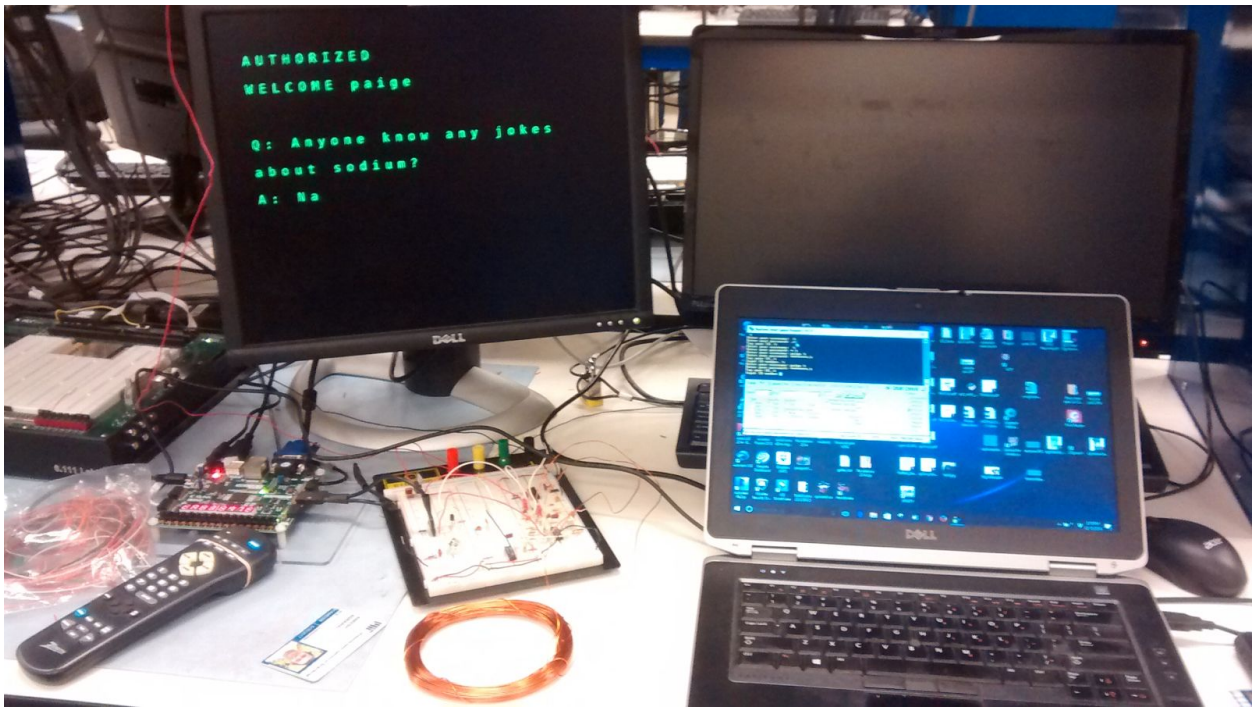
MIT Multifactor Authentication System

6.111 Fall 2016 Final Report



By: Aneesh Agrawal and Paige Studer

Abstract

Our MIT Multifactor Authentication System implements a login flow that requires a username, password, MIT ID card tap, and an ID number input in order to access a secure joke database. The username and password are inputted via serial, using RS-232 serial communication to transfer data from the computer to and from the Nexys4 board. The ID number is inputted through the use of a remote utilizing Sony Infrared Commands. Once the login has been complete if the the user is in the database, then he or she will be redirected to an "Authorized" page which displays a randomized joke. If the user is not authorized, he or she is redirected to a screen that states "Unauthorized" and eventually times out to restart. Every stage of the login is timed, so for the instance that the user walks away, no one else will be able to continue where the original user left off.

Table of Contents

Overview

We wanted to create a complex system that required an ID tap. MIT students must tap their IDs to open doors and get into any dorm. In the dorms, if you tap your ID, the desk worker is able to view a photo of you as well as other information. We wanted to create a similar system, where by tapping an ID we would be able to get some information about the user. Overtime this morphed into a slightly different but still difficult project. We ended up combining the element of logging in with a username and password, that you might find when trying to log on to an Athena cluster computer, with an ID tap before checking to see if a user was authorized or not.

Lab 4, which was the car alarm was an influencer in our system. Similarly to the car alarm, at every state we would instantiate a timer so that if a user walked away no one else could get into the system. In the car alarm lab, if the driver successfully gets in the car without the alarm going off, the driver is able to drive away. In our case, we wanted to reward the user in some way for being an authorized user. We came up with the idea to display a randomized joke for when the user is authorized, so that after all the work the user did to become authorized they could have a good laugh! For added complexity we used serial communication, the SIRC receiver used in Lab 5, and XVGA display.

