Solution To More Involved Problems of HW1

Fundamental difference between **$stop** and **$finish**:

$stop ends the simulation but keeps the simulator open, and the waveforms continue to display. This is used if you want to debug the simulation results (i.e. inspect the waveforms)

$finish not only ends the simulation, but exits ModelSim entirely. This would be used if you had self checking testbenches that you ran in a regression suite via a script. When the test hits **$finish** the simulator would relinquish control of the thread back to the script, so the script could kick off the next test in the suite.

Question 19 (Edge detector for asynch signal)

You need to double flop first for metastability. Then you need a 3rd flop and some logic for edge detection.

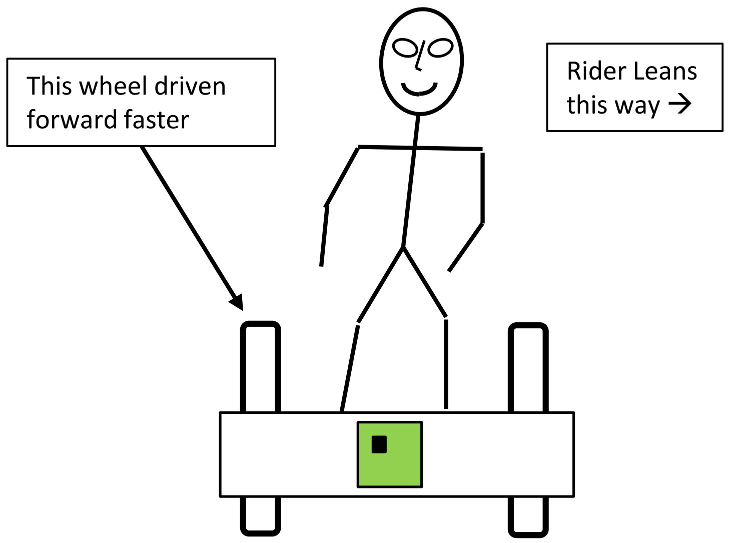
```
wire DFF1,DFF2,DFF3,DFF2_n;

dff FF1(.D(asynch_sig), .clk(clk), .Q(DFF1));
dff FF2(.D(DFF1), .clk(clk), .Q(DFF2));
dff FF3(.D(DFF2), .clk(clk), .Q(DFF3));
not inv1(DFF2_n,DFF2);
and and1(fall_edge,DFF2_n,DFF3);
```

Research **PID** control. Discuss it in relation to self balancing scooter like a Segway.

PID control uses the concept of a desired outcome vs a measured outcome. The difference between the desired and the measured form an error term. The control inputs are then a combination of terms proportional to:

1.) The error (this is the **P** term)
2.) The integral of the error (this is the **I** term) and is used to drive the error to zero.
3.) The derivative of the error (this is the **D** term) and is needed when the system has significant lag, or inertia.

Can a "Segway" get away with just **PI** control? No. A Segway is a system that has considerable inertia. The desired result is a level platform. If the platform is tilting forward the motors need to be driven forward to compensate. As the motors are driven forward and the platform is becoming level, it has angular momentum, so the drive has to be reversed before the platform actually reaches level. It is the **D** term of **PID** that would accomplish this.



Signal:	Direction:
clk, rst_n	in
tmr_full	in
sum_gt_min	in
diff_gt_1_4	in
diff_gt_15_16	in
clr_tmr	out
en_steer	out

We are designing a “Segway” like device, except ours will not have a handle for steering. Instead it will steer by the rider leaning side to side.

