EE108 Lab 1

Password Hashing

Lab 1 demo video parts 1, 2 available at (part 1, part 2)

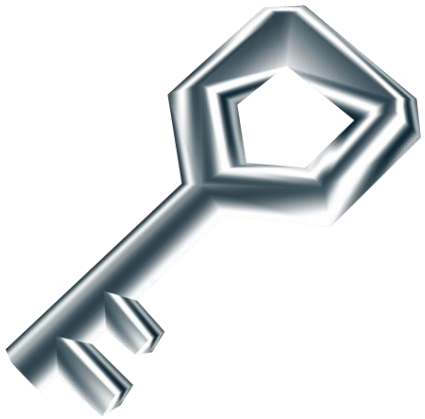
Lab Due: **9/30/20 midnight**

Objective: The purpose of lab 1 is to give you experience designing combinational logic and writing a wide variety of Verilog constructs.

1. Introduction

So, what are we doing here?

In this lab we'll be designing a module that is given a username and password as input and outputs a single bit that reports if the username and password combination is valid. This will require us to use some fundamental components and data representations we'll see throughout the field of digital logic. We'll see how our hardware username-password-checker differs from an equivalent software implementation, and the great advantages this brings.



Our goal here is to get accustomed to writing combinational logic in Verilog, so we'll walk through the implementation of our system step-by-step. In the next lab, you'll be implementing a design of similar complexity on your own.

We'll spend the rest of section 1 going over some of the basic concepts we'll be dealing with in this lab, and we'll get to how the system is actually implemented in section 2. We'll walk through coding each module step-by-step in section 3, and then we'll have you submit all of your hard work in section 4. Section 5 covers what you'll demonstrate for this lab.

Before we go in, just to make you feel a little better: yes, there are a lot of pages in this handout, but this is because we make sure to walk you through every step of the process. If you do the lab carefully

as you read along in the handout you'll be done by the end, and then you can celebrate by making paper airplanes out of the pages (maybe not this year, but at least we are saving some trees).

Representing text with bits

Digital logic operates solely on binary bits while our usernames and passwords are written in English letters and numerals, so we have to choose a way of representing these symbols in binary. Almost universally, digital systems (including your computer) do this with the **ASCII** standard (American Standard Code for Information Interchange). ASCII is pretty straightforward – it is just a table that assigns each of the numbers 0 to 255 to a unique symbol, including the upper and lower case glyphs of the English alphabet. We only care about a small range of these symbols, reproduced in the table below:

65	A		73	I		81	Q		89	Y		54	6
66	B		74	J		82	R		90	Z		55	7
67	C		75	K		83	S		48	0		56	8
68	D		76	L		84	T		49	1		57	9
69	E		77	M		85	U		50	2		00	NULL
70	F		78	N		86	V		51	3			
71	G		79	O		87	W		52	4			
72	H		80	P		88	X		53	5			

We'll store each symbol as an 8 bit number according to the table above, and limit our usernames and passwords to 8 symbols. That way, the longest username or password string will be 8*8=64 bits long. To pass these strings between modules in Verilog, we'll use 64-bit wide wires. The number 8'd00 (or the 'null' character) represents the end of a string of characters – it does not get drawn on the screen, and anything after it is meaningless garbage. This is useful because we have to design our Verilog modules to fit the maximum length value, and so we'll have 64 bits of data even if the input was only one ASCII character long. With a trailing null we can figure out how long the original input was.

Searching our list of usernames

Since character strings are just a bunch of bits, we can check if two of them are equal the same way we check if two numbers are equal - using a big comparator block. Simple enough. Let's say we have a list

of 8 registered users. We'll just compare the input string to each of these 8 stored strings, and if one of those comparisons turns out true then the input username is valid.

This is the first place we'll see that hardware has a huge advantage over software: in an equivalent computer program we would loop over each stored username and check if the input matched – one at a time. In hardware we'll instantiate a separate comparator for each username in our database, and these comparators will evaluate their results in parallel. If our user list gets twice as long the software will take twice as long to run, while the hardware will take the same amount of time as it did on one user.[1] This massively parallel list search is called **associative search**, and is something hardware is very good at. As an aside, in EE180 you will learn about how your computer uses associative search extensively to improve the performance of its memory.

Hashing, in general

Systems that require users to log in with a password don't actually store the user's actual password text in their internal database. Instead, they run the text through a mathematical algorithm called a **hash function**. Hash functions take a set of input bits and spit out a seemingly random sequence of output bits. The key is that they're both deterministic *and* unpredictable; they will always output the same sequence of bits for a given input, yet that sequence of output bits will look completely different from the input. Given a hash, you generally can't recover the input that generated it. Because of this property we call our hash a "one-way" function – no reversing what's already been done.

We store the *hash* of a password. When a user tries to log in, we hash their input and compare it to our stored hash. The same password should consistently hash to the same value, so if our two hashes are equal we can be reasonably sure the inputted password was correct[2].

Password hashes are routinely used instead of the actual passwords for security reasons. You can't log into an account if all you have is the password hash – entering the hash into the password box will just cause the hash string to be run through the hash function, generating a different (incorrect) string of bits. This way, we don't have to worry about hackers getting access to our accounts even if they gain

---

1 Okay, okay, if you've studied computer science at all you might be saying "this is unfair, a computer program wouldn't use a linear search for large databases like this. The performance gap wouldn't be that large." The point of this example is to illustrate that digital hardware is (1) massively parallel and (2) running literally at the speed of light, which no matter how you cut it will beat functionally equivalent software in running time and energy.

2 *Reasonably* sure – if the hash is fewer bits than the input, collisions where multiple inputs hash to the same output are inevitable. In general, though, colliding inputs are impractical to find and look nothing like legible text a human would choose for a password. Hash functions in the real world can be subtly tricky – many mathematicians and computer scientists have dedicated their careers to both developing and cracking hash functions. Ask your TAs.

access to a service's user database. We don't even have to trust the people running the service – they can't see your password either.

So, our hardware module will store password hashes and we'll build a module to hash the user's password input. Sounds good.

Hashing, in this lab

In this lab, we'll be implementing the "Brownie" hash function which is designed to be easy to implement entirely with combinational hardware. It's *very loosely* based off of the MD2 hashing algorithm.

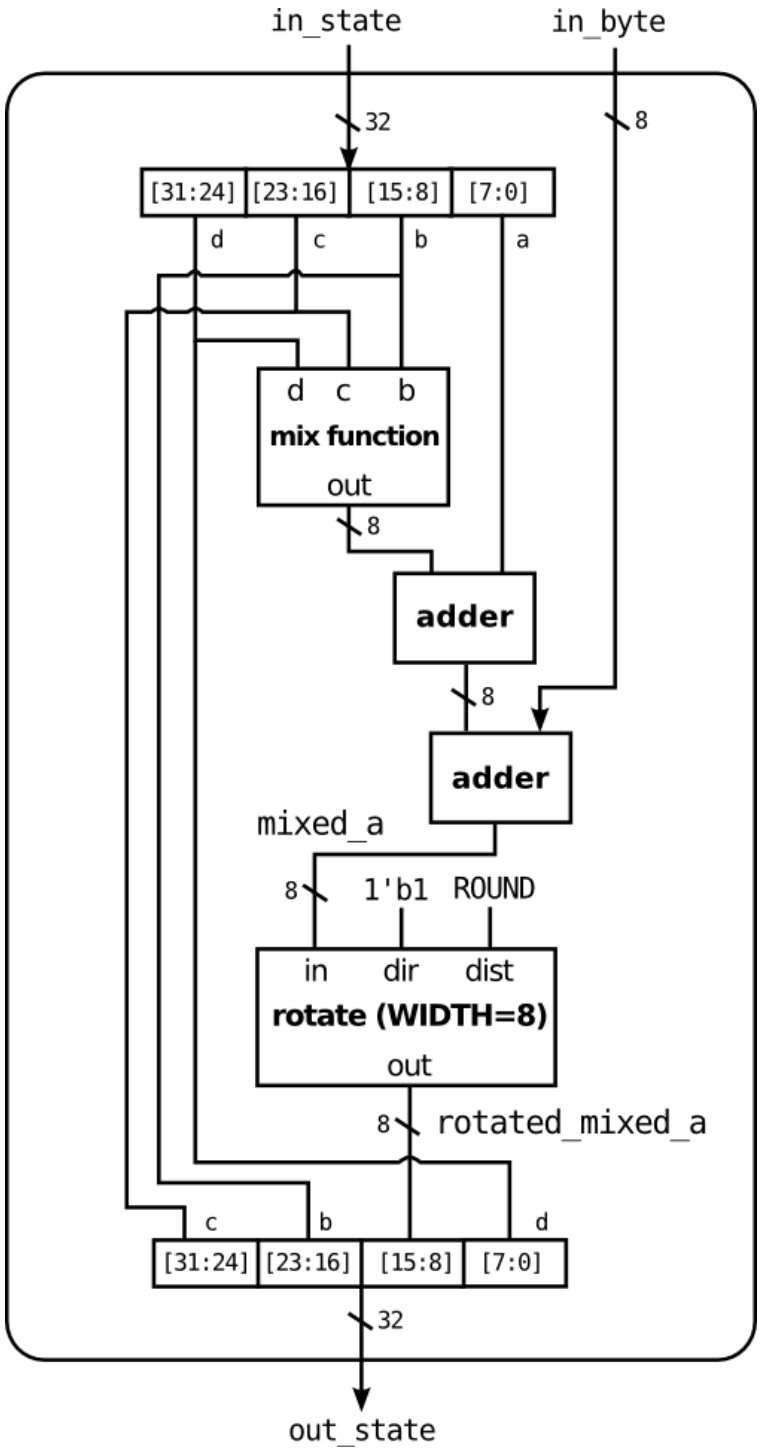
In real life, designers choose password hashing functions that are **cryptographically secure**, meaning it is truly impractical to undo the hash and get back the original text. Brownie Hash is **way way way not cryptographically secure** and if you use it in any practical applications, say in your first job out of college, **you would totally get fired**. We're using it here because it's comparatively simple and it'll allow us to explore some features of Verilog.

Anyway, Brownie Hash is made up of **hashing rounds** which take two inputs, the internal **hash state** and a single byte of the input data, and have one output, the next hash state. It's like mixing up cake batter, where we slowly fold the dry ingredients into the wet ingredients.[3] Here, we'll pass a bunch of bits between the rounds and call it our hash state. In each round, we'll mix one byte of our input data into this state.

