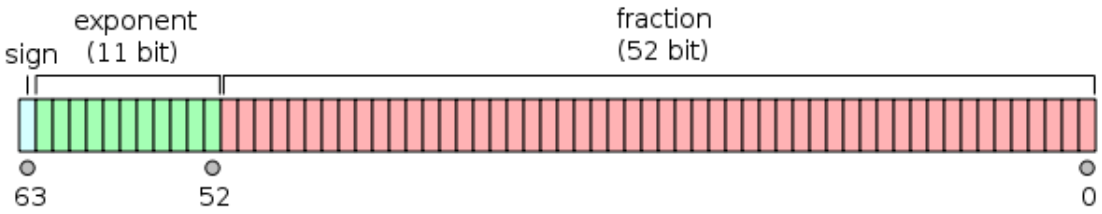
ECE 410/510
Introduction to SystemVerilog
Summer 2011
Homework 2

Reusability is one way to improve productivity and reduce errors. If care is taken when developing and testing models and testbenches they can be re-used later in other contexts, reducing development time. Because they've already been debugged and verified, they can also result in fewer errors than if new code were written.

1. Modify your floating pointer adder from Homework 1 to support IEEE 754 double precision floating point numbers instead of single precision. Be sure to modify your testbench to test the new design. The format for double precision floating point is shown below. Note that the exponent bias in double precision is 1023.



   Submit the original code from your Homework 1 as well as your new code.

2. Modify the finite state machine from Homework 1 to use SystemVerilog's new **`always_comb`** and **`always_ff`** blocks as appropriate.

   After you've verified the modified design, make a copy and perform the following experiments. Be sure to hand in both the correct modified model as well as the models demonstrating your experiments below. When describing any messages you see from your SystemVerilog tools be sure to copy the message exactly and indicate what tool (by name and function, e.g. "vlog compiler" or "vsim simulator") produced it.

   a) Modify your model to cause an illegal state to be generated (be sure your testbench triggers this behavior). What, if anything, do the tools report?

   b) Modify your model so that there is at least one path through the code in the **`always_comb`** block where the lamp outputs are not set. What messages, if any, do you get? What would have happened had you used an **`always`** block instead?

c) Modify your testbench so that not all states are exercised.
Add code to the testbench to keep track of all states that the FSM has been in. At the end of the simulation report any states that were never entered.
Hint: Create a scoreboard with an associative array indexed by state.

3. Create a typedef for a union that is capable of modeling MIPS R, J, and I format instructions. For a reference see:
http://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats

4. Write a function that accepts a single input of the type you created above and prints a MIPS instruction's fields in hexadecimal format depending upon the format