Project 4 **USB Host: 341's "3B"** **100 pts**

Now we'll explore designing parts of a larger — and more real — interface. The USB interface will be the basis for the project, however only part of the standard will be implemented. What's new here as compared to other projects are the multiple FSM-Ds that it takes to implement the different layers of the protocol in hardware. You need to identify these layers and synchronize their activity.

We will assign a lab partner to you, the goal being to share in the work. If your group is late, each group member will be docked the same number of day-late passes. If, for instance, your group is two days late and one person has two passes available and the other only has one, the person with one late pass will lose points.

Alternately (and rarely), you may also work with a partner but where each develops their own full solution. In this case you may talk with that person extensively about developing a solution, even helping with debugging. But your code should completely be your own. Tell us immediately — and your partner! — if you're going to work this way.

Use Piazza to ask questions or come visit our office hours. If neither of these works, send email to arrange an appointment. We want to see you succeed, but you have to ask for help. There's a lot of work here, pace yourself and keep up.

Day-late passes only apply to the project due date, not the prelab.

Important dates

Available on BlackBoard: 10/14/15. **Prelab due:** Anytime before 10/27/2015 at class time. **Lab Due:** 11/3/15 at class time. **Drop dead date:** Tuesday 11/5/15 at class time. Special situation: if you took this course before and dropped, you are not allowed to use your previous code.

A special warning: the prelab due date is set to be late – you should really strive to do this as early as possible, maybe by Oct 22 or 23 or earlier! It's set late for flexibility with your other assignments.

The Project

Using SystemVerilog, design and simulate part of the Serial Interface Engine (SIE) of a USB 2.0 Host. We will not implement the whole standard — you can start breathing again — and not everything will be exactly as the spec does it. So, don't go reading the spec for answers.

Your host will be connecting to one device, a thumb drive model provided by us — no USB hub. Your host will be reading/writing blocks of data from/to the thumb drive. The testbench that interacts with your host can have implicit FSMs to mimic an operating system that is making read/write requests. Your host will then carry out the requests by controlling the USB to send and receive packets of type OUT, IN, DATA0, and ACK/NAK.

Your usbHost will have several coordinated FSM-Ds to handle all of the activity from the bit-level stuff (NRZI, bit stuffing, CRC, …) through to the higher protocol-level stuff (handling the data, ACK/NAK, errors, retries, …). Call these the *bit-level* and *protocol-level* FSMs. Each of these levels should be implemented with coordinated explicit-style FSM-Ds. You'll need to specify the interface protocols between these. These will all be part of your SIE.

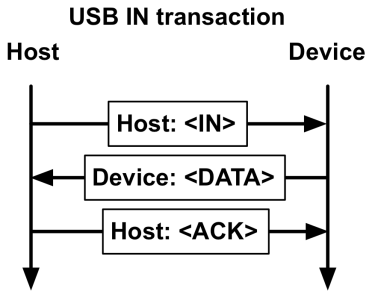
We mentioned errors and retries. Oh my! How could there be errors? Remember, we're providing the thumb drive device and we'll work in a few errors too — maybe a few more than we think.

Packet types to implement

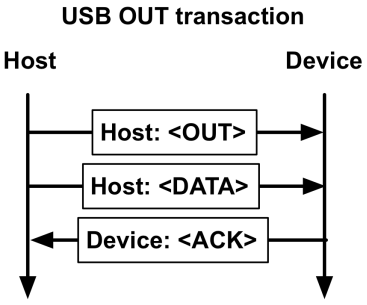
We won't implement all USB packet types, only those in this table.

