The University of Texas at Austin

EE460M Lab Manual

Dept. of Electrical and Computer Eng.

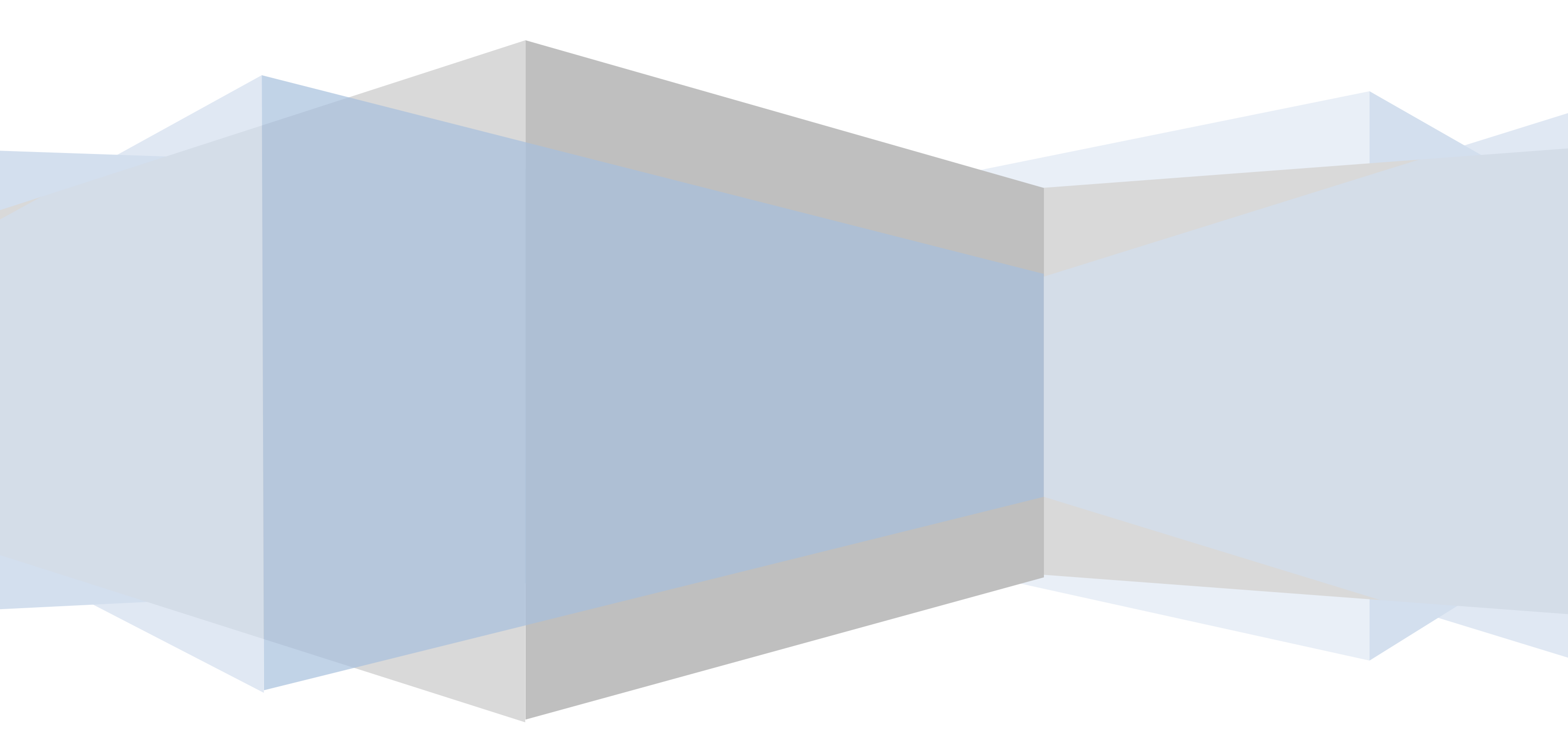


Table of Contents

TABLE OF CONTENTS 2

ABOUT THE MANUAL 3

LABS AT A GLANCE 4

LAB POLICIES 5

FREQUENTLY ASKED QUESTIONS 6

LAB ASSIGNMENT #0 16

LAB ASSIGNMENT #1 18

LAB ASSIGNMENT #2 22

LAB ASSIGNMENT #3 27

LAB ASSIGNMENT #4 4

LAB ASSIGNMENT #5 4

LAB ASSIGNMENT #6 13

LAB ASSIGNMENT #7 18

LAB ASSIGNMENT #8A 2

LAB ASSIGNMENT #8B 22

LAB ASSIGNMENT #9 25

LAB ASSIGNMENT #10 28

APPENDIX 31

LAB ASSIGNMENT – ARM PROCESSOR 32

About the manual

This document was created by consolidation of the various lab documents being used for EE460M (Digital Design using Verilog). It is intended to serve as a lab manual for students enrolled in EE460M at the University of Texas at Austin.

The creation process started towards the end of Spring 2011 and was accomplished by Aman Arora (TA, EE460M) under the guidance of Prof. Lizy John. In its present form, this document includes several changes (additions, deletions and modifications) incorporated over three semesters – Spring 2011, Fall 2011,Spring 2012. During the Spring/Fall 2013 semesters, all the labs were translated from VHDL to Verilog by Daniel Arulraj.

Several important modifications include:

1. Re-organization of Lab#1 and Lab#2 to remove several unimportant and quaint problems
2. Consolidation of tutorials which were spread over Lab#1 and Lab#2 into Lab#0
3. Addition of Lab#6B, which caters to design for test (DFT) concepts
4. Changes in values/design parameters in various labs
5. Re-organization of Lab#5 into three parts
6. Adding the ARM processor lab and the bowling score keeper lab in the appendix
7. Addition of several important details to improve clarity
    a. Mostly answers to students doubts
    b. Several diagrams
    c. Additional explanations
8. Convert the lab manual to Verilog
9. Added Lab#8,9,10

This document is currently maintained by Daniel Arulraj. He can be contacted through email at daniel.arulraj@utexas.edu. Please write to him in case of any questions or concerns or suggestions.

Important: Do not print this entire document. This document will be updated during the semester.

Labs at a glance

S.No.	Brief Description	Objective	Duration	Points Possible
0	Tutorials – ModelSim and Xilinx ISE and Nexys2 Board	Introduction to digital design using FPGAs. Introduction to simulation and synthesis.	1 week	50
1	Subtractor and ALU	Simple combinational circuit design	1 week	100 (40+40+20)
2	Excess-3 code converter and BCD counter	Simple sequential circuit design	1 week	100 (40+30+30)
3	Package sorter and Traffic Light Controller	More digital design. Introduction to testbenches.	1.5 weeks	120 (20+50+50)
4	Parking Meter	Advanced digital design. Interfacing with 7-segment display and push buttons.	2 weeks	150
5	A basic SNAKE game	Interfacing with PS/2 Keyboard and VGA display	2 weeks	180 (50+50+80)
6	Stack Calculator	Using Block RAMs on FPGAs	1 week	100
7	MIPS Processor	Basic microprocessor design	2 weeks	150
8	Memory BIST	Understanding JTAG and BIST	1 week	100
9-OPT	Bowling Score Keeper	State machines, logic design	2 weeks	HW (6%)
10-OPT	Floating Point Unit	Arithmetic Units, logic design	1.5 weeks	HW (4%)

Important: Please check the schedule sheet on Canvas for the lab due dates

The OPTIONAL labs (9 and 10) are an alternate for paper and pencil homeworks.

Lab Policies

1. You will (have access to and) work in the lab in ENS 302. This is also where TA office hours will be held.
2. This document, available on Canvas, will serve as the lab manual for the entire semester. The document contains all the lab information you need to do the labs (except for few codes in labs 6 and 7). You can work on your own pace throughout the semester, but you have to follow the due dates for submission (listed in the schedule document) and the check out procedures.
3. All communication will be done through Canvas. So, please keep checking Canvas for notifications and updates. Important information will also be emailed.
4. 15-minute lab discussion sessions will be held at appropriate dates (listed in the schedule document) before the lecture. These will be conducted by the TAs. It is advisable to read about that lab from the lab manual before coming to the class, so that you are better prepared to ask questions and resolve doubts.
5. Labs 0, 1 and 2 are to be done individually. Labs 3 through 7 can be done in groups of two. Also, working in groups does not mean that you work on separate parts of the lab. Both the group members are supposed to know and answer questions about all parts of the lab. You can switch partners whenever you want.
6. Grading will occur in two parts: submission and demo (checkout).
7. For submission, upload all relevant files (specified with each lab under the ‘Submission Details’ section) via Canvas. One of the members from each group should log into Canvas and go to “Assignment” section and then upload all the necessary files under the appropriate link.
8. Lab due dates (submission dates) are specified in the course schedule document on Canvas.
9. After you submit your files, you have to demonstrate your designs to one of the TA’s in the ENS 302 lab. Once the lab is submitted, DO NOT make changes! You must demo with the code you submitted. In the event you decide to change the code for the demo, the day of the demo will be considered the turn-in date, and the appropriate late penalty will be applied.
10. A checkout sign-up sheet is available on Canvas. After every lab due date, the TAs will email the class to sign-up for a checkout slot. Put your name in that sign-up sheet and reserve a time-slot for your check out. Please reach the lab at least 5 minutes before your slot. In case of group labs (lab 3 and above), only one member of the group should submit the files but both members of a group must checkout together. So, the entries in the checkout slot registration sheet should contain two names.
11. In case you miss your check out slot, you can check out for that lab during office hours anytime before the next lab’s due date. In other words, the TA’s will not entertain requests for checking out labs older than the previous lab.
12. The possible points for each lab are mentioned in the ‘Labs at a glance’ section of this manual. Late submissions (not late checkouts) will lead to penalty according to the following rules:
    a. One day late submission – less 10% of your normal score
    b. Two day late submission – less 20% of your normal score
    c. Three day late submission – less 30% of your normal score

    Submissions late by more than 3 days will not be accepted and you will be marked zero (unless you have taken permission from the professor).
13. Sundays are not counted for late submissions. So, if a lab is due on Saturday and you submit it on Monday, it will be considered 1-day late submission.

Frequently Asked Questions

MODELSIM

Q. In ModelSim 6, when I click the message saying *x* errors in the transcript window, the window that pops up does not show me any errors?
This is because you file name (complete path) has spaces in it. While using ModelSim, please make sure that the file name doesn't have any white spaces. In other words, do not have your programs saved on a path like "xyz\Documents and Settings\user1\lab 1\file.v". Please make a folder on the Z: drive of the computer you work on and keep your project/source files there.

Q. When I click on ModelSim, it gives me an error saying failed to checkout license.
In case invoking ModelSim shows a licensing error on the lab computers, please run the Licensing Wizard first (Start->Programs->ModelSim->Licensing Wizard), and then launch ModelSim.

Q. How do I create and run a do-file?
The ModelSim tutorial talks about creating a file of commands in the end (called a do-file), but does not explain how to do it clearly. Here is how you can do this: Basically, the commands like force, run, etc that you provide on the transcript window can be saved in a file and that file is called a "do-file". The benefit of having a do-file is to be able to re-run all the commands by just a single click, rather than typing them again and again. For example, if you have your do-file ready during the checkout, you can just execute it instead of typing the individual commands all over again.

There are two ways of creating a do-file.

1. You can manually write those commands in a file using a text editor and save it with a ".do" extension.
2. You can type the commands on the transcript window, and then have ModelSim create the file for you. For this, type the commands in the transcript window (keep the transcript window selected). Then go to "File->Save As", and then provide the name of the file with a .do extension.

To execute the commands in the do-file, make sure the transcript window is active. Then, go to "File -> Load" and then provide your do-file to the tool.

Q. Can I view variables on waveforms?
Viewing variables on the waves is just like viewing signals (of course, you should be simulating your design to view the waves). The ‘Objects’ window shows you the signals in a design. Similarly, the “Locals” window shows the variables in the selected module/always block. For seeing variables, go to the ‘View’ menu and click on ‘Locals’. A ‘Locals’ window will appear.

When you are simulating, you can see that a “Sim” pane appears near to your “Project” and “Library” panes. Click on the “Sim” pane and it will show you the design hierarchy. You can click on any module or a line number of an always statement whose variables you want to see. Now, in the “Locals” window, you can click on variables and then drag to the waveform window.

Q. Some signals in my design are not visible in the "Objects" window, and so I can't view their waveforms.
This is because ModelSim performs a series of optimizations on your design and can get rid of some signals. The 'optimized out' signals cannot be seen in the "Objects" window. You can disable optimization in two ways:

1. While starting simulation, instead of just double clicking on the module name in the "Library" window, right click and say "Simulate without Optimization".
2. On the transcript window, append " –novopt" to the "vsim" command

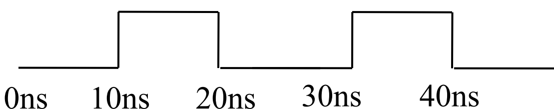
Q. Can I view waveforms of signals inside the design hierarchy (modules other than the top module)
When you are simulating a design, you can see that a "Sim" pane appears near to your "Project" and "Library" panes. Click on the "Sim" pane and it will show you the design hierarchy. You can click on any module in the design. When you click on a module, the "Objects" window shows the signals in that module. Now, in the "Objects" window, you can click on signals and then drag to the waveform window (or you can right click a signal and say Add->To Wave->Selected Signals).

Q. How can I change the way signals are shown on the waveforms (To change viewing 000101 to 5)
Right on the signal in the "Waves" window, got to "Radix" and select the one you want.

Q. How can I create a clock using the force statements in ModelSim during simulation?
To create/generate a clock, you can use the following command:

force clk 0 0 ns, 1 10 ns –repeat 20 ns

This command creates a clock of period 20 ns with 50% duty cycle as shown below:



You can change the period and duty cycle as you want by modifying the force statement appropriately.

Q. I used the 'force' command to force a signal. Now I want the design to drive it. But it is just stuck to that value.
A force statement forces the specified value onto the specified signal at the specified time and then that value remains on that signal for the entire simulation. It can only be changed by another force statement.
Adding "-deposit" option to the "force" command puts the specified value on the specified signal at the specified time, but lets it change anytime after that (if another driver wants to modify/override it; for example an assignment statement in the design).
For example, let's assume that you have an output that you want to initialize to 0 at the beginning of the simulation. Assuming also that you have not initialized this output to 0 in your code, you may simply type: force Z 0 0 ns. You will note after running the simulation that Z never changes. To overcome this problem, change the above statement to: force –deposit Z 0 0 ns. The deposit will simply deposit the value of 0 to Z at 0ns instead of freezing it at 0.
The "-cancel" option cancels the force on a signal at a specified time.
You can look into more options of the force statement by going to "Help -> PDF Documentation -> Reference Manual" in ModelSim.

XILINX ISE

Q. Xilinx ISE is so slow! What should I do?

It is recommended, in general, to work on files in the local directories (C: drive, for example) while working with Xilinx ISE and ModelSim in the lab. Your desktop is a networked drive and these tools work really slow when they have to fetch files over the network. But since the files in local directories get cleaned up when you log out, make sure you make a copy somewhere before you logout.

Q. What is a UCF File? Can I hand write it instead of using the PACE window?

The UCF file is the file which tells Xilinx ISE to map the inputs and outputs of your design to specific pins on the FPGA. The file also has other things like clock constraints etc, but we are not going to be concerned about them in this lab. The PACE tool helps you graphically create the UCF file. However, you can manually write a UCF file too (it is just a text file with a specific format). This may sometime be required if the PACE tool does not work. Assuming the the PACE tool works in your first lab, to view the UCF generated by it, click on the UCF filename in the Design Hierarchy panel. Now, in the processes panel, expand 'User Constraints' by clicking on the '+' sign. Now double click, 'Edit Constraints'. This will open the UCF file in the right hand side of the window. The syntax is self-explanatory. So, if PACE doesn't work in any lab, open the UCF from a previous project, copy it to the current project and modify it manually for the current project's constraints.

VERILOG

Q. Can I model combinational logic using always statements? How?

Ideally, concurrent statements are used to model combinational logic and always statements are used to model sequential logic (flip flops and latches). However, always statements are not restricted to that. You can model combinational logic using them. But it is important to note that when using an always statement to make combinational logic, the sensitivity list of the always statement should contain all the signals which are being 'read' in that always block. In other words, to synthesize combinational logic using an always block, all inputs must appear in the sensitivity list.

For example, if you were to model a mux, you would say:

```
always @(a, b, sel)
begin
    if (sel == 1) z <= a;
    else z <= b;
end
```

Using a always statement to model combinational logic is handy because statements like if, case, etc (which are very useful and intuitive) can only be written inside always statements.

Q. What care should I take when using the always statement to write sequential logic?

When using an always statement to model sequential logic, the only thing in the sensitivity list of the always statement should be the clock (or a reset signal, if it is an asynchronous reset). And there should be a' 'posedge' or 'negedge' in the sensitivity list before the clk. This is because flip-flops are edge triggered elements.

Flip-flop without a reset

```
always @(posedge clk) //positive edge triggered
begin
      q <= d;
end
```

Flip-flop with an async reset

```
always @(posedge clk, negedge rst) //positive edge triggered with reset
begin
if(rst == 0)     //async active low reset
    begin
    q <= 0;
    end
else
    begin
    q <= d;
    end
end
```

Flip-flop with a sync reset

```
always @(posedge clk) //positive edge triggered
begin
if(rst == 0)     //sync active low reset
    begin
    q <= 0;
    end
```

```
else
    begin
    q <= d;
    end
end
```

On the other hand, a latch is a level triggered element. A resettable latch can be modeled as:

```
always @(en, rst, d)
begin
if(rst == 0)
    begin
    q <= 0;
    end
else if(en == 1)
    begin
    q <= d;
    end
end
```

Q. Why can I not instantiate a module inside an 'if' statement (or an always block, for that matter)?

It is important to realize that a module is not like 'calling' a function in C. It is an instantiation of that module. Therefore, it cannot be conditional. If you have to instantiate a block in your design, it will be always present there.

Let us take an example. Say you have an adder and a subtractor. You design's specifications say that when the input MODE is '1', the design should work as an adder, while when the MODE is '0', the design should work as an subtractor. Now, this does not mean that you can have something like this:

```
always(...)
begin
    if(MODE == 1)
        adder adder_inst(A,B,Sum);
    else
        subtractor subt_inst(A,B,Diff);
end
```

Since we are modeling hardware, we cannot say that if MODE is 1, Adder is 'called' and when MODE is 0, subtractor is 'called'. This is a wrong way of thinking.

Instead you should think of this as: Adder and Subtractor are always present. The output of the design can be driven by either the Adder or the Subtractor depending on MODE. So you should have something like this:

```
adder adder_inst(A,B,Sum);
subtractor subt_inst(A,B,Diff);

always(...)
begin
    if(MODE == 1)
        output_ALU <= Sum;
    else
        output_ALU <= Diff;
end
```

GENERAL

Q. What tests should the 'do' file that I submit on canvas contain?

It is always better to submit a do-file which has sufficient number of input combinations (not just the ones given in the lab description).

Q. I am getting a multiple drivers error. What should I do?

A multiple driver error is because there is more than one thing driving a signal. This can happen if you are driving a signal from two sources: like one always block and one concurrent statement, or two always blocks. Realistically, it is not possible to do so (without having contention, which we are staying away from). There is nothing you can do to get rid of this, other than changing your design.

Q. My design compiles successfully in ModelSim. When I simulate, I get weird errors (error loading design, etc) and I can't simulate.

The compilation process looks at individual modules in your design and checks for syntactical and semantic correctness. Simulation lets you apply inputs and observe outputs. Between compilation and simulation, is a step called elaboration (which is usually hidden from you, and happens when you start simulation in ModelSim). During this step the design hierarchy is generated. Connections between various modules, and search for entities referenced as components in a design, etc are done at this stage. If there is a problem at this stage (for example, there is a component declaration in your top module but the module for that component is missing), they are reported just before simulation. So, now you know where to look for when you get errors just when you start simulation.

Q. Will setup and hold time be met in my simulation? Or If I add some logic between two stages in my design, will the delay affect the output? Or should I force my input sometime before the clock edge to satisfy setup and hold time constraints?

Remember that the simulations that you are doing in the lab are all RTL simulations. They are zero-delay simulations (assuming you are not modeling delays using '#' statements). Therefore, there is no concept of delays of gates or setup-hold time of flip-flops. If we were doing post-synthesis simulations, then we would have concerned timing issues.

GOOD DESIGN PRACTISES

Q. Are there any general 'good' design practices that I should follow?

1. Writing Verilog feels like writing software. But it is a good idea to think 'hardware' while writing code!
2. Do not use '#' (delay) statements in your designs in the lab. Testbenches may use these. Eg. To generate a clock signal in a testbench you can say "#10 clk = ~clk;"
3. A state machine can be designed using either a single always block (like Figure 2.56 in the text) or using two always blocks (like Figure 2.54 in the text). Both ways are correct. However, it is easier to design it using a single always block. Generally, the single always block partakes less debugging effort.
4. Stay away from 'variables' unless you are absolutely sure.
5. Concurrent statements are continuous drivers. Do not use them for initializations.
6. It is a good idea to have a reset signal in your design (even if not mentioned in the lab description). Use this signal to reset all the things you want to.
7. While simulating your design, it is always a good idea to stagger your inputs with respect to the active clock edge. For example, if your active clock edge is occurring at 10ns, apply your inputs sometime before 10ns, say at 8ns. This ensures that when your design was clocked, the input was successfully read. If your active edge occurs at 10ns and your input also changes at 10ns, then it becomes hard to see whether the input was successfully captured by the clock edge or not. Debugging becomes harder if you have your inputs like that.
8. Don't limit your testing to the input sequences mentioned with the problem statement. During the checkouts, the TAs will apply several input combinations to test your design. So, make sure to do a thorough testing of your design using sufficient number of inputs.
9. Generally, we tend to ignore warnings from the tools. But make sure you look at all the warnings after the synthesis process is completed. Sometimes there are problems in your design like missing connections, latches, etc. Such issues make the tool infer your design differently from what you want or expect it to be. These warnings might contain the reason why your design does not work on the board.

Q. My design works in simulation. But it does not work on the board. What should I do?

There is no one sentence answer to this question. You can try the following things to help you debug your problem:

1. Follow the good design principles discussed above.
2. Look for any warnings in the synthesis report.
3. Make sure there are no latches in the synthesized output.
4. Follow the synthesis-friendly code guidelines discussed in the next question

Q. My design works in simulation. But Xilinx ISE throws an error during synthesis, saying "Bad Synchronous Description". What am I doing wrong?

The one line answer to this question is that you are not writing synthesizable code. Here are a few tips:

1. posedge/negedge should only be used on clocks
2. An always statement used to model sequential logic should only have clock (and reset, if you need one) in the sensitivity list, and an always statement to model combinational logic should not have clock in the sensitivity list. (this is illustrated in detail below)
3. A signal cannot change on both negative and positive edges of clock (This is specific to the design you do in the lab because the FPGA hardware does not have dual edge triggered flops. This is true for

most industrial design also. However, there may be some very high end designs which use dual edge triggered flops, in which case this constraint on your code gets removed.)
4. Make sure the tool is able to decipher the value of each signal under each condition.

Synthesis friendly 'always' statements

A. If you use always statement for a combinational logic, make sure the sensitivity list contains all inputs. And the clock should not be amongst those inputs! If you feel like you need the clock, it means you want to write sequential logic. Think again!

B. All always blocks other than the ones used for combinational logic will have a structure similar to this:

```
always @(posedge clk, negedge rst)
begin
if(rst == 0)    //async active low reset
    begin
    //initializations
    end
else
    begin   //positive edge triggered sequential logic
    //actual stuff
    end
end
```

C. So, any always block in your design should fall into either of the following categories:

```
always @(posedge clk, negedge rst) //model posedge triggered sequential logic with
reset
begin
if(rst == 0)    //async active low reset
    begin
    //initializations
    end
else
    begin
    //actual stuff
    end
end

always @(posedge clk) //model positive edge triggered sequential logic
begin
    //stuff
end

always @(a,b,c) //model combinational logic
begin
    //stuff
end
```

Q. Xiinx ISE reports there are latches in my design. Where am I going wrong?

Latches are caused when you forget an 'else' block in an 'if' or 'case' statement in a always block intended to make combinational logic. Look at your design and find such cases.

Example:

The following always statement was intended to make do some selection. It was expected that a mux will be generated for both f and g.

```
always @(sel, a, b, c)
begin
    case (sel)
        3'b000 : f <= a; g <= c;
        3'b001 : f <= b; g <= d;
        3'b010 : f <= a; g <= c;
        3'b011 : f <= b; g <= d;
        3'b101 : f <= b; g <= d;
        3'b110 : f <= a; g <= b;
    endcase
end
```

But notice that the assignment to 'g' was missed in one case. And one case (100) was not mentioned. Therefore, latches were inferred for both 'g' and 'f'. Here is the correct way to write this:

```
always @(sel, a, b, c)
begin
    case (sel)
        3'b000 : f <= a; g <= c;
        3'b001 : f <= b; g <= d;
        3'b010 : f <= a; g <= c;
        3'b011 : f <= b; g <= d;
        3'b101 : f <= b; g <= d;
        3'b110 : f <= a; g <= b;
        default : f <= a; g <= b;
    endcase
end
```

Also, in an always block used to model combinational logic, if you forget to assign all signals under all conditions, you will end up with latches. So, to synthesize combinational logic using an always block, all signals must be assigned under all conditions.