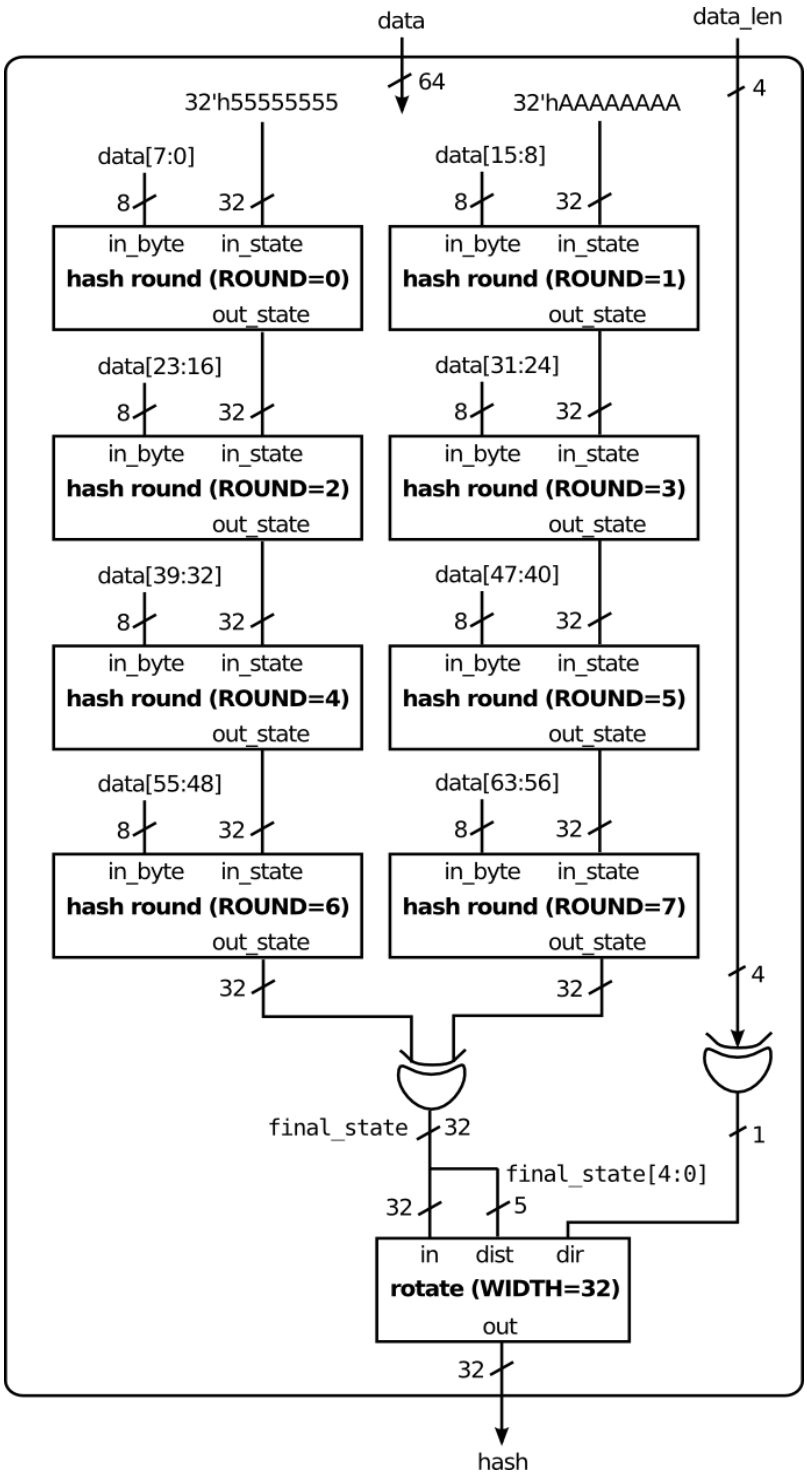
The hash state is 32 bits large, and we'll split it into four bytes called **A**, **B**, **C**, and **D**. In each round, we will mix up the bits of **B**, **C**, and **D** with a simple boolean mixing function and add that result to **A** (we'll look at the mixing function in detail later, when we actually implement it). Then we'll add a byte

[3] If this analogy doesn't make sense to you, a good practice exercise is to bake cakes from scratch until everything clicks and then give those cakes to your TA.

from the input data to **A** too. Next, we'll rotate the bits of **A** to the left and finally swap around all of the state bytes. The whole process is illustrated in the block diagram below.



In the top-level hashing module we'll have two parallel threads of hash rounds. At the end, we'll XOR the two parallel states together and bit-rotate the whole result. That'll be our final output hash.



Memories

We'll be using two types of **memory** to store these usernames and password hashes. Memory is a technical term here – digital systems use memory circuits to store large amounts of data, and we'll be

working with them a lot this quarter. Here, we'll be storing the usernames in a **CAM** (content addressable memory) and the hashes in a **ROM** (read only memory).

Let's talk about the ROM first, since it's simpler and more common. A ROM is essentially a numbered list of values. We call the values **words,** and the position of a word in the list is its **address**. The address is the ROM's only input, and the word stored at that address is the ROM's only output. Large ROMs get a little more complicated, which we'll discuss later in the quarter (after we learn about sequential logic).

A CAM is sort of the reverse of a ROM – it's still a numbered list of values, but the input is the value and the output is that value's position in the list. Our simple CAM will be implemented with a big associative search as described earlier in this handout. In the event the input value is not in the CAM, we'll have a second output, called 'valid', that will be 1 when the input was found and 0 when it was not. Just for some context, CAMs (specifically, a variety called TCAMs) are used *extensively* in networking hardware, though that's unfortunately beyond the scope of this class.

Putting it all together

We'll set up a CAM for the names and a ROM for the password hashes. The user will enter a name and password. Our system will use the CAM to get the address of that name, and will plug that address into the ROM to get that account's password hash. While all of this is happening, we'll also be hashing the input password (remember, this is hardware, so unrelated computations can happen entirely in parallel). Finally, we'll compare the hashed input with what we got out of the ROM and if the two hashes are equal, output a '1' to tell the user they've logged in successfully.

The user database

Here's a list of all our system's users and their password hashes, in both plaintext and hexadecimal form. Each pair of hex digits represents a single character, but they appear in the reverse order as they

do in ASCII text. This is because the first character in the string is encoded by the *least* significant byte of data, and so on.

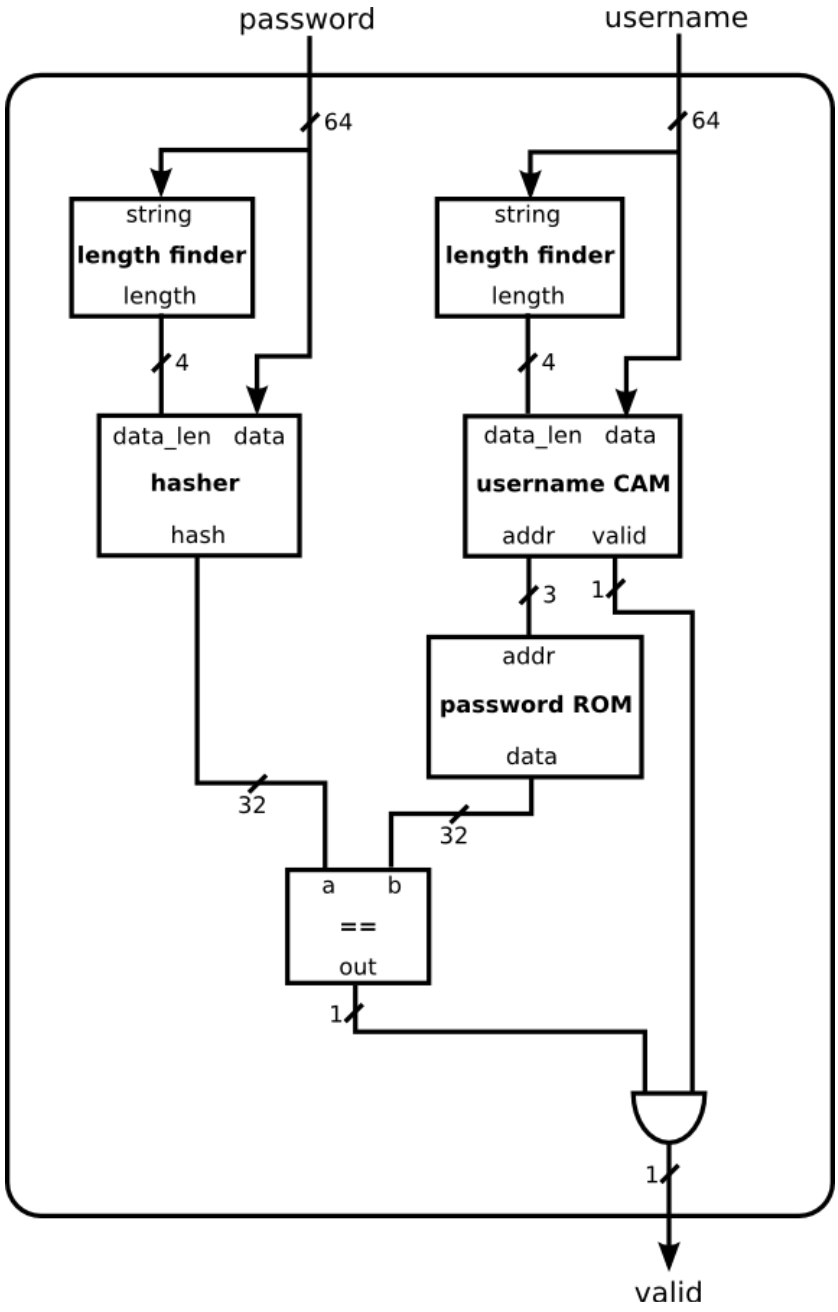
	Username			Password	
#	Text	Hex		Text	Hash
0	LEO	64'h00000000004F454C		DFA1979	32'hDC1A2C9E
1	AARON	64'h0000004E4F524141		COWSRULE	32'hDC2EA8E4
2	HOLLY	64'h000000594C4C4F48		RUNRUN	32'h355FACC3
3	DAVID	64'h0000004449564144		ICARUS	32'hAAF4ADC9
4	CLAIRE	64'h0000455249414C43		XIEXIE	32'h13D41CED
5	FRANK	64'h0000004B4E415246		P3L3GRNO	32'h7EBCF8A8
6	LANCE	64'h00000045434E414C		F0RNWALT	32'hF3CDDB9B
7	RYAN	64'h000000004E415952		AR1STA	32'h9948E6BE

2. Implementation Overview

We're going to walk through the design of each module together, in the handout, and leave translating it into Verilog to you (don't worry, we'll give you some hints!). The goal here is to get used to translating

block diagrams into Verilog code, but pay attention to how we're organizing things at the block diagram level since in lab 2 you'll have build the whole system from scratch.

We'll start with very simple building blocks like encoders and arbiters, and then use those in progressively more specialized modules. Let's take a look at a top level block diagram just to get an overall sense of what we're building and how things are connected to each other:



Make a Vivado project

Create a new, empty Vivado project named 'lab1'.

The starter code is located on Canvas. Add the starter code to your project the same way you did in lab 0, using the “Add Sources” tab on the far left.