High Level Block Diagram

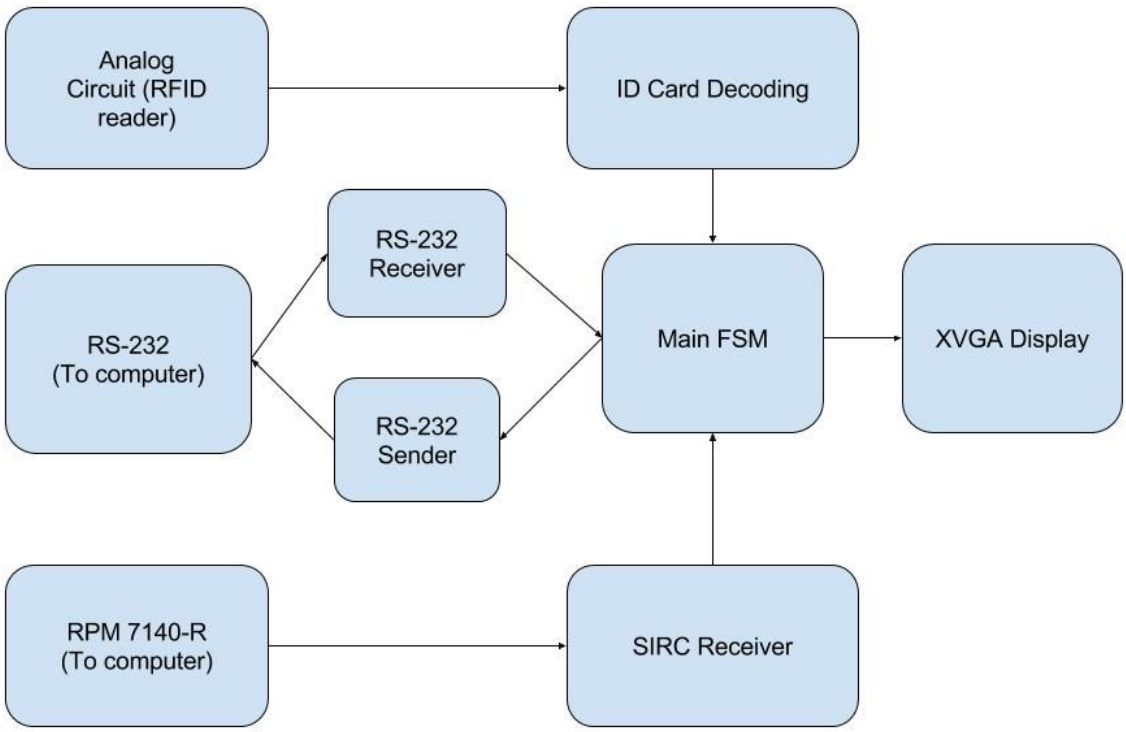


Figure 1: Overall (simplified) Block Diagram

Major Modules

The system has 5 main parts: an ID card reading pipeline, RS-232 input and output modules, an SIRC command decoding pipeline, the main control FSM, and an XVGA rendering subsystem, all of which run on the Nexys4 board.

Global and Common Modules

Many of the modules in our system are self-contained modules which are not part of any particular subsystem (e.g. the RS232 subsystem), but are used globally or in multiple subsystems. Many

of these are small, self-contained, single-purpose modules meant to be instantiated in many contexts as common utilities.

Prologue - Aneesh

A custom “prologue” module was created to store common macros and settings that should be used for all modules; this module was `` `included `` at the top of each other module to ensure consistency. This module configures a `` `default_nettype `` of `none`, and additionally sets a standard `` `timescale ``. (Note that this is not included for IP generation; see Appendix A for information about why.) This module also included a set of macros used for general purpose code as well as testbench specific macros. The most important macro, and the main reason for the creation of this module, was the `CLOG2` macro, which performs the same function as the `$clog2` Verilog function. `$clog2` is not supported by Vivado in many expression based context, so writing a replacement macro allows exposing this functionality in many places - inside bus width declarations, parameters, and other free-form expressions. Other macros include `endian_swap_32` and `endian_swap_64`, which are used to convert between big and little endian representations, `leftrotate` which performs a left circular shift, and `with_width`, which takes a signal of any size, left pads with zeros and truncates to a given width to create a version of that signal at the given width, thus handling both too-small and too-large signals.

The prologue module also includes a set of macros used for testbenches; ensuring consistency amongst testbenches is critical for ensuring testbench result scrape-ability in the build and test system (see Appendix A for details). These comprise: a `CLK` macro, which allows building symbolic expressions related to a number of cycles for expressing delays in testbenches; an `assert` macro, which checks a signal against an expected value and provides nicely formatted pass/fail results; a `pulse` macro, which makes it easy to express signals which need to be asserted for only a single cycle, such as on a start or reset line; and a `start_test` macro, which makes it easy to delineate between tests of different pieces of functionality within a particular testbench. Many of these macros would have been obviated if the project was written in SystemVerilog, which has a much richer set of tools and techniques around verification and testing, and acted as good-enough polyfills for our project.

Reset and Clock Domain modules - Aneesh

Our system uses 2 clock domains, the native 100MHz clock domain of the Artix-7 and the 65MHz clock required for XVGA. Almost all modules run off in the 65MHz clock domain, to avoid synchronization between domains for data already inside the FPGA; only a few modules run off in the 100MHz domain, which is just those modules necessary to bootstrap the 65MHz clock, generated by a clockgen IP described in a following section. The clockgen IP requires a reset line, so a debounce module in the 100MHz clock domain is used to debounce the `BTNR` button. This is used as a raw reset line for the clockgen IP, which is also run from the 100MHz clock. The clockgen IP includes a `locked` output, which is asserted when the derived 65MHz clock is stable. Because we do not want all the other modules to run until their clock is ready, the reset button signal is ORed with the inverse of the `clocks_locked` signal, and then passed through a synchronizing module into the 65Mhz clock domain to produce the global `reset` line for all other modules in the system. This “startup dance”, while slightly complicated, ensures all other modules are able to talk to each other successfully.

Debounce and Synchronize - Aneesh

These modules are based on the debounce and synchronize modules provided by the 6.111 staff, with a few adjustments to conform to our common code style. Additionally, the debounce module was parameterized to take a variable `DELAY`, which was important to handle the multiple clock rates we used. Neither of these modules had a testbench as they were provided and known to be working.

Edge Detector - Aneesh

This module watches a signal for either low to high transitions, or high to low transitions, and outputs single cycle pulses when it encounters one. This is useful to convert a level shifted signal into a edge triggered one, such as only taking an action when a button is first pressed as opposed to as long as it is pressed. It takes a parameter which specifies which type of edge to detect, using custom `POSEDGE` and `NEGEDGE` macros as possible parameter values in the lack of enums in Verilog. There is also a parameterized testbench for this module which tests both configurations.

Timer and Divider - Aneesh

The divider outputs a pulse every `N` cycles, and the timer accepts a timer length at runtime and signals timer expiration after that many seconds. These modules were adapted from their Lab 4 incarnations to accept new parameters, such as the clock frequency, to work in multiple clock domains, and their testbenches were updated to match.

Sampler and Downsampler - Aneesh

These modules are used for (over)sampling an input signal (usually from the external world) and downsampling it to filter out high frequency noise. The sampler module uses an internal divider module to take a sample of its input each time the divider fires, thus sampling the input at a given frequency. The downsampler buffers `N` samples, then uses a simple majority wins algorithm to determine whether to output a `0` or a `1`. Both of these modules had testbenches written, with the downsample testbench being parameterized for various downsampling ratios.

Hex Display - Aneesh

To display data for debugging, we used the hex display on the Nexys4 board. The code used in Lab 4 is meant to be run from the 100MHz and had fairly messy formatting, so we wrote our own version which is parameterized to work with any clock frequency, and implemented a time-division multiplexing (TDM) scheme to control all eight 8-segment hex display simultaneously, aiming for each display to be on for 16ms continuously before moving to the next one. We also implemented a small lookup table to display incoming data as hex digits on the display, making it easy to read debug information off of the FPGA.