Example:

```
always @(state, a, b, c, d, e)
begin
 case (state)
    0: if (a == 0) next_state <= 1; //IDLE STATE
    1:  //INITIAL STATE
    begin
        if (a == 1) next_state <= 2;
        else next_state <= 3;
    end
    2:  ...
```

Since 'next_state' is not assigned when a is '1', a latch is inferred. To avoid unwanted latches, a good way is to make sure you assign all signal under all possible conditions.

```
always @(state, a, b, c, d, e)
begin
 case (state)
    0: //IDLE STATE
    begin
        if (a == 0) next_state <= 1;
        else next_state <= 0;
    end
    1: //INITIAL STATE
    begin
        if (a == 1) next_state <= 2;
        else next_state <= 3;
    end
    2:  ...
```

But an easier (sometimes; depends on functionality) way can be to create a default assignment for all the variables in the always block.

```
always @(state, a, b, c, d, e)
begin
 next_state <= 0; //default assignment
 case state
    0: if (a == 0) next_state <= 1; //IDLE STATE
    1: //INITIAL STATE
    begin
        if (a == 1) next_state <= 2;
        else next_state <= 3;
    end
    2:  ...
```

Lab Assignment #0

This lab is a tutorial lab. You don't have to design anything in this lab, just go through the tutorials and perform them on the lab computers individually. In this course, in almost all the labs we will be doing the following steps:

Step 1: Writing Verilog code of the circuit we want to implement
Step 2: Simulating the Verilog code using a simulator (ModelSim) to check if the intended functionality has been achieved
Step 3: Synthesizing the Verilog code using a tool form Xilinx called ISE so that it can be programmed onto an FPGA
Step 4: Programming the FPGA (a Spartan3E series FPGA from Xilinx) on the lab board (called Nexys2 board) using a tool called Adept
Step 5: Applying inputs to and observing outputs from our circuit using the peripherals (like switches, buttons, LEDs, etc) on the Nexys2 board

To be able to do all this, we need to learn how to use ModelSim and Xilinx ISE tools, and we also need to understand the capabilities of the Nexys2 board and how can we program the Xilinx Spartan3E FPGA on it using Adept. The following activities will help you go through all the steps so you can learn and use the concepts in the upcoming labs.

Activity 1: ModelSim tutorial

Mentor Graphic's Modelsim tool will be used to perform the functional simulation of our Verilog code for the course. This software is available in all of the ENS labs. Modelsim is also available as a free download with Xilinx's Webpack software so you can install it on your own computer..

Go through the "**Modelsim Tutorial**" posted on Canvas under "Files/Labs/Documentation". This tutorial goes through the basic steps in compiling and simulating within the Modelsim environment using a simple D-flipflop as an example.

Activity 2: Xilinx ISE tutorial

The XILINX ISE tool is used to synthesize circuits and place & route them for a particular FPGA. Then, a BIT file needs to be generated (we use the Digilent Adept tool for that) which can be programmed onto the FPGA so that the FPGA now contains the circuit you designed. Go over the XILINX tutorial that has been posted on Canvas. You may also visit www.xilinx.com and browse the Spartan 3e manuals for help.

Go through the "**Xilinx ISE tutorial**" posted on Canvas under "Files/Labs/Documentation".

Activity 3: Nexys2 board tutorial

Read through the **Nexys2 Board User Manual** on Canvas under "Files/Labs/Documentation" to understand the features and capabilities of the board to be used in all the labs. Then go through **Nexys2 Board Configuration** manual under "Files/Labs/Documentation". This document describes how to program the FPGA on the board.

~~Activity 4: Xilinx ChipScope tutorial~~

~~The XILINX ChipScope tool is used for debugging FPGA based designs. It is a software based logic analyzer that allows monitoring the status of selected signals in a design in order to detect possible design errors. The basic concept is that you generate some components (called cores) using a tool (called Core Generator), add these cores to your design, synthesize it and then put it on the board. Then you open up a tool on your computer (called ChipScope Analyzer) and you can observe the signals of your design on the screen. Which signals you want to observe is declared when you integrate the cores in your design.~~

~~Go through the "**Xilinx ChipScope tutorial**" posted on Canvas under "Course/Labs/Lab Info".~~

Questions

You should be able to answer (almost all of) the following questions after going through these tutorials:

1. What is the ModelSim? What is the role of the transcript window which appears on the bottom of the main ModelSim window?
2. What is a delta cycle in a Verilog simulator like ModelSim?
3. How do you create a do-file of commands entered in the transcript window in ModelSim?
4. Describe the roles and functionality of the following tools in the Xilinx ISE suite: Project Navigator, RTL schematic viewer, and PACE.
5. What is the purpose of using the Adept software?
6. Is it possible to display two digits using the 7SEG LEDs at the same time on the Nexys2 boards? Note there are only 7 pins corresponding to a single 7-segment digit.
7. If we want to use the push buttons on the board reliably, what should we do first to the incoming signal into the FPGA? How do we implement this in Verilog?

Submission and Checkout details

Submit a text/doc/pdf file containing the answers to the questions given above on Canvas. Name the file "your_last_name.extension". You need to demonstrate that you performed the tutorials during the checkout. Also, you will be asked questions about various aspects covered in the tutorial. Your ability to answer them and your demonstration will decide your score.

Lab Assignment #1

Guideline
This lab is to be done individually. Each person does his/her own assignment and turns it in.

Objective
To learn designing basic combinational circuits in Verilog and implementing them on an FPGA.

Problem 1: Subtractor Design
a. Write Verilog code for a 1-bit full subtractor using logic equations (Difference = A-B-Bin). If you use delays, make sure to simulate for long enough to see the final result.

b. Write Verilog code for a 4-bit subtractor using the module defined in part (a) as a component. If you use delays, make sure to simulate for long enough to see the final result. Test it for the following input combinations:
   1. A = 1001, B = 0011, Bin = 1
   2. A = 0011, B = 0110, Bin = 1

1-bit full subtractor truth table:

A	B	Bin	Diff	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Verify that your design works correctly by using the “force” and “run” commands in the transcript window to provide inputs and observe outputs on the waveform window.

Problem 2: ALU Design
Design an Arithmetic and Logic Unit (ALU) that implements 8 functions as described in Table 1. Table 1 also illustrates the encoding of the control input.

The 4-bit ALU has the following inputs:
- A: 4-bit input
- B: 4-bit input
- Cin: 1-bit input
- Output: 4-bit output
- Cout: 1-bit output
- Control: 3-bit control input

Table 1: ALU Instructions

Control	Instruction	Operation
000	Add	Output <= A + B + Cin; Cout contains the carry
001	Sub	Output <= A – B - Cin; Cout contains the borrow
010	Or	Output <= A or B
011	And	Output <= A and B
100	Shl	Output <= A[2:0] & ‘0’
101	Shr	Output <= ‘0’ & A[3:1]
110	Rol	Output <= A[2:0] & A[3]
111	Ror	Output <= A[0] & A [3:1]

The following points should be taken care of:

- Use a case statement (or a similar ‘combinational’ statement) that checks the input combination of “Code” and acts on A, B, and Cin as described in Table 1.
- The above circuit is completely combinational. The output should change as soon as the code combination or any of the input changes.
- You can use arithmetic and logical operators to realize your design.

Simulate this circuit by using the “force” and “run” statements in the transcript window to provide inputs and observe outputs on the waveform window.

Problem 3: Synthesizing and implementing the subtractor on the FPGA

Create a new project in Xilinx ISE. Use the code for the 4-bit subtractor that you wrote in Problem 1. Synthesize and implement the design on the Spartan3E FPGA on Nexys2 board. Use the following pin assignments for creating the UCF file:

A	Switches[7->4]
B	Switches[3->0]
Bin	BTN0
Diff	LED[3->0]
Bout	LED4

Download the design onto the board and make sure it works as expected. Include the *design_name.bit* file that you download to the board in your Canvas submission.

Useful Information

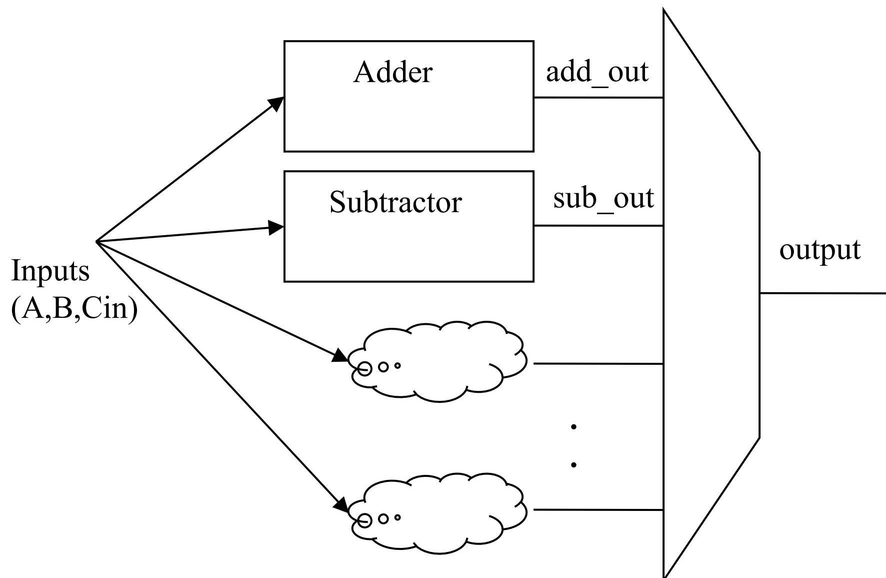
1. For problem 2, you can use the subtractor block from problem 1 for doing the subtraction (although just using the arithmetic operators will make your design easier). If you use the subtractor from problem 1, remember that we are designing hardware. So, doing something like the following is incorrect:

```
module xyz(…)
always @(…)
case (control)
0 : four_bit_sub sub_inst(in1,in2,bin,output);
……
```

First of all, it is important to realize that instantiating a module is not like 'calling' a function in C. Once instantiated, the module is always evaluating its inputs. Therefore, it cannot be conditional. It is always present. So, you should do something like this:

```
module xyz(…)
four_bit_sub sub_inst(in1,in2,bin,sub_out)
always @(…)
case (control)
0 : output <= sub_out;

……
```



2. Make sure that your designs work by testing them sufficiently thoroughly. You should not just use the test inputs in the lab description. Also, it is always better to submit a do-file which has sufficient number of input combinations (not just the ones given in the lab description).

3. Do not use # statements in your design for providing delays.
   #15 X <= A or B;
   In fact, you should never use the delay statement in the lab during the semester.

Submission Details

All parts of this lab will be submitted on Canvas only. You will not need to submit anything as a hard copy. Please zip all relevant files into a single folder with the following naming scheme: ***Lastname_Lab#.zip***

Problem	Submission Requirements
1	• Verilog file(s) • Do-file
2	• Verilog file(s) • Do-file
3	• Bit-file • UCF File

Checkout Details

You will be expected to describe briefly the codes for problems 1 and 2, simulate and show waveforms in Modelsim, and answer verbal questions. Also, for the last problem you will have to demonstrate that your circuit works on the board.

Lab Assignment #2

Guideline

This lab is to be done individually. Each person does his/her own assignment and turns it in.

Objective

To learn designing basic sequential circuits in Verilog and implementing them on an FPGA.

Problem 1: Excess-3 code converter design

In this problem, you will be designing an FSM using three different styles of Verilog coding: behavioral, dataflow, and structural. The following is the problem for which you will be designing the FSM:

A sequential circuit has one input (*X*), a clock input (CLK), and two outputs (*S* and *V*). X, S and V are all one-bit signals. *X* represents a 4-bit binary number *N*, which is input least significant bit first. *S* represents a 4-bit binary number equal to *N* + 3, which is output least significant bit first. At the time the fourth input occurs, *V* = 1 if *N* + 3 is too large to be represented by 4 bits; otherwise, *V* = 0. The value of *S* should be the proper value, not a don't care, in both cases. The circuit always resets after the fourth bit of *X* is received. Assume the sequential circuit is implemented with the following state table. The outputs are (S,V). All state changes occur on the falling edge of the clock pulse.

Present State	Next State		Output	
	X=0	X=1	X=0	X=1
S0	S1	S2	1,0	0,0
S1	S3	S4	1,0	0,0
S2	S4	S4	0,0	1,0
S3	S5	S5	0,0	1,0
S4	S5	S6	1,0	0,0
S5	S0	S0	0,0	1,0
S6	S0	S0	1,0	0,1

a. Write a *behavioral Verilog description* using the state table shown above. Compile and simulate your code using the following test sequence:

X = 1011 1100 1101 ←

The first input bit is at the far right. This is the LSB of the first 4-bit value. Therefore, you will be adding 3 to 13, then to 12, and then to 11. While simulating, keep the period of the CLK to be 10ns. Change X 1/4 clock period after the rising edge of the clock.

b. Write a *data flow Verilog description* using the next state and output equations to describe the state machine. You can use Logic Aid to derive the logic equations. Assume the following state assignment:

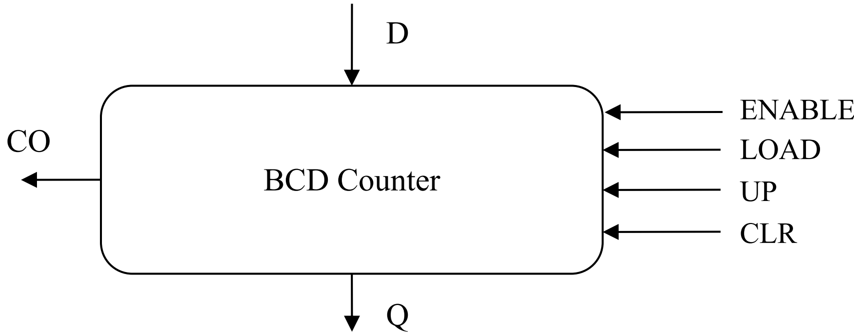
S0 = 000, S1 = 010, S2 = 001, S3 = 101, S4 = 011, S5 = 100, S6=111

Compile and simulate your code using the same test sequence and timing as (a).

c. Write a *structural model* of the state machine in Verilog that contains the interconnection of gates and D flip-flops. Compile and simulate your code using the same test sequence and timing as (a).

Problem 2: BCD Counter Design

Implement a 1 digit BCD (binary coded decimal) counter. It should be a synchronous (4-bit) up/down decade counter with output *Q* that works as follows: All state changes occur on the rising edge of the *CLK* input, except the asynchronous clear (*CLR*). When *CLR* = 0, the counter is reset regardless of the values of the other inputs. You can keep the time period of the CLK signal to 10ns for simulating your design.



If the *LOAD* = ENABLE = 1, the data input *D* is loaded into the counter.
If *LOAD* = 0 and *ENABLE* = *UP* = 1, the counter is incremented.
If *LOAD* = 0, *ENABLE* = 1, and *UP* = 0, the counter is decremented.
If *ENABLE* = 1 and *UP* = 1, the carry output (*CO*) = 1 when the counter's value is 9.
If *ENABLE* = 1 and *UP* = 0, the carry output (*CO*) = 1 when the counter's value is 0.

a. Write a Verilog description of the counter. You may implement your design in any style you wish. It will be easier to use a behavioral description which can be either written in the algorithmic way (eg. Count <= Count + 1 – Figure 2.46 in the text) or a state machine way (eg. State <= Next_State – Figure 2.54/2.56 in the text). You may also use dataflow or structural descriptions, although that will be more work. Use the following simulation for your waveforms:

1. Load counter with 6
2. Increment counter four times. You should get 9 and then 0.
3. Decrement counter once. You should get 9.
4. Clear the counter.

b. Write a Verilog description of a decimal counter that uses two of the above counters to form a two-decade decimal up/down counter that counts up from 00 to 99 or down from 99 to 00. In other words, instantiate two single digit counters in a top module (the two-digit counter). You may need some extra logic in the top module too other than these instantiations. The top module will have these inputs and outputs: CLR, CLK, ENABLE, LOAD, UP, D1, D2, Q1, D2, CO. Use the following simulation for your waveforms:

1. Load counter with 97
2. Increment counter five times.
3. Do nothing for 2 clock periods
3. Decrement counter four times.
4. Clear the counter.

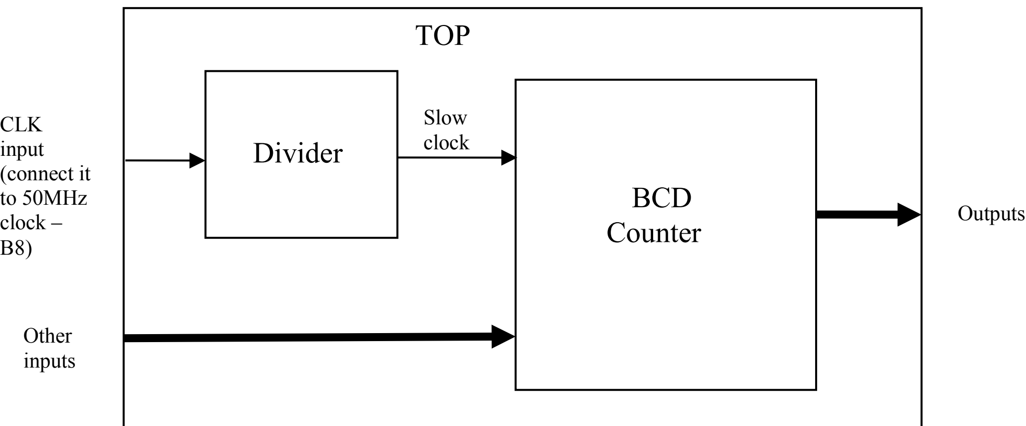
Problem 3: Synthesizing and implementing the BCD counter on the FPGA

Use the code for the single digit BCD counter that you wrote in Problem 2a. Before you synthesize it and implement it on the board, you will have to modify your code a little bit. This is because the CLK signal available on the board is a high frequency signal (50 MHz). If you use this high frequency for your circuit, you will not be able to give proper inputs or see proper outputs to your design.

So, you need to add a clock divider to your Verilog description. Create two more entities in your design. Call one as *top* and another as *divider.* Make connections as shown in the following figure. Look at the codes given in the end of this document, understand them and see how they can be used as clock dividers.

To look for latches in your synthesized design, open the synthesis report generated by ISE by clicking “View Synthesis Report” under the “Synthesize-XST” option. In the synthesis report, look for “Macro Statistics” and see if any latches are being shown. Alternatively, you can look for “cell usage” in the report and there should not be any cells under “Flip Flops/Latches” having names starting with “L”. Ensure that there are no latches in your design.

Also, after adding the counter/clock divider block to your design, simulate the top module in Modelsim before directly synthesizing using ISE to ensure that the counter/divider works. And while simulating, reduce the large values (like 5000000) in the counter to small values (say 50), so that simulation takes less time and the waveforms are legible. Don't forget to switch to the correct (large) value before synthesizing.



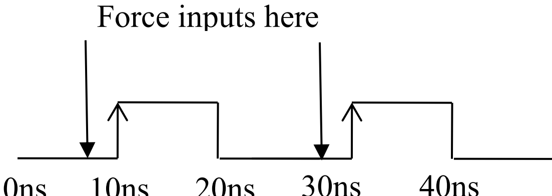
Synthesize the top module (which includes the divider and the 1-digit bcd counter) and use the following pin assignments. Download the design onto the board and make sure it works as expected.

LOAD	BTN0
D	SW[3:0]
ENABLE	SW4
UP	SW5
CLK	B8
COUNT	LED[3:0]
CO	LED4
CLR	SW6

Useful Information

1. Don't limit your testing to the input sequences mentioned with the problem statement. During the checkouts, the TAs will apply several input combinations to test your design. So, make sure to do a thorough testing of your design using sufficient number of inputs.

2. While simulating your design, it is always a good idea to stagger your inputs with respect to the active clock edge. For example, if your active clock edge is occurring at 10ns, apply your inputs sometime before 10ns, say at 8ns. This ensures that when your design was clocked, the input was successfully read. If your active edge occurs at 10ns and your input also changes at 10ns, then it becomes hard to see whether the input was successfully captured by the clock edge or not.



3. A state machine can be designed using either a single always statement (like Figure 2.56 in the text) or using two always statements (like Figure 2.54 in the text). Both ways are correct. However, it is easier to design it using a single always statement. Generally, the single always statement partakes less debugging effort. This is good guideline to observe during the entire semester.

Submission Details

All parts of this lab are to be submitted on Canvas. No hard-copy submission is needed. Please zip all your files into a single folder with the following naming scheme: ***Lastname_Lab#.zip***

Problem	Submission Requirements
1	• Verilog file(s) • Do-file
2	• Verilog file(s) • Do-file
3	• Verilog file(s) • Bit-file and UCF File

Checkout Details

During your checkout you will be expected to demonstrate each of the problems in the assignment and answer verbal questions about the assignment.

Example 1

```
module simpleDivider(clk50Mhz, slowClk);
  input clk50Mhz;   //fast clock
  output slowClk;   //slow clock

  reg[26:0] counter;
  assign slowClk= counter[26];      //(2^26 / 50E6) = 1.34seconds

  initial
  begin
    counter = 0;
  end

  always @ (posedge clk50Mhz)
  begin
    counter <= counter + 1;     //increment the counter every 20ns (1/50 Mhz) cycle.
  end

endmodule
```

Example 2

```
module complexDivider(clk50Mhz, slowClk);
  input clk50Mhz;   //fast clock
  output slowClk;    //slow clock

  reg[26:0] counter;

  initial
  begin
    counter = 0;
  end

  always @ (posedge clk50Mhz)
  begin
    if(counter == 25000000) begin
        counter <= 1;
        slowClk <= ~slowClk;
    end
    else begin
         counter <= counter + 1;
    end
  end

endmodule
```

Lab Assignment #3

Guideline

This lab can be done with a partner. In fact, partnership is encouraged.

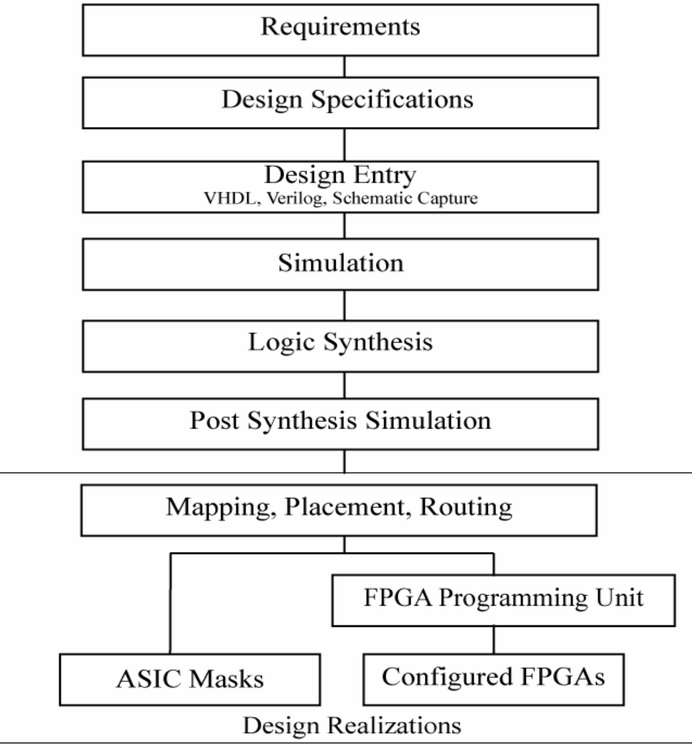
Objective

1. Understanding the ASIC/FPGA design flow
2. More digital design - sequential and combinational circuits.
3. Learn writing and using testbenches in Verilog
4. Implementing circuits on FPGA

Problem 1: ASIC/FPGA Design Flow

The following figure shows the design flow as described in chapter 2 of the text. Annotate each box in this figure with the answers to the following questions:

a. What is the function of each box (answer in one line)?
b. Which tool do you use in the lab to perform this step? If a step is not performed in the lab, mark it.
c. What inputs are needed at each stage and what outputs are delivered at each stage?



Problem 2: Package Sorter (simulation only – using a testbench)

Design a package sorter to classify packages based on their weights and to keep track of packages of different categories. The sorter has an active high asynchronous reset and will keep track of packages since the last reset. Packages should be classified into 6 groups:

Spring 2014: Use configuration 2

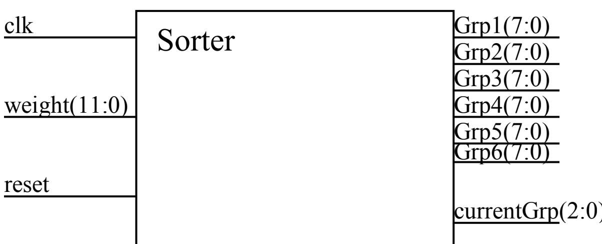
Configuration #1

i) between 1 and 200 grams
ii) between 201 and 500 grams
iii) between 501 and 800 grams
iv) between 801 and 1000 grams
v) between 1000 and 2000 grams
vi) greater than 2000

Configuration #2

i) between 1 and 250 grams
ii) between 251 and 500 grams
iii) between 501 and 750 grams
iv) between 751 and 1500 grams
v) between 1501 and 2000 grams
vi) greater than 2000

You need to decode weight measurements and classify them into various groups. The input to the circuit will be a 12-bit *unsigned* binary number (indicating the weight of the package), a clock signal, and a reset. One of the outputs will be *currentGrp*, a 3-bit unsigned number representing the current group number. There will also be six 8-bit *unsigned* outputs *Grp1-Grp6* representing the number of items weighed in each category since the last reset. The reset line is provided as input to allow these counts to be cleared.



The output lines have the following functionality:

currentGrp[2:0]: Outputs the group number for the weight currently being applied to the sorter. When a weight of zero is applied, it should output a zero. This should update as soon as a package weight changes and may not necessarily reflect the last group that a package was assigned to.

Grp1-Grp6[7:0]: Outputs the number of objects that have been weighed in each group since the last reset. These outputs should be zero when reset='1'.

Notice that the functionality of the two outputs is such that the description of *currentGrp* will be purely combinational since it does not depend on any previous inputs. But the description of *Grp1-Grp6* will be sequential since it depends not just on the current input but also on the previous inputs.

Any sequential output should change on the falling edge of the clock. Notice that the *clk* signal will be significantly faster than the duration of the weight signal. As such, you must ensure that the count is only updated once for a given input weight. Secondly, new objects can only be detected and sorted if the weight is allowed to go to zero. This is to ensure that any fluctuations in the weight after it has been sampled are not considered new items. Only the first weight after 0 updates a group count.

Test your design by using **a Verilog testbench** similar to Fig. 2-68 in the text. The testbench should use arrays (to set the inputs, to store the expected group counts and currentGrp values). Do not just use the example input. It is for illustrating the desired functionality. You are responsible for adequately testing your design, so make sure you test everything described for this problem.

Example input sequence

For Configuration 1

Reset → Put 250grams on → Take off → Put on 300 grams → Take off → Put on 501grams → Put 512 grams more
[In your waveforms, this input sequence will look like this: reset -> 250 -> 0 -> 300 -> 0 -> 501 -> 1013]

At the end of this sequence, the outputs should be:

grp1 = grp4 = grp5 = 0x00
grp2 = 0x02
grp3 = 0x01
currentGrp = 0x5

Note that after 501 grams is sampled in grp3, adding 512 grams only updates the current group and not the grp5 count.