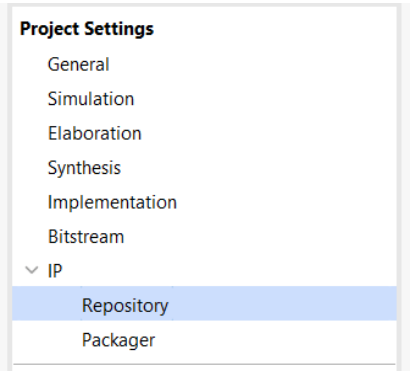
1) First, add all the design sources using “Add Sources -> Add or create **design sources** -> Add files.” Select all the files that DON’T end in “_tb.v” and add them.
2) Repeat step 1, but select “Add or create **simulation sources**” instead. Select all the files that DO end in “_tb.v” and add them.
3) Repeat step 1, but for the **constraint source** (lab1.xdc).

Next, set up your github repository. As you may have seen in lab 0, Vivado creates a lot of folders and log files when simulating and synthesizing. To keep it simple, just track the starter files (the .v and .xdc files). You may also find it helpful to track your .bit file, which will be located in lab1 -> lab1.runs -> impl_1 -> lab1_top.bit once you generate it.

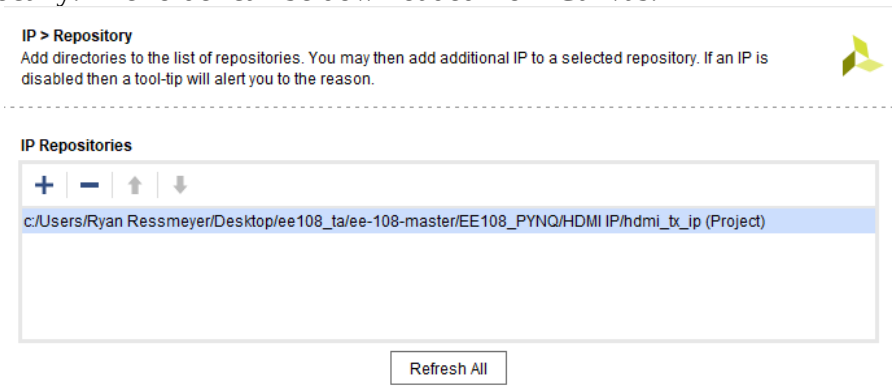
Now, we’ll need to do something we didn’t do in lab 0: adding IPs. These IP blocks are modules provided to us by Xilinx that do frequently performed tasks. We’ll be adding some IP blocks that

convert our video to HDMI format and convert clock signals (we'll explain why in lab 3). We'll need to do this for all labs for the rest of the class.

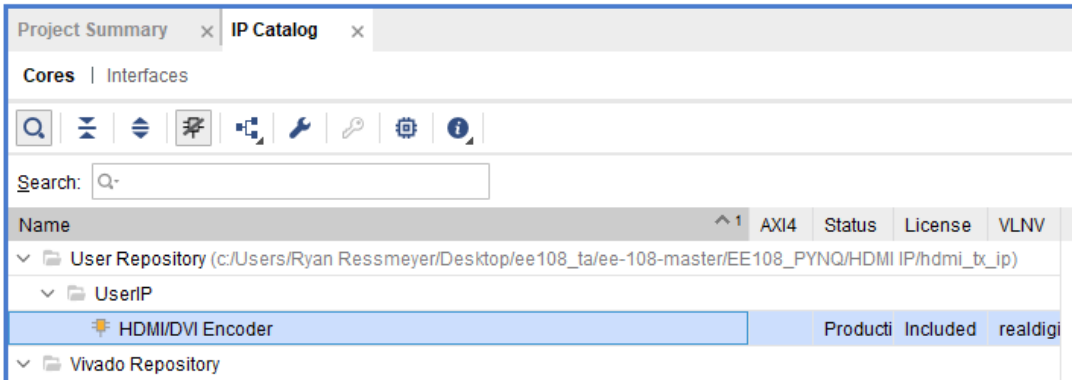
1) Go to "Project Manager -> Settings" and expand the IP tab. Then, click on "Repository."



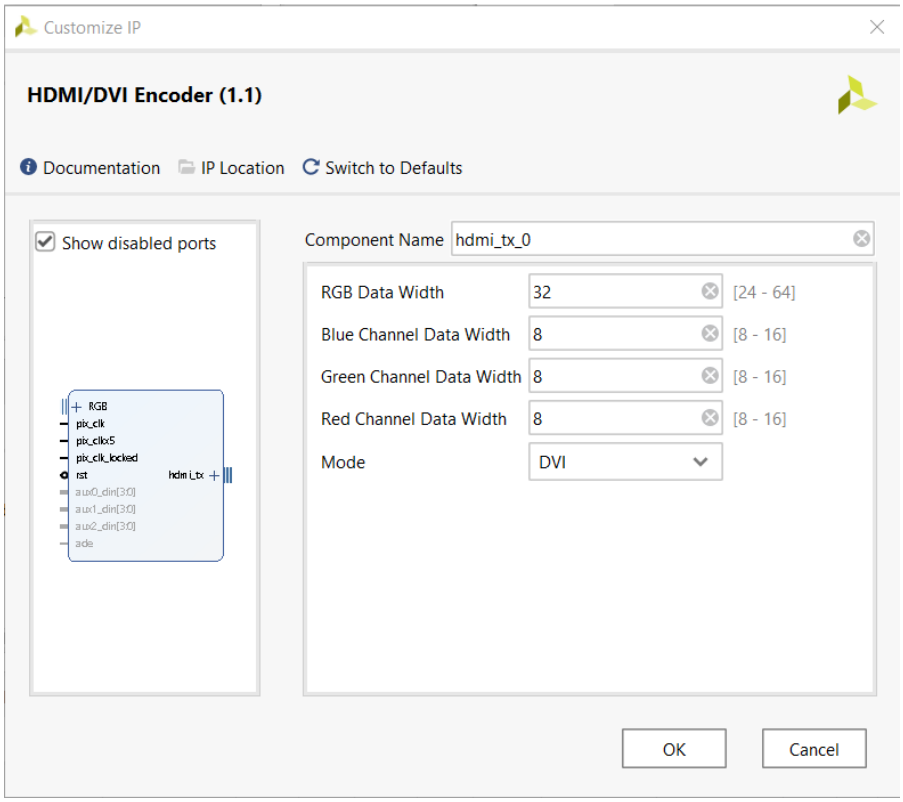
2) Under IP Repositories, click the plus sign and add the hdmi_tx_ip folder from wherever you saved it locally. The folder can be downloaded from Canvas.



3) Click apply, then OK.
4) Now, go to Project Manager -> IP Catalog. A new tab will show up on the right, which shows 1 line of User Repository. Expand User Repository -> User IP and right click HDMI/DVI Encoder, then click Customize IP.

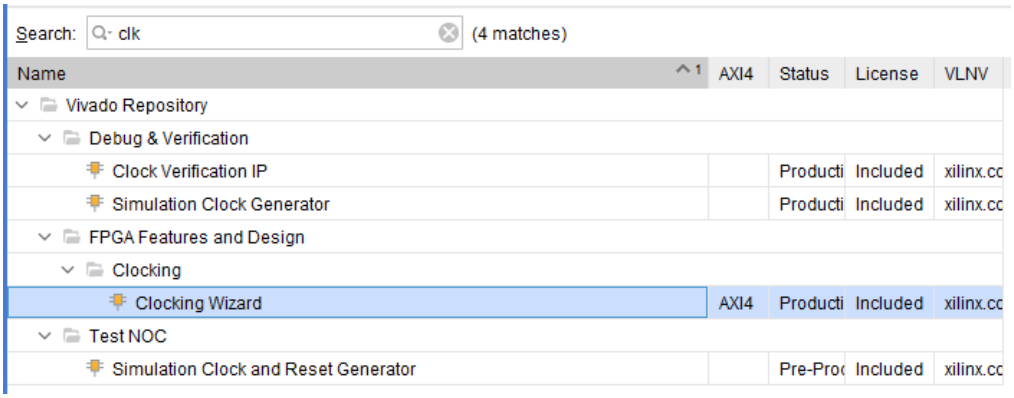


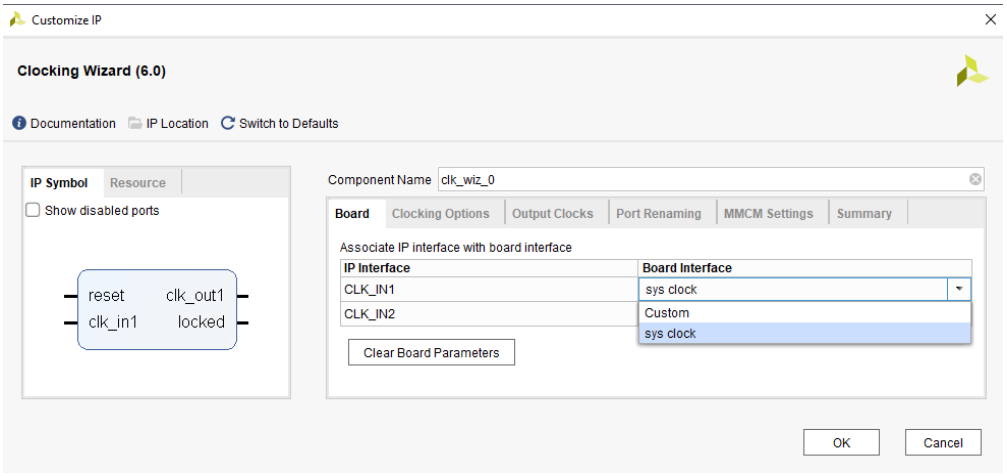
5) The Customize IP tab will show up. **Make sure the component name is hdmi_tx_0! Otherwise, your code will not synthesize.** Then, press OK twice, then Generate. You'll see a window with some progress show up, which will take a couple of minutes (maybe 3-5).

