Lab 5.1

Lab 5.1

- In this lab, you will:
  - Learn how to instantiate modules using named and positional connection
  - Become familiarized with the port connection rules
  - Learn how to create hierarchy in a top-down fashion
  - Learn how to control *inout* ports

Lab 5.1

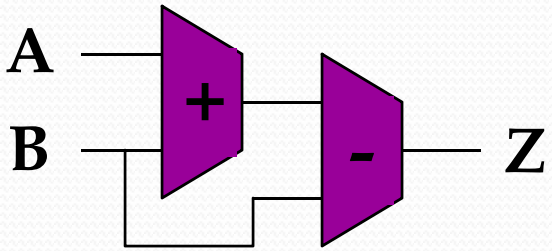
- In directory ~/lab5.1, you will find the following files:
  - adder.v 32-bit adder
  - sub.v 32-bit subtractor
  - noop.v Top-level model
- The top-level model contains two 32-bit regs ("A" and "B") and one 32-bit wire ("Z"). It also contains an *initial* block that provides various values to "A" and "B", then checks the value on "Z" against an expected value.

Lab 5.1

- Add the necessary statements to instantiate and connect the adder and subtractor to effectively create a "no-operation" structure solving the equation.

```
Z = (A + B) - B
```

- It can be performed using the following structure:



Lab 5.1

- Compile, then simulate your model to verify that your structural model is correct. The code provided will make the necessary checks.
- Verify that the correctness of your model does not depend on the order in which the filenames are specified or compiled.
- If you used positional connection specification, modify your instantiations to use named connection (or vice-versa).

Lab 5.1

- In the same directory, you will find a file named "rules.v". It contains two modules, one instantiating the other. The lower-level module has three pins: an input, an output, and an inout.
- The top-level module contains a declaration for a reg named "R" and a wire named "W".
- In the top-level module, try connecting the input pin of the lower-level module to the reg "R". Compile to verify if that is correct. Repeat with the wire "W". Repeat for the output and inout pins.

Lab 5.1

- Find a combination of connections to "R" and "W" that will not cause any syntax errors.
- In the lower-level module, try to declare the input pin as a reg. Compile to verify if it is acceptable. Repeat with the input pin declared as a wire. Repeat with the output and inout pins.
- Fill in the following table with the allowed pin connections (wire, reg, both):

Direction	Upper-level	Lower-level
Input		
Output		
Inout		

Lab 5.1

- In the same directory, you will find a file named "design.v". It contains an *always* block modeling a stack and an *initial* block testing the stack operation for correctness.
- Create a second module named "STACK" in another file and move all statements modeling the stack into this new module. Provide all necessary pins.
- Replace the original *always* block in the original module with an instantiation of the new STACK module. Simulate the new structure to verify that correctness is maintained.

Lab 5.1: Optional

- Modify the model of the stack to use a single *inout* pin instead of using separate pins for the new and top-of-stack values.
- Whenever "POP" is '1', the stack should drive the *inout* pin. Whenever "PUSH" is '1', the outside has ownership of the *inout* pin. Nobody should drive this pin when both control signals are '0'.
- Modify the top-level design module to match the new behavior.
- Simulate to verify the correctness of operation.

Lab 5.2

Lab 5.2

- In this lab, you will:
  - Learn how to use the '*timescale* directive
  - Witness the effects of different timescales on identical models
  - Get familiarized with the $*time* and $*realtime* tasks
  - Learn how to use the $*timeformat* task

Lab 5.2

- In directory ~/lab5.2, you will find the following files:
  - inverter.v Model of an inverter
  - testinv.v Inverter tester
- Compile and simulate both files at the same time, but specified in the following order

```
testinv.v inverter.v
```

  - Example (XL):

```
% verilog testinv.v inverter.v
```

- Repeat with the file specified in the opposite order

```
inverter.v testinv.v
```

- Why is the order suddenly significant?