For Configuration 2

Reset → Put 270grams on → Take off → Put on 300 grams → Take off → Put on 501grams → Put 512 grams more
[In your waveforms, this input sequence will look like this: reset -> 270 -> 0 -> 300 -> 0 -> 501 -> 1013]

At the end of this sequence, the outputs should be:

grp1 = grp4 = grp5 = 0x00
grp2 = 0x02
grp3 = 0x01
currentGrp = 0x04

Note that after 501 grams is sampled in grp3, adding 512 grams only updates the current group and not the grp4 count.

Problem 3: Traffic Light Controller (implementation – on an FPGA)

Design a traffic light controller for an intersection with a main street, a side street, and a pedestrian crossing.

Traffic light A consists of three lights: Green (Ga), Yellow (Ya), and Red (Ra).
Similarly, traffic light B consists of three lights: Green (Gb), Yellow (Yb), and Red (Rb).
Lastly, the walk indicator consists of two lights: Green (Gw) and Red (Rw).

The normal sequence of operation is as follows: Ga Rb Rw, Ya Rb Rw, Ra Gb Rw, Ra Yb Rw, Ra Rb Gw, Ra Rb Rw, Ga Rb Rw… (repeat). The timings are as follows:

Spring 2014 : Use configuration 2

Configuration 1

Main (A) Street:

- Green: lasts 4 seconds.
- Yellow: lasts 2 seconds.
- Red: lasts 10 seconds.

Side (B) Street:

- Green: lasts 3 seconds.
- Yellow: lasts 1 seconds.
- Red: lasts 12 seconds.

Pedestrian Crossing:

- Green: lasts 2 second.
- Red: Flashes 4 seconds at 1Hz, then solid for 10 seconds

Maintenance mode:

- RST=1: Ra, Rb, and Rw all flash at 1Hz
- RST=0: Traffic lights should resume operation with Ga,Rb,Rw as initial state

Configuration 2

Main (A) Street:

- Green: lasts 3 seconds.
- Yellow: lasts 2 seconds.
- Red: lasts 8 seconds.

Side (B) Street:

- Green: lasts 3 seconds.
- Yellow: lasts 1 seconds.
- Red: lasts 9 seconds.

Pedestrian Crossing:

- Green: lasts 2 second.
- Red: Flashes 2 seconds at 2Hz, then solid for 9 seconds

Maintenance mode:

- RST=1: Ra, Rb, and Rw all flash at 1Hz
- RST=0: Traffic lights should resume operation with Ga,Rb,Rw as initial state

The above mentioned delays can be obtained through the use of counters, just like you divided 50MHz clock to generate a 1Hz (1 sec period) in Lab#2.

Your design steps are listed below:

1. Start by designing a state graph for the controller. You do not need to derive any equations, since you can model the state graph using behavioral Verilog code. Note that on designing your state graph, you will transition from one state to the other when the appropriate time has elapsed.

2. Write behavioral Verilog code that represents your state graph. For purposes of checking the functionality of your code, reduce the counter time to a small number during simulation, otherwise you may have to simulate your code through several simulation pages.

3. Once your code simulates properly, proceed to synthesizing it and implementing it on the FPGA. Read through the FAQs at the beginning of this manual to understand and clarify doubts about how to use always statement to make combinational logic and sequential logic, and how to avoid latches. The following table gives the IO connections for implementing the traffic light controller:

Green Light street A:	LED2
Yellow Light street A:	LED1
Red Light street A:	LED0
Green Light street B:	LED7
Yellow Light street B:	LED6
Red Light street B:	LED5
Green Light Ped Xing:	LED3
Red Light Ped Xing:	LED4
Rst (Maintenance mode)	SW0

There is something else that you need to do as well for this part of the lab. You need to generate three reports while implementing your design:

1. The synthesis report – to find out the digital elements used by your design
2. The Place and Route report – to find out the number of slices of the FPGA used by your design
3. The Static Timing report – to find out the critical path in your design

To view these reports, go to the "Design Summary" tab in the Xilinx ISE window. The synthesis report can be seen by double-clicking "Synthesis Report" under "Detailed Reports". You can locate the digital elements (like gates, flops and latches) used by your design in this report. Of course, to be able to see this report, you should have synthesized your design. Ensure that there are no latches in your design. In the synthesis report, look for "Macro Statistics" and see if any latches are being shown. Alternatively, you can look for "cell usage" in the report and there should not be any cells under "Flip Flops/Latches" having names starting with "L".

The Place and Route Report will be located in the "Design Summary" tab under Detailed Reports -> Place and Route Report. The Static Timing Report should be in the same tab under "Detailed Reports" -> "Static Timing Report". For you to be able to see these reports, you should have run the "Implement Design" step. If the Post-PAR Static Timing Report is not generated, expand the "Implement Design" entry under the "Processes" window. Then expand the "Place & Route"

entry and run the "Generate Post-Place & Route Static Timing" process. In the place and route report, circle or otherwise note the number of slices used by your design, and in the static timing report, circle or otherwise note the critical delay of your design. Please note that Xilinx Timing Reports are sorted in 3-4 groups.

i. Path from input port to register, reported in Setup/Hold to clock CLK group.
ii. Path from register to output port, reported in Clock CLK to Pad group.
iii. Path from register to register, reported in Clock to setup on destination clock CLK group.
iv. Path from input port to output port, reported in Pad to Pad group.

For finding the critical delay in your design you need to look for the longest delay within each of these four groups.

Useful Information

1. For the reports (STA, PAR and SYNTH), just copy the entire reports into text/doc files and highlight the parts which contain relevant information (like cell count, path delays, etc)
2. Sometimes Xilinx ISE does not show all the paths in the timing report. It will show just one or two paths. This is okay. Probably there is a bug in the tool. Just submit whatever you got.

Submission Details

All parts of this lab are to be submitted on Canvas. No hard-copy submission is needed.

- Problem 1
  - Text file/Word document containing the answers
- Problem 2
  - Typed Verilog Code (.v file)
  - Typed Testbench Code (.v file)
- Problem 3
  - Typed Verilog Code (.v file)
  - Synthesis report (.txt or .doc file)
  - Place and Route Report with number of slices noted (.txt or .doc file)
  - Post-Place and Route Static Timing Report with critical delay noted (.txt or .doc file)
  - [filename].bit file from compilation
  - UCF file

Checkout Details

During your checkout you will be expected to demonstrate each of the problems (both simulation and implementation if required for the problem) in the assignment and answer verbal questions about the assignment.

Lab Assignment #4

Guideline

This lab can be done with a partner.

Objective

Your objective in this lab is to design (code, simulate and implement) a parking meter much like the ones around Austin. It should be able to simulate coins being added and show the appropriate time remaining. Also, it should flash slowly when less than 200 seconds are remaining and flash quickly when time has expired.

Description

You will design a finite state machine that will simulate the operation of a traffic meter. The buttons on the board will represent different coin denominations and the seven segment LED display will output the total amount of seconds remaining before the meter expires.

Spring 2014 : Use configuration 2

Configuration 1

Button 0	Add 30 seconds
Button 1	Add 120 seconds
Button 2	Add 180 seconds
Button 3	Add 300 seconds
Switch 0	Reset time to 15 seconds
Switch 1	Reset time to 185 seconds

As soon as a button is pushed, the time should be added immediately. When less than 180 seconds remain, the display should flash with period 2 seconds and duty cycle 50% (on for 1 sec and off for 1 sec; so you will see alternate counts on the display eg- 185,blank,183,blank,181…). When time has expired, the display should flash with period 1 sec and duty cycle 50% (on for 0.5 sec and off for 0.5 sec). For example, when the board starts, it should be in the 0 time remaining state and be flashing 0000 at a 0.5 second rate. If button 3 is then pushed, the display should read 300 seconds and begin counting down. When the time counts down to 200 seconds and button 2 is pushed, the display should then read 380 seconds (200 + 180). If switch 0 goes high, then the time should change to 15 seconds and flash accordingly.

Configuration 2

Button 0	Add 50 seconds
Button 1	Add 150 seconds
Button 2	Add 200 seconds
Button 3	Add 500 seconds
Switch 0	Reset time to 10 seconds
Switch 1	Reset time to 205 seconds

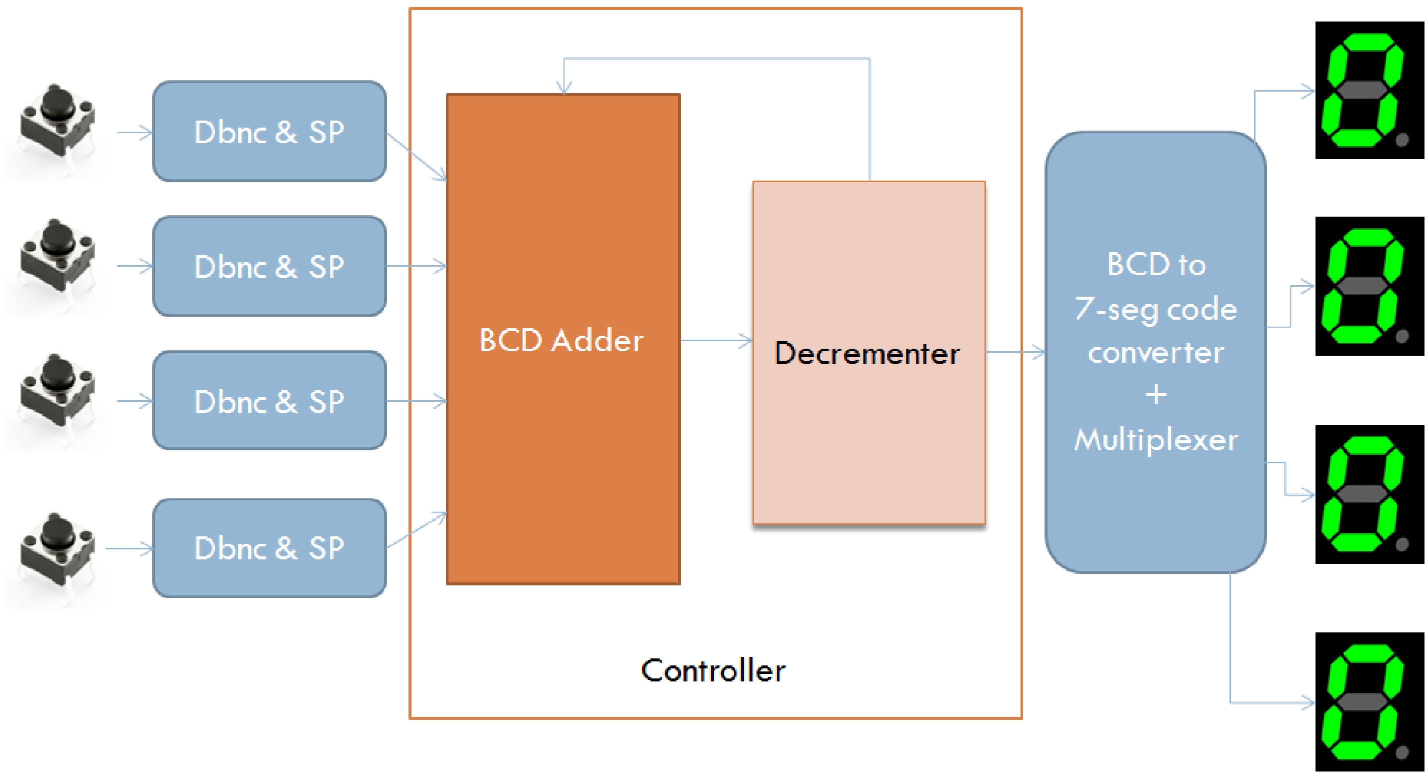
As soon as a button is pushed, the time should be added immediately. When less than 200 seconds remain, the display should flash with period 2 seconds and duty cycle 50% (on for 1 sec and off for 1 sec; so you will see alternate counts on the display like 200,blank,198,blank,196,…). Make sure you blink such that even values show up and odd values are blanked out. When time has expired, the display should flash with period 1 sec and duty cycle 50% (on for 0.5 sec and off for 0.5 sec).

For example, when the board starts, it should be in the 0 time remaining state and be flashing 0000 at a 0.5 second rate. If button 3 is then pushed, the display should read 500 seconds and begin counting down. When the time counts down to 180 seconds and button 2 is pushed, the display should then read 380 seconds (200 + 180). If switch 0 goes high, then the time should change to 10 seconds and flash accordingly.
The max value of time will be 9999 and any attempt to increment beyond 9999, should result in the counter defaulting to 9999 and counting down from there.

From a structural perspective, your circuit will consist of three parts:

- The input module, which takes the input from the buttons on the board,
- The output module, which displays the output on the 7-segment display, and
- The controller

Although it is not mandatory to follow this structural hierarchy, it is recommended that you implement the input and output parts in separate modules and make sure they are working correctly before putting the whole design together. You need to implement a de-bouncing circuit to make the input module work. The best way to this is to read, understand, and then implement the de-bouncing circuitry described in the textbook. For the output module you need to read the board manual and understand how to correctly drive the multi-digit 7-segment display. In the controller module, you will need to add the time count whenever a button is pushed and subtract the time count every second. You can design in a way such that you use all BCD operations (by having BCD addition and subtraction like the one shown in the following figure). However, you can also keep your counts in binary and then convert binary numbers to multiple digit BCD numbers before you send them to the output module. **However, please note that you cannot divide by 10 using the Spartan 3E FPGA hardware. If you use the division operator in your Verilog, it will not synthesize to anything. Therefore, you CANNOT use any binary-to-BCD conversion methods that rely on dividing by 10.**



Useful Information

1. Debouncer and Single Pulser circuitry is explained in section 4.7 of the text.
2. BCD Adder is described in section 4.2 of the text. If you don't want to use a BCD adder, you can use an approach similar to problem 4.13 (in the text) for binary to bcd conversion
3. BCD to 7-segment decoder is described in section 4.1 of the text. However, note that the polarities of signals (anodes and cathodes) are not the same as the ones in the text. Please refer to the board's manual for proper polarities.
4. Make sure you go through the Nexys board manual to understand how to multiplex the 7-segment displays.
5. Check for the overflow condition (saturation at 9999) in your code and make sure it works.
6. If you make the BCD Adder and Decrementer as a single always blocks running on 50MHz clock, your design might become easy. **However, it is up to you to make them as two separate always blocks.**
7. It is recommended that you simulate the design using either a test bench or by using the force & run commands from the transcript window.
8. In case your design does not work on the board, submit the testbench Verilog file and/or the .do file, and show the simulation during checkout for partial credit.
9. Ensure that there are no latches in your design when you implement it.

Submission Details

All parts of this lab are to be submitted on Canvas. No hard-copy submission is needed.

1. All Verilog code
2. Any testbench code or do-files that you use
3. Bit file and UCF file

Checkout Details

During checkout you will demonstrate the parking meter working on the board as well as in simulation. Also, you will be judged on how well you understand your code and other concepts like de-bouncing, multiplexing 7-segments and BCD addition.

Lab Assignment #5

Guideline
This lab can be done with a partner.

Objective
To develop a basic SNAKE game by interfacing a PS/2 Keyboard and VGA display with theboard

Reading
Before you start working on this lab, please read pages 7-12 of the Nexys2 Board User Manual.
More info on the PS/2 protocol: http://pcbheaven.com/wikipages/The_PS2_protocol/
More info on the VGA Standard: http://www.ece.msstate.edu/~reese/EE4743/lectures/displays/displays.pdf

Description
In this lab you will be required to create a simple keyboard controller and a VGA controller. The keyboard controller will enable communication from the keyboard. The VGA controller will be used to display some simple graphic patterns on the computer monitor attached to the board.

For the entire lab, keep in mind "How can I test this during early design and simulation stages?" It is recommended that you simulate the core components of your design to ensure the basic logic works correctly. After this, you can use the hardware to begin testing your design. Debugging through a relatively opaque hardware interface is difficult (e.g., trying to debug a graphics controller if the monitor doesn't display anything). Try to make your design very clear, simple, and modular. This allows you to relatively quickly diagnose problems and create potential solutions.

Submission details
Submit the following things on Canvas:

- Verilog codes (design and testbenches/dofiles) for each part
- Waveforms for part (b)
- Bit files and UCF files for each part

Checkout details
Demonstrate the each part during the checkout to the TA.

Grading:

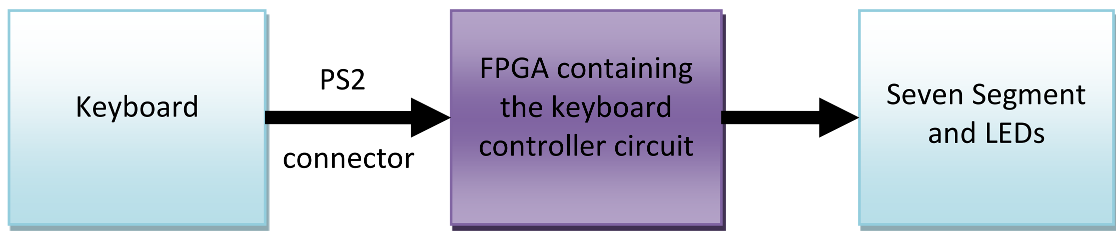
Part A = 50pts (28%)
Part B = 50pts (28%)
Part C = 71pts (39%) + 9pts (5%) = 80pts
Total: 180pts

Note: 5% of your grade will be based on creativity/uniqueness you add to your design in part C!

Part A: Keyboard interface design (50pts)

In this part of the lab, you will be designing an interface for accepting values from the keyboard. In previous labs, we have been limited to accepting inputs from the 8 switches or the 4 buttons. In this lab, we will expand the input functionality by implementing a PS2 keyboard interface. The values sent from the keyboard will be displayed on the seven segment display on the Nexys2 board.

The PS2 protocol is a simple two-wire scheme that uses serial transmission to transmit the data to the board. While the two-wire bus is bi-directional in design, we will only be using it as an input to the FPGA. [Typically writing to the keyboard is used to reset, turn on the various indicator lights, etc.]



When a key is pressed, a sequence of bytes is sent serially over the two-wire bus. Each key on the keyboard is given a unique "scancode" (see Nexys2 board user manual). In order to detect when keys are initially pressed and then released, the keyboard will send a sequence of bytes for each key press. The first byte sent by the keyboard is typically called the "make code" and it represents the key that is pressed. The final byte sent by the keyboard is the "break code" which represents the key that was released.

For example, consider the situation where a user presses the letter 'a':

1) User presses the 'a' key
2) Keyboard sends "make code" (which is '1C' for the 'a' key) serially. The keyboard keeps sending the "make code" every 100ms until the user releases the key.
3) User releases the 'a' key
4) Keyboard sends the "key up" code 'F0'serially
5) Keyboard sends the "break code" (which is '1C' for the 'a' key) serially

We will only need to look for the "break code" bytes. So we can simply monitor the bits for the "key up" scan code which indicates that the key has been released. When this byte has been sent, the "break code" for the released key will be sent.

To transmit the sequence of bytes, the keyboard first forces the *DATA* line low to create the start bit. Bits are transmitted using the falling edge of *CLK* for synchronization. This is illustrated in Figure 1. The *DATA* signal changes state when the *CLK* signal is high, and *DATA* is valid for reading on the falling edge of *CLK*.

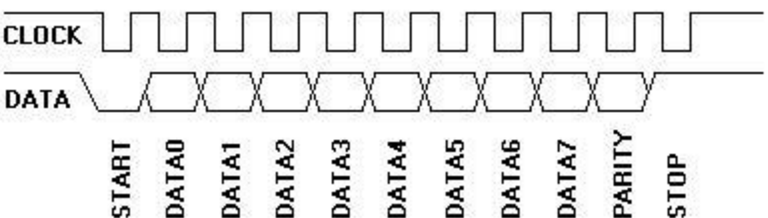


Figure 1. Device to Host Communication

So the basic operation of your design is as follows:

- On the falling edge of *CLK*, use a shift register to capture each bit of data
- When all 11 bits have been sent (start, scancode, parity, stop), you can look at the scancode and decide what to do
- If the scancode is a "key up" i.e. 'F0', you know that the next data sequence sent by the keyboard is the final scancode that you need.
- Capture the final scancode by following the same steps as above and output this value from your keyboard controller.

Display the lower 2 hex digits of the scancode received by the controller on the lower two seven segment displays (Note that some keys scan codes have 2 digits and some have 4 digits, see Fig. 14 on page 9 of the Board User Manual). You should also have a strobe signal to indicate that the keyboard controller is outputting a new keypress. A strobe signal is a short pulse on one of the board LEDs.

Useful Information

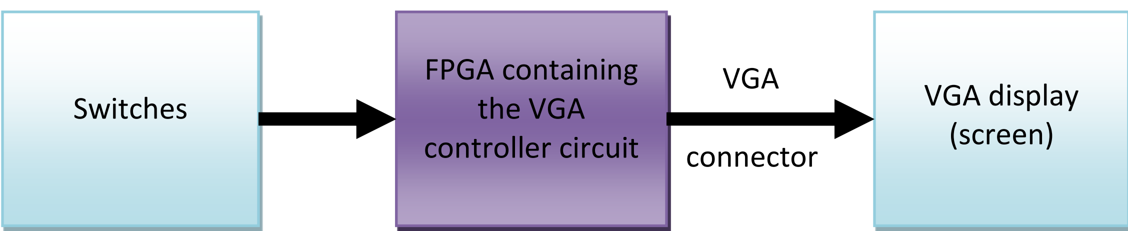
1. Some keyboards in the LAB have a problem with the SPACE bar (probably the ones from HP). So, don't panic if your code breaks when you hit the SPACE bar. Just change the keyboard, and everything will be fine.
2. The 7-segment display should show the keycode of a key until a new key is pressed, at which time it starts to show the keycode of the new key.
3. The strobe signal's duration can be as much as you wish unless it is visible to us with naked eye.
4. There is an easy way of implementing the keyboard interface by using a 22-bit shift register in your design. Think about it! Talk to the TAs about it.
5. Some keys on the keyboard (like the arrow keys) are special in the sense that they send an additional code "E0" ahead of the scan code. Such keys are called extended keys. When an extended key is released and "E0 F0" code is sent followed by the scan code. So, irrespective of the type of key, the last two chunks of data when a key is released will be "F0" and the scan code.
6. The Nexys2 Board Manual shows the keycode for the key "z" as "1Z". This is a typo, the actual code is "1A".
7. For this design you should use the keyboard clock as an input to your module. Disregard what the Nexys2 Board Manual says.
8. Although the keyboard data signal is birectional, we will only be using it as an input for this lab.
9. In case you are getting an error related to the keyboard clock during the Place & Route step in Xilinx ISE (which says something like "Clock IOB/ clock component is not..."), please add the following line to your UCF file and then re-run the Place & Route step again.

   ```
   NET "KCLK" CLOCK_DEDICATED_ROUTE = FALSE;
   ```

   where KCLK is the name of keyboard clock signal in your design. To edit your UCF file, click on the UCF file name in the 'Sources' part of the window in Xilinx ISE. Now expand the "User constraints" in the "Processes" part of the window. Now double-click on "Edit Constraints". Now the UCF file will be loaded in the right side of the window. Edit the file as a normal text file and hit "Save".

Part B: VGA Interface Design (50pts)

In this part of the lab you will design a VGA controller to output graphics to the computer monitor connected to the Nexys2 board. In previous labs, we were limited to either the seven-segment display or the LEDs. In this lab, we expand on this functionality to allow graphical images to be displayed from the FPGA board.



A VGA monitor operates using an electron beam that scans the screen row by row, starting at the upper left corner and ending at the lower-right corner. This beam moves using two synchronization signals, called *hsync* (horizontal synchronization), and *vsync* (vertical synchronization). The *hsync* signal tells the beam when to move to the next row. The *vsync* signal tells the beam when to move to back to the top of the screen. To display a picture on the screen, we simply generate these synchronization signals and provide the pixel color to display on the screen.

In this lab, you are required to create a 640 pixel x 480 pixel screen display. A pixel clock operating at 25 MHz will be used. To get 640 pixels horizontally, a horizontal synchronization frequency of approximately 31.5 KHz is required. This corresponds to approximately 800 clock periods of the pixel clock. During the first 640 clock periods of the pixel clock for a row, visible pixels can be displayed; however, the last 160 clock periods for the row are called the "retrace period" or "blanking region" where nothing is displayed while the beam retraces back to the left of the screen. To get 480 pixels vertically, a vertical synchronization frequency of 60 Hz is required. This corresponds to approximately (800x)525 clock periods of the pixel clock. This can be thought of as generating 480 visible rows followed by 45 blank rows during which the beam retraces back to the top of the screen. This is illustrated in Fig. 2.

Figure 3 shows the timing of the *hsync* signal. It is made low starting on the 659th pixel clock period for the row and made high again on the 755th pixel clock period. During the first 640 pixel clock periods, visible pixels are generated. During the last 160 pixel clock periods, nothing is generated on the screen.

Figure 4 shows the relationship between the *vsync* signal and the *hsync* signal. The digits represent the line count. As stated earlier, the first 480 lines are displayed while the last 45 lines correspond to the retrace period for the beam to get back to the top left corner of the screen.