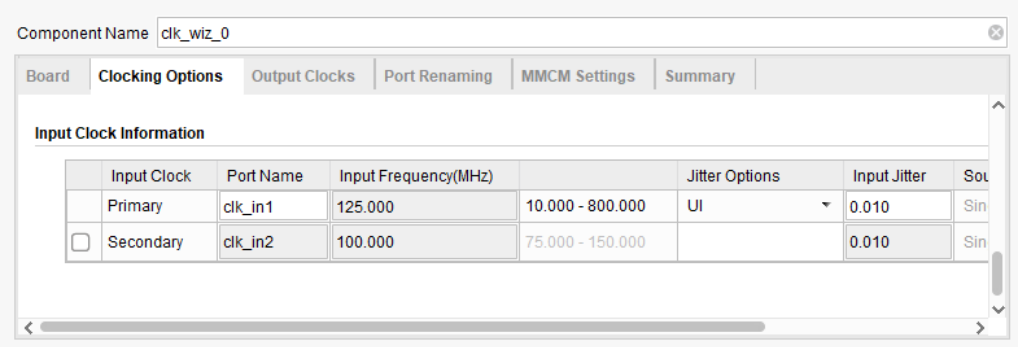


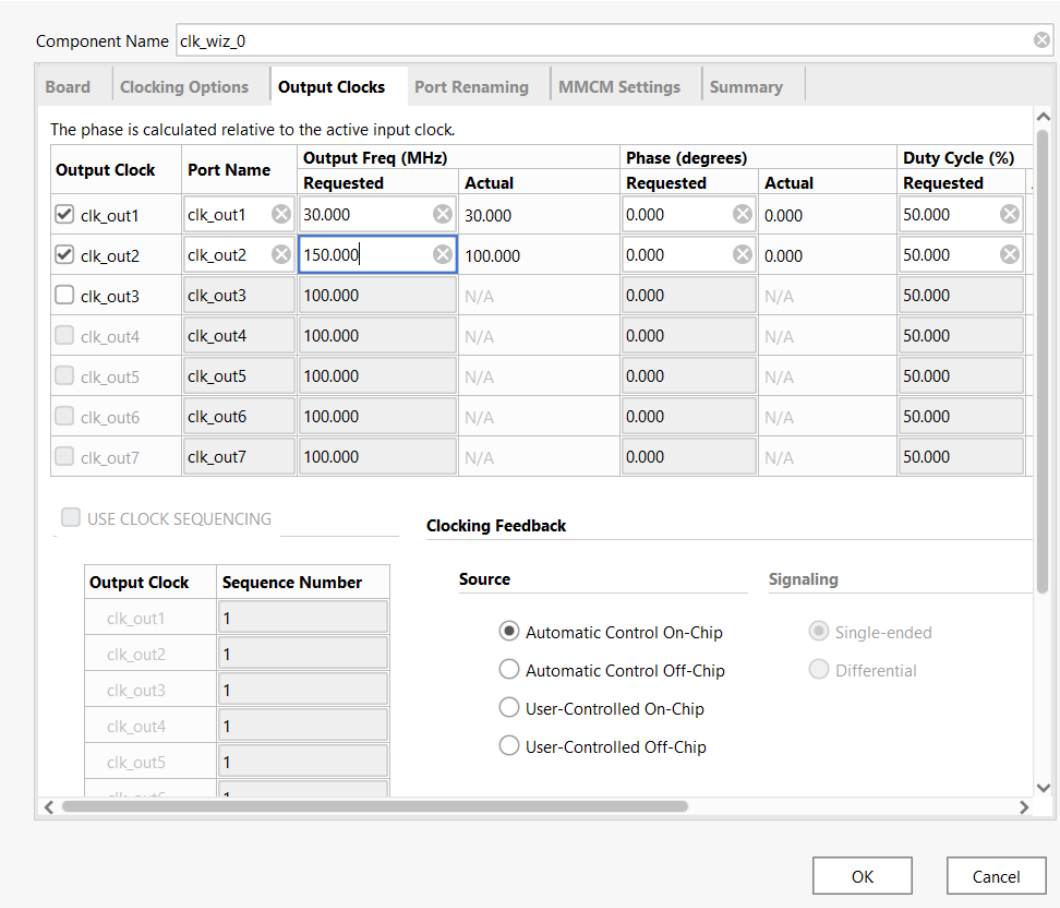
6) Now, scroll down to FPGA Features and Design -> Clocking -> Clocking Wizard and right click it. (Pro Tip: Search for "clk" to find it quickly!) Click Customize IP, and make sure the component name is **clk_wiz_0**. Under the Board tab, set CLK_IN1 to the board interface for sys clock. Double check that this was completed correctly by looking under the Clocking Options tab. clk_in1 should have an input frequency of 125 MHz. Finally, Under the Output Clocks tab,

fill in the settings pictured below (30.000 and 150.000). Then press OK, then Generate.









The Clock Wizard takes a known input clock frequency and converts it to whatever frequency we need. You'll learn more about timing and sequential logic later in the course, but for now just know that we need these clock frequencies for HDMI video output.

Remember, you'll need to do these customize IP steps for every project from here on out. Without it, your code won't synthesize.

Testing as we go along

It'll save you a lot of time and sanity to test each module in this lab right after you're finished writing it. We'll start with really simple modules and work our way up to more complex pieces of the design. This way, if a testbench fails and you have to debug your Verilog, you know the problem is in the current module under test and not from some code you had written previously.

We've written starter testbenches for each of the modules you're going to implement (the starter code files with names that end in "_tb"). They already instantiate copies of the modules and provide some

very basic test cases. Some of them are complete testbenches, while you have to add thorough test cases to others.

Specifically, you have to add test cases for the following modules:

- encoder
- arbiter
- length_finder
- rotator

The rest of the testbenches are fleshed out for you, but it is still up to you to make sure all of your modules pass these tests.

Recall from lab 0 that you can run a testbench by going to ‘Simulation Sources’ in the ‘Sources’ pane, then setting the desired testbench to run as “top.” Then, click “Run Simulation” on the far left pane. **After you finish each module, you should run its testbench (adding test cases if necessary) and ensure it behaves correctly.**

If the text written to the simulation log looks incorrect you'll have to debug your implementation. The wave viewer will be very useful for this. A good place to start is to look at all of the top-level testbench signals – click on the top-most node in the Scope pane, select all the signals from the Object pane, and drag them onto the signal viewer. Then, click restart, then run all. Understanding how to use this

viewer will be essential while debugging your Verilog. If you need more help with the waveform viewer, please refer back to lab 0, or ask a TA.

Before submitting, make absolutely sure that your module passes the top level "verifier_tb" testbench. Even if all of your submodules are working perfectly, there still could be errors even at the top-level module which you didn't catch!

3. Implementation

Binary Encoder

The binary encoder takes an 8-bit one-hot input and outputs the index of the bit set to '1', as a 3-bit binary number. For example, '00000001' is 0 (3'b000), '00000010' is 1 (3'b001), '00000100' is 2 (3'b010), etc.

You're going to want to use a **case statement** here. But keep in mind you can only put case statements inside of **always blocks**, and if you give a signal a value inside an always block it **must be a reg**. And one last thing to remember: all case statements need a **default case**.

Here's the skeleton, you should fill in the blanks and save the result in "encoder.v":

```
module encoder ( _________ _______ [___:___] in,
                 _________ _______ [___:___] out);

// module body goes here

endmodule
```

Arbiter

This arbiter is going to take an arbitrary 8-bit input and give us a one-hot 8-bit output. The least significant '1' in the input is also a '1' in the output, and all other output bits are '0'. Essentially, the arbiter is "choosing" the least significant '1' bit of the input and letting it pass through, while blocking all the other input '1's. Arbiters are good at choosing things.

Just some examples: 8'b00011010 → 8'b00000010, 8'b00100000 → 8'b00100000, 8'b00000000 → 8'b00000000, 8'b11111111 → 8'b00000001.

The arbiter is going to look very similar to the encoder, except we're going to use a **casex statement** instead of a case statement. The only difference is we can write 'x's in the case labels as well as '0's and

'1's to indicate we **don't care** what value that bit has - the case applies if the bit is 0 or 1. As an example, in the casex block:

```
casex (value)
      4'bxxx1: //case a
      4'bxx10: //case b
      default: //case c
endcase
```

Any `value` with '1' in the first position will match `case a`, such as 0001, 0011, and 1001. Failing that, any `value` with a '1' in the second position will match `case b`. This includes 0010 and 1010, but **not** 0111. Can you figure out why?

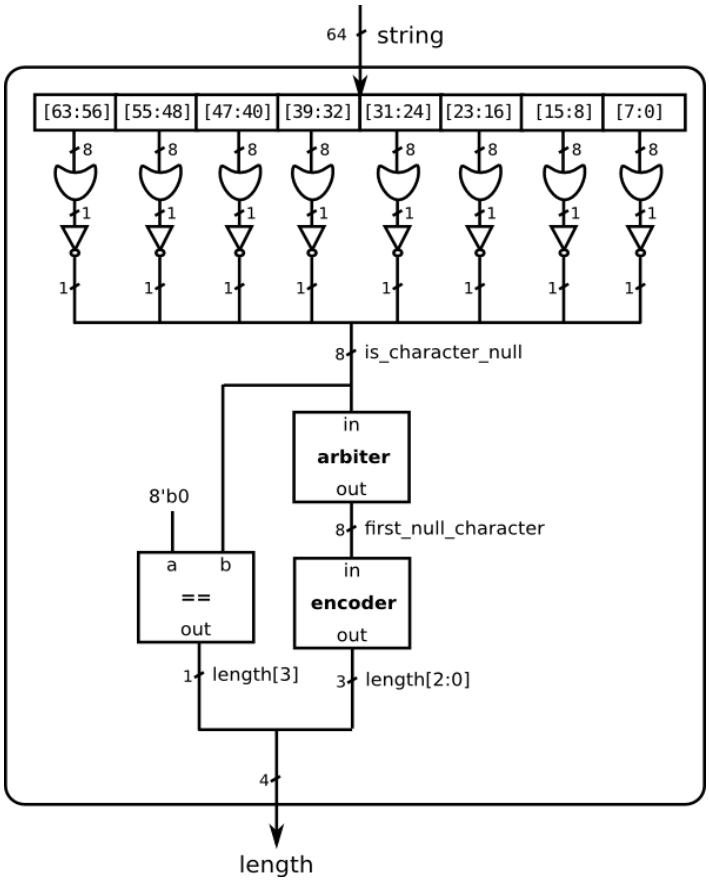
Anyway, let's implement an 8-bit arbiter using a casex statement. The same considerations from our encoder apply here, but additionally we need to make sure we **select the least significant '1'** and not the most significant '1'.

```
module arbiter ( _________ ________ [___:___] in,
                 _________ ________ [___:___] out);

// module body goes here

endmodule
```

Put this in “arbiter.v”. You know the drill.

