EECS150: Lab 5, Mini-Project

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

1 Time Table

ASSIGNED	Friday, September 25^{th}
DUE	October $12^{th} - 13^{th}$, during your assigned lab section

2 Motivation

This lab you will teach you to use the ChipScope Integrated Logic Analyzer to help you debug your designs once they have been moved to hardware. Last week you learned about Simulation, specifically with Modelsim. Simulation is an invaluable tool for verifying designs that you can place in a controlled setting (e.g. a "Testbench"). Some designs, however, require board-level testing. The prime example of such a design would be one that communicates off-FPGA to some external device. As we cannot typically model the communication between the FPGA and the off-FPGA device, and sometimes can't even model the off-FPGA device itself, we cannot encapsulate these designs in a testbench. As a result, we cannot simulate these designs, and must resort to other means of debugging.

In this lab, you will see an example of a design that can't be fully simulated and requires board-level testing. Specifically, you will be building a communications protocol between a "UART" (Universal Asynchronous Receiver and Transmitter) an MIPS CPU on the FPGA. The UART will allow you to communicate with the FPGA from your laptop or desktop computer. This interface will be identical to the one that you use in your project to interface your MIPS150 CPU to a computer.

Through this lab, you will have to perform simulation in Modelsim and direct on-board testing to make sure everything works correctly. You will also learn about the basics of the UART[1] and how you can get information on and off the FPGA. Lastly, you will build a small but important part of your final project. After you complete this lab, you will have come full circle with the CAD tools used in this class and be ready to start the project.

3 Coming "Full Circle" with the CAD Flow

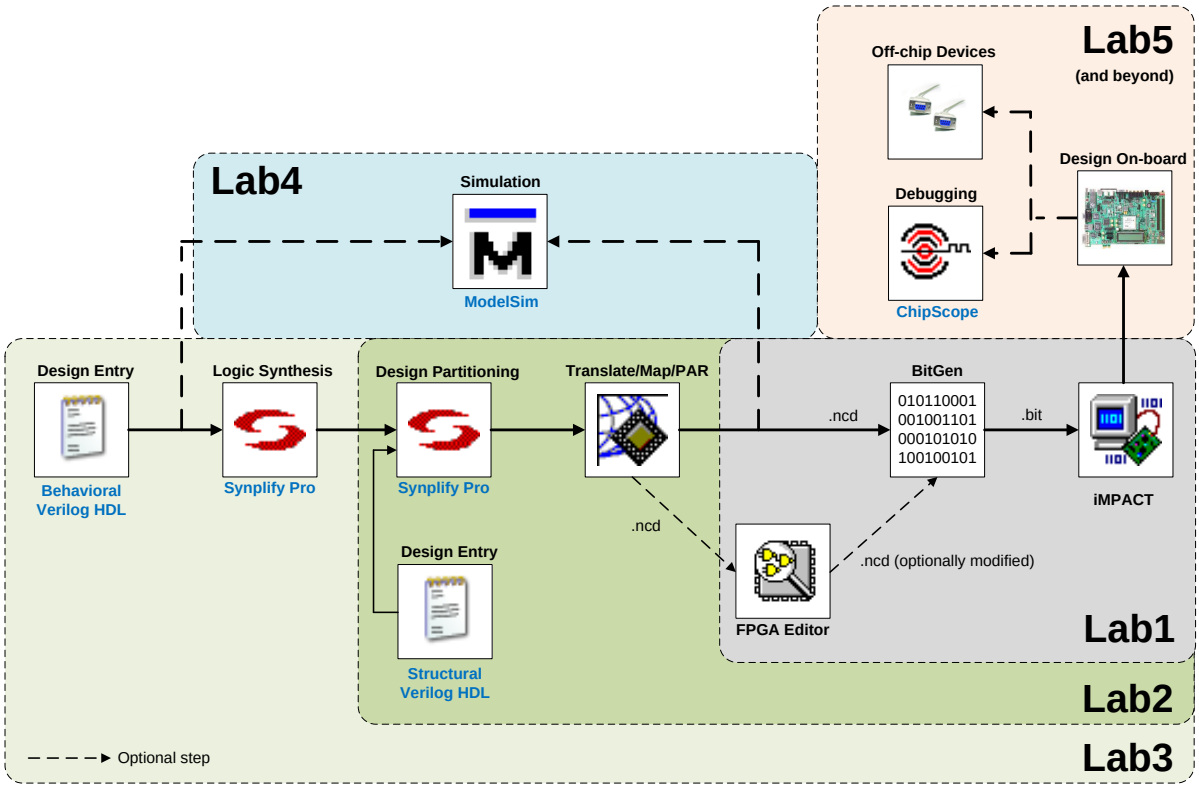
Figure 1 shows the CAD tool flow from lab 1 to lab 5. In lab 1, you were introduced to the bare-bones FPGA architecture, through FPGA Editor, and were allowed to look at and modify already placed-and-routed (PARed) designs. In lab 2, Structural Verilog, design partitioning and PAR were formally introduced. At the end of lab 2, you could create simple gate-level circuits in a textual HDL and have those circuits be implemented in actual hardware on the FPGA. In lab 3, the idea of the HDL was taken a step farther with Behavioral Verilog and logic synthesis. Using logic synthesis and more abstract Verilog constructs, you were able to map a simple Finite State Machine (FSM) to hardware in relatively few lines of code (compared to implementing it in Structural Verilog, namely gates). In lab 4, to complement your knowledge of Behavioral Verilog, you were given a Simulation tool that you could use to debug your designs. With these tools under your belt, you became able to build complex circuits, implement those circuits in hardware, and use simulation-level debugging to show correctness and catch errors.

