Step Impediment Detection for Visually Impared Persons (SID VIP)

Designed and Developed by: (Left to right)
Austin
Joshua Kulwicki
Michael Muller
Niranjan

I. *Table of contents*

II. *Abstract*

Over the past 20 years, technology has advanced at a substantial pace. The availability, low cost of, and ease of access to hardware provides us the opportunity to expand technology beyond convenience. We set out to bring the speed, precision, and low cost of sensors and feedback modules to visually impaired persons, who rely on their own physical capabilities to navigate daily life. Currently, the market of products that assist this demographic is limited and expensive. A similar system using ultrasonic sensors which is fitted to a shoe will cost around $4,000 for the finished product. SID will not replace the cane, but rather provide additional confidence and comfortability when using a cane through audible and haptic feedback warnings. This goal doesn't come without its drawbacks. Our FGPA is only able to provide 3.3v which limits the quality of the ultrasonic sensor we can use. In addition to sensor quality, each person has a unique stride and step which cannot be accounted for in SID's current state. However, we've tested multiple different thresholds of detection and have landed on an average detection range that should feel natural in general.

III. *Introduction*

Our system needed to be designed to accurately and quickly identify objects that may appear while walking. Once an object has been detected, we should provide physical feedback to the user that allows them to identify the distance of the object and react accordingly. As stride length varies depending on the user, we either need to provide varying thresholds to activate on, or use an average that will be adequate in general. Since our FPGA can only provide 3.3v, we needed to use a lower end ultrasonic sensor that could function. The sensor selected is only able to take 16 measurements per second, which accounts for timeouts if the sonic pulses do not return. We were able to use this slow measurement time to perform calculations and comparisons at a much higher clock speed during the timeout time provided by the sensor.

In the following sections we will take a look at the top level design to provide an overview of the inputs and outputs utilized in each component and how the system should function at a high level. Our system consists of four individual design components, the *Ultrasonic*, *Display, UART, and Servo* components. We will dive into each of these component's designs and break down where they fall in the overall architecture and how they are driven by the Ultrasonic component controller. After defining the components, we will discuss the system performance and any hurdles overcome to achieve its current level of performance. Future improvements to our project will be covered in our conclusion as well as a short summary of our findings throughout this project.

IV. *Top-level Design*

Given the nature of the system design, the entire functionality of SID is determined by the pulse time provided through the ultrasonic sensor. It was decided early on that this component would control any subsequent components that relied on information from the sensor or calculations thereof. The Ultrasonic Top component functions as the "brain" of the system. As for the feedback modules, initially the servo motor was part of the handheld device. However, seeing as how SID was meant to be applied to a shoe we decided to move the servo to the FPGA. Therefore haptic feedback would be applied to the shoe that is detecting an object, giving the user more information about their surroundings.

The haptic feedback from the system is only applied to objects classified as *near* or *close*. Vibrations are much easier to react to than audible chimes, as a vibration can be felt. Provided that an object is far away and not an immediate concern, we merely want to inform the user of a potential obstruction via a chime and no vibration. Once the object is classified as an impediment, we will apply both haptic and acoustic feedback to the user. The handheld device will communicate through Wifi with the on board module.

Displaying the distance to an object on the 7-segment display was added to confirm the functionality of the object classification at the specified thresholds. These thresholds had been decided based on research of average stride length for both male and female Americans and precision from our calculations. Average stride length, 2.2 ft and 2.5 ft for females and males respectively was used as our maximum threshold where we detect an object as far. We extended this to around 80 cm to allow for a full step to be taken before the user would collide with the object. We then use 75% and 50% of the max stride length, in centimeters, to classify near and close objects respectively. At which point, both haptic and acoustic feedback is expected. *(References [A.1][A.2])*

For the UART communication to the on board wifi module, a baud rate of 9600 was selected arbitrarily. Since only a single byte is being sent to the module and the measurement rate is 16 times per second, we have plenty of time to transmit a full byte and then expect the wifi module to send the UDP packet to the handheld device. The UDP protocol was selected to ensure quick wifi transmission for acoustic feedback.

V. *VHDL Design*

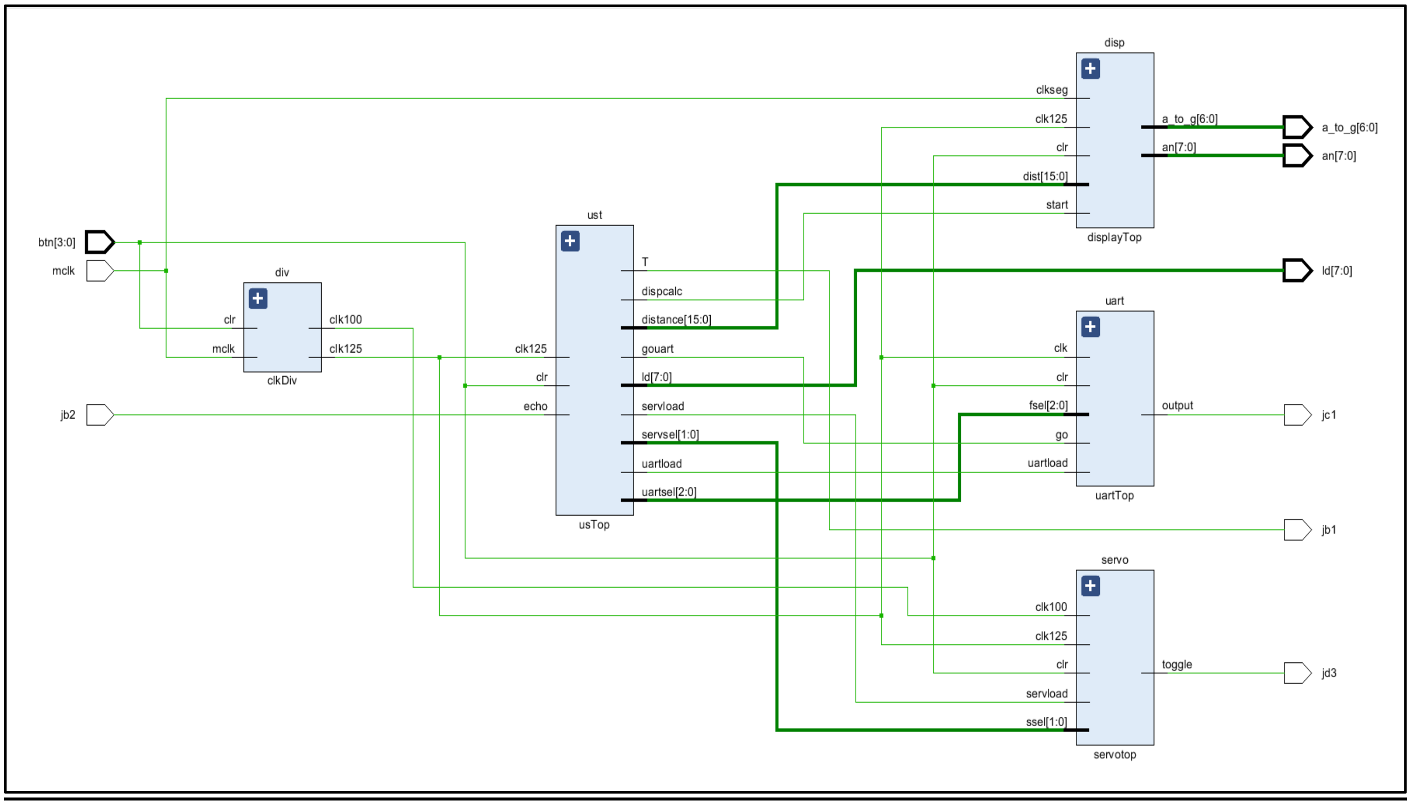


Figure 1: SID Top-level design

As described in the previous section, the UST functions as the “brain” of the system, providing the relevant information needed by each other component, as well as any conditions for those components to begin functioning. A clock frequency of 12.5 Mhz was used to drive the ultrasonic sensor and allow for easier bit manipulation for calculations. As measurements are taken every 60 ms, we have more than enough cycles to complete any feedback requests and distance calculations before the next measurement.