Length Finder

Alright, we have everything we need to build our string length finder module. Our goal is to find the first byte that is all '0's, and output its position in the original input. This one is a little more complicated than the arbiter and encoder, so let's organize our thoughts visually with a block diagram:



Remember, our Verilog module is nothing more than a textual representation of this block diagram. A good place to start is with our wire and reg declarations. Write wire declarations for all of the named lines in this block diagram. Make sure to specify the width of the wire if it's more than 1 bit.

```
module length_finder ( _________ ________ [___:___] string,
                       _________ ________ [___:___] length);

// wire declarations

endmodule
```

We'll need to assign a value to each bit of the is_character_null signal. Just a reminder, each 8-bit chunk of our input is a character and the character is null if all 8 bits are 0. There are many ways to check if all 8 bits are 0, so we're going to choose a more complex one that flexes our Verilog skills: if we boolean-OR all 8 bits together, the result will be 0 if and only if all the bits were 0. Then we'll invert this result to get a bit that is 1 if the byte was null, and 0 if otherwise.[4]

Try writing the assign statement for the first is_character_null bit using a **unary OR operator** (|), **bitwise invert operator** (~), and **range select** (signal_name[high_index:low_index]). Just a refresher on unary OR – when you place an OR symbol to the left of a signal name, it has the effect of smashing

[4] How could you rewrite this zero-check using De Morgan's laws? How about using as few typed characters as possible? There are many different ways to do one thing in Verilog – in general go for the one that *looks* the most clear to you.

its individual bits together with binary OR into a single bit output. We've got 8 input bytes, so you're going to need 8 of these assign statements. At this point your module should look like this:

```
module length finder ( _________ ________ [___:___] string,
                       _________ ________ [___:___] length);

// wire declarations

// is_character_null logic (8 lines of it!)

endmodule
```

At this point we should have 8 bits indicating the presence of each null byte in our input string. We have to feed that into an arbiter, whose output will tell us where the *first* null byte is in the input. We've already written the arbiter module, so just instantiate one here. All 8 bits of our is_character_null signal form its input, and you should have already declared a wire to carry its output. Remember, the syntax for module instantiation is:

```
module_name this_specific_instance_name (
      .port_1_name (some_wire),
      .port_2_name (some_other_wire),
      …
);
```

And now we're almost there!

```
module length finder ( _________ ________ [___:___] string,
                       _________ ________ [___:___] length);

// wire declarations

// is_character_null logic (8 lines of it!)

// arbiter

endmodule
```

This arbiter is still outputting a one-hot signal, though. We need to convert that into a binary number telling us how many characters are in the string. An encoder will do exactly this! Instantiate one just

like you instantiated the arbiter. Hook the output of the arbiter up to the input of the encoder, and the output of the encoder up directly to the 'length' output of our module.

But hold it, the length output is 4 bits. The encoder's output is only 3 bits. Think about why this is – if our string is 8 characters long, there will be no space for a null byte. All of the is_character_null signals will be 0. So we're going to need a few tricks to make this work.

First of all, our encoder should drive the 3 LSBs of the 'length' output. Use range select notation in the port declaration to do this:

```
module_name this_specific_instance_name (
      …
      .three_bit_port (four_bit_wire[2:0]),
      …
);
```

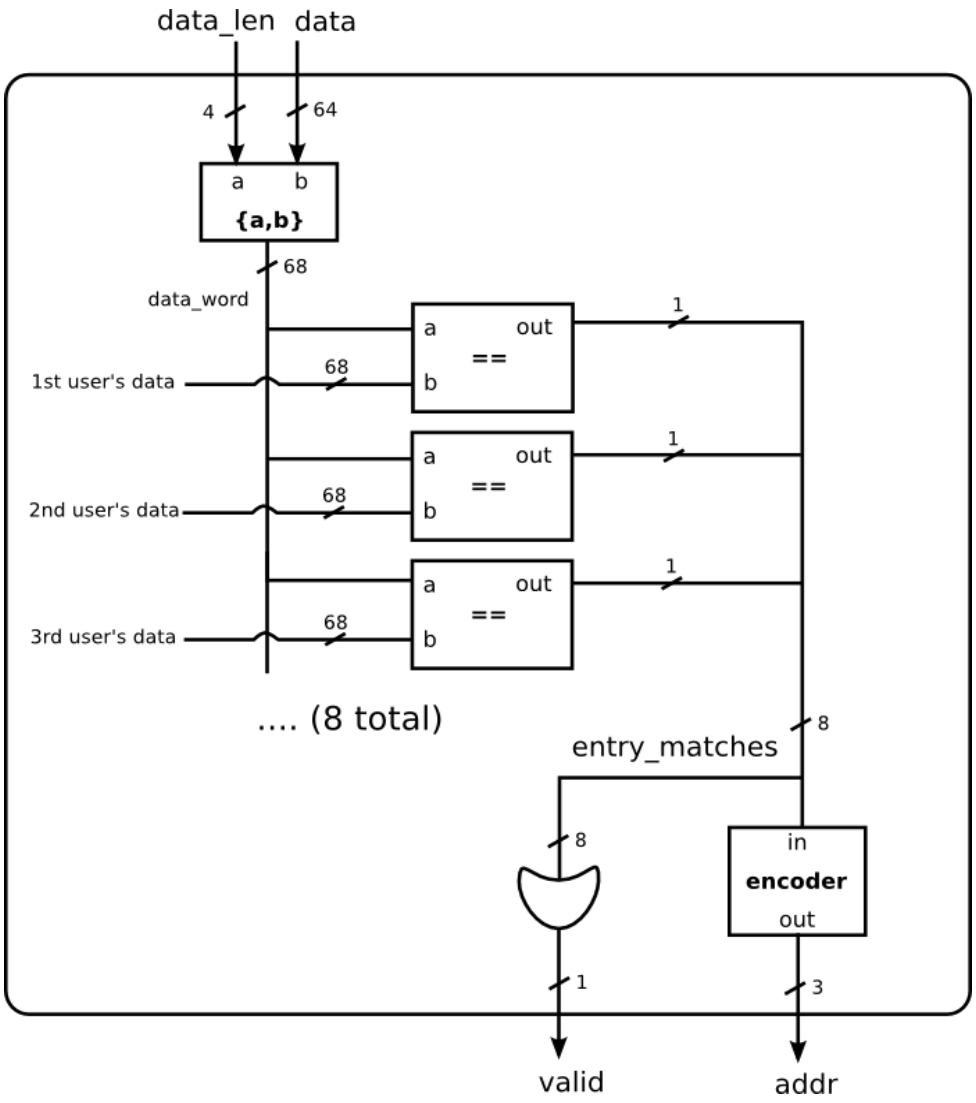
Finally, we have to define the logic for the fourth bit of the 'length' output. Use an **assign statement** and an **equality comparator** (==) to make **length[3]** equal to 1 only when all bits of is_character_null are 0. Don't use an always block here – just an assign statement.

Let's think about what this is doing – when the string is 8 characters long, its length will be the binary number 4'b1000. The 3 LSBs of that length are driven by our encoder. The MSB is driven by our assign statement. The MSB will only ever be 1 when all of the LSBs are 0's. This accomplishes what we want.

Username CAM

Our CAM takes in the username that was just entered and tells us if it's in our user database, and it's going to do this by comparing the username string to each possible valid username that could be entered. Remember that because this is hardware, we can do each of these comparisons in parallel!

So, let's take a look at the block diagram:



Because this is hardware and our input data is 64 bits large no matter what we do, we append the length of input username to the most significant end of the string to distinguish between short strings that are padded out to 64 bits with 0s and longer strings that look the same but were actually intended by the

user to have 0s at the end of them. It's not actually necessary in this lab, but this technique of appending metadata to our input is very useful when using CAMs out in the real world.

Anyway, starting like we did last time, declare wires for all of the named signals you see in the block diagram.

```
module cam ( _________ ________ [___:___] data,
             _________ ________ [___:___] data_len,
             _________ ________ [___:___] addr,
             _________ ________ valid );

// wire declarations

endmodule
```

Next we'll glue data and data_len together. Concatenation in Verilog looks exactly the same as it's drawn in the block diagram: a comma-separated list of signals within curly braces ({}s). Use curly brace concatenation and an assign statement to form our query word:

```
module cam ( _________ ________ [___:___] data,
             _________ ________ [___:___] data_len,
             _________ ________ [___:___] addr,
             _________ ________ valid );

// wire declarations

// data concatenation

endmodule
```

Now we implement the comparison against each username in the database. Use 8 **assign** statements to connect each bit of entry_matches to logic that outputs a 1 if the query word equals the corresponding username in our database and is the correct length. For example, entry_matches[2] should be 1 if and

only if the input data was the string 'HOLLY' and was 5 characters long (that's not including the trailing null). For convenience, here's our user database again:

	Username			Password	
#	Text	Hex		Text	Hash
0	LEO	64'h00000000004F454C		DFA1979	32'hDC1A2C9E
1	AARON	64'h0000004E4F524141		COWSRULE	32'hDC2EA8E4
2	HOLLY	64'h000000594C4C4F48		RUNRUN	32'h355FACC3
3	DAVID	64'h0000004449564144		ICARUS	32'hAAF4ADC9
4	CLAIRE	64'h0000455249414C43		XIEXIE	32'h13D41CED
5	FRANK	64'h0000004B4E415246		P3L3GRNO	32'h7EBCF8A8
6	LANCE	64'h00000045434E414C		F0RNWALT	32'hF3CDDB9B
7	RYAN	64'h000000004E415952		AR1STA	32'h9948E6BE

Don't just copy the 64 bit username hex values into your module – put the length of the string at the most significant end to make them 68 bits long. For clarity, Verilog allows you to put underscores (_) anywhere within number constants to make them more easy to read. For example:

```
68'h4_000000004E415952 // {4, "RYAN"}
```

```
module cam ( ________ ________ [___:___] data,
             ________ ________ [___:___] data_len,
             ________ ________ [___:___] addr,
             ________ ________ valid );

// wire declarations

// data concatenation
// 8 equality comparisons

endmodule
```

Now we need to take these bits and turn them into a binary number. Let's instantiate another one of those encoder modules, just like we did in the length finder.

```
module cam ( ________ ________ [___:___] data,
             ________ ________ [___:___] data_len,
             ________ ________ [___:___] addr,
             ________ ________ valid );

// wire declarations

// data concatenation

// 8 equality comparisons
```

```
// encoder instantiation

endmodule
```

But what happens if the input username isn't in our database? Our 'valid' output should be 1 only if the input data actually matched against something in our database. Write the logic for the 'valid' output using an **assign statement** and a **unary OR**.

```
module cam ( _________ ________ [___:___] data,
             _________ ________ [___:___] data_len,
             _________ ________ [___:___] addr,
             _________ ________ valid );

// wire declarations

// data concatenation

// 8 equality comparisons

// encoder instantiation

// valid logic

endmodule
```

Stored password ROM

Recall that our password ROM is the inverse of our username CAM – where the CAM takes data and gives us the data's address in our database, our ROM takes an address and gives us the data. In this case, the data is the hash of each user's password.