<table>
<tr><th>USB Packet type</th><th>Name, PID [3:0] See lecture notes</th><th>Description</th><th>Packet Format
Each of these formats has a SYNC before it and an EOP after it.</th></tr>
<tr><td rowspan="2">Token</td><td>OUT
0001</td><td>Address and endpoint numbers specify the device that will receive the output transaction (from host to device). CRC calculated on ADDR and ENDP.</td><td>Token Format
<table><tr><th>Field</th><th>PID</th><th>Addr</th><th>ENDP</th><th>CRC5</th></tr><tr><td>Bits (lsb<->msb)</td><td>8</td><td>7</td><td>4</td><td>5</td></tr></table></td></tr>
<tr><td>IN
1001</td><td>Address and endpoint numbers specify the device that will send the input transaction (from device to host). CRC calculated on ADDR and ENDP.</td><td>Token Format
<table><tr><th>Field</th><th>PID</th><th>Addr</th><th>ENDP</th><th>CRC5</th></tr><tr><td>Bits (lsb<->msb)</td><td>8</td><td>7</td><td>4</td><td>5</td></tr></table></td></tr>
<tr><td>Data</td><td>DATA0
0011</td><td>Data packet. Sent by host after an OUT, or sent by device in response to IN. We'll assume that the Data field will always be 8 bytes in this project. CRC16 calculated on DATA.</td><td>Data Packet Format
<table><tr><th>Field</th><th>PID</th><th>Data</th><th>CRC16</th></tr><tr><td>Bits (lsb<->msb)</td><td>8</td><td>64</td><td>16</td></tr></table></td></tr>
<tr><td rowspan="2">Handshake</td><td>ACK
0010</td><td>Got it</td><td>Handshake Packet
<table><tr><th>Field</th><th>PID</th></tr><tr><td>Bits (lsb,<->msb)</td><td>8</td></tr></table></td></tr>
<tr><td>NAK
1010</td><td>Didn't get it. For those of us who are often told we just don't get it.</td><td>Handshake Packet
<table><tr><th>Field</th><th>PID</th></tr><tr><td>Bits (lsb,<->msb)</td><td>8</td></tr></table></td></tr>
</table>

Packets, Transactions, and Reads and Writes



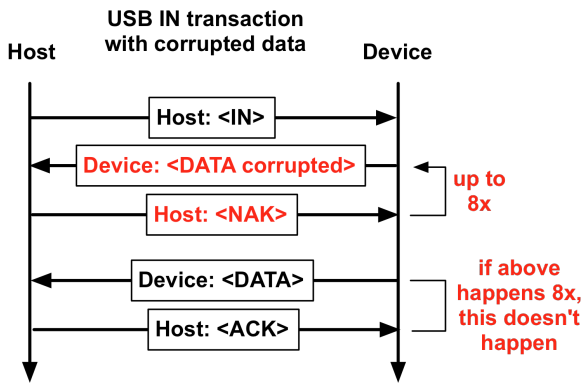
In USB-speak, there are IN *packets* as described above (officially, they're Token packets with a PID saying IN, but we'll just call them IN *packets*). There are also USB IN *transactions*. A USB IN *transaction* starts with the host sending an IN *packet* to a device, waiting for the device to send a DATA *packet*, and then the host sending an ACK *packet*. This *transaction,* built out of a sequence of *packets,* is shown in the process flow diagram on the right.

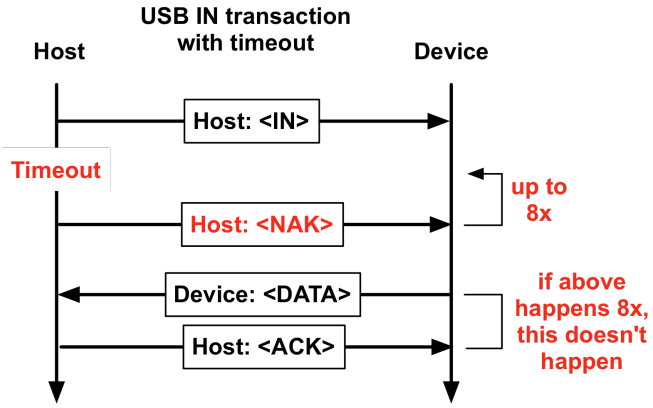


Similarly, there are OUT *packets* as described above and USB OUT *transactions*. An OUT *transaction* starts with the host sending an OUT *packet* to the device. The device then knows to expect a DATA *packet*. When the DATA *packet* is sent, the device sends an ACK *packet* to the host. This *transaction*, built out of a sequence of *packets*, is shown in the process flow diagram on the right.