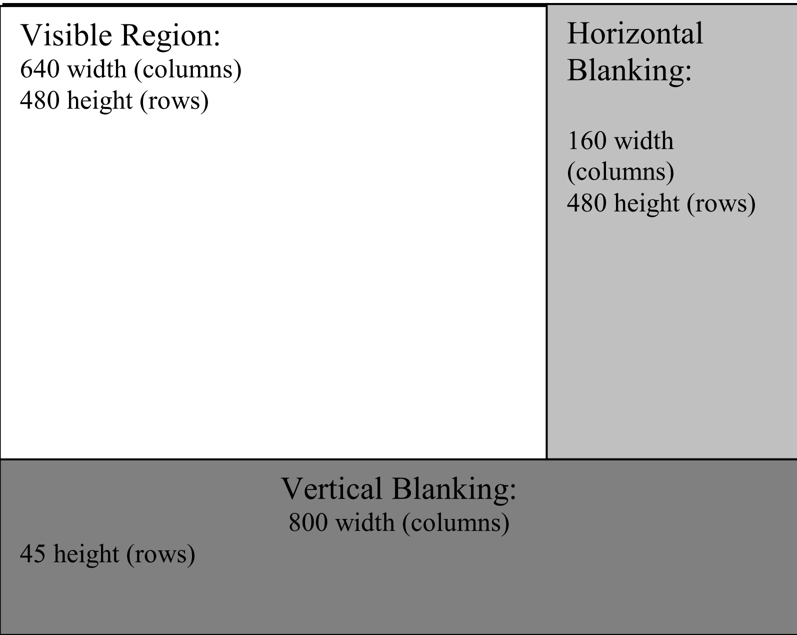


Figure 2. Display Regions

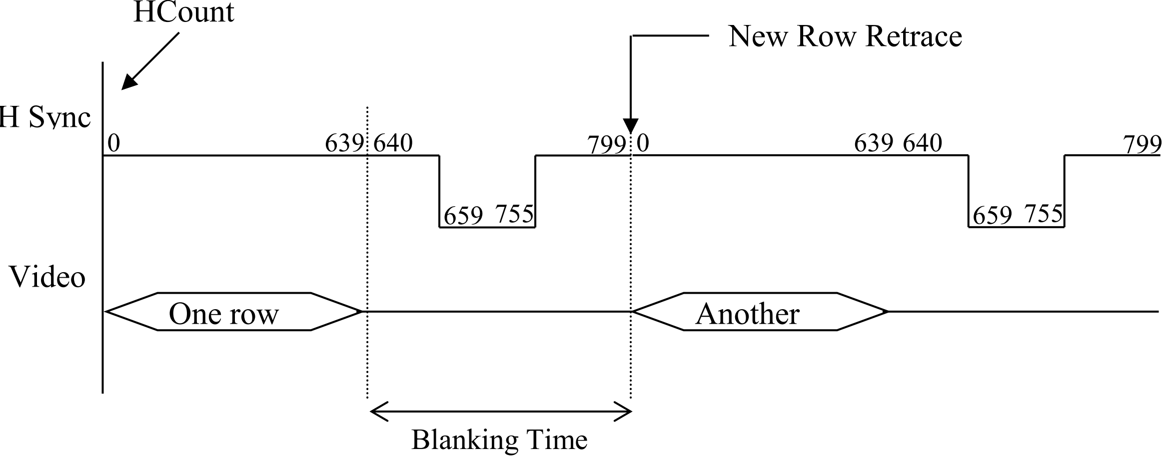


Figure 3. Horizontal Sync Timing

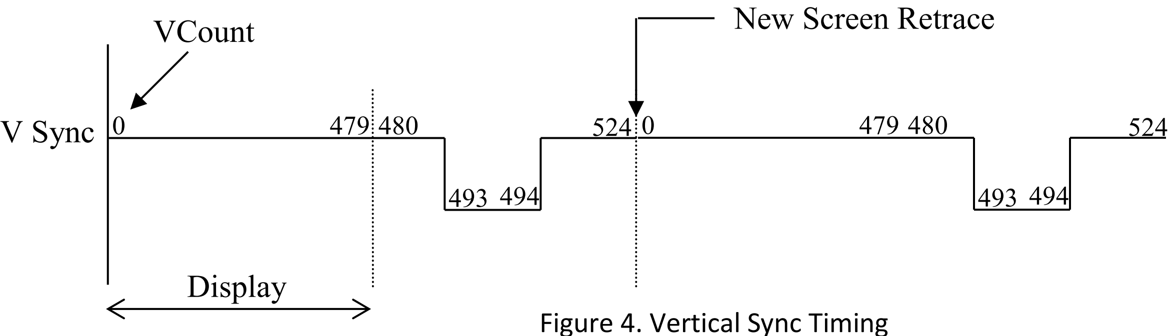


Figure 4. Vertical Sync Timing

So the basic operation of the design you implement is as follows:

- Use 2 counters to store the values of hcount and vcount
- Generate a 25MHz pixel clock by dividing the 50MHz clock
- On the rising edge of the pixel clock, increment the hcount. Increment vcount when hcount has reached the end of the row.
- Generate the hsync signal based on the value of hcount as illustrated in Fig. 3. vsync is generated in a similar fashion as illustrated in Fig. 4.
- Generate a signal to determine whether the pixel is in the visible region as illustrated in Figure 2.
- When in the visible region, output the pixel color value {R, G, B}, otherwise when in the blanking region, output {0, 0, 0}.
- Finally, put all of the outputs {R, G, B, hsync, vsync} thru flip-flops to ensure no combinational logic delays will interfere with the output display

The VGA controller that you design in this part of the lab will take as inputs the 25 MHz pixel clock and the pixel_color to display each clock cycle. The 25Mhz will be generated from the 50Mhz clock and the pixel_color will come from which switch is ON. The following table shows the color of the screen for each switch. The VGA controller will generate as an output the hsync, vsync, R, G, and B signals. It will also output the current horizontal coordinate (0-799) and vertical coordinate (0-524). The screen will display the color depending on which switch is ON (complete screen filled with one color).

Switch	**Color on the VGA display**
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Yellow
7	White
None	Black

(Don't consider the cases when more than one switch is ON.)

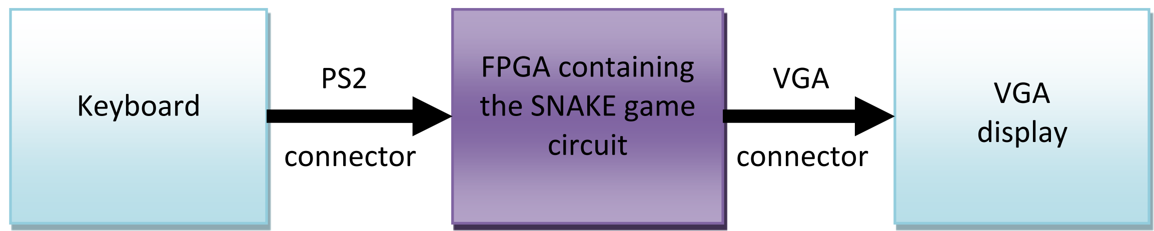
Important: It is mandatory to simulate this part before synthesizing and downloading to the FPGA. You can either use a testbench or the commands (like force, etc) directly.

Useful Information

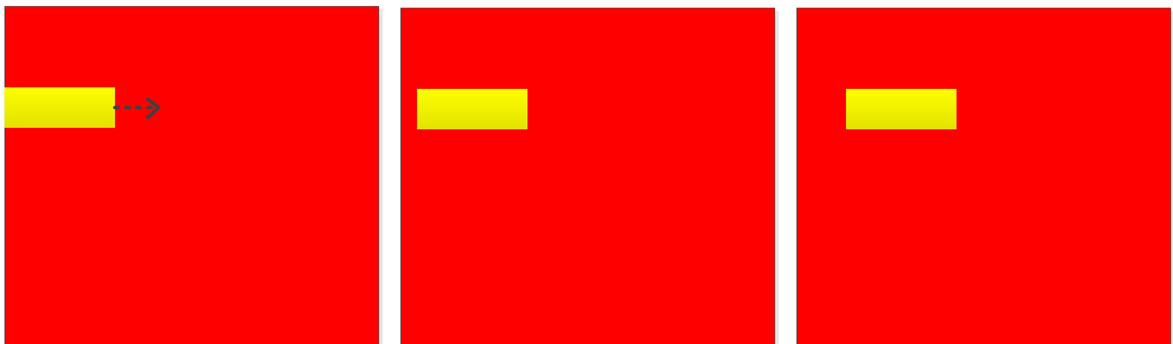
1. You must follow the VGA protocol exactly as mentioned in this document. Do not change the numbers for generating hsync and vsync.
2. To generate different colors, you can refer to the 8-bit VGA color codes where each R, G and B are encoded in 8 bits. http://cloford.com/resources/colours/namedcol.htm You can develop color codes (total RGB = 8 bits, which is what you need for this lab) using that.

Part C: Snake game (80pts)

In this part of the design, you will implement the master controller which receives input from the keyboard controller and uses the VGA controller to output the appropriate pixels to the monitor. You will implement a simple snake game in this part.



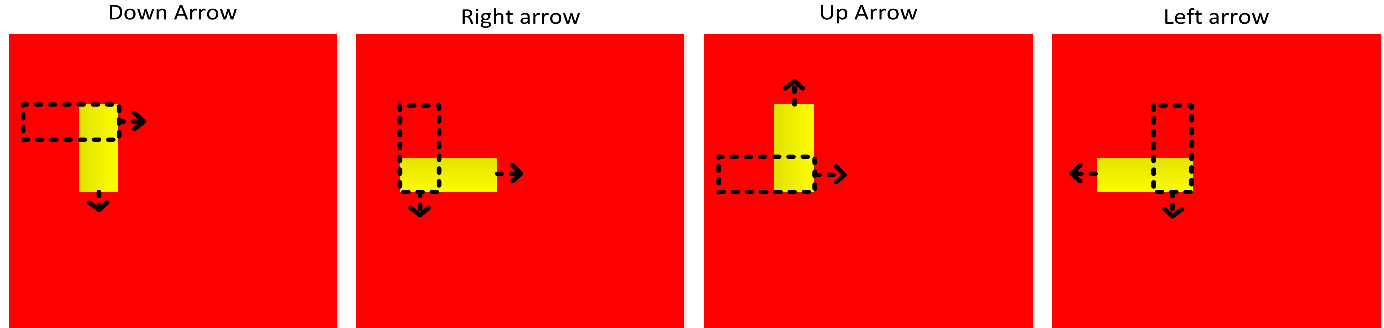
The screen is blank in the beginning. Pressing 'S' on the keyboard starts the snake game- a 'snake' graphic at the left edge of the screen that automatically starts scrolling right as shown in the figure below.



This scrolling graphic will respond to arrow key pushes in the following way:

Original Orientation	Change	
Horizontal	Up arrow	Flip vertical and scroll up
	Down arrow	Flip vertical and scroll down
	Left arrow	No change
	Right arrow	No change
Vertical	Up arrow	No change
	Down arrow	No change
	Left arrow	Flip horizontal and scroll left
	Right arrow	Flip horizontal and scroll right

The following figures show the change in the graphic due to a few arrow key pushes:



Note that the snake turns from its front head instead of the tail. Pressing the button 'P' on the keyboard pauses the game (freezes the screen) and pressing 'R' resumes the game from its paused state. Pressing 'ESC' exits from the game (blanks out the screen).

Other game properties:

- The width of the snake, the color of the snake, the background color of the screen and the scrolling speed of the snake that you need to keep are given at the end of this lab description. Note that the snake should scroll smoothly. The snake should not jump 50 pixels every second.
- On a game over, make sure that only the required part of the snake is visible. For example, if the snake is moving towards the right direction very close to the top edge, and the user presses an up-arrow key, the game will end and only a part of the snake should be visible.
- When the snake touches any edge of the screen, the entire screen should freeze and no longer respond to arrow pushes or R/P presses. But it should still respond to 'ESC' and 'S' presses.
- Pressing 'R' in the unpaused state does nothing. Pressing 'P' in the paused state does nothing. Pressing 'ESC' **anytime** exits the game (blanks out the screen) and pressing 'S' **anytime** starts the game.

A block diagram for the complete design is shown in Figure 5.

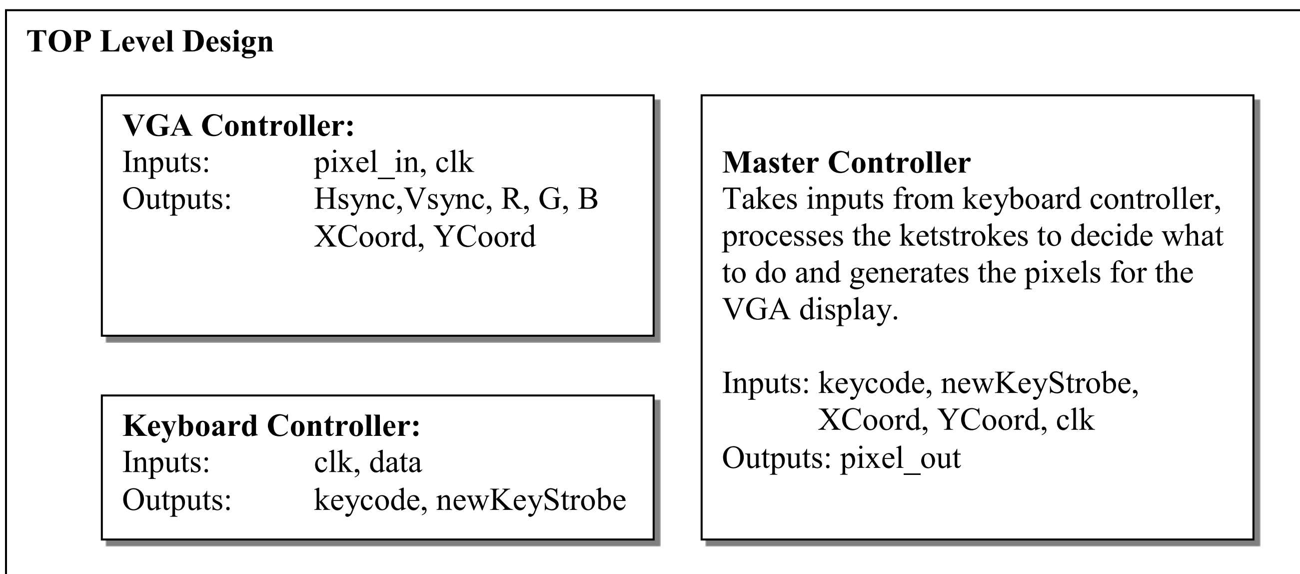


Figure 5. Block Diagram for Complete Design

Parameters of the Snake Game

Background color of the screen	White
Color of the snake	Blue
Length and width of the snake	40 x 10
Speed of the snake	50pixels per second

Creativity Factor (5%)

You will need to make some modification to your snake game! You have freedom to modify any parameters as long as the original functionality of your game remains intact. Your grade will be based the "cool" feature(s) added to your game!

Lab Assignment #6

Guideline

This lab can be done with a partner.

Objectives

- To implement a stack calculator.
- To get more familiar with block RAMs on an FPGA and understand memory interfacing
- To understand how to model buses in Verilog.

Description

In this lab, you will write Verilog code to implement a stack calculator using a memory module (block RAM on the FPGA) and the board I/O.

A stack calculator stores its operands in a stack structure and performs operations (e.g. addition, subtraction, etc) on the top two values of the stack. The operands are popped off the stack and the result is pushed back on the top of the stack, so the stack is one element less than it was before the operation. The output of the calculator is always the value at the top of the stack. The stack may contain more than two operands at any time, but operations are only performed on the top two values. This is similar to the Reverse Polish Notation (RPN) used by old TI calculators.

You will implement a simple stack calculator using Xilinx BlockRAM as the storage for the stack. We will provide you with code to implement/model a memory using the BlockRAM. The memory supplied is byte-addressable and 8 bits (1 byte) wide. Please check the synthesis report to make sure that Xilinx ISE is synthesizing your design using a BlockRAM and not distributed LUT-RAM. If it is not BlockRAM, then you need to change it to BlockRAM under synthesis properties. Right-click "Synthesize-XST" and click on "Properties". In the dialog box that appears, click "HDL Options". Select "Block RAM" from the drop-down menu for the "RAM Style" option.

We will also provide you the code for top block which integrates the controller, memory along with the data bus. You will need to modify the supplied code to implement the bus interface to memory and the controller. This will involve using tri-state buffers. Through tri-state buffers, we will be able to ensure that only one driver drives the data bus at a time.

You will also need to implement a master controller for the calculator. This controller contains three registers, a stack pointer (SPR), a display address register (DAR), and a display value register (DVR). The SPR will contain the address of the next free address past the top of the stack. The DAR will hold the address of the data that should be displayed on the output. Whenever the SPR is updated, the DAR should be updated to SPR plus 1. The DVR contains the value that should be displayed on the output. The content of the DVR is the value stored at the memory location pointed to by the DAR. This should be updated every time the DAR changes by reading from the memory location contained in the DAR. The memory will be 128 bytes total giving a 7-bit address. So the SPR and DAR will both be 7 bits wide and the DVR will be 8 bits wide. The master controller will be responsible for taking in the inputs, updating its registers accordingly, as well as performing the operations of the calculator and displaying the outputs. Before use, a reset/clear operation should be used to initialize the calculator. The SPR should be initialized to 0x7F, the DAR to 0x00, and the DVR to 0x00 (don't read from memory this time). As data is pushed on to the stack, the SPR will decrement. In other words, the stack will grow towards decreasing addresses.

The calculator will use all of the inputs/outputs on the Xilinx board. The seven segment displays will show the contents of the DVR (only two of them will be used because the data size is 8 bits). Values will be entered, 8 bits at a time, using the switches on the board (SW0 maps to the LSB). LED[6:0] show the contents of the DAR bits 6 down to 0 and LED[7] will map to an 'EMPTY' flag. If the stack is empty (the SPR contains 0x7F), the EMPTY flag (LED[7]) will be set to '1'.

The buttons will provide the operational inputs to the controller. Each button will implement a function as defined in the table below:

Mode	BTN3 BTN2	BTN1	BTN0
Push/Pop	0 0	Delete/Pop	Enter/Push
Add/Subtract	0 1	Subtract	Add
Clear/Top	1 0	Clear/RST	Top
Dec/Inc	1 1	Dec Addr	Inc Addr

- **Enter/Push**: Reads the value from switches on the board and pushes it on the top of the stack. To do this, keep BTN3 and BTN2 at 0 (ie. unpressed) and press BTN0
- **Delete/Pop**: Pops and discards the 8-bit value on the top of the stack. To do this, keep BTN3 and BTN2 at 0 (ie. unpressed) and press BTN1
- **Add**: Pops the top two 8-bit values on the stack, adds them, and pushes the 8-bit result on the top of the stack, discarding the carry bit. To do this, keep BTN3 at 0 (unpressed) and BTN2 at 1 (pressed) and press BTN0
- **Subtract**: Pops the top two 8-bit values on the stack, subtracts them, and pushes the 8-bit result on the top of the stack, discarding the borrow bit (High Addr minus Low Addr). To do this, keep BTN3 at 0 (unpressed) and BTN2 at 1 (pressed) and press BTN1
- **Clear/RST**: Resets the SPR to 0x7F, the DAR to 0x00, and the DVR to 0x00. The stack should be empty now (EMPTY flag should be set to 1). To do this, keep BTN3 at 1 (pressed) and BTN2 at 0 (unpressed) and press BTN1
- **Top**: Sets the DAR to the top of the stack (SPR+1; will cause the DVR to update). To do this, keep BTN3 at 1 (pressed) and BTN2 at 0 (unpressed) and press BTN0
- **Dec Addr**: Decrements the DAR by 1. To do this, keep BTN3 1 (pressed) and BTN2 at 1 (pressed) and press BTN1
- **Inc Addr**: Increments the DAR by 1. To do this, keep BTN3 1 (pressed) and BTN2 at 1 (pressed) and press BTN0

The following figure shows the block diagram of your design:

Top-Level Design

Stack Calculator Controller

Address

Memory

Data Bus

Useful Information

1. Note that the overflow, underflow, and pushing both BTN0 and BTN1 at the same time are not considered in this lab. In general you can assume the calculator will be used as described, i.e. you do not have to worry about the error conditions like POPing without having pushed anything, decrementing DAR beyond the lowest address. Keep it simple.
2. All data on the stack should be considered unsigned.
3. Also note that the INC and DEC commands affect the DAR and DVR. They don't modify SPR. INC and DEC are just to be able to see the contents of various locations on the stack. Similarly, the POP/PUSH/ADD/SUBTRACT will use the SPR (however, they will update the DAR and DVR as well).
4. Simulation is NOT a requirement to get full credit if your design works perfectly on the board. If it does not work on the board, please have simulation ready for partial credit.
5. The simplest way to approach the design of the controller is to use a large state machine. The first state will be state will waits for inputs from the user. You will jump from this state to others depending on the inputs.
6. The memory works on 50MHz, but has single cycle latency. This means that when reading from the memory, if you make WE 0 in one clock cycle (in other words, in one state of the controller), you should read data from the data bus in the next clock cycle (in other words, in the next state of the controller). Similarly, when writing to the memory, you should make WE 1 in one clock cycle and wait for one clock cycle to let the memory write the data.
7. The controller can use as many cycles (states) as it wishes to perform the tasks (like POP, PUSH, ADD, etc). Since the clock frequency is 50MHz, even if the controller takes 10 cycles (say) to perform an operation, the user won't be able to see the lag with the naked eye.
8. If you need, you can modify the ports of the modules given to you. But we would want you to not change the memory module at all.

Submission Details

Submit the following files through Canvas. No hard copy submission is required.

- Typed Verilog Code (.v files)
- [filename].bit file
- [filename].ucf file
- Synthesis report showing that your final design does not contain any latches and that block RAM has been used in the design

Checkout Details

Demonstrate the calculator working on the board to the TA during checkout.

Example

Suppose you want to add 0x92 (binary form: 10010010) and 0x25 (binary form: 00100101). You should first push these two numbers on the top of the stack (the stack at this time can contain other numbers). Perform the following sequence to enter 0x92 and then 0x25 into the stack:

0. Reset the calculator, hold BTN3 and push BTN1. At this point, LED[7:0]=1_0000000 and 7-seg are 00
1. Set the switches (SW7 downto SW0) to 10010010.
2. Push BTN0. At this point, LED[7:0] are 0_1111111 and the 7-segs are 92.
3. Set the switches (SW7 downto SW0) to 00100101.
4. Push BTN0. At this point, LED[7:0] are 0_1111110 and the 7-segs are 25.
5. Hold BTN2 and push BTN0. At this point, LED[7:0] are 0_1111111 and the 7-segs are B7.
6. After steps 2 or 4, you can pop (delete) the numbers you have entered by pressing BTN1.

Starter Code

Top Module

```
module top(clk, btns, swtchs, leds, segs, an);
  input clk;
  input[3:0] btns;
  input[7:0] swtchs;
  output[7:0] leds;
  output[6:0] segs;
  output[3:0] an;

  //might need to change some of these from wires to regs
  wire cs;
  wire we;
  wire[6:0] addr;
  wire[7:0] data_out_mem;
  wire[7:0] data_out_ctrl;
  wire[7:0] data_bus;

  //CHANGE THESE TWO LINES
  assign data_bus = 1; // 1st driver of the data bus -- tri state switches
                       // function of we and data_out_ctrl

  assign data_bus = 1; // 2nd driver of the data bus -- tri state switches
                       // function of we and data_out_mem


  controller ctrl(clk, cs, we, addr, data_bus, data_out_ctrl,
      btns, swtchs, leds, segs, an);

  memory mem(clk, cs, we, addr, data_bus, data_out_mem);

  //add any other functions you need
  //(e.g. debouncing, multiplexing, clock-division, etc)

endmodule
```

Controller

```
module controller(clk, cs, we, address, data_in, data_out, btns, swtchs, leds, segs, an);
  input clk;
  output cs;
  output we;
  output[6:0] address;
  input[7:0] data_in;
  output[7:0] data_out;
  input[3:0] btns;
  input[7:0] swtchs;
  output[7:0] leds;
  output[6:0] segs;
  output[3:0] an;


  //WRITE THE FUNCTION OF THE CONTROLLER

endmodule
```

Memory

```
module memory(clock, cs, we, address, data_in, data_out);
  input clock;
  input cs;
  input we;
  input[6:0] address;
  input[7:0] data_in;
  output[7:0] data_out;

  reg[7:0] data_out;

  reg[7:0] RAM[0:127];

  always @ (negedge clock)
  begin
    if((we == 1) && (cs == 1))
      RAM[address] <= data_in[7:0];

    data_out <= RAM[address];
  end
endmodule
```

Lab Assignment #7

Guideline
This lab can be done with a partner.

Objective
1. Become familiar with the MIPS ISA
2. Synthesize and implement a basic MIPS processor on the Nexys2 board
3. Learn how to use Verilog Text-IO to initialize a memory image for simulation
4. Extend the MIPS ISA by adding ARM-like instructions

Reading
Please read chapter 9 of you textbook *Digital Systems Design Using Verilog* for background on the MIPS ISA and the basic MIPS implementation.

Summary of tasks
You will use a model of a MIPS processor that handles a subset of the MIPS instructions. This model is provided for you in the book. In Part A you will write a testbench for this model that uses the Verilog text-IO package to initialize the instruction memory. Once the processor is verified in simulation using this testbench, you will synthesize and implement it on the board and run a simple MIPS program to light up some of the board LEDs. Next you will augment the MIPS ISA by adding ARM-like instructions in Part B.

Part A:

Description
Chapter 9 in the book gives you the Verilog MIPS model for you. You will be give code for the MIPS memory and register file. You are also provided a template testbench. This code is appended at the end of the lab description.

Here are your tasks:

Simulation

1. Take the complete MIPS model presented in Figure 9-9 and compile it in Modelsim. You will need to include the MIPS processor, memory, and register file as supporting modules (either as separate entities in the same file, or as separate entities in separate files).

2. Write a testbench to test your MIPS processor. You will use the skeleton testbench provided in this document. Utilize your knowledge of delay and 'display' statements to test the functionality of every instruction in your design. The idea here is to get you familiar with using the Verilog text-IO. Your testbench will use text-IO to initialize the memory with a set of instructions that you will provide in a text file (called the instruction text file hereafter). After initializing the memory with your instructions, the testbench will run the processor for as many cycles as you need to see your program working. You will need to hard-code test instruction to your instruction text file in hex or binary.

Synthesis

3. Modify the above model by adding code to interface it to the input switches and LEDs on the Nexys2 board. Your interface must be able to halt operation of the MIPS processor and display the lower 8 bits of register $1 on eight LEDs. Your interface must also divide the prototyping board's internal clock to provide the model with a slow clock.
    a. You should add a new 'Halt' port to the MIPS top level. When Halt is high your processor should complete the current instruction and not proceed to the next instruction. When halt goes back to zero you should keep executing the normal flow of instructions starting from the next instruction.
    b. Map switches SW0 and SW1 to Reset and Halt, respectively.
    c. Make necessary changes to your register file so that you can use register $1 as a top-level output and map it to LEDs [7:0].
    d. The slow clock can be used to execute instructions in a manner that makes your outputs visible (when you implement this on the board). You may choose the frequency of the slow clock. Using a 50 MHz clock will make any program outputs on the LEDs a blur.

4. Write a program in MIPS assembly to create a rotating light on the LED outputs. The light rotates from one LED to the next. This rotation should not stop. So, if a number denotes a specific LED being lit, the result should be like 0,1,2,3,4,5,6,7,0,1,2,3,… In other words, the LED rotation should happen indefinitely.

5. Translate this program to machine code and put it in your instruction text file. Run the testbench and analyze the processor outputs using the Modelsim Waveform Viewer. Verify that the correct values are being written to register $1 and that they are showing up on your LED outputs. Also verify that reset and halt are working.

6. Synthesize your modified MIPS model and implement it on the Nexys2 board.
    a. You will not need to synthesize the testbench.
    b. The memory should be initialized with your instructions by using Verilog text-IO. Simply call your 'readmemh' function in an initial block and Xilinx will correctly instantiate block RAM and initialize them with these values for you.
    c. Correctly map the switches and LEDs.

