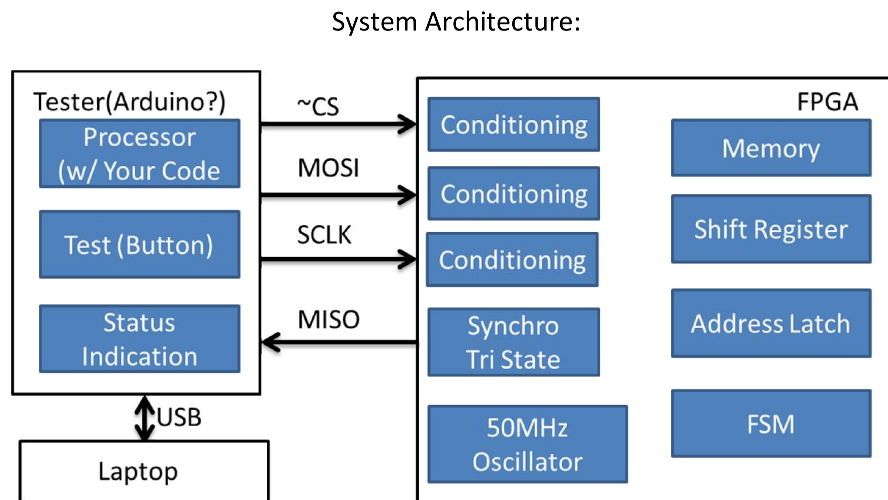
ENGR xD52: MP b010

In this lab you will create an SPI Memory and instantiate it on an FPGA. You will also create an automated harness for End of Line testing. You will then use your tester to verify your own memory, as well as the memories of the other groups in the class. Lastly, intentional failure modes will be injected and (hopefully) detected.

Due 11/3/14 11PM EST

System Architecture:



Work Plan

The first portion of this subdeliverable is due within 36 hours as an email to comparch14@gmail.com with the subject line formatted as [MP2 Work Plan] Name1 Name2 Name3". Any questions regarding this section should be directed to personal email addresses. Questions sent to the assignment submission address **will not** be read in a timely manner.

Read Joel Spolski's blog article about evidence based scheduling: http://www.joelonsoftware.com/items/2007/10/26.html

Create a work plan for this lab using the provided excel template. Break down the lab in to small portions, and for each portion predict how long it will take (in hours) and when it will be done by (date). An appropriate number of subtasks for this lab may be in the 7-12 items range. Tasks that are predicted to take longer than 4 hours should be broken into smaller tasks.

Submit a scheduling update on 10/27/14.

Provide the final scheduling analysis in your final deliverable.

A NOTE ABOUT EXTENSIONS

The initial work plan is the time to ask for an extension if you think you might need one. Acceptable reasons for an extension include: going to a foreign country or being away from Olin for many days, getting the flu, or being abducted by aliens. Unacceptable reasons include: starting the night before, poor time management, or realizing you won't finish in time half an hour before it's due. We will not be granting extensions after the first 24 hours unless you have a *very* good reason.

Input Conditioning

The Input Conditioning subcircuit provides three services:

1) Input Synchronization: The pair of D Flip Flops at the front of this unit synchronizes the external signal to the internal clock domain. The first flip flop will almost certainly have its setup and hold requirements violated – its input can occur at any phase offset with respect to the internal oscillator. The second flip flop takes the partially synchronized signal and brings it fully into phase with the internal domain.
2) Input Debouncing: Buttons and Switches are notoriously noisy, and may be unstable for several milliseconds after a transition due to mechanical oscillations. Purely electrical signal sources may also show similar (but much less severe) oscillations due to noise and signal reflections. This circuit cleans up that oscillation by waiting for it to stabilize.
3) Edge Detection: These signals are indicated for a single clock cycle on each positive and negative edge of the external signal. These are used by other subcircuits to emulate "@(posedge ___)" type behaviors without creating extra clock domains.