Clockgen IP - Aneesh

We used the Xilinx clockgen wizard to create a clockgen IP module to generate a 65MHz clock for the system from the main 100MHz clock. We configured the wizard to include a `locked` output line, as well as disabling input buffering since the 100MHz clock is already buffered. We checked-in a small TCL script to reproducibly generate the IP core each time a build is run; since the clockgen IP is fairly small, this did not increase build times by very much. See Appendix A for more information on how the IP core was integrated into our build system.

ID Card Pipeline

This part of the design is responsible for actually reading MIT ID cards and deciphering the RFID data on them. This is a one-way pipeline, where data flows continuously and in one direction. Most of these modules will have no local state but simply transform their inputs in a combinatorial way, buffering them for the next stage to avoid glitches. This cuts down complexity by eliminating feedback paths and minimizing local state, and makes each module easy to test in isolation.

This part of the circuit required both analog and digital subsections to read and decode the RFID code on the data. Starting with a 125Khz square wave carrier generated by the FPGA, an analog circuit was responsible for boosting and emitting the RFID signal, as well as reading, detecting and filtering the input signal. After passing through an ADC, the data stream passes through a phase-shift keying decoder, a command format recognizer, and a module that reverses the proprietary FlexSecure protection.

The previous MIT student projects we based the deciphering modules on were done approximately 15 years ago, and included various suggestions about how to improve the security of the cards (which was fairly low). Unfortunately for us, it seems that new cards have incorporated those security improvements and thus we are unable to read the data on them properly, but we are still able to use our existing modules to detect when an MIT ID card is tapped (as opposed to non-RFID card or an RFID card which works on a different frequency, etc.)

Square Wave Gen - Aneesh

This module creates a 50% duty cycle square wave with a parameterized period (expressed in clock cycles) by using a simpler counter and flipping the output when the counter overflows. This module is used to create the 125KHz carrier frequency used by the passive RFID MIT ID cards. The output is passed through an FPGA I/O pin and then through a level shifting circuit which is comprised of a 2N70000 small-signal MOSFET and a pair of resistors, which converts the FPGA output from the native 3.3V level to a 5V level. Note that the RFID system itself works on off of sinusoidal waves, but the square wave will be passed through a buffering amplifier, which requires logical outputs, hence a simple square wave output is desired. (The L/C resonator in the analog circuitry creates the sinusoidal waves required.)

Analog Circuitry - Aneesh

Figure 2: Custom coil used as the antenna

A multi-stage analog circuit is used to boost, emit, read and quantize the RFID signal. Special thanks goes to a 6.101 SP2014 project group, whose design we have used with slight modifications. An LM18293 push-pull driver is used to amplify the square wave signal using a 9V supply to provide enough power to read the RFID signal. It drives an LC circuit comprised of a 1.1 mH coil (inductor) and 1.5 nF capacitor, which provide a resonant frequency of 125KHz. When an ID card is tapped, an embedded chip on the card modules how much a coil on the card couples to our emitting coil, changing the power draw of the coil. The coil was created by making ~80 turns of wire at ~150mm diameter, which can be seen in Figure 2.

The output of the LC circuit is AC coupled to an envelope detection circuit which uses 2 diodes, a resistor and capacitor to extract the 62.5KHz modulated signal from the 125KHz carrier signal, then A/C coupled and DC-biased into a 3-pole filter created with an LC filter and a sallen-key filter. This signal is then passed through a LM324 op-amp to amplify the signal, and a Schmitt trigger which picks up on the positive and negative peaks of the signal and provides clean digital outputs that can be returned to the FPGA (at the 3.3V output level).

Picking appropriate trigger level for the Schmitt trigger was one of the most challenging parts of the project. The LC resonator output a fairly strong signal, but the reader was very finicky about the position and orientation of the card, and slight movements would drastically vary the magnitude of the output signal from the amplifier. Additionally, even with amplification and filtering the peaks in the signal were not very high, and the baseline signal was noisy. It seemed that each time we returned the

project, whether after lunch or after a good night's rest, the circuit would have changed slightly, requiring retuning the trigger levels. This required careful probing of the output of the amplifier on an oscilloscope, adjusting various cursors to find appropriate trigger levels, using their average to compute a bias point, performing a few simple calculations to find appropriate resistor ratios to create that bias point using a voltage divider as well as affect the width of the trigger zone, and replacing those components in the circuit, only to see the circuit suddenly become slightly more or less responsive, necessitating another round of adjustments.

An LM7805 was used to provide a 5V supply rail for most components.

Digital Input - Aneesh

Due to the schmitt trigger providing clean digital data, the XADC on the Nexys4 was not used to sample the ID card reader input. Instead, the input was passed through a synchronizing module, then oversampled at 8 times the 4KHz frequency of the signal, aka 32Khz, and then downsampled by a factor of 8. This was accomplished using the aforementioned common modules.

PSK Decoding - Aneesh

The MIT ID card utilizes phase-shift keying (PSK) modulation, so the next module was a PSK decoder. This is a fairly simple module which outputs zeros as long as the input remains the same, and outputs a one when the input changes (phase transitions).

FlexSecure Descrambling - Aneesh

Finally, the demodulated data was passed to a final module which implements a descrambling algorithm to defeat the proprietary FlexSecure protection on the card data. This protection is very weak, boiling down to an XOR and a few shuffled bits, and cannot be considered to be a proper encryption scheme. The algorithm used was extracted from a project by Josh Mandel, Austin Roach, and Keith Winstein, and is computed over a 32-bit window of the input. Of the 32 bits of input, one bit is duplicated, two bits are XORed into one bit, and other bits are shifted around, then the transformed data is XORed with a "secret" key, producing the output. A simple testbench was written for this module to confirm compatibility with the example given in Winstein's presentation.

RS-232 Serial Input/Output Pipelines - Aneesh

To enable input of the username and password as the first factor, the emulated RS-232 interface on the Nexys4 was used to interface with an external computer, where prompts for data and inputs will be displayed and entered in a serial terminal. As RS-232 is a full-duplex protocol, this was enabled by two isolated independent pipelines that can operate at the same time, one that receives input and sends it to the main FSM, and a second that receives data from the main FSM and sends them out. The Nexys4 does not have a native RS-232 interface, but contains an FTDI chip to emulate RS-232 signals over the microUSB port, which is mapped via the master XDC constraints file to pins on the Artix-7 FPGA, enabling serial communication with a laptop or desktop via USB. (On the computer side, a virtual serial port over USB is

used with freely available drivers from FTDI, along with a serial terminal such as Hyperterminal or picocom.)