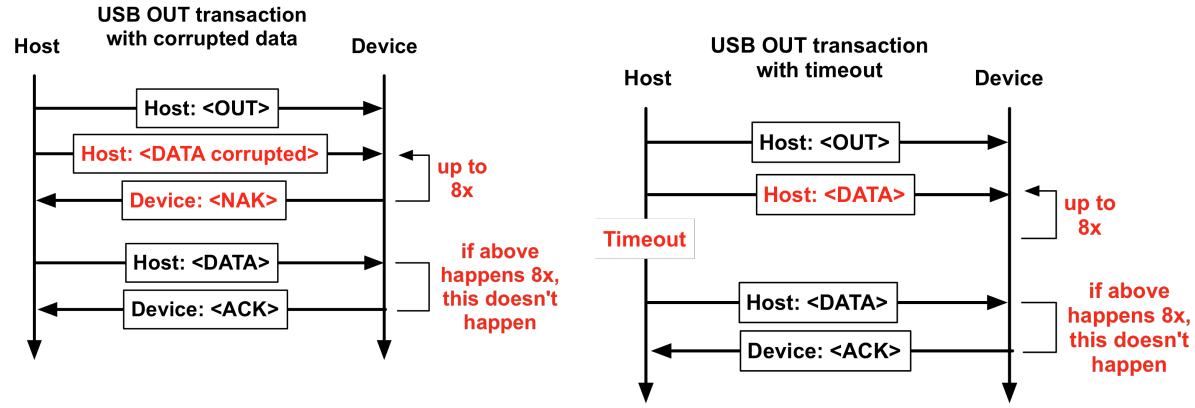
Your host will be requested by the testbench to do READs and WRITEs with the thumb drive. To do a READ transaction (a term we made up), the USB host would do an OUT transaction to send an address to the device (it would be in the DATA packet). It would then do an IN transaction to get the device to send it the data read from the thumb drive memory. In a WRITE transaction (something else we made up), the USB host would do an OUT transaction to send an address to the device (it would be in the DATA packet). It then would do another OUT transaction to send the data to be written in the thumb drive's memory. In all cases, the DATA packet will send 8 bytes. When sending the address, a DATA packet will be used and the address will be in the last two bytes sent; all others will be zero.

The above diagrams assume that everything was correct, i.e., there were no CRC errors and no timeouts due to someone missing a packet. Below are two modified process flow diagrams for IN transactions where errors occur. On the left, corrupted data is sent by the device — it may be caused by noise on the line, etc. The host responds with a NAK and the device tries again, up to 8 times. Hopefully, that data finally gets through and the host sends an ACK. However, if it tries and fails 8 times, the whole transaction is cancelled. On the right, nothing is received. After 255 clock periods, the host times out and sends a NAK. At this point, the device should send another DATA packet. If the timeout occurs 8 times, then the whole transaction is cancelled.





A similar thing happens with OUT transactions and data either being corrupted or there being a timeout. The diagrams are similar and won't be explained.



At the transaction layer, these are the only types of errors that you have to consider. Notice though that you will need a protocol FSM-D to keep track of these possible situations. This Protocol FSM-D should embody the above flow diagrams. Don't get caught NAKed.

J's, K's, D+, D-, SE0 …

See the definitions for J and K as presented in lecture.

- The wires on the USB bus are named DP and DM (for D+ and D-). They have tri0 net types to have them pulled down. Driving the bus wires should be done with tri-state drivers.
- Start of Packet (SYNC): This is 0000_0001.
- End of Packet (EOP): This is XXJ, where X means single-ended zero (SE0). In an SE0, both of the D lines are 0 (DP == 0 and DM == 0).

NRZI, Bit Stuffing and CRCs

The different types of fields (within packets) that are sent are SYNC, PID, ADDR, ENDP, DATA, CRCremain, and EOP. However, not all packets contain all of those fields. See the above table of packet types.