The only system input is through PMOD pin Jb2 which is connected to the ultrasonic sensor echo pin. Otherwise, this system outputs requests to modules to carry out any feedback or data transmission.

Ultrasonic Component

The Ultrasonic component is used to drive the ultrasonic sensor, perform a distance calculation, and classify object distance. After these steps are performed, the US controller will load the subsequent component registers and initiate the “go” command to each component minus the servo module.

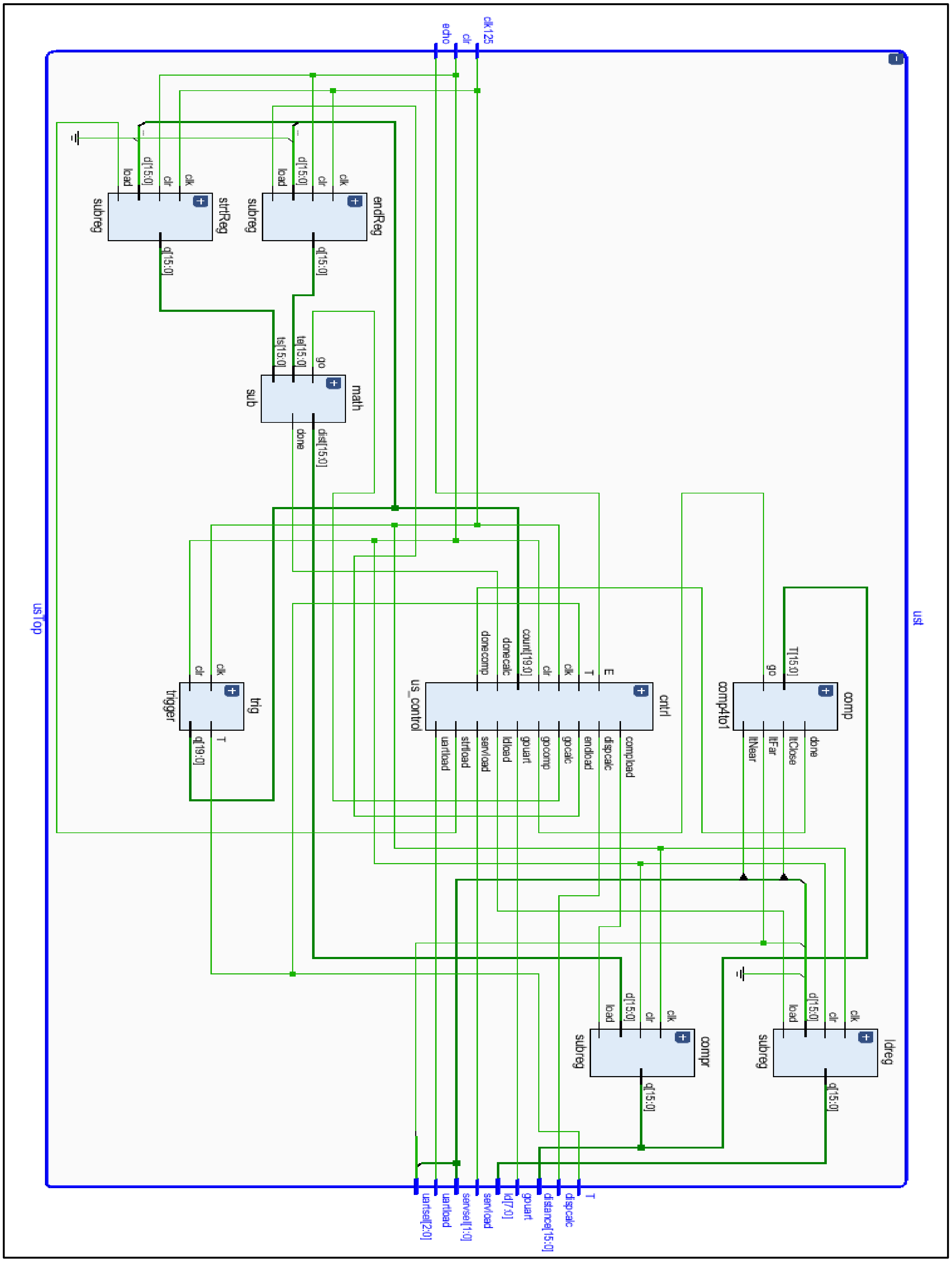


Figure 2: US Top component

The Ultrasonic component uses a counter to drive the trigger pin of the connected sensor. As provided by the sensor data sheet, the trigger pin needs to be pulled high for at least 10 microseconds. At which point, the sensor will send out 8 pulses at 40 Khz, once all pulses have been sent out, the echo pin will go high. Once the pulses have been received by the module or a timeout occurs, the echo pin will go low. The time between the echo pin going high and low is used to determine distance to an object. We use the following base equation to determine distance in centimeters, *distance* = [time(us)/343(m/s)]/2 *(Reference [B.1])*

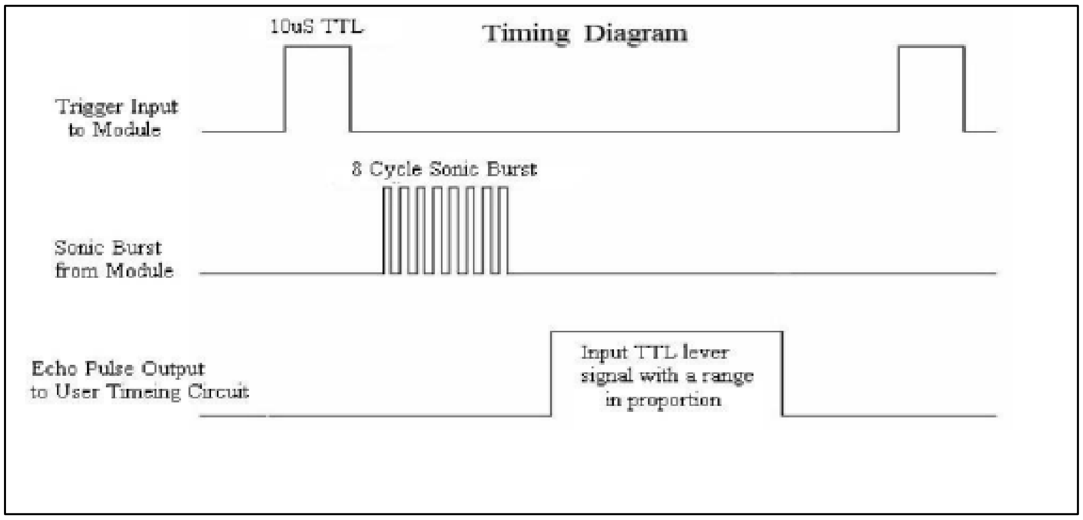

10uS TTL
Timing Diagram
Trigger Input to Module
8 Cycle Sonic Burst
Sonic Burst from Module
Echo Pulse Output to User Timeing Circuit
Input TTL lever signal with a range in proportion

Figure 3: HC-SR04 Timing Diagram [B.1]

To pull the trigger pin high for 10 us and ensure a max measurement time of no less 60 ms at 12.5 Mhz, we use a trigger pin count of 128 for the duty cycle count, and then pull the pin low for the rest of the 750,000 trigger count. The time value recorded from the sensor is stored in 10 us buckets to reduce signal size. We will discuss lossy precision for the duty cycle and time count in the performance section. The following state machine and data path describe the behavior of the ultrasonic sensor calculations. *(Appendix A.1)*

