

Exercise 12 (PID from PID_Math)(HW3 Prob 3):

- Copy **Ex07/PID_Math.sv** file to **Ex12/PID.sv**. Change module name from **PID_Math** to **PID**.
- You will now augment this file to include the sequential elements needed and create the full **PID** controller for the Segway.
- Interface changes:
 - Add **clk** & **rst_n** (*synch elements now part of design*)
 - Add a **vld** input that will indicate when a new inertial sensor reading is valid.
 - Remove **integrator[17:0]** as input (*this will now be an internal register*)
 - Add **ss_tmr[7:0]** as an output which will be the upper bits of a timer used to effect a soft start (*consumed in SegwayMath.sv*)
 - Add two new single bit inputs called **pwr_up** & **rider_off** (*will discuss these later*)

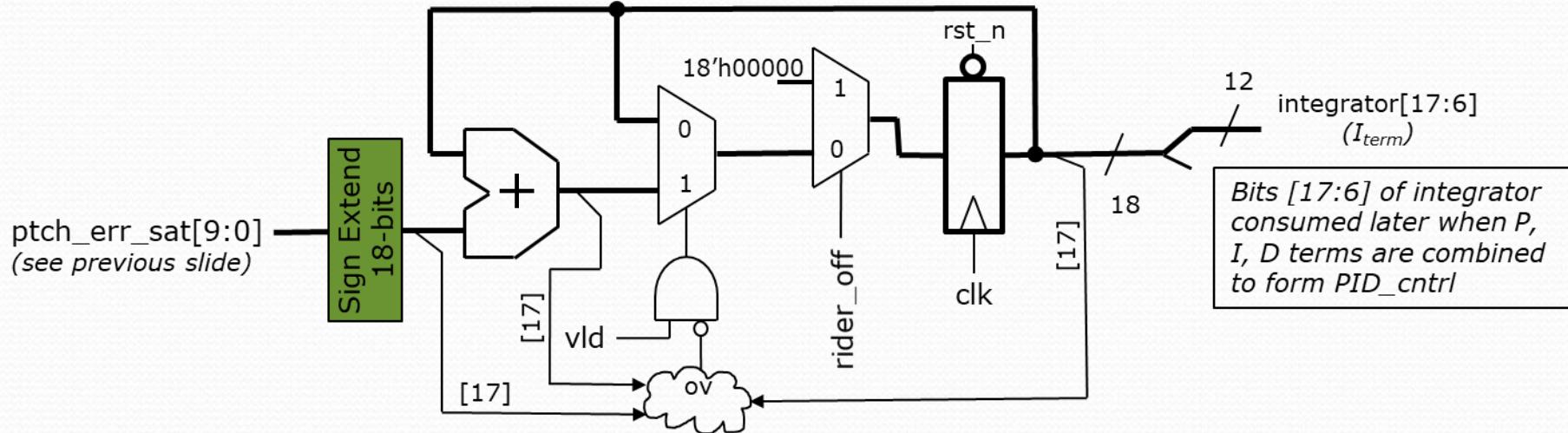
PID (Interface and the P & D of PID)

Signal:	Dir:	Description
clk,rst_n	in	50MHz system clock & active low reset
vld	in	High whenever new inertial sensor reading (ptch) is ready
ptch[15:0]	in	Pitch of Segway from <i>inertial_intf</i>
ptch_rt[15:0]	in	Pitch rate (degrees/sec). Used for D_term of PID
pwr_up	in	Asserted when Segway balance control is powered up. Used to keep ss_tmr at zero until then.
rider_off	in	Asserted when no rider detected. Zeros out integrator.
PID_cntrl[11:0]	out	This is the main output we are looking to make
ss_tmr[7:0]	out	Soft start timer...scales drive in SegwayMath

- The **P_term** of **PID** is unchanged from **PID_Math**.
- The **D_term** of **PID** is also unchanged from **PID_Math**
- The **I_term** needs to be an internal accumulator register. You will have to declare a signal of type **reg** (or **logic**) i.e. **reg [17:0]** integrator; // see next slide for more detail.