RS-232 Input Pipeline - Aneesh

RS-232 is a fairly simple serial protocol, and will be handled with a chain of input modules. The input line is mapped to a pin on the FPGA and synchronized into the 65Mhz clock domain. Using a divider module to generate the appropriate baud rate (9600 in our case) times the oversampling ratio of 8, the synchronized input is passed to a sampling module which oversamples by 8x, then downsampled by a downsampling module. Next, an RS-232 receiving FSM buffers RS-232 input frames, checking for frame format validity and parity (malformed or incomplete packets will be silently discarded.) The module is parameterized for number of parity, stop and data bits. Each received character is output on a wire bus (sized by the data bit parameter, which will be set to 8 for the actual implementation), along with an 1 bit enable line to signal new data. The enable line is asserted for one cycle when new data is available, although the data byte will be buffered until the next valid data byte is read. This is read directly by the main FSM.

One obstacle was that this pipeline originally missed characters and frames sent from the computer. Because the sampler and downsampler run continuously but data can be sent at any time, the downsampler would sometimes attempt to average over a window of 8 bits that included a line transition, causing glitchy data. To combat this, an edge detector module was used to detect possible start bits starting (on negedge, or 1 to 0, transitions), and the downsampler was reset whenever one of these bits was detected. Because the downsampler works on windows of 8 samples, missing one or two samples due to resetting after the first sample in a new start bit does not matter, but instead aligns the downsampler windows with the actual baud of the RS232 signal. Additionally, a `busy` output was added to the receiving FSM which signals when a frame is currently being processed, and this was used to avoid resetting the downsampler spuriously during negedge transitions within a frame, which cannot be start bits. This limited use of feedback made the input pipeline much more robust.

RS-232 Output - Aneesh

A separate minor FSM is used to enable sending characters over the RS-232 link. This module uses its own internal divider module to generate the baud rate (no need for a multiplier), and accepts 8 bits of data along with a 1 bit send signal as inputs. On each send request (each cycle where send is asserted), this FSM starts sending the data in RS-232 format at the baud rate, starting with stop bits, then the data bits in sequence, followed by parity bits and stop bits, where these values are also parameterized. When this FSM is finished sending, it asserts a 1 bit done signal for one cycle to signal to the main FSM that it is ready to accept another character.

Serial Prompt - Paige

We wanted to be able to send prompts to the user via serial as well. To do this we instantiated a serial prompt module that utilized cases. If a start signal was received, we would transition to the send state. In the send state, we would go through a string and output a character each time we received a "done" signal for the previous character.

Originally each prompt had its own module for sending the prompt. To eliminate the use of four different modules, we took in a prompt, and its length as parameters. By doing this we were able to use a single module to handle all of the serial prompts.

SIRC Receiver - Aneesh

The SIRC receiver reads 12-bit SIRC commands from a Samsung TV IR remote. This implementation was mostly based on Lab 5b.

SIRC Receiving Pipeline - Aneesh

An RPM7140-R IR receiver is used to accept the IR signal and convert it to a digital signal, which is then input into the FPGA. Because the RPM7140-R outputs an inverted signal, the signal is inverted on the FPGA and passed through a synchronizing module. The synchronized signal is oversampled every 75us, then downsampled by 8x (the native protocol switches at multiples of 600ms). Finally, an SIRC decoding module watches the input for a 2.4ms start pulse, then buffers 12 bits of data to read a full command. Each bit is decoded by a simple state machine that considers a 1 then a 0 as a 0, and a 1, 1, 0 pattern as a 1; invalid patterns cause the state machine to ignore the bit and restart itself, dropping the in-progress command.

SIRC Number Conversion - Paige

This was a simple module that converted the SIRC command to match the number hit. It took in the SIRC enable signal and the four-bit SIRC command signal and outputted a new SIRC enable signal and a new SIRC command. For example, if a user hit the button “1”, the original SIRC command would be a “0” and the SIRC Number Conversion module would change it back to a “1” and a new enable signal would be sent out.

De-duplicator Module - Paige

One of the issues that we experienced was receiving characters too quickly from the SIRC receiver. When we tried to input our ID numbers, instead of inputting one number at a time for every hit, there would be multiple of the same number displayed. The first step was to display one number at a time. To do this, if we received a signal, we would display that number but not display any other numbers until we received a new command. This worked great except for the fact that some ID numbers have the same number multiple times in a row. After trying to debug for a few days, I was forced to use the simple approach of allowing the user to input one number every second, due to running out of time to work on the project. The de-duplicator module takes in the 4-bit new SIRC command and enable signal and immediately begins a timer as well as outputting that 4-bit new SIRC command and enable signal. Once the timer has completed, the de-duplicator will take in another command. We connected this module to LEDs so that the user can easily identify when to press a new command. When the LEDs are on, then the user can input and when they are off, even if the user enters a number it will not be displayed. The only downside of this module is that it does require the user to take at least nine seconds to input all nine digits of his or her ID number.

Main FSM - Paige

The goal of the main FSM was to keep track of the login flow and inform the display and serial communication what to do. The user is sequentially shown screens asking for their username, displaying their username, asking for their password, displaying asterisks for their password, asking for an ID tap, then confirmation of the ID tap, then asking to input their ID number, displaying their ID number input, and finally whether or not they are authorized or not authorized. On the laptop, the user is show prompts for each state as well as echos the characters input for the username and asterisks for the password (with the exception of backspaces). It should be noted that all of the input buffers were initially set to all spaces, so that they had value but did not display anything on the monitor.

The main FSM is currently set up to be a multifactor authentication system where all of the information is entered together and checked simultaneously. Inputs to the main FSM were a serial enable signal, the 8-bit serial character received, a SIRC enable signal, the 4-bit SIRC command, and the 32-bit ID card data. The outputs included all of the stored username, password, and ID input information as well as all of the display enable signals. If escape was entered at any time, the system was reset. To reset our system we made a macro called "`main_fsm__reset`" that reset all the variables involved. If neither reset nor escape had been hit, then we enter into our states.

The first state was `STATE_IDLE`. We waited here until we received an "enter" character before changing state to `STATE_SEND_USERNAME_PROMPT`. The reason why we did this was because it takes a little bit of time for the user to plug in the USB and set up Realterm on his or her computer. By pressing "enter", this lets our system know that the user's computer is sending and receiving serial characters, and we can move to the next state.