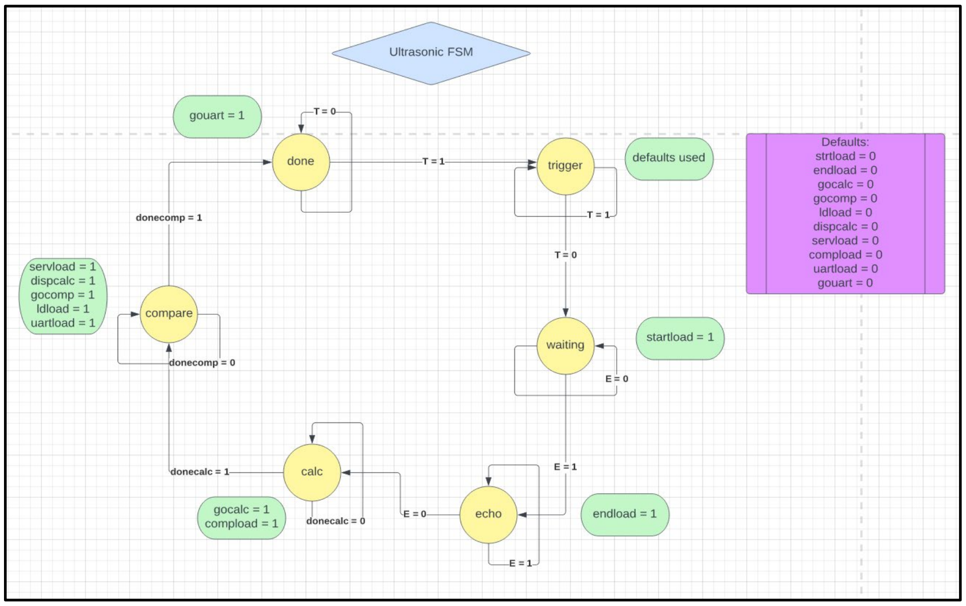


Figure 6: UST State Machine

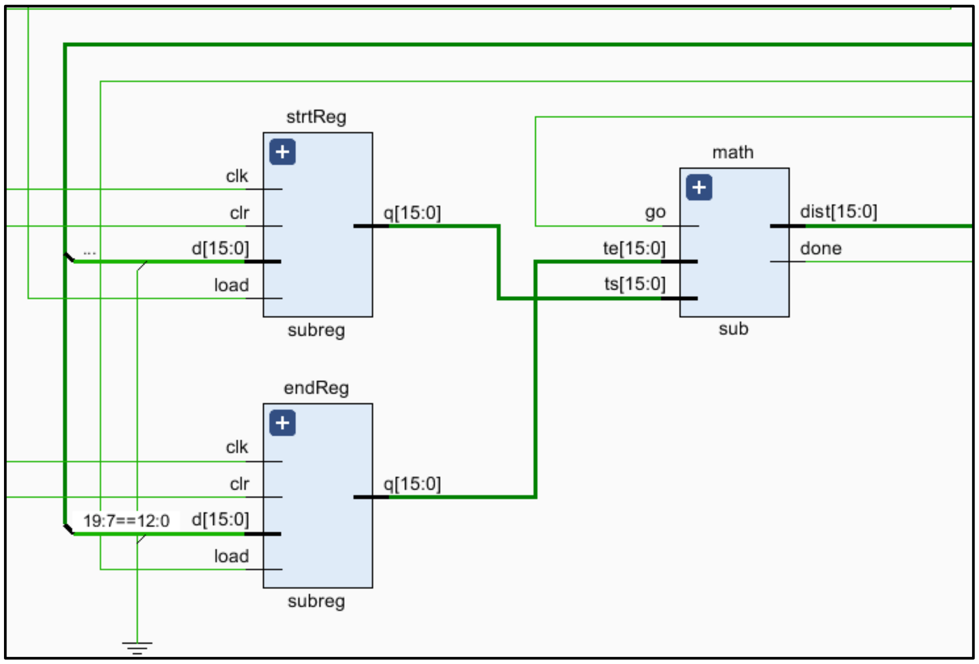


Figure 7: Distance Calculation Design

When the *trigger* pin goes low we wait for the *echo* pin to go high, indicating all pulses have been transmitted and the controller loads the start time into the ***strtReg***. We move to the ***echo*** state, where we wait for the sensor to receive all pulses or timeout. At which point the echo pin will go low, and we will load ***endReg*** with the end time and transition to the ***calc*** state. The ***math*** component will perform the following calculation on the two 16 bit values of 10us/count. We will also begin loading the ***comparatorReg***, which is used by the display component, for the next state. *(Appendix A.3)* As referenced in appendix A.3, while the temp distance is stored as a 32 bit value, we know the constraints provided by the sensor timeout and max range, so we only require the lower 16 bits.

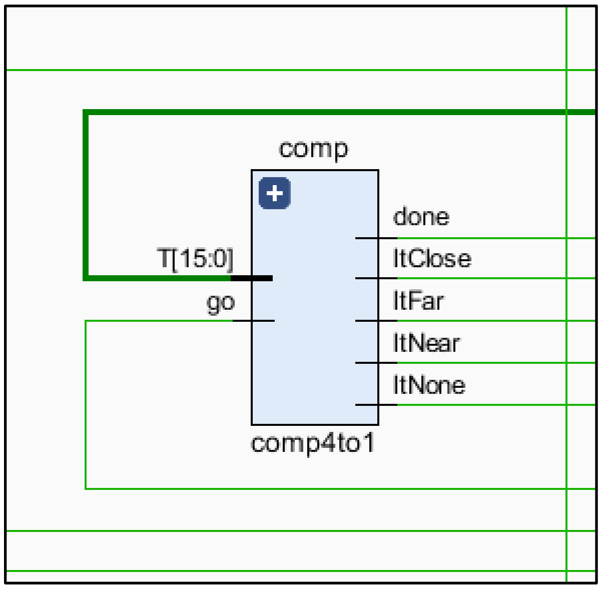


Figure 9: Compare component

Once the distance has been calculated we will transition to the **compare** state and begin comparing the distance against the thresholds defined in the table below. The threshold values will be discussed in more detail in the performance section.

Internal Distance (cm)	Classification	Haptic Duty Cycle(%)	Acoustic interval (ms)
39 <=	close	100	constant
59 <=	near	50	125 ms
79 <=	far	0	250 ms

Table 1: Object classification and feedback table

During this time, we allow the **display** and **servo** component to begin loading their respective registers for feedback and display. We also output the object classification to the on board leds. (ld[3:1] <= ltClose & ltNear & ltFar) This component will output the object classification to the ***servo*** and ***uart*** component if one is detected at all. Then the ***donecomp*** signal is populated with ‘1’ and sent back to the controller to then transition to the ***done*** state and tell ***uart*** to begin transmitting data. This completes the state machine life cycle.

<u>*Display Component*</u>

The Display component will read the distance value loaded into the ***bcdreg*** register by the controller. We’ve decided to implement a conversion method from binary to Binary Coded Decimal(BCD) for the seven segment display using the *Double Dabble* algorithm. This allowed us to better test the physical measurement thresholds and accuracy of the sensor. Due to a time constraint, the Verilog design component for converting to Binary Coded Decimal was imported from another source. *(References Geek [D.1][D.2])* The design was modified to work for 5 digit BCD values.

