EEE-102 Term Project: Automatic Ticket Scanner

Mert Örnek EEE 102-3 22302052 19.05.2024

YouTube Link: *https://www.youtube.com/watch?v=RoMD62wqEBM*

Purpose

The purpose of this project is to build a setup that can guide a person to a specific position, photographically identify the ticket on that person and check that person into the system according to the validity of the ticket by using VHDL on BASYS3.

Design Specifications

The design can be split into two parts: the guidance system and the scanning system. These two parts make up the fundamental functions of the setup and are integrated into a finite state machine.

Due to the complexity of the design, the modules will be individually explained in detail first before talking about their relationship with the top finite state machine module and how the system operates under different sets of circumstances.

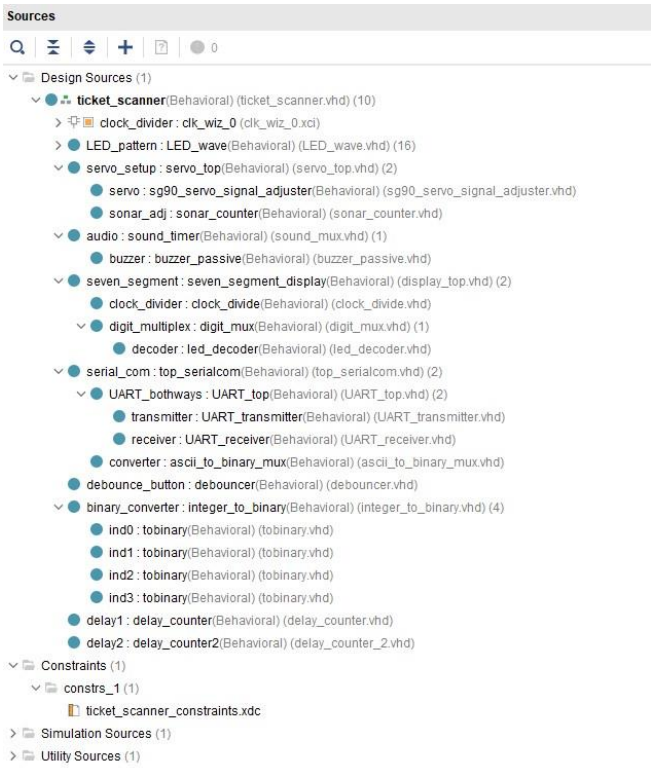

Figure 1: Design Sources

The first submodule of the system is an **IP-generated clock divider**, converting the 100MHz clock of the BASYS3 to a 5MHz clock. The reason for the inclusion of this module was that -through the power consumption diagrams found under the implementation bar- it was found that a system that even a few of these modules used together would have excessive power consumption. As a result, the main clock was divided before being used in the logic functions of the system to conserve power and prevent the board from exceeding the maximum junction-temperature. The code for this module is not included in the appendices, as it is a box-standard IP-catalog item configured to convert a 100MHz clock to a 5MHz one. Before concluding this section, it is worth noting that every single submodule found in this project uses the 5MHz clock frequency as standard, and not the 100MHz one which is only used to initiate this module.

The second submodule used in this project is the **LED wave generator**(LED_wave.vhd). This submodule can be thought of as a 16-bit 8-to-1 multiplexer that drives its own submodule, LED timer(LED_timer.vhd). The LED timer submodule has the 5MHz clock signal(i_clk), an integer(i_delay) and a trigger signal(i_trigger) as inputs, and a single output(o_signal) that gives the desired periodic pattern to a single LED. This submodule consists of

a counter that has a period of 1 second that has a duty cycle of 15%, meaning the selected LED will stay on for 0.15 seconds within that period. However, the counter determines when to give off the high signal based on the integer input, which converts to a delay of 0.025s per integer. For example, an LED that has an integer input of 8 will stay turned off for 0.2s, turn on for 0.15s, and turn off for the rest of the period at 0.65s. This submodule is used in the higher module(LED_wave.vhd), which is a 5MHz clock and a 3-bit logic vector input, and a 16-bit logic vector output to drive all 16 of BASYS3's LEDs. The higher module essentially consists of a multiplexer that assigns different delays to LEDs depending on the 3-bit input and relays them to the LEDs. These combinations include: "000" for no LED signal, "001" for left to right sweep, "010" for right to left sweep, "011" for an inwards wave from the sides of the LED vector, "100" for an outward wave from the center to the sides, and all other conditions correspond to a placeholder wave that turns all LEDs on and off simultaneously.

The third submodule of the setup(servo_top.vhd) is used to drive **the servo motor**, which itself consists of two submodules: one to generate the PWM signal necessary to operate the servo(sg90_servo_signal_adjuster.vhd) and one to generate the oscillatory counter effect seen in the project(sonar_counter.vhd). Starting off with the servo signal adjuster, it has the 5MHz clock(i_clk), a reset input(i_res) and an integer(i_high) that determines the duty cycle as inputs; and a single output that is the PWM signal(o_clk). For additional context, an sg90 servo motor operates on the premise that it receives signals with a total period of 20ms with a duty cycle ranging from 5%(1ms) to 10%(2ms). The duty cycle of the signal determines the angle servo motor makes with its vertical axis, with 1.5ms corresponding to 90 degrees from the surface and 1ms and 2ms corresponding to 0 degrees and 180 degrees respectively. Returning back to the module (sg90_servo_signal_adjuster.vhd), the counter of the module generates a PWM signal with a total period of 20ms and determines the duty cycle based on the integer input, with an input of '100' generating a signal with a 5%(1ms) duty cycle and an input of '200' generating a signal with a 10%(2ms) duty cycle, with all other integers in between having a directly correlational effect on the duty cycle. With this in mind, we can move on to the second submodule which is the oscillatory counter(sonar_counter.vhd). This counter has a clock(i_clk) and a reset(i_res) input, and a single integer output(o_waveperiod) that has a range between 100 and 200. What this module does is generate an oscillating integer output with a period of 1.6 seconds. The counter starts from 150 and goes up to 200. Once it reaches 200, it starts decreasing down to 100, at which point it starts going back up again. In terms of its functioning, it can be tough of as a finite state machine with a signal(s_indicator) that turns on and off based on the last reached end-point value, determining the active state. Now moving on to the top module of the servo(servo_top.vhd), it has a

clock(i_clk) and a binary switch(i_sw) input, and a PWM signal output(o_servosignal) that directly drives the servo. This module acts like a 4-to-1 multiplexer that determines the movement of the servo at a particular instance. The switch inputs(i_sw) correspond to: “00” for oscillating signal between 0 and 180 degrees, “01” for 90 degrees, “10” for 0 degrees and “11” for 180 degrees. As a general remark, it is worth noting that the servo motor used in my project is not connected to a level shifter, meaning it operates on 3.3V of the BASYS3 rather than the 5V the datasheet specifies. This results in the servo not going all the way back or forth. However, the oscillatory periods still match and -based on oscilloscope and simulation results- the code does generate the signal specified in the datasheet (see “Results” section).

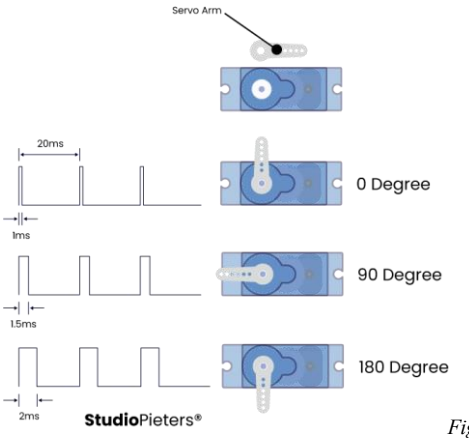

Figure 2: Angle/duty cycle relation of an SG90 (Pieters)

Moving onto the **piezoelectric buzzer** submodule of the project(sound_mux.vhd), it has a single submodule of its own(buzzer_passive.vhd) which generates the adjustable signal needed for the piezo buzzer(type: MH-FMD). For added context, a piezo buzzer is an electronic buzzer that can output audio at different frequencies depending on the frequency of the signal it receives. Starting off with the module that generates the buzzer signal(buzzer_passive.vhd), it has a clock input(i_clk), a switch input(i_switch), a 4-bit logic vector(i_type) determining the signal frequency, and the buzzer signal output(o_buzzsignal). The module essentially consists of a 16-to-1 multiplexer that determines the signal frequency based on the 4-bit logic vector(i_type) and a clock divider that creates a signal with a 50% duty cycle at the specified frequency. The notes corresponding to the switch inputs, in the rising order from “0000” to “1111”, are as follows: A, B, C, D, E, F, A#/Bb, G#/Ab, G, F#/Gb, D#/Eb, C#/Db, C5, C6, G6 and C7. Here it is worth noting that the project only utilizes 4 of these notes -one for each directional signal-, and the rest were added at initial stages of development to trial the buzzer. They were kept on after the tones used for each LED direction signal were finalized. Now moving on to the higher module(sound_mux.vhd), it has the same inputs and outputs as the lower module(buzzer_passive.vhd). The

purpose of this upper module is to regulate the period of buzzer signals, with each signal giving the buzz specified by the type input(i_soundtype) buzz for a period of 0.1s once the 1-bit switch input(i_switch) is triggered before shutting off permanently unless the switch input(i_switch) is shut off('0') and turned back on('1').

Now moving onto the **7-segment display**(display_top.vhd), it consists of a clock divider submodule(clock_divide.vhd) and a multiplexer that determines which segment to display(digit_mux.vhd), with the multiplexer being connected to its own decoder submodule(led_decoder.vhd), which uses a binary input from "0000"(decimal 0) to "1001"(decimal 9) to determine which segments to light up (all other binary values are ignored and result in no segments lighting up). I will not go into as much detail with the 7-segment display as I did with the other submodules, as it essentially is a modified version of the 7-segment display constructed in Lab5. However, a few general comments will be made as to how the 7-segment display of the BASYS3 works: The 7-segment display of the BASYS 3 is driven by a 7-bit cathode(segment) input and a 4-bit anode input, with anodes determining which of the 4-digits of the display to light up and the cathodes determining the displayed value. Since -without a clock signal- the BASYS3 can light up one digit at a time, the anode and cathode outputs are configured so that the BASYS3 oscillates between the digits at a frequency of 200Hz generated by the clock divider(clock_divide.vhd).

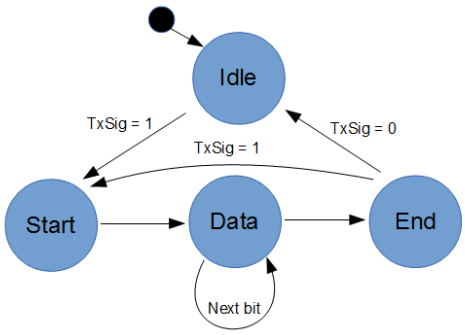

Figure 3: FSM diagram of UART Transmitter(A UART implementation in VHDL)

Now moving onto the submodule that facilitates **the serial UART communication**(top_serialcom.vhd). This submodule consists of two main submodules: the top module for the transmitting and receiving functions of the UART(UART_top.vhd) -which in turn consists of the UART transmitter(UART_transmitter.vhd) and receiver(UART_receiver.vhd) submodules- and an ASCII-to-binary 3-to-8 decoder(ascii_to_binary_mux.vhd).

Here, it is worth noting that the name of the component is a carryover from initial design stages and does not reflect the circuit component representing the current iteration of the module. Starting with the concept of serial communication, it essentially is a method of digital communication that consists of a single or two wires and sends and/or receives data a single bit at a time (Serial Communication Basic Knowledge). The BASYS3 is equipped with a USB-RS232 interface, meaning that it can engage in serial data transmission or receive data that is compatible with a UART cable. UART data is of a high('1') signal by default. Data transmission or reception is initiated when a first "start" bit consisting of a low('0') signal is sent or received. After the communication has been initiated, any arbitrary number of data bits or parity bits can be transmitted or received until the stop bit is detected or transmitted. For the purposes of this project, 8 data bits, no parity bit and a single stop bit at a baud rate of 9600 bits/s was utilized, as this was the configuration of the device BASYS3 was meant to communicate with. Both the receiver and transmitter portions of the UART were configured in a finite-state machine form with four states: s_idle, s_start, s_data and s_stop. The idle state is the default state of the FSM, and it remains at idle unless either the start bit is detected by the receiver or generated by the transmitter depending on the start input. Once this bit is either transmitted or received, FSM moves into the start state. This state solely exists to wait for the period of the start bit so as to not disrupt the periodic sequence necessary for successful serial communication. Once the timer is done, the next state is initiated, which is data. At the data state, the FSM matches the pre-determined baud rate and periodically stores or transmits data by using a shift register. Once all 8 bits are either transmitted or received -which is tracked by a counter whose period of shifting depends on the specified baud rate- the FSM moves onto the stop state. This state exists to determine the end of the serial communication procedure depending on the specified number of stop bits (1 in this case). After this state, the FSM returns back to idle and waits there until either a request to transmit data is sent to generate a start bit via the transmitter or a start bit is detected by the receiver. The UART transmitter and receiver are different modules, but their inputs and outputs are grouped together in a higher module(UART_top.vhd). For the purposes of this project, both the receiver and the transmitter need to be able to send two types of data each. The transmitter sends one of two data strings, which are "snap_" and "wait_", and the receiver identifies one of two 8-bit data strings: "01000101"('e' in ascii) and "01010010" ('r' in ascii). The "snap_" signal communicates to the computer to initiate the ticket scanning sequence, and the "wait_" signal shuts the computer program down and ends the scanning sequence remotely. These strings are transmitted by using a 2-to-1 Multiplexer, a 3-to-8 Decoder (ascii_to_binary_mux.vhd) and a 15-bit shift register. Here's how it works: Depending on the type of signal desired to be sent, the submodule needs to be primed first by being told which of the two strings are going

to be sent by a 1-bit input (i_starttrans(0)). This activates the 2-to-1 MUX and assigns the 15-bit (3 bits * 5 characters) binary string of the specified string to the 15-bit shift register. After the data is registered, it is saved there until the signal to start transmitting is received (i_starttrans(1)). If received, the shift register starts shifting 3-bits at a time from the least to the most significant bit and feeds the data to the 3-to-8 decoder. The decoder uses their strings to convert each 3-bit input to one of 8 specified 8-bit ascii data strings. These strings are updated each clock cycle and are kept track of by a counter that keeps track of the number of transmitted characters. The counter counts up to 5 characters and once they are all transmitted, no new character can be transmitted unless the counter is reset (i_starttrans(0)). The receiver -on the other hand- is much simpler. It updates an 8-bit register each rising edge of the 'done' output of the receiver module and communicates the relayed character to the upper modules by first identifying it through an 8-bit comparator and then relaying the output via a 2-to-1 multiplexer to the top finite state machine module.