Useful Information

1. To read instructions from a file you can use the following options:

    Store the instruction file in **HEX** format: readmemh("file name", mem, start addr, end addr)

    Store the instruction file in **BINARY** format: readmemb("file name", mem, start addr, end addr)

    The text file is in strictly **HEX** or **BINARY** format with a value on each line. You do not have to put "0x" or anything else to denote a HEX value.

    Example hex input file with three entries:

    ```
    01
    02
    03
    ```

2. Since the memory size is very large and you will have a small number of instructions, you can use a for loop in the initial block to fill the rest of the unused locations to zero.

Part B:

Description

In this part of the lab, you will extend the basic MIPS processor you implemented in Part A so that it can execute new instructions. The following table contains a summary of the new instructions you will be required to execute. Details of each instruction and their encodings appear towards the end of this document.

Instruction	Description
JAL	Jump and Link
LUI	Load Upper Immediate
MULT	Multiply Word
MFHI	Move from special register HI
MFLO	Move from special register LO
ADD8	Byte-wise addition
RBIT	Reverse bits in a word
REV	Reverse bytes in a word
SADD	Saturating ADD
SSUB	Saturating Subtract

You will be modifying your processor to execute these instructions and testing your modifications using a test program that is provided (the assembly language version of the test program is available at the end of this lab's description and also on Canvas). The test program uses three switches and two buttons from the board to perform certain operations and show certain results on the 7 segment display. The following table summarizes the functions and display modes of the test program.

SW3	SW2	SW1	Task	BTN1	BTN0	Value to Display on 7 segment
0	0	0	MULT $4, $5 MFLO $2 MFHI $3	0	0	lower 16 bits of $2
				0	1	upper 16 bits of $2
				1	0	lower 16 bits of $3
				1	1	upper 16 bits of $3
0	0	1	ADD8 $2, $4, $5	0	0	lower 16 bits of $2
				0	1	upper 16 bits of$2
0	1	0	LUI $2, imm	0	0	lower 16 bits of $2
				0	1	upper 16 bits of$2
0	1	1	RBIT $2, $5	0	0	lower 16 bits of $2
				0	1	upper 16 bits of$2
1	0	0	REV $2, $4	0	0	lower 16 bits of $2
				0	1	upper 16 bits of$2
1	0	1	SADD $2, $4, $5	0	0	lower 16 bits of $2
				0	1	upper 16 bits of $2
1	1	0	SSUB $2, $5, $4	0	0	lower 16 bits of $2
				0	1	upper 16 bits of $2

Summary of Test Program

Three input switches from the board will be used to load a value into register $1. The assembly program will be running in a loop and will be constantly looking at the value in $1. When the value in $1 changes, the program will jump to a subroutine that performs the Task indicated in Table 2. The program will use JAL to jump to subroutines, so you should make sure this instructions works perfectly. The subroutines use $4 and $5 which will be loaded with some constant values. At the end of the subroutine, the program will continue looping, waiting for a change in $1. While the program is looping, you should be able to press BTN1 and BTN0 in the appropriate combinations to display the value in the result register $2 or $3 on the 7 segment display. The constants that are loaded into $4 and $5 for the computations will be changed during checkouts to make sure your implementation works.

Your tasks

1. Modify your processor model from Part A – extend its functionality so that it can execute the instructions summarized in Table 1 (and detailed in Table 3)
2. In order to run the test program, you have to interface three board switches to one of the registers in the register file. Modify your processor such that SW2, SW1, and SW0 map to the LSB three bits of register $1. You are not restricted to the MIPS interfaces when doing this. Register $1 will be used to branch to various sub-routines that will test the new instructions.
3. In order to view the results from the test program, you need to interface two registers to the 7 segment display. As shown in Table 2, register $2 is used for output in most cases except for the HI part of the multiply result. You should write some code that takes BTN1 and BTN0 as inputs and then displays the upper or lower bytes of $2 or $3 on the 7-segment display as required.
4. Once you have made the necessary modification to your MIPS modules, you will translate the provided assembly language test program to machine code.
5. Initialize your memory using the machine code
6. Synthesize the design and implement it on the board

Useful Information

This part of the lab has only an implementation requirement. However, simulation is recommended to debug your design. If you are unable to implement, be ready with simulation waveforms/do-file/testbench for partial credit.

Submission details

Submit the following things on Canvas:

- All Verilog code (modified MIPS and testbenches/dofiles)
- MIPS assembly program
- Instruction text file containing the machine code of your program
- Bit file and UCF file, if any

Checkout details

The following things will be checked during check-out:

- Your modifications to the code and the testbench.
- Correct functionality of the program on the board.
- Run the test program that you have loaded into your memory and check that all the switches and buttons give the expected result on the 7-segment display for part B.

Skeleton Testbench

```
module MIPS_Testbench ();
  reg CLK;
  reg RST;
  wire CS;
  wire WE;
  wire  [31:0] Mem_Bus;
  wire  [31:0] Address;

  initial
  begin
    CLK = 0;
  end

  MIPS CPU(CLK, RST, CS, WE, Address, Mem_Bus);
  Memory MEM(CS, WE, CLK, Address, Mem_Bus);

  always
  begin
   #10 CLK = !CLK;
  end

  always
  begin
    RST <= 1'b1; //reset the processor

    //Notice that the memory is initialize in the in the memory module not here

    @(posedge CLK);
    // driving reset low here puts processor in normal operating mode
    RST = 1'b0;

    /*  add your testing code here  */
    // you can add in a 'Halt' signal here as well to test Halt operation
    // you will be verifying your program operation using the
    // waveform viewer and/or self-checking operations

    $display("TEST COMPLETE");
    $stop;
    end

endmodule
```

Complete MIPS

```
module Complete_MIPS(CLK, RST, A_Out, D_Out);
/*  Will need to be modified to add functionality   */
  input CLK;
  input RST;
  output [31:0] A_Out;
  output [31:0] D_Out;

  wire CS, WE;
  wire [31:0] ADDR, Mem_Bus;

  assign A_Out = ADDR;
  assign D_Out = Mem_Bus;

  MIPS CPU(CLK, RST, CS, WE, ADDR, Mem_Bus);
  Memory MEM(CS, WE, CLK, ADDR, Mem_Bus);

endmodule
```

Memory

```
module Memory(CS, WE, CLK, ADDR, Mem_Bus);
  input CS;
  input WE;
  input CLK;
  input [31:0] ADDR;
  inout [31:0] Mem_Bus;

  reg [31:0] data_out;
  reg [31:0] RAM [0:127];


  initial
  begin
   /*   Write your Verilog-Text IO code here    */
  end

  assign Mem_Bus = ((CS == 1'b0) || (WE == 1'b1)) ? 32'bZ : data_out;

  always @(negedge CLK)
  begin

    if((CS == 1'b1) && (WE == 1'b1))
      RAM[ADDR] <= Mem_Bus[31:0];

     data_out <= RAM[ADDR];
  end
endmodule
```

Register File

```
module REG(CLK, RegW, DR, SR1, SR2, Reg_In, ReadReg1, ReadReg2);
  input CLK;
  input RegW;
  input [4:0] DR;
  input [4:0] SR1;
  input [4:0] SR2;
  input [31:0] Reg_In;
  output reg [31:0] ReadReg1;
  output reg [31:0] ReadReg2;

  reg [31:0] REG [0:31];
  integer i;

  initial
  begin

    ReadReg1 = 0;
    ReadReg2 = 0;

  end

  always @(posedge CLK)
  begin

    if(RegW == 1'b1)
        REG[DR] <= Reg_In[31:0];

      ReadReg1 <= REG[SR1];
      ReadReg2 <= REG[SR2];
  end
endmodule
```

Details of New Instructions

JAL	Encoding	Bits 31–26: JAL 0 0 0 0 1 1 (6) Bits 25–0: instr_index (26)
	Format	JAL Target
	Description	JAL is used for procedure calls. JAL Target puts the return address (PC+1) in the register $31 and then goes to Target for the next instruction.
	Operation	$31 = PC + 1 New_PC = (PC & 0xf0000000) | (Target);
LUI	Encoding	Bits 31–26: LUI 0 0 1 1 1 1 (6) Bits 25–21: 0 0 0 0 0 0 (5) Bits 20–16: rt (5) Bits 15–0: immediate (16)
	Format	LUI $t, imm
	Description	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
	Operation	$t = imm << 16
MULT	Encoding	Bits 31–26: SPECIAL 0 0 0 0 0 0 (6) Bits 25–21: rs (5) Bits 20–16: rt (5) Bits 15–6: 0 0 0 0 0 0 0 0 0 0 0 (10) Bits 5–0: MULT 0 1 1 0 0 0 (6)
	Format	MULT rs, rt
	Description	The 32-bit word value in reg *rt* is multiplied by the 32-bit value in reg *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.
	Operation	prod = rs[31:0] * rt[31:0] LO = prod[31:0] HI = prod[63:32]
MFHI	Encoding	Bits 31–26: SPECIAL 0 0 0 0 0 0 (6) Bits 25–16: 0 0 0 0 0 0 0 0 0 0 0 (10) Bits 15–11: rd (5) Bits 10–6: 0 0 0 0 0 0 (5) Bits 5–0: MFHI 0 1 0 0 0 0 (6)
	Format	MFHI rd
	Description	The contents of special register HI are stored in the GPR rd
	Operation	rd ← HI

<table>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="4">MFLO</td><td>Encoding</td><td>
<table>
<tr><td>31 26</td><td>25 16</td><td>15 11</td><td>10 6</td><td>5 0</td></tr>
<tr><td>SPECIAL
000000</td><td>0
00 0000 0000</td><td>rd</td><td>0
00000</td><td>MFLO
010010</td></tr>
<tr><td>6</td><td>10</td><td>5</td><td>5</td><td>6</td></tr>
</table>
</td></tr>
<tr><td>Format</td><td>MFLO rd</td></tr>
<tr><td>Description</td><td>The contents of special register LO are stored in the GPR rd</td></tr>
<tr><td>Operation</td><td>Rd ← LO</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="4">ADD8</td><td>Encoding</td><td>
<table>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 11</td><td>10 6</td><td>5 0</td></tr>
<tr><td>SPECIAL
000000</td><td>rs</td><td>rt</td><td>rd</td><td>0
00000</td><td>ADD8
101101</td></tr>
<tr><td>6</td><td>5</td><td>5</td><td>5</td><td>5</td><td>6</td></tr>
</table>
</td></tr>
<tr><td>Format</td><td>ADD8 rd, rs, rt</td></tr>
<tr><td>Description</td><td>This perform byte-wise addition as illustrated below</td></tr>
<tr><td>Operation</td><td>rd[31:24] = rs[31:24]+rt[31:24]
rd[23:16] = rs[23:16]+rt[23:16]
rd[15:8] = rs[15:8]+ rt[15:8]
rd[7:0] = rs[7:0]+ rt[7:0]</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="4">RBIT</td><td>Encoding</td><td>
<table>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 6</td><td>5 0</td></tr>
<tr><td>SPECIAL
000000</td><td>rs</td><td>rt</td><td>0
00 0000 0000</td><td>RBIT
101111</td></tr>
<tr><td>6</td><td>5</td><td>5</td><td>10</td><td>6</td></tr>
</table>
</td></tr>
<tr><td>Format</td><td>RBIT rs, rt</td></tr>
<tr><td>Description</td><td>Reverse the bits in a word</td></tr>
<tr><td>Operation</td><td>for(i=0;i<32;i++){
rs[i]=rt[31-i]}</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="4">REV</td><td>Encoding</td><td>
<table>
<tr><td>31 26</td><td>25 21</td><td>20 16</td><td>15 6</td><td>5 0</td></tr>
<tr><td>SPECIAL
000000</td><td>rs</td><td>rt</td><td>0
00 0000 0000</td><td>REV
110000</td></tr>
<tr><td>6</td><td>5</td><td>5</td><td>10</td><td>6</td></tr>
</table>
</td></tr>
<tr><td>Format</td><td>REV rs, rt</td></tr>
<tr><td>Description</td><td>Reverse the bytes in a word</td></tr>
<tr><td>Operation</td><td>rs[31:24]=rt[7:0]
rs[23:16]=rt[15:8]
rs[15:8]=rt[23:16]
rs[7:0]=rt[31:24]</td></tr>
</table>

SADD	Encoding	Bits 31–26: SPECIAL 000000 (6); 25–21: rs (5); 20–16: rt (5); 15–11: rd (5); 10–6: 0 00000 (5); 5–0: SADD 110001 (6)
	Format	SADD rd, rs, rt
	Description	Saturating addition
	Operation	If ((rs + rt) > 2^32 – 1), then rd = 2^32 – 1; else rd = rs + rt;
SSUB	Encoding	Bits 31–26: SPECIAL 000000 (6); 25–21: rs (5); 20–16: rt (5); 15–11: rd (5); 10–6: 0 00000 (5); 5–0: SSUB 110010 (6)
	Format	SSUB rd, rs, rt
	Description	Saturating Subtraction
	Operation	if ((rs - rt)< 0) then rd = 0; else rd = rs - rt;

Test Program

```
start:
     addi $6, $1, 0
     andi $8, $8, 0
     lui  $4, 28672
     lui  $5, 32767
     ori  $8, $8, 11

loop:
     beq  $6, $1, loop
     addi $6, $1, 0
     sll  $7, $1, 1
     add  $7, $8, $7
     jr   $7
     j    loop

call_table:
     jal operation0
     j loop
     jal operation1
     j loop
     jal operation2
     j loop
     jal operation3
     j loop
     jal operation4
     j loop
     jal operation5
     j loop
     jal operation6
     j loop
```

```
operation0:
     mult  $4,$5
     mflo $2
     mfhi $3
     jr $31

operation1:
     add8 $2, $4, $5
     jr $31

operation2:
     lui $2, 4096
     jr $31

operation3:
     rbit $2, $5
     jr $31

operation4:
     rev $2, $4
     jr $31

operation5:
     sadd $2, $5, $5
     jr $31

operation6:
     ssub $2, $4, $5
     jr $31
```

Lab Assignment #8a

Guideline

This lab can be done with a partner.

Objective

1. To introduce you to JTAG
2. To introduce you to DFT, in particular Memory BIST

Description

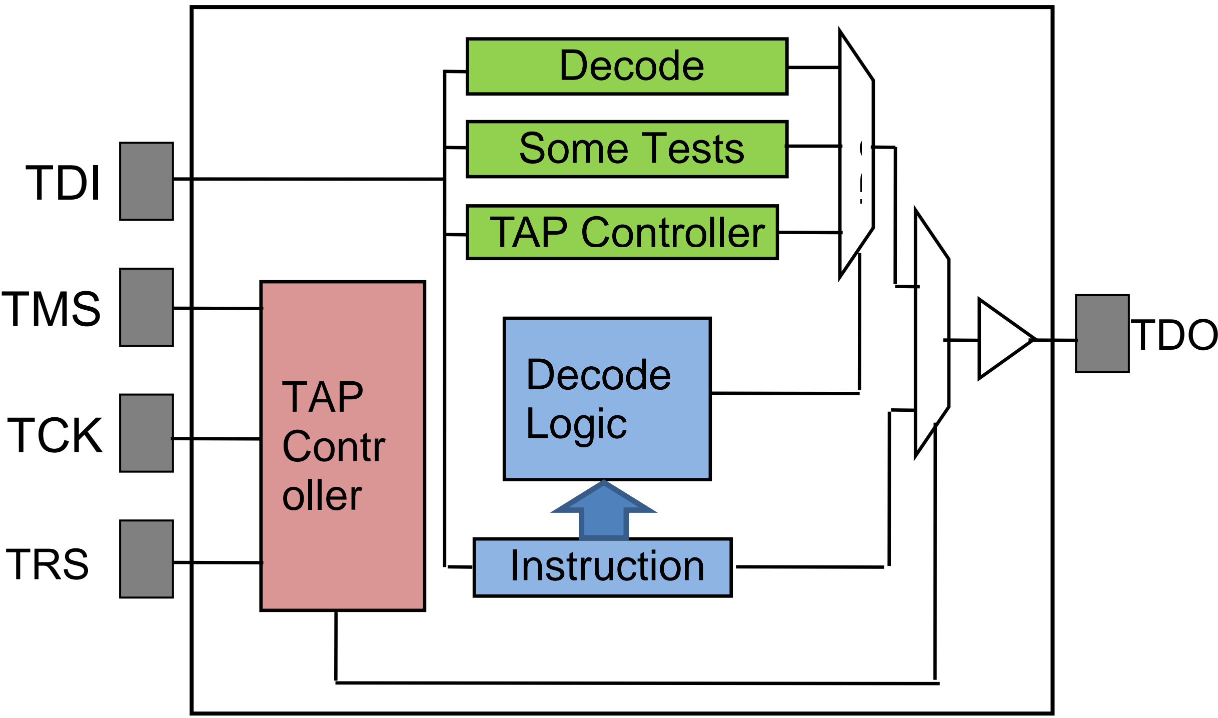
In this lab, we will add a memory BIST engine to the memory module we had in Part A. The memory BIST engine will test the memory in your design. It will be operated by a register which will be programmed by a JTAG interface.

JTAG is a 4 pin serial protocol. It is an IEEE standard. (The 4 pins have standard names and functions. 5th pin –TRST – is optional). The most common use of JTAG (in fact, the reason why it was developed) is boundary scan, which you would have read in class. But JTAG finds its use in several other places. E.g. the programming of the FPGA on the lab board is done using a JTAG interface. Another common example of JTAG application is to perform testing.

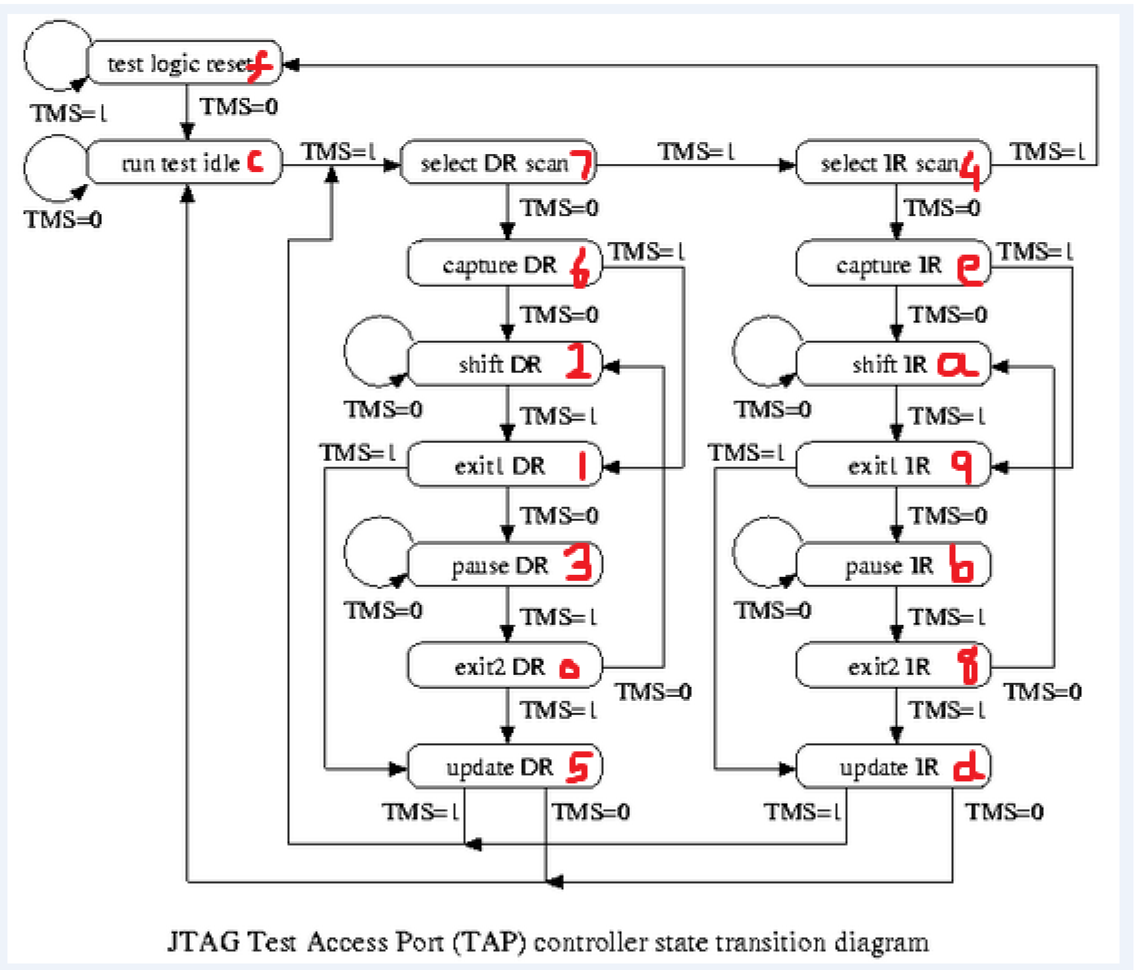
The basic concept that makes JTAG so versatile is that it is comprises of a state machine (called TAP controller) which can be used to program a register (or many registers). These registers can be used for any purpose in the design. For the boundary scan purposes, the register that is programmed is called the Boundary Scan Register. For memory BIST, say, you can have a memory BIST register. For some other purpose, you may have any other register.

The merit of JTAG is that it lets you achieve a lot by just using 4/5 pins (this is the general advantage of any serial protocol). Let us take an example of a testing scenario. Let's say that for testing a particular aspect of your chip, you need to control 10 signals in your design and you need to observe 15 signals in your design. If you use the simplest possible approach, you can add 25 pins on the top level of the chip and your work is done. But it is not justified to add so many pins (let us say the chip overall has 50 pins; then adding 25 pins is 50% overhead) just for the purposes of test. If you have JTAG in your design (which a state-of-the-art will definitely have for certain other reasons), you can use the 4/5 JTAG pins to control and observe as many signals as you like in the design, by just adding an extra register.

A simplified diagram of JTAG is given below. Refer to section 10.4 of text for details.



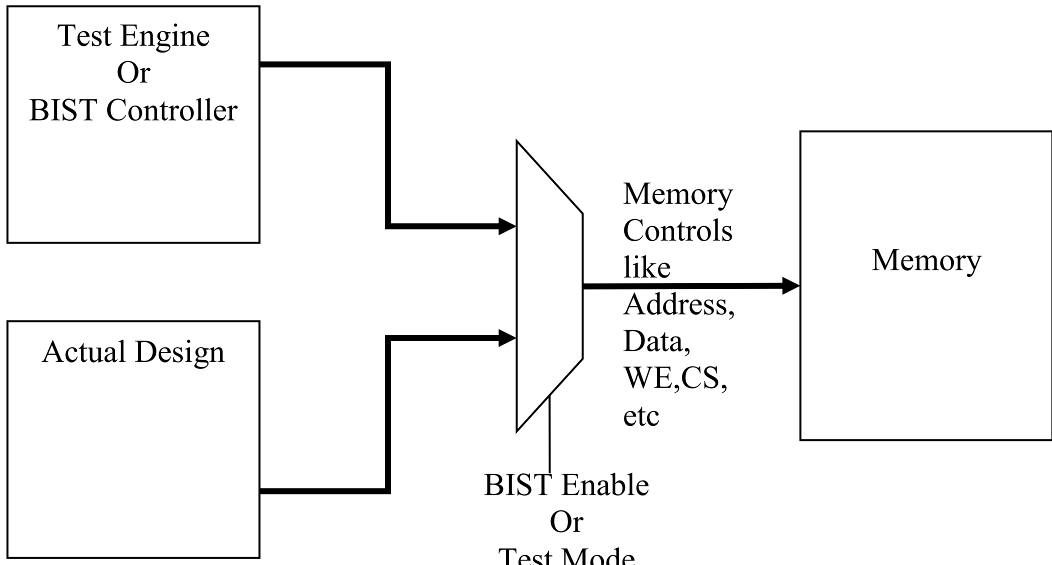
And the TAP state machine is also given here. The states are annotated with numbers which are used in the lab code.



JTAG Test Access Port (TAP) controller state transition diagram

Let us take some time to understand what memory BIST is. The purpose of memory BIST is to be able to test an on-chip memory (just like we have in our stack calculator design in lab 6a) by designing a testing engine which resides on the chip itself. So, during the normal operation of the chip (called the functional or system mode), the design works normally, as if nothing else is present. In our case, the stack calculator controller accesses the memory normally. But when we have to test the memory (ie. when we are in the test mode or the memory BIST mode, to be specific), this testing engine grabs the control of the memory interface and tests it. The muxing logic used to grab the control of the memory ports is called 'BIST Collar'.

Testing a memory involves very sophisticated algorithms (the most common one being the MARCH algorithm), which are beyond the scope of this lab. But the simplest memory tests work like this. We write some data to an address and then read it back. If we obtain the same data, it means the memory is fine. Otherwise the memory is bad. We will use a similar testing approach for this lab.