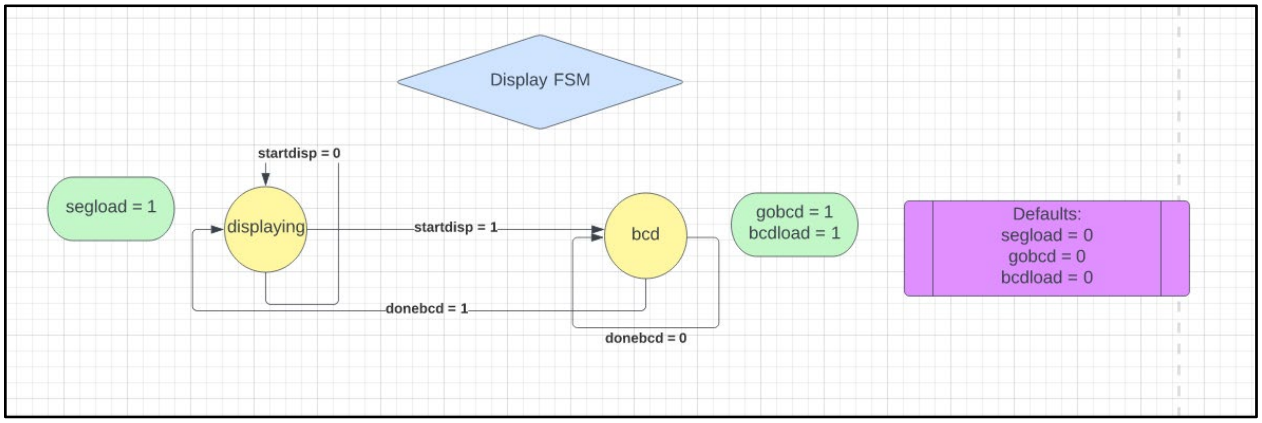


Figure 10: Display State Machine

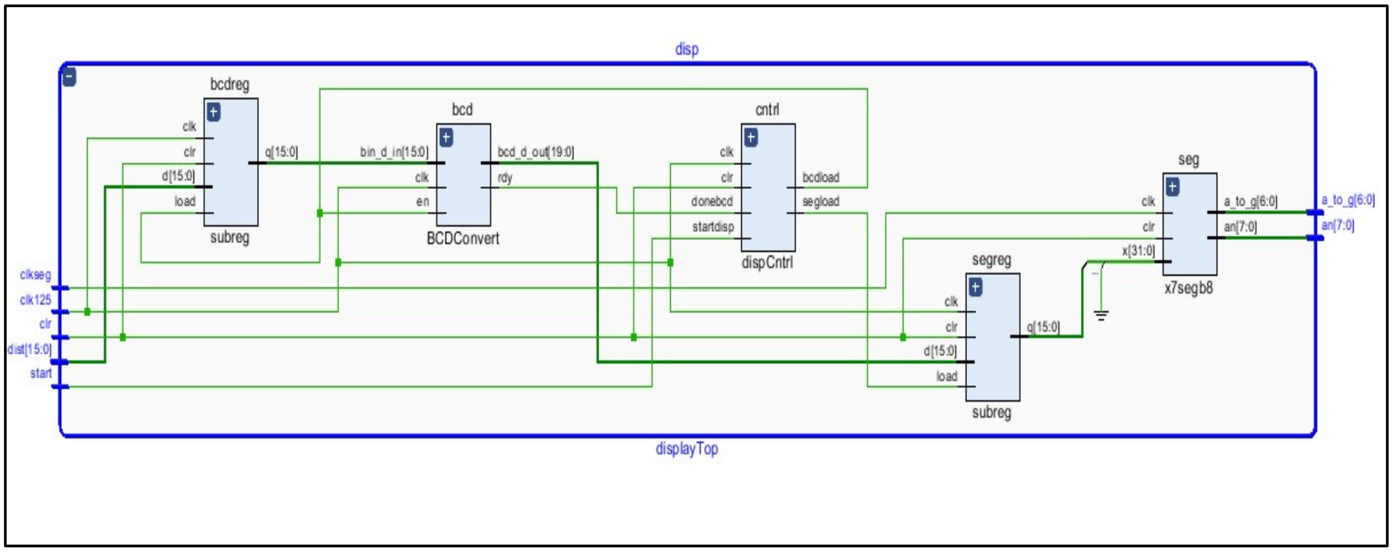
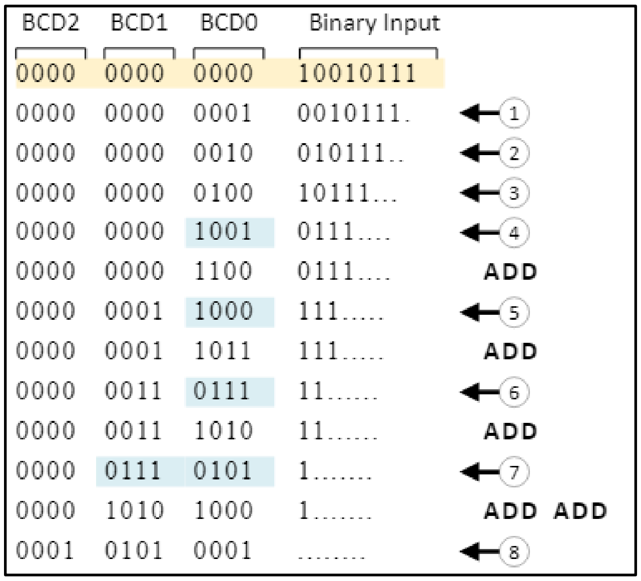


Figure 11: Display Component data path

For *Double Dabble,* we concatenate the 16 bit binary value from the ***bcdreg*** with a 20 bit temporary signal. Each cycle, we check each nibble for a value greater than 5. If a number greater than 5 is detected, we add 3 to it and once all nibbles have been checked, we continue shifting. Once the entire binary value has been shifted, we are left with a BCD value. *(Reference [C.1])*



BCD2	BCD1	BCD0	Binary Input	
0000	0000	0000	10010111	
0000	0000	0001	0010111.	← 1
0000	0000	0010	010111..	← 2
0000	0000	0100	10111...	← 3
0000	0000	1001	0111....	← 4
0000	0000	1100	0111....	ADD
0000	0001	1000	111.....	← 5
0000	0001	1011	111.....	ADD
0000	0011	0111	11......	← 6
0000	0011	1010	11......	ADD
0000	0111	0101	1.......	← 7
0000	1010	1000	1.......	ADD ADD
0001	0101	0001	....	← 8

Figure 12: Binary Coded Decimal conversion (Reference [C.2])

The Display component will begin calculating the BCD when the UST controller sets ***start*** to '1' and then transition to the ***bcd*** state. While calculating the BCD we allow the segment display register to be loaded with the output. Once the BCD component completes its conversion, it will set the ***rdy*** signal to '1' and the Display controller will transition to the ***display*** state and send the BCD to the on board display. This completes the life cycle of the display state machine.

<u>UART Component</u>

The UART component takes a selector signal, *fsel[2:0],* and a go signal, *go,* from the UST controller to indicate when the system is ready to begin transmitting the potential feedback request to the ESP module. When the UART component is instructed to begin transmitting it will take the *fsel[2:0]* signal and run it through the ***feedmux*** component to determine which acoustic feedback interval is to be requested. *(Appendix B.1)*