Since the logic circuit utilizes a button to initiate the state transitions of the FSM, it was necessary to implement a **button debouncer** module (debouncer.vhd). A button debouncer exists to counteract the oscillatory signal patterns associated with physical button components. Once can build a debouncer in many different configurations, however this specific one can be thought of as using a clock divider, two D-Flip Flops and an AND gate. The clock divider divides the 5MHz signal to 5Hz, meaning that the button output has a period of 0.2 seconds once the button(i_btcn) is triggered during the rising edge of the clock cycle. After this period of 0.2s for which '1' signal is given, the button can either be let go of or kept pressed down, the debounced button signal will become '0' either way.

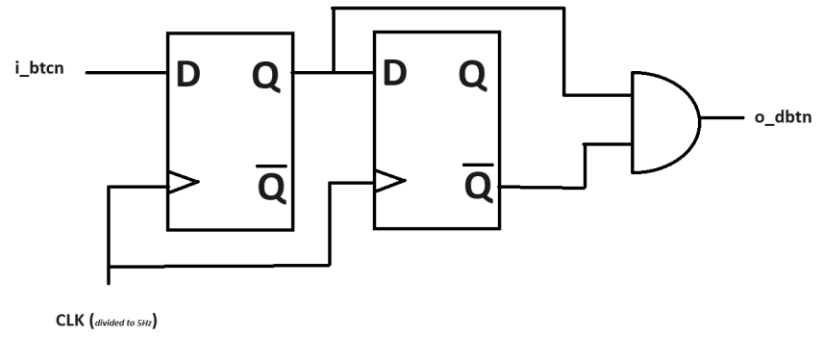

Figure 4: Diagram of the debouncer

Our last two submodules are the **integer to binary converter** and the two **delay counters** . Starting with the delay counters, they follow the one-shot counter logic explained previously in the design sources. They each have clock(i_clk) and reset(i_reset) inputs and a done(o_done) output that confirms when the specified time is up. The first of these counters has a period of 4s(delay_counter.vhd) until o_done's rising edge when initiated(i_res <= '0'), and the second has a period of 8s(delay_counter2.vhd). These counters exist to solicit certain state transitions of the FSM. Now moving onto the integer to binary converter (integer_to_binary.vhd), this module operates four instances of the same submodule (tobinary.vhd), which essentially is an integer to binary encoder for single digit values. The top module these four submodules connect to(integer_to_binary.vhd) does two things: It combines all binary digit inputs into a 16-bit logic vector which is transmitted and broken down again in the 7-segment display module and -initally- splits the integer value into its digits by utilizing arithmetic operations in VHDL such as integer division and mod functions.

Now that all the submodules have been explained in detail, we can actually move onto the top module of the scanner(ticket_scanner.vhd) which is a **Finite State Machine** consisting of 7 states: s_off, s_advance, s_backwards, s_snap, s_forward, s_aftermathP and s_aftermathN. The state transitions of the FSM are synchronized to the rising edge of the 5MHz clock and have outputs that either depend on just the state or both state and input. In that regard, it can be considered a hybrid or combination of Moore and Mealy FSMs. Before explaining the transitions and implications of each state in detail, an FSM diagram of the project will be drawn. Keep in mind that, due to the large number of inputs and outputs involved, this diagram will only explicitly state the condition that solicits the conditions necessary to drive a state transition and will generalize all the other conditions for the sake of keeping the diagram simple and understandable. After this diagram, the information I just mentioned will be explained in detail. Here is the simplified diagram of the FSM:

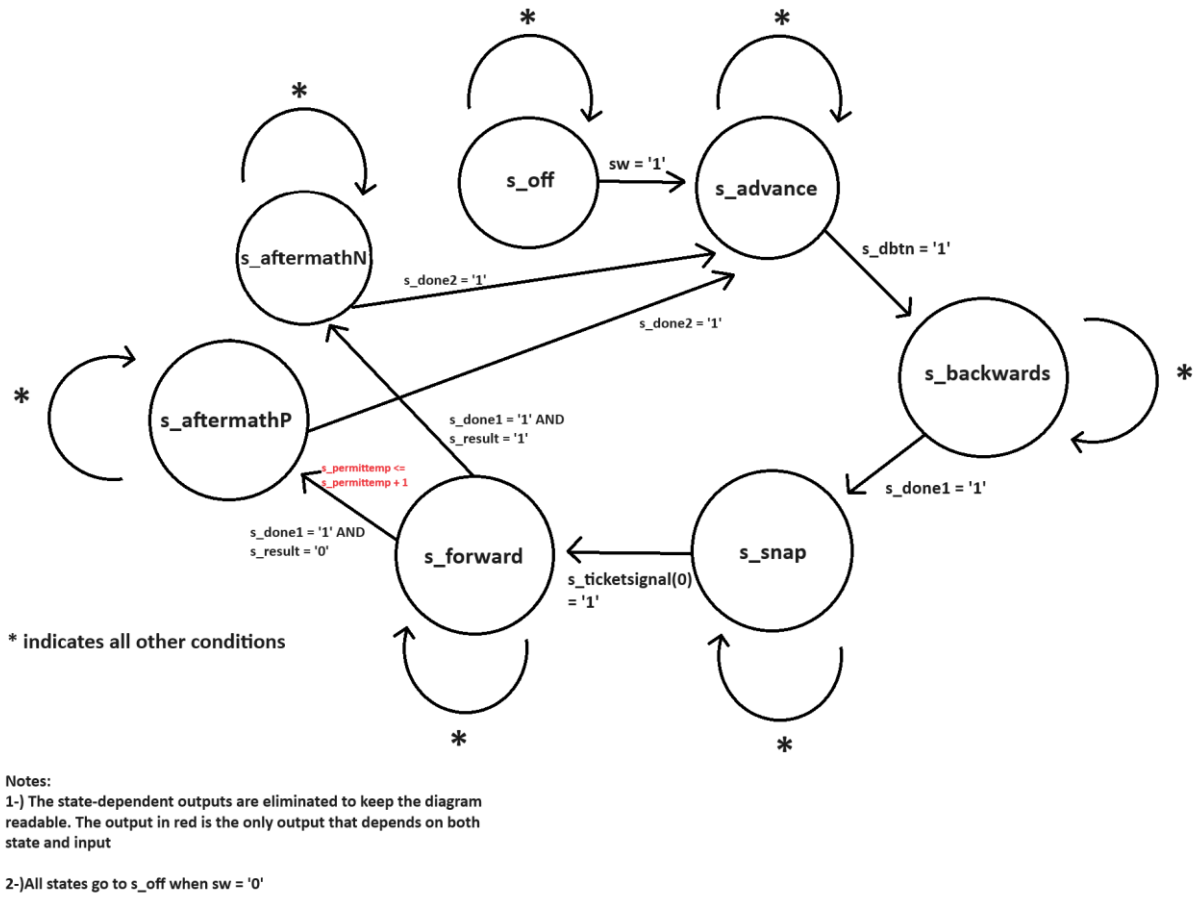

Figure 5: FSM diagram of the project

Here are the descriptions of outputs and transitions out of each state:

s_off (initial): This state represents the automatic ticket scanner being turned off. This state shuts off all the LEDs, lock the servo in a 90 degree upwards position, resets the s_permittemp -a counter used to identify the number of ticket approvals- to 0, resets the 4(s_reset1 <= ‘1’) and 8(s_reset2 <= ‘1’) second counters and uses the UART transmitter to send the “wait_” string to the laptop, which shuts the program down. The only possible next state out is s_advance, which is transitioned to when the switch component(sw) is turned on(‘1’).

s_advance: The “advance” state represents the base operating state when the system is turned on. This state does not send any UART signals but primes the module for a “wait_” signal, starts oscillating the servo in a periodic manner, signals LEDs to start oscillating from right to left(“s_ledtype <= “001”) and sets the buzzer tone to note C7 (s_soundtype <= "1111"). It also sets specific LEDs as the positions at which the buzzer switches on and off, with the buzzer turning on at LED(0) and off at LED(8). There are two ways out of this state: turning the

switch(sw) off(‘0’) will result in transitioning back to s_off and utilizing the debounced button will result in a transition to the next state in the cycle, which is “s_backwards”.

s_backwards: This state represents the brief period before the ticket is scanned in which a person is expected to position themselves in a way that is exposed to the camera. This state sets the LED type as “100” -which is a center to corners wave pattern that signifies going backwards-, sets the buzzer note to F(s_soundtype <= “0101”) and sets the LED at which the buzzer turns on to 8 and turns off to 12. The servo keeps oscillating, however the UART is primed for the “snap_” signal to be used during one of the following states. This state, while keeping the reset of the 8s counter on(s_reset2 <= '1'), it initiates the 4 second one-shot counter. There are two possible different next-state transitions. If the switch(sw) is turned off(‘0’) it will transition back to s_off alongside priming the UART transmitter for a “wait_” signal to be transmitted during that state. The other option is to wait 4 seconds until the one-shot counter is done counting(s_done1 = '1'), at which point the FSM will transition to the state “s_snap”.

s_snap: This state is named after the UART signal it transmits to the computer, which is “snap_”. This signal, when received by the Python program interfaced with the UART connection, triggers the program to take a picture, scan the visible QR code in that picture, check the database to see if the number it represents has been scanned before, and send a signal that communicates if its valid(‘e’ in ascii) or not(‘r’ in ascii) via the UART receiver of the BASYS3. As is implied, the state sends a “snap_” signal via the UART transmitter, sets LED type to “111” which is a non-directional uniform blinking pattern, keeps the servo oscillating, resets the 4 and 8 second one-shot counters and does not turn the piezo buzzer back on. This state also has two possible transitions out of it: The first case is -as is found in all states- turning the switch(sw) off and transitioning back to s_off. The second possible transition is triggered if the binary signal for the ascii representation of either letter ‘e’ or ‘r’ is received, in which case the next state becomes s_forward. The type of letter is also stored in the register s_result, which will be used in one of the later transitions.

s_forward: After successfully taking a picture and scanning the QR code, this state essentially undoes the requested guidance movement of the state s_backwards, which was to move further away from the camera to a

spot from which a picture could be taken. This state does the opposite, meaning the person is asked to move forward. The outputs of this state include an oscillating servo, UART transmitter reprimed with the “wait_” signal(s_starttrans <= "00"), LEDs making a come forward pattern from the outer edges to the middle("011"), the LED at which the buzzer turns on being set to LED(15) and turns off to LED(11). The buzzer note is also changed to a B("0001"). The 8 second one-shot counter remains untouched(s_reset2 <= ‘1’) while the four second one-shot counter is initiated(s_reset2 <= ‘0’). This state has three possible external transitions: The switch(sw) being turned off which leads to s_off, and two other possible external transitions depending on if the 4 second counter is finished(s_done1 = ‘1’) and the s_result register we used in the previous state of s_snap. If (s_result = ‘0’) and (s_done1 = ‘1’), then the next state becomes s_aftermathP. Else, if (s_result = ‘1’) and (s_done1 = ‘1’), the next state becomes s_aftermathN. Here, it is worth noting that transitioning from s_forward to s_aftermathP increments the count of approved tickets(s_permittemp) by integer 1 once during the process.

s_aftermathP: This state represents one of two possible final stages of the ticket scanning process which is the scanned ticket being unique and the resulting scan being successful. The outputs of this state include: the servo motor pointing left(s_servoswitch <= "11")-the same direction LEDs were pointing in s_advance- to let the person know that they are fine to proceed, the LED and buzzer configurations being set to the way they were back in s_advance, which were a right to left wave pattern(“001”) and the note C7(“1111”). This state also resets the 4 second one-shot counter(s_reset1 <= ‘1’) and initiates the countdown of the 8-second counter(s_reset2 <= ‘0’). There are two possible transitions out of this state: Turning the switch(sw) off and going back to s_off, or waiting for the 8-second countdown to complete(s_done2 = ‘1’) and going back to s_advance.

s_aftermathN: This state represents the second possible outcome of the ticket scanner, which is a duplicate or previously scanned ticket resulting in a negative scan. The outputs of this state are: The servo motor (s_servoswitch <= "10") and oscillating LED pattern(“010”) pointing towards right -the opposite of the direction in s_advance- meaning the person is not granted permission to proceed. The buzzer is set to note G6 and the indexes of the LEDs at which it turns on and off become 15 and 7 respectively. This state -just like s_aftermathP- resets the 4 second counter(s_reset1 <= ‘1’) and initiates the countdown of the 8-second counter(s_reset2 <= ‘0’). There also are two possible transitions out of this state: Turning the switch(sw) off and going back to s_off, or waiting for the 8-second countdown to complete(s_done2 = ‘1’) and going back to s_advance.

This marks the end of the design specifications section. The physical implementation of the design will be shown and discussed in the “Results” section of the report.

