Signed Operations in Verilog:

- Addition & Subtraction are agnostic operations.
  - An adder to add two signed numbers is no different than an adder to add two unsigned numbers.

- Multiplication is not agnostic
  - A multiplier for signed numbers is a little different than a multiplier for unsigned numbers.

- Magnitude comparison is not agnostic

- Right shifting is not agnostic *(arithmetic vs logical)*

Verilog is stubborn…it wants to be unsigned:

- To infer a signed multiplier all three signals must be declared as type signed. *(dest signal and both operands)*
  - **assign** product = operand1 * operand2;

- There is also a $signed() casting operator that can be used:
  assign product = value * **$signed**(6);

- Assign Pterm = pitch_err_sat * $signed(PCOEFF);

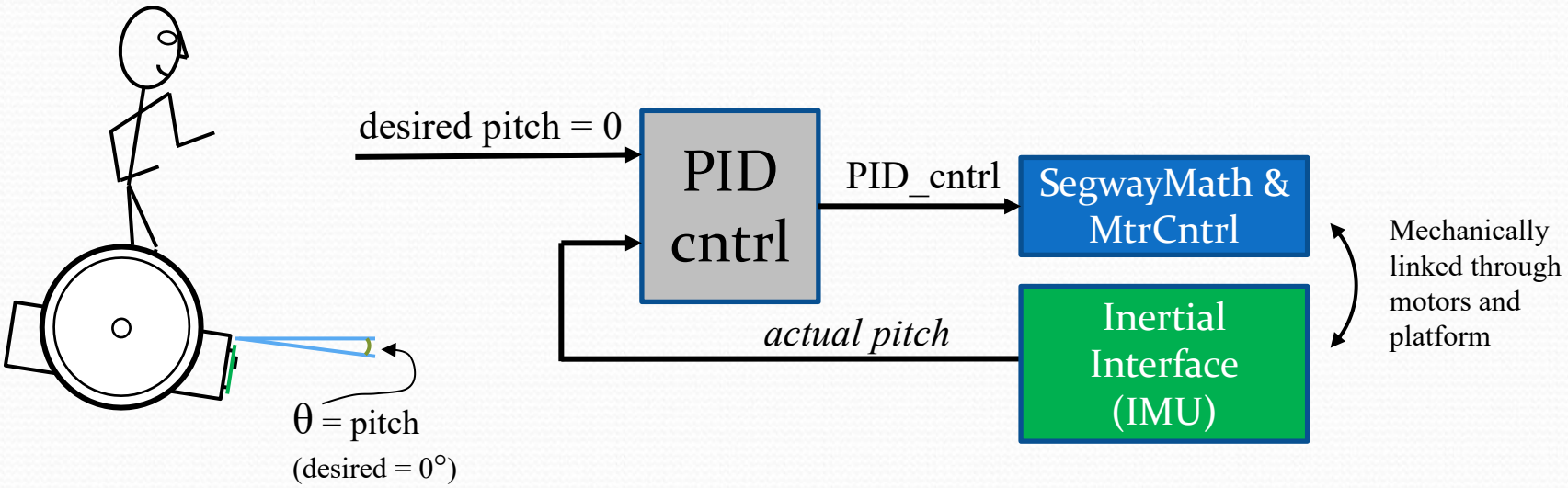
- If both numbers are **signed** then a magnitude comparison is signed. Otherwise you could used **$signed()** as well to force a signed magnitude comparison.

- If you want to use the >>> operator you need to ensure both the dest signal and operand being arithmetically right

Exercise 7: What is PID Control?

- **PID** is a very common control scheme.
- Lets take an example from our Segway project.
  - We want to drive the left & right motors to balance the platform
  - Our actual control knob *(duty cycle of the PWM)* controls motor speed/torque
  - We can measure *(using inertial sensor)* the levelness of the platform, and we can affect it by how we drive the motors, but how do we assert “control” over it?