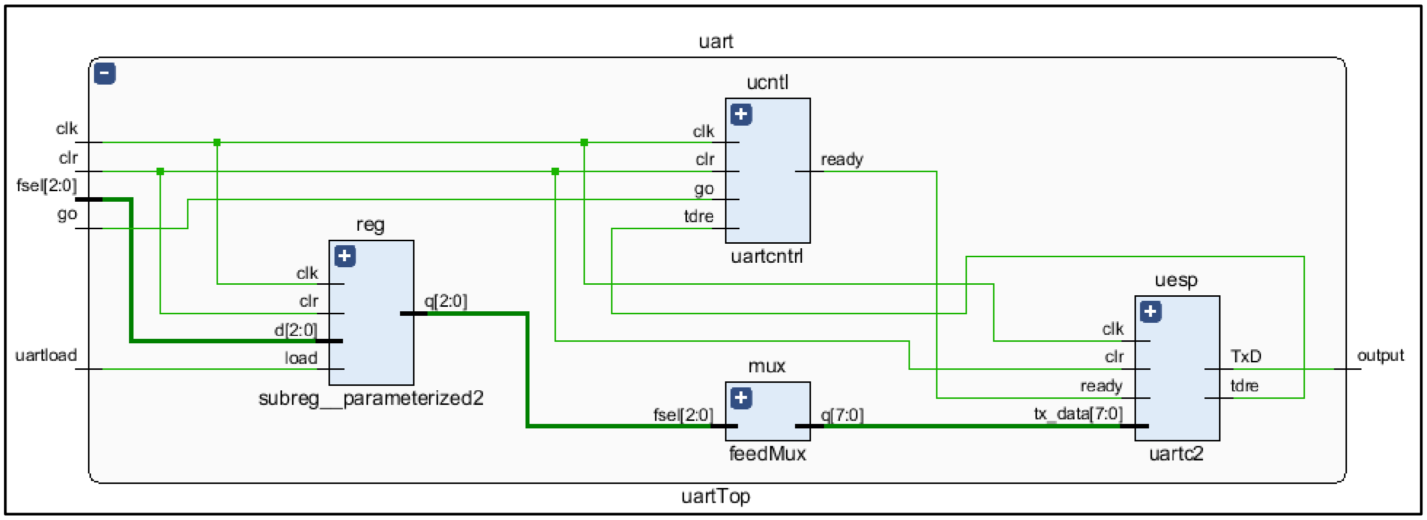


Figure 13: Uart Top-level design

The mux only checks the respective bits from *close -> far,* in case we somehow classify an object in twice. The UART controller will determine when the ***uesp*** component can begin transmitting. At which point, the incoming feedback byte ***tx_data[7:0]*** will be shifted to the ESP modules Rx buffer at a baud rate of 9600.

Fsel[2:0]	[2] Far	[1] Near	[0] Close

Table 2: Fsel bitmap

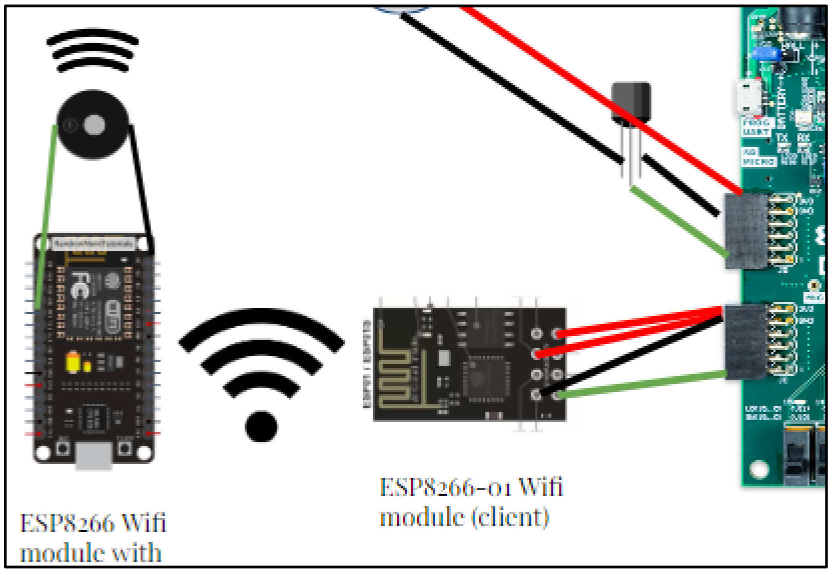


Figure 15: ESP Wifi Communication

Once the full byte has been transmitted to the ESP module and it has connected to the handheld device hosting the server, it will check the current buffer data against last cycles byte and transmit the byte over UDP to the server. If the byte of data is the same, it will not send the byte. This reduces the number of UDP packets and the load on the server. *(References [E.1][E.2][F.1][F.2])*

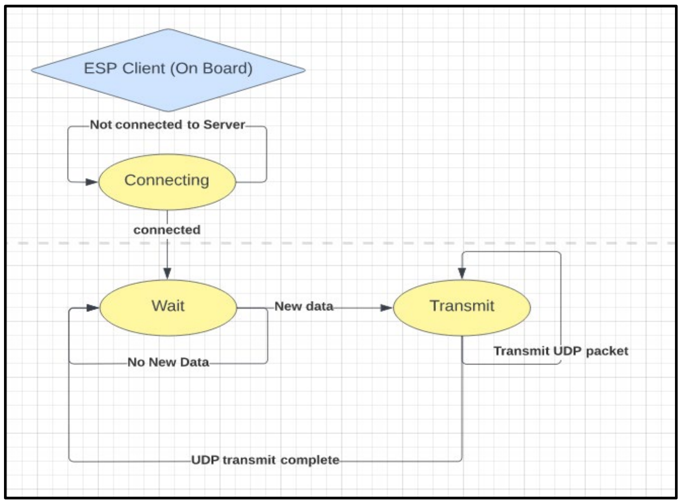


Figure 16: ESP Client State Machine (Arduino)

The ESP server hosts an access point for the client to connect to. At which point, it begins listening on a UDP port for incoming packets. When a new packet is received it will compare the data against the previous packet and determine if new feedback is required. If at any point, an object is no longer detected, we will finish out the current feedback request to prevent toggling. *(References [F.1][F.2])*

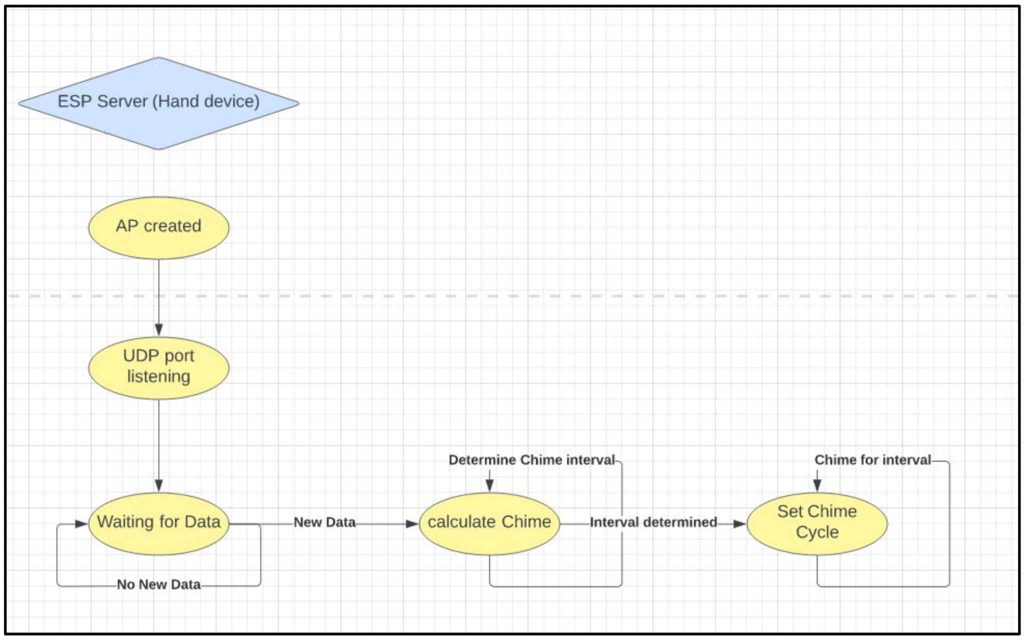


Figure 17: ESP Server State Machine (Arduino)

<u>*Servo Component*</u>

Finally, the servo component drives the haptic feedback for the system. The UST controller, during the ***compare*** state, will load the register ***server*** with the input selector, *ssel[1:0],* which is used to determine the duty cycle for the servo module.

ssel[1:0]	[1] Near	[0] Close

Table 3: ssel bitmap

Unlike the previous components, this one doesn't require a *start* signal from the controller and takes two separate clock frequencies. The 100 Khz clock is used to drive the servo module. This was done to reduce the total count per duty cycle and made the calculation easier to work with. The counter component that controls the PMOD pin Jc1, to trigger the haptic vibration, is only enabled when an object is classified as either *'close'* or *'near'*. This is done by using an *xor* gate with both bits of the *ssel* signal as the inputs. This could be changed to an *or* gate to protect against a multi classification for an object, though this shouldn't occur under normal circumstances.