Methodology

The methodology of the project consisted of generating the idea and finding a way to bring the idea to life using what we learned throughout the term in EEE102. To this end, a number of possible combinations were considered before settling on the setup described in the design specifications section. The setup was meant to fulfill the following functions: Guide a person to a spot the camera can take an image from with reasonable precision and automation, take a picture and scan the QR code in the picture in an accurate manner, react in a way that reflects the validity of the QR code ticket and keep track of the previous tickets and react accordingly for repetitions of the previous three steps. These objectives were implemented in a design by using the following: The LEDs on the BASYS3, a servo motor attached to a cardboard arrow and a piezo buzzer for physical guidance; a debounced button as an initiation mechanism for the system; the receiving and transmitting functions of BASYS3’s UART connection to communicate with the laptop which both takes an image and performs the necessary computations regarding the ticket; and the seven-segment display of the BASYS3 to keep track of number of scanned valid tickets.

Results

The setup was constructed and trialed in a way that reflected all possible state routes, meaning starting off from s_off, going through all of the other states sequentially to end up on either s_aftermathP or s_aftermathN depending on the validity of the ticket, and finally shutting the switch down to see if the logic circuit performs its intended functions when it is turned off.

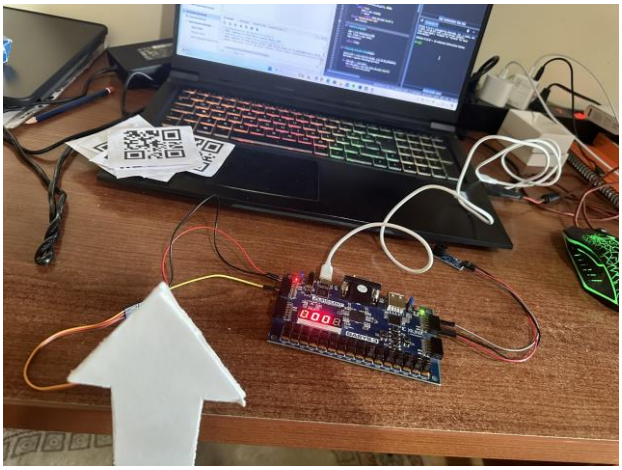

Figure 6:The setup in state "s_off"

The above image(Figure 6) represents the moment right after the BASYS3 is programmed with the bitstream file. As you can see, the value displayed on the 7-segment display (s_permittemp) is 0 and the arrow attached to the sg90 servo motor is in a constant state and pointing straight up (20ms signal with 7.5% duty cycle).

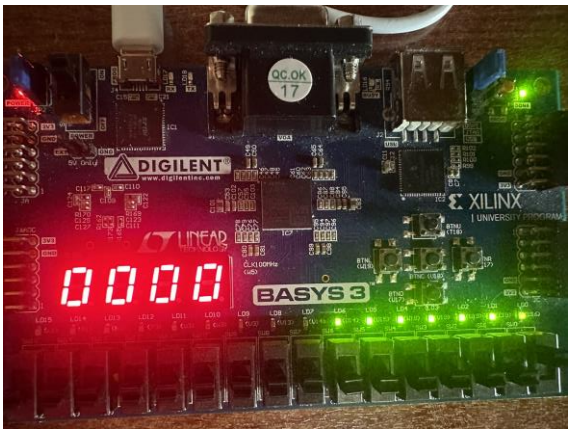

Figure 7: LED pattern in "s_advance"

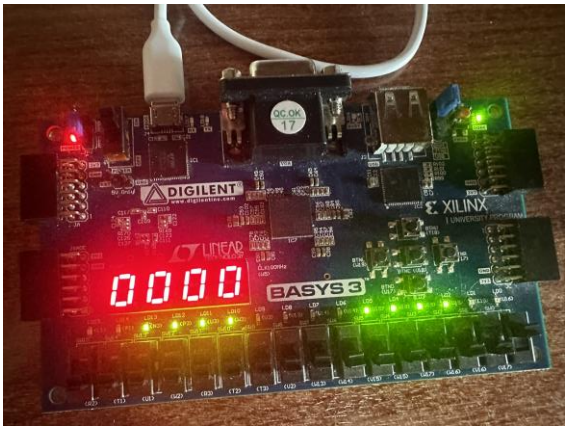

Figure 8 : LED pattern in "s_backwards"

The first image seen above(Figure 7) represents the moment after the switch is turned on (state s_aftermath). While not clearly identifiable due to the constant state of the image, the LEDs are generating a visual wave pattern that moves from the right side of the BASYS3 towards the left. The image right next to it(Figure 8) -on the other hand- represents the state right after the center button (btnc) is pressed and the state is transitioned to s_backwards, meaning that the LEDs move in a center-to-edges pattern. While still difficult to identify due to the pictures being constant, the differences between it and the right-to-left pattern seen in the first picture are distinguishable.

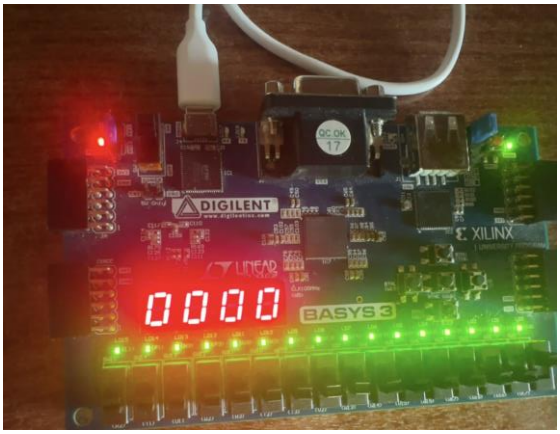

Figure 9: LED pattern in "s_snap"

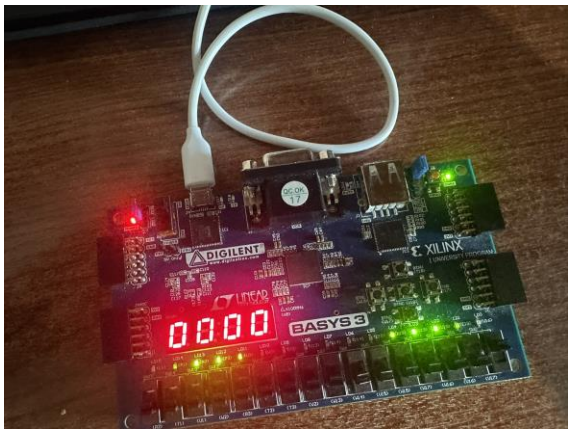

Figure 10 : LED pattern in "s_forward"

The image above on the left(Figure 9) displays a picture taken during the (s_snap) sequence, the point at which the LEDs turn on and off in unison and the "snap_" signal is sent. The image next to it(Figure 10) demonstrates the state right after it, which is s_forward. As is visible by the fluctuation of the LEDs, they are making a pattern that starts from the center of the bar towards the outer ends of it. Here -again- due to the pictures being constant images, the pattern made by the LEDs are not entirely distinguishable. However, they are different enough to identify that the state has transitioned.

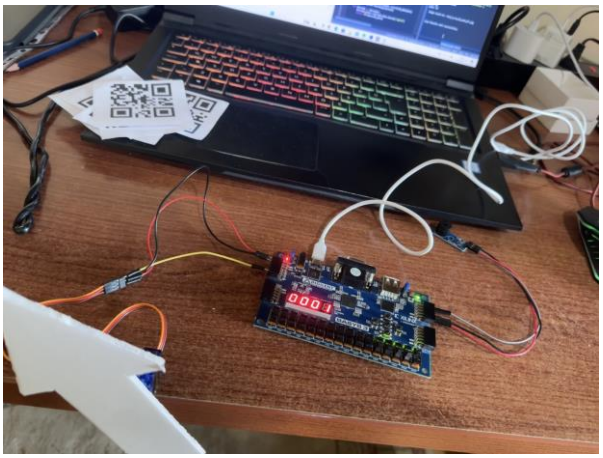

Figure 11: The setup in the state "s_aftermathP"

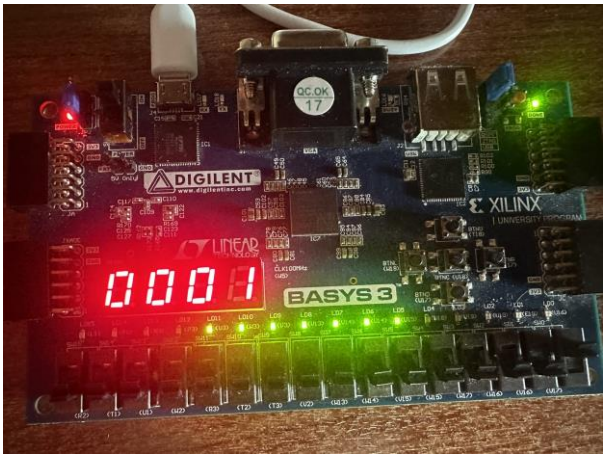

Figure 12: The setup after returning to "s_advance"

The two images above represent the states s_aftermathP and reverting back to s_advance respectively. In the picture to the left(Figure 11), you can see the changes the setup goes through after a successful ticket scan: the value on the 7-segment display has been incremented by 1, signifying a ticket has been scanned successfully, and the picture to the right(Figure 12) shows the return to s_advance. Again, the patterns the LEDs make are

indistinguishable due to the picture being constant. However, considering that s_advance and s_aftermathP share the same LED pattern, you can get an idea of the direction of motion by looking at the first picture and contrasting it with the second.

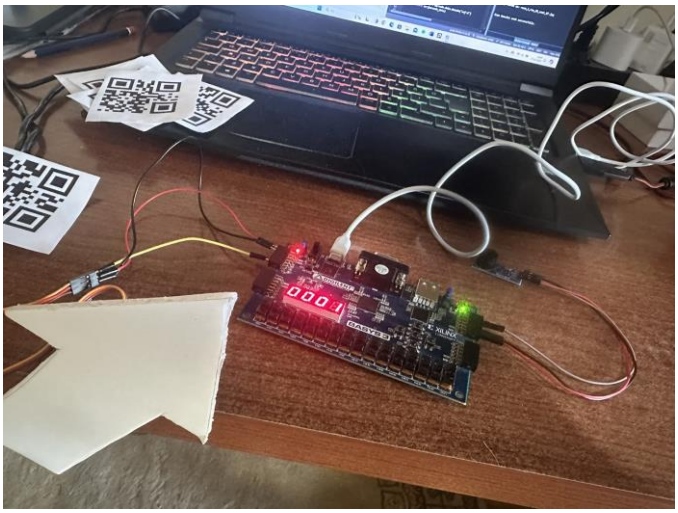

Figure 13: The setup in the state "s_aftermathN"

The image you see above(Figure 13) is of state "s_aftermathN" that took place after the previous "s_aftermathP" but before the turning of the switch and transitioning back to "s_off". As you can see, while the number of successfully scanned tickets is maintained, the arrow is now pointing towards the right which is telling the person that their ticket has been denied.

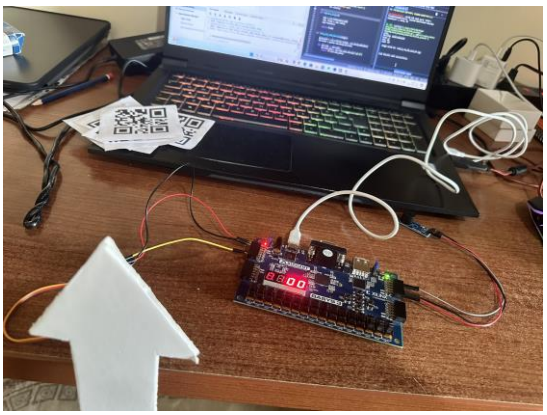

Figure 14: The setup after returning to "s_off"

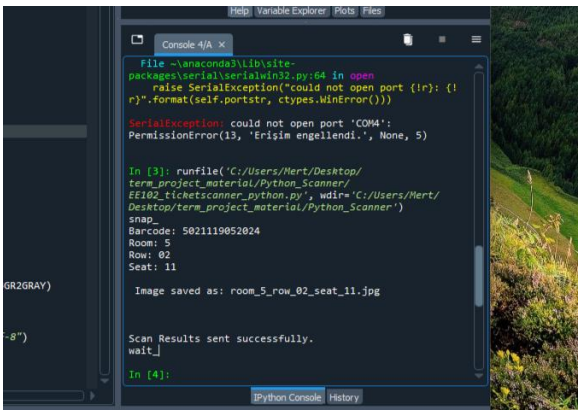

Figure 15: The Python kernel once returned to "s_off"

The images above represent the condition of the setup after the switch has been turned off and s_off has been re-initiated. On the left(Figure 14), you can see that -while the Python kernel in the background shows that a successful scan has taken place prior to that moment- the 7-segment display is displaying 0 due to s_permittemp being reverted back to 0. Also note that the arrow is stuck in a straight-up position. On the right(Figure 15), you can also see the Python kernel that is seen in the background of the picture on the left. While the Python code is not of concern from a logic-design standpoint, it has been configured to print the signals it receives. As is visible, prior to the “wait_” signal that has shut the program down, a “snap_” signal was transmitted to Python which initiated a seemingly successful barcode scan. The aftermath of that moment consists of the state “s_aftermathP”, implications of which were previously discussed.

