Homework #3

Assigned: Feb 11, 2020

Due: Feb 24, 2020 at 11:59 PM

Do not work in teams. Submit all files in a single zip-file.

Submit via Blackboard.

This homework is intended to provide experience with Verilog behavioral modeling of arithmetic units and structural Verilog coding style. Read and follow the instructions very carefully and include adequate comments in your model file as well as testbench.

1) Finite State Machine Design: Vending Machine (20 points)

You are asked to develop controller logic for a food vending machine that is to be deployed in Electrical Engineering department. The vending machine sells food for $0.10 to $5.00. One of your colleagues has already designed a module that provides an output “money[8:0]” that tells you how much money (max $5) client has deposited and “price[8:0]” corresponding to the item purchased. Another colleague designed a module, which provides active HIGH single-bit outputs “avail_B”, “avail_Q”, “avail_D”, “avail_N”, “avail_P” indicating whether the vending machine has one-dollar-bills (B), quarters (Q), dimes (D), nickels (N), and pennies (P) available to be dispensed. You are required to implement a Finite State Machine that returns coins from the coin dispenser stack of one-dollar-bills (B), quarters (Q), dimes (D), nickels (N), and pennies (P). Your state machine should produce “disp_B”, “disp_Q”, “disp_D”, “disp_N”, “disp_P” that would drive the stack dispenser input enable. (disp_B, disp_Q, disp_D, disp_N, disp_P are all 1-bit.)

```
module coin_return (disp_B, disp_Q, disp_D, disp_N, disp_P, done, money, price, start, clk, reset, avail_D, avail_Q, avail_D, avail_N, avail_P);
```

When “**start**” (synchronous active HIGH) is present (active only for 1-clock cycle), your state machine should consider all input values to be valid and stable and begin coin dispensing. Your machine should check if a coin-available signal (avail_B, avail_Q, avail_D, avail_N, avail_P) is asserted during this initial cycle before enabling any corresponding dispense-enable signal during the transaction. The machine dispenses one coin at a time. Each stack drops a coin if its dispense enable input is HIGH at the POSITIVE edge of the clock only if there are coins available in the stack to be dropped. If the coin-available signal was not asserted for a particular coin at the beginning of the transaction, your machine should not dispense that particular coin and use smaller denomination coins for returning the change. When the transaction is complete, your state machine should pull-up its **“done”** output. The machine uses synchronous active HIGH **“reset”** and in idle-state, the **“done”** output should remain HIGH. Your design should dispense coins as fast as possible. In order to save the time of dispensing coins, the machine should first dispense as many as possible dollars, followed by quarters, dimes, nickels and pennies respectively. In addition, if none of the coin-available signals are valid, then the machine should simply go into a state where the **"done"** is asserted without dispensing any coins. You may assume that when start is asserted, the value of money will always be greater than or equal to the value of price.

Draw and submit your implementation of state machine (vm_fsm.pdf). Test your design for various combinations of the input signals.

Submission: vm.v, tb_vm.v, vm_fsm.pdf

2) Structural Multiplier Design (20 points)

i. Using built-in gates only, develop both half-adder and full-adder cells using a STRUCTURAL VERILOG coding style. Write separate test-benches to verify the operations of both cells with all possible input combinations. Generate the output of your simulation in “ha.out” and “fa.out”.

```
module ha (a, b, sum, carry);
module fa (a, b, ci, sum, carry);
```

Where a, b, ci are inputs and sum, carry are outputs.

ii. Using only these cells and other built-in gates as needed, specify a 4-bit X 4-bit Two’s complement Baugh-Wooley carry-save multiplier (generating 8-bit output). Be sure to comment your code appropriately, especially to highlight which parts of the code correspond to parts of the multiplier (e.g., row 1, final adder, etc.). Use meaningful instance and net names so that the structure of the multiplier is obvious. Draw a simple diagram to show the overall architecture and connections of adder cells. Name your diagram multiplier.pdf

```
module mult4x4 (InA, InB, Product);
```

Where InA, InB are inputs and Product is the output of the multiplier.

iii. Develop a testbench that exhaustively tests all input combinations and outputs each result and automatically tests against expected results. At the end of the simulations, output whether all tests passed successfully or if not indicate which tests failed by printing the generated result versus the expected output result. All output should be directed to a file “mult4x4.out”. The output should contain InA, InB and Product value and the Expected value in binary format.

```
<InA InB Product_value Expected_value>
```

Submission: ha.v, tb_ha.v, ha.out, fa.v, tb_fa.v, fa.out, mult4x4.v, tb_mult4x4.v, mult4x4.out, multiplier.pdf

3) Floating-Point Multiplier (behavioral modeling) (25 points)