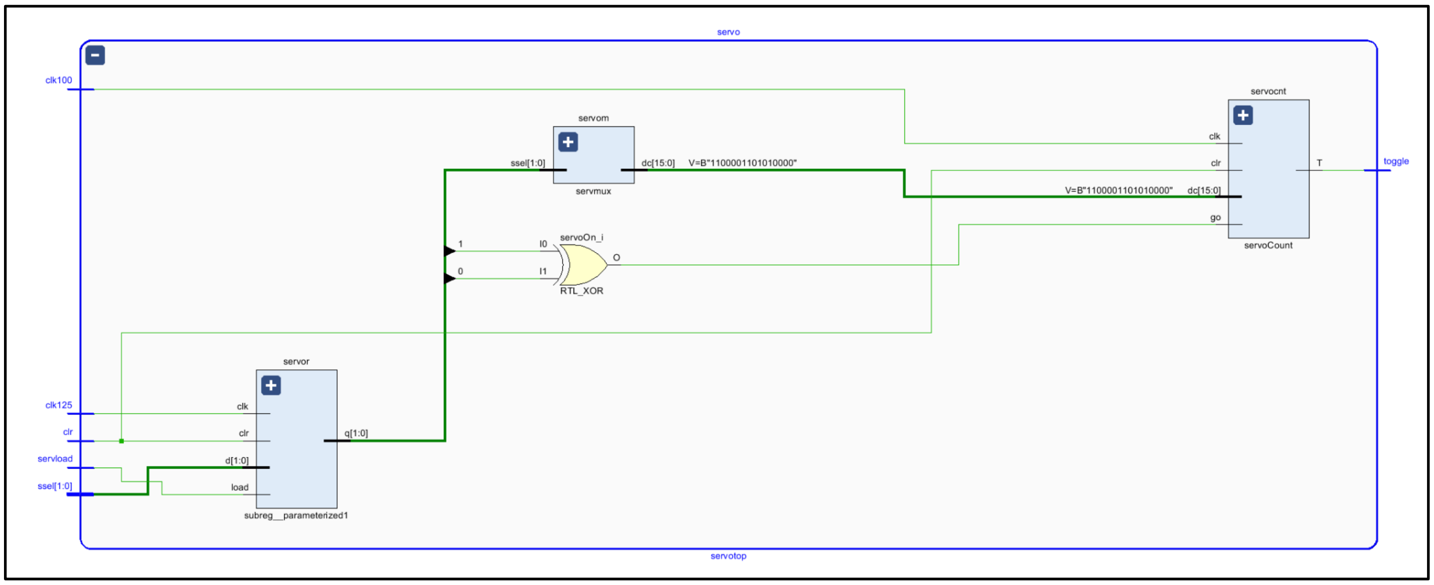


Figure 18: Servo Top-level design

The *ssel[1:0]* signal is also used to select the duty cycle for the servo module, set to 50% or 100% if an object is classified as *near* or *close* respectively. The max count for the ***servocnt*** component is 50,000 (X"C350") which at 100 Khz equates to a 500ms max interval time. The ***servocnt*** component is designed to update its duty cycle during an interval if a new object classification is detected. *(Appendix C.1)*

VI. Project Results

Performance

Overall, we're very pleased with the performance of our system. However, there are some noteworthy compromises we made to simplify the design. Provided a 12.5 Mhz clock frequency, each count for the ultrasonic sensor is equal to 13 nanoseconds. We pull the trigger pin high for 128 cycles which is 10.24 microseconds. In our distance calculation we read in a signal which is the total count from the *trigger* component, and shift it right by 8 bits. Since each 128 count is 10us, the ***math*** component interprets each 10 us' as 1 and the *sound* speed used. This is much easier than performing an actual floating point division. Since we only care about the distance to the object, we divide 343 meters / second by 2. Normally, this would leave our *sound* signal as 172, however, since each count is 10 us, we need to divide this by 10. After performing all the simplifications and division, we are left with the following distance precision loss of:

*Internal sound (340(m/s) / 343 (m/s)) * internal count (10us/10.24us) = .968 <=* ***actual Distance (%)***
Eq. 1: Actual Distance equation

We lose just under 3% of the actual measured distance to rounding. Given the short distance we are working with, we have adjusted our thresholds accordingly to *overestimate* the measured distance. This will prevent objects from being interpreted as farther away than they actually are. We used the ***actual Distance*** equation above to determine our thresholds, which would be compared against real world measurements later.

Thresholds (cm)	Desired Dist. (cm)	Measured Dist. (cm)	Error from threshold (%)
Close (39 cm)	40	38.72 ~= 38	2.6%
Near (59 cm)	60	58.08 ~= 58	1.7%
Far (79 cm)	80	77.44 ~= 77	2.5%

Table 4: Threshold error calculations from measurements

The results of our threshold testing are shown in the above table. As you can see, our initial calculations were very close to the original values we had set. We prefer our interpreted distance to be slightly below the threshold to remove any issues regarding sensor precision, so these results were desired. We believe these error values to be an acceptable loss of precision for the convenience gained while designing our system. Even though the ultrasonic sensor may introduce some inaccurate timing readings, given its quality, we can at least ensure that our system will react on the safer side.

Challenges

While SID contains a lot of moving parts and individual components all function pseudo-independently from one another, there were only two issues which required extensive testing and debugging. Most others related to determining signal size and ensuring the lossy precision of the sensor was accounted for, which was discussed previously.

The first issue being a hardware failure state with the ultrasonic sensor. While we aren't sure the exact conditions to cause the trap, we could consistently put the system into situations where, over time, either the sensor or UST state machine would go out-of-sync with the ***trigger*** component. This was most likely due to the built in sensor time out which would transition our state machine at an unknown time by setting the *echo* pin low unexpectedly. To simplify, at some point the ***trigger*** component would output '0' to the PMOD pin Jd1 when it should be outputting '1'. The solution to this problem was to immediately transition to the ***done*** state, in the ***UST controller*,** when this occurs. We would reset any measurement data, and no valid requests would be sent through the system. This solution did not introduce any noticeable impact to performance. *(Appendix A.2)*

The second challenge was the transmission of data to the ESP module through UART. Initially, the UART communication was successful from the board to PMOD pin jd1, however the ESP module was reading junk data. We unfortunately spent a very long time debugging this issue, trying different baud rates and frequencies, when it was discovered that our baud rate calculation was off by 1. We originally were using a baud rate of X"515", and the ESP module was sensitive enough to pick up on this. Once this was adjusted to X"514" communication was working as expected. *(Appendix B.2)*

Improvements

Aside from the general design cleanup, we do have a few major hardware and system changes that would greatly improve SIDs functionality. The first would be the addition of an accelerometer. With the addition of this module we could take into consideration the angle at which the sensor is pointed and eliminate false positives from the sensor detecting the ground. We would also be able to keep a running average of the tilt of the device and relevel ourselves if we begin to travel up or down hill. An accelerometer would also give us the capability to detect step length per individual and adjust our thresholds accordingly. A calibration mode for general step length would be added for default thresholds, and then another running average would be used to vary the object classification distances based on recent step length.

The use of a bluetooth module instead of the ESP modules would be hugely effective at utilizing the power of smart phones. In some cases, we may even be able to use the smart phone as a way to calculate step length or steps taken and relay this info to SID. Bluetooth is also much cleaner, in this case, than wifi as the access point hosted on wifi is not private and anyone would be able to see a network for the handheld device. This opens up our system to deauthentication attacks as it's a 2.4 Ghz network.