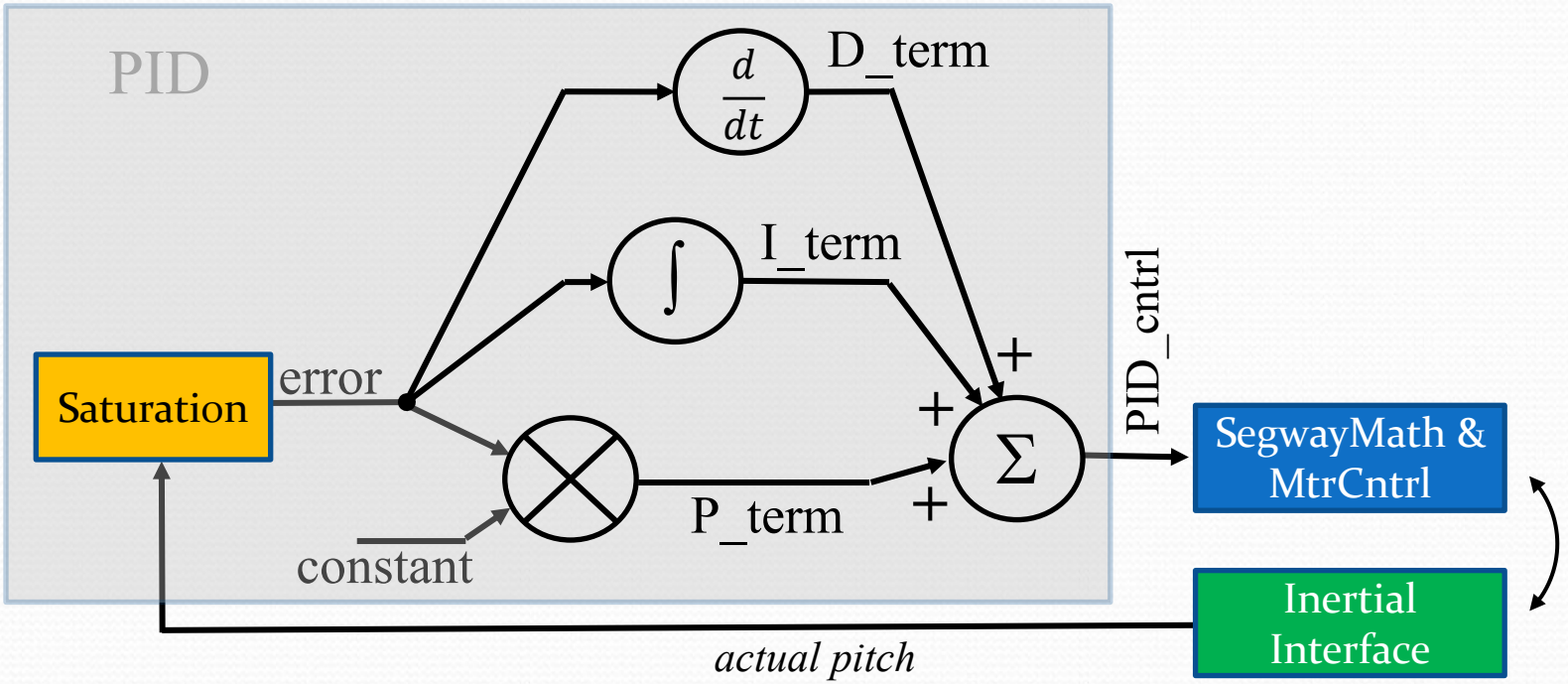
Exercise 7: What is PID Control?



- The pitch of the platform is measured via an inertial sensor.
- The desired pitch (zero) can be achieved by driving the motors frwrd/rev the correct amount.
- Determining the duty cycle to drive the motors frwrd/rev is the job of the PID block.

Exercise 7: What is PID Control?

- PID ➔ **P**roportional, **I**ntegral, **D**erivative

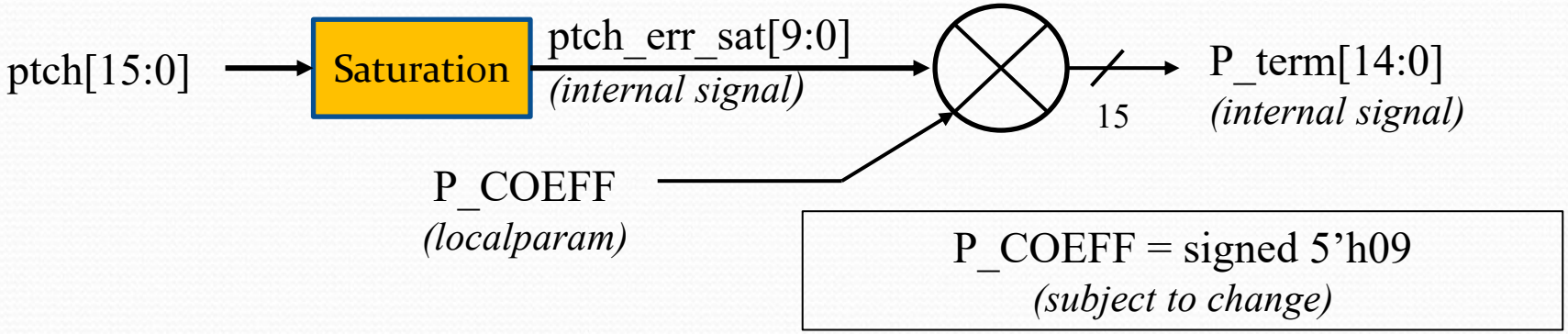


- We will get into the details of designing the PID block in a later exercise when we know more.
- For this exercise we will just "practice" some of the math that needs to occur inside the PID unit.