[1] Specifically, how you can *use it*. Although the material, should you want to learn about how it actually works, is available in the lab as well.

3.1 On-Board Debugging

It is the importance of debugging which brings us to lab 5. Specifically, lab 5 addresses the need to perform debugging in actual hardware. When you perform a functional or even time-accurate simulation using a tool such as Modelsim, you must design a environment to test your circuits in. Typically, this takes the form of a Testbench and is written by you alongside the rest of your design. Testbenches work very well when you can encapsulate your whole design in a Verilog model that imitates the behavior of the entire design. For example, it is simple to write a Testbench for an FSM: you can just probe the FSM with different input vectors and observe the outputs. When your designs reach across different hardware components, however, writing a Testbench becomes more difficult and oftentimes impractical. Examples of such designs are those that communicate with third-party integrated circuits. As you cannot always design a behavioral model of a third-party circuit in Verilog, it is difficult to encapsulate your design, which uses the third-party circuit, in a Testbench. Thus, as a final means of debugging, we use a tool called **ChipScope**, which reports actual and on-board signal values to you in a wave window similar to that which you saw in Modelsim.

Figure 1 Labs 1-5 tool flow.



3.2 ChipScope Integrated Logic Analyzer

ChipScope is an in-circuit, software based logic analyzer. By inserting an "integrated controller core" (**icon**) and an "integrated logic analyzer" (**ila**) into your design and connecting them properly, you can monitor any or all of the signals in your design. ChipScope provides you with a convenient software based interface for controlling the "integrated logic analyzer," including setting the triggering options and viewing the waveforms.

There are seven main steps to using ChipScope:

1. Create a new Xilinx COREGen project.
2. Generate an "integrated controller core" or **icon**.

3. Generate one or maybe more "integrated logic analyzers" or **ila**s.
4. Connect the **ila**s to the **icon** and make all of these modules part of your design.
5. Synthesize, and implement your design (including the **icon** and **ila**) as normal.
6. Program the FPGA board.
7. Run the ChipScope software to access and use the **ila**s (the ChipScope software requires the **icon** to gain access to the **ila**s).

You will learn how to perform each of these steps in this lab. For detailed instructions, which you will have to use in Section 6, please refer to the ChipScope Tutorial on the website.

3.3 ChipScope vs. Modelsim

As has already been discussed, ChipScope is sometimes necessary just because you cannot design a Testbench, that can be used in Modelsim, for your circuit. Assuming you can, however, when is it appropriate to use one over the other? Understanding when to use each (as each are extremely useful in different situations) is very important in maximizing time efficiency when you are debugging. We will now discuss the benefits of each in detail.

Modelsim is the tool of choice when you have just mapped a circuit to Verilog and want to find simple bugs in your code. To run a Modelsim simulation and resimulate after you find a bug takes very little time because Modelsim is software-based. Furthermore, Modelsim can easily give you full visibility of your design. In other words, Modelsim allows you to add **every** signal from **every** module to the wave window with the click of a button. This allows you do, given time, track down any bug in your circuit.

ChipScope is the tool you want to use when you have thoroughly tested your design in Modelsim and it looks like it works correctly (to the point where you think it might work in hardware). ChipScope doesn't depend on a Testbench. Thus, if your Testbench was buggy (thereby implying that Modelsim wouldn't have tested your circuit correctly), ChipScope will reveal those bugs to you and allow you to analyze waves so that you can fix them. ChipScope, however, **requires** that you push your entire design to hardware (as it is hardware-based). This takes **significantly** more time than it does to simulate a design in Modelsim. Furthermore, due to the physical constraints of the on-FPGA memory that ChipScope uses, it will seldom be able to show you all signals in your design. When using ChipScope, you must be selective in what you want to see, **and if you want to change what you want to see, you will have to rerun all of the tools yet again.**

The cardinal rule that you should live by is that if it doesn't work in Modelsim, it **will not** work on the FPGA. As it is easier to debug in Modelsim, you should **always** debug in Modelsim until your circuit works correctly under the set of test vectors that you provided it. When you have a circuit that works in simulation, you can make the push to hardware and see if your circuit works on the FPGA. If it does not, your Testbench probably made incorrect assumptions about how your design would interact with the circuit you were simulating. In this case, you may want to consider using ChipScope to find out exactly where the problem actually was.

4 UART/CPU Adaptor Over Serial

Your primary objective in this lab will be to implement a UART/CPU Adaptor that will allow your MIPS150 CPU to communicate with a UART. Communicating with the UART will allow your CPU to interact with a desktop or laptop computer. When your CPU has a connection to an actual computer, you will be able to do useful and interesting things such as see the state of different registers and send commands to your CPU from a text console.

As with every design you will see in CS150, building the UART/CPU Adaptor will come in three very important steps: design, implementation and testing. The first step is understanding exactly what handshake the UART/CPU Adaptor will use to communicate between the UART and the CPU, and what

interface it will expose. The second step, actually building the UART/CPU Adaptor, will be figuring out exactly how to map the handshake down to hardware that can be implemented on the platform you are using, the FPGA. Finally, the third step will be to figure out how to adequately test the interface, given that UART cannot easily be simulated in a Modelsim testbench and that you don't have a real CPU to work with.