Lab 5.2

- Add the following '*timescale* directive to the inverter model, then verify that the compilation order is no longer significant.

```
`timescale 1ns/1ns
```

- The inverter tester reports two pieces of information:
  - The rise delay
  - The fall delay

Lab 5.2

- Fill in the following table for various timescales on the tester and inverter modules:

Tester	Inverter	Rise	Fall
1ns / 1ns	1ns / 1ns		
1ns / 1ns	1ns / 100ps		
1ns / 1ns	1ns / 10ps		
1ns / 1ns	1ns / 1ps		
1ns / 1ns	1ps / 1ps		
1ns / 1ps	1ns / 100ps		
1ps / 1ps	1ns / 100ps		

Lab 5.2

- Edit the tester module and make the following modifications:
  - Add a $*timeformat* statement to the *initial* block.
  - Modify the $*write* statements to display the fall and rise delay values using the global time format (%t) rather than a floating-point format.
- Run the simulation with the following timescale pairs. Are the reported delays different from before?

Tester	**Inverter**
1ns / 1ns	**1ns / 100ps**
1ps / 1ps	**1ns / 100ps**

Lab 5.2

- Edit the tester module and make the following modifications:
  - Replace the calls to the $*realtime* task by calls to the $*time* task.
- Run the simulation with the following timescale pairs. Are the reported delays different from before?

Tester	Inverter
1ns / 1ns	**1ns / 1ns**
1ns / 1ns	**1ns / 100ps**

Lab 5.3

Lab 5.3

- In this lab, you will:
  - Learn how to use parameters
  - Learn how to use parameters to specify delays
  - Learn how to use parameters to specify constants
  - Learn how to turn a size-hardcoded model into a size-generic model

Lab 5.3

- In directory ~/lab5.3, you will find the following files:
  - inverter.v Model of an inverter
  - testinv.v Tester for the inverter
- The inverter tester reports the rise and fall delays of the inverter. Run the simulation and note the rise and fall delays.
- The rise and fall delays are hardcoded in the inverter model. Modify the inverter model to have the delays specified using parameters. Use the current hardcoded delay values as default and declare the rise-delay parameter first.

Lab 5.3

- Simulate your new inverter without any modifications to the tester module. Ensure that the delays are as before.
- Modify the tester module to override the rise and fall delays, using the positional notation:

```
INVERTER #(4.5, 1.6) DUT(...);
```

- Simulate with the new tester module. What are the delays of the inverter now?

Lab 5.3

- Delete the override specification for the fall delay, then simulate again. Make sure only the rise delay was overridden.

```
INVERTER #(4.5) DUT(...);
```

- Is it possible to override only the fall delay parameter if it is declared second?
- Change the order in which the parameters are declared in the inverter module then simulate again. Which parameter got replaced?

Lab 5.3

- Modify the parameter override from positional to using a *defparam* statement (using the proper names for your parameters):

```
defparam DUT.RISE = 4.5,
         DUT.FALL = 1.6;
```

- Simulate and note the rise/fall delays.
- Change the order in which the parameters are declared, then simulate. Any changes?
- Can you specify an override value for the rise-delay parameter only? For the fall-delay parameter only? How? Verify your hypothesis.

Lab 5.3

- Parameters need not be used only for user-redefinable values. They can be used to define symbolic constants to make the code easier to understand.
- In the tester module, add three parameters defined to 0.8, 1.0, and 1.2 to represent scaling factors for best, typical, and worst-case operating conditions, respectively.
- Modify the tester to specify worst-case rise delay and best-case fall delay for the inverter. Simulate to verify the correctness.

```
defparam DUT.RISE = 4.5 * WORST,
         DUT.FALL = 1.6 * BEST;
```

Lab 5.3: Optional

- In the same directory, you will find the following files:
  - stack.v Model of a stack
  - teststck.v Stack tester
- The model of the stack is currently hardcoded with an 8-bit word width and a 4-word depth.
- Modify the stack model so parameters can be used to resize the width of the words and its depth.
- Modify the stack tester to be able to test any configuration of a stack using two parameters to define constants specifying the width and depth of the stack to be tested.