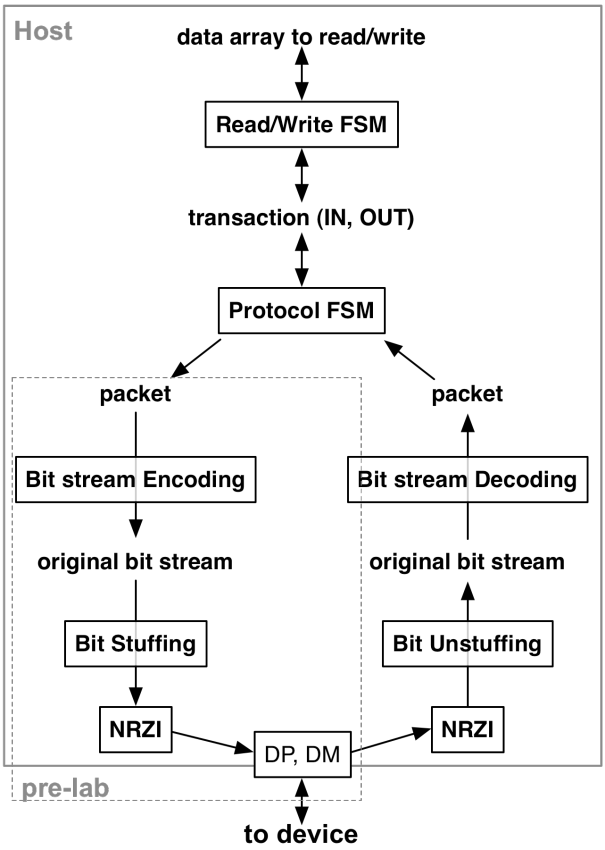
USB's **NRZI** has a different definition than Wakerly's. We'll be using USB's — output changes on an input of 0, and stays the same on input of 1. See lecture notes. The first bit of the NRZI should be output as if the previous bit sent was a 1. All field types are sent through NRZI except the EOP. Illustration: after SYNC (0000_0001) passes through NRZI, it will be KJKJ_KJKK.

Bit-stuffing: A zero is inserted after 6 consecutive 1's. Everything is stuffed except for SYNC, PID, EOP, and your Thanksgiving turkey. Bit-stuffing starts counting with the first bit after the PID and continues through the end of the CRC (this isn't actually how the standard does it, just how we do it).

CRC: Some packets don't use CRC, some use CRC16 (the DATA0), and some use CRC5. See the table above to determine which, and what fields they are calculated over. The generator polynomial and residue for each of the CRCs is shown to the right. The CRC is calculated by the "bit stream encoding" FSM-D as shown in the following diagram. That is, the protocol FSM provides the fields to send and the bit stream encoding calculates the CRCremainder as the bits are being sent serially to the bit stuffer, etc.

	Polynomial	Residue
CRC5	$X^5 + X^2 + X^0$	5'b01100
CRC16	$X^{16} + X^{15} + X^2 + X^0$	16'h800D

Finally, the Whole Enchilada



The layered organization of the FSM-Ds for your host is shown in the figure. The boxes are the FSM-Ds and the unboxed labels represent data (e.g., the array to read/write, the original bit stream, or a packet, etc.). Everything below the bit stream encoding or decoding box is serial; everything above has registers or arrays with data in them. For instance, the Protocol FSM won't do anything until it sees that the whole packet has arrived. And it will do something different depending on whether the packet was correct or not.

Think of each of these FSM-Ds as doing a piece of the work — providing a service — and handing it to the next. Thus there are some not-shown signal and data lines between each of the FSM-Ds that define their inter-FSM-D protocol. You need to define these.

Read/Write FSM: The role of the read/write FSM is to receive the task calls from the testbench (see below). The calls will either be for a read or write of the memory. The FSM creates a sequence of IN and/or OUT transactions to implement the request. These are individually passed to the protocol FSM.

Protocol FSM: The role of the protocol FSM is to turn an IN or OUT transaction given to it by the read/write FSM into a series of packets to implement the transaction. It also receives packets from the thumbdrive. Thus it handles all the ACKs, NAKs, retries and time outs. Consider the bit stream decoder again, it might have received a packet, but found that the CRC was wrong. The Protocol FSM, seeing such an indication, will then know to send a NAK.

For the serial parts of the system, the serial bits pass through the FSM-Ds (encoding, bit stuffing, NRZI) — they're each Mealy machines. (i.e., don't calculate the CRC on the whole packet, and then stuff bits as needed, and then calculate the NRZI, and then begin sending out the bits. Rather, it's one bit stream going through some Mealy machines. Besides, the other way sounds like software — don't go there!)