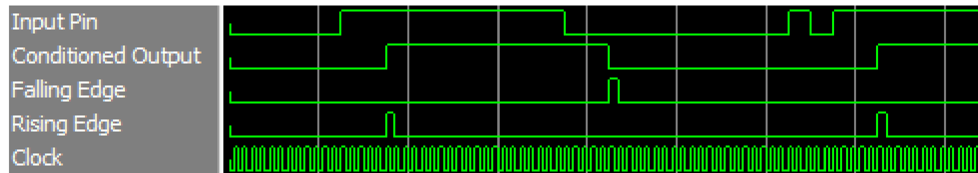
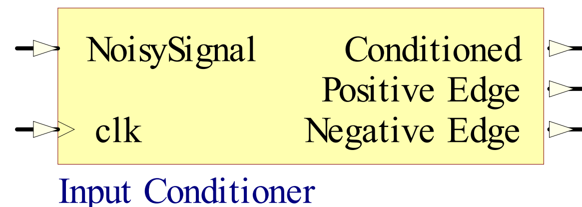


Figure 1: Glitch Suppression and Edge Detection

Start with the module described in Behavioral Verilog in the attached "InputConditioner.v":



Modify it so the positive edge and negative edge signals are correctly generated. These signals are high for exactly one clock period when clean signal has a positive or negative edge, **in the same clock period that clean signal transitions**.

Note: There are several ways to generate the edge signals. Assign statements work on wires, so the signals should not be declared as reg. Always statements work on registers, so that declaration should include that.

```
module inputconditioner(clk, noisysignal, conditioned, positiveedge,
negativeedge);
output reg conditioned = 0;
output reg positiveedge = 0;
output reg negativeedge = 0;
input clk, noisysignal;

parameter counterwidth = 3;
parameter waittime = 3;

reg[counterwidth-1:0] counter =0;
reg synchronizer0 = 0;
reg synchronizer1 = 0;

always @(posedge clk ) begin
       if(conditioned == synchronizer1)
              counter <= 0;
       else begin
              if( counter == waittime) begin
                     counter <= 0;
                     conditioned <= synchronizer1;
              end
              else
                     counter <= counter+1;
       end
       synchronizer1 = synchronizer0;
       synchronizer0 = noisysignal;
end
endmodule
```

Input Conditioner Deliverables

The complete module should be in “InputConditioner.v” together with the test bench you used to verify its behavior.

Include InputConditioner.do. This do file should execute the test bench, view appropriate signals and then zoom the wave form to fit.

In the write up include a drawn circuit diagram of the structural circuit this could create. Acceptable primitives include d flip flops, adders, muxes, and basic gates.

If the main system clock is running at 50MHz, what is the maximum length glitch that will be suppressed by this design for a waittime of 10? Include the analysis in your Writeup.

Shift Register

Create a Shift Register. This shift register will support both "Serial In, Parallel Out" and "Parallel In, Serial Out" modes of operation. It should have the following module definition:

```
module shiftregister(clk, sclkEdge, parallelLoad, parallelDataIn, serialDataIn,
parallelDataOut, serialDataOut);
parameter width = 8;
input  clk;                        // FPGA Clock
input  sclkEdge;                   // Edge indicator
input  parallelLoad;               // Load Shift Register with parallelDataIn
output[width-1:0]   parallelDataOut;    // The contents of the Shift Register
output              serialDataOut;      // Positive Edge Synchronized
input[width-1:0]    parallelDataIn;     // load Shift Register in parallel
input               serialDataIn;       // load Shift Register serially
```

The shift register is clocked by the main system oscillator running at 50MHz. All behaviors are synchronous to this clock:

1) When the peripheral clock has an edge, the shift register advances one position: serialDataIn will be in the LSB (Least Significant Bit), and everything else will shift up by one. This uses the Input Conditioner's edge detection capabilities.
2) When parallelLoad is asserted, the shift-register will take the value of parallelDataIn.
3) serialDataOut always presents the Most Significant Bit of the shift register.
4) parallelDataOut always presents the entirety of the contents of the shift register.

Shift Register Deliverables

The complete shift register module should be in "ShiftRegister.v", together with a testbench that proves out the two behaviors.

Include ShiftRegister.do. This do file should execute the test bench, view appropriate signals and then zoom the wave form to fit.

In the write up, describe the test bench strategy.