The second state is the `STATE_SEND_USERNAME_PROMPT`. This state starts a timer for the user to input his or her username as well as sends the prompt to send the username prompt via serial before moving on to `STATE_ECHO_USERNAME_CHAR`. In this state we first check to see if the timer has expired. If it has then we reset the system. If not, we wait for a serial enable signal. Once we see a serial enable, we check a few things. First we see if the character received is an "enter." If it is an "enter" then we want to move into `STATE_SEND_PASSWORD_PROMPT`. If the character received is the backspace character, then we change the current username input character to a space so that it is not displayed and subtract one from the current username position. This way we emulate the backspace function that we use when typing. In addition, we must check to make sure that the current position is not 0; if it is zero then we do nothing. The next item to check is whether or not the user has entered too many characters. If the username position is equal to the number of characters allowed and is not the character "enter", then we send a bel enable signal. Lastly, if the username position is not equal to the number of characters we allow the username, then we want to add that character to our username input and increase the username position by one.

`STATE_SEND_PASSWORD_PROMPT` functions the same as `STATE_SEND_USERNAME_PROMPT`, because it sends the password serial prompt start signal and begins the timer for the password before moving on the `STATE_ECHO_PASSWORD_CHAR`. `STATE_ECHO_PASSWORD_CHAR` functions the same as `STATE_ECHO_USERNAME_CHAR` with the exception of how the characters stored are different from the characters displayed. The characters are stored in `password_input`, but the characters displayed are in the `dummy_password_input`.

The dummy_password_input does exactly what the password_input does, except instead of inputting the serial character that was received it inputs an asterisk. This way when the user inputs his or her password, it will be shown as asterisks on the XVGA display.

When the user hits "enter", we move onto STATE_SEND_ID_TAP_PROMPT, which begins the ID tap timer and the ID tap serial prompt start signal before moving to STATE_WAIT_ID_TAP. If the card data received does not equal the default value, then we recognize that a card has been tapped. We send the ID tapped enable signal, send the ID number input prompt start signal, start a new timer, and move to the STATE_ID_NUMBER. In the STATE_ID_NUMBER, if the timer has not expired and the ID number position does not equal the number of ID number characters, which in this case was 9, then if a SIRC enable signal was received, then we would add the prefix 4'b0011 to the SIRC command to make it an 8-bit ASCII number and add that to id_number_input. If all 9 numbers had been input, we then move to STATE_WAIT_VALIDITY and start the identity check.

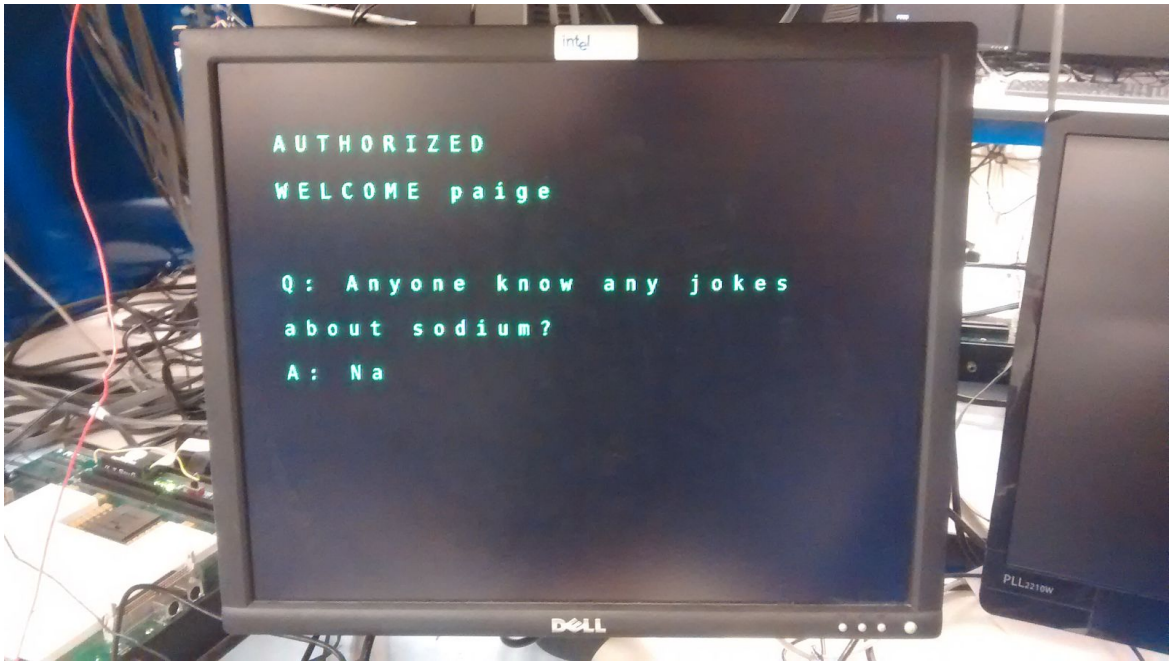


Figure 3: Example authorized state display

In STATE_WAIT_VALIDITY, if the identity check is done we see if authorized is enabled. If it is we go to STATE_AUTHORIZED and if not we go to STATE_UNAUTHORIZED and begin a timer for the unauthorized user. In STATE_AUTHORIZED, the authorized enable signal is sent to the display and does not leave this state until escape has been hit. Figure 3 shows what the authorized state looks like with a welcome note to the user and a randomized joke displayed all in a green font. The STATE_UNAUTHORIZED enables the unauthorized signal, which simply shows the word "Unauthorized" in red on the XVGA. When the unauthorized timer expires, the system resets itself.

Identity Database - Paige

The identity database module's main function is to determine whether or not a user is an authorized user. It takes in a start signal and the user's username, password, and ID number and outputs when it is done checking as well as whether or not the user is authorized. In this module we create an identity rom by reading and assigning data from a database that we specify. To create this database we wrote a .txt file with the username, password, and ID number. With the exception of the ID number, which was always 9 characters, the username and passwords were not all necessarily the same length. For example, the username "aneesh" is six characters whereas "paige" is only five. Aneesh wrote a Python script that took these inputs and made sure that each input had eight characters for the username and passwords by filling in any extra spaces with the space character. This way we could easily add new identities to the database without having to manually fill in extra spaces. We also use md5, which is instantiated at the top. We also created two macros: `identity_database__restart` and `identity_database__reset`. The restart changes the state to idle and restarts the system whereas the reset simply sets the done and authorized signals to zero.

Using cases, we would begin in the idle state until we received a start signal at which point we enable the hashing start signal and we move to the hashing state. In the hashing state, we wait until we receive a done signal, and then move to the searching state. In the searching state, if the inputs are equivalent to the data in the identity rom at a particular address, then we recognize that the person is authorized by sending a done signal and an authorized signal to the output before restarting the system. If the address is equal to the number of entries, this means that we have gone through the entire rom and have not found a match. In this case, we send a done signal but not the authorized signal before restarting the system. Lastly, if neither of these cases are true, we simply want to increase the address by one so we can check the next values and see if they match in the inputted data. Since we only had about 5 entries in the database, checking did not take much time, but if we added the whole MIT student population the checking might take a bit longer.