4.1 Design

From the CPU's perspective, the UART/CPU Adaptor will look like regular memory. This is because the UART/CPU Adaptor is **memory mapped I/O**[2] that recognizes when it is being used if the CPU asserts certain address values on its address line out to data memory. It is the UART/CPU Adaptor's responsibility to decode the address lines and recognize when the CPU wants to talk to it. This enables the CPU to talk to an arbitrary number of memory mapped devices through a single address line.

The UART/CPU Adaptor is made up of four registers: two for **data** and two for **control**. Each register has a name and an address, as shown in Table 1.

Table 1 UART/CPU Adaptor memory map.

Register Name	Address
ControlInReg	0 xffff_0000
DataInReg	0 xffff_0004
ControlOutReg	0 xffff_0008
DataOutReg	0 xffff_000c

So, for example, any time the CPU asserts 0 xffff_000c on the address bus out to memory, it should be accessing DataOutReg, one of the UART/CPU Adaptor registers. The exact purpose and structure of each register will be discussed in Section 4.1.1.

In stepping back and looking at the whole design (see Figure 2), the UART/CPU Adaptor will sit between the UART and the CPU and through a set of registers (Data/Control) in both directions, pass messages back and forth between the UART and the CPU. In order to successfully pass a message from the UART to the CPU, and back, the UART/CPU Adaptor must be sure that the side that is sending data has actually presented some data, and that the side that is to receive the data is indeed ready to receive data. If the sender doesn't have a message to send, the receiver better not try to take a message, as it will be garbage! If the receiver isn't ready for incoming data and it is presented anyway, it will be lost. To coordinate message passing between the UART and CPU, the UART/CPU Adaptor must implement a **handshaking** scheme that guarantees no data loss.

Recognize that in Figure 2, the UART/CPU Adaptor only talks directly to the UART and the CPU. This means that to pass its messages correctly, it need only interface with those two blocks. Thus, a large part of designing the UART/CPU Adaptor will be to determine its interface[3] to the two blocks. It is ultimately up to you to determine the UART/CPU Adaptor's interface. Know, however, that the interfaces used by the CPU and UART will be static. Thus, you must design an interface that is compatible with both of them. Specifically, the UART interface is shown in Table 2 and the CPU's interface is shown in Table 3.

Although that it is important that you understand all of the signals on the UART interface (Table 2), some are irrelevant relative to the UART/CPU Adaptor because they connect to the output pins on the FPGA. These ancillary lines are SIn and SOut. All of the other ports are relavent to the UART/CPU Adaptor. **Pay close attention to the ... Ready and the ... Valid ports (there is a pair for DataIn and a pair for DataOut).** These are handshaking signals that coordinate when

[2]In a memory mapped system, different devices use the same address line and only respond to certain address ranges. For example, if the address is between 0x0040_0000−0x0000_0000, one device might realize that it needs to do some work. If on the other hand the address is between 0 xffff_0000 −0x7fff_0000, another device might do some work. The important point is that from the perspective of the CPU, all of the devices are just one large memory (accessible through one set of data and address lines). In other words, the devices can hide themselves from the CPU by only doing their jobs when the address is within their specified range.

[3]The UART/CPU Adaptor's interface is its input/output port specification: namely what it exposes to modules that need to talk to it.

Figure 2 UART/CPU Adaptor big picture.