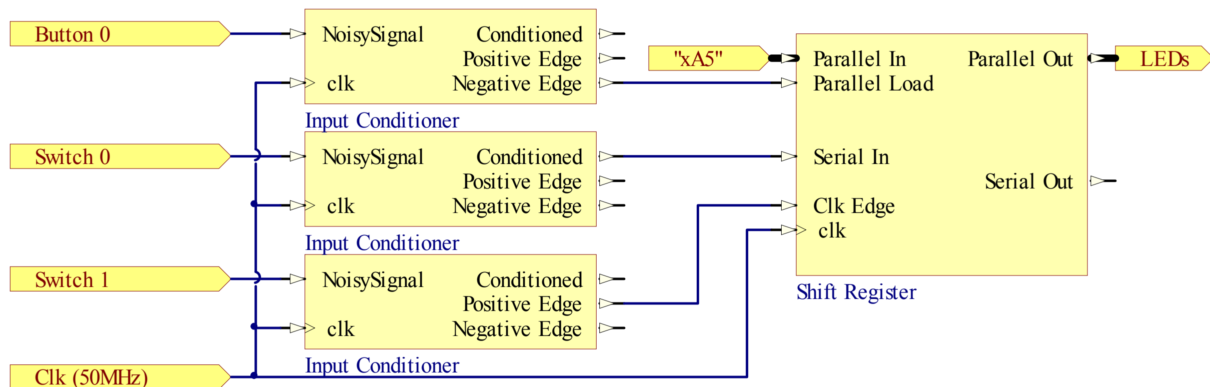
Shift Register Hints

Each of the 4 behaviors can be implemented in one or two lines of behavioral Verilog. You may want to look at Verilog's {} syntax for implementing the serial behavior.

It is good design practice to decide which behavior will "win" if a parallel load and a serial shift happen in the same clock edge. Otherwise the synthesizer will have to make that decision. I tried 4 different syntaxes, only of them threw a warning related to this.

Shift Register Bring Up

Create a top level module with the following structure and load it onto the FPGA:



The Parallel Load feature is tied to the constant value of 0xA5, and is activated when Button 0 is pressed.

Switches 0 and 1 allow manual control of the serial input.

The LEDs show the state of the shift register.

Start by copying the “mp2.v” file to “TopLevelShiftRegisterBringup.v”. Add the provided UCF, and add the “InputConditioner.v” and “ShiftRegister.v” files you’ve already created. Note that the provided top level file may include inputs or outputs that you do not need – you can safely ignore them.

Shift Register Bringup Deliverables

The Top Level test module should be in “TopLevelShiftRegisterBringup.v”. Testing is done by hand; no verilog test bench file module is required for this file.

Design a test sequence that shows the successful operation of this portion of the lab. In your writeup provide a written description of what the test engineer is to do, and what the state of the LEDs should be at each step. Record and link to a short (60 seconds or less) video of the test being executed.

Include an analysis of the summary statistics provided by ISE. This should be roughly a paragraph that touches on at least the following:

1) How many Logic Slices are used?
2) What Logic elements are used? Counters, FlipFlops, etc
3) What is the maximum clock speed for the clk net?

SPI Memory

You now have everything you need to create the complete SPI Memory. This is a 128 byte memory.

It will have the following definition:

```
module spiMemory(clk, sclk_pin, cs_pin, miso_pin, mosi_pin, leds, fault_pin);
input        clk;
input        sclk_pin;
input        cs_pin;
output       miso_pin;
input        mosi_pin;
output[7:0]  leds;
input        fault_pin;
```

The SCLK, CS, MISO, and MOSI signals obey the SPI standard. The Fault Injector will be used in the next section, we're just scaffolding it in for now. The LED outputs are in case you need debugging information.

Behavior

Each transaction begins with the Chip Select line being asserted Low. Whenever Chip Select is high the memory ignores all other inputs, tri-states MISO, and resets any communication state machines.