Lab 5.3: Optional

- You may want to use the $*random* task that returns a 32-bit random number to initialize the test pattern array.
- Use the parameters of the tester to configure the instantiated stack model.
- Simulate your modifications with a few combinations of width and depth.
  - Can you make it work with a width > 32?

Lab 5.4

Lab 5.4

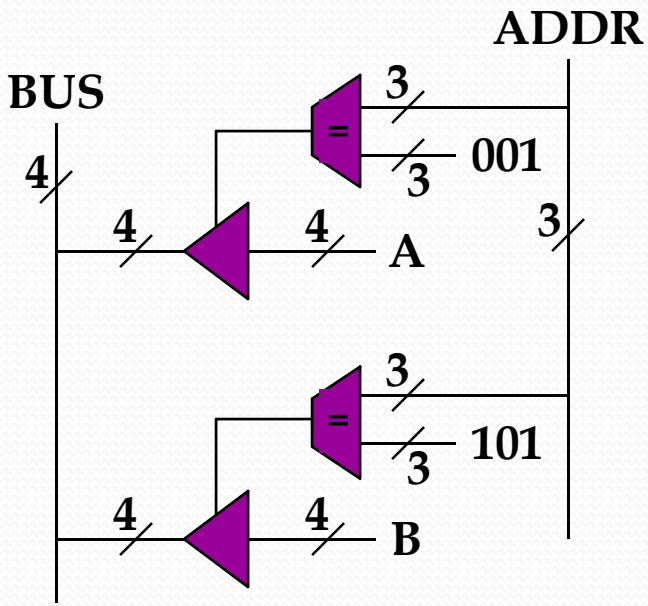
- In this lab, you will:
  - Learn how to use primitives
  - Compare modeling using primitives and behavioral code
  - Experiment with primitive delays
  - Learn how timescales affect primitive delays

Lab 5.4

- In directory ~/lab5.4, you will find a file named "testnot.v". It is similar to the inverter tester previously used, but it uses a "not" primitive instead of instantiating an inverter. The inverter rise and fall delays are specified as hardcoded values.
- Run the simulation. What are the delay values?
- Modify the timescale to "100ps/10ps" then simulate again. What are the delay values? How does that compare to using an inverter module rather than an inverter primitive? You may want to refer to the previous lab.

Lab 5.4

- In the same directory, you will find a file named "testdsgn.v". It verifies the following design for correctness:



Lab 5.4

- You will also find files named "rfilebeh.v" and "rfileprm.v" that contain identical module definitions for the previously described (partial) register file.
- In the first file, complete a behavioral description of the design, using two continuous assignments and the "?:" operator. Use the provided tester module to determine if your model is correct. Note how long it takes you to complete this step.
- In the second file, complete a gate-level description using primitives. Simulate to verify correctness and note how long it takes you to complete this step.

Lab 5.4

- Compare the simulation performance of both descriptions by simulating each one separately and noting the CPU time required.
- Whoops! Sorry! There was an error in the specification! Please modify both models, as follows:
  - The address of register "B" is "110" instead of "101".
  - Add an extra bit of address.
- Simulate using the file "testeco.v" instead of "testdsgn.v" to verify your changes.
- Which model was quickest to write/modify?
- Which model simulated the fastest?

Lab 5.4

- For most people, there isn't a significant time difference between writing the behavioral model and writing the gate-level model.
- However, when comparing writing times, you have to remember that the time required to write the behavioral model probably included the time necessary for you to understand in detail the desired functionality. Gaining this understanding may have required some debug cycles. When you started writing the gate-level model, you already had a very good idea of what was required and did not have to "debug" your understanding. You only had to debug your model.

Lab 5.4

- In the same directory, you will find a file named "sample.v". It is a slightly modified version of the (now familiar) module that contains various sampling procedures.
- A sampling procedure using a buffer primitive has been added. Compare the resulting waveforms with the others. Explain the differences.