PID (The I of PID (Integral))

The 18-bit integrator used to be an input. Now we need to form the integrator accumulator as an internal register.



The primary function of the I_{term} is quite simple. On every **vld** reading from the inertial sensor the saturated version of pitch error (**ptch_err_sat**) is accumulated into an 18-bit accumulator register. We then use the upper bits ([17:6]) of this accumulator to form our I_{term} that summed with our P_{term} and D_{term} to form **PID_cntrl**.

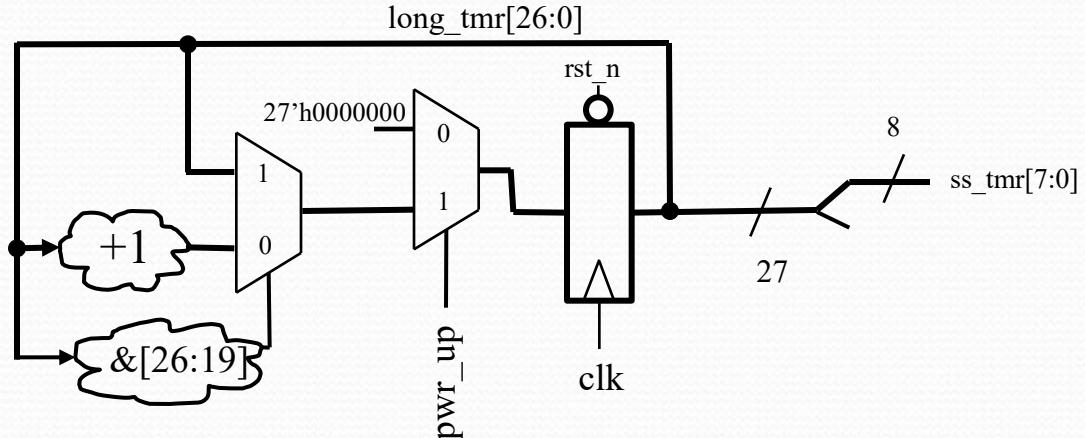
When the rider steps off the "Segway" it is possible the I_{term} was kind of "wound up". We don't want the "Segway" ramming them or running away after they step off, so we clear the integrator on **rider_off**.

We don't want to let the integrator roll over (either beyond its most positive number or its most negative number). The easiest way to accomplish this is to inspect the MSBs of the two numbers being added, if they match, yet do not match the result of the addition then overflow occurred. If overflow occurs we simply freeze the integrator at the value it was at last which must have been pretty close to a full positive or negative number.

PID (Soft Start Timer)

- The **PID** unit will also generate the soft start timer (**ss_tmr[7:0]**) that is used in **SegwayMath** to ensure on power up the Segway does not jerk to a start.
- We want this ramp of **ss_tmr** to be in the 2-3 second range. Which given our clock speed (50MHz) takes quite a wide timer.
- **ss_tmr[7:0]** will be formed from the upper 8-bits of a 27-bit timer.
- The FPGA of the Segway is powered and running initially, but it is waiting for a code from the **auth_blk** (authorization block) to actually enable the balancing feature and allow a rider to ride it. When this code comes in via a Bluetooth link the **pwr_up** signal will be asserted.
- The counter that forms **ss_tmr** should be held in a zero state until **pwr_up** is asserted.
- See the next slide for more detail on **ss_tmr** implementation.

PID (Soft Start Timer)(continued)



- When **pwr_up** is deasserted the soft start timer should remain zero.
- It is a one shot timer. Once it starts counting and gets near full (*bits [26:19] set*) it freezes.

Exercise 12 (PID Initial Testing):

- A testbench (**PID_dbg_tb.v**) is provided for you. Run your DUT against this self-checking testbench and debug.
- **PID_dbg_tb.sv** code is commented fairly well with what it is trying to test. Look at the comments when debugging your code. This code running perfectly does not ensure your DUT is good.
- In a later exercise we will test this unit more thoroughly using random testing.
- Modify the testbench to include your name in the output messages.
- **Submit:** **PID.sv** and **proof** it passed the testbench (*or as much as you get done by end of class...this will have to be completed for HW3*).