A way to think about the project is that your testbench will call a task (e.g., readData). The task will set up registers that will start the read/write FSM doing its thing (in this case a read of some memory location). The readwrite FSM will tell the protocol FSM to start doing a transaction (like In or Out). The protocol FSM will send a packet and make sure the packet is received (handling the error conditions, ACKs, NAKs, etc). When the data read has been returned in a DATA packet, the protocol FSM will tell the read/write FSM of success (or failure). The testbench task will return with the value of success (and the data if it was a read) and the testbench can decide what to do next.

Your FSM-Ds should be written in the explicit style. They should be synthesizable although we will not synthesize them.

You need to define the protocols — the signaling bits and data, and their relation to each other — at each of the FSM-D boundaries that you have. (This is part of the prelab.) BTW, you don't need to stick to this exact figure when organizing your implementation. But don't think you'll be able to do this as just one biggie-sized FSM-D! The layered view is quite useful in constructing (for transmitting) or deconstructing bit streams (for receiving) — the FSM-D partitioning should follow from it. Simpler coordinated FSM-Ds are the way to go.

The code sample to hand in for prelab is shown in the dashed line. Once you see that it's working with our prelab-testbench, then you can continue building the other parts of your USB.

The Device

Important numbers:

- The USB address of the thumb drive device, used in token packet ADDR fields, is 5. Don't confuse this with the address of the memory's data that is being read or written.
- To send the address to be used when reading or writing the memory, send an OUT transaction to ADDR 5, ENDP 4. The DATA0 packet that follows as part of the OUT transaction will be used to address the memory.

After the address has been sent:

- To do a read, send an IN packet to the device at ADDR 5, ENDP 8.
- To do a write, send an OUT packet to the device at ADDR 5, ENDP 8.

The DATA0 packet that follows is either the data being read or written. In all cases, don't forget ACKs and NAKs, etc.

Using the SystemVerilog Interface construct

We'll make use of the SystemVerilog Interface construct in this project, although it will only be minimal usage (See SVBook 6.5.1). We have already instantiated the interface for you in the `top.sv` file. Your host module already has it as a port (boy, are we nice guys…). The signal lines themselves are of type `tri0`, so they are pulled down to 0 when nothing is driving them. Make sure you use tristate drivers.

What you need to know…

```
interface usbWires; // this defines the interface
   tri0 DP;
   tri0 DM;
endinterface

//then, in another galaxy, far, far away…

usbWires wires(); // this instantiates usbWires with instance name "wires"

assign wires.DP = enable ? foo : 1'bz;    //a tristate driver
assign wires.DM = enable ? foobar : 1'bz;

assign yourVar1 = wires.DP;      // getting access to DP and DM
assign yourVar2 = wires.DM;
```

Your testbench:

Your testbench, pretending to be an OS, will call these tasks, which reside in your usbHost, to set up memory accesses. So that the TA testbench can initiate reads and writes, you must stick to these task headers. The status returned (success) indicates if the transaction completed successfully, as opposed to there being 8x bad packets or an 8x timeout. We'll provide several encrypted thumb drives for you corresponding to various levels of the protocol. They'll print information telling you what they received and/or what they're sending back to you. We'll provide some with typical errors.

```
   task readData           // host sends memPage to thumb drive and
                           // then gets data back from it; then returns
                           // the data and status to the caller (the tb)
```

```
    (input bit [15:0]  memPage,   //Page to read (address to read)
     output bit [63:0] data,      //the 8 bytes read, return to tb
     output bit        success);  //True if successful, False if not


task writeData             // host sends memPage to thumb drive and
                           // then sends data; then returns status to
                           // the caller
    (input bit [15:0]  memPage,   //Page to write
     input bit [63:0]  data,      //the 8 bytes to write
     output bit        success);  //True if successful, False if not
```

Of course, you will probably want to write some smaller testbenches along the way to check smaller functionality (e.g., the CRC).

Our Testbenches

Our testbenches come in different flavors.