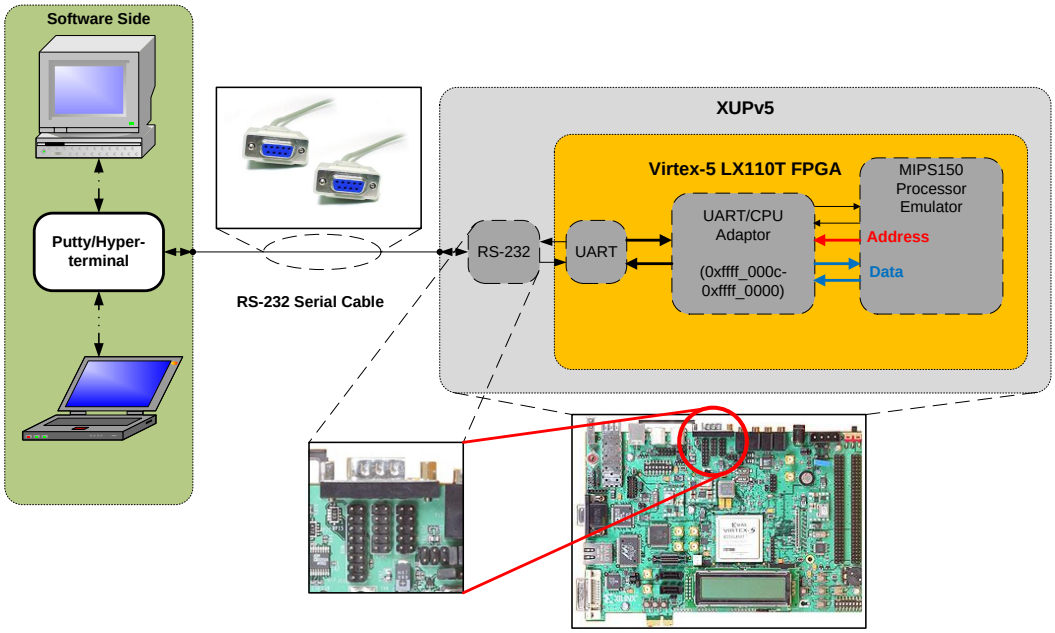


Table 2 Port Specification for UART.v

Signal	Width	Dir	Description
Clock	1	I	The clock.
Reset	1	I	Resets the UART.
DataIn	8	I	A byte of data, from another module, that is to be sent through the UART and out over the serial line.
DataInValid	1	I	Indicates that DataIn is valid.
DataInReady	1	O	Indicates that the UART is ready to receive data on the DataIn port.
DataOut	8	O	A byte of data that was received by the UART over the serial line and has been reassembled into a word of data.
DataOutValid	1	O	Indicates that the data on DataOut is valid.
DataOutReady	1	I	Indicates to the UART that the module that is going to take the data on DataOut is ready for the data.
SIn	1	I	The serial line that connects the RS-232 pins that receive the serial bitstream to the UART.
SOut	1	O	The serial line that connects the UART's serial bitstream output to the RS-232 pins on the FPGA.

the UART expects to receive data on the DataIn line and when the UART will give up data on the DataOut line. **For more information on how the "Ready/Valid" (also known as the "FIFO Interface") handshake is performed, please refer to the Ready/Valid tutorial posted on the "Documents" page of the website.**

Table 3 Port Specification for the MIPS150 CPU.

Signal	Width	Dir	Description
Clock	1	I	The clock.
Reset	1	I	Resets the CPU.
DataIn	32	I	Data inbound from a memory mapped device to the CPU.
DataOut	32	O	Data outbound from the CPU.
Address	32	O	The address associated with DataIn on a MemRead and DataOut on a MemWrite.
MemRead	1	O	Indicates that the CPU is performing a read.
MemWrite	1	O	Indicates that the CPU is performing a write.

All parts of the CPU interface (Table 2) are relevant to the UART/CPU Adaptor. As an aside, bear in mind that this is the *exact same* interface that your class project CPU will have. In other words, **the UART/CPU Adaptor that you design in this lab will be used in your project for the rest of the semester.** Ensuring that your UART/CPU Adaptor is designed well now will save you hours down the road.

As mentioned before, beyond the UART and CPU interfaces, which you cannot change, you are free to design the UART/CPU Adaptor however you please. The only requirement is that it can implement the data lossless handshake between the CPU and the UART. As you have not yet designed the CPU and do not know how it will eventually talk to the UART/CPU Adaptor, we will now discuss the handshake that you have to implement in detail.

4.1.1 UART/CPU Adaptor Handshake

The handshake between the CPU and the UART will be explained in terms of which registers in the UART/CPU Adaptor change during each part of the handshake. The four registers that make up the UART/CPU Adaptor were introduced in Section 4.1. We will now cover them in detail.

The UART/CPU Adaptor can be analyzed in terms of what the control registers are supposed to do and what the data registers are supposed to do. Specifically:

Control Registers are each one (1) bit wide. Each control register's bit is known as the **Ready bit**, which tells the **CPU** the state[4] of the data in each data register.

Data Registers are each eight (8) bits wide. Each contains a byte of data which is to be transported from the UART to the CPU, or visa versa.

From Table 1, you will see that there are In and Out registers for each control and data. The In registers (ControlInReg and DataInReg) are paired together and transport data **from the UART to the CPU.** The Out registers, conversely, send data **from the CPU to the UART.** To elaborate on the semantics that will be used for the control registers:

1. If the Ready bit in ControlInReg is asserted, then the UART has placed a byte of data in DataInReg. In this case, it is appropriate for the CPU to read the data in DataInReg.

2. If the Ready bit in ControlOutReg is asserted, the UART has read a byte of data that was placed in DataOutReg, by the CPU, some time in the past. In other words, the CPU can (when the Ready bit in ControlOutReg is asserted) safely write another byte of data to DataOutReg.

[4] If it has been read yet (on the UART→CPU side) or if it has been received by the UART (CPU→UART side).

Thus, as the name suggests, the Ready bit being asserted means that the **CPU** can interact safely with the data register on whichever side corresponds to that register (In or Out). This is an important subtlety that must be further explained: the Ready bit doesn't mean that the **UART** should consider the UART/CPU Adaptor to be ready. The semantics of "ready" only apply from the processor's perspective.

Given what has been said thus far, we have a way of controlling data transfers from the UART to the CPU and back. To make this work correctly, however, the recipient, in any data transaction, must have a way of acknowledging that it has collected the data when it indeed does so. **This acknowledgement is made through:**

1. The CPU implicitely **clearing**[5] the Ready bit on a read (UART→CPU side). **Thus, when the CPU performs a read from a DataInReg, the UART/CPU Adaptor must clear the Ready bit associated in ControlInReg automatically.**

2. The **UART setting**[6] the Ready bit on a read (CPU→UART side).

Notice how the implicit modifications are opposite (one side "sets" the Ready bit implicitely and the other side "clears" it) for each side of the UART/CPU Adaptor.