Write RTL Verilog code that models a single precision floating-point multiplier. A specification for this unit is given at the end of this assignment. Your design must treat the exponent and fraction parts of inputs/output as separate bit-fields that are operated on with binary arithmetic. Use high-level operators for arithmetic functions. For example, to add values you may simply use the + operator.

```
module fpmul (oprA, oprB, Result);
	input [31:0] oprA, oprB;
	output [31:0] Result;
```

Write a testbench to verify your design. Copy the provided input vectors from the DEN website to test your FP-multiplier. Use a cycle time of 5ns to assign successive input vectors. You may wish to perform pre-tests on subcomponent modules before integrating all the modules into the FP-multiplier design. Print the result in "fpmul.out" and your output format should look like (where x will be replaced by value):

```
----- Test Case 1 -----
oprA:  x  xxxxxxxx  xxxxxxxxxxxxxxxxxxxxxxx
oprB:  x  xxxxxxxx  xxxxxxxxxxxxxxxxxxxxxxx
Result:  x xxxxxxxx xxxxxxxxxxxxxxxxxxxxxxx
----- Test Case 2 -----
oprA:  x  xxxxxxxx  xxxxxxxxxxxxxxxxxxxxxxx
oprB:  x  xxxxxxxx  xxxxxxxxxxxxxxxxxxxxxxx
Result:  x xxxxxxxx xxxxxxxxxxxxxxxxxxxxxxx
.
.
```

Submission: fpmul.v, tb_fpmul.v, fpmul.out

4) Traffic Light Controller (25 points)

Write a Verilog RTL for a traffic light controller. Each traffic light can represent seven types of traffic signals: green, red, yellow, green left arrow, green right arrow, flashing-red, and flashing-yellow. The following table specifies the representations of the signals:

Traffic Signal Type	**Code to be assigned**
Green	3’b110
Green Left Arrow	3’b101
Yellow	3’b100
Red	3’b011
Green Right Arrow	3’b010
Flashing Red	3’b111
Flashing Yellow	3’b000

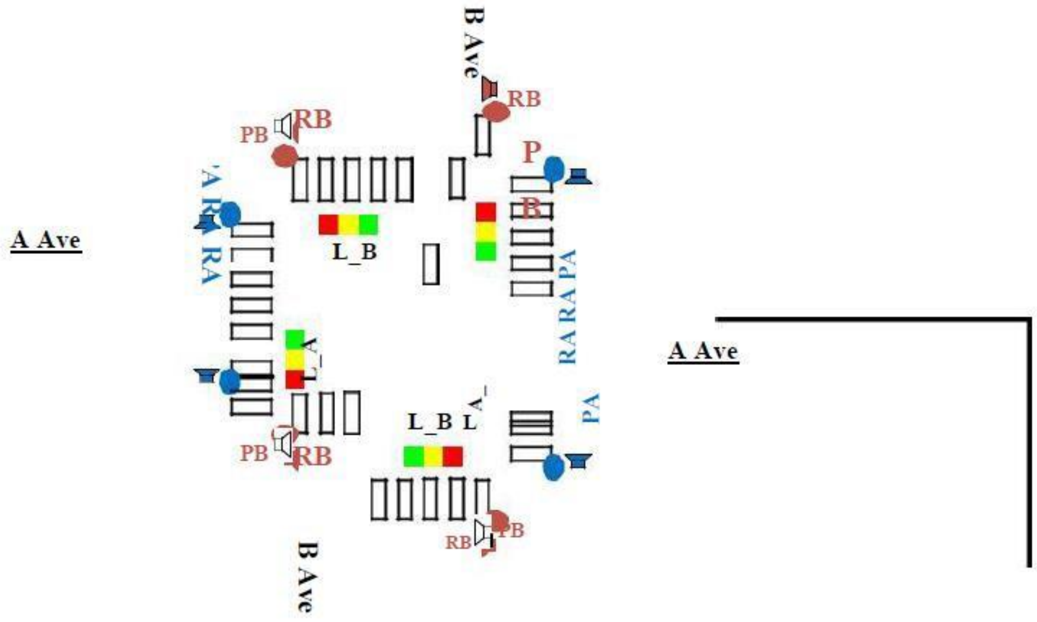


Figure 1: Traffic Controller

The above map illustrates the scenario that you are going to deal with A Ave and B Ave forming an intersection. There are two sets of traffic lights, L_A, and L_B, which control the traffic movement on A Ave and B Ave respectively. PA and PB represents the push switches that can be used by pedestrians to indicate desire to cross A Ave and B Ave respectively. RA and RB represent the audio response to indicate acceptance of pedestrian requests on A Ave and B Ave respectively.

You need to consider following scenarios for operations of traffic lights L_A and L_B. The following table shows the timing for each scenario that you need to follow. Your state machine should continuously cycle through state1 to state6 and remain in each state for the designated number of clock cycles.