Testbench name	How to use it	What's it all about
TA_tb_simple.svp	make simple	This test checks basic read and write functionality. It compiles top.sv, TA_tb_simple.svp, usbBusAnalyzer.svp and usbHost.sv with our good thumb drive.
TA_tb_full.svp	make full	This test checks advanced (non-faulty) read and write functionality. It compiles top.sv, TA_tb_full.svp, usbBusAnalyzer.svp and usbHost.sv with our good thumb drive.
TA_tb_faults.svp	make faulty	This will cause errors, see below. This compiles top.sv, TA_tb_faults.svp, usbBusAnalyzer.svp and usbHost.sv with our faulty thumb drive. "make faultyTowers" will stream a great TV comedy on your screen.
TA_tb_prelab.svp	make prelab	This tests the functionality needed for the prelab. It compiles top.sv, prelab_tb.svp, usbBusAnalyzer.svp and usbHost.sv with our prelab thumb drive.
tb.sv (your testbench)	make student	This test allows you to run your own testbench. It compiles top.sv, your tb.sv, usbBusAnalyzer.svp and usbHost.sv with our good thumb drive.

After you "make" one of the executables above, type "./simv" to execute it. Or "./simv +debug" will invoke our bit-stream debugger. It will watch your DP and DM connections and tell you what packet fields it sees.

The errors caused by make faulty:

- Garbled Data: This test will write data to the flashdrive and attempt to read it back. The first attempt to read from the flashdrive will be corrupted and a NAK will be expected to be sent back to the flashdrive. This is for the IN transaction.
- Timeout: This test will timeout on the data received from the thumbdrive. This will test your ability to send a NAK and recover from that situation.

- NAK test: This test will send a NAK after the thumbdrive receives a DATA packet from the usbHost, regardless of how well formed it is. This tests your ability to receive a NAK and recover.

Model Organization

We have prepared a handout folder for p4 at /afs/ece/class/ece341/handout/p4/. Copy over the files located there to your own area, and hand in only the **bolded** files:

- Makefile – use this to simulate your usbHost.sv with one of the various testbenches (listed above)
- README.txt – this authoritatively documents the usage of the Makefile
- Various encrypted testbench files — TA_tb_full.svp, TA_tb_faults.svp, usbBusAnalyzer.svp
- top.sv – the top, please don't modify this file
- **usbHost.sv** – your host
- **tb.sv** – your testbench — it calls the tasks in your usbHost.sv

For Credit

Turn in these parts to the folders listed later and at the times listed later. These parts include:

- **A pre-lab report and upload file.** The report will be a diagram of the FSM-Ds and their interconnections. What are the handshake signals that are used between them? What registered values are passed between them. Upload code that will send an OUT packet with ADDR=5 and ENDP=4. The packet should have SYNC and EOP too. It should drive the wires in the interface in top.sv. We'll receive it and check the packet with the bus analyzer.
- A final, short write-up explaining your system's organization. Alter the pre-lab diagram as necessary; explain the signaling variables between the FSM-Ds. Show the state transition diagrams for the non-testbench parts of the system. If you are working with a partner, you must specify in detail who did what.
- Your SystemVerilog modules "appropriately" written. e.g., clean writing style and correct use of SV language semantics. The only connections between your usbHost and out usbDevice are on the usbWires interface in top.sv, the common clock, and reset. These models should not use any global variables. (Common clock? That's right, we're not implementing PLLs.)

Given the above files, we will do the following:

- Run the testbenches. We will run our script file with our testbench with your design automatically. That's why it's important to conform to the bus signal, task, and module names and file organization. This will keep your personal demo shorter.
- Ask you some questions. Come early prepared to start.

If you can't get all of this to work, please note from the grading sheet at the end of this document that we'll give partial credit for seeing certain parts of this working (these parts tie into the "whole enchilada diagram" above). Making sure that your partial solution shows evidence of these parts working as this will help us give you partial credit for what you did. Of course, if it's all working you don't need to show all these intermediate points.

Don't try to write the whole thing and then test it. Write and test parts, and keep adding. This will also help with partial credit.

How to turn in your solution

We assume you know the drill.

- Prelab filenames are: p4_prelab_yourID.pdf, and prelab.sv.
- Final files are: usbHost.sv, tb.sv, and p4_yourID.pdf.

If you have a lab partner and you're turning in common code, then you only need to upload the above information to one of your afs areas. Make sure both names are in the header of the reports.