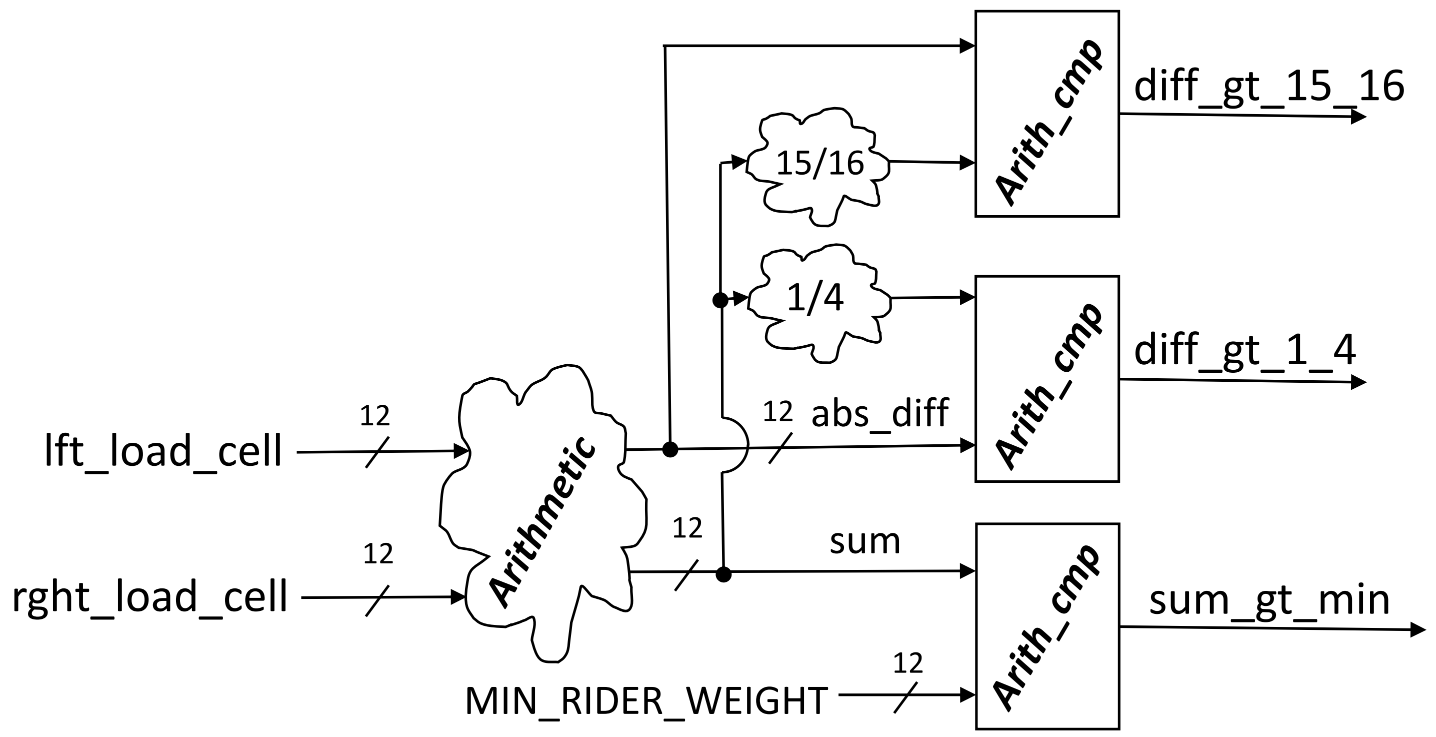
Load sensors (left and right) in the floor of the platform will detect rider leaning to one side or the other. In the case shown the wheel on the left would be driven harder than the wheel on the right.

You are designing a state machine with this interface to control when steering is enabled.

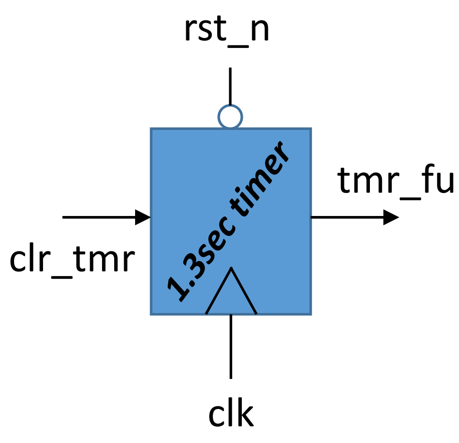
Now imagine initially stepping on the platform. The device is on and in active balance mode. When you step on with one foot it is going to think you are leaning hard to that side and that wheel will be driven backwards and the other wheel will be driven forwards. It would be pretty hard to get on right?

We need to enable steering only after we know the rider has mounted the platform and is situated. We will do this by observing the reading of the load cells. We also want to disable steering as soon as we think the user is dismounting (when one load cell reads substantially greater than the other). Finally we want to return to our initial state when we know the user has gotten off (load cells read below min rider weight).