The implicit or "behind the scenes" writes to the control registers are an important detail and should not be confused with the alternative scheme: having the CPU or the UART explicitely write to one of the control registers. For example, the CPU (given the interface in Table 3) can only send Read or Write commands (through MemRead and MemWrite). Given the "behind the scenes" scheme, the CPU should **never** send a Write command to address 0 xffff_0000 or 0 xffff_0008 as these addresses correspond to the control registers in the UART/CPU Adaptor (see Table 1). Instead, when the CPU reads from address 0 xffff_0004 , the UART/CPU Adaptor should automatically **clear** ControlInReg (which lives at address 0 xffff_0000). Likewise, whenever the UART reads from address 0 xffff_000c , the UART/CPU Adaptor should automatically **set** ControlOutReg (address 0 xffff_0008). Automatically setting and clearing the control registers behind the scenes is an important task that **your** implementation of the UART/CPU Adaptor **must** perform correctly.

The other side of this story is that the CPU and the UART must be able to read (or otherwise have access to) the values of each of the two Ready bits in the control registers. For example, the CPU cannot be allowed to read data from DataInReg unless the data in that register is ready. Thus, as the CPU interface doesn't (**and cannot be made to**) have a port which is tied to the Ready bit in ControlInReg, it must **explicitely** perform a read (using MemRead) to see what value is currently held in ControlInReg. This brings up an important issue: the Ready bit is a single bit but the data line which is connected to DataIn on the CPU is 32 bits wide. **Your UART/CPU Adaptor must account for the different bit widths between the control/data registers and the actual width of the data lines that will carry those values to and from the CPU.** More information on how to pass data from the UART/CPU Adaptor to the CPU (given the varying signal widths) is provided in the MIPS Echo.s code that has been distributed with this lab (more information will be given on this in Section 4.3.1). All that will be said here is that it will be **your responsibility to implement the code in Echo.s and thus it will be your responsibility to understand how that code is treating data in lw and sw instructions**.[7]

As you will find, telling the UART what the Ready bits in the control registers are set/cleared is easier than doing the same for the CPU. This is because of the Ready/Valid interface supported by the UART (the UART doesn't have a single "address" and "data" interface like the CPU). Becoming familiar with the Ready/Valid interface as explained in the tutorial on the "Documents" page of the website will help you to see why this is.

4.2 Implementation

You will be implementing the UART/CPU Adaptor in Verilog, using any and all syntax and/or techniques that you have been introduced to so far. As was shown in Figure 2, there are more parts to this picture than just coding the UART/CPU Adaptor, however. To honestly verify your circuit, you will need an

[5] Clearing a bit means deasserting the bit.
[6] Setting a bit means asserting the bit.
[7] lw and sw instructions are the CPU instructions which need to use the MemRead and MemWrite signals, respectively.

FPGA_TOP_ML505, UART, MIPS150 CPU and computer capable of communicating through a serial cable to the UART on the FPGA. Covering each of these:

FPGA_TOP_ML505 As usual we have provided an FPGA_TOP file for you to use. We mention it here because we have only given you a bare skeleton FPGA_TOP_ML505. Specifically, it is up to you how best to use it. **Since this lab intersects the project, you should consider using this FPGA_TOP_ML505 for the rest of the semester.** Treat it well and keep it organized!

UART We have provided you a complete UART implementation (included in /Framework) called UART.v. You need not add anything to this implementation and (believe us) it works correctly. Furthermore, this module has already been partially instantiated for you in FPGA_TOP_ML505. Note that we have already set the parameters to this module for you but have not tied all of the ports to other modules in the design. It is your responsibility to learn how to use the UART.v interface posed in Table 2. It is **not** your responsibility to learn how to set the parameters to UART.v.[8]

MIPS150 CPU We **have not** provided you a black-boxed MIPS150 CPU to use in this lab (as we are sure you have been wondering through reading Section 4.1). Read Section 4.3 for details.

Computer with Serial Communications We will be using **PUTTY** (a light-weight terminal program already installed on the lab computers) to communicate through a serial cable to the FPGA. Specifically, the XUPv5 (and ML505) boards already have a serial cable port, as do the lab computers. You will connect the two together using a serial cable. You will then be able to open up a PUTTY console and send characters over the serial cable from computer to FPGA. More instructions on how to setup PUTTY to do this are available in Section 6.

4.3 Testing

As discussed in Section 3, you have come full circle with the CAD flow at this point. To help you debug your UART/CPU Adaptor, you are expected to use Modelsim, the XST RTL Schematic and the XST Technology Schematic. In short, you are expected to use any and all tools that you have been taught how to use to complete the task at hand. This will be the attitude taken with regards to the tools for the rest of the semester. The labs were meant to aquant you with the tools in preparation for the project. It is now time to put what you have learned to the test.

The above said, there are certain elements of this lab which cannot be simulated very well. Specifically, the UART and serial communication cannot be easily tested in Modelsim. To help you debug these parts of your design, you will have to employ **on-board debugging tools**, specifically **ChipScope** (as was mentioned in Section 3.2). As you will find with the rest of the project, when your circuit works inside of Modelsim but not on the board, the only tool that you will have is ChipScope, and ChipScope will serve as your last line of defense. Thus, it is very important to become comfortable with ChipScope in this lab (before the project officially begins). **Part of check-off will be demonstrating that you can use ChipScope to debug your design.**

Even with on-board debugging, however, it will be somewhat difficult to finally verify your UART/CPU Adaptor. This is because it is not clear when a handshake protocol is working correctly unless it is doing something interesting that you can identify immediately with your own two eyes. For this, you need an actual MIPS150 CPU. As hinted at in Section 4.2, you will not be given a black-boxed CPU. Instead, you will build a **CPU Emulator** which will behave just like an actual MIPS150 CPU, but only perform a single function rather than execute arbitrary MIPS assembly code.