And finally, upgrading the quality of the ultrasonic sensor would greatly improve speed and accuracy of the system. The sensor we used for this project, as mentioned, ran at a frequency of 16 Hz, and a better module can measure at atleast 25 Hz. That alone would make a huge difference.

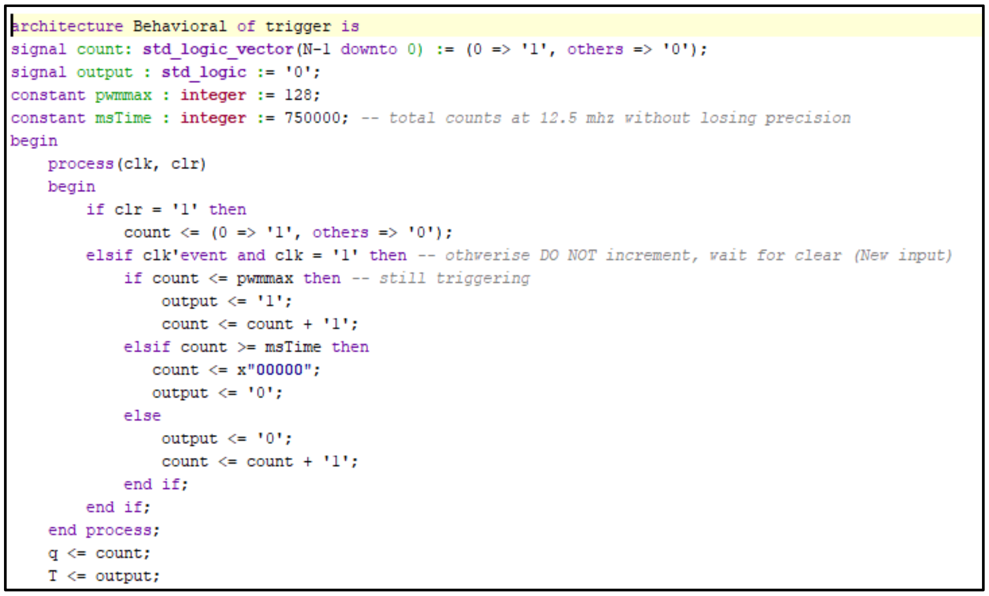
VII. Conclusion

We spent a lot of time and effort working through the design limitations and hardware kinks to get SID where they are now. We're exceptionally proud of the performance and overall functionality we were able to achieve in such a short period of time. We hope to continue on with SID and iterate on its already impressive implementation. As we've said in the beginning, technology is at a point in time where there are no excuses to keep affordable and available solutions from people with disabilities. We genuinely believe that SID has the potential to make a difference and at a cost that won't impose more stress on the affected individuals who would use it.

VIII. *Appendices*

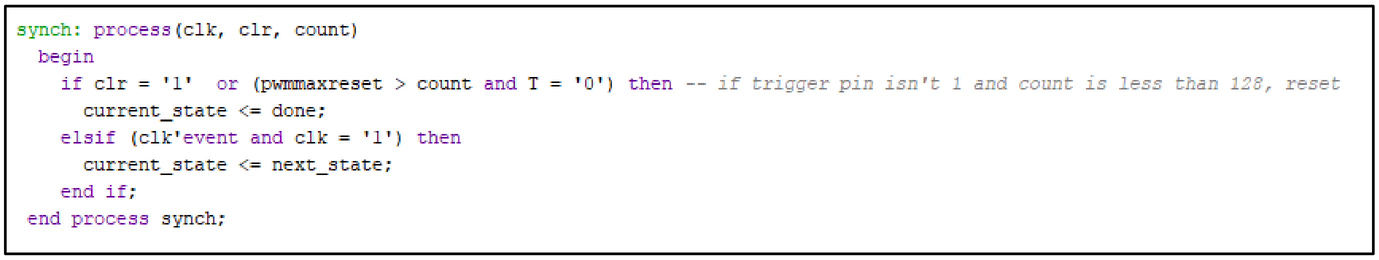
A. *Ultrasonic Top components*

1. *Trigger Component*



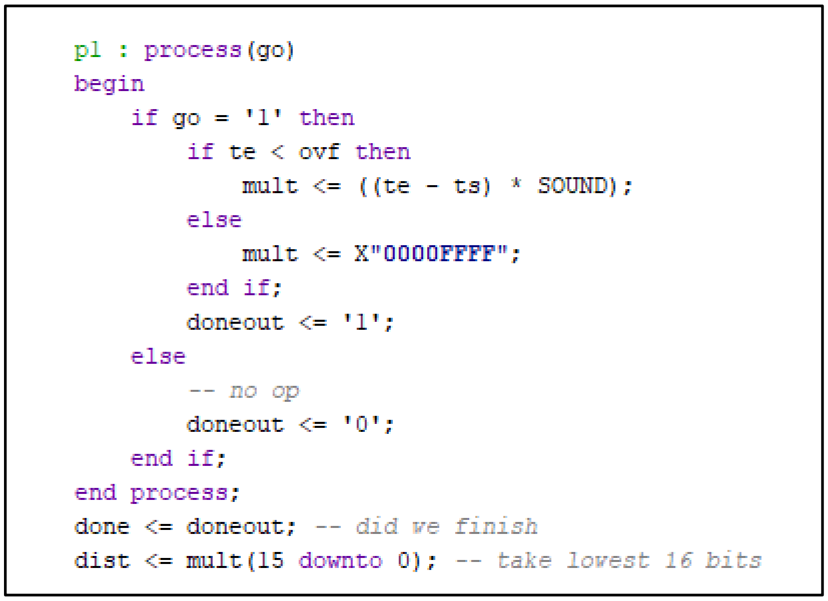
```
architecture Behavioral of trigger is
signal count: std_logic_vector(N-1 downto 0) := (0 => '1', others => '0');
signal output : std_logic := '0';
constant pwmmax : integer := 128;
constant msTime : integer := 750000; -- total counts at 12.5 mhz without losing precision
begin
    process(clk, clr)
    begin
        if clr = '1' then
            count <= (0 => '1', others => '0');
        elsif clk'event and clk = '1' then -- othwerise DO NOT increment, wait for clear (New input)
            if count <= pwmmax then -- still triggering
                output <= '1';
                count <= count + '1';
            elsif count >= msTime then
               count <= x"00000";
               output <= '0';
            else
                output <= '0';
                count <= count + '1';
            end if;
        end if;
    end process;
    q <= count;
    T <= output;
```

2. *State Machine Trap Solution*



```
synch: process(clk, clr, count)
  begin
    if clr = '1'  or (pwmmaxreset > count and T = '0') then -- if trigger pin isn't 1 and count is less than 128, reset
      current_state <= done;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
    end if;
 end process synch;
```

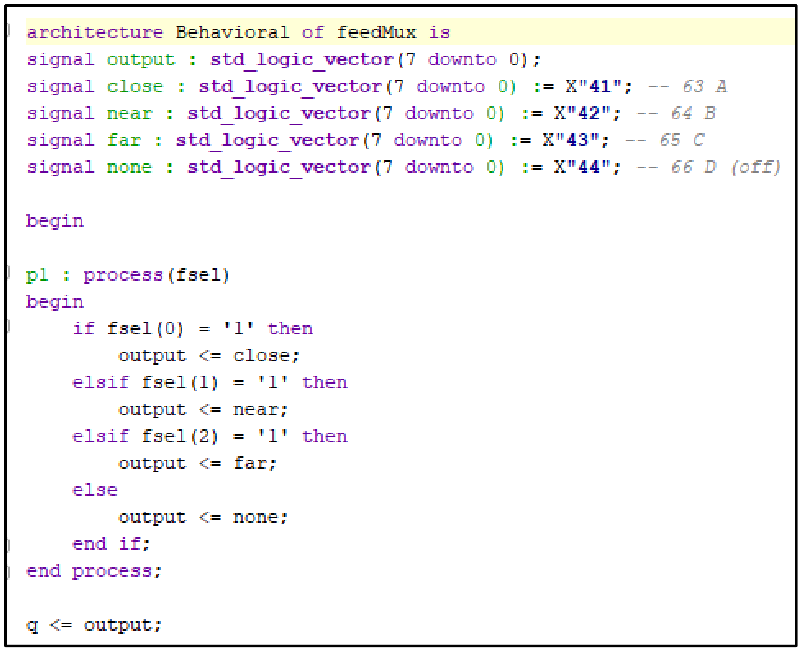
3. *Math Component*



```
pl : process(go)
begin
    if go = '1' then
        if te < ovf then
            mult <= ((te - ts) * SOUND);
        else
            mult <= X"0000FFFF";
        end if;
        doneout <= '1';
    else
        -- no op
        doneout <= '0';
    end if;
end process;
done <= doneout; -- did we finish
dist <= mult(15 downto 0); -- take lowest 16 bits
```

