Lab 7.1

Lab 7.1

- In this lab, you will:
  - Learn to recognize common mistakes with compiler directives
  - Learn the difference between the *if* statement and the *ifdef* directive
  - Learn how to create reconfigurable models

Lab 7.1

- In directory ~/lab7.1, you will find a file named "syntax.v". It contains a model that initializes a memory with an incrementing value, then displays the sum of all values stored in the memory.
- Compile the file and fix any syntax errors you may find.
- Ensure that the sum is 136.

Lab 7.1

- In the same directory, you will find the following files:
  - rom.v Model of a generic 16xN-bits ROM
  - dump2rom.v Dump the content of 2 ROMs
  - rom1.dat Data file for ROM#1
  - rom2.dat Data file for ROM#2
- These files are similar to those used during a previous lab. However, the ROM uses `*define*'d symbols to determine the width, depth, and initialization file of the ROM instead of parameters.

Lab 7.1

- Simulate both source files, specifying the file "rom.v" first. Try a second time with the files specified in the opposite order. Why is the compilation order-dependent?
- Look at the content of "rom1.dat". Why is the reported content of the first ROM incorrect?
- ***Optional:*** Modify the ROM model to use parameters instead of `*define*'d symbols.
  - Make the necessary modifications to the ROM dumper as well.
  - Simulate to verify the correctness of the model.

Lab 7.1

- In the same directory, you will find the following files:
  - stack.v Model of a stack
  - teststck.v Stack tester
- Both the tester and the stack contain debug display statements that were left behind to help diagnose a problem. You must now get rid of them.
- One option would be to delete these statements outright. However, you may need them in the future. It is probably best to leave them in and use *if* statement to jump over them if the debug mode is not set.

Lab 7.1

- Edit the file "stack.v" and define a register named DEBUG, initialized to '0'.
- Embed all debugging display statements inside an *if* statement:

```
if (DEBUG) begin
   <display-statement>
end
```

- Run the simulation to verify that it now executes quietly.
- Edit the file "stack.v" again, and initialize the DEBUG register to '1'. Simulate again to ensure that all messages came back.

Lab 7.1

- Turning the debug mode on required editing a file. Using a `*define* symbol would allow you to define the DEBUG flag on the command line.
- If you plan on doing the optional part of this lab, make a backup copy of the file "stack.v" right now.
- Edit the file "stack.v" again, replacing the "if (DEBUG) ..." with *ifdef*/*endif* directives.
  - You can take out the entire *initial* block for the $*monitor* statement using:

```
`ifdef DEBUG
initial ...
`endif
```

Lab 7.1

- Run the simulation to verify that it now executes quietly again.
- Simulate again, this time defining the "DEBUG" symbol on the command line, to ensure that all messages came back.
  - If you use a compiled simulator with different commands for compilation and simulation, the *+define+DEBUG* option must be specified on the compilation command.

Lab 7.1: Optional

- The goal of this optional section is to measure the performance difference between *if* statements and *ifdef* directives.
- Edit the file "teststck.v" and add a "`repeat (50000)`" statement around the outer *for-loop* statement.
- Simulate the file "stack.v" with the debug messages turned off. Note the CPU time required.
- Repeat with your back-up copy of the file "stack.v" (which uses *if* statements). Compare the CPU time required.

Lab 7.2

Lab 7.2

- In this lab, you will:
  - Learn how to specify simulation files using the -y option
  - Learn how to specify simulation files using the -v option
  - Learn how to manage simulation configurations using manifest files and the -f option

Lab 7.2

- In directory ~/lab7.2, you will find the following files:
  - test_add.v Testbench for 8-bit adder
  - addbeh.v Behavioral 8-bit adder
  - addstr.v Structural 8-bit adder
  - faddbeh.v Behavioral full-adder
  - faddgate.v Gate-level full-adder
- There are three possible 8-bit adder model configurations that can be simulated:
  - #1: Behavioral
  - #2: 8-bit structural + 8 x behavioral full adder
  - #3: 8-bit structural + 8 x gate-level full adder + ? gates

Lab 7.2

- Try simulating all files using the command:

```
% ... *.v
```

- Each configuration of the 8-bit adder requires a different command line to invoke Verilog with the proper files.
- The format of the command line will be of the form:

```
% ... test_add.v <config>
```

where <config> is the sequence of command-line options that will use the appropriate configuration.