4.3.1 The CPU Emulator

The MIPS150 CPU emulator will imitate an **Echo** program. Specifically, when you load your design, open PUTTY and type a character at the console, you should see that character appear on the console immediately afterwards. You might be tempted to think that the console is just printing the character. In fact, however, the character is being sent over the serial cable, passed through the UART and your

[8] If you are interested, however, their functions are thoroughly documented inside of the comments header for UART.v.

UART/CPU Adaptor, looped back through your CPU emulator, passed through the other side of the UART/CPU Adaptor and UART on the other side, and finally displayed on the console again.

In order to properly imitate your soon-to-be-born MIPS150 implementation, your CPU emulator **must honestly behave like a MIPS CPU would in passing and controlling data.** To help specify the exact behavior of your CPU emulator, refer to the file Echo.s included in the /Framework directory of this lab. **Your CPU emulator must functionally imitate this program's behavior.** A CPU emulator implementation that functionally does what this code does will be bounded by the CPU interface introduced in Table 3. A CPU emulator that performs based on this code need not have an actual pipeline, or look anything near as complicated as a CPU, however. Think about the decision-making circuits that you have learned about so far which will help you implement the CPU emulator in the **simplest way possible**. It is your job, again, to use what you have learned this semester to accomplish a task with a certain set of constraints.

4.3.2 Testing UART/CPU Adaptor Bandwidth

Although Echo is an eye-catching test to show that the UART/CPU Adaptor works correctly, it is imperfect as a means of verifying the handshake outlined in Section 4.1.1. Firstly, if your CPU emulator does not faithfully imitate a MIPS CPU (i.e. doesn't follow the handshake proposed in Echo.s), your UART/CPU Adaptor will not necessarily work with your actual MIPS150 CPU later in the semester. Secondly, the Echo.s program itself only tests one bandwidth scenario that your final project might have to deal with. More specifically, if you type a character at the PUTTY console, by the time you have time to type another character, the first will have come back through the serial cable and appeared at the screen. So, your UART/CPU Adaptor accepts one character and then receives one character on either side. In this case, the UART/CPU Adaptor is not tested for **back pressure**. Back pressure is the UART/CPU Adaptor's way of saying to the CPU "stop! Do not give me more data!" This equates to the CPU writing a byte of data into DataOutReg and then writing another byte into DataOutReg before the UART can pick up the first byte. In other words, the CPU performs a second write even though the ControlOutReg Ready bit is still low.

This is buggy behavior and is the either the CPU or the UART/CPU Adaptor's fault. If the UART/CPU Adaptor doesn't set the Ready bit when the CPU performs a Write, the CPU will think that it can do another Write immediately afterward.[9] If the CPU or CPU doesn't poll the ControlOutReg Ready bit properly, it has the potential to overwrite a previous value stored (but not received by the UART) in DataOutReg. Unfortunately, even if your implementation exhibits this bug, it might not be detected by Echo because of the relative speed that you type next to the speed of looping a character back to the console.

To properly detect bugs in your **back pressure** scheme, we will stipulate that your CPU emulator perform an additional task in addition to merely echoing the console. Specifically, your CPU is to detect when the character sequence:

c s 1 5 0

is typed (in that order) to the console. When it detects this sequence, the emulator is to send a single message:

Dusk till Dawn\n

back to the console without any additional input from you. This tests the back pressure case because for a single character of additional input by you, the CPU will send multiple characters back through the UART/CPU Adaptor to the PUTTY console. If back pressure is not taken into account, your implementation will overwrite back-to-back characters in the above sequence, and you won't see the full message at the console.[10]

[9] If you don't understand why this is the case, read Echo.s and try to see that the CPU is **polling** the ControlOutReg to see when it can perform a Write.

[10] This is because relative to the clock rate that your emulator will be running at, the UART can only transmit a very small amount of data back to your computer. In other words, in between when the UART picks up one byte, sends that one byte and picks up another byte, your CPU **will have to wait** before sending multiple additional bytes. If it doesn't you **will** incur data loss.

5 PreLab

Please make sure to complete the prelab before you attend your lab section. This week's lab will be very long and frustrating if you do not do the prelab ahead of time. Note that regardless of the time you spend on the PreLab, this lab may very well take longer than 3 hours. If this is the case, don't be discouraged. By spending ample time ensuring that this lab works as intended, you will be making good progress on implementing your semester project.

1. Read Sections 2, 3 and 4 thoroughly and ask questions ahead of time if anything is confusing.
2. Come up with a design which you will be able to transform into an implementation for the system described in Section 4.
   (a) **DO NOT write any Verilog when you are working on your design.**
   (b) Express your ideas as block diagrams[11], timing diagrams and textual descriptions.
3. Think about the different parts of the lab and prepare yourself for an in-lab design review with your TA.
4. **Again: DO NOT write Verilog ahead of time for this lab (contrary to the policy that has stood fast so far in the semester).** If you write your Verilog and your design is incorrect, or leaves out a critical detail, you will have wasted time writing the Verilog. One hour spent on a solid design will save you 20 down the road in implementation time.