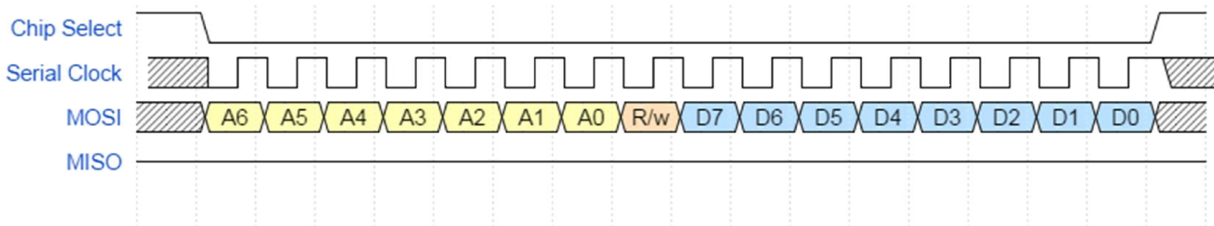
Then, 8 bits are clocked in by the Master. The first 7 bits are the memory address, Most Significant Bit first. The 8th bit is the $R/\overline{W}$ flag: Read when high, Write when low.

For a Write Signal, the master will then clock in 8 bits of data and de-assert Chip Select.

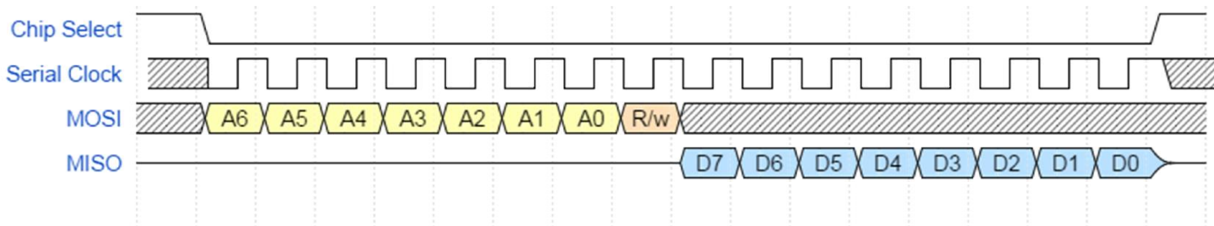
For a Read Cycle, the slave will assert MISO and clock out the data found at Address.

Data is always presented on the falling edge, and always read on the rising edge of SCLK.

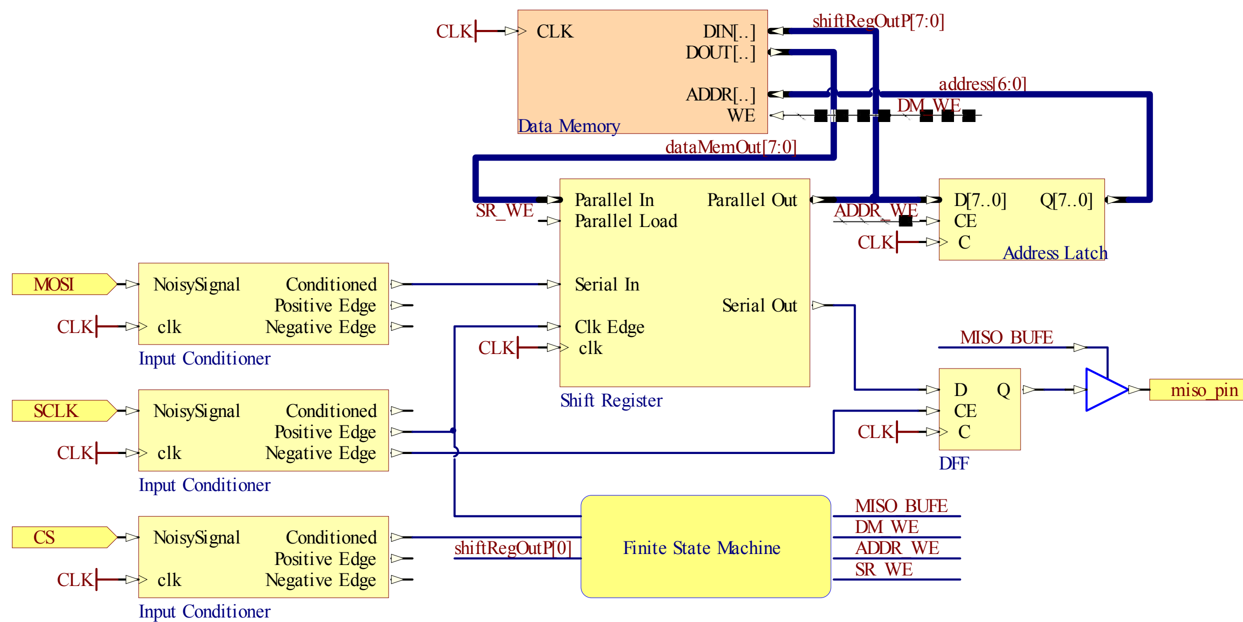
Write:



Read:



Schematic View



This is a _recommended_ schematic for the SPI Memory. The next few pages provide additional scaffolding for this design route, but you may choose to implement your SPI Memory by any means you find appropriate. Stubbed signals are controlled by the Finite State Machine. You may require additional signals as inputs to the FSM. System Clock routed as a net label for clarity.

The Serial Out pin is synchronized to the falling edge of SCLK to obey the standard we are using (Data is presented on the falling edge, and captured on the rising edge).

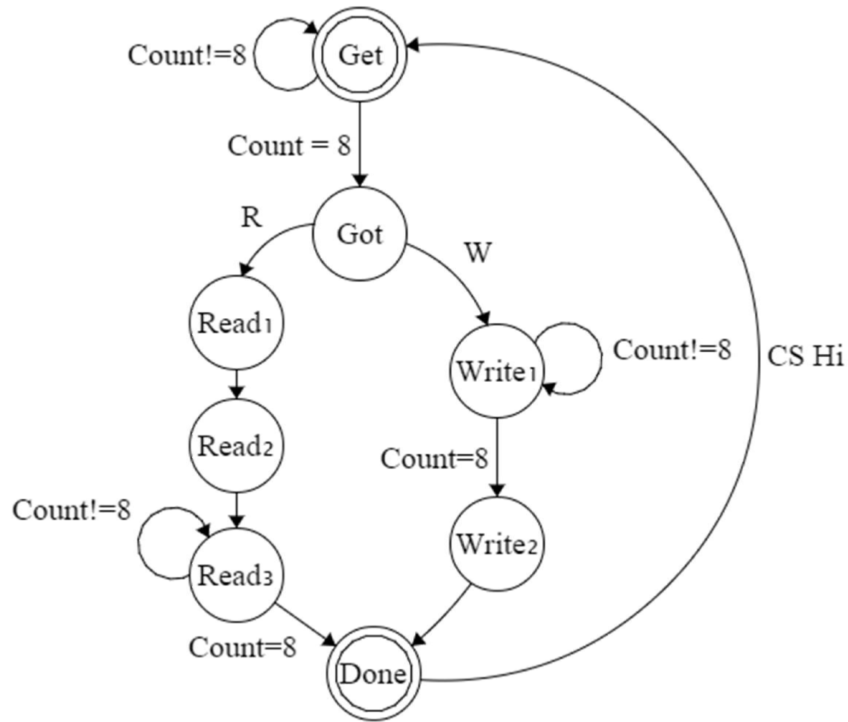
Finite State Machine

This state machine provides the appropriate control signals to drive the schematic above. It begins in "Get (address)", where it waits for 8 SCLK strobes. It then proceeds to the intermediate step "Got (address)", where it grabs the address from the shift register. This extra step provides an extra internal clock cycle for the shift register to finish its job and propagate the answer. Based on the $R/\overline{W}$ it then proceeds to Read1 or Write1.

Read1 is the cycle in which the DataMemory is read. Read2 proceeds immediately after, which pushes that value into the shift register. It then hangs out in Done, as the shift register will handle the rest of the Read Transaction. In this incarnation further checking is not necessary, as we have not defined the behavior when more than 8 bits are read out.

Write1 does count the number of bits strobed in, and when it reaches 8 it transitions to Write2. Write2 commits the data value to data memory.

If at any time CS is de-asserted (raised high), this state machine resets the counter to zero and the state to Get. These transitions are suppressed in the diagram to avoid Cthulu's Spaghetti. This is represented by the RESET line in the FSM description.