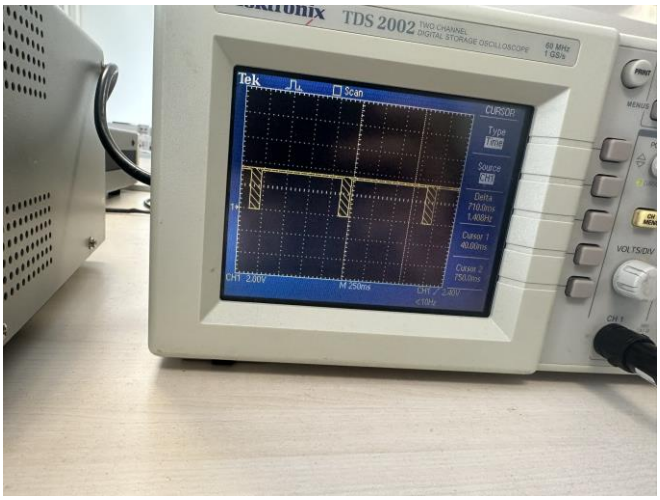

Figure 16: Oscilloscope results for buzzer I/O at “s_advance”

The image you see above(Figure 16) belongs to the I/O pin of the MH-FMD type piezo buzzer during the state “s_advance”. The first thing to note about this image is that the buzzer is of an active-low design. Another thing worth noting is the periodical differences between signal oscillations throughout the guidance cycle. As is visible, for short intervals during each period, the signal deviates from its high status and oscillates at a rate much higher than that of the sound frequency of the buzzer, which is indicated by the diagonal lines seen in the oscilloscope screen. For reference, the buzzer beeps once every LED cycle period which is a 1s period or a frequency of 1Hz and the note C7 has a much higher frequency at 2093 Hz(Songstuff).

Figure 17: Simulation results for oscillating servo signal

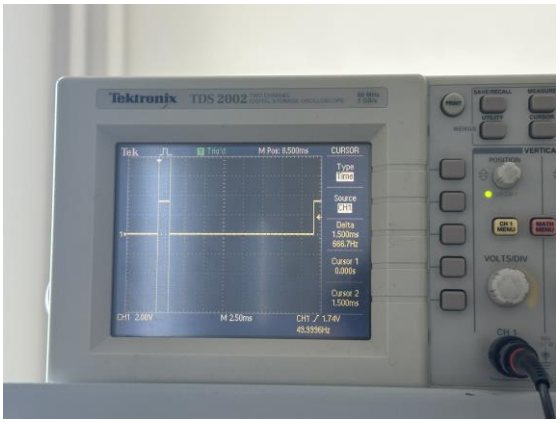

Figure 18: Oscilloscope results for 90-degree constant servo signal

The image seen above on the left(Figure 17) represents the testbench(servo_sim.vhd) results for the top module of the servo(servo_top.vhd). As can be seen by the time markers, the 20ms periodic signal that drives the servo progressively increases its duty cycle from 7.5%(1.5ms) to 8.4%(1.68ms) within 3 servo signal periods(3*20ms). The image above on the right(Figure 18) belongs to an oscilloscope connected to the servo pin during the state "s_off", meaning the servo receives a constant signal with a 7.5% duty cycle(1.5ms) that commands the servo to point upwards in a 90-degree direction with the normal of the surface. The period of the high signal is highlighted by the cursors.

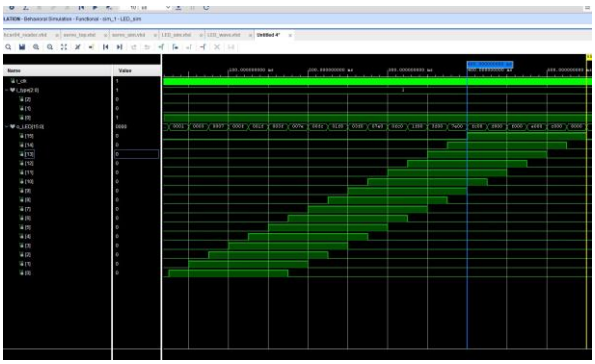

Figure 19: Simulation results for right-to-left LED signal

Figure 20: Simulation results for corner-to-center LED signal

The images above represent the testbench(LED_sim.vhd) simulation results for the LED wave generator module(LED_wave.vhd). The image on the left(Figure 19) belongs to the right-to-left oscillation pattern("001") used as a state-dependent output active on states "s_advance" and "s_aftermathP". The image on the right(Figure

20) belongs to the edge-to-center LED wave meant to indicate coming forward(“011”), which is also a state-dependent output found at state “s_forward”. In the image on the left(Figure 19), you can see that the rightmost LED of the BASYS3 is activated first, and LEDs are activated and shut down going from the right to the left. The second image(Figure 20) also demonstrates a similar phenomenon, except this time the order is from the outer edges to the center of the BASYS3’s LEDs. In both simulations, the turn-on period of each LED is 150ms which matches the design specifications of 0.15s. Other LED signal patterns, such as left-to-right or center to outer edges, are similar to these patterns except they are symmetrical reflections of these images with respect to the vertical axis.

This marks the end of the “Results” section. Due to the FSM in question having more dynamic design aspects than static ones, simple images may not be sufficient to give you an idea of how the system operates. To have a better understanding of how the system works, you can watch the YouTube presentation from the link found below the title, which demonstrates the system in action.

Conclusion

The purpose of this project was to construct a system that could guide a person to a specific spot upon initiation, photographically scan the ticket and react depending on the validity of said ticket. This was accomplished through designing a finite state machine and organizing the modules in a way that complemented the purpose of each stage. While -initially- the option of purchasing and operating a physical camera module rather than having the camera module as part of the computer meant to do the necessary scanning computations was considered, this plan was later cancelled due to available camera module components on the market being incompatible for UART communication due to the necessary rate of data transmission they require being far greater than that of a UART cable, which was going to be utilized either way. As a result, the idea of using an individual camera component as opposed to using the computer as the camera component was shelved. While it was the computer directly operating the camera as opposed to the BASYS3, it was the BASYS3 communicating to the computer what action to take at what time and it was the BASYS3 receiving information regarding the status of these tickets and reacting accordingly. As a result, the use of Python in this project is negligible compared to the amount of VHDL logic design that took place, as it was the BASYS3 that did most of the heavy lifting. Another

adjustment made to this project was the exclusion of the ultrasonic sensor due to practical issues. While an operating instance of the ultrasonic sensor was shown during the progress demo, when used in conjunction with the FSM specified in the design specifications, it proved to be unreliable in ways such as triggering at random at times when nobody was present or returning inconsistent signal outputs that did not match the distance it was scanning. These issues could have been exclusive to the specific sensor I acquired; however, it did not change the fact that using it would have damaged the precision and automation of the setup drastically. As a result, the idea of the ultrasonic sensor sensing when someone was standing in front of the setup was shelved in favor of using a debounced button, a much more reliable means of telling if someone was there and initiating the necessary state transitions. It was also mentioned in the proposal that the QR codes in question would be placed on a hat. However, such a placement would have no different implications on the functioning of the system in any meaningful form, so printed QR codes were utilized for the presentation and the report. This was also touched on during the video presentation. In the end, the system constructed for this project matched that of the capabilities of the one outlined in the proposal. Outside of these issues, the process of constructing this project went with minimum issues and was completed within the set timeframe. All the modules fulfilled their functionalities within their intended purposes and the finite state machine design -as far as trials went- was found to have no noticeable design errors. Based on these factors, this project could be considered successful as it fulfilled the core functionalities underlined in the proposal in a meaningful way. This project was also a good exercise in terms of time and resource management when constructing a system from ground up within a set amount of time. There is also a remark I would like to make about the video presentation: When scanning the 3rd QR code, I claim that the ticket I was scanning was one of the duplicates while it was not. This can actually be seen right before I proceed with that scan at the 2.30 mark. Each ticket has a corresponding number written at the back of it except for the duplicates, which also have the word “Duplicate” written below this number. If you look closely at the tickets seen at that point of the presentation, you can make out that while one of the tickets has a number and the word “Duplicate” written below it, the other just has a number, meaning it is not the duplicate of the other. I later correct this mistake by scanning the previous ticket again, which replicates what I initially intended to demonstrate successful.

Works Cited

Pieters, Achim. SG90 Servo – Pinout. 17 May 2022, www.studiopieters.nl/tower-pro-micro-servo-s9-sg90.

"Serial Communication Basic Knowledge -RS-232C/RS-422A/485- | CONTEC." *CONTEC*, www.contec.com/support/basic-knowledge/daq-control/serial-communicatin/#:~:text=Serial%20communication%20is%20a%20communication,one%20bit%20at%20a%20time.

Songstuff. "Music Note Fundamental Frequencies - Songstuff." Songstuff, 13 Jan. 2023, www.songstuff.com/recording/article/music_note_fundamental_frequencies.

"A UART Implementation in VHDL." Domipheus Labs, 15 Sept. 2015, domipheus.com/blog/a-uart-implementation-in-vhdl.

Appendices

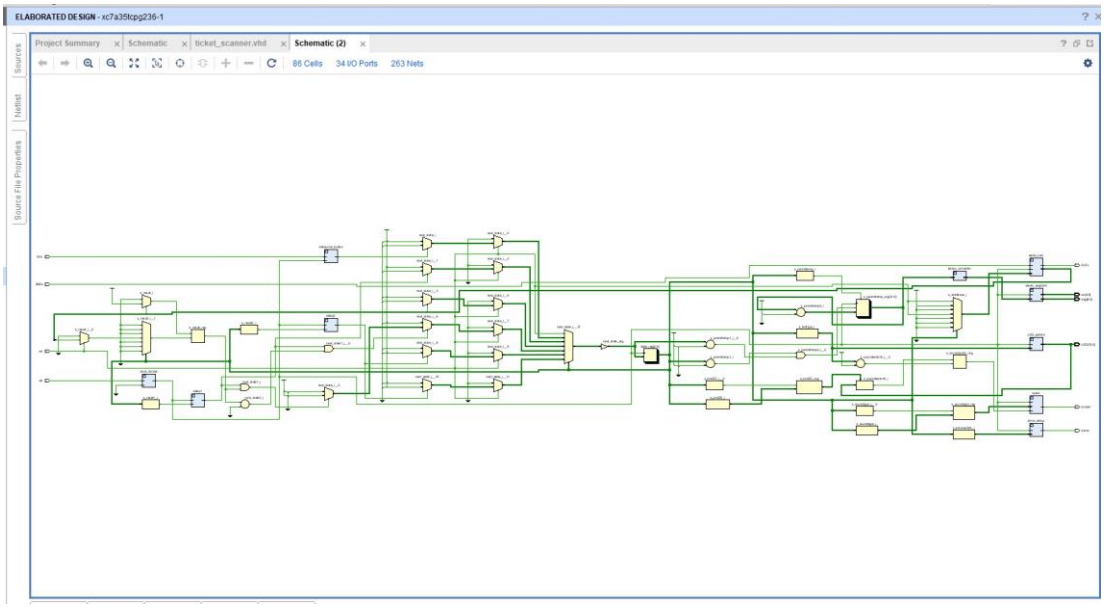

Figure 21: RTL schematic of the design with unexposed modules

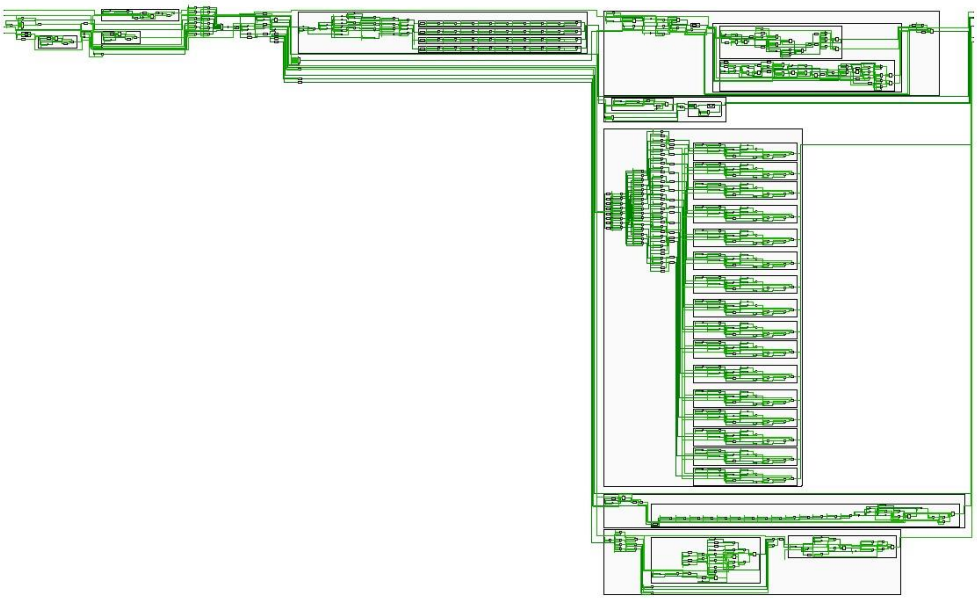

Figure 22: RTL schematic of the design with exposed modules

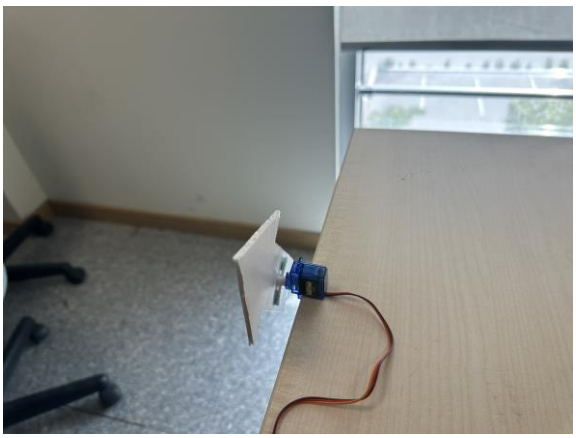

Figure 23: Placement of the SG90 servo motor

ticket_scanner.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity ticket_scanner is
    Port (clk: in std_logic ;
```