6 Lab Procedure

The bulk of this lab is to implement the circuit as described in Section 4. Before you can test your circuit, however, you first need to configure your desktop to send data over the serial cable in the way that the xc5vlx110tFPGA expects. To do this, refer to Section 6.2 below. Additionally, there is a detailed tutorial on how to setup and use ChipScope here: ChipScope Tutorial. This tutorial can also be found on the class website, specifically on the "Documents" page of the website. You will need to refer to this tutorial to complete the ChipScope verification requirement which is a part of lab check-off. What you have to do with ChipScope to pass this requirement is discussed in detail in Section 6.1.

6.1 ChipScope Verification

This section assumes that you are familiar with how to use ChipScope, namely with the material covered in the ChipScope Tutorial.

Although you are encouraged to use ChipScope to debug whichever part of your design you please, you will be required to trigger on a specific signal and display another set of specific signals for check-off. Specifically, it is very useful to see that the Ready/Valid transactions between the UART and the UART/CPU Adaptor are behaving correctly. One way to show this is to **trigger** on when DataOutValid and DataOutReady (see Table 2) are both high. Given the Ready/Valid interface's specifications, data should be transferred from DataOut on the UART's side to the UART/CPU Adaptor when this condition is satisfied. Furthermore, DataOutReady should (for at least a brief time) go low to indicate that the UART/CPU Adaptor has received the data from DataOut and is waiting for the CPU to pick it up.

To test this condition, we will **trigger** on the condition when:

$$\{\text{DataOutValid, DataOutReady}\} = 2\text{'b11}$$

Furthermore, we will assign to the ChipScope data bus[12] the signals DataOutValid, DataOutReady and DataOut. Thus, when both DataOutValid and DataOutReady are high, our wave window on the

[11] Block diagrams, for this lab, may be detailed down to the level of gates. If you draw a block diagram at the gate-level, be sure to define the interfaces you have established between modules very clearly. Furthermore, make sure everything is **especially** neat. It is easy for a gate-level to become incomprehensible.

[12] You may place the signals that you are to display, in the wave window, on either the trigger or data bus (see the ChipScope Tutorial for details). The only requirement is that the signals do indeed appear in the wave window.

Desktop/Laptop will display the signals DataOutValid, DataOutReady and DataOut for some period of time.

If you implement this correctly, you should be able to type a character at the PUTTY console and see (almost immediately afterwards) this triggering condition be satisfied. Think about why this is the case. When you have it working, and finish the rest of the lab, show how to trigger the condition and the waves, themselves, to your TA for check-off (Question 2).

6.2 Setting up Serial Communications

Setting up serial communications is done in three parts. First, we will configure the desktop/laptop to talk across the serial cable at the correct rate and with correctly formatted messages (Section 6.2.1). Next, we will configure PUTTY to use the console to transmit data over this newly established serial connection (Section 6.2.2). Finally, we will test that all went well with a simple loopback trick (Section 6.2.3).

6.2.1 Setting up Desktop/Laptop Serial Communication

Before we can talk over serial, we must configure our computer to talk correctly. Note that the below instructions may differ slightly based on what version of Windows you are using. Regardless of which version, however, you are looking for the **Device Manager** and the "Serial Cable" entries under it.