B. *UART Top Component*

1. *ESP Feedback Byte Mux*



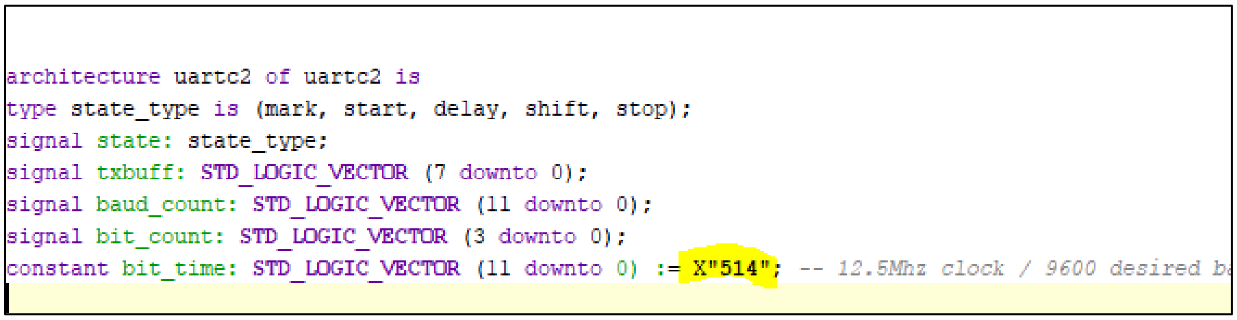
```
architecture Behavioral of feedMux is
signal output : std_logic_vector(7 downto 0);
signal close : std_logic_vector(7 downto 0) := X"41"; -- 63 A
signal near : std_logic_vector(7 downto 0) := X"42"; -- 64 B
signal far : std_logic_vector(7 downto 0) := X"43"; -- 65 C
signal none : std_logic_vector(7 downto 0) := X"44"; -- 66 D (off)

begin

p1 : process(fsel)
begin
    if fsel(0) = '1' then
        output <= close;
    elsif fsel(1) = '1' then
        output <= near;
    elsif fsel(2) = '1' then
        output <= far;
    else
        output <= none;
    end if;
end process;

q <= output;
```

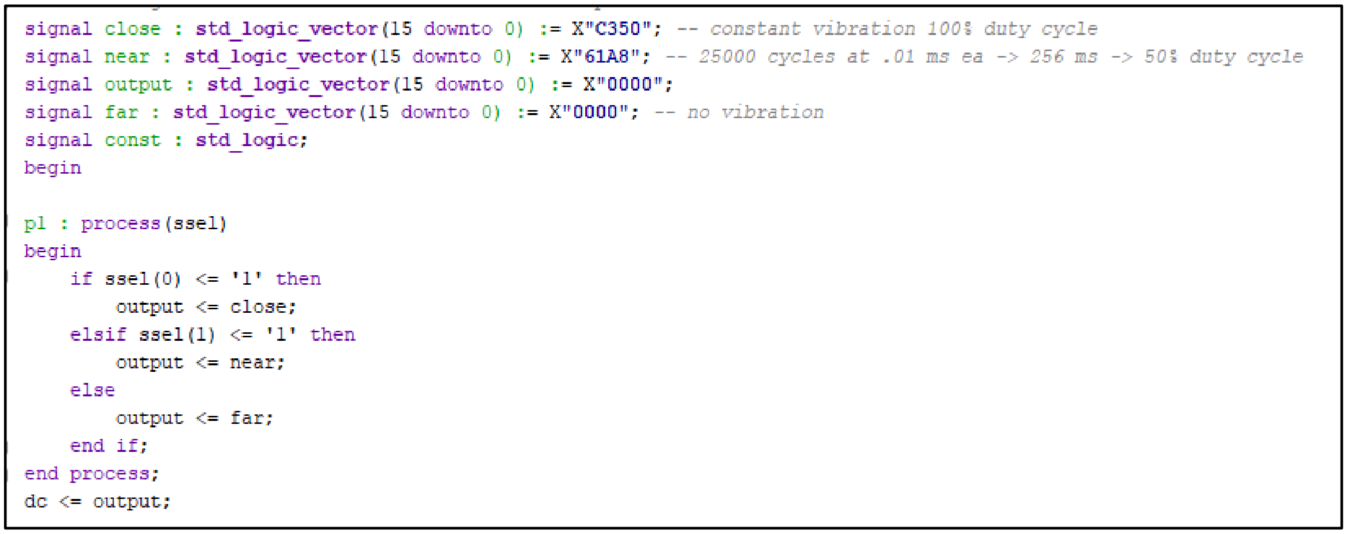
2. UART Baud Rate



```
architecture uartc2 of uartc2 is
type state_type is (mark, start, delay, shift, stop);
signal state: state_type;
signal txbuff: STD_LOGIC_VECTOR (7 downto 0);
signal baud_count: STD_LOGIC_VECTOR (11 downto 0);
signal bit_count: STD_LOGIC_VECTOR (3 downto 0);
constant bit_time: STD_LOGIC_VECTOR (11 downto 0) := X"514"; -- 12.5Mhz clock / 9600 desired b
```

C. Servo Top Component

1. Servo Module Counter



```
signal close : std_logic_vector(15 downto 0) := X"C350"; -- constant vibration 100% duty cycle
signal near : std_logic_vector(15 downto 0) := X"61A8"; -- 25000 cycles at .01 ms ea -> 256 ms -> 50% duty cycle
signal output : std_logic_vector(15 downto 0) := X"0000";
signal far : std_logic_vector(15 downto 0) := X"0000"; -- no vibration
signal const : std_logic;
begin

p1 : process(ssel)
begin
    if ssel(0) <= '1' then
        output <= close;
    elsif ssel(1) <= '1' then
        output <= near;
    else
        output <= far;
    end if;
end process;
dc <= output;
```

IX. References

A. *Average Stride length*
   1. *https://livehealthy.chron.com/average-walking-stride-length-7494.html*
   2. *https://marathonhandbook.com/average-stride-length/*
B. UltraSonic sensor data sheet
   1. https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf
C. Double Dabble
   1. https://www.realdigital.org/doc/6dae6583570fd816d1d675b93578203d
   2. https://www.realdigital.org/img/05824566b6b8b7e5fb9da9451a8018a4.svg
D. Double Dabble Creator Link/repo (GEEK)
   1. https://www.youtube.com/watch?v=Q-hOCVVd7Lk
   2. https://github.com/josh-macfie/VerilogDoubleDabble/blob/master/BCDConvert.v
E. Uart
   1. https://moodle.oakland.edu/mod/resource/view.php?id=5863610 (transmit)
   2. https://moodle.oakland.edu/mod/resource/view.php?id=5863609 (general)
F. ESP Server/Client
   1. https://randomnerdtutorials.com/esp8266-web-server/
   2. https://randomnerdtutorials.com/esp8266-nodemcu-client-server-wi-fi/