Lab 7.2

- The simplest way to specify a particular configuration is to explicitly list all the required filenames on the command line.
- To simulate configuration #1, the behavioral description of the 8-bit adder is required. This description is fully contained in the file "addbeh.v". Use the command:

```
% ... test_add.v addbeh.v
```

Lab 7.2

- Try to simulate the structural description of the 8-bit adder using the command:

```
% ... test_add.v addstr.v
```

- Verilog complains that it is missing module "FADD". The behavioral description of a full-adder can be added to the simulation by adding the filename "faddbeh.v" to the command line:

```
% ... test_add.v addstr.v faddbeh.v
```

Lab 7.2

- Configuration #3 calls for using the gate-level full-adder. Try to simulate this configuration by specifying the name of the file containing this model:

```
% ... test_add.v addstr.v faddgate.v
```

- Several different modules are now missing: the models for the gates used to implement the gate-level full-adder. The name of the files containing their model could be added to the command line. However, in a real-life design where hundreds of different gates are used, it would be impractical to have to specify the names of hundreds of files on the command line.

Lab 7.2

- If you look into the subdirectory "gates", you will see a collection of files that contain models for individual gates.
- The **-y** and **+libext** options can be used to instruct Verilog to scan a directory for files that contain the missing models.
- Simulate configuration #3 using the following command:

```
% ... test_add.v addstr.v faddgate.v \
          -y gates +libext+.v
```

Lab 7.2

- If you look into the file "gates/gates.v", you will see a collection of models for individual gates.
- The **-v** option can be used to instruct Verilog to scan a file that contains the missing models.
- Simulate configuration #3 using the following command:

```
% ... test_add.v addstr.v faddgate.v \
        -v gates/gates.v
```

- The difference between the **-y** and the **-v** option is that the former scans directories, the latter scans files.

Lab 7.2

- It quickly becomes an overwhelming task of remembering the command-line options that make up a particular configuration. Not to mention typing the same thing over and over.
- Command-line options can be put in a file; Verilog can then be instructed to read this file. The content of the file is interpreted as if it had been typed on the command line itself.
- Command-line option files are specified using the **-f** option.

Lab 7.2

- Command-line option files can be used to create configuration specifications, or *manifest* files.
- Create three manifest files "conf1.mft", "conf2.mft", and "conf3.mft", each specifying the required command-line options to simulate configurations #1, #2, and #3 of the 8-bit adder, respectively.
- Simulate the three configurations using the commands:

```
% ... test_add.v -f conf1.mft
% ... test_add.v -f conf2.mft
% ... test_add.v -f conf3.mft
```

Lab 7.2: Optional

- Go back through the previous steps of this lab and try various ordering of the command-line options. Does it make a difference?
- Modify the "conf3.mft" manifest file to use the **-v** option to locate the missing modules.
- What happens if you forget to specify the **-v** option in front of the file "gates/gates.v"?

Lab 7.2: Optional

- Create a subdirectory in the current working directory, then change your working directory to this new subdirectory.
- Try to simulate any of the configurations from your current location, referencing the manifest files in "..". Why isn't it working?

Lab 7.3

Lab 7.3

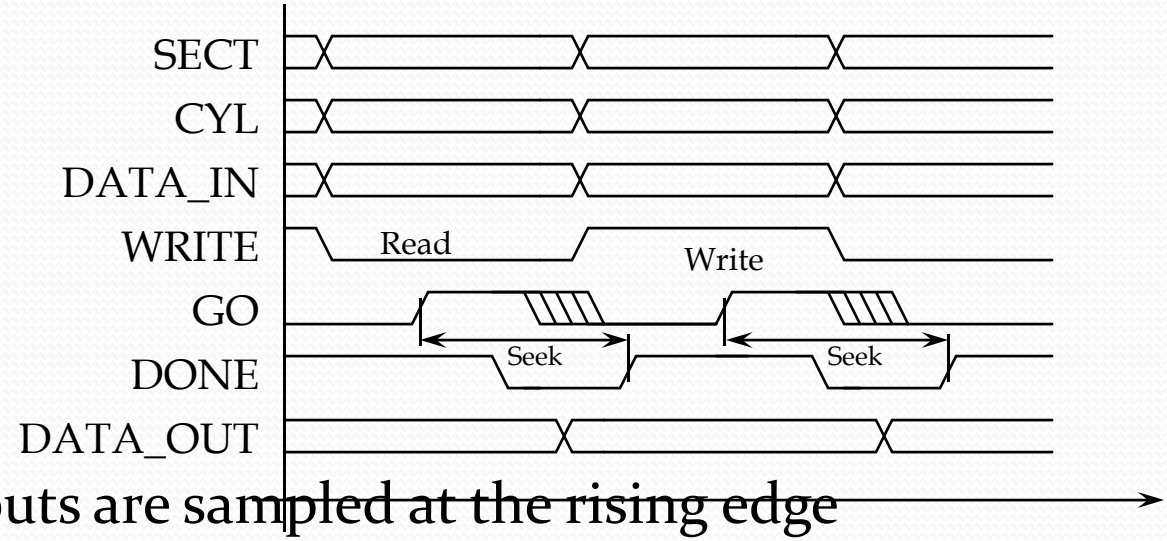
- In this lab, you will:
  - Learn how to write a behavioral testbench
  - Learn how to produce meaningful results