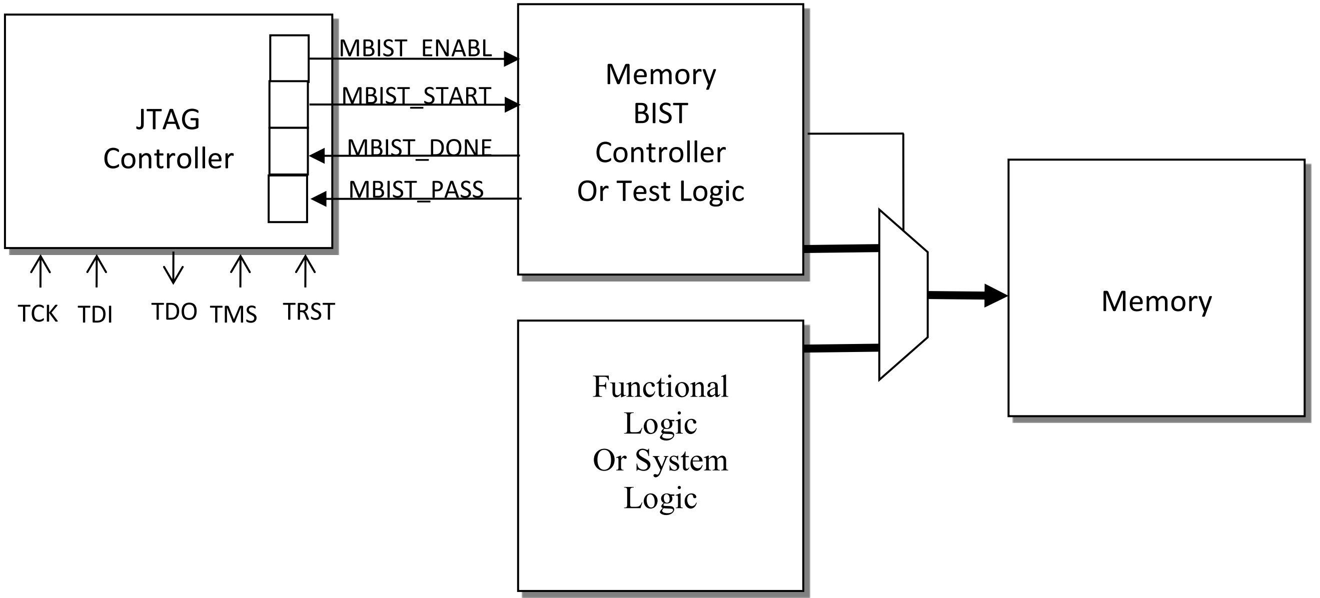


In this lab, we will use JTAG to program a memory BIST register. By using 5 top level pins (TCK, TRST, TMS, TDI, TDO), we will program a register (we will call it MBIST register) via the JTAG protocol. This register will have four bits:

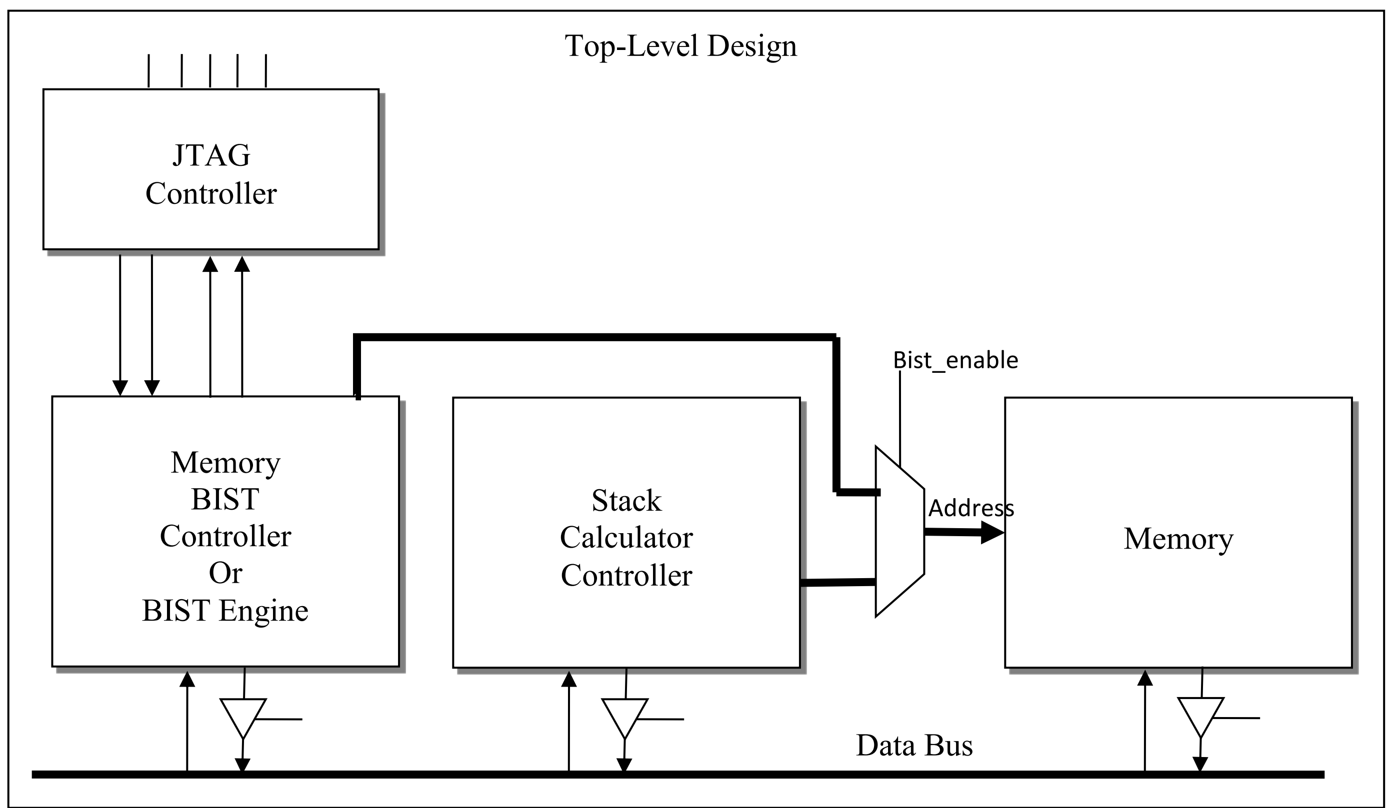
Bit 3 (MSB)	Bit 2	Bit 1	Bit 0 (LSB)
MBIST_PASS	MBIST_DONE	MBIST_START	MBIST_ENABLE

The first two bits will control the MBIST test engine in the design, and the last two bits will observe the results of the test. MBIST_ENABLE lets the BIST engine take control of the memory ports (address, data, we, cs etc). MBIST_START starts the testing operation (writing to an address, reading it back, writing to the next address and so on). MBIST_DONE is an 'observe' signal. It tells if the testing operation is complete. MBIST_PASS is also an 'observe' signal which tells whether the test passed or not.

So, overall the top level design will look like this:



In our case, because the memory has a special interface, the top level would look like this:



So, just to re-iterate the entire flow of the design:

JTAG controller sends and receives signals from the BIST controller. The BIST controller controls the memory in TEST mode, while the stack calculator controls the memory in FUNCTIONAL mode. The following steps discuss this in a little more detail:

1. User programs the MBIST register to being BIST
   a. User selects the MBIST register in the JTAG controller (TAP IR branch)
   b. User programs the MBIST register (TAP DR branch)
2. MBIST controller receives the START signal (the START bit in the register is SET in step 1b) and the MBIST collar receives the ENABLE signal (the ENABLE bit in the register is SET in step 1b).
3. MBIST controller starts testing the memory – writing to and reading from memory locations based on the algorithms which were incorporated while designing the controller
4. When the entire memory is tested, the MBIST controller makes the DONE signal HIGH. Also, the PASS signal reflects whether the test passed or failed. These signals are connected to the MBIST register
5. User shifts out the results of the test
   a. User selects the MBIST register in the JTAG controller (TAP IR branch)
   b. User shifts out the MBIST register (TAP DR branch)

Important Information

This lab is a simulation-only lab. We have given you most of the files you will need for this lab. You need to fill in the areas marked with ***. You will need to design the other parts.**

Your tasks

1. Design the Memory BIST controller

   Our memory BIST controller will be simple random testing logic. You will have to write two LFSRs – one of data and one for address. Refer to section 10.5 in the text for details on designing LFSRs. The address LFSR should be maximal length so that all addresses are tested. The data LFSR will be 8 bits long while the address LFSR will be 7 bits long. The controller will essentially be a small state machine. It will write to a random address (generated by the address LFSR) in the memory, a random data (generated by the data LFSR). Then it will read the data from the memory and compare it with what was written. Then it will move on to the next random address and so on. Finally it will end when 128 addresses have been generated. Keep in mind that the LFSR cannot generate an all zero number. Thus, the 0th address will remain untested. You can add testing the 0th address for a bonus of 10 points. You don't need to have separate entities for the LFSRs. They can just be incorporated in one state of the BIST engine.

2. Integrate and develop the TOP module

   The JTAG controller is being given to you as an IP. Integrate the given JTAG controller and the Memory BIST controller you developed in (1) with other parts as shown in the figure above. Also write the logic which lets the BIST controller gain access to the memory in the BIST test mode.

3. Complete the testbench

   The testbench given to you contains code for selecting the IDCODE register and shifting it out. You need to extend it such that in addition to what it does already, it should do the following operations:

1. Program IR to select the mbist register.
2. Program mbist register to set these bits: mbist_enable and mbist_start.
3. Wait for the number of cycles you expect MBIST to run (you can wait for more cycles).
4. Program IR to select mbist register.
5. Shift out mbist register to confirm that mbist_done is set. Check mbist_pass bit. It should be '1'.

4. Now run the testbench for appropriate time using ModelSim. You should be able to observe MBIST_DONE and MBIST_PASS. Now restart simulation, from the transcript window in ModelSim, force address 0x23 with 0x00, and re-run the testbench. This time mbist_pass should be 0.

Submission Details

Submit all Verilog code through Canvas. No hard copy submission is required.

Checkout Details

Demonstrate the MBIST operation (both a passing and a failing case) to the TA during checkout.

Useful Information

1. The TAP state machine has two main parts: the Instruction Register (IR) side and the Data Register (DR) side. [It is important to note that there is one IR but there can be multiple DRs in the design – like IDCODE Register, Boundary Scan Register, Bypass Register, MBIST register etc]. When using the TAP state machine, the first thing is to program the instruction register. This step uses the IR side of the TAP state machine. Programming the instruction register instructs the TAP to select a particular register as the data register (for example, we would want to select the MBIST register as the data register in our case). Then we can program (or read out) the contents of the selected data register. This step will use the DR side of the TAP state machine.

To move through the state machine, we have to wiggle the pins TMS and TCK of the design. To shift data in and out of the registers (whether IR or the DRs), we have to use the TDI and TDO pins. The TRST pin is used as an async. reset.

Let's look at how we can program the instruction register.

Step1: Keep TMS high and give more than 5 TCK cycles. This will bring the TAP machine in reset.
Step2: Make TMS 0 and pulse TCK. This takes the machine to Run-Test-Idle.
Step3: Make TMS 1 and pulse TCK. This takes the machine to Select-DR-Scan
Step4: Make TMS 1 and pulse TCK. This takes the machine to Select-IR-Scan
Step5: Make TMS 0 and pulse TCK. This takes the machine to Capture IR
Step6: Make TMS 0 and pulse TCK. This takes the machine to Shift IR. When you are in Shift IR, you must remain in Shift IR for N number of cycles (where N is the width of the IR) by keeping TMS 0. For these N cycles, you should drive TDI (at or just before posedge of TCK) with the value that you want to program into the IR, one bit at a time.
Step7: Make TMS 1 and pulse TCK. This takes the machine to Exit1-IR.
Step8: Make TMS 1 and pulse TCK. This takes the machine to Update-IR.
Step9: Make TMS 0 and pulse TCK. This takes the machine to Run-Test-Idle.

Now you have successfully programmed the IR. Let us say you programmed the IR to select the IDCODE register as the data register. Now lets us look at the steps which we need to observe the value of the IDCODE register. [You should not reset the machine by going to the Test-Logic-Reset state after having programmed the IR]

Step1: (We are in the Run-Test-Idle state). Make TMS 1 and pulse TCK. This takes the machine to Select-DR-Scan
Step2: Make TMS 0 and pulse TCK. This takes the machine to Capture DR

Step3: Make TMS 0 and pulse TCK. This takes the machine to Shift DR. When you are in Shift DR, you must remain in Shift DR for N number of cycles (where N is the width of the data register which you had selected when you programmed the IR) by keeping TMS 0. For these N cycles, you should observe TDO (at or just after negedge of TCK). The bits on TDO are the value of the data register.
Step4: Make TMS 1 and pulse TCK. This takes the machine to Exit1-DR.
Step5: Make TMS 1 and pulse TCK. This takes the machine to Update-DR.
Step6: Make TMS 0 and pulse TCK. This takes the machine to Run-Test-Idle.

A similar set of operations has been done in the Verilog testbench given to you. You have to modify it to get program the MBIST register and read out the same register after the BIST operation is complete.

2. It is a good idea to approach the design in parts. Get the MBIST Engine working first. Simulate just the BIST Engine with the memory to see if the BIST Engine is working properly. Then put in the JTAG controller.

3. If we assume that 0 is a data that will never be generated by the Data LFSR, forcing 0 on the data at an address in the memory should lead to BIST_PASS=0. Also, since the LFSR is maximal length, each address in the memory is being tested and hence, forcing only one of the addresses to 0 should lead to BIST_PASS=0. To force the data at a particular address to 0, just modify the always block that drives any location in memory to 0.

4. You do not need to create separate entities for LFSRs (this complicates the design). Just create an LFSR like this:

```
fb <= addr[1] ^ addr[5];  --this is just an example

addr <= {fb, addr[5:0]};
```

5. You can make BIST_DONE=1 and BIST_PASS=0 the moment you see the first failure. You do not need to test the complete memory in that case.

6. After waiting for an appropriate amount of time in your testbench for the tests to complete, you might have to wait for a certain number of @(negedge tck) signals (our solution needed 2 after waiting a long time). Be careful with your testbench's timing, as this could throw off when you are reading the output of the mbist reg.

```
////TESTBENCH
////YOU HAVE TO ADD CODE TO THIS MODULE

module testbench;
    reg tdi, tck, tms, trst_b;
    wire tdo, mbist_done, mbist_pass, mbist_start, mbist_enable;
    wire [3:0] idcode_inst = 4'b0011;
    wire [3:0] mbist_inst = 4'b0010;
    wire [3:0] mbist_reg = 4'b1111;

    //***** add additional wires/regs here

    integer i;

    //***** add the instantiation (port map) of the DUT (top)

    initial begin
        tck = 0;
    end

    always begin
        #10 tck = ~tck;
    end

    always begin
        tms <= 1'b1;
        trst_b <= 1'b0;
        #200; //reset the design
        trst_b <= 1'b1;
        @(negedge tck);
        //start traversing the tap state machine
        tms <= 1'b1;
        @(negedge tck);

        tms <= 1'b1; //remain in test logic reset
        @(negedge tck);

        tms <= 1'b1; //remain in test logic reset
        @(negedge tck);

        tms <= 1'b0; //takes to run test / idle
        @(negedge tck);

        tms <= 1'b1; //takes to select dr scan
        @(negedge tck);

        tms <= 1'b1; //takes to select ir scan
        @(negedge tck);

        tms <= 1'b0; //takes to capture ir
        @(negedge tck);

        //shift in the idcode instruction through the tdi
        for(i = 0 ; i <= 3 ; i=i+1) begin
            tms <= 1'b0;
            @(negedge tck);

            tdi <= idcode_inst[i];
        end
```

```
        tms <= 1'b1; //takes to exit1 ir
        @(negedge tck);

        tms <= 1'b1; //takes to update ir
        @(negedge tck);

        tms <= 1'b1; //takes to select dr scan
        @(negedge tck);

        tms <= 1'b0; //takes to capture dr
        @(negedge tck);
        for(i = 0 ; i <= 3 ; i=i+1) begin
            //this is when the id code is being shifted out bit by bit on tdo
            tms <= 1'b0; //takes to shift dr
            @(negedge tck);
        end
        tms <= 1'b1; //takes to exit 1 dr
        @(negedge tck);

        tms <= 1'b1; //takes to update dr
        @(negedge tck);

        tms <= 1'b0; //takes to idle
        @(negedge tck);
        //*****add code here for
        //*****    1. selecting the mbist register
        //*****    2. making the start and enable bits '1'
        //*****    3. waiting till bist done becomes '1'
        $stop;
    end
endmodule
```

```
////TOP LEVEL DESIGN
////YOU HAVE TO ADD CODE TO THIS MODULE

module top(tdi, tck, tms, trst_b, tdo, stack_calc_clk, swtchs, btns, segs, leds, an);

    input tdi, tck, tms, trst_b;
    output tdo;
    input stack_calc_clk; // :in bit := '0'; HOW TO INITIALIZE???
    input [7:0] swtchs;
    input [3:0] btns;
    output [7:0] segs, leds;
    output [3:0] an;
    //*****components
    //--jtag_controller
    //--memory
    //--bist_engine
    //--stack_calc_controller

    //*****internal signals
    //*****bist collar muxes for address, we, cs
    //*****tristates for data bus

    //*****port maps for the components
endmodule
```

```
//--BIST ENGINE
//--YOU HAVE TO ADD CODE TO THIS MODULE

module bist_engine(clk, start, cs, we, address, data_in, data_out, pass,
done);

    input clk, start; output cs, we;  output [6:0] address;
    input [7:0] data_in;
    output [7:0] data_out;
    output pass, done;

    //**** write the architecture of the bist engine
    //**** you don't need to have lfsr's as separate modules

endmodule
```

```
//--STACK CALCULATOR CONTROLLER
//--YOU MAY USE YOUR LAB6A MODULE HERE OR JUST LEAVE THIS AS IS

module stack_calc_controller(clock, swtchs, btns, segs, leds, an, cs, we, data_in,
data_out,
address);

    input clock;
    input [7:0] swtchs;
    input [3:0] btns;
    output [7:0] segs, leds;
    output [3:0] an;
    output cs, we;
    input [7:0] data_in;
    output [7:0] data_out;
    output [6:0] address;

endmodule
```

```
//--MEMORY
//--USE THIS CODE AS IS (uncomment always block to force location 0x23 with 0x00)

module memory(clock, cs, we, address, data_in, data_out);
    input clock, cs, we;
    input [6:0] address;
    input [7:0] data_in;
    output [7:0] data_out;

    reg [7:0] data_out;
    reg [7:0] RAM[0:127];
    /*
    always begin
        #1 RAM[23] = 8'b00000000;
    end
    */
    always @ (negedge clock) begin
        if((we == 1'b1) && (cs == 1'b1))
            RAM[address] <= data_in[7:0];

        data_out <= RAM[address];
    end
endmodule
```

```
//--YOU SHOULD NOT NEED TO MODIFY ANY MODULES THAT FOLLOW
//--JTAG CONTROLLER

module jtag_controller(tdi, tck, tms, trst_b, tdo, mbist_done, mbist_pass, mbist_start,
mbist_enable);

    input tdi, tck, tms, trst_b;
    output tdo;
    input mbist_done, mbist_pass;
    output mbist_start, mbist_enable;


    wire instr_sel;
    wire update_dr, update_ir;
    wire capture_dr, capture_ir;
    wire shift_dr, shift_ir;
    wire inst_tdo, mbist_tdo, idcode_tdo;
    wire idcode_reg_sel, mbist_reg_sel;

    tap_cntl i_tap(tms, tck, trst_b, instr_sel, update_dr, update_ir,
        shift_dr, shift_ir, capture_dr, capture_ir);

    instruction_reg i_instruction_reg(tck, trst_b, tdi,
        capture_ir, shift_ir, update_ir,
        inst_tdo, mbist_reg_sel, idcode_reg_sel, ); //last port is OPEN
        //(implicitly disconnected)

    mbist_reg i_mbist_reg(tck, trst_b, tdi,
        capture_dr, update_dr, shift_dr, mbist_pass, mbist_done,
        mbist_start, mbist_enable, mbist_tdo, mbist_reg_sel);

    idcode_reg i_idcode_reg(tck, trst_b, tdi,
        capture_dr, update_dr, shift_dr, idcode_tdo,
        idcode_reg_sel);

    tdo_select i_tdo_select(instr_sel, idcode_reg_sel,
        mbist_reg_sel, inst_tdo, mbist_tdo, idcode_tdo,
        tdo);

endmodule
```

```
//--TAP CONTROLLER
module tap_cntl(tms, tck, trst_b, instr_sel, update_dr, update_ir, shift_dr, shift_ir,
capture_dr,
capture_ir);

    input tms, tck, trst_b;
    output reg instr_sel;
    output reg update_dr, update_ir;
    output reg shift_dr, shift_ir;
    output reg capture_dr, capture_ir;

    reg [3:0] state, next_state;
    reg state_rst_b;

    parameter EXIT2_DR_STATE       = 4'h0;
    parameter EXIT1_DR_STATE       = 4'h1;
    parameter SHIFT_DR_STATE       = 4'h2;
    parameter PAUSE_DR_STATE       = 4'h3;
    parameter SELECT_IR_SCAN_STATE = 4'h4;
    parameter UPDATE_DR_STATE      = 4'h5;
    parameter CAPTURE_DR_STATE     = 4'h6;
    parameter SELECT_DR_SCAN_STATE = 4'h7;
    parameter EXIT2_IR_STATE       = 4'h8;
    parameter EXIT1_IR_STATE       = 4'h9;
    parameter SHIFT_IR_STATE       = 4'hA;
    parameter PAUSE_IR_STATE       = 4'hB;
    parameter IDLE_STATE           = 4'hC;
    parameter UPDATE_IR_STATE      = 4'hD;
    parameter CAPTURE_IR_STATE     = 4'hE;
    parameter RESET_STATE          = 4'hF;

    //next state logic
    always @ (posedge tck, negedge trst_b) begin
        if(!trst_b)
            state <= RESET_STATE;
        else
            state <= next_state;
    end

    //control lines
    always @ (negedge tck, negedge trst_b) begin
        if(!trst_b) begin
            update_ir <= 1'b0;
            update_dr <= 1'b0;
            capture_ir <= 1'b0;
            capture_dr <= 1'b0;
            state_rst_b <= 1'b0;
            shift_ir <= 1'b0;
            shift_dr   <= 1'b0;
        end
        else begin
            case(state)
                EXIT2_DR_STATE, EXIT1_DR_STATE, SELECT_IR_SCAN_STATE,
SELECT_DR_SCAN_STATE, EXIT2_IR_STATE,
                EXIT1_IR_STATE, PAUSE_IR_STATE,
                IDLE_STATE, PAUSE_DR_STATE: begin
                    update_ir  <= 1'b0;
                    update_dr  <= 1'b0;
                    capture_ir <= 1'b0;
```

```
        capture_dr <= 1'b0;
        state_rst_b <= 1'b1;
        shift_ir   <= 1'b0;
        shift_dr   <= 1'b0;
    end

    CAPTURE_DR_STATE: begin
        update_ir   <= 1'b0;
        update_dr   <= 1'b0;
        capture_ir  <= 1'b0;
        capture_dr  <= 1'b1;
        state_rst_b <= 1'b1;
        shift_ir    <= 1'b0;
        shift_dr    <= 1'b0;
    end

    CAPTURE_IR_STATE: begin
        update_ir   <= 1'b0;
        update_dr   <= 1'b0;
        capture_ir  <= 1'b1;
        capture_dr  <= 1'b0;
        state_rst_b <= 1'b1;
        shift_ir    <= 1'b0;
        shift_dr    <= 1'b0;
    end

    SHIFT_DR_STATE: begin
        update_ir   <= 1'b0;
        update_dr   <= 1'b0;
        capture_ir  <= 1'b0;
        capture_dr  <= 1'b0;
        state_rst_b <= 1'b1;
        shift_ir    <= 1'b0;
        shift_dr    <= 1'b1;
    end

    SHIFT_IR_STATE: begin
        update_ir   <= 1'b0;
        update_dr   <= 1'b0;
        capture_ir  <= 1'b0;
        capture_dr  <= 1'b0;
        state_rst_b <= 1'b1;
        shift_ir    <= 1'b1;
        shift_dr    <= 1'b0;
    end

    UPDATE_DR_STATE: begin
        update_ir   <= 1'b0;
        update_dr   <= 1'b1;
        capture_ir  <= 1'b0;
        capture_dr  <= 1'b0;
        state_rst_b <= 1'b1;
        shift_ir    <= 1'b0;
        shift_dr    <= 1'b0;
    end
    UPDATE_IR_STATE: begin
        update_ir   <= 1'b1;
        update_dr   <= 1'b0;
        capture_ir  <= 1'b0;
```

```
                capture_dr  <= 1'b0;
                state_rst_b <= 1'b1;
                shift_ir    <= 1'b0;
                shift_dr    <= 1'b0;
            end

            RESET_STATE: begin
                update_ir   <= 1'b0;
                update_dr   <= 1'b0;
                capture_ir  <= 1'b0;
                capture_dr  <= 1'b0;
                state_rst_b <= 1'b0;
                shift_ir    <= 1'b0;
                shift_dr    <= 1'b0;
            end
            default: begin
                update_ir   <= 1'b0;
                update_dr   <= 1'b0;
                capture_ir  <= 1'b0;
                capture_dr  <= 1'b0;
                state_rst_b <= 1'b1;
                shift_ir    <= 1'b0;
                shift_dr    <= 1'b0;
            end
        endcase
    end
end
always @ (state, tms) begin
    case(state)
        EXIT2_DR_STATE: begin
            if(tms == 1'b1)
                next_state <= UPDATE_DR_STATE;
            else
                next_state <= SHIFT_DR_STATE;
        end
        EXIT1_DR_STATE: begin
            if(tms == 1'b1)
                next_state <= UPDATE_DR_STATE;
            else
                next_state <= PAUSE_DR_STATE;
        end
        SHIFT_DR_STATE: begin
            if(tms == 1'b1)
                next_state <= EXIT1_DR_STATE;
            else
                next_state <= SHIFT_DR_STATE;
        end
        PAUSE_DR_STATE: begin
            if(tms == 1'b1)
                next_state <= EXIT2_DR_STATE;
            else
                next_state <= PAUSE_DR_STATE;
        end
        SELECT_IR_SCAN_STATE: begin
            if(tms == 1'b1)
                next_state <= RESET_STATE;
            else
                next_state <= CAPTURE_IR_STATE;
        end
```

```
UPDATE_DR_STATE: begin
    if(tms == 1'b1)
        next_state <= SELECT_DR_SCAN_STATE;
    else
        next_state <= IDLE_STATE;
end
CAPTURE_DR_STATE: begin
    if(tms == 1'b1)
        next_state <= EXIT1_DR_STATE;
    else
        next_state <= SHIFT_DR_STATE;
end
SELECT_DR_SCAN_STATE: begin
    if(tms == 1'b1)
        next_state <= SELECT_IR_SCAN_STATE;
    else
        next_state <= CAPTURE_DR_STATE;
end
EXIT2_IR_STATE: begin
    if(tms == 1'b1)
        next_state <= UPDATE_IR_STATE;
    else
        next_state <= SHIFT_IR_STATE;
end
EXIT1_IR_STATE: begin
    if(tms == 1'b1)
        next_state <= UPDATE_IR_STATE;
    else
        next_state <= PAUSE_IR_STATE;
end
SHIFT_IR_STATE: begin
    if(tms == 1'b1)
        next_state <= EXIT1_IR_STATE;
    else
        next_state <= SHIFT_IR_STATE;
end
PAUSE_IR_STATE: begin
    if(tms == 1'b1)
        next_state <= EXIT2_IR_STATE;
    else
        next_state <= PAUSE_IR_STATE;
end
IDLE_STATE: begin
    if(tms == 1'b1)
        next_state <= SELECT_DR_SCAN_STATE;
    else
        next_state <= IDLE_STATE;
end
UPDATE_IR_STATE: begin
    if(tms == 1'b1)
        next_state <= SELECT_DR_SCAN_STATE;
    else
        next_state <= IDLE_STATE;
end
CAPTURE_IR_STATE: begin
    if(tms == 1'b1)
        next_state <= EXIT1_IR_STATE;
    else
        next_state <= SHIFT_IR_STATE;
```

```
            end
            RESET_STATE: begin
                if(tms == 1'b1)
                    next_state <= RESET_STATE;
                else
                    next_state <= IDLE_STATE;
            end
            default: begin
                next_state <= RESET_STATE;
            end
        endcase
    end


    always @ (state) begin
        if((state == EXIT2_IR_STATE) || (state == EXIT1_IR_STATE) ||
            (state == SHIFT_IR_STATE) || (state == PAUSE_IR_STATE) ||
            (state == UPDATE_IR_STATE) || (state == CAPTURE_IR_STATE) ||
            (state == IDLE_STATE) || (state == RESET_STATE))
            instr_sel <= 1'b1;
        else
            instr_sel <= 1'b0;
    end
endmodule
```

```
//--INSTRUCTION REGISTER
module instruction_reg(tck, trst_b, tdi, capture_ir, shift_ir, update_ir, inst_tdo,
  mbist_reg_en, idcode_reg_en, bypass_reg_en);

  input tck, trst_b, tdi;
  input capture_ir, shift_ir, update_ir;
  output reg inst_tdo;
  output mbist_reg_en, idcode_reg_en, bypass_reg_en;

  reg [3:0] shift_reg;
  reg [3:0] instruction_reg;


  //shift register
  always @ (posedge tck, negedge trst_b) begin
    if(trst_b == 0)
      shift_reg <= 4'b0000;
    else begin
      if(capture_ir == 1'b1)
        shift_reg <= instruction_reg;
      else if(shift_ir == 1'b1)
        shift_reg <= {tdi , shift_reg[3:1]};
    end
  end

  //instruction register
  always @ (negedge tck, negedge trst_b) begin
    if(trst_b == 0)
      instruction_reg <= 4'b0000;
    else begin
      if(update_ir == 1'b1)
        instruction_reg <= shift_reg;
    end
  end

  //tdo flop
  always @ (negedge tck, negedge trst_b) begin
    if(trst_b == 0)
      inst_tdo <= 1'b0;
    else
      inst_tdo <= shift_reg[0];
  end

  assign mbist_reg_en   = ((instruction_reg == 4'b0010) && (trst_b)) ? 1'b1 : 1'b0;
  assign idcode_reg_en = ((instruction_reg == 4'b0011) && (trst_b)) ? 1'b1 : 1'b0;
  assign bypass_reg_en  = ((instruction_reg == 4'b1111) && (trst_b)) ? 1'b1 : 1'b0;
endmodule
```

```
//--IDCODE REGISTER

module idcode_reg(tck, trst_b, tdi, capture_dr, update_dr, shift_dr, idcode_tdo,
idcode_reg_en);
  input tck, trst_b, tdi;
  input capture_dr, update_dr, shift_dr;
  output reg idcode_tdo;
  input idcode_reg_en;

  wire [3:0] idcode;
  reg [3:0] shift_reg;


  assign idcode = 4'b0101;

  //shift register
  always @ (posedge tck, negedge trst_b) begin
    if(!trst_b)
      shift_reg <= 4'b0000;
    else begin
      if(capture_dr == 1'b1)
        shift_reg <= idcode;
      else if(shift_dr == 1'b1)
        shift_reg <= {tdi , shift_reg[3:1]};
    end
  end

  //tdo flop
  always @ (negedge tck, negedge trst_b) begin
    if(!trst_b)
      idcode_tdo <= 1'b0;
    else
      idcode_tdo <= shift_reg[0];
  end

  always @ (negedge tck, negedge trst_b) begin
    if(!trst_b)
      ; //do nOTHING!!!
    else begin
      if((idcode_reg_en) && (update_dr))
        ; //do nOTHING!!!
    end
  end

endmodule
```

```
//--MBIST REGISTER
//mbist_reg[0] -> mbist_enable
//mbist_reg[1] -> mbist_start
//mbist_reg[2] -> mbist_done
//mbist_reg[3] -> mbist_pass

module mbist_reg(tck, trst_b, tdi, capture_dr, update_dr, shift_dr,
  mbist_pass, mbist_done, mbist_start, mbist_enable, mbist_tdo, mbist_reg_en);

  input tck, trst_b, tdi;
  input capture_dr, update_dr, shift_dr;
  input mbist_pass, mbist_done;
  output reg mbist_start, mbist_enable;
  output reg mbist_tdo;
  input mbist_reg_en;

  wire [3:0] mbist_reg;
  reg [3:0] shift_reg;


  assign mbist_reg = {mbist_pass , mbist_done , mbist_start , mbist_enable};

  //shift register
  always @ (posedge tck, negedge trst_b) begin
    if(!trst_b)
      shift_reg <= 4'b0000;
    else begin
      if(capture_dr == 1'b1)
        shift_reg <= mbist_reg;
      else if(shift_dr == 1'b1)
        shift_reg <= {tdi , shift_reg[3:1]};
    end
  end

  //tdo flop
  always @ (negedge tck, negedge trst_b) begin
    if(!trst_b)
      mbist_tdo <= 1'b0;
    else
      mbist_tdo <= shift_reg[0];
  end

  //mbist register
  always @ (negedge tck, negedge trst_b) begin
    if(!trst_b) begin
      mbist_start <= 1'b0;
      mbist_enable <= 1'b0;
    end
    else begin
      if((mbist_reg_en == 1'b1) && (update_dr == 1'b1)) begin
        mbist_start <= shift_reg[1];
        mbist_enable <= shift_reg[0];
      end
    end
  end
endmodule
```

```
//--TDO SELECT

module tdo_select(inst_reg_sel, idcode_reg_sel, mbist_reg_sel,
  inst_tdo, mbist_tdo, idcode_tdo, tdo);

  input inst_reg_sel, idcode_reg_sel, mbist_reg_sel, inst_tdo, mbist_tdo, idcode_tdo;
  output reg tdo;


  always @ (inst_reg_sel, idcode_reg_sel, mbist_reg_sel, inst_tdo, idcode_tdo, mbist_tdo)
begin
    if(inst_reg_sel == 1'b1)
      tdo <= inst_tdo;
    else if(idcode_reg_sel == 1'b1)
      tdo <= idcode_tdo;
    else if(mbist_reg_sel == 1'b1)
      tdo <= mbist_tdo;
  end
endmodule
```