State	**Traffic Light A (L_A)**	**Traffic Light B (L_B)**	**Timing (clocks)**
State 0 **Pedestrian-crossing**	Flashing red	Flashing red	6
State 1	Green	Red	8
State 2	Green Arrow Left	Green Arrow Right	3
State 3	Yellow	Green Arrow Right	3
State 4	Red	Green	8
State 5	Green Arrow Right	Green Arrow Left	3
State 6	Green Arrow Right	Yellow	3
State 7 **External-error**	flashing yellow	flashing yellow	N/A

i. If the pedestrian wants to go across A Ave, they can push pedestrian crossing switch PA (synchronous active HIGH). The controller should register PA by activating RA and enter pedestrian-crossing mode (state0) when it reaches at the end of either state3 or state6 and then continue on its regular loop through state1 to state6. (For Example if you enter state0 at the end of state3, then you should return to state4. Similarly, if you entered state0 at the end of state6, then you should return to state1. The RA will remain activated till the end of the pedestrian-crossing state. The traffic light L_B follows the same rule for its corresponding pedestrian crossing request button PB on B Ave and activation of RB.

ii. If the pedestrian wants to go across A Ave when the traffic light is in pedestrian- crossing mode, pushing PA does not affect the system at all. You can assume that the pedestrian would pass the street safely in this scenario. This requirement also applies to the traffic lights on B Ave.

iii. Outside the traffic light controller, there is an error sensor. In case of an unusual traffic situation, the error-sensor sends an active HIGH signal ERR to your controller. Your traffic light controller will take ERR (synchronous active HIGH) as an input and make all the traffic lights on A Ave and B Ave start flashing-yellow. When ERR is removed (pulled LOW), the system will begin by first allowing all the pedestrians to cross the intersection and then resuming its normal operation beginning with state1. RA and RB will remain active during the pedestrian-crossing mode.

iv. When reset (synchronous active HIGH) goes high, the system will go back to external-error state by activating traffic lights on A Ave and B Ave in flashing-yellow. When reset is deactivated, the system begins its normal operation by following steps similar to exiting external-error state as described in (iii).

v. ERR, being an external input to controller, gets priority over reset and pedestrian crossing inputs. In other words, when ERR is active, reset is to be ignored and pushing pedestrian crossing request buttons PA and PB do not affect system.

```
module traffic_control (CLK, reset, ERR, PA, PB, L_A, L_B, RA, RB);
```

Submission: traffic.v, tb_traffic.v

5) Parity with task and function (10 points)

Parity is used for error detection. A parity bit is a bit that is added to ensure that the number of bits with the value one in a set of bits is even or odd. Parity bits are used as the simplest form of error detecting code. There are two variants of parity bits: even parity bit and odd parity bit. When using even parity, the parity bit is set to 1 if the number of ones in a given set of bits (not including the parity bit) is odd, making the number of ones in the entire set of bits (including the parity bit) even. If the number of on-bits is already even, then the parity bit is set to a 0. When using odd parity, the parity bit is set to 1 if the number of ones in a given set of bits (not including the parity bit) is even, keeping the number of ones in the entire set of bits (including the parity bit) odd. And when the number of set bits is already odd, the odd parity bit is set to 0. In other words, an even parity bit will be set to "1" if the number of 1's + 1 is even, and an odd parity bit will be set to "1" if the number of 1's +1 is odd. (Reference: http://en.wikipedia.org/wiki/Parity_bit)

In this question, you are asked to design the parity encoding modules.

1. Based on even parity, design a task that can generate parity bit. The output of the task is the even parity bit. The output of the parity_task is input along-with parity bit occupying LSB position.

```
module parity_task (in, out);
        input [8:0] in;
        output [9:0] out;
        taskeven_parity_bit;
```

   Submission: task.v

2. Based on odd parity, design a function that can generate parity bit. The output of the function is the odd parity bit. The output of the parity_function is input along-with parity bit occupying LSB position.

```
module parity_function (in, out);
        input [8:0] in;
        output [9:0] out;
        functionodd_parity_bit;
```

Submission: function.v

Submission Process

We will be using makefile structure for HW3. Seperate all files in each question into a folder. Keep all your design, testbench, output log/.txt/.pdf files in the respective design/ tb/ reports/ directories.

“zip” all the required files for HW#3 and use the upload link of HW3 in Blackboard for electronic submission.

Single-Precision Floating-Point Multiplier Specification

The single-precision floating-point multiplier (FP_MUL) assumes IEEE-754 format as shown in Figure 1.