Exercise 7: PID Math: (forming saturated 10-bit error term & P_term)

PID controllers work on an error term (difference between measured control variable and the desired). Since our desired control variable *(pitch of the platform)* is always zero there is no need to subtract the desired from the actual to form an error term *(subtracting zero is worthless)*.

However, we do want to saturate the incoming signed 16-bit **ptch** to a signed 10-bit **ptch_err_sat** term.



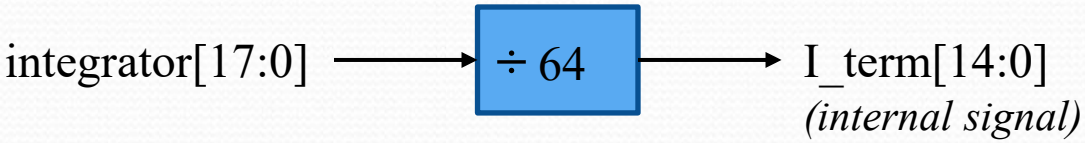
The P in PID is proportional. So **P_term** is simply formed by multiplying the error term by a coefficient. **NOTE:** this needs to be a signed multiply.

Exercise 7 PID Math: (I_term)

Integration is summing over time, which really is not that hard. Still, we don't quite have the skills yet to make the integrator accumulator so we will assume we already have an 18-bit signed signal called **integrator[17:0].**

I_term will be a 15-bit signed quantity and is simply **integrator**/64. *One might question why I_term needs to be 15-bits. An 18-bit value divided by 64 should be able to fit in a 12-bit value. This is true, but later we will see we have to modify some of the math to speed up simulations. For this reason we will keep I_term at 15-bits.*

>>>6

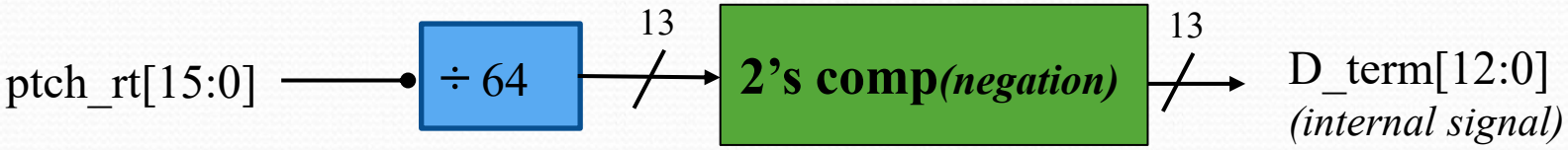


{{6{integrator[17]}},integrator[17:6]};

Exercise 7 PID Math: (D_term)

Differentiation is actually not all that difficult, but in our case it is even easier. The inertial senor actually produces angular rate of the platform, not angular position (**ptch**). We actually integrate this angular rate inside ***inertial interface*** to obtain angular position. So the **D_term** is simply proportional to the angular rate (**ptch_rt**) coming out of the inertial sensor (a MEMs gyro).

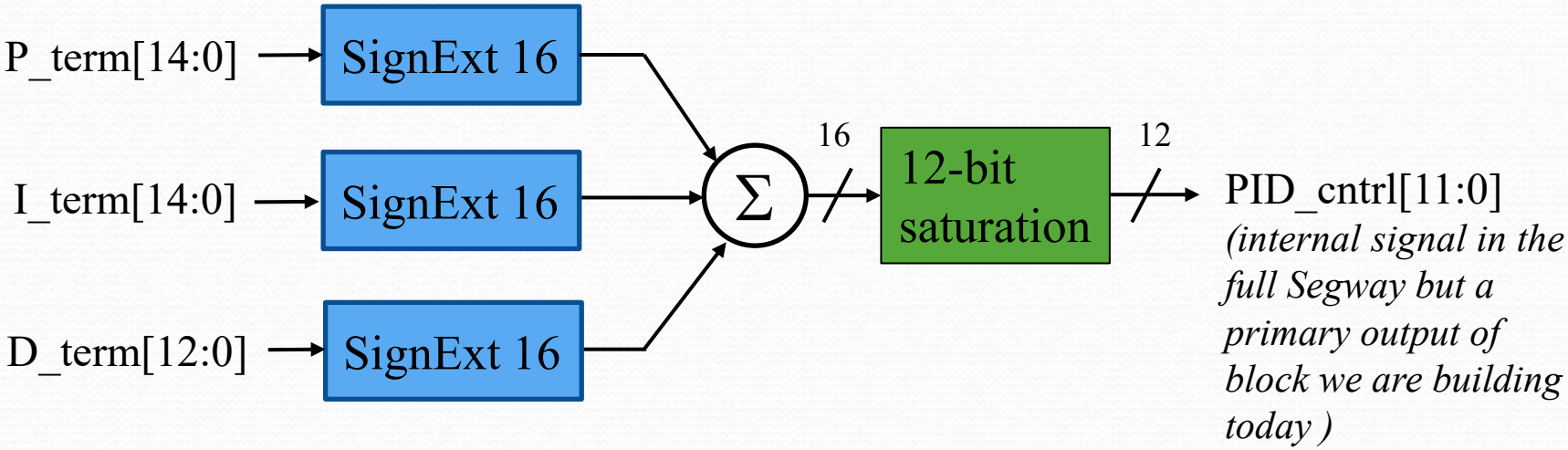
D_term = -(ptch_rt/64)