```
    sw: in std_logic  ;        --on/off switch
    btnc: in std_logic ;        --button to initiate process
    RsRx: in std_logic;     --UART Receiver
    RsTx: out std_logic ;     --UART Transmitter
    LED : out STD_LOGIC_VECTOR (15 downto 0);   --LEDs of BASYS 3
    servo: out std_logic ;    --sg90 servo
    buzzer: out std_logic ;       --MH-FMD buzzer
    seg: out std_logic_vector (6 downto 0) ;  --segments
    an : out std_logic_vector (3 downto 0) ); --anodes
end ticket_scanner;

architecture Behavioral of ticket_scanner is

component clk_wiz_0 is
Port (clk_in1 :in std_logic := '0';
reset : in std_logic := '0';
clk_out1: out std_logic := '0');
end component;

component LED_wave is
    Port ( i_clk : in STD_LOGIC;    --5MHz
           i_type : in STD_LOGIC_VECTOR (2 downto 0);
           o_LED : out STD_LOGIC_VECTOR (15 downto 0));
end component;

component servo_top is
    Port ( i_clk : in STD_LOGIC;  --5MHz
           i_sw : in std_logic_vector(1 downto 0);
           o_servosignal : out STD_LOGIC);
end component;




component sound_timer is
    Port ( i_clk : in std_logic := '0'; --5MHz
           i_switch : in STD_LOGIC := '0';
           i_soundtype : in STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
           o_signal : out STD_LOGIC := '1');
end component;

component seven_segment_display is
```

```
    Port (clk: in std_logic; --clock operating at 5MHz
    sw: in std_logic_vector (15 downto 0);
    seg: out std_logic_vector (6 downto 0);
    an : out std_logic_vector (3 downto 0)
     );
end component;



component top_serialcom is
Port (i_clk: in std_logic := '0';   --5MHz
     i_starttrans: in std_logic_vector(1 downto 0) := (others => '0');
     i_RsRx: in std_logic := '1';     --UART Receiver
     o_RsTx: out std_logic := '1';     --UART Transmitter
     o_ticketsignal: out std_logic_vector(1 downto 0) := (others => '0')
     );
end component;



component debouncer is
    Port (i_clk : in std_logic := '0';     --5MHz
    i_btn: in std_logic := '0';
    o_dbtn: out std_logic := '0');
end component;



component integer_to_binary is
    Port (i_integer:  in integer := 0;
    o_binary: out std_logic_vector( 15 downto 0) := (others => '0') );  --not a direct conversion, converts digits per 4 bits of output
end component;

component delay_counter is
    Port ( i_clk : in STD_LOGIC;
           i_reset : in STD_LOGIC;
           o_done : out STD_LOGIC);
end component;

component delay_counter2 is
    Port ( i_clk : in STD_LOGIC;
           i_reset : in STD_LOGIC;
           o_done : out STD_LOGIC);
end component;
```

```
signal s_clk5MHz: std_logic := '0';

--module inputs/outputs
signal s_ledtype: STD_LOGIC_VECTOR (2 downto 0);
signal s_servoswitch : std_logic_vector(1 downto 0) := (others => '0');
signal s_soundtype: STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
signal s_soundswitch: std_logic := '0';
signal s_digitbinary: std_logic_vector (15 downto 0):= (others => '0');
signal s_LED : STD_LOGIC_VECTOR (15 downto 0);
signal s_starttrans: std_logic_vector(1 downto 0) := "00";
signal s_ticketsignal: std_logic_vector(1 downto 0) := "00";
signal s_dbtn: std_logic := '0';

signal s_reset1: std_logic := '1';
signal s_done1: std_logic := '0';
signal s_reset2: std_logic := '1';
signal s_done2: std_logic := '0';



--signals
signal s_onLED: integer:= 0;
signal s_offLED: integer:= 0;
signal s_confirmed: std_logic_vector(2 downto 0) := (others => '0');  --i0 for first cont, i1 for horizontal conf., i3 for vertical conf
signal s_result: std_logic := '0'; --ticket confirmation status
signal s_permittemp: integer := 0;

--states
type states is (s_off, s_advance, s_backwards, s_snap, s_forward, s_aftermathP, s_aftermathN);
signal next_state, state: states := s_off;



begin

-------

LED <= s_LED;
-------
```

```
clock_divider: clk_wiz_0
Port Map (clk_in1 => clk,
reset => '0',
clk_out1 => s_clk5MHz);

LED_pattern: LED_wave
   Port Map ( i_clk => s_clk5MHz,    --5MHz
         i_type => s_ledtype,
         o_LED => s_LED );

servo_setup: servo_top
   Port Map ( i_clk => s_clk5MHz,  --5MHz
         i_sw => s_servoswitch,
         o_servosignal => servo);


audio:sound_timer
   Port Map ( i_clk => s_clk5MHz, --5MHz
         i_switch => s_soundswitch,
         i_soundtype => s_soundtype,
         o_signal => buzzer);

seven_segment: seven_segment_display
   Port Map (clk => s_clk5MHz, --clock operating at 5MHz
   sw => s_digitbinary,
   seg => seg,
   an => an
    );


serial_com: top_serialcom
Port Map (i_clk => s_clk5MHz,   --5MHz
    i_starttrans => s_starttrans,
    i_RsRx => RsRx,    --UART Receiver
    o_RsTx => RsTx,     --UART Transmitter
    o_ticketsignal => s_ticketsignal
    );


debounce_button: debouncer
   Port Map (i_clk => s_clk5MHz,     --5MHz
```

```
   i_btn => btnc,
   o_dbtn => s_dbtn);


binary_converter: integer_to_binary
Port Map (i_integer=> s_permittemp,
   o_binary => s_digitbinary);  --not a direct conversion, converts digits per 4 bits of output


delay1: delay_counter
Port Map ( i_clk => s_clk5MHz,
          i_reset => s_reset1,
          o_done => s_done1);


delay2: delay_counter2
Port Map ( i_clk => s_clk5MHz,
          i_reset => s_reset2,
          o_done => s_done2);




ticket_FSM: process (state, next_state, sw, s_dbtn, s_clk5MHz, s_ticketsignal, s_permittemp, s_LED, s_onLED, s_offLED, s_result,
s_ledtype, s_done1, s_done2)
begin


-------------------------------------------------------------------------------------------
--matching state transition to clock cycle
if rising_edge(s_clk5MHz) then
   state <= next_state;


   --counter of the seven segment display
   if (state = s_off) then
       s_permittemp <= 0;
   elsif (state = s_forward) and (next_state = s_aftermathP) then
       s_permittemp <= s_permittemp + 1;
   end if;


end if;



-------------------------------------------------------------------------------------------
case state is
-----------------------------------
```

```
    when s_off =>

        s_ledtype <= "000";
        s_servoswitch <= "01";
        s_starttrans <= "10";
        s_reset1 <= '1';
        s_reset2 <= '1';




        if (sw = '1') then
            next_state <= s_advance;
        else
            next_state <= s_off;
        end if;

-----------------------------------
    when s_advance =>

        s_starttrans <= "00";   --primed for next transmission
        s_servoswitch <= "00";  --oscillating in an uncertain manner



        s_reset1 <= '1';
        s_reset2 <= '1';



                s_ledtype <= "001"; --right to left wave
                s_onled <= 0;
                s_offled <= 8;
                s_soundtype <= "1111";



        if sw = '0' then
            next_state <= s_off;
        elsif s_dbtn = '1' then
            next_state <= s_backwards;
        else
            next_state <= s_advance;
        end if;
```

```
-----------------------------------
   when s_backwards =>



      s_servoswitch <= "00";  --oscillating in an uncertain manner
      s_starttrans <= "01";   --message selected
      s_reset1 <= '0'; --counter started
      s_reset2 <= '1';



      s_ledtype <= "100";     --backward
      s_onled <= 8;
      s_offled <= 12;
      s_soundtype <= "0101";




         if sw = '0' then
            s_starttrans <= "00";   --select other message
            next_state <= s_off;
         elsif (s_done1 = '1') then
            next_state <= s_snap;
         else
            next_state <= s_backwards;
         end if;
-----------------------------------
   when s_snap =>


      s_starttrans <= "11";   --take a picture


      s_ledtype <= "111";     --non-directional


      s_servoswitch <= "00";  --oscillating in an uncertain manner
      s_reset1 <= '1';  --counter reprimed
      s_reset2 <= '1';  --counter reprimed



       if (sw = '0') then
          s_starttrans <= "00";   --select other message
          next_state <= s_off;
```

```
        elsif (s_ticketsignal(0) = '1') then
                if (s_ticketsignal(1) = '1') then
                   s_result <= '0';
                   next_state <= s_forward;

                elsif (s_ticketsignal(1) = '0') then
                   s_result <= '1';
                   next_state <= s_forward;



                end if;

        else
           next_state <= s_snap;

        end if;
-----------------------------------
    when s_forward =>

       s_servoswitch <= "00";  --oscillating in an uncertain manner
       s_starttrans <= "00";    --reprimed




       s_reset1 <= '0'; --counter started
       s_reset2 <= '1';



              s_ledtype <= "011";     --forward
              s_onled <= 15;
              s_offled <= 11;
              s_soundtype <= "0001";



           if (sw = '0') then
              next_state <= s_off;
           elsif (s_done1 = '1') and (s_result = '0') then
              next_state <= s_aftermathP;

           elsif (s_done1 = '1') and (s_result = '1') then
              next_state <= s_aftermathN;
```

```
        else
            next_state <= s_forward;

        end if;
-----------------------------------
    when s_aftermathP =>


            s_servoswitch <= "11";  --pointing left
            s_starttrans <= "00";
            s_reset1 <= '1';
            s_reset2 <= '0';  --counter started

            s_ledtype <= "001"; --right to left wave
            s_onled <= 0;
            s_offled <= 8;
            s_soundtype <= "1111";



        if (sw = '0') then
            next_state <= s_off;
        elsif (s_done2 = '1') then
            next_state <= s_advance;
        else
            next_state <= s_aftermathP;
        end if;
-----------------------------------
    when s_aftermathN =>


            s_servoswitch <= "10";  --pointing right
            s_starttrans <= "00";

            s_reset1 <= '1';
            s_reset2 <= '0';  --counter started


            s_ledtype <= "010"; --left to right wave
            s_onled <= 15;
```

```
            s_offled <= 7;
            s_soundtype <= "1110";


        if (sw = '0') then
            next_state <= s_off;
        elsif (s_done2 = '1') then
            next_state <= s_advance;
        else
            next_state <= s_aftermathN;
        end if;

end case;


--buzzer reset adjuster
-------------------------------------------------------------------------------------------
if (s_ledtype /= "111") then    --111 is a blank state used for visual enchantment
   if (s_LED(s_onled) = '1') then
      s_soundswitch <= '1';

   elsif (s_LED(s_offled) = '1') then
      s_soundswitch <= '0';

else
    s_soundswitch <= '0';

   end if;
end if;
-------------------------------------------------------------------------------------------



end process;