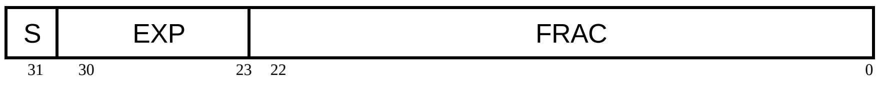


Figure 1: IEEE-754 Single-Precision Floating-Point Format

The IEEE-754 standard uses sign-magnitude representation with a biased exponent where the resulting value of the represented number is given by

$$\mathbf{(-1)^{S} \times 2^{(EXP-127)} \times (1.FRAC)}$$

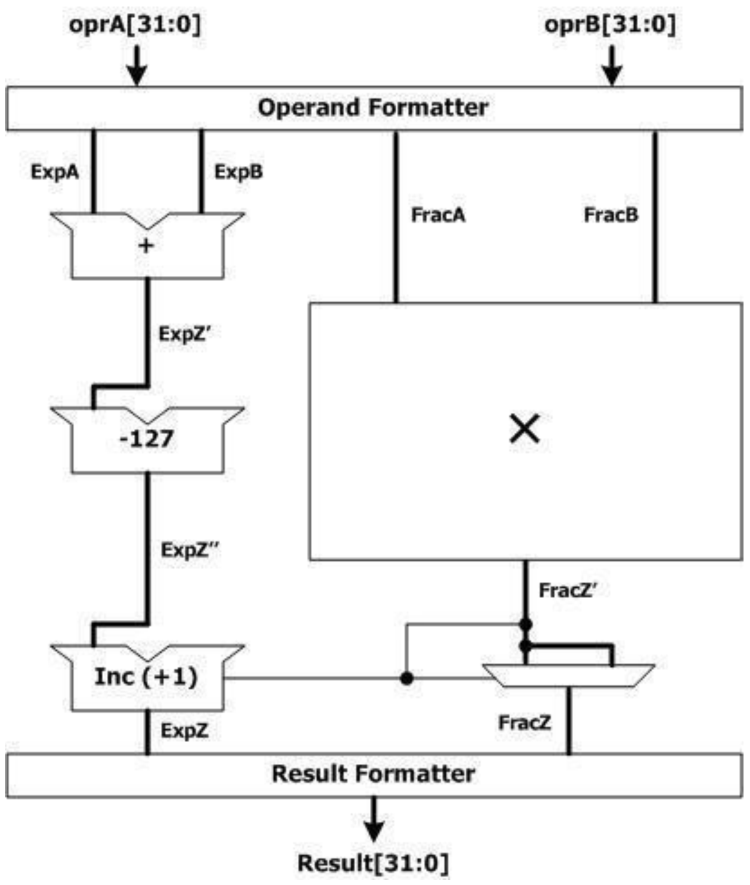
- **S**: Sign bit (1 bit)
  1 = negative
  0 = positive

- **EXP**: Exponent (8 bit) is biased by 127
  Biased representation: Since the exponent can be positive or negative, some method must be chosen to represent its sign. The IEEE standard does not use two's complement method, but instead uses a sign-magnitude representation with biased exponent. In the case of single precision, where the exponent is stored in 8 bits, the bias is 127. What this means is that if EXP is the value of exponent bits interpreted as an unsigned integer, then the represented exponent of the floating-point number is EXP – 127. The bias allows the concatenation of the EXP and FRAC fields to be used a single unsigned number for comparison purposes.

- **FRAC**: Fraction (23 bit)
  Since floating-point numbers are always normalized, the most significant bit of the fraction is always 1, and there is no reason to waste a bit of storage representing it. Formats that use this trick are said to have a "hidden" bit. Since a hidden bit is used in the IEEE-754 format, the significand of the represented number is really 24 bits, although only the 23-bit fractional part is represented in the format. Therefore, a leading 1 is implicit for normalized number: Significand (or mantissa) = 1.FRAC

Although the IEEE-754 standard allows for many special data formats, the FP_MUL operates only on normalized numbers. That is, all inputs are assumed to be normalized and all results are assumed to be capable of being normalized within the specified format. Even though several exception cases (i.e. overflow, underflow…) are specified in the IEEE-754 standard, they don't need to be implemented in this assignment and provided input vectors will not cause any exception cases.

The block diagram of the FP_MUL is shown in Figure 2. The FP_MUL receives two 32- bit inputs oprA and oprB, then it generates 32-bit output Result.



A brief description of multiply operation is given below.

The inputs oprA and oprB are multiplied using floating-point arithmetic. The sign bit of the result is simply the exclusive-or of the oprA and oprB sign bits. The result exponent is simply the addition of the oprA and oprB exponent fields except that since both values are biased by 127, 127 must be subtracted from this addition result for the proper preliminary exponent field. The significand's are simply multiplied (don't forget to prepend 1's before both FRAC fields). A 48-bit result will form from this multiplication where only 24 bits are needed. The selection of the proper 24 bits depends on where the leading 1 is. Depending on the input operands, the result may need to be normalized, so that a 1 is the first bit of the result significand before the hidden 1 bit is stripped off. In the process of this normalization, the significand may need to shift right and the exponent must be adjusted accordingly.