Implement the password ROM using a **case statement**. Remember to use an always block, and remember that any signal changed inside an always block must be a **reg**.

```
module hash_rom( _________ ________ [___:___] addr,
                 _________ ________ [___:___] data );

// Module body

endmodule
```

For convenience, here's the database again:

	Username			Password	
#	Text	Hex		Text	Hash
0	LEO	64'h00000000004F454C		DFA1979	32'hDC1A2C9E
1	AARON	64'h0000004E4F524141		COWSRULE	32'hDC2EA8E4
2	HOLLY	64'h000000594C4C4F48		RUNRUN	32'h355FACC3
3	DAVID	64'h0000004449564144		ICARUS	32'hAAF4ADC9
4	CLAIRE	64'h0000455249414C43		XIEXIE	32'h13D41CED
5	FRANK	64'h0000004B4E415246		P3L3GRNO	32'h7EBCF8A8
6	LANCE	64'h00000045434E414C		F0RNWALT	32'hF3CDDB9B
7	RYAN	64'h000000004E415952		AR1STA	32'h9948E6BE

Rotator

We're going to need a bit rotator module for our hash function. A rotator shifts bits over like a conveyor belt, with bits falling off one end being pushed onto the other. 00010 rotated to the right twice is 10000, 01101 rotated to the left once is 11010, and so on.

We need two different sizes of rotator – one for 8-bit signals in each round, and one for the 32-bit hash state at the very end. Instead of implementing two separate modules, we'll just make one and **parameterize** it. Parameters are variables within a module that you can change from one instantiation to another but are fixed in place after we synthesize into hardware. Here is a quick example of a parameterized (yet completely useless) module:

```
module add_N #(parameter WIDTH=8, parameter N=5)
(  input wire [WIDTH-1:0] in,
   output wire [WIDTH-1:0] out
);
      assign out = add + N;
endmodule
```

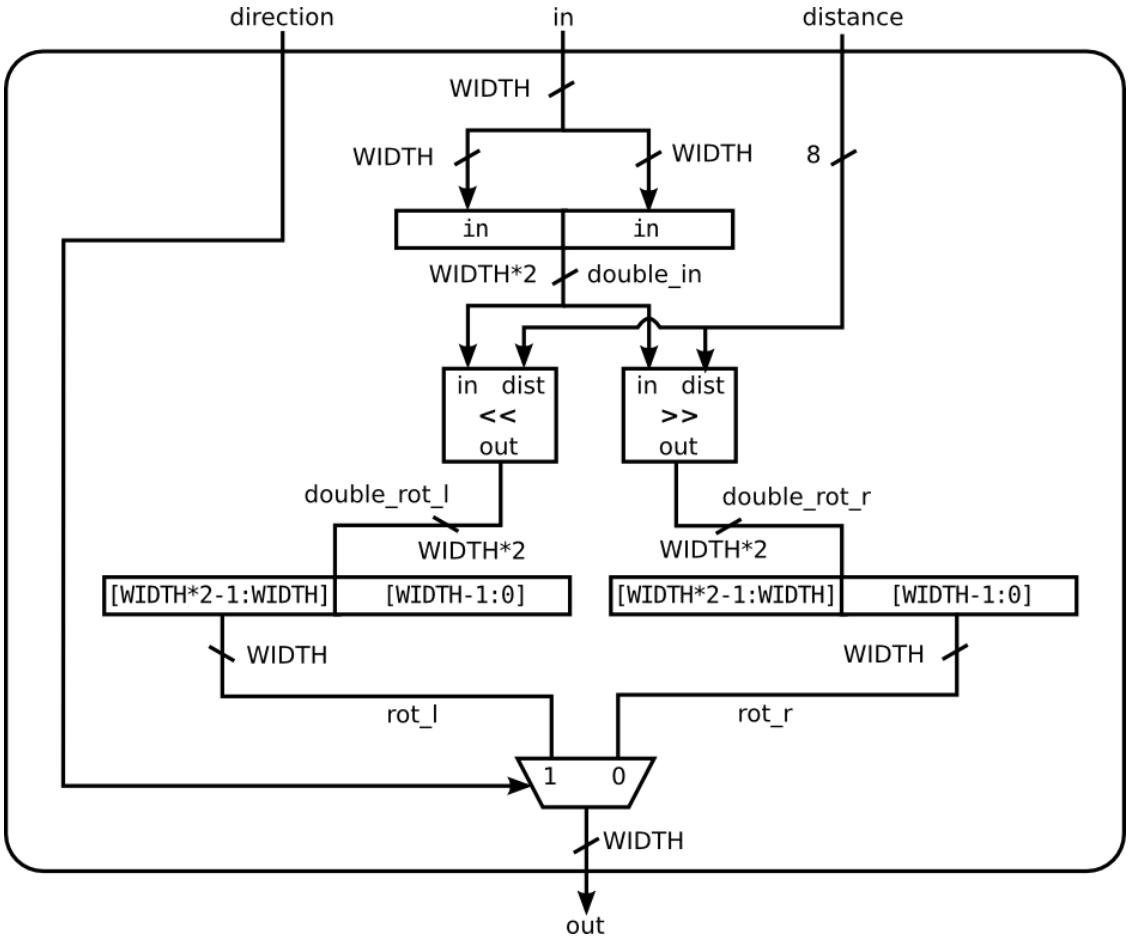
The module's parameters are defined in the parameter list (the list at the top preceded by a #), along with their default values. We can use these parameter names anywhere in the module we would

normally put those numbers. If we just instantiate add_N like a normal module, WIDTH and N will be replaced with 8 and 5 during simulations and synthesis. We can also do something like this:

```
add_N #(.WIDTH(16), .N(2)) rad_adder (.in(blah), .out(blahblah));
```

This instance of add_N will expect 16-bit inputs and outputs, and add 2 to the input. We can leave parameters out of the list when we instantiate the module and they'll just take on their default values. If we leave the #() list out of the instantiation entirely, everything will be default.

So let's get started on our parameterizable rotator. Here's the block diagram:



Even though the width is parameterizable, to simplify things we're going to make the distance input fixed at 8 bits. Please feel free to parameterize that too, if you want to!

Notice that we're going to end up with hardware that computes the rotation in **both directions**, rot_l and rot_r, even though we're only going to use one of the results. This is another reminder that we're

not writing a computer program here – both of these circuits are going to exist and be computing their outputs all the time, likely right next to each other.

First things first, we'll need to write in any parameters and wire declarations we see in the block diagram.

```
module rotator //parameter list goes here
( ________ _______ [___:___] in,
  ________ _______ [___:___] out,
  ________ _______ [___:___] distance,
  ________ _______ [___:___] direction
);

// Declarations

endmodule
```

To create two copies of 'in' right next to each other we can use Verilog's **replication operator**:

```
// {N{m}} outputs N copies of m, back to back. This is
// equivalent to: {m, m, m, … } if we wrote m N times. So:
wire [2:0] sig = 3'b110;
wire [8:0] bigsig = {3{sig}}; // = 9'b110110110
```

Use the replication operator to drive 'double_in'. Then use **logical shift operators** (<< and >>) to drive 'double_rot_l' and 'double_rot_r'. Use **range select notation** to split off the most significant and least

significant halves of these into 'rot_l' and 'rot_r', respectively. You'll want to use **assign statements** for each of these signals.

```
module rotator //parameter list goes here
( _________ ________ [___:___] in,
  _________ ________ [___:___] out,
  _________ ________ [___:___] distance,
  _________ ________ [___:___] direction
);

// Declarations

// Rotating logic

endmodule
```

Now we choose which rotation result to use based on the direction input. Instead of implementing the mux with an always-block/if-statement, lets use an assign statement and the **ternary operator**. As a reminder, the ternary operator looks like this:

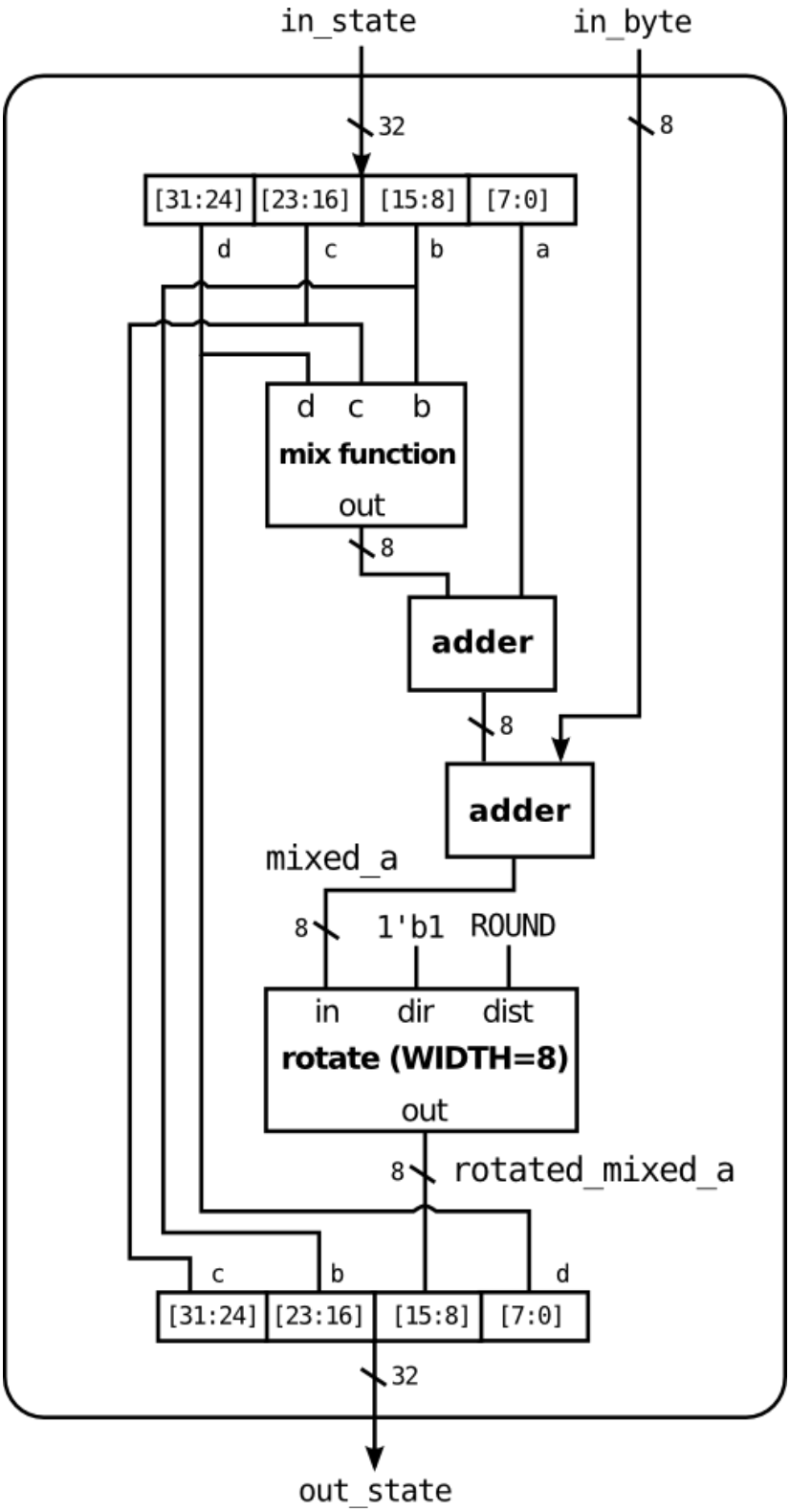
```
condition ? chosen_if_condition_is_1 : chosen_if_condition_is_0
```

Now we should have a bit rotator:

```
module rotator //parameter list goes here
( _________ ________ [___:___] in,
  _________ ________ [___:___] out,
  _________ ________ [___:___] distance,
  _________ ________ [___:___] direction
);

// Declarations

// Rotating logic

// Output mux logic

endmodule
```

Hash Round

Let's take a look at the block diagram for a hash round again:



Okay, so first things first. Let's do port and wire declarations. We're also going to make the round number a parameter, so we can use the same module for all rounds.

```
module hash_round //parameter list goes here
(  _________ ________ [___:___] in_byte,
   _________ ________ [___:___] in_state,
   _________ ________ [___:___] out_state
);

// Declarations

endmodule
```

Now let's split up the input state into our a, b, c, and d bytes using **compound assignment**: The same way we can smash signals together into one when we use {}s on the right side of an equals sign, we can split one signal into many smaller ones using {}s on the left side of an equals sign. For example:

```
wire [2:0] three_MSBs;
wire       middle_bit;
wire [5:0] six_LSBs;
wire [9:0] ten_bit_signal;
assign {three_MSBs, middle_bit, six_LSBs} = ten_bit_signal;
```

Use compound assignment to split in_state into a, b, c, and d. Take care to put things in the right order – d is the most significant byte of the state.