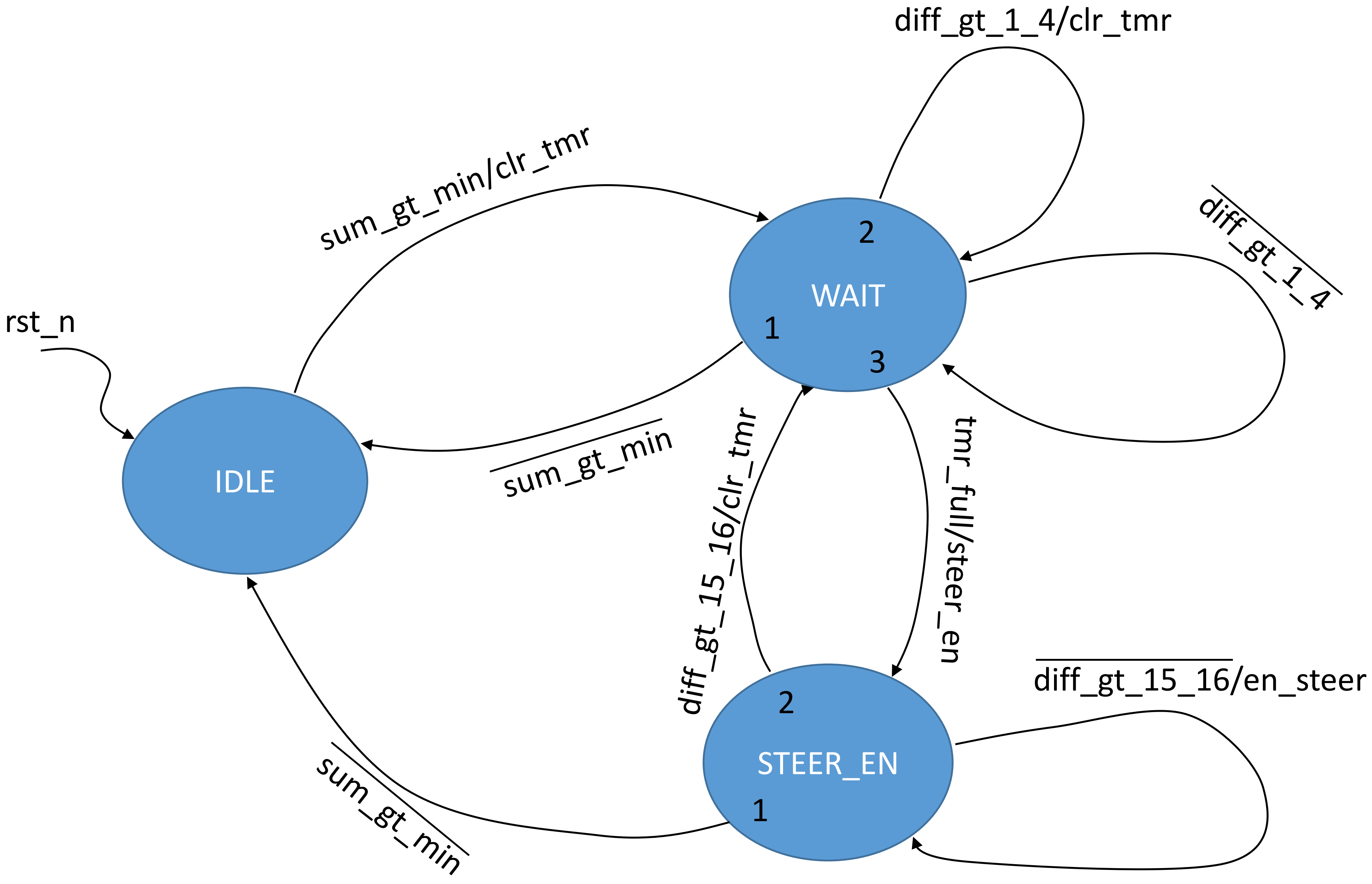
Assume the hardware elements shown here are available to you. You are figuring out the SM bubble diagram



Timer always counts up, but you can clear it.