end Behavioral;
```

LED_wave.vhd

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity LED_wave is
    Port ( i_clk : in STD_LOGIC;    --5MHz
           i_type : in STD_LOGIC_VECTOR (2 downto 0);
           o_LED : out STD_LOGIC_VECTOR (15 downto 0));
end LED_wave;

architecture Behavioral of LED_wave is

component LED_timer is
    Port (i_clk: in std_logic := '0';   --5 MHz
    i_delay: in integer := 0;  -- delay time = i_delay * 0.10 s
    i_trigger: in std_logic := '0';
    o_signal: out std_logic := '0'  --LED will turn on once for 0.20s once the delay time is over
    );
end component;


signal s_trigger: std_logic := '0';

signal s_delay0: integer := 0;
signal s_delay1: integer := 0;
signal s_delay2: integer := 0;
signal s_delay3: integer := 0;
signal s_delay4: integer := 0;
signal s_delay5: integer := 0;
signal s_delay6: integer := 0;
signal s_delay7: integer := 0;
signal s_delay8: integer := 0;
signal s_delay9: integer := 0;
signal s_delay10: integer := 0;
signal s_delay11: integer := 0;
signal s_delay12: integer := 0;
signal s_delay13: integer := 0;
signal s_delay14: integer := 0;
signal s_delay15: integer := 0;

begin

process(i_type)
```

```
begin

if (i_type = "000") then
    s_trigger <= '0';

elsif (i_type = "001") then-- left to right sweep, turn right indicator

s_delay0 <= 1;
s_delay1 <= 2;
s_delay2 <= 3;
s_delay3 <= 4;
s_delay4 <= 5;
s_delay5 <= 6;
s_delay6 <= 7;
s_delay7 <= 8;
s_delay8 <= 9;
s_delay9 <= 10;
s_delay10 <= 11;
s_delay11 <= 12;
s_delay12 <= 13;
s_delay13 <= 14;
s_delay14 <= 15;
s_delay15 <= 16;

s_trigger <= '1';

elsif (i_type = "010") then-- right to left sweep, turn left indicator

s_delay0 <= 16;
s_delay1 <= 15;
s_delay2 <= 14;
s_delay3 <= 13;
s_delay4 <= 12;
s_delay5 <= 11;
s_delay6 <= 10;
s_delay7 <= 9;
s_delay8 <= 8;
s_delay9 <= 7;
s_delay10 <= 6;
s_delay11 <= 5;
s_delay12 <= 4;
```

```
s_delay13 <= 3;
s_delay14 <= 2;
s_delay15 <= 1;

s_trigger <= '1';

elsif (i_type = "011") then-- inwards, forward indicator

s_delay0 <= 1;
s_delay1 <= 2;
s_delay2 <= 5;
s_delay3 <= 6;
s_delay4 <= 9;
s_delay5 <= 10;
s_delay6 <= 13;
s_delay7 <= 14;
s_delay8 <= 14;
s_delay9 <= 13;
s_delay10 <= 10;
s_delay11 <= 9;
s_delay12 <= 6;
s_delay13 <= 5;
s_delay14 <= 2;
s_delay15 <= 1;

s_trigger <= '1';

elsif (i_type = "100") then-- outwards, backward indicator

s_delay0 <= 14;
s_delay1 <= 13;
s_delay2 <= 10;
s_delay3 <= 9;
s_delay4 <= 6;
s_delay5 <= 5;
s_delay6 <= 2;
s_delay7 <= 1;
s_delay8 <= 1;
s_delay9 <= 2;
s_delay10 <= 5;
s_delay11 <= 6;
```

```
s_delay12 <= 9;
s_delay13 <= 10;
s_delay14 <= 13;
s_delay15 <= 14;

s_trigger <= '1';

else    --no direction applied

s_delay0 <= 8;
s_delay1 <= 8;
s_delay2 <= 8;
s_delay3 <= 8;
s_delay4 <= 8;
s_delay5 <= 8;
s_delay6 <= 8;
s_delay7 <= 8;
s_delay8 <= 8;
s_delay9 <= 8;
s_delay10 <= 8;
s_delay11 <= 8;
s_delay12 <= 8;
s_delay13 <= 8;
s_delay14 <= 8;
s_delay15 <= 8;

s_trigger <= '1';

end if;

end process;



LED_0 : LED_timer
   Port Map (i_clk => i_clk,   --5 MHz
   i_delay => s_delay0,  -- delay time = i_delay * 0.10 s
   i_trigger => s_trigger,
   o_signal => o_LED(0)  --LED will turn on once for 0.20s once the delay time is over
   );
```

```
LED_1 : LED_timer
   Port Map (i_clk => i_clk,   --5 MHz
   i_delay => s_delay1,  -- delay time = i_delay * 0.10 s
   i_trigger => s_trigger,
   o_signal => o_LED(1)  --LED will turn on once for 0.20s once the delay time is over
   );

LED_2 : LED_timer
   Port Map (i_clk => i_clk,   --5 MHz
   i_delay => s_delay2,  -- delay time = i_delay * 0.10 s
   i_trigger => s_trigger,
   o_signal => o_LED(2)  --LED will turn on once for 0.20s once the delay time is over
   );

LED_3 : LED_timer
   Port Map (i_clk => i_clk,   --5 MHz
   i_delay => s_delay3,  -- delay time = i_delay * 0.10 s
   i_trigger => s_trigger,
   o_signal => o_LED(3)  --LED will turn on once for 0.20s once the delay time is over
   );

LED_4 : LED_timer
   Port Map (i_clk => i_clk,   --5 MHz
   i_delay => s_delay4,  -- delay time = i_delay * 0.10 s
   i_trigger => s_trigger,
   o_signal => o_LED(4)  --LED will turn on once for 0.20s once the delay time is over
   );

LED_5 : LED_timer
   Port Map (i_clk => i_clk,   --5 MHz
   i_delay => s_delay5,  -- delay time = i_delay * 0.10 s
   i_trigger => s_trigger,
   o_signal => o_LED(5)  --LED will turn on once for 0.20s once the delay time is over
   );

LED_6 : LED_timer
   Port Map (i_clk => i_clk,   --5 MHz
   i_delay => s_delay6,  -- delay time = i_delay * 0.10 s
   i_trigger => s_trigger,
   o_signal => o_LED(6)  --LED will turn on once for 0.20s once the delay time is over
   );
```

```
LED_7 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay7,  -- delay time = i_delay * 0.10 s
    i_trigger => s_trigger,
    o_signal => o_LED(7)  --LED will turn on once for 0.20s once the delay time is over
    );

LED_8 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay8,  -- delay time = i_delay * 0.10 s
    i_trigger => s_trigger,
    o_signal => o_LED(8)  --LED will turn on once for 0.20s once the delay time is over
    );

LED_9 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay9,  -- delay time = i_delay * 0.10 s
    i_trigger => s_trigger,
    o_signal => o_LED(9)  --LED will turn on once for 0.20s once the delay time is over
    );

LED_10 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay10,  -- delay time = i_delay * 0.10 s,
    i_trigger => s_trigger,
    o_signal => o_LED(10)  --LED will turn on once for 0.20s once the delay time is over
    );

LED_11 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay11,  -- delay time = i_delay * 0.10 s
    i_trigger => s_trigger,
    o_signal => o_LED(11)  --LED will turn on once for 0.20s once the delay time is over
    );

LED_12 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay12,  -- delay time = i_delay * 0.10 s
    i_trigger => s_trigger,
    o_signal => o_LED(12)  --LED will turn on once for 0.20s once the delay time is over
```

```
    );

LED_13 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay13,  -- delay time = i_delay * 0.10 s
    i_trigger => s_trigger,
    o_signal => o_LED(13)  --LED will turn on once for 0.20s once the delay time is over
    );

LED_14 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay14,  -- delay time = i_delay * 0.10 s
    i_trigger => s_trigger,
    o_signal => o_LED(14)  --LED will turn on once for 0.20s once the delay time is over
    );

LED_15 : LED_timer
    Port Map (i_clk => i_clk,   --5 MHz
    i_delay => s_delay15,  -- delay time = i_delay * 0.10 s
    i_trigger => s_trigger,
    o_signal => o_LED(15)  --LED will turn on once for 0.20s once the delay time is over
    );

end Behavioral;
```

LED_timer.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity LED_timer is
    Port (i_clk: in std_logic := '0';   --5MHz
    i_delay: in integer := 0;  -- delay time = i_delay * 0.05 s
    i_trigger: in std_logic := '0';
    o_signal: out std_logic := '0'  --LED will turn on once for 0.30s once the delay time is over
    );
end LED_timer;

architecture Behavioral of LED_timer is
```

```
signal s_temp : integer := 0;

begin

process (i_clk, i_delay, i_trigger)
begin

if (i_trigger = '0') then
   s_temp <= 0;
   o_signal <= '0';

else
      if rising_edge(i_clk) then
         if (s_temp = (5000000 - 1)) then  --1s for all 16 LEDs to finish their cycles
            o_signal <= '0';
            s_temp <= 0;
         elsif (s_temp < (i_delay * 125000)) then
            o_signal <= '0';
            s_temp <= s_temp + 1;
         elsif ((s_temp < 7500000 ) AND (s_temp < i_delay * 125000 + 750000 )) then
            o_signal <= '1';
            s_temp <= s_temp + 1;
         else
            o_signal <= '0';
            s_temp <= s_temp + 1;
         end if;

      end if;
end if;
end process;

end Behavioral;
```

servo_top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;

entity servo_top is
```

```
    Port ( i_clk : in STD_LOGIC;  --5MHz
           i_sw : in std_logic_vector(1 downto 0);
           o_servosignal : out STD_LOGIC);
end servo_top;

architecture Behavioral of servo_top is

component sonar_counter is
    Port (i_clk: in std_logic;  --100MHz default
    i_res: std_logic := '0';
    o_waveperiod: out integer range 100 to 200);
end component;

component sg90_servo_signal_adjuster is
    Port ( i_clk : in STD_LOGIC;                    -- assumed to be 100MHz as is std Basys 3 clk
    i_res : in std_logic;                           -- reset
    i_high: in integer range 100 to 200 := 100;         --high period: i_high/100
    o_clk: out std_logic);
end component;


signal s_wave: integer range 100 to 200 := 150 ;
--signal s_wavetemp: integer := 0 ;
signal s_high: integer range 100 to 200 := 150 ;
signal s_res1 :std_logic := '0';
signal s_res2 :std_logic := '0';

begin


servo: sg90_servo_signal_adjuster
    Port Map ( i_clk => i_clk,        --5MHz
    i_res => s_res1,                            -- reset
    i_high => s_high,      --high period: i_high/100
    o_clk => o_servosignal);


sonar_adj: sonar_counter
    Port Map (i_clk => i_clk,  --5MHz
    i_res => s_res2,
    o_waveperiod => s_wave);
```

```
process(s_wave,i_sw)
begin
   if (i_sw = "00") then
      s_high <= s_wave;
      s_res2 <= '0';
   elsif (i_sw = "01") then    --off
      s_high <= 150;
      s_res2 <= '1';
   elsif (i_sw = "10") then    --negative
      s_high <= 100;
      s_res2 <= '1';
   elsif (i_sw = "11") then    --positive
      s_high <= 200;
      s_res2 <= '1';
   end if;

end process;
end Behavioral;
```

sg90_servo_signal_adjuster.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity sg90_servo_signal_adjuster is
   Port ( i_clk : in STD_LOGIC;                  -- clock divided to 5MHz
   i_res : in std_logic;                         -- reset
   i_high: in integer range 100 to 200 := 100;         --high period: i_high/100
   o_clk: out std_logic);
end sg90_servo_signal_adjuster;

architecture Behavioral of sg90_servo_signal_adjuster is

signal s_counttemp: integer := 0;
signal s_countfixed: integer := 0;  -- 50 Hz
signal s_countselected: integer := 0;
```

```
signal s_temp: std_logic := '0';

begin

s_countselected <= (50*i_high) ;
s_countfixed <= 100000 - 1 ;

process(i_clk, i_res)
begin
   if (i_res = '1') then
      s_counttemp <= 0;
      s_temp <= '0';
   elsif rising_edge(i_clk) then
      if (s_counttemp = s_countselected) then
         s_temp <= '0';
      elsif (s_counttemp = s_countfixed) then
         s_counttemp <= 0;
         s_temp <= '0';
      elsif (s_counttemp < s_countselected) then
         s_temp <= '1';
      end if;

      if (s_counttemp /= s_countfixed) then
         s_counttemp <= s_counttemp + 1;
      end if;
   end if;
end process;

o_clk <= s_temp;

end Behavioral;
```

sonar_counter.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity sonar_counter is
   Port (i_clk: in std_logic;  --clock divided 5Mhz
   i_res: std_logic := '0';
```

```
    o_waveperiod: out integer range 100 to 200);
end sonar_counter;

--o_waveperiod will oscillate between 100 and 200 with a period of 8s, therefore a frequency of 12.5 Hz per increment

architecture Behavioral of sonar_counter is

signal s_temp: integer := 0;
signal s_modclk: std_logic := '0';
signal s_tempwaveperiod: integer range 100 to 200 := 150;
signal s_indicator: std_logic := '0';   -- '0' for increasing and '1' for decreasing direction

begin

o_waveperiod <= s_tempwaveperiod;

process(i_clk, i_res, s_modclk)
begin
if (i_res = '1') then
   s_temp <= 0;
   s_modclk <= '0';
   s_tempwaveperiod <= 150;

elsif rising_edge(i_clk) then
         if (s_temp = (20000-1)) then          -- 100MHz / (400,000 * 2) = 250 Hz
            s_modclk <= NOT(s_modclk);
            s_temp <= 0;
         else
            s_temp <= s_temp + integer(1);
         end if;
end if;

if rising_edge(s_modclk) then   -- drop to 25Hz
   if (s_indicator='0') then
      if (s_tempwaveperiod < 200) then
         s_tempwaveperiod <= s_tempwaveperiod + 1;
      elsif (s_tempwaveperiod = 200) then
         s_tempwaveperiod <= s_tempwaveperiod - 1;
         s_indicator <= '1';
      end if;
   elsif (s_indicator='1') then
```

```
        if (s_tempwaveperiod > 100) then
            s_tempwaveperiod <= s_tempwaveperiod - 1;
        elsif (s_tempwaveperiod = 100) then
            s_tempwaveperiod <= s_tempwaveperiod + 1;
            s_indicator <= '0';
        end if;
    end if;
end if;
end process;



end Behavioral;
```

sound_mux.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;



entity sound_timer is
    Port ( i_clk : in std_logic := '0'; --5MHz
           i_switch : in STD_LOGIC := '0';
           i_soundtype : in STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
           o_signal : out STD_LOGIC := '1');
end sound_timer;

architecture Behavioral of sound_timer is

component buzzer_passive is
    Port (
        i_clk: in std_logic := '0'; --5MHz
        i_switch : in STD_LOGIC := '0';
        i_type: in std_logic_vector (3 downto 0) := (others => '0');
        o_buzzsignal : out STD_LOGIC := '0'
    );
end component;




signal s_counttemp: integer := 0;
signal s_buzz : STD_LOGIC := '0';
```

```
begin

buzzer:buzzer_passive
   Port Map (
      i_clk => i_clk, --5MHz
      i_switch => s_buzz,
      i_type => i_soundtype,
      o_buzzsignal => o_signal
   );

process(i_switch, i_clk, s_counttemp)
begin

if (i_switch = '0') then
   s_buzz <= '0';
   s_counttemp <= 0;

elsif (i_switch = '1') then
   if rising_edge(i_clk) then
      if (s_counttemp = 500000) then
         s_buzz <= '0';

      else
         s_buzz <= '1';
         s_counttemp <= s_counttemp + 1;

      end if;
   end if;
end if;
end process;