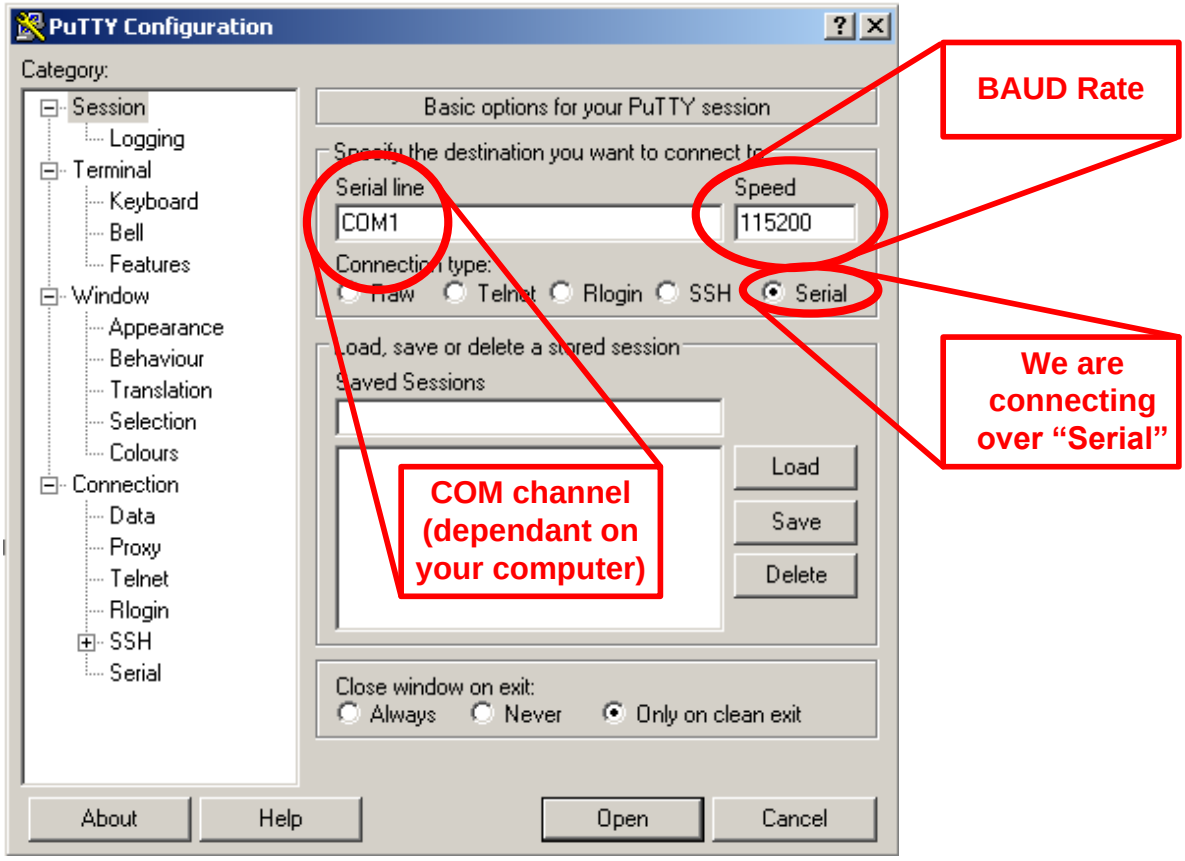
1. Make sure that your serial cable is hooked up to the **serial port** of your computer or through a USB port if you have a serial to USB adaptor.
2. Open **Start** $\longrightarrow$ **Control Panel** $\longrightarrow$ **Performance and Maintenance** $\longrightarrow$ **System**.
3. On the **Hardware** tab, select **Device Manager**.
4. Under the list of hardware devices, find the heading labelled "**Ports (COM & LPT)**" (or similar).
5. Double-click on the entry you see under this heading (it should be named "Serial Cable" or similar). Take a mental note of which **COM** the device is listed as using. You will need this later.
6. In the **Port Settings** tab, set the following values to the following items:
   (a) Bits per second: **115200**
   (b) Data bits: **8**
   (c) Parity: **None**
   (d) Stop bits: **1**
   (e) Flow control: **None**
7. Click **Ok** and exit out of the Device Manager when you are finished.

6.2.2 Setting up PUTTY

Once you have configured your Serial Connection, it is time to setup the PUTTY terminal so that you can talk over the Serial cable.

1. In the /Framework directory of the lab, extract putty.exe and place it somewhere on your computer.
2. **Double-click** on putty.exe to open the GUI. You should see an interface that looks like what is shown in Figure 3.
3. Configure PUTTY.
   (a) Set the **Serial line** text field to COMX where X is the channel that you saw was acquired when you hooked up your serial cable in Section 6.2.1. (For example, X= 1 in Figure 3).
   (b) Set the **Speed** to **115200**.
   (c) Under the **Connection type** set of radio boxes, select **Serial**.
   (d) **Click Open** at the bottem of the GUI to open the serial console.

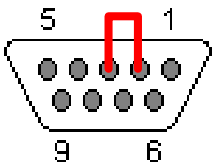
Figure 3 PUTTY user interface.



6.2.3 Verifying your Serial Connection

Once you have setup your Serial Connection (see Section 6.2.1) and configured PUTTY (Section 6.2.2), you can verify that everything is hooked up correctly by using the **serial loopback trick**[13]. The serial loopback trick is where you tie pins 3 and 4 (TXD and DTR) of a **female-side** serial cable together using either a **paperclip** or a **wire** (as shown in Figure 4).

Figure 4 Serial loopback trick.



Once you have connected the pins together, you can type at the PUTTY console and expect to see characters appear again instantly. **Thus, the loopback trick performs the equivelent function of the Echo.s program, without the bandwidth functionality added.** Obviously, it won't be enough for check-off, but it is a great sanity check for getting the serial connection setup correctly.

[13] Courtesy of Alex Williams.

7 Lab 5 Checkoff

ASSIGNED	Friday, September 25^{th}
DUE	October $12^{th} - 13^{th}$, during your assigned lab section

Man Hours Spent	**Total Points**	**TA's Initial**	**Date**	**Time**
	/100		/ /	
Name	**SID**	**Section**		

1. PreLab ..... _____ (25%)
   (a) Came prepared with a design.
   (b) Showed knowledge of the system described in Section 4.
2. ChipScope ..... _____ (25%)
   (a) Demonstrated that ChipScope can properly trigger on valid data being presented to the UART/CPU Adaptor.
   (b) Displayed the byte of data (on the input of UART/CPU Adaptor) in the wave window, using either a trigger or data line).
3. UART/CPU Adaptor ..... _____ (50%)
   (a) Demonstrates a working Echo to the console.
   (b) Demonstrates the CS150 → Dusk till Dawn\n special case.

Rev.	**Name**	**Date**	**Description**
D	Austin Doupnik	9/25/2010	Updates for Fall 2010.
C	Ilia Lebedev	2/22/2010	Updates for Spring 2010.
B	Chris Fletcher	2/24/2009	Fixed the explanation of the "Ready bit" in Section 4.1.1.
A	Chris Fletcher & John Wawrzynek	2/12/2009	Wrote new lab. Designed to replace old "Logic Analyzers" lab) written by Greg Gibeling and Chris Fletcher.