MD5 - Aneesh

One of our stretch goals for the system was performing password authentication in a more secure manner than simply checking them for equality in plain text. To avoid having to change the laptop client serial terminal (for both Windows, which Paige uses, and Linux, which Aneesh uses), we opted to continue sending the password in plain text, but use MD5 hashing to protect the password in the identity database. While MD5 is extremely broken, it's also simple enough to implement easily in hardware. The python script that builds the identity database computed a hex MD5 digest of each password, which is actually stored in the ID database BROM. At runtime, the MD5 module computes the MD5 sum of the password when a start line is asserted. The MD5 module is parameterized to work with a specific input size and requires all of its input to be provided at the same time, as opposed to working in a streaming or chunked fashion. This makes certain implementation tasks easier, such as appending the length of the input to the input stream, as these values are static and known at compile time. MD5 itself operates on a series of 512 bit "chunks" in the input, and the MD5 implementation closely mirrors the official implementation as presented on Wikipedia. The module operates as a state machine which uses a state to perform setup, then for each chunk uses a state for per-chunk setup, a state that performs 64 rounds of

hashing and mixing operations themselves, and a state for per-chunk finalization, and then a final state that finalizes the MD5 hash and outputs the 128 bit result. Macros are carefully used to swap byte order on various values which need to be interpreted as little endian in the algorithm, but are natively in big-endian order in the rest of the system. 2 small BROMs (each storing only 64 values, one with a width of 6 bits and the other with a width of 32 bits) are used to store constants used in the MD5 calculation. A parameterized testbench was also written, with variants used to test the MD5 hash of the empty string, a small sample input, and a larger input that spans multiple MD5 chunks.

Joke Database - Paige

The joke database was implemented using a case ROM due the lack of time left to finish the project. The module took in a read enable signal and outputted three lines of 224-bit joke data. For this particular database we had 8 jokes, and therefore instantiated a counter that could only count up to 8. We wanted to make the jokes displayed random, so we increased the count by one on every clock cycle. If the read enable signal was received we would then enter the case statement and output the three lines of joke that the counter was on.

Serial Output Selector - Paige

The serial output selector outputs the final serial enable signal and the 8-bit ASCII character to send to the RS232 Send Character module. It takes all of the possible items to send through the prompt such as echoing a character, a sending a prompt, or sending a bel as inputs. The module then decides the priority of the different signals and outputs the correct item to send. For example, if a bel enable signal is ever sent, we will send a bel to the compute.

Renderer

The rendered subsytem is where information from the FSM in interpreted and used for display on a screen, with a pipeline that converts pixel addresses to pixel values. The main FSM tells the Address Calculator module what we want to display. The Address calculator then determines what pixel address we want to access from the character spritemap ROM. The Background Selector module then takes in inputs from the main FSM and the charcter spritemap ROM to determine what color the pixels displayed will be and outputs 4-bit RGB values.

XVGA Signal Gen - Paige

For the display, we used XVGA which specifies a 1024x768 resolution, requiring a 65MHz clock at a screen refresh rate of 60Hz. The Nexys4 uses 4-bit color, so each pixel is a 12-bit RGB color value to display. This module was taken from Lab 3, the pong game lab.

Background selector - Paige

This module should really be renamed to character colormap. Originally we were going to change the background of the display depending on whether the user was in the login, authorized, or

unauthorized states. However, after testing this out, it did not look as nice as just simply changing the color of the characters displayed on the screen.

An authorized enable, unauthorized enable, and character pixel value were all taken as inputs and 4-bit R, G, and B values were given as outputs. If the authorized enable signal was on, depending on the character pixel also received the output would be either black or green. This was the exact same for the case of the unauthorized enable signal. If neither of these enable signals were on, the RGB value was just 4 times whatever the current character pixel was, which meant the login was black or white.

Address Calculator - Paige

The address calculator was a very large module that outputs the location of the pixel that needs to be accessed at the character spritemap, as well as a new hsync and vsync. The address calculator takes in the three 224-bit lines of joke data, a 64-bit username input, a 64-bit password input, and a 72-bit ID number input in addition to enable signals for the MIT login, the username display prompt, the username input, the password display prompt, the password input, the ID tap prompts, the ID number display, the authorized display and the unauthorized display. For characters that did not need to be inputted, such as all of the display prompts, these were instantiated within the module as registers. For example one register in the Address Calculator was 72-bits that said "MIT LOGIN:".

For each display that we used, they each had a local parameter that defined its location on the monitor. The first location was where we wanted the top of the 32 by 32 pixel character box to start. The second was the bottom of the character box. This took care of where the characters would be displayed vertically on the screen. Finally we needed the left side of the left-most character box to determine where we want to start. Usually this was simply another local parameter we called "INDENT." In addition each display needed an integer current location so that we could keep track of how far we were into reading a string.

Next, for each display we utilized a for loop that would run through the string given. If `hcount` and `vcount` were within the specified boundaries and the enable signal was on, then we would output the character, the character row, and character column to then send to the character spritemap to then display on the screen. We had a bit of trouble with reading information, such as the username input, because the location we were reading from was not a constant. It turned out that we needed to use a very particular format (Verilog bit-select syntax) in order to read different parts of the string. Lastly, we delayed hsync and vsync by one cycle to match the calculation latency.

Character Spritemap - Aneesh

We displayed text on the screen by using a character spritemap containing a 32 by 32 pixel sprite for each of the 128 ASCII characters; in the spritemap, each pixel is 1 bit, either black or white. We generated this spritemap by using Codehead's Bitmap Font Generator (CBFG), available online at http://www.codehead.co.uk/cbfg. We used this program (a Windows executable which also runs under WINE) to rasterize the DejaVu Sans Mono font, choosing a monospace font so they would nicely fit a grid and spacing each character to be in the middle of its sprite. CBFG output a bitmap (BMP) file, which we then post-processed using a Python 3 script that leveraged the Pillow library to read the pixels of the bitmap and write out a file containing each pixel (bit) on a separate line as ASCII ones and zeros. This file was then used as the backing store for a `readmemb` call in Verilog, which we used to load the character

spritemap packed array BROM with values. We found this approach to be more scripting and git friendly than attempting to use the MATLAB script, COE files and the Xilinx BROM wizard. The character spritemap array was wrapped in a module that accepted requests for a pixel in a given character at given vertical and horizontal offsets in the sprite, and returned the pixel at that location one cycle later by looking it up in the BROM. Due to the ordering of the bits in the file and the use of power-of-2 sized sprites, we were able to easily index into the BROM by simply concatenating the character, row, and column values to produce an index, which is a clever optimization as compared to the general case which requires multiple multiplications, additions, and subtractions, and would add additional latency. This spritemap let us convert the address calculator outputs into actual pixel data.