Demos and late penalty

We will have demo times outside of class times on or near the due date. If you are working with a partner, you must both attend the demo (one time slot). If you both wrote your own code, you must still demo together, hopefully in back-to-back time slots. They must be with the same TA.

Define Late: Lateness is determined by the file dates of your files.

Late penalty: 15% per day based on the file modification date wrt class starting time on the due date. You can use your day-late passes to remove the penalty, but you still must turn the project in by the drop dead date.

Drop Dead Date: The last time to turn in this project. Upload what you have by that time and we'll do a "demo" with you to see what partial credit we can give. Even if you're code doesn't completely compile, execute, or solve the problem, upload what you have. i.e., it's time to get some credit for what you've done. More importantly: Time to move on.

Not uploading anything and not showing up to explain what you've done is not attempting the project — see details in the syllabus.

Some SystemVerilog things — tasks

As described above, tasks are to be used as the interface between your testbench (and ours too) and your read/write FSM. See 11.5.2 in the SV book for a discussion on tasks. There are examples of how to write a task.

SystemVerilog has two types of subroutines: tasks and functions. The main differences are that tasks can contain time operators (#, @, and wait) while functions cannot; and functions return a value that can be used in an assignment statement (e.g., A <= B + fun(C);) — you can't use tasks this way. Functions and tasks have arguments. When a function or task is called, the arguments labeled input are copied into it. When a task returns, the output variables are copied back into the calling procedure. (Functions can do that too but most often a value is returned as you would in a software function.)

How does this apply here? Given the task header listed below, if your testbench was going to do a read, it would pass the address to be written, and call task readData. When it returns, you can check if it was successful. See code below:

```
//in your TB
bit [15:0] addr;
bit [63:0] dataRead;
bit did_it_right;

initial begin
    …
    readData (addr, dataRead, did_it_right);
    if (did_it_right)
          //the read was successful! Do some checking
```

```
    …
end
```

Thus, your testbench could be a series of reads and writes, and checking to see if you get the right info back.

What's in the task that's called? It might look something like this:

```
task readData (input bit [15:0] memPage, output bit [63:0] dataRead,
                        output bit success);
    FSMmempage <= memPage;
    startFSM <= 1;
    wait  (read_write_FSM_done);  //  this  variable  should  be  set
                        synchronously
    startFSM <= 0;
    dataRead <= valueRead;
    success <= isValueReadCorrect;
endtask
```

This task would be inside the module that is your FSM-D. The variables FSMmempage, startFSM, valueRead, and isValueReadCorrect are meant to be variables used by the read/write FSM-D (they would be defined outside the task but inside the module — other organizations can work too). You may have different variable names. BTW, startFSM is a signal to the FSM to start doing its thing; set it back to zero so it doesn't do it twice! The point is that you call this task, it sets some variables, the read/write FSM sees the variables and starts doing its thing, then the tasks waits for the read/write FSM to be done, then it returns the values to the testbench. Essentially, the task (an implicit FSM) and your read/write FSM (an explicit FSM) are handshaking on reading and writing the memory.

This is the readData task copied from above.

```
task readData           // host sends memPage to thumb drive and
                        // then gets data back from it; then returns
                        // the data and status to the caller (the tb)
(input bit [15:0]  memPage,   //Page to read (address to read)
 output bit [63:0] data,      //the 8 bytes read, return to tb
 output bit        success);  //True if successful, False if not
```

18-341 Project 4 Grading Sheet

Student Name(s):_____ Andrew ID(s):_____

TA Name/Date_____

Category	Max	Your Score	Comments
1) Prelab demo and report (demonstrate sending packet)	20%		
2) Demo Run and question(s)	5%		
3) What's working: • Simple part 1 • Simple part 2 • Full • Faulty part 1 • Faulty part 2 • Faulty part 3	 10% 10% 10% 10% 10% 10%		
5) Code style	10%		
6) Final write-up	5%		
Subtotal	100%		
Late penalty (–15% day) calculation: # actual_days_late minus # passes_used_here = {15, 30}			Data for day_late calculation: # actual days late (TA: from file dates) _____ # day-late passes available (in gradebook) _____ # day-late passes used here (from student) _____
Total	100%		