end Behavioral;
```

buzzer_passive.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
```

```
entity buzzer_passive is
    Port (
        i_clk: in std_logic := '0'; --5MHz
        i_switch : in STD_LOGIC := '0';
        i_type: in std_logic_vector (3 downto 0) := (others => '0');
        o_buzzsignal : out STD_LOGIC := '1'
    );
end buzzer_passive;

architecture Behavioral of buzzer_passive is

signal s_counttemp: integer := 0;
signal s_countmax: integer := 0;



begin



process(i_clk, i_switch, i_type)
begin
if (i_type = "0000") then --A
    s_countmax <= 5682;
elsif (i_type = "0001") then --B
    s_countmax <= 5062;
elsif (i_type = "0010") then --C
    s_countmax <= 9555;
elsif (i_type = "0011") then --D
    s_countmax <= 8513;
elsif (i_type = "0100") then --E
    s_countmax <= 10433;
elsif (i_type = "0101") then --F
    s_countmax <= 7159;
elsif (i_type = "0110") then --A#/Bb
    s_countmax <= 5363;
elsif (i_type = "0111") then --G#/Ab
    s_countmax <= 6020;
elsif (i_type = "1000") then --G
    s_countmax <= 6378;
elsif (i_type = "1001") then --F#/Gb
    s_countmax <= 6757;
```

```
elsif (i_type = "1010") then --D#/Eb
    s_countmax <= 8035;
elsif (i_type = "1011") then --C#/Db
    s_countmax <= 9019;
elsif (i_type = "1100") then --C5
    s_countmax <= 4778;
elsif (i_type = "1101") then --C6
    s_countmax <= 2389;
elsif (i_type = "1110") then --G6
    s_countmax <= 1896;
elsif (i_type = "1111") then --C7
    s_countmax <= 1194;

else
    s_countmax <= 5682; --A is default

end if;

if (i_switch = '0') then
     s_counttemp <= 0;
     o_buzzsignal <= '1';   --buzzer is silent when the signal is constant
elsif rising_edge(i_clk) then
   if (s_counttemp = s_countmax / 2) then
      o_buzzsignal <= '1';
   elsif (s_counttemp = s_countmax) then
      s_counttemp <= 0;
      o_buzzsignal <= '1';
   elsif (s_counttemp < s_countmax / 2) then
      o_buzzsignal <= '0';
   end if;

   if (s_counttemp /= s_countmax) then
      s_counttemp <= s_counttemp + 1;
   end if;
end if;
end process;

end Behavioral;
```

display_top.vhd

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity seven_segment_display is
    Port (clk: in std_logic; --clock operating at 5MHz
    sw: in std_logic_vector (15 downto 0);
    seg: out std_logic_vector (6 downto 0);
    an : out std_logic_vector (3 downto 0)
     );
end seven_segment_display;

architecture Behavioral of seven_segment_display is



component clock_divide is
    Port (i_clock : in  STD_LOGIC;
        i_res  : in  STD_LOGIC;
        o_clock: out STD_LOGIC);
end component;


component digit_mux is
    Port (i_s: in std_logic_vector(1 downto 0);
    i_num : in std_logic_vector (15 downto 0);
    o_data : out std_logic_vector (10 downto 0));
end component;


signal current_number: std_logic_vector (15 downto 0);
signal clk_2: std_logic;
signal lednum: std_logic_vector (1 downto 0);
signal segan: std_logic_vector(10 downto 0);


begin

current_number <= sw;

clock_divider: clock_divide
    Port Map (i_clock => clk,
```

```
        i_res  => '0',
        o_clock => clk_2);



process(clk_2)
begin
  if rising_edge(clk_2) then
      case lednum is
        when "00" =>
          lednum <= "01";
        when "01" =>
          lednum <= "10";
        when "10" =>
          lednum <= "11";
        when "11" =>
          lednum <= "00";
        when others =>
          lednum <= "00";
        end case;
   end if;
end process;




digit_multiplex: digit_mux
    Port Map (i_s => lednum,
    i_num => current_number,
    o_data => segan);

an(3 downto 0) <= segan (10 downto 7);
seg(6 downto 0) <= segan(6 downto 0);

end Behavioral;
```

clock_divide.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity clock_divide is
    Port (i_clock : in STD_LOGIC;
    i_res : in STD_LOGIC;
    o_clock: out STD_LOGIC);
```

```
end clock_divide;

architecture Behavioral of clock_divide is
    signal s_tempmem: STD_LOGIC;
    signal s_count : integer := 0;

begin

process (i_clock, i_res)
begin
if (i_res = '1') then
    s_count <= integer(0);
    s_tempmem <= '0';

elsif rising_edge(i_clock) then
    if (s_count = 12500) then -- 5MHz / (12,500 * 2) = 200 Hz
        s_tempmem <= NOT(s_tempmem);
        s_count <= 0;

    else
        s_count <= s_count + integer(1);



    end if;
end if;

end process;

o_clock <= s_tempmem;
end Behavioral;
```

led_decoder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity led_decoder is
    Port (i_dtype: in std_logic_vector(3 downto 0);
    o_segdata: out std_logic_vector(6 downto 0));
end led_decoder;
```

```
architecture Behavioral of led_decoder is

begin

   with i_dtype select
      o_segdata <= "1000000" when "0000",--0
                   "1111001" when "0001",--1
                   "0100100" when "0010",--2
                   "0110000" when "0011",--3
                   "0011001" when "0100",--4
                   "0010010" when "0101",--5
                   "0000010" when "0110",--6
                   "1111000" when "0111",--7
                   "0000000" when "1000",--8
                   "0010000" when "1001",--9
                   "1111111" when others;--all other possibilities

end Behavioral;
```

digit_mux.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity digit_mux is
   Port (i_s: in std_logic_vector(1 downto 0);
   i_num : in std_logic_vector (15 downto 0);
   o_data : out std_logic_vector (10 downto 0));
end digit_mux;

architecture Behavioral of digit_mux is

component led_decoder is
   Port (i_dtype: in std_logic_vector(3 downto 0);
   o_segdata: out std_logic_vector(6 downto 0));
end component;

signal s_digit: std_logic_vector (3 downto 0);
signal s_segdata: std_logic_vector (6 downto 0);
```

```
signal s_an: std_logic_vector (3 downto 0);

begin

   with i_s select
      s_digit <= i_num (3 downto 0) when "00",
                 i_num (7 downto 4) when "01",
                 i_num (11 downto 8)  when "10",
                 i_num (15 downto 12)  when "11",
                 "1111" when others;

decoder: led_decoder
   Port Map (i_dtype => s_digit,
   o_segdata => s_segdata );

with i_s select
   s_an (3 downto 0) <= "1110" when "00",
        "1101" when "01",
        "1011" when "10",
        "0111" when "11",
        "1111" when others;

o_data(6 downto 0) <= s_segdata (6 downto 0);
o_data(10 downto 7) <= s_an (3 downto 0);

end Behavioral;
```

top_serialcom.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity top_serialcom is
Port (i_clk: in std_logic := '0';   --5MHz
    i_starttrans: in std_logic_vector(1 downto 0) := (others => '0');
    i_RsRx: in std_logic := '1';     --UART Receiver
    o_RsTx: out std_logic := '1';     --UART Transmitter
```

```
    o_ticketsignal: out std_logic_vector(1 downto 0) := (others => '0')
    );
end top_serialcom;

architecture Behavioral of top_serialcom is

component ascii_to_binary_mux is
   Port ( i_sw : in STD_LOGIC_VECTOR (2 downto 0);
          o_binary : out STD_LOGIC_VECTOR (7 downto 0));
end component;

component UART_top is
generic(c_freq: integer := 5000000; --5MHz
c_baud: integer := 9600;
c_stopbit: integer := 1
);

Port(i_clk: in std_logic := '0';
i_receiver: in std_logic := '1';
i_datatrans: in std_logic_vector(7 downto 0) := (others => '1');
i_starttrans: in std_logic := '0';
o_transmitter: out std_logic := '1';
o_datareceived: out std_logic_vector(7 downto 0) := (others => '0');
o_transdone: out std_logic := '0';
o_receivedone: out std_logic := '0'
);

end component;




signal s_datatrans: std_logic_vector(7 downto 0) := (others => '1');
signal s_starttrans: std_logic := '0';
signal s_datareceived: std_logic_vector(7 downto 0) := (others => '0');
signal s_transdone:  std_logic := '0';
signal s_receivedone:  std_logic := '0';

signal s_asciisw: std_logic_vector (2 downto 0) := (others => '0');
signal s_lettercounter: integer := 0;
signal s_swshiftreg: std_logic_vector(14 downto 0) := (others => '0');
```

```
-------------------------------------------------------------------------------------------------------------------
--constants for UART
constant c_freq: integer := 5000000; --5MHz
constant c_baud: integer := 9600;
constant c_stopbit: integer := 1;

begin

-------------------------------------------------------------------------------------------------------------------



UART_bothways: UART_top
generic map(c_freq => c_freq, --5MHz
c_baud => c_baud,
c_stopbit => c_stopbit
)

Port Map(i_clk => i_clk,
i_receiver => i_RsRx,
i_datatrans => s_datatrans,
i_starttrans => s_starttrans,
o_transmitter => o_RsTx,
o_datareceived => s_datareceived,
o_transdone => s_transdone,
o_receivedone => s_receivedone
);

converter:ascii_to_binary_mux
   Port Map ( i_sw => s_asciisw,
         o_binary => s_datatrans);

-------------------------------------------------------------------------------------------------------------------

receive_data:process(s_receivedone, i_starttrans)
begin

if (i_starttrans(1) = '1') then
   if rising_edge(s_receivedone) then
   --least significant bit of o_ticketsignal demonstrates if it is a positive return signal, most significant demonstartes type
      if (s_datareceived (6 downto 0) = "1000101") then    --e, LS 6 digits / out of 7
```

```
            o_ticketsignal <= "11";
        elsif (s_datareceived (6 downto 0) = "1010010") then --r, LS 6 digits / out of 7
            o_ticketsignal <= "01";
        end if;
    end if;
else
    o_ticketsignal <= "00";
end if;


end process;
-------------------------------------------------------------------------------------------------------------


s_asciisw <= s_swshiftreg(2 downto 0);

send_text:process(i_starttrans,s_lettercounter,s_transdone)
begin

if (i_starttrans(1) = '0') then
    s_starttrans <= '0';
    s_lettercounter <= 0;

    if (i_starttrans(0) = '0') then
            s_swshiftreg <= "000100011010001";  --_tiaw
    elsif (i_starttrans(0) = '1')then
            s_swshiftreg <= "000111010110101";  --_pans
    else
            s_swshiftreg <= (others => '0');    --____ is default
    end if;



else

    if (s_lettercounter = 5) then
        s_starttrans <= '0';
    else
        s_starttrans <= '1';

        if rising_edge(s_transdone) then
```

```
            s_lettercounter <= s_lettercounter + 1;
            s_swshiftreg(11 downto 0) <= s_swshiftreg(14 downto 3);
        end if;
    end if;


end if;




end process;

end Behavioral;
```

UART_top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity UART_top is
generic(c_freq: integer := 5000000; --5MHz
c_baud: integer := 9600;
c_stopbit: integer := 1
);

Port(i_clk: in std_logic := '0';
i_receiver: in std_logic := '1';
i_datatrans: in std_logic_vector(7 downto 0) := (others => '1');
i_starttrans: in std_logic := '0';
o_transmitter: out std_logic := '1';
o_datareceived: out std_logic_vector(7 downto 0) := (others => '0');
o_transdone: out std_logic := '0';
o_receivedone: out std_logic := '0'
);
```

```
end UART_top;

architecture Behavioral of UART_top is

component UART_transmitter is

generic(c_freq: integer := 5000000; --5MHz
    c_baud: integer := 9600;
    c_stopbit: integer := 2
    );

Port (i_clk: in std_logic := '0';
    i_start: in std_logic := '0';
    i_data: in std_logic_vector (7 downto 0) := (others => '1');
    o_done: out std_logic := '0';
    o_dataout: out std_logic := '1'
    );
end component;   --baud rate set at 9600

component UART_receiver is
generic(c_freq: integer := 5000000; --5MHz
    c_baud: integer := 9600
    );

Port (i_clk: in std_logic := '0'; --5MHz
     i_datain : in std_logic := '1';
     o_dataout: out std_logic_vector (7 downto 0) := (others => '1');
     o_done : out std_logic := '0');
end component;

begin
--module assignments
-------------------------------------------------------------------------------
transmitter: UART_transmitter
generic map(c_freq => c_freq, --5MHz
    c_baud => c_baud,
```

```
    c_stopbit => c_stopbit
    )

Port map(i_clk => i_clk,
    i_start => i_starttrans,
    i_data => i_datatrans,
    o_done => o_transdone,
    o_dataout => o_transmitter
    );

receiver: UART_receiver
generic map(c_freq => c_freq, --5MHz
    c_baud => c_baud
    )

Port Map (i_clk => i_clk, --5MHz
     i_datain => i_receiver,
     o_dataout => o_datareceived,
     o_done => o_receivedone
);

end Behavioral;
```

UART_transmitter.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity UART_transmitter is

generic(c_freq: integer := 5000000; --5MHz
    c_baud: integer := 9600;
    c_stopbit: integer := 1
    );

Port (i_clk: in std_logic := '0';
    i_start: in std_logic := '0';
    i_data: in std_logic_vector (7 downto 0) := (others => '1');
    o_done: out std_logic := '0';
```

```
    o_dataout: out std_logic := '1'
    );
end UART_transmitter;   --baud rate set at 9600

architecture Behavioral of UART_transmitter is

type states is (s_idle,s_start,s_data,s_stop);

constant c_maxbittimer : integer := c_freq/(c_baud);
constant c_maxstopbit : integer := c_maxbittimer * c_stopbit;



signal state: states := s_idle;

signal s_shiftreg: std_logic_vector(7 downto 0) := (others => '0');
signal s_bittimer: integer range 0 to c_maxstopbit := 0;  --520 = 5*10^6 / 9600*2
signal s_bitno: integer range 0 to 7 := 0;

begin

transmitter: process(i_clk, state, i_start, s_bittimer, s_bitno)
begin
if rising_edge(i_clk) then
    case state is

        when s_idle =>

            o_done <= '0';
            o_dataout <= '1';
            s_bitno <= 0;

            if (i_start = '1') then
                s_shiftreg <= i_data;
                o_dataout <= '0';
                state <= s_start;
            end if;

        when s_start =>

            if (s_bittimer = c_maxbittimer - 1) then
                state <= s_data;
```

```
        o_dataout <= s_shiftreg(0);
        s_shiftreg(6 downto 0) <= s_shiftreg(7 downto 1);
        s_shiftreg(7) <= s_shiftreg(0);     --shift right
        s_bittimer <= 0;

      else

        s_bittimer <= s_bittimer + 1;

    end if;

when s_data =>

    if (s_bitno = 7) then
        if (s_bittimer = c_maxbittimer - 1) then
                s_bittimer <= 0;
                s_bitno <= 0;
                state <= s_stop;
                o_dataout <= '1';
            else
                 s_bittimer <= s_bittimer + 1;
            end if;

    else
            if (s_bittimer = c_maxbittimer - 1) then
                s_shiftreg(6 downto 0) <= s_shiftreg(7 downto 1);
                s_shiftreg(7) <= s_shiftreg(0);
                o_dataout <= s_shiftreg(0);
                s_bitno <= s_bitno + 1;
                s_bittimer <= 0;
            else
                 s_bittimer <= s_bittimer + 1;
            end if;

    end if;

when s_stop =>
     if (s_bittimer = c_maxstopbit - 1) then
            state <= s_idle;
            s_bittimer <= 0;
            o_done <= '1';
```