Design Experience

Paige

Before this class, I didn't know what a bit was nor had I worked with Verilog (I didn't take 6.004, which apparently is the class to take). I also had never had to design a project to this extent. In the past, labs were always pre-defined or had a block diagram that I could follow to complete the project. I went from having an explicit path to having to create that path for myself. Thinking of an idea and not having a very specific finished goal in mind can make the design process difficult. When Aneesh and I began talking about the project, we had a lot of ideas, but didn't have a clear idea of what the end goal would be. Over the next few weeks, we did come to a complete idea, but at the beginning it was difficult to design because we didn't have that clear picture of the end project. In addition, initially conceptualizing the design was difficult for me because I knew what need to be accomplished but I didn't know what to do in order to accomplish the goal. Aneesh and I went to the whiteboard a few times before we decided we had a system that could possibly function.

One of the biggest obstacles that I faced was working with the visuals on the monitor. I spent the first few weeks learning how to display an image on the monitor by using the MATLAB script provided to change a BMP file into a COE file to then upload to the block memory provided by Vivado. After a few weeks, Aneesh and I realized that this approach would not work for us and was taking up too much memory as well as using an unnecessarily large amount of wires. Instead Aneesh built a ROM that could access one pixel at a time. Once this was completed the next step was to figure out how to access a specific pixel to display a character on the screen. The code went from being able to display a single character, to being able to go through a string and displaying each letter in the string. I moved the text around on the screen a lot and Aneesh taught me to parameterize everything so that changes were easier to manage.

I was also in charge of the main FSM. When working on the FSM for the car alarm for Lab 4, I really had a difficult time and didn't fully grasp the concept of states and how to transition. Thank goodness I learned about it all in Lab 4, because by the time we got to our final project I felt like a pro. The main FSM structure and transitions were relatively simple. The difficult part of the main FSM was keeping track of all of the different components, such as keeping a buffer of characters, sending timer prompts, and sending enable signals.

If I could change one thing it would be to have more time! There were a few things that I worked on near the end that I would have liked to do a better job on but with the time crunch at the end did not have time for. For example, the joke database is currently a case ROM, but I would have liked to instantiate a block ROM that read from an uploaded file similar to the identity database.

Overall this was a great project for me to work on and learn more about digital systems. Aneesh was a very helpful partner teaching me about Git, parameterizing where I can, and teaching me how to write code so that it is more efficient, and easier to read and understand. I would be coding things one way and have no idea about another better way to do it which Aneesh would enlighten me to. And although it may have taken me a bit longer to finish my code, I feel more confident in my skills and can tell if my code is being written well or not. This class has given me an interest in digital systems and I hope I can continue to practice building my knowledge in the future.

Aneesh

This project has been a huge part of my life for the last month and a half - Gim suggested allocating 3-4 hours per day in lab on average, but I feel like I spent more than half my waking time there! It's been an exhausting yet rewarding experience.

One of the first cool parts of the project was winding my own inductive coil to function as the main antenna in the ID card reader. I learned about different gauges of magnet wire, the best ways to wind wire, got creative with coffee bean jars and traffic cones to wind the coil around, and made a prototype before winding a second coil.

At the same time, I was spending my first week and a half on the project learning how to wrangle Vivado to make building and simulating our code scripted and reproducible. I learned a new programming language (TCL), wrote a bunch of code to control Vivado, read through hundreds of pages of Xilinx documentation, and finally had Vivado in a place where I could keep an eye on it. After using git for other projects, I couldn't imagine working on a collaborative project without it, and spending the time to make Vivado integrate with git was well worth it towards the end of the project when Paige and I were able to work on completely separate parts of the project, run git merge, and be greeted with a surprising lack of merge conflicts - the integration boogeyman never haunted our project.

As someone who enjoys using programming languages, tools and systems at the cutting edge of industry and research, I'm used to high level of abstraction in the tools I use, and often ran into many limitations of Verilog that caused me to have to go beyond the concepts taught in the class - such as introducing includes and header guards, (ab)using macros everywhere, and writing lots of custom testbench code. There were many days when I was combing through StackOverflow looking for guidance on how to accomplish a task, lamenting at how easy it would be in SystemVerilog yet impossible in regular Verilog.

Even though this is a digital system class, dealing with analog circuitry was another sticking point during the project. As mentioned above, the ID card reading circuit was fairly delicate, and I often had 3 or 4 oscilloscope probes on the circuit at a time. After using state-of-the-art Agilent scopes, working with the Techtronix scopes also felt like walking through molasses (it's scary how much you miss the little things like push to reset!). The Schmitt trigger had to be constantly adjusted, which ate up a lot of my time. Another example is the level shifting circuit I used to translate the square wave from the FPGA to the push pull driver - this is a fairly simple circuit involving a MOSFET and a couple of resistors and I've

taken 6.012, the basic semiconductors class which teaches the theory and operation of MOSFETS, Diodes, etc., but it took me multiple hours just to find an appropriate MOSFET to use. This breakdown between theory - “just whack in a MOSFET” - and practice - “which one do I use?” - seems to be a hallmark of electrical engineering, with so much knowledge only attainable by working in a lab along experienced engineers.

Fortunately, I was able to draw on a lot of previous experience and knowledge for this project, ranging from previous work with the RS232 protocol with microcontrollers in the 6.115 class, to being able to reuse modules I had written for the labs in the project. Following the UNIX tradition of having each module do one small thing, and writing testbenches to make sure they do those things, was in my opinion a key factor in our ability to complete the project on time as well as a stretch goal.

Near the end of the term, I spent a week in Hawaii for a conference and was unable to work on the project. I’d like to give Paige an extra special shout-out for carrying the team during this time (the final week before the checkoff!) and integrating all the modules, writing the final bits of functionality, and polishing it to a shine. I’m thankful to have had the chance to work with a great partner as well as the course staff, including Alex and Gim. I invested a lot of time into the project, am proud of what we’ve been able to make, and am looking forward to future projects involving FPGAs (especially the iCE40 from Lattice Semiconductor).