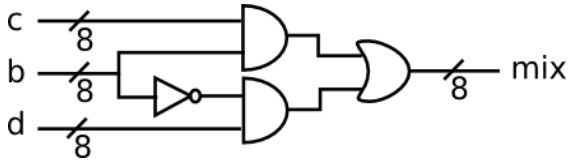
```
module hash_round //parameter list goes here
(  _________ ________ [___:___] in_byte,
   _________ ________ [___:___] in_state,
   _________ ________ [___:___] out_state
);

// Declarations

// State splitting

endmodule
```

Next, we have all the declarations in place for mixed_a's value, so on the line you declare mixed_a instead of ending immediately with a ";", put an "=" and write out the sum shown in the block diagram.

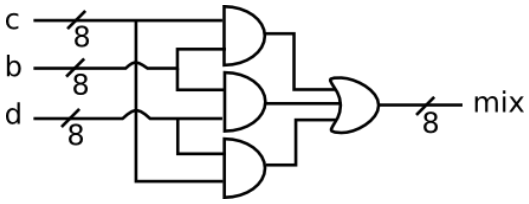
This will save us having to write an assign statement for it later. You'll want to use the **plus operator (+)** here, so you don't have to write out the logic for an adder yourself.

The mixing function we use will change every few rounds:

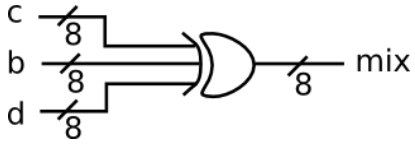
- For rounds 0-2:



- For rounds 3 and 4:



- For rounds 5-7:



(that's XOR)

Use an always block and an if statement to assign 'mix' depending on what the ROUND parameter is. Use **bitwise operators** (recall that OR is "|", AND is "&", NOT is "~", and XOR is "^").

```
module hash_round //parameter list goes here
(  _________ ________ [___:___] in_byte,
   _________ ________ [___:___] in_state,
   _________ ________ [___:___] out_state
);

// Declarations

// State splitting

// Mix function

endmodule
```

More than likely, any FPGA synthesis tool will see that the round number is constant at runtime (because it's a parameter) and during synthesis optimize out each hash_round's unused mixing

functions. This is good! If you read the messages Vivado generates during synthesis, you might even see where it did this.

We need to instantiate a rotator to shuffle around the bits of our mixed_a sum. Put that in and remember to set the width parameter of the module instance using a #(...) list – we want to rotate an 8-bit signal here.

```
module hash_round //parameter list goes here
(  _________ ________ [___:___] in_byte,
   _________ ________ [___:___] in_state,
   _________ ________ [___:___] out_state
);

// Declarations

// State splitting
// Mix function
// Rotator

endmodule
```

Like the round number mux/if-statement we wrote earlier, notice that the 'direction' bit is fixed at a *constant* 1 here. The rotate-right logic will likely be optimized out in this specific rotator instance.

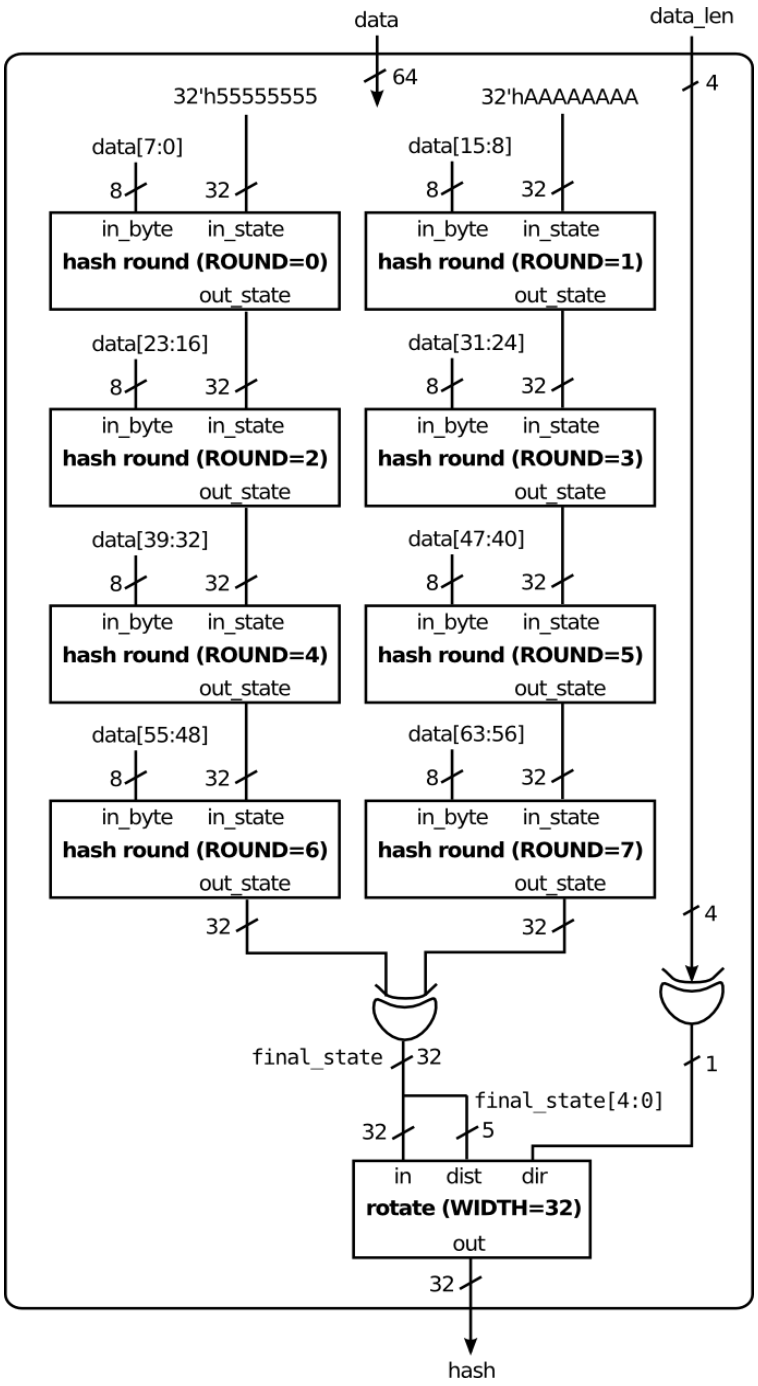
Finally, we need to combine everything into our out_state. Let's use an assign statement concatenation syntax (“{}”s) to do the opposite of our compound assignment at the top of the module. Combine rotated_mixed_a, b, c, and d in the order shown in the block diagram and assign it to out_state.

```
module hash_round //parameter list goes here
(  _________ ________ [___:___] in_byte,
   _________ ________ [___:___] in_state,
   _________ ________ [___:___] out_state
);

// Declarations

// State splitting

// Mix function

// Rotator

// Output state assignment

endmodule
```

Any tiny mistakes in this module, even ones that will synthesize without warnings, will cause your hasher to produce completely different output than expected. It'd be a good idea to look over your HDL and the block diagram one more time to make sure everything matches.

Hasher

Our hasher is just going to string a bunch of instantiations of hash_round together, one after the other, and feed the result into a 32-bit wide rotator. Reproducing the block diagram from earlier:



You know the drill at this point. Make sure to set the parameters on the hash_rounds and rotator you instantiate – other than that, nothing you haven't done already! In the above diagram, the rotator's `dist`

input has width 5. If you didn't parameterize the `dist` input of rotator, then your rotator's dist input is of width 8. You can address this by padding 3 zeros to the three MSBs of the `dist` input.

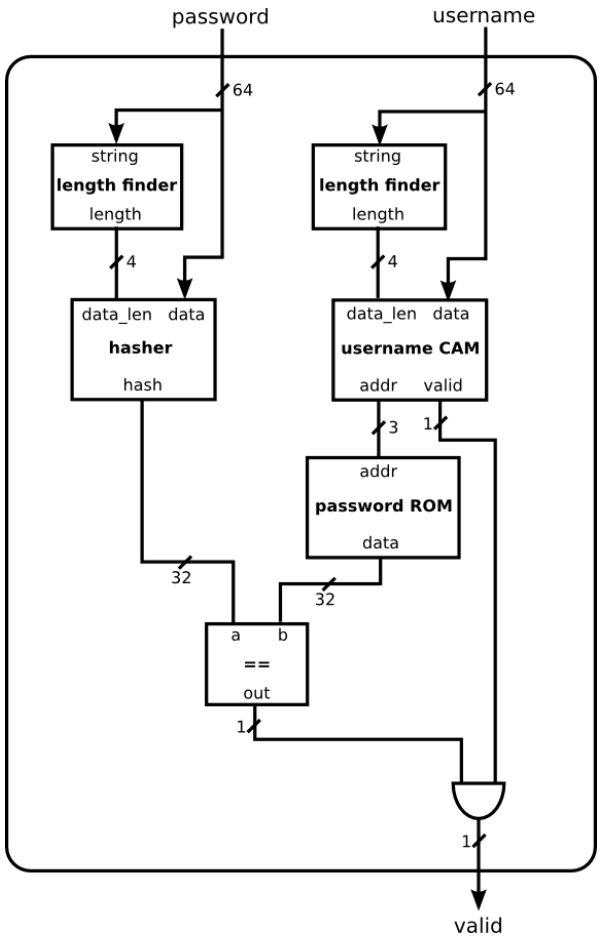
A tip to save lines: if you're declaring more than one wire or reg of the same width Verilog allows all of the declarations on one line separated by commas, like this:

```
wire [7:0] first, second, third; // Three 8-bit wires
```

```
module hasher
(  _________ ________ [___:___] data,
   _________ ________ [___:___] data_len,
   _________ ________ [___:___] hash
);

// DO IT GORDON

endmodule
```

Top level module

Our top level verifier module is even simpler than the hasher. Here's the block diagram again:



You've written all of the modules in the diagram. You know what to do.

```
module verifier
(   _________ ________ [___:___] username,
    _________ ________ [___:___] password,
    _________ ________           valid
);


endmodule
```

4. Testing on the FPGA

Notice: This section of the lab is optional if no one in your group has a lab-kit. If you do have a lab-kit, share your results and experience with your teammates.