Lab Assignment #8b

Guideline

This lab can be done with a partner.

Objective

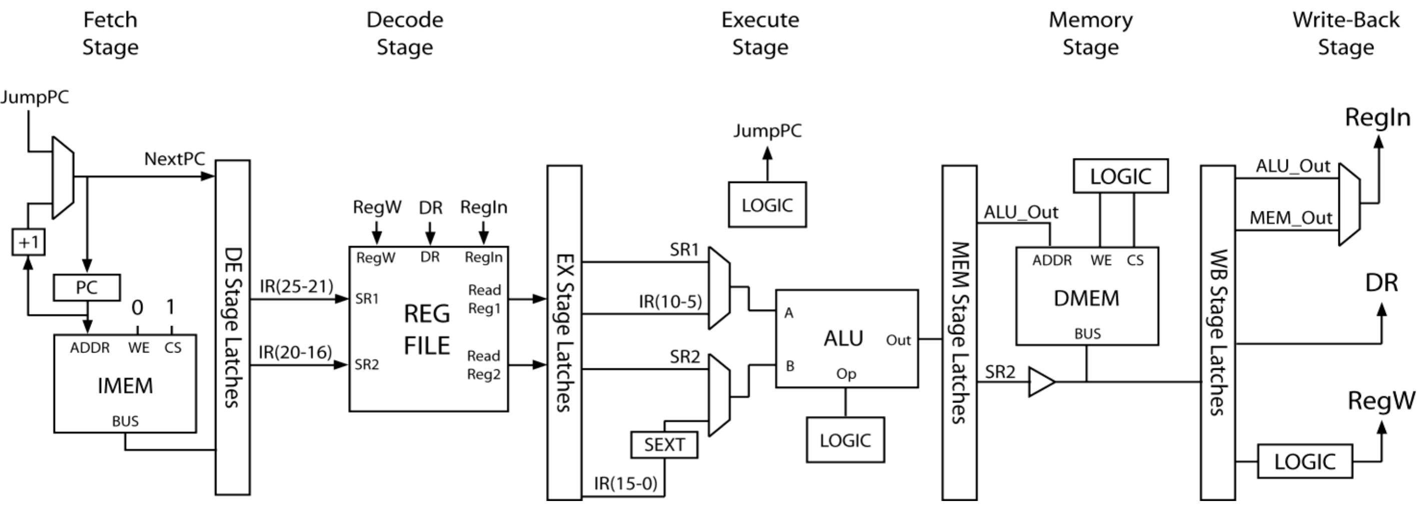
1. Implement and synthesis a 5 stage pipelined MIPS processor from the MIPS model used in Lab 7
2. Become familiar with pipeline design and dependencies

Summary of tasks

You will implement a 5 stage pipelined MIPS processor in Verilog using the model used in Lab 7. Your processor should support all the instruction in table 1. First simulate your design in Modelsim using the testbench provided by TAs. Then synthesis your design and run a simple program on your MIPS processor.

Description

Modern microprocessors employ pipelining to improve instruction throughput. Consider a 5-stage pipeline consisting of fetch, decode and read registers, execute, memory access, and register write-back stages. During the first stage, an instruction is fetched from the instruction memory. During the second stage, the fetched instruction is decoded. The operand registers are also read during this stage. During the 3rd stage, the arithmetic or logic operation is performed on the register data read during the 2nd stage. During the 4th stage, in load/store instructions, data memory is read/written into memory. Arithmetic instructions do not perform any operation during this stage. During the 5th stage, arithmetic instructions write the results to the destination register.



Between each stage of the pipeline, flip-flops (called latches) store the state of the current instruction being processed. In the fetch stage, PC, NextPC, the instruction fetched from the Instruction Memory and a Valid bit are stored in the Decode Stage Latch. In the Decode stage, all the control signals needed to execute the instruction are generated and latched into the Execute state latches. The next stages use the generated control signals to perform ALU operations, memory accesses, and write back. Every stage latches a **Valid bit** to represent a valid instruction in that stage pipeline. Stalls are implemented by setting this valid bit to zero. A zero propagated to the next stage, is called a bubble or NOP. **Please read the "dependencies" document found under in the Lab Documentation folder for detailed information about stalls and dependencies. You are not required to implement data forwarding.**

Problem

Design a pipelined implementation of the MIPS design in Figure 9-8. Write Verilog code, synthesize it for an FPGA target, and implement it on an FPGA prototyping board. Assume that each stage takes one clock cycle. While implementing on the prototyping board, use a 50 MHz clock.

Assume that instruction memory access and data memory access takes only one cycle. Instruction and data memories need to be separated (or must have 2 ports) in order to allow simultaneous access from the 1st stage and 4th stage.

An instruction can read the operands in 2nd stage from the register file, as long as there are no dependencies with an incomplete instruction (ahead of it in the pipeline**). If such a dependency exists, the current instruction in decode stage must wait until the register data is ready.** Each instruction should test for dependencies with previous instructions. This can be done by comparing source registers of the current instruction with destination registers of the incomplete instructions ahead of the current instruction. When there is a dependency stall in decode, the fetch stage must stall as well because it is waiting the previous instruction to be decoded.

It is important to note that branch and jump instructions also need special logic to insert bubbles and update the PC. Both jump and branch can be handled in the Execute stage for simplicity. Once the instruction is decoded as a branch or jump instruction, the fetch stage inserts a bubble by setting the valid bit to zero. This is done because the next instruction being fetched needs to know the new PC determined by the branch/jump instruction. In the next cycle, the branch/jump instruction enters the Execute stage where the new PC can be evaluated for a branch instruction based on ALU result. The new PC of a jump instruction is determined directly from the instruction encoding. This new PC is stored into the PC register in the Fetch stage as seen in the figure above. Now the pipeline can continue to fetch and execute from the new PC.

The register file is written into during stage five and read from during stage two. A reasonable assumption to make is that the write is performed during the first half of the cycle and the read is performed during the second half of the cycle. Assume that data written into the destination register during the first half of a cycle can be read by another instruction during the second half of the same cycle. Note that you only need to check for dependencies in the Execute and Memory stages because the register is written back in during the first half of the Write-Back stage, and read during the second half of the Decode stage.

Your Tasks

1. You will modify your MIPS processor from Lab 7 to design a pipelined implementation of the MIPS instructions in table 1. Use the Verilog test-IO package to initialize instruction and data memory. To test your processor, you will be given a testbench by the TAs.

2. Once you have verified the correct functionality of your design, synthesis your pipelined MIPS processor and run the following instructions 127 times. Map the bottom 8 bits of register $1 to LEDS[0-7] to verify the correctness of your program. Is this faster than your non-pipelined implementation in Lab 7?

```
addi $1, $1, #1
addi $2, $2, #1
add $1, $1, $2
```

3. How many cycles does it take to execute *N* instructions with no dependencies?

Table 1

Instructions
addi
andi
ori
slt
sll
srl
lw
sw
beq
bne
jump

Submission details

Submit the following things on Canvas:

- All Verilog code (modified MIPS and testbenches/dofiles)
- MIPS assembly program
- Instruction text file containing the machine code of your program
- Bit file and UCF file, if any

Checkout details

The following things will be checked during check-out:

- Your modifications to the code and the testbench.
- Correct functionality of the program on the board and simulation.

Lab Assignment #9

Guideline

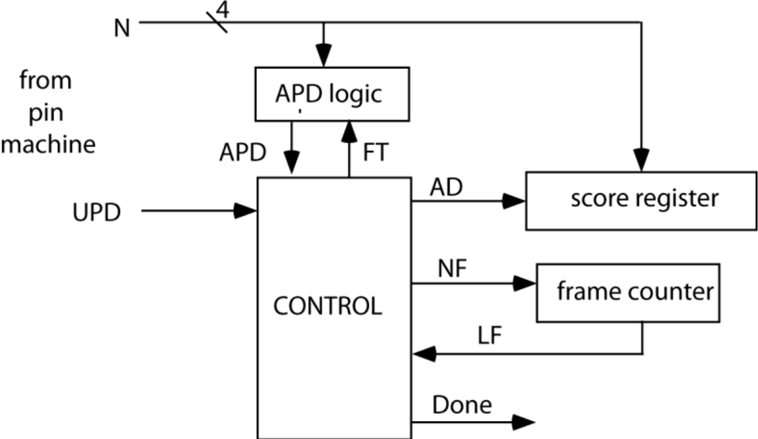
This lab is to be done with a partner.

Objective

Design a digital system to keep score for a bowling game. The score should be displayed on a 10-bit register in BCD form rather than in binary. You need to use a testbench to test your design in Modelsim. Also, you are not required to implement your code on the board.

Problem: Bowling Score Keeper

The digital system shown below will be used to keep score for a bowling game. The score keeping system will score the game according to the following (regular) rules of bowling: A game of bowling is divided into ten frames. During each frame, the player gets two tries to knock down all of the bowling pins. At the beginning of a frame, ten pins are set up. If the bowler knocks all ten pins down on his or her first throw, then the frame is scored as a *strike*. If some (or all) of the pins remain standing after the first throw, the bowler gets a second try. If the bowler knocks down all of the pins on the second try, the frame is scored as a *spare*. Otherwise, the frame is scored as the total number of pins knocked down during that frame.



The total score for a game is the sum of the number of pins knocked down plus bonuses for scoring strikes and spares. A strike is worth 10 points (for knocking down all ten pins) plus the number of pins knocked down on the next two *throws* (not frames). A spare is worth 10 points (for knocking down ten pins) plus the number of pins knocked down on the next throw. If the bowler gets a spare on the tenth frame, then he/she gets one more throw. The number of pins knocked down from this extra throw are added to the current score to get the final score. If the bowler gets a strike on the last frame, then he/she gets two more throws, and the number of pins knocked down are added to the score. If the bowler gets a strike in frame 9 and 10, then he/she also gets two more throws, but the score from the first bonus throw is added into the total *twice* (once for the strike in frame 9, once for the strike in frame 10), and the second bonus throw is added in once. The maximum score for a perfect game (all strikes) is 300.

An example of bowling game scoring follows:

Frame	First Throw	Second Throw	Result	Score
1	3	4	7	7
2	5	5	spare	7 + 10 = 17
3	7	1	8	17 + 7 (bonus for spare in 2) + 8 = 32
...	...	...	...	87
9	10	-	strike	87 + 10 = 97
10	10	-	strike	97 + 10 (for this throw) + 10 (bonus for strike in 9)
-	6	3	-	117 + 6 (bonus for strike in 9) + 6 (bonus for strike in 10) + 3 (bonus for strike in 10) = 132

For additional resources with respect to keeping the bowling score, please visit the following websites which provides a java applet for keeping score.

http://www.bowlinggenius.com/

The score keeping system has the form shown above diagram. The control network has three inputs: *APD* (All Pins Down), *LF* (Last Frame), and *UPD* (update). *APD* is 1 if the bowler has knocked all ten pins down (in either one or two throws). *LF* is 1 if the frame counter is in state 9 (frame 10). *UPD* is a signal to the network that causes it to update the score. *UPD* is 1 for exactly one clock cycle after every throw the bowler makes. There are many clock cycles between updates.

The control network has four outputs: *AD, NF, FT*, and *Done*. *N* represents the number of pins knocked down on the current throw. If *AD* is 1, *N* will be added to the score register on the rising edge of the next clock. If *NF* is 1, the frame counter will increment on the rising edge of the next clock. *FT* is 1 when the first throw in a frame is made. *Done* should be set to 1 when all ten frames and bonus throws, if applicable, are complete.

Use a 10-bit score register and keep the score in BCD form rather than in binary. That is, a score of 197 would be represented as 01 1001 0111 . When *ADD* = 1 and the register is clocked, *N* should be added to the register. *N* is a 4-bit binary number in the range 0 through 10. Use a 4-bit BCD counter module for the middle BCD digit. Note that in the lower four bits, you will add a binary number to a BCD digit to give a BCD digit and a carry.

Creating a testbench:

Simulate your code with the following test scenario (test bench) as shown in the table below:

X	X	7 /	9 /	9 -	X	7 /	X	9 /	X X 7	
27	47	66	85	94	114	134	154	174	201	201

Frame ->	1	2	3	4	5	6	7	8	9	10	Bonus	Total
First Throw	10	10	7	9	9	10	7	10	9	10	10	Score
Second Throw	-	-	3	1	0		3		1		7	=
Score	27	47	66	85	94	114	134	154	174	201		201

Create your own additional test benches. We will be testing with other test benches also. So you need to make sure all corner cases are covered.

You should synthesize the code and implement it on the Xilinx board. Display the score in the 7-segment LED's. Also indicate the output control signals (AD, NF, FT and DONE) on the LED's. You can use switches to input N.

Submission Details

You must submit the Verilog source code, testbench and waveforms (showing the above test scenario) in Canvas. If you have implemented the design on the board, then we also require you to submit the place and route report as well.

Checkout Details

1. Annotated simulation waveform showing all the pins down in every throw.
2. All source code.

Lab Assignment #10

Guideline

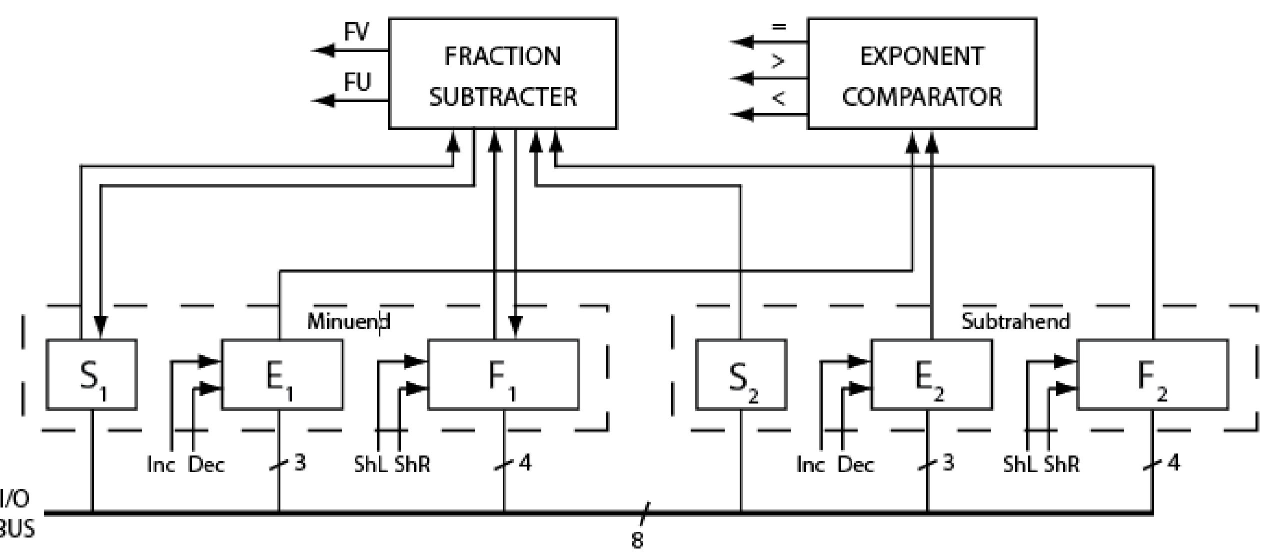
This lab is to be done with a partner.

Objective

There are two problems in this lab. First, you have to design a floating-point subtractor. Assume that the inputs to the subtractor are properly normalized, and the result should be properly normalized as well. The second problem is to create a floating-point arithmetic unit, which could do add, subtract, and multiply operations. For both part 1 and 2, you have to create .do files for demonstrating the correctness on ModelSim. In addition, you also have to synthesize and run your part 2 on FPGA for Lab Checkout.

Problem 1: Floating-Point Subtractor

The Block diagram shown below indicates the design of a floating-point subtractor. The fractions are 4 bits, the exponents are 3 bits, and the sign is 1 bit. The floating-point format is the IEEE 3-biased FP format.



The floating-point subtraction is the same as the floating-point addition, except that we must subtract the fractions instead of adding them. The rest of the steps remain the same. (Your subtractor should be able to handle the special case of 0. It is required to deal with infinity, unnormalized, and not-a-number formats.)

Here is an example of floating-point subtraction. You could use it to test your program.

$F_1 * 2^{E1} = 0.1 * 2^{-1}$ (0.25 in decimal, 00010000 in IEEE 3-biased FP format)

$F_2 * 2^{E2} = 0.1 * 2^{-2}$ (0.125 in decimal, 00000000 in IEEE 3-biased FP format)

$$(0.1 * 2^{-1}) - (0.1 * 2^{-2})$$

$$= (0.1 * 2^{-1}) - (0.01 * 2^{-1})$$

$$= 0.01 * 2^{-1}$$ (00000000 in IEEE 3-biased FP format)

Two operands are in 8 bits IEEE 3-biased FP format, and your result should be in this format as well. In addition, if there is an overflow or underflow, your program should be able to detect it and turn on the FV or FU bit. This problem is only required to be implemented in simulation and demonstrated on ModelSim.

Hint: The floating-point adder Verilog module has already been provided in the textbook. Please use it as a reference code.

Problem 2: Floating-Point Arithmetic Unit

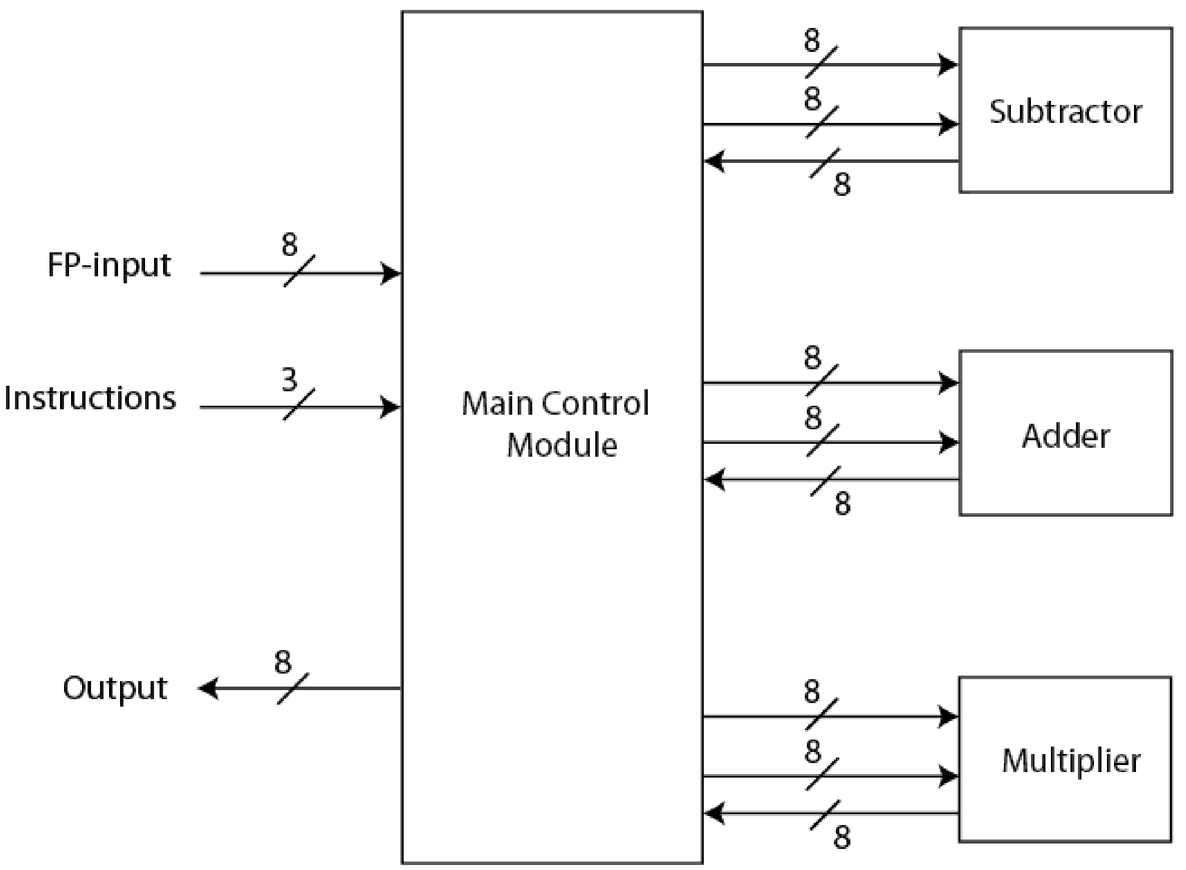
Design a floating-point arithmetic unit. Each floating-point number should have a 4-bit fraction, 3-bit exponent, and 1-bit sign. The unit should be able to accept the following floating-point instructions:

001 FPL: Load floating-point accumulator (8 bit)
010 FPA: Add floating-point operand to accumulator
011 FBS: Subtract floating-point operand from accumulator
100 FPM: Multiply accumulator by floating-point operand
110 RF: Refresh the arithmetic unit to erase the existing accumulator data

The result of each operation (4-bit fraction, 3-bit exponent, 1-bit sign) should be in the floating-point accumulator. All output should be properly normalized. The accumulator should always be displayed as hex digits on 7 segment LEDs. Use an LED to indicate an overflow or underflow.