Potential Applications and Expansion

Since our digital system was a login system there are countless applications that require some sort of login verification. The reward in our case was a randomized joke, but this could be expanded to control other systems, for example if you are an authorized user you would be able to unlock a room door.

When working on this project we came up with a lot of additional ideas to make the system even better in the future. We would change the FSM into a multi-step verification system where each step would be checked before the next step could happen, enhancing the user experience, although providing a small reduction in theoretical security. Another extension that would benefit the user would be to enable additional line editing of serial inputs allowing keys such as delete and the arrow keys to work properly along the already implemented backspace.

One thing that we had originally thought we would be able to do based off of the Winstein paper was to read the MIT ID cards. Unfortunately between the time the paper was written and our project, whoever is charge of the ID cards changed the way the ID cards are read so we were not able to differentiate between different ID cards. It would be an awesome project to figure out the new format and actually be able to read the MIT ID cards. If this happens, then we could eliminate the stage where the user must enter his or her ID using the remote. In addition it would be cool to read other RFID cards as well such as a T-card. All of these options would be interesting to explore in the future.

Conclusion

For anyone who is pursuing a similar project in the future, it is really helpful to do all of the things that we didn’t learn in class first. By this, I mean that it is better to save the things that you know how to do, such as construct an FSM, for the end when you don’t have much time, and try to work on all

of the things you may not be familiar with, such serial communication or creating and accessing a spritemap, at the beginning when you are not in a time crunch. This was critical to the completion of our project. For the first few weeks we didn't have much to show due to learning curves, but once we got the hard items out of the way the project flowed relatively smoothly. For example, spending a week and a half on git integration didn't produce any Verilog, but made it much easier to collaborate and combine our work.

Overall, we were happy with our resulting project. After spending countless hours in lab, we did have a completely working system. We were able to send and receive characters using serial communication, we were able to display prompts and inputs on the monitor, we were able to detect an ID card being tapped, and we were able to utilize a ROM to create an identity database. We accomplished almost everything we had hoped to during this project and continued our learning of digital systems with the completion of this project.

Acknowledgements

We would like to thank Gim and all of the TA's for making this course so wonderful and working with the students to accomplish the labs and our final project. Your patience, dedication, and enthusiasm to helping and guiding us students through our labs and projects is truly appreciated. We would also like to thank Alex for being our project mentor and believing that we could accomplish everything we did this semester. 6.111 has been a wonderful experience, and it wouldn't have been as great without all the support from the staff, so again, thank you!

Appendix A: Build System (Aneesh)

Another major component of our system that I invested a week and a half into actually wasn't Verilog code, but a build + test scaffolding system for the project. This made it possible to build the project reproducibly, merge code easily (using textual formats), and to build and test parts of the project in the user friendly manner. The main impetus for this was twofold. First, during lab 5 I would often encounters seemingly random errors where Vivado would refuse to build my code, due only to differing orders of adding files to the project, which caused me to waste a lot of time. Second, to work effectively collaboratively, I'm a firm believer in using the git version control system, and this meant finding a way to avoid checking in binary Vivado project files.

I started by reading lots of Xilinx documentation, from their users guides which are hundreds of pages long, to forum posts and Xilinx Q&A, to blog posts on the wider internet. I learned a lot about the processes by which Vivado builds Verilog, including place and route, bitstream generation and more, the project and non-project modes in which it's possible to coordinate Vivado, and the various tools used to run simulations, which is separate from the main tool suite. I also learned the TCL programming language, since nearly all EDA tools, Vivado included, are only scriptable in TCL. I used this knowledge to build a set of scripts that go from a fresh copy of the source code to a built FPGA bitstream file, and also allow running one or all testbenches in the simulator or headlessly. These scripts operate in the non-project mode, running Vivado purely in memory to exert maximum control.

All source code is stored in a `src` directory, while built assets are created in a `build` directory which is pruned and created afresh for each build.

The build system starts with a wrapper bash script, when invokes Vivado with the path to a TCL file. This TCL file controls the main build process: it sets up some common variables, uses a Python script to generate the identity database from a text version which is checked in, generates various IP cores for inclusion in the project, reads all of the Verilog and constraints files, runs synthesis, various optimizations, place and route, writes out a bitstream file, and finally opens the Vivado GUI to allow for programming. To avoid requiring changing this script each time a new module was added to add it to compile list, we used Verilog includes and header guards on all modules so that each module would include its dependencies and dependencies are not processed twice, acting as a crude import system. Note that testbench files don't get header guards, because you can only run one testbench at a time. Each module also included a custom prologue which set common settings such as `` `default_nettype none ``, which helped us catch many bugs. However, this prologue was only read after IP generation, because the Xilinx IP cores do not function without implicit nettypes. The recommended way to integrate IPs is to check in XCI files, which are XML files listing all properties of a given IP. However, to avoid this, I instead created a custom TCL script for each IP, which instantiates a copy of that IP, and customizes it with any changed parameters. This is semantically similar to what we did when running the wizard, and also makes it much easier to understand what is happening when attempting to merge changes.

The simulation system operates on a separate subdirectory of the source folder, which holds only testbenches. Each testbench includes the module it is testing, which in turn includes all of its dependencies. A custom Python script is used to run the testbenches; it allows running all the testbenches together (without graphics) to check that all the tests still pass after making a change, or running a single testbench for faster iteration and feedback times. It also supports opening up the simulator graphically for visual inspection of timing. Many modules take various parameters, so their testbenches are often parameterized as well to support testing different combinations of parameters. These are driven from a variants.json file which lists all the different combinations of parameters a given testbench should be invoked with; the user can also pass custom parameters from the command line. A custom TCL script is used to automatically add all waveforms to the simulation and run the simulation directly. One hurdle here was that the simulator does not provide any way to determine if a testbench finished successfully or due to an error, so the simulation script instead manually scrapes the output of the testbenches, checking for FAILures in the output. This was enabled by the use of a common `assert` macro used across all testbenches.

I spent a lot of time on this part of the project and made it fairly part and project independent. This is something that most students would not be able, and should not have to, make as part of their own projects, but that makes the experience much smoother by making it easy to work with Vivado. I would love to donate this system to the staff to make it available to future 6.111 classes, so they are able to spend more time on building interesting systems on FPGAs!

Appendix B: Source Code

All the code written for this project is available online in our git repository. It can be accessed and browsed on the web at https://gitlab.com/aneeshusa/wildcat, or cloned and checked out via anonymous git at https://gitlab.com/aneeshusa/wildcat.git or via SSH at git@gitlab.com:aneeshusa/wildcat.git.