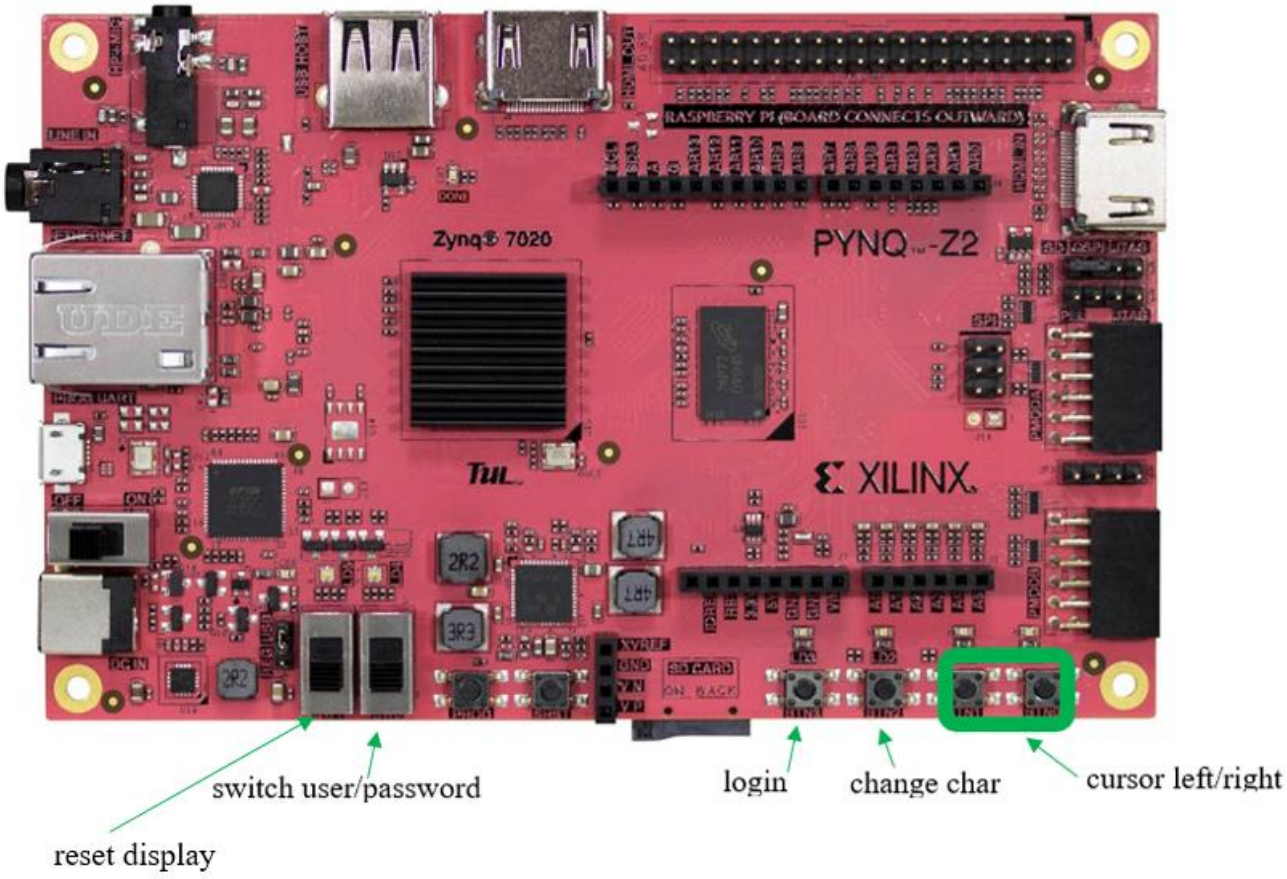
Once you're **positive** your design is working according to your testbenches, you're ready to synthesize. Synthesize your design according to the process outlined in Lab 0. Make sure 'lab1_top' is selected as

the top-level module in the hierarchy before you generate the bit file. Once everything is complete, download your design to the FPGA, as described in lab 0 using the Vivado hardware manager.

You should see two text boxes labeled USERNAME and PASSWORD appear on the screen. If you don't see anything, make sure SW1 is in the down position. You can also toggle SW1 to refresh the display.

Use the right two pushbuttons (BTN0/BTN1) to move the cursor left and right. You can use BTN2 to change the currently highlighted character, and SW0 to choose whether to enter the username or password.

When you've typed in everything, hit the leftmost push button (BTN3) to send the username and password off to your account verifier! Try logging in as a few different valid accounts from the database, and a few invalid accounts as well.



5. Lab Submission (due on 9/30/20 at 11:59 PM)

Make sure to back up your files before following the submission instructions below. Don't delete your original files as you will be working with them again during the lab session.

Create a **.zip** file with the following contents:

1. All of your Verilog files (*.v files), for both your modules and testbenches. Make sure you include the correct versions of your files! Sometimes Vivado duplicates your files and caches old versions. This includes:

- arbiter.v, arbiter_tb.v
- cam.v, cam_tb.v
- encoder.v, encoder_tb.v
- hash_rom.v, hash_rom_tb.v
- hash_round.v, hash_round_tb.v
- hasher.v, hasher_tb.v
- length_finder.v, length_finder_tb.v
- rotator.v, rotator_tb.v
- verifier.v, verifier_tb.v

2. Your bit file, located at lab1/lab1.runs/impl_1/:
   lab1.bit **(Submission of this file is optional if no one in your group has a lab-kit)**
3. A file lab1_sims_<group_number>.pdf containing the text output of **all** testbench simulations running in ISim. This includes the testbenches you didn't modify! Annotate the text to point out your new test cases and what they test for. To see examples of good annotations, refer back to lab 0.

Name the archive either lab1_group<group_number>.zip and submit on Gradescope. Only one person in your group should do this, and they should tag their teammates on the submission. Make extra sure that **everything listed above** is included in the .zip. No need to include any files not listed above.

Be mindful that the TAs will test your design using randomly generated inputs, both valid and invalid inputs.

5. Demo

Notice: For your demo material, you will be sending us your design and we will verify that it works. Also before submission, make sure that your lab1_top_tb.v test bench produces the expected results and that all the different ranges of input cases are covered in your tests. Finally, it is encouraged but not required to upload your lab1_top.bit file to the FPGA using Vivado hardware manager to test the expected functionality especially if your group has a lab kit.

Upload answer to the following questions in a **.txt** or **.pdf** file:

1. An FPGA is made up of a huge number of objects called “slices,” which consist of look-up tables (LUTs), flip-flops (we'll learn about these later), and some extra specialty hardware. The FPGA implements your design by setting the values of the LUTs and wiring the slices together, subject to complex constraints (that's why it takes so long to synthesize).

   After synthesizing, go to “Window -> Reports” and double click on “impl_1_place_report_utilization_0.”
   - What fraction of the FPGA's slice LUTs did your implementation consume? You can find this information under section 2: Slice Logic Distribution.
   - How many slices are occupied?

The video and input wrapper code provided for you takes up most of this space, but it's still worth it to begin being aware of what kind of resources we're using.

2. Open Window -> Reports -> **synth_1_synth_synthesis_report_0** and text search (ctrl+f) for “latch”. This report will let us know if the design is accidentally inferring any latches with the

warning “WARNING: [Synth ____] inferring latch for variable ‘_____’” where the blanks will change depending on what signal is being accidentally latched. If you have any of these warnings, you must fix all of them before submitting your files. If you did infer any latches, report where they were and how you removed them.

3. On the left side panel, click on “Implementation -> Open Implemented Design.” After a bit of time, it will display a visualization of the FPGA itself, giving you an idea of what you actually built. You should see the slices being used highlighted in bright blue, and if you zoom in can even see the individual components being used in each slice. Submit a screen shot of the overall layout, and a screen shot of a single LUT slice.

4. Explain the difference between synthesizable and non-synthesizable Verilog. Give examples of pieces of syntax that you have used already that are non-synthesizable (you know of at least two).

5. Please include any comments about this lab so we can make it better in the future!

Then create a **.zip** archive with the following contents:

- The text file with answers to the lab questions above.
- Your FPGA layout screenshots.
- The synthesis and timing reports. The synthesis report is the report you read in step 2 earlier. The timing report isn’t super meaningful to us now, but will be starting in Lab 3. These files can be found in your lab0 directory at:
  - lab1 -> lab1.runs -> synth_1 -> **lab1_top.vds**
  - lab1 -> lab1.runs -> impl_1 -> **lab1_top_timing_summary_routed.rpt**

Upload this archive to Gradescope under the lab 1 assignment along with the **.zip** file you created in the steps previous to the demo.

6. Appendices

So what?

Believe it or not, even though the final design you produced in this lab seems useless (and don't worry, it is), the work you've done could actually be spun into something useful. If you wanted to **reverse** a hash, one way to do so would be to just hash a whole boatload of inputs until you found one that produced the correct output. This is called a **brute force attack**, because it takes a whole lot of effort and there is absolutely nothing intelligent about it.

Brute force attacks in software take a long time to carry out and only allow us to try one input at a time (or one input per core, whatever). In hardware, however, you could instantiate hundreds or thousands of your hashing module, depending on how large your FPGA was, and produce hashed outputs much faster than the equivalent software.

The orders of magnitude increase in speed and parallelism that dedicated hardware bring make a brute force attack much more feasible, and in fact professional bad guys (or good guys, depending on your

political ideology) do this all the time. A long time ago, the final project for this class was to use a similar technique to break the encryption on RSA-encrypted text files.

A note about === and !==, the strict equality operators

In the testbenches we've written for you, you'll see the special "===" and "!==" operators. These operators are **only for testbenches** and are **not synthesizable**. They allow us to catch when broken outputs from your module are undefined (show up as either X's or Z's in your simulation).

If you were to use the normal equality operator "==" with an undefined signal, the result would also evaluate to undefined, and the if statement condition would evaluate to false. If you're using if statements in your testbenches to check if outputs are correct, using "===" and "!==" protects against mistakes like this.

It's okay if you don't understand this. If you would like to but you're still confused, though, please ask your TA!