Exercise 17 (Balance Control Random Testing)

Before you start this exercise ensure the following **localparam** in PID.sv is set as follows:

localparam P_COEFF = 5’h09

Ensure **fast_sim** = 1 for this simulation

If you are still failing the below random tests then take a look at running **PID_fastsim_tb.sv** from Ex15.

If you are still failing after that….good luck!

Exercise 17 (Balance Control Random Testing)

- You will test your **balance_cntrl.sv** unit using stimulus and expected response read from a file. On the Canvas page you will find **balance_cntrl_stim.hex** and **balance_cntrl_resp.hex**. These represent stimulus and expected response.

- For **balance_cntrl_stim.hex** the vector is 49-bits wide and is assigned as follows:

Stimulus Bit Range:	Signal Assignment:
stim[48]	rst_n
stim[47]	vld
stim[46:31]	ptch
stim[30:15]	ptch_rt
stim[14]	pwr_up
stim[13]	rider_off
stim[12:1]	steer_pot
stim[0]	en_steer

- There are 1500 vectors of stimulus and response. Read each file into a separate memory using **$readmemh**.

- For **balance_cntrl_resp.hex** the vector is 25 bits wide and is assigned as follows:

Response Bit Range:	Signal Assignment:
resp[24:13]	lft_spd
resp[12:1]	rght_spd
resp[0]	too_fast

- Create a testbench to apply the stimulus and check the results. Call it **balance_cntrl_chk_tb.sv**

- Loop through the 1500 vectors and apply the stimulus vectors to the inputs as specified. Then wait till #1 time unit after the rise of **clk** and compare the DUT outputs to the response vector (self check). Do all 1500 vectors match?

- Submit **balance_cntrl.sv**, **balance_cntrl_chk_tb.sv** to the dropbox

NOTE: You need a **force** statement to force **iDUT.ss_tmr** to 0xFF

Exercise 17 balance_cntrl_chk_tb.v Hints:

- Instantiate DUT (**balance_cntrl.sv**) *(name the instance **iDUT**)* connecting all the inputs to the respective bits of a 49-bit wide vector of type **reg**.
- Declare a "memory" of type **reg** that is 49-bits wide and has 1500 entries. This is your stimulus memory
- Declare a "memory" of type reg that is 25-bits wide and has 1500 entries. This is your expected response memory
- Inside the main "initial" block of your testbench do a **$readmemh** of the provided **.hex** files into the respective "memories"
- Ensure you have a: **force iDUT.ss_tmr = 8'hFF;** in the **initial** block
- Start **clk** at zero and toggle it every #5 time units the way we often do
- In a **for** loop going over 1500 entries assign an entry of the stim memory to the stim vector that drives the DUT inputs
- Wait for @(**posedge** clk), then wait for **#1** more time unit. Now check, do DUT outputs match the respective bits of the response vector?

Loading Memory Data From Files

- This is very useful (memory modeling & testbenches)
  - $readmemb("<file_name>",<memory>);
  - $readmemb("<file_name>",<memory>,<start_addr>,<finish_addr>);
  - $readmemh("<file_name>",<memory>);
  - $readmemh("<file_name>",<memory>,<start_addr>,<finish_addr>);

- **$readmemh** ➔ Hex data…**$readmemb** ➔ binary data
  - But they are reading ASCII files either way (just how numbers are represented)

```
// addr     data
@0000 10100010
@0001 10111001
@0002 00100011
```

example "binary" file

```
// addr     data
@0000   A2
@0001   B9
@0002   23
```

example "hex" file

```
//data
A2
B9
23
```

address is optional for the lazy

Example of **$readmemh**

```
module rom(input clk; input [7:0] addr; output [15:0] dout);

reg [15:0] mem[0:255];      // 16-bit wide 256 entry ROM
reg [15:0] dout;

initial
   $readmemh("constants",mem);

always @(negedge clk) begin
   ///////////////////////////////////////////////////////////
   // ROM presents data on clock low //
   /////////////////////////////////////////////////////////
   dout <= mem[addr];
end

endmodule
```