Lab 2.1

Lab 2.1

- In this lab, you will:
  - Learn to use various operators
  - Witness the effect of operation sizes
  - Use registers as targets of assignments
  - Use concatenated assignment targets
  - Learn the difference between a vector and a memory

Lab 2.1

- In directory ~/lab2.1, you will find a file named "try.v". It contains an *initial* block with a single $*write* statement that is used to display the binary result of an expression.
- Replace the "<expression>" in the $*write* statement with each of the expressions on the next slide.
- What are the results you expect?
- Run the simulation to determine if you were right.
- Repeat the exercise, this time assigning the result of the expression to an 8-bit register, then displaying the contents of the register. Explain any differences.

Lab 2.1

Use a single *initial* block

- `1`
- `1'b1`
- `-1`
- `-1'b1`
- `1 << 1`
- `1'b1 << 1`
- `&4'b1111`
- `&4'b111`
- `|4'b0x10`
- `|4'b0x00`
- `4'b1111 & 1`
- `~4'b1000`
- `!4'b1000`
- `(1'b1) ? 1'bz : 3'bx`
- `(1'bx) ? 1'b1 : 1'b0`
- `(!1'bx) ? 1'b1: 1'b0`
- `(2'b10 & 2'b01) ? 1'b1 : 1'b0`
- `(2'b10 && 2'b01) ? 1'b1 : 1'b0`
- `(1'bx == 1'bx) ? 1'b1 : 1'b0`
- `(1'bx === 1'bx) ? 1'b1 : 1'b0`
- `{1'b1, {2'b01, 3'b1}}`
- `{3{1'b1, 2'bz}}`

Lab 2.1

- In that same directory, you will find a file named "*reversal.v*". It contains a declaration of a 4-bit register and an *initial* block that sets the register with a bit pattern, attempts to reverse it, then displays the final content of the register.
- Try to compile and simulate the Verilog model.
- You receive an error during compilation and never have the chance to simulate. Why?

Lab 2.1

- A bit reversal of a register can be performed in two ways:

```
R = {R[1], R[2], R[3], R[4]};
```

```
{R[1], R[2], R[3], R[4]} = R;
```

- Replace the faulty statement with one of the above statements and simulate. Try it with the other statement and simulate to verify that the result is the same.
  - *Note to ModelSim 5.1e users: At the time these notes were written, a bug in the simulator produced a wrong result in the second case.*
- Try both statements above, but this time with register R declared:

```
R[4:0]
```

Lab 2.1: Optional

- In the same directory, you will find a file named "regvsmem.v". It contains a declaration of a 4-bit register, then some operations on that register.
- Modify the register declaration to turn it into a 4x1-bit memory element:

```
reg R[4:1];
```

- Try to simulate without further modifications. Why are syntax errors now present?
- Fix any syntax errors and modify the source code to obtain an identical behavior as the previous simulation.

Lab 2.2

Lab 2.2

- In this lab, you will:
  - Learn how to use the *if* and *case* statements
  - Learn how unknowns are handled in the *if* and *case* statements
  - See how a *case* statement can be used to model an FSM

Lab 2.2

- In directory ~/lab2.2, you will find a file named "if.v".
- This file contains a model that attempts to ensure that a register was set to 1'b1. However, this register was clearly set to 1'bx. Why isn't the error message displayed?
- Can you fix the problem with the *if* statement?

Lab 2.2

- In a new file, create a module that contains a single *always* block and an 8-bit register named R.
- Translate the following pseudo-code into Verilog in the *always* block:

```
if R equals all X's then
   set R to all 0's
else if R is less than 10
then
   increment R by 3
else if R is less than 120
then
   shift R left by 2
else
   display R then terminate
end if
```

- Simulate your model. Verify that the answer is 8'hCo.

Lab 2.2

- In the same directory, you will find a file named "case.v": It contains a *case* statement that implements a state machine.
- After 8 transitions, the machine is stuck in state 3'b010. Why?
- Notice how the digits 'x' and 'z' are used like '0' and '1'. Does the *case* statement act like the "==" or the "===" operator?
- Change the *case* statement to a *casez*. What is the transition sequence? Why? Repeat for a *casex* statement.

Lab 2.2 : Optional

- Using a single *case* statement (no *casez* nor *casex*), translate the following pseudo-code into Verilog.

```
M(3 downto 0) := "011X";
if M(0) = '1' then i = 0;
elsif M(1) = '1' then i = 1;
elsif M(2) = '1' then i = 2;
elsif M(3) = '1' then i = 3;
else i = -1;
end if;
write("i = %d", i);
```

- Simulate to verify that the answer is 1.

Lab 2.3

Lab 2.3

- In this lab, you will:
  - Learn how and when to use the various loop statements
  - Learn the difference between the *always* and *forever* statements
  - Learn to recognize common mistakes that create infinite loops

Lab 2.3

- In directory ~/lab2.3, you will find a file named "forever.v". It contains an *initial* block with an infinite loop.
- Compile the file and fix any syntax errors you may find.
- Why isn't an *always* statement valid in an *initial* statement?

Lab 2.3

- The following pseudo-code generates walking 1s. Translate it into Verilog, using a *repeat* loop and a single *initial* block.

```
PATTERN = 1;
repeat 16 times:
    Display pattern
    shift pattern left by 1
terminate
```

- Simulate to verify that walking 1s are generated.
- ***Optional***: Add the necessary code to generate walking 0s after the walking 1s.

Lab 2.3

- The following algorithm finds the index of the least significant bit set in a register. Translate into Verilog using a single *while* loop.

```
R(15 downto 0) = E7A0 (hex);
i = 0;
while bit #i is not set
   increment i
display i
```

- Simulate to verify that the answer is 5.
- ***Optional:*** *What* happens if R is set to 16'hoooo? Can you modify your model to handle this case?

Lab 2.3

- The following algorithm reverses and negates the content of a register. Translate into Verilog using a single *for* loop.

```
R(15 downto 0) = E7A0 (hex);
for i in 0 to 15 loop
   tmp(15-i) = ~R(i);
end loop
display tmp
```

- Simulate to verify that the answer is FA18.
- ***Optional:*** *What* happens if "i" is declared as a 4-bit register?

Lab 2.4

Lab 2.4

- In this lab, you will:
  - Learn to recognize common syntax errors in functions and tasks
  - Learn how to use functions and tasks
  - Learn how arguments are passed to functions and tasks

Lab 2.4

- In directory ~/lab2.4, you will find a file named "syntax.v".
- Try to compile it: You will have syntax errors.
- Fix the syntax errors until the compilation is clean.
- Run the simulation and make sure the result is A5.

Lab 2.4: Optional

- In the same directory, you will find a file named "strange.v".
- If you simulate this model, it displays two different values for a register: one comes from a function, the other from a task.
- Why are the results different?
- Can you modify the task so the results will be the same?