To select one of the six operations, you have to use the 3 buttons on our Spartan 3E FPGA. To load a new accumulator into the FPGA, you should use the 8-switch on Spartan 3E to display the accumulator and press the FPL operation buttons (001). If you want to do a subtraction, you have to display the operand by 8-switch and press the subtraction buttons (011). After that, the corresponding result (2 hex digits) should be displayed on the 7 segment LEDs.

The following diagram shows the connection between main control module and FP operation modules. The input accumulator, operands, and operation selections should be put into the main control first. Main control will pass them into each operator and collect the result. Then, based on the operation selection, it will display the corresponding result on 7 segment LEDs. You could directly re-use the subtraction module from Problem 1. Addition and Multiplication modules have already been written in the textbook with Verilog code. Your .do file of this problem should be able to show the correctness of all possible operations.



Different from Problem 1, you have to demo your program on both ModelSim and Xilinx FPGA when you do the checkout with TA.

Submission Details

You must submit the Verilog source code, .do files (showing the above test scenario) in Canvas. If you have implemented the design on the board, then we also require you to submit the place and route report as well.

Checkout Details

I. Annotated simulation waveform/list showing all the pins down in every throw.
II. All source code.

APPENDIX

This section contains the lab documents which were created but are not currently being used. Some time in future these may be needed.

Lab Assignment – ARM Processor

Guideline

This lab is to be done individually. Each person does his/her own assignment and turns it in.

Introduction

In this lab, you will implement a basic ARM processor. ARM (Advanced RISC Machines) microprocessors are very popular in the embedded systems market. The ARM is a 32-bit RISC architecture and features the main characteristics of such systems:

- Large number of registers
- Load-store model of data processing
- Small number of addressing modes
- Uniform fixed length instructions

In addition, ARM provides some additional features:

- A Shifter on one input to the ALU
- Conditional execution of instructions

Pipeline: The ARM pipeline also consists of basically five stages: Fetch, Decode, Execute, MemoryAccess, RegisterWrite. This is very similar to the MIPS pipeline.

Registers: There are 16 user registers (R0 through R15) and a status register (PSR), each 32 bits in size, in an ARM machine. R0 through R12 are general purpose registers, while R13, R14 and R15 are special purpose. R13 is the stack pointer (SP), R14 is the link register (LR) and R15 is the program counter (PC). The status register (PSR) contains the condition code flags (like negative, zero, carry and overflow). The PC stores the address of the instruction to be executed next. The LR stores the address of the instruction to return to, after completing a branch to a subroutine.

Instructions: There are several kinds of instructions in the ARM ISA, ranging from data movement instructions to logical & arithmetic instructions to flow control instructions. But for this lab you will be implementing only a subset of the instructions of the ARM ISA.

Objective

1. To design the given subset of an ARM ISA in Verilog and implement it on the lab board. [You are required to do both simulation and synthesis for this lab.]
2. To convert the given assembly program (which uses the given subset of instructions) into its machine code, initialize the processor memory with the program and execute the program on the processor on the board.

Background

Please briefly read through the ARM architecture manual and/or the ARM instruction set manual before proceeding, to get an idea of the ARM architecture and instructions. One such document is available on Canvas under “Labs -> Other documents/files”.

Instructions to be implemented

	Branch
1	B
2	BL
	Arithmetic & Shift
3	ADD
4	ADD8
5	SADD
6	SSUB
	Comparisons
7	TEQ
	Movement
8	MOV
9	LDR
10	STR
	Others
11	SWP
12	RBIT
13	CLZ

Conditional Execution

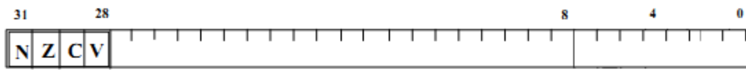
The ARM instruction set is unique in that it allows the conditional execution of all the instructions (not just branch instructions). All instructions contain a "condition field" which determines whether the CPU will execute them. The general format of an ARM machine-language instruction is:

<Condition> <Opcode> <Operands>

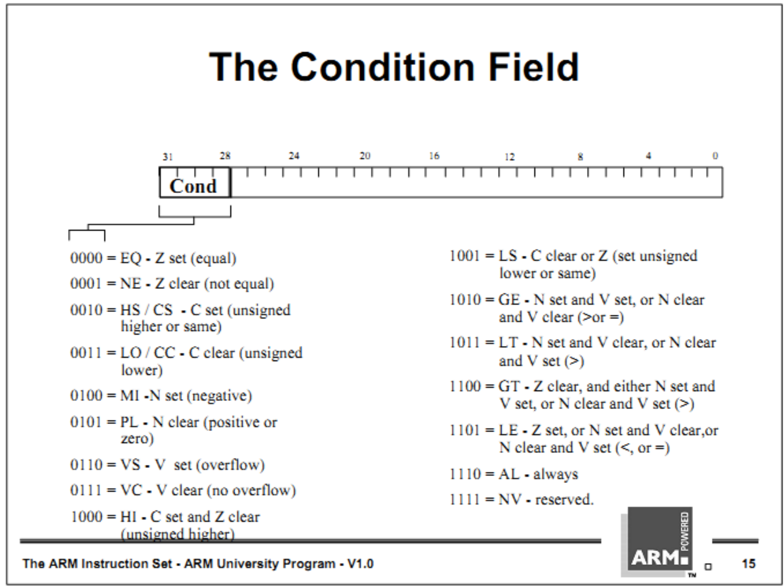
The following figure shows the condition fields used by the ARM instruction set. You will only have to implement three conditions: EQ, NE, AL for each instruction.

So, basically when executing each instruction, the processor checks for the flags in the PSR against the condition field in the instruction. For example, if PSR has the ZERO flag set, then an instruction with a condition "NE" will not execute.

The following image gives the location of the flag bits in the PSR register.

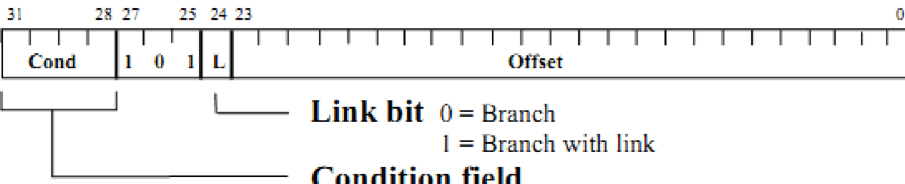


Condition Code Flags
N = **N**egative result from ALU flag.
Z = **Z**ero result from ALU flag.
C = ALU operation **C**arried out
V = ALU operation o**V**erflowed



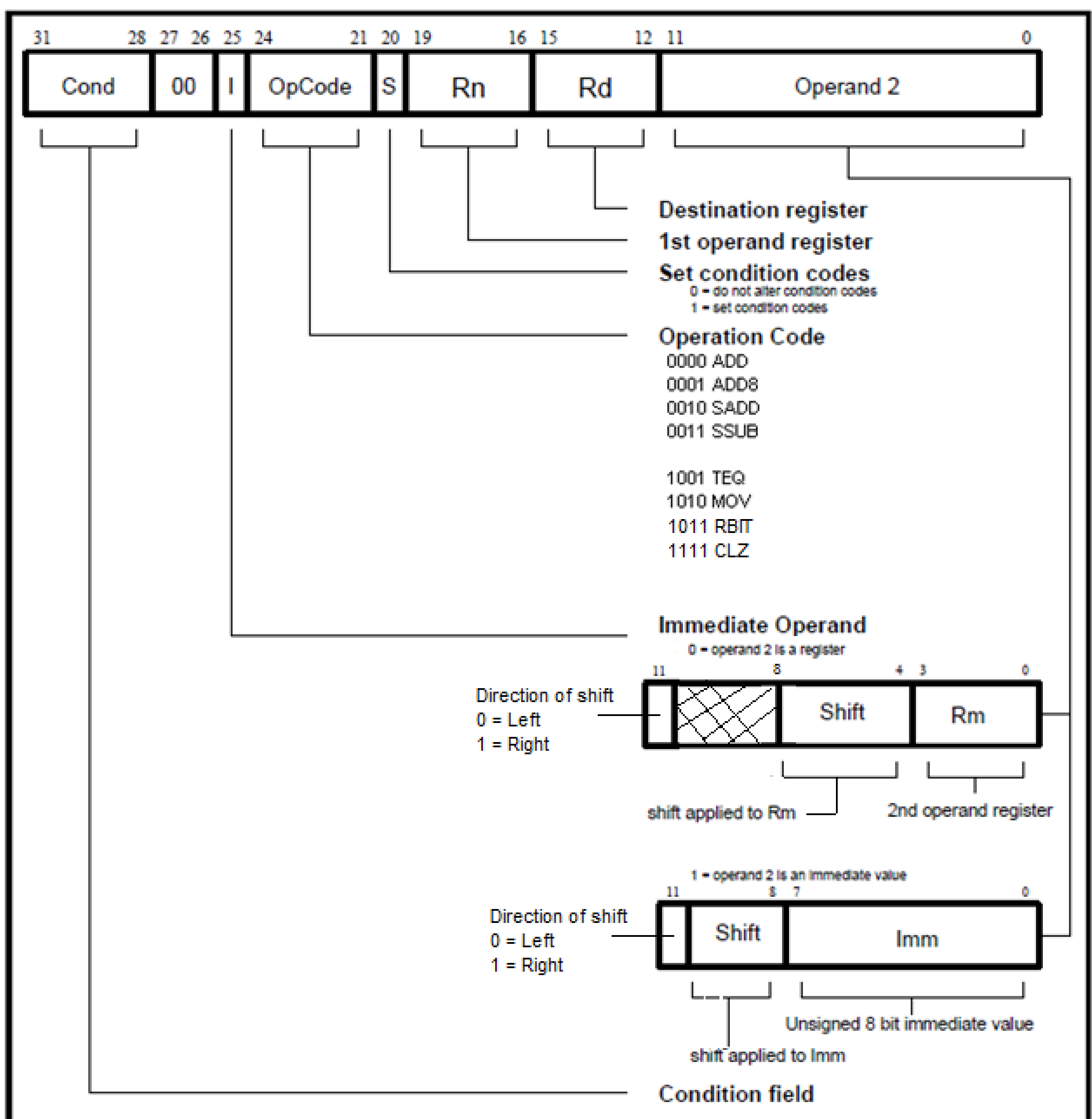
Instruction Details

1. B (Branch) and BL (Branch and Link)



B		
	Format	B{cond} label
	Description	Branch to label; This is used for jumps.
	Operation	PC = PC + sign_ext(Offset *4)
BL		
	Format	BL{cond} label
	Description	Branch to label, save PC+4 in link register; This is used for subroutine calls.
	Operation	PC = PC + sign_extend(Offset *4) LR = PC + 4

2. ADD, ADD8, SADD, SSUB, MOV, TEQ, RBIT, CLZ



Another interesting feature in the ARM architecture is the way shifts are handled. The ARM has a shifter on one of the input paths to the ALU. There are no dedicated shift instructions. Shifting is done by providing a shift in the second operand. You will implement logical left shift (LSL) and logical right shift (LSR). For example, the following ADD operation shifts R1 left by 2, adds it to R2 and stores the result in R0:

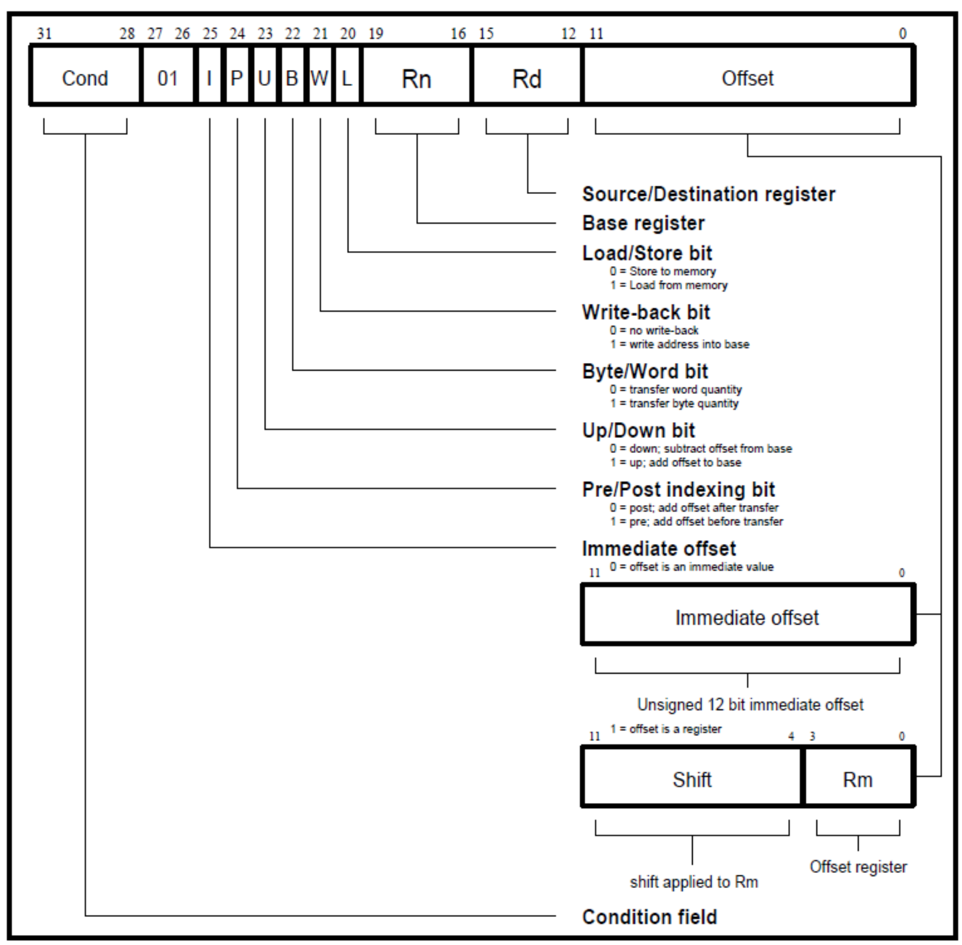
ADD R0, R2, R1, LSL #2

The shift can be specified either as an immediate value or in a register. ***Implementing the shift feature is not mandatory, but will earn you a bonus of 5%.*** *You have to implement the shift only for ADD and MOV instructions.*

ADD		
	Format	ADD{cond}{S} Rd,Rn,Op2
	Description	Add two operands and store results in a register
	Operation	Rd = Rn+Op2
ADD8		
	Format	ADD8{cond}{S} Rd, Rn, Op2
	Description	This perform byte-wise addition as illustrated below
	Operation	Rd[31:24] = Rn[31:24]+ Op2[31:24] Rd[23:16] = Rn [23:16]+ Op2[23:16] Rd[15:8] = Rn [15:8]+ Op2[15:8] Rd[7:0] = Rn [7:0]+ Op2[7:0]
SADD		
	Format	SADD{cond}{S} Rd, Rn, Op2
	Description	Saturating addition
	Operation	If ((Rn + Op2) > $2^{32} – 1$), then Rd = $2^{32} – 1$; else Rd = Rn + Op2;
SSUB		
	Format	SSUB{cond}{S} Rd, Rn, Op2
	Description	Saturating Subtraction
	Operation	if ((Rn – Op2) < 0) then Rd = 0; else Rd = Rn – Op2;
MOV		
	Format	MOV{cond}{S} Rd, Op2 (Note that Operand 1 is not used here)
	Description	Move data from Op2 to Rd
	Operation	Rd = Op2
TEQ		
	Format	TEQ{cond}{S} Rd, Op2 (Note that Operand 1 is not used here)
	Description	Compare Rd and Op2
	Operation	PSR set based on (Rd == Op2)
RBIT		
	Format	RBIT Rd, Op2 (Note that Operand 1 is not used here)
	Description	Reverse the bits in a word
	Operation	for(i=0;i<32;i++){ Rd[i]=Op2[31-i]}

CLZ		
	Format	CLZ Rd, Op2 (Note that Operand 1 is not used here)
	Description	Count the leading zeros in a word
	Operation	Rd = number of leading zeros in Op2;

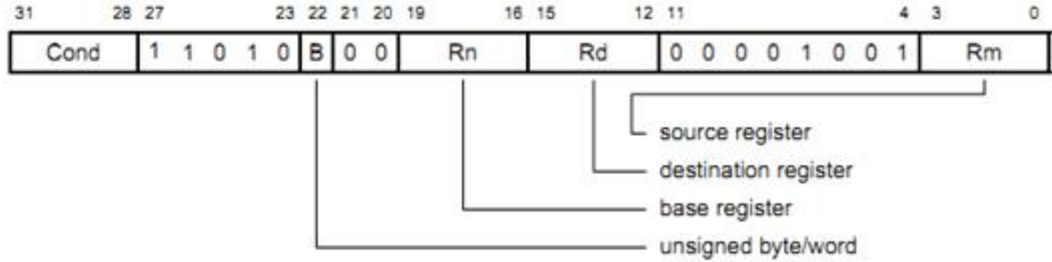
3. LDR and STR



LDR		
	Format	LDR{cond} Rd, [Rn]
	Description	Load value from memory into Rd. The memory address is contained in a register.
	Operation	Rd = mem[Rn]
STR		
	Format	STR{cond} Rd, [Rn]
	Description	Store value in Rd into memory. The memory address is contained in a register.
	Operation	mem[Rn] = Rd

You don't have to implement shift in the Load and Store instructions. Also, you don't have to implement the functionality of the Write Back bit. Implementing the functionality of the Byte/Word bit, Post/Pre indexing bit and Up/Down bit is not mandatory (and if you don't, the Offset part of the instruction is not used), but will earn you 5% bonus. Look into the ARM manual on Canvas for details of these.

4. SWP



SWP		
	Format	SWP{cond} Rd, Rm, [Rn]
	Description	Swap contents of memory location pointed to by Rn with value in Rd.

	Operation	temp = mem[Rn];
		mem[Rn] = Rm;
		Rd = temp;
		This implements actual SWAP if you make Rd=Rm

You don't have to implement the functionality of the B bit

Test program

You will test your Verilog model of the ARM processor using a test program that is provided (the assembly language version of the test program is available on Canvas). The test program uses three switches and two buttons from the board to perform certain operations and show certain results on the 7 segment display. Table 2 summarizes the functions and display modes of the test program.

SW 4	**SW 3**	**SW 2**	**SW 1**	**Task**	**BTN1**	**BTN0**	**Value to Display on 7 segment**
0	0	0	0	ADD R2, R4, R5 ADD R3, R2, R4, LSL #4	0	0	lower 16 bits of R2
					0	1	upper 16 bits of R2
					1	0	lower 16 bits of R3
					1	1	upper 16 bits of R3
0	0	0	1	ADD8 R2, R4, R5	0	0	lower 16 bits of R2
					0	1	upper 16 bits of R2
0	0	1	0	SADD R2, R4, R5 SADD R3, R2, R4	0	0	lower 16 bits of R2
					0	1	upper 16 bits of R2
					1	0	lower 16 bits of R3
					1	1	upper 16 bits of R3
0	0	1	1	SSUB R2, R5, R4 SSUB R3, R5, R4, LSR #1	0	0	lower 16 bits of R2
					0	1	upper 16 bits of R2
					1	0	lower 16 bits of R3
					1	1	upper 16 bits of R3
0	1	0	0	MOV R2, R4	0	0	lower 16 bits of R2
					0	1	upper 16 bits of R2

<table>
<tr><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">RBIT R2, R4</td><td>0</td><td>0</td><td>lower 16 bits of R2</td></tr>
<tr><td>0</td><td>1</td><td>upper 16 bits of R2</td></tr>
<tr><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">CLZ R3, R4</td><td>0</td><td>0</td><td>lower 16 bits of R3</td></tr>
<tr><td>0</td><td>1</td><td>upper 16 bits of R3</td></tr>
<tr><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">MOV R0, #0
ADD R0, R0, #1
SWP R2, R2, R0</td><td>0</td><td>0</td><td>lower 16 bits of R2</td></tr>
<tr><td>0</td><td>1</td><td>upper 16 bits of R2</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">STR R0, [R5]
LDR R2, [R5]</td><td>0</td><td>0</td><td>lower 16 bits of R2</td></tr>
<tr><td>0</td><td>1</td><td>upper 16 bits of R2</td></tr>
</table>

Four switches from the board will be used to load a value into register R1. The assembly program will be running in a loop and will be constantly looking at the value in R1. When the value in R1 changes, the program will jump to a subroutine that performs the Task indicated in the table above. The program will use BLEQ to jump to subroutines, so you should make sure this instruction works perfectly. At the end of the subroutine, the program will continue looping, waiting for a change in R1. While the program is looping, you should be able to press BTN1 and BTN0 in the appropriate combinations to display the value in the result registers R2 or R3 on the 7 segment display. The constants that are loaded into R4 and R5 for the computations should be chosen carefully to prove the functionality of the instructions you implement.

Your tasks for this lab

7. Write a Verilog model for an ARM processor capable of executing the instructions listed in this document.
8. Convert the assembly language program into machine language. Then add these machine codes to the “memory” block in your design (ie. initialize the memory of ARM with the test program).
9. Add a 7-segment display” block in your design (you can reuse this block from the previous labs). This block will take BTN1 and BTN0 as inputs and then displays the upper or lower bytes of R2 or R3 on the 7-segment display as required.
10. Modify the “register” block in your design to expose certain registers.
    a. Map any four switches to the lower 4 bits of R1.
    b. Expose R2 and R3 to the “7-segment display” block.
11. Simulate the model using ModelSim and observe the outputs on the waveforms.
12. Synthesize the design and implement it on the board.

Submission details

Submit the following things on Canvas:

- Verilog codes
- Machine code of the assembly program
- Bit file and UCF file, if any

Checkout details

The following things will be checked during check-out:

- Correct functionality of the program on the board
- Your Verilog code

Some helpful explanations

1. You may not have the PSR implemented as a part of the register file. It can be just a signal/variable in the ARM program.

2. The registers R13,R14 and R15 are the same as SP, LR and PC. You can keep these registers outside the register file, in which case, the register file will only have 13 registers - R0 through R12.

3. The condition bits are different from the PSR flags. The condition bits (4 in number) are a part of the op-code of each instruction. Their values will either be 0000 (for EQ), 0001 (for NE) or 1111 (for AL).

The PSR flags are the most significant 4 bits of the register called PSR. They are status flags and are by instructions. Also, they are checked by the instructions depending on the condition bits in the instruction. There are four flags: N (negative), Z (zero), C (carry), V(overflow).

4. The condition flags work like this:

Let us say that the assembly instruction is : ADDEQ R1, R2, R3

The machine code for this instruction will have the condition bits (bits 31 thru 28) as "0000". When the processor encounters this instruction, it sees the condition bits. Since they are "0000", it needs to check for an equality condition. So, it checks the PSR "Z" bit. If the "Z" bit is 1 (meaning that some instruction prior to this instruction resulted in a ZERO), then the processor executes the ADD instruction (R1 <= R2 + R3). If the "Z" bit is "0", then the processor does not execute the ADD instruction.

5. The "S" bit works like this:

Let us say that the assembly instruction is : ADDS R1, R2, R3

The machine code for this instruction will have the S bit set [Also, the condition bits (bits 31 thru 28) will be "1111" because the instruction is expected to run unconditionally]. When the processor encounters this instruction, it sees the condition bits. Since they are "1111", it decides that it has to execute the instruction without looking at the PSR. Then, it executed the ADD instruction (R1 <= R2 + R3). And then it sets the PSR accordingly (ie. if the ADD instruction results in a zero value, it sets the "z" bit in the PSR. if the ADD instruction results in an overflow, it sets the "V" bit in the PSR. and so on).

6. While choosing the values in the test program, use the values that actually test the behavior of an instruction. For example, for SADD, don’t use smaller numbers whose sum does not overflow. Use large numbers. Since you can not directly move very large numbers into registers, add some instructions in the test code. For example, you can move 0x01 in a register and apply RBIT instruction on it. This will make its magnitude huge.

7. For branch instructions, you need to consider your arguments to be signed. For all other instructions, you may use unsigned numbers to operate on. Specifically, for the SADD and the SSUB instructions, the operands should be assumed to be UNSIGNED. So, there are no negative numbers. The bounds are 0 to ((2^32) - 1).

8. You may choose not to simulate in this lab. If your code works fine on the board, you will get full credit. But if you want partial credit for something that does not work, please be ready with simulation waveforms during checkout. Also, if you plan to do simulation only, then you may get rid of the seven segment display block in your design.

9. This lab is slightly open ended. Please make sure you understand what is required and what is not. Do not implement any unnecessary features. Do not get into managing error conditions. Keep it simple. You can start from the MIPS code from Lab 7a and modify it by adding some extra states. You may modify the given test program according to your needs.

10. Many ARM features have been modified for the purpose for creating this lab. For example, it uses different opcodes from an actual ARM processor. Please follow the descriptions used in this document. You can use other documents to help improve understanding of your concepts.

11. You may choose not to increment PC by 4 to move to the next instruction. You can increment it by 1, if that is easier. But it is important to understand that an actual ARM would always do PC=PC+4.

12. For the SWP instruction, you will swap the contents of a memory location with the contents of a register. Since the memory contains your instructions (machine level program code) and you don't want to corrupt a memory with some arbitrary data (from a register), you can use a memory address where there are no instructions. For example, if your instructions are upto address 105, increase the size of the memory to say 106 and in the SWP instruction, swap the contents of location 106 with a register's contents.