State	ShiftReg_WriteEnable	Reset_Counter	DataMem_WriteEnable	Address_LatchEnable	MISO_Enable	Next State	Condtion	Next State	Condition
N/A RESET		1				Get	*		
0 Get						Got	Count = 8	Get	*
1 Got		1		1		Read	R/W = 1	Write	R/W = 0
2 Read						Read2	*		
3 Read2	1					Read3	*		
4 Read3					1	Done	Count = 8	Read3	*
5 Write						Write2	Count = 8	Write	*
6 Write2			1			Done	*		
7 Done		1				Done	*		

SPI Memory Bringup

Repeat the process in "Shift Register Bringup" to create "TopLevelSPIMemory.v". The top level module should be little more than an instantiation of spiMemory and hooking the ports up correctly.

Tester

Construct a tester that will physically plug into your FPGA and verify proper operation. An Arduino is one way to do so. This portion of the lab can be done in parallel with the other parts.

Specifications:

Electrical Interface

The tester should provide the following signals on a 100mil 1x5 Male connector:

1) Ground
2) Chip Select
3) MOSI
4) MISO
5) SClock

SClock should run at about 500kHz. Memories will be tested at this speed, make sure your input ranking is set at a level that can handle this.

User Interface

The control interface is a single “TEST” button. When this is depressed, it will become happy again by initiating the test sequence. This could be your board’s reset button, or you could make it prettier.

Three LEDs will provide feedback during the test:

1) In Progress – This LED is lit during the test sequence.
2) Pass – This LED lights at the end of the test sequence if the unit passed.
3) Fail – This LED lights at the end of the test sequence if the unit failed.

Optionally include feedback through a USB serial port. This is not required, but highly recommended.

Strategy

Your write up should include a detailed analysis of your tester strategy. Why did you choose the test sequence you did?

Breakable SPI Memory

You now have a functioning SPI Memory and a tester capable of confirming its behavior. The final step is to break your memory and make sure that your tester can detect the failure. This is known as "Fault Injection".

The failure mode you will be injected should simulate a manufacturing defect. That is to say that the design is correct, but one of the thousands of copies coming off the line was damaged during manufacture. Look through your design and identify a single gate or wire to break. It can break by being stuck High, stuck Low, or shorted to a nearby net.

Copy and modify your existing SPI Memory so that it is now sensitive to the "`faultinjector_pin`" signal. When it is low, the memory should work correctly. When it is high, the memory should exhibit the injected failure mode.

This signal is tied back to a switch. You should now have a device that will break at the flip of a switch.

When you modify your hierarchy, you'll need to propagate the fault injection signal. In doing so, make sure not to break any of your existing modules. Instead, create copies of those modules and append "_breakable" to their module name. Add the fault signal as the last input to these breakable modules. Place these breakable module definitions in the same file as their indestructible partners.

For example:

```
module inputconditioner_breakable(clk, noisysignal, conditioned, positiveedge,
negativeedge, faultactive);
```

Breakable SPI Memory Deliverables

In your writeup, describe the fault you are injecting. Include a schematic of the area local to the fault to better identify which gate or wire is broken.

In your writeup, explain one specific test pattern that will identify this failure mode.

Example Broken SPI Memory

Our injectable failure mode is that the 1st bit of the address bus is permanently 1. This could occur if this d-flip flop (schematic) was broken during manufacturing. We simulate this by ORing this signal with the fault injection signal. This can be identified during test by noticing that adjacent memory locations have been merged. For example, writing to address 10 affects address 11 as well.

Test Day

We will have an in-class Test Day. Teams will bring their FPGA and their Tester. We'll keep friendly score to see whose testers can detect whose faults.

Notes / Hints

You may need to adjust your deglitching period differently for when it is driven by switches vs when it is driven by the tester. Switches are much noisier.

Autograde Nicety

The autograder will compile **all** Verilog files that are in your archive. This means that things like "InputConditioner_OLD.v" may cause naming conflicts. Make sure that you do not have multiple modules with the same name.

Do File updates:

To change ModelSim's tab spacing, use the following command:

```
set PrefSource(tabs) 4
```

This can be done in the command window. I've added it to my own do-files, as ModelSim seems to have amnesia.