```
            else
               s_bittimer <= s_bittimer + 1;
            end if;
    end case;
end if;
end process;

end Behavioral;
```

UART_receiver.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;




entity UART_receiver is
generic(c_freq: integer := 5000000; --5MHz
    c_baud: integer := 9600
);

Port (i_clk: in std_logic := '0'; --5MHz
    i_datain : in std_logic := '1';
    o_dataout: out std_logic_vector (7 downto 0) := (others => '1');
    o_done : out std_logic := '0');
end UART_receiver;

architecture Behavioral of UART_receiver is

type states is (s_idle,s_start,s_data,s_stop);
signal state: states := s_idle;

constant c_maxbittimer : integer := c_freq/(c_baud);

signal s_shiftreg: std_logic_vector(7 downto 0) := (others => '0');
signal s_bittimer: integer range 0 to c_maxbittimer := 0;
signal s_bitno: integer range 0 to 7 := 0;

begin

o_dataout <= s_shiftreg;
```

```
receiver: process(state, i_clk)
begin

if rising_edge(i_clk) then
   case state is

      when s_idle=>

         o_done <= '0';
         s_bittimer <= 0;

         if (i_datain = '0') then
            state <= s_start;
         end if;

      when s_start=>
         if (s_bittimer = (c_maxbittimer/2) -1) then
            state <= s_data;
            s_bittimer <= 0;
         else
            s_bittimer <= s_bittimer + 1;

         end if;

      when s_data=>
         if (s_bittimer = c_maxbittimer -1) then
            if (s_bitno = 7) then
               state <= s_stop;
               s_bitno <= 0;
            else
               s_bitno <= s_bitno + 1;
            end if;

            s_shiftreg (6 downto 0) <= s_shiftreg (7 downto 1); --shift right
            s_shiftreg (7) <= i_datain;
            s_bittimer <= 0;
         else
            s_bittimer <= s_bittimer + 1;
         end if;
```

```
        when s_stop=>
            if (s_bittimer = c_maxbittimer-1) then
                o_done <= '1';
                s_bittimer <= 0;
                state <= s_idle;
            else
                s_bittimer <= s_bittimer + 1;
            end if;
end case;
end if;



end process;


end Behavioral;
```

ascii_to_binary_mux.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity ascii_to_binary_mux is
    Port ( i_sw : in STD_LOGIC_VECTOR (2 downto 0);
           o_binary : out STD_LOGIC_VECTOR (7 downto 0));
end ascii_to_binary_mux;

architecture Behavioral of ascii_to_binary_mux is

begin

process(i_sw)
begin

case i_sw is
    when "000" => o_binary <= "01011111"; --'_'
    when "001" => o_binary <= "01110111"; -- w
    when "010" => o_binary <= "01100001"; -- a
    when "011" => o_binary <= "01101001"; -- i
    when "100" => o_binary <= "01110100"; -- t
    when "101" => o_binary <= "01110011"; -- s
```

```
    when "110" => o_binary <= "01101110"; -- n
    when "111" => o_binary <= "01110000"; -- p

    when others => o_binary <= "01011111"; --'_'    default

end case;




end process;

end Behavioral;
```

debouncer.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity debouncer is
    Port (i_clk : in std_logic := '0';     --5MHz
    i_btn: in std_logic := '0';
    o_dbtn: out std_logic := '0');
end debouncer;

architecture Behavioral of debouncer is

signal s_dbtn: std_logic := '0';
signal s_Q1 : std_logic := '0';
signal s_Q2bar : std_logic := '0';
signal s_tempmem: std_logic := '0';
signal s_temp: integer := 0;

begin

debounce: process(i_clk, s_temp, s_tempmem)
begin

if rising_edge(i_clk) then
```

```
    if (s_temp = 1250000) then -- 5MHz / (1250000 * 2) = 2 Hz => '1' for 0.5s before shutting off unless rising edge(button)
        s_tempmem <= NOT(s_tempmem);
        s_temp <= 0;
    else
        s_temp <= s_temp + 1;
    end if;
end if;

if rising_edge(s_tempmem) then      --D-FF no.1
    s_Q1 <= i_btn;

end if;

if rising_edge(s_tempmem) then      --D-FF no.2
    s_Q2bar <= not s_Q1;

end if;
end process;

o_dbtn <= s_Q1 and s_Q2bar;



end Behavioral;
```

integer_to_binary.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity integer_to_binary is
    Port (i_integer:  in integer := 0;
    o_binary: out std_logic_vector( 15 downto 0) := (others => '0') );  --not a direct conversion, converts digits per 4 bits of output
end integer_to_binary;

architecture Behavioral of integer_to_binary is

component tobinary is
    Port (i_digit: in integer := 0;
    o_bin: out std_logic_vector(3 downto 0) := (others => '0'));
end component;
```

```
signal s_dig_i0 : integer := 0;
signal s_dig_i1 : integer := 0;
signal s_dig_i2 : integer := 0;
signal s_dig_i3 : integer := 0;

signal s_binary: std_logic_vector( 15 downto 0) := (others => '0');

begin

o_binary <= s_binary;

s_dig_i3 <= integer(i_integer / 1000);
s_dig_i2 <= integer((i_integer mod 1000) / 100);
s_dig_i1 <= integer((i_integer mod 100) / 10);
s_dig_i0 <= integer(i_integer mod 10);

ind0: tobinary
   Port Map (i_digit => s_dig_i0,
   o_bin => s_binary(3 downto 0));

ind1: tobinary
   Port Map (i_digit => s_dig_i1,
   o_bin => s_binary(7 downto 4));

ind2: tobinary
   Port Map (i_digit => s_dig_i2,
   o_bin => s_binary(11 downto 8));

ind3: tobinary
   Port Map (i_digit => s_dig_i3,
   o_bin => s_binary(15 downto 12));

end Behavioral;
```

tobinary.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
```

```
entity tobinary is
    Port (i_digit: in integer := 0;
    o_bin: out std_logic_vector(3 downto 0) := (others => '0'));
end tobinary;

architecture Behavioral of tobinary is

begin

process(i_digit)
begin

if (i_digit = 0) then
    o_bin <= "0000";
elsif (i_digit = 1) then
    o_bin <= "0001";
elsif (i_digit = 2) then
    o_bin <= "0010";
elsif (i_digit = 3) then
    o_bin <= "0011";
elsif (i_digit = 4) then
    o_bin <= "0100";
elsif (i_digit = 5) then
    o_bin <= "0101";
elsif (i_digit = 6) then
    o_bin <= "0110";
elsif (i_digit = 7) then
    o_bin <= "0111";
elsif (i_digit = 8) then
    o_bin <= "1000";
elsif (i_digit = 9) then
    o_bin <= "1001";
else
    o_bin <= "0000";

end if;
end process;

end Behavioral;
```

delay_counter.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;



entity delay_counter is
    Port ( i_clk : in STD_LOGIC;
           i_reset : in STD_LOGIC;
           o_done : out STD_LOGIC);
end delay_counter;

architecture Behavioral of delay_counter is

signal s_temp: integer := 0;

begin

process(i_clk,i_reset)
begin

if (i_reset = '1') then
    s_temp <= 0;
    o_done <= '0';
else
    if rising_edge(i_clk) then
        if (s_temp = 20000000 - 1) then     --4 second counter
            o_done <= '1';
        else
            s_temp <= s_temp + 1;
            o_done <= '0';
        end if;
    end if;
end if;

end process;

end Behavioral;
```

delay_counter_2.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;



entity delay_counter2 is
    Port ( i_clk : in STD_LOGIC;
           i_reset : in STD_LOGIC;
           o_done : out STD_LOGIC);
end delay_counter2;

architecture Behavioral of delay_counter2 is

signal s_temp: integer := 0;

begin

process(i_clk,i_reset)
begin

if (i_reset = '1') then
    s_temp <= 0;
    o_done <= '0';
else
    if rising_edge(i_clk) then
        if (s_temp = 40000000 - 1) then     --8 seconds
            o_done <= '1';
        else
            s_temp <= s_temp + 1;
            o_done <= '0';
        end if;
    end if;
end if;

end process;

end Behavioral;
```

servo_sim.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity servo_sim is

end servo_sim;

architecture Behavioral of servo_sim is

component servo_top is
    Port ( i_clk : in STD_LOGIC;  --5MHz
           i_sw : in STD_LOGIC;
           o_servosignal : out STD_LOGIC);
end component;

signal i_clk :  STD_LOGIC := '0';  --5MHz
signal i_sw :  STD_LOGIC;
signal o_servosignal : STD_LOGIc;


begin


servo:servo_top
Port Map ( i_clk => i_clk,
i_sw => i_sw,
o_servosignal => o_servosignal);


clk5MHz:process
begin
  while now < 990000000 ns loop
```

```
    i_clk <= not i_clk;
    wait for 100 ns;
  end loop;
  wait;
end process;



stimuli:process
begin

i_sw <= '0';

wait for 250000000 ns;

end process;



end Behavioral;
```

LED_sim.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity LED_sim is

end LED_sim;

architecture Behavioral of LED_sim is

component LED_wave is
    Port ( i_clk : in STD_LOGIC;    --5MHz
           i_type : in STD_LOGIC_VECTOR (2 downto 0);
           o_LED : out STD_LOGIC_VECTOR (15 downto 0));
end component;
```

```
signal i_clk :  STD_LOGIC := '0';  --5MHz
signal i_type :  STD_LOGIC_VECTOR (2 downto 0);
signal o_LED :  STD_LOGIC_VECTOR (15 downto 0);


begin


servo:LED_wave
Port Map ( i_clk => i_clk,
i_type => i_type,
o_LED => o_LED);


clk5MHz:process
begin
  while now < 990000000 ns loop
     i_clk <= not i_clk;
     wait for 100 ns;
  end loop;
  wait;
end process;



stimuli:process
begin

i_type <= "011";

wait for 900000000 ns;

end process;



end Behavioral;
```

The Python Code:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 28 21:24:44 2024

@author: Mert
"""

import cv2
import serial
from pyzbar import pyzbar

def send_binary_to_com(binary_string, com_port):
    try:
        print('\n')
        ser = serial.Serial(com_port, baudrate=9600, timeout=1)
        binary_bytes = int(binary_string, 2).to_bytes((len(binary_string) + 7) // 8, byteorder='big')
        ser.write(binary_bytes)
        print("Scan Results sent successfully.")
        ser.close()

    except Exception as e:
        print("An error occurred:", e)



def receive_data(com_port):
    ser = serial.Serial(com_port, 9600)
    try:
        while True:
            data = ser.read(5)
            if data:
                return data.decode('ascii')
    except KeyboardInterrupt:
        ser.close()



def capture_image():

    cap = cv2.VideoCapture(0)
    ret, frame = cap.read()
    cap.release()
```

```
    return frame


def find_and_read_barcode(image):

    grayscale = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    barcodes = pyzbar.decode(grayscale)

    for barcode in barcodes:
        barcode_data = barcode.data.decode("utf-8")
        return str(barcode_data)



file=open('ticket_database.txt','w')
file.write('TICKET DATABASE \n')
file.close()


if __name__ == "__main__":

    com_port = 'COM4'
    work=True

    while work==True:
        data=receive_data(com_port)
        print(data)

        if (data == 'snap_'):
            image = capture_image()
            barcode = find_and_read_barcode(image)



            room = barcode[0]
            row = barcode[1:3]
            seat = barcode[3:5]

            repetition=bool(False)

            with open('ticket_database.txt', 'r') as database:
```

```
            for line in database:
                if barcode in line:
                    repetition = True
                    break
        if repetition == False:
            img_name = f'room_{room}_row_{row}_seat_{seat}.jpg'
            database=open('ticket_database.txt','a')
            database.write('Barcode: ' + barcode + '\t' + 'Room: ' + room + '\t' + 'Row: ' + row + '\t' + 'Seat: ' + seat+'\n')
            print('Barcode: ' + barcode + '\n' + 'Room: ' + room + '\n' + 'Row: ' + row + '\n' + 'Seat: ' + seat)
            database.close()
            try:
                cv2.imwrite(img_name, image)
                print('\n',"Image saved as:", img_name,'\n')
            except Exception :
                print("Error saving image")

            send_binary_to_com("01000101", com_port)    #e for enable


        else:
            img_name = f'DUPLICATE_room_{room}_row_{row}_seat_{seat}.jpg'

            print('Barcode: ' + barcode + ' has already been scanned. Access denied.')

            try:
                cv2.imwrite(img_name, image)
                print('\n',"Image saved as:", img_name,'\n')
            except Exception :
                print("Error saving image")

            send_binary_to_com("01010010", com_port)    #r for refuse
    elif (data == 'wait_'):
        work = False
        break

        database.close
```