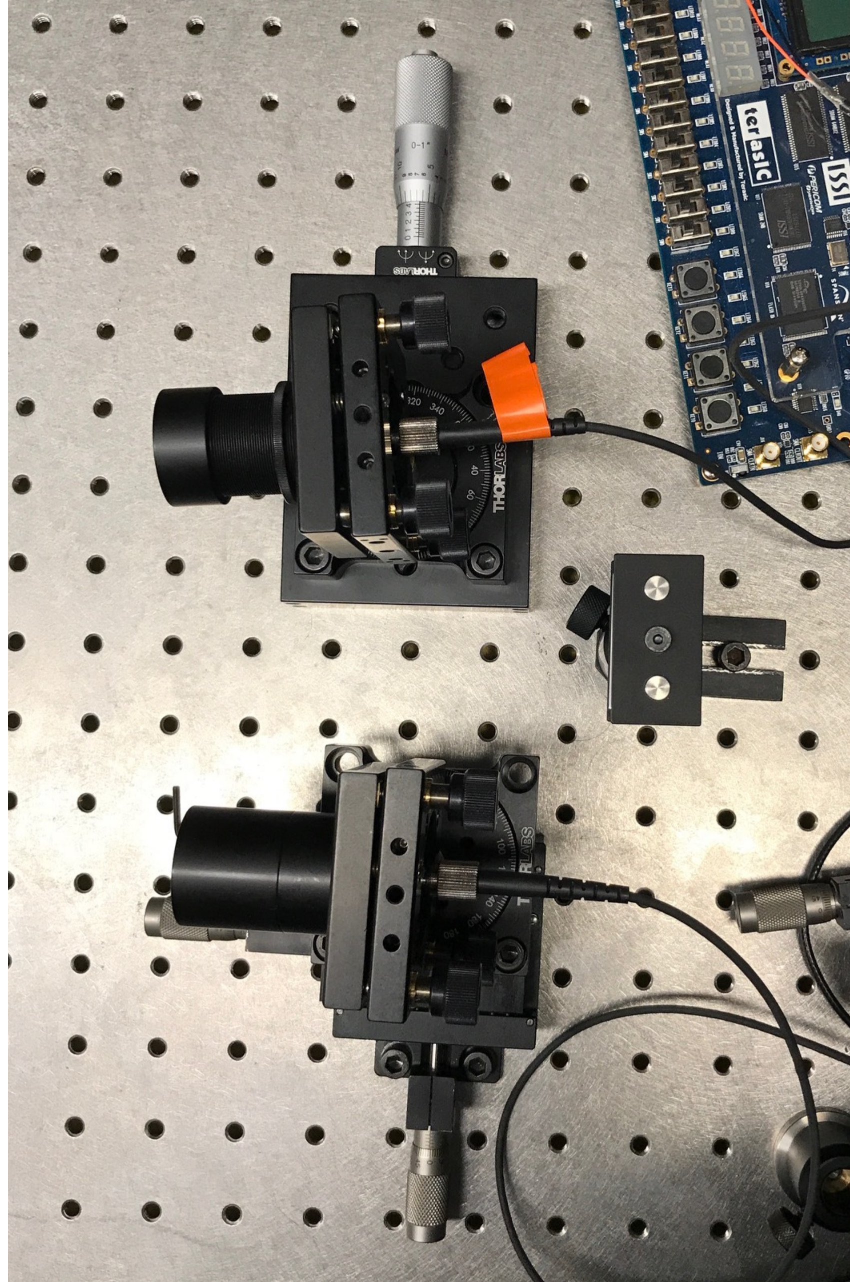


Figure 4: Sensor-end of the board that yielded successful coincidence detection. (Translating stages might require slight modification from this exact figure.)