Lab 7.3

- In directory ~/lab 7.3, you will find a file named "drive.v". It contains a model of a disk drive that performs operations with variable seek times.
- The disk drive has 16 random-access 16-bit sectors on each of 8 cylinders. Sector 7 of each cylinder contains a read-only bitmap, generated at power-up, of good sectors in that cylinder.
- Marketing believes that this 240-byte disk drive will be a commercial success if the average seek time is less than 10ns with the maximum seek time less than 15 ns.

Lab 7.3

- The read and write operations, as well as the nominal seek time for each operation, are described by the following timing diagram:



- All inputs are sampled at the rising edge of "GO".

Lab 7.3

- Write a testbench that performs 25 read operations and 25 write operations, as fast as possible:

```
GO = 1'b1;
@ (negedge DONE);
GO = 1'b0;
@ (posedge DONE);
```

- For each operation, report seek time.
- At the end of the test sequence, report the maximum and average seek times
  - Will this disk be a commercial success?

Lab 7.3: Optional

- Modify your testbench to verify that:
  - Sector 7 of each cylinder is read-only
  - Good sectors are read-write
  - Bad sectors are indeed bad

Lab 7.4

Lab 7.4

- In this lab, you will:
  - Learn how to write simple bus-functional models
  - Use bus-functional models to test a device

Lab 7.4

- In directory ~/lab7.4, you will find a file named "testdffs.v". It contains a partially completed testbench for a scannable D flip-flop.
- The testbench module already contains the proper instantiation and signal declarations. It also contains a partially completed test procedure with the RESET and DATA_IN tasks.

Lab 7.4

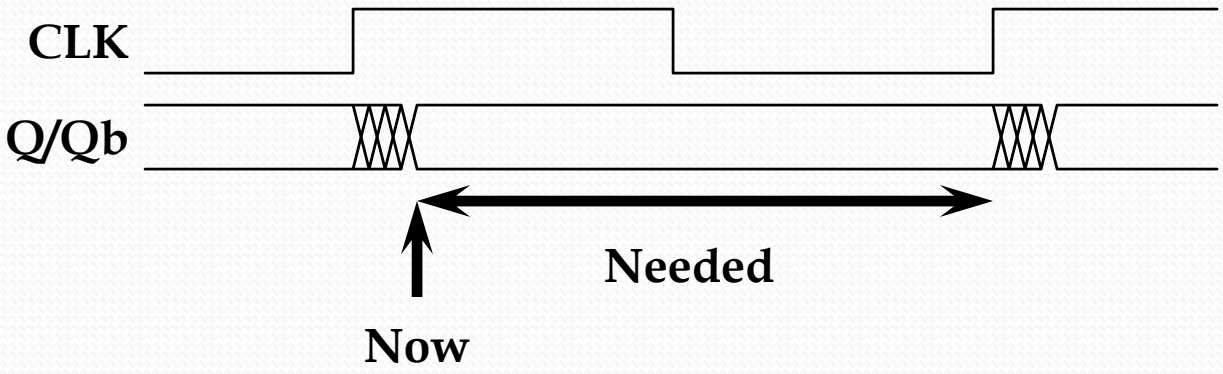
- Add an *always* block generating a free-running clock on register CLK with a period of 100 ns. The clock should start with the low half-period first (i.e. '0' for the first 50 ns).
- In the module, add a task abstracting the scan-in procedure. Cutting-and-pasting from the DATA_IN task is legal and strongly encouraged.
- Add the necessary calls to the new SCAN_IN task to fully test the D flip-flop.

Lab 7.4

- The directory also contains files "dffs1.v", "dffs2.v", and "dffs3.v". Each contain a model of a scanable D flip-flop.
- Using you testbench, determine which ones work and which ones do not by simulating each model, one at a time.
  - Looking at the source code of the flip-flop models to answer this question would **definitely** be cheating!

Lab 7.4 : Optional

- Currently, all test tasks check the Q and Qb outputs after the propagation delay time after the rising edge of the clock. However, the output of a flip-flop must remain stable from that point until the next rising edge of the clock.



Lab 7.4 : Optional

- Add an *always* block that checks that the Q and Qb outputs remain stable after the clock-to-Q delay, until the next rising edge of the clock.
- Simulate the three flip-flop models again. Are there any flip-flops that no longer meet their specification?