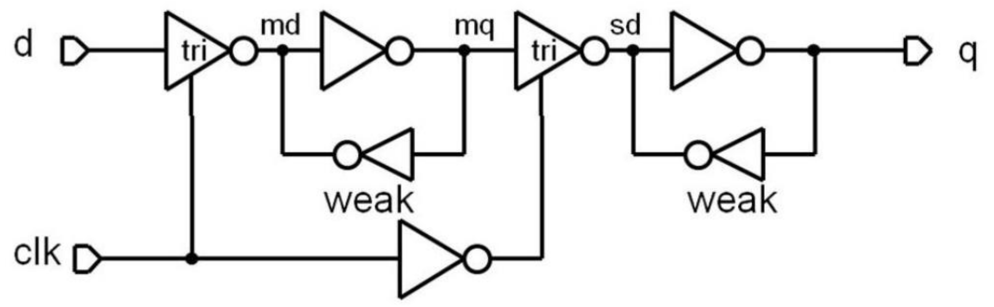


First we look to see that the sum of the two load cells exceeds the minimum rider weight. If that condition is met (**sum_gt_min**) we next look at **diff_gt_1_4** to see that the absolute value of the difference between the left and right load cells does not exceed 1/4 of the total weight for 1.3 seconds. If the difference has been consistently below 1/4 of the sum for 1.3 seconds then steering is enabled (hold **en_steer** high). We can leave the steering enabled state one of two ways. The rider suddenly gets knocked off the device (**sum_gt_min** goes low), or the difference between the load cells exceeds 15/16 of the sum (rider is stepping off). Under both conditions we exit the steering enabled state. If the user is stepping off we return to the state for waiting for balance for 1.3 sec. If the user is knocked off we return to the initial state. If the user is stepping off and we are in the wait state checking for balance we have to look for them to completely step off (**sum_gt_min falls**) or to regain balance. **Draw the bubble diagram** for this SM. Take a picture with your phone and submit it. It must be legible!



diff_gt_1_4/clr_tmr
sum_gt_min/clr_tmr
rst_n
WAIT
2
1
3
$\overline{\text{diff\_gt\_1\_4}}$
IDLE
$\overline{\text{sum\_gt\_min}}$
diff_gt_15_16/clr_tmr
tmr_full/steer_en
$\overline{\text{diff\_gt\_15\_16}}$/en_steer
STEER_EN
2
1
$\overline{\text{sum\_gt\_min}}$

One can put priority numbers leaving states. For example the "1" leaving state "WAIT" indicates this is the highest priority. If sum_gt_min is low we will leave WAIT no matter what else. The 2nd check is if diff_gt_1_4 is high we return to "WAIT". This nomenclature is similar to the **if/else** way you will code it, and simplifies the labels on the arcs.



This is a negative edge triggered M/S Flip Flop. You can tell it is negative edge because the output can only change on clock low.

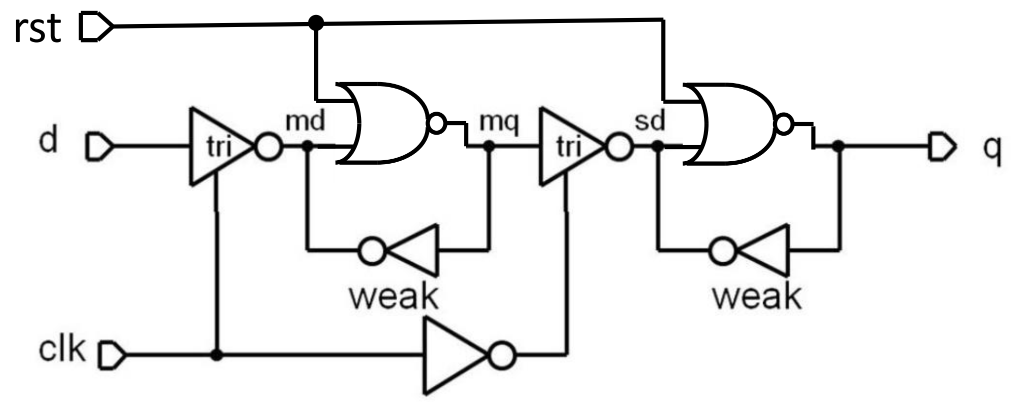
How do we add an active high reset signal?

This can't be done by gating off the D input…that is synchronous reset, not asych.

It can't be done by knocking down the Q output with some gate. The affect of that would disappear as soon as the reset signal went away.

To reset a M/S FF you need to affect the contents of the memory loop of both the master and slave latches.

No matter how I ask it. (active high or active low) (set or reset) it always involves changing two of the inverters into either NAND or NORs



Making an active high reset.

If I had asked for an active low reset what would that have looked like?