The following commands flush the work library. This makes issues associated with stale module definitions fail sooner with better error messages:

```
vdel -lib work -all
vlib work
```

The first command will fail if the library does not exist yet. You may need to manually execute the second one when you create a new project.

FPGA Preparation

Make sure that each of your always blocks' sensitivity lists are only "`always @(posedge clk)`". Other sensitivity lists may cause clock gating.

The Xilinx synthesizer obeys the "initial" block syntax with varying degrees of success. To initialize a register, do so in its declaration:

```
reg regname = 0;
```

This technique does not work with 2 dimensional arrays.

FPGA Clock Setup

In order to ensure clock consistency on this lab, find the pins on your FPGA labeled 100MHz, 50MHz, and 25MHz and ensure that there is no jumper on these pins. (You can store the jumper on the pins labeled "jumper storage".) No jumper means that the clock will run at 50MHz. See page 4 of the FPGA manual (http://www.digilentinc.com/Data/Products/NEXYS/Nexys_rm.pdf) for more information.

FSM Debugging

ModelSim doesn't have enumeration support. To make your debugging life a little easier, define your state machine states with parameters and virtual signals.

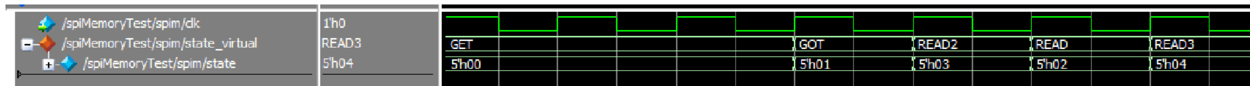
Using parameters local to the module that uses the states keeps your name space a little cleaner.

```
parameter state_GETTING_ADDRESS   = 0;
parameter state_GOT_ADDRESS       = 1;
parameter state_READ_1            = 2;
parameter state_READ_2            = 3;
```

In your .do file you can define a virtual type:

```
virtual type {{0 GET} {1 GOT} {2 READ} {3 READ2} {4 DONE}} state_type
virtual function {(state_type) /spiMemoryTest/spim/state} state_virtual
add wave -position insertpoint  \
       sim:/spiMemoryTest/spim/state_virtual
```

This syntax creates virtual type state_type, where 0 is mapped to "GET", 1 is mapped to "GOT" and so on. The next line copies the state machine state as a new virtual signal with the type state_type. This gets injected into the simulation GUI. You can then plot it like a normal signal.



You will need to keep the parameter and virtual definitions synchronized by hand.

Glossary

Input Synchronization - Synchronizing a signal to a clock domain.

Input Debouncing - Suppressing the oscillations on a input signal

Grading Rubric

```
Work Plan
       Initial
       Mid Lab
       Final
Input Conditioner
       Module Works (Autograded)
              Edges Generated
              Glitching Suppressed (Tested with waittime = 5)
       Include InputConditioner.do (Yes or No)
Circuit Diagram (Drawn, in final PDF)
Glitch Suppression Analysis (Written, in final PDF)
Shift Register
Shift Register Bringup
```

Test Vector explanation (Written, in final PDF)
Test Vector execution (Video, linked)
Synthesis Analysis (Written, in final PDF)

SPI Memory

Device will be subjected to a 10 part autotest sequence. Partial credit proportional to the number of tests that pass. (should we provide access to this?

Tester

Breakabe SPI Memory

Explanation of Failure Mode (Written, in final PDF)
Schematic of Failure Mode (Drawn, in final PDF)
Breakage is "Reasonable" (Hand Graded by NINJAs)
Actually breaks (Tested on Test day by your classmates)

Change Log

10/23/14

4: Added explanation of what circuit primitives are allowed in the schematic.

5: 1) When the peripheral clock has an edge, the shift register advances one position: serialDataIn will be in the LSB (Least Significant Bit), and everything else will shift up by one. This uses the Input Conditioner's edge detection capabilities.

6: Removed "lr" from "Shift Register Bringup Deliverabes

6: Changed TopLevel.v to mp2.v

6: Removed extraneous input from Shift Register

15: Added note about being nice to the Autograder