D_term is a 13-bit result *(again could theoretically be 9-bits, but we will make it 13-bits for reasons you will see in the future).* This means you need to sign extend ptch_rt by 3-bits as you drop the lower 6-bits.

Exercise 7 PID Math: (Putting it altogether (PID_cntrl))

PID_cntrl is simply the sum of **P_term**, **I_term**, and **D_term** saturated to 12-bits

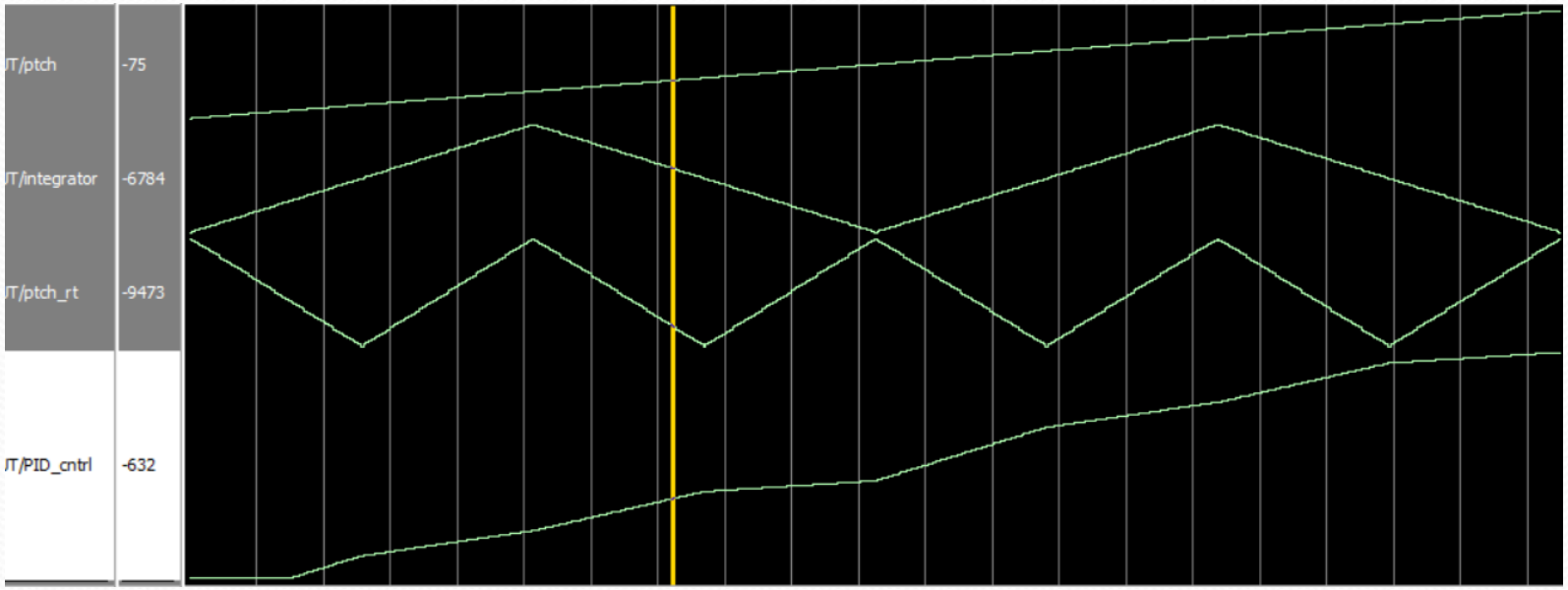


Exercise 7 PID_Math: (PID_Math.sv interface)

Signal:	Dir:	Description:
ptch[15:0]	in	Signed 16-bit pitch signal from ***inertial_interface***.
ptch_rt[15:0]	in	Signed 16-bit pitch rate from ***inertial_interface***. Used for D_term.
integrator[17:0]	In	18-bit integrator accumulation register
PID_cntrl[11:0]	out	12-bit signed result of PID control

Exercise 7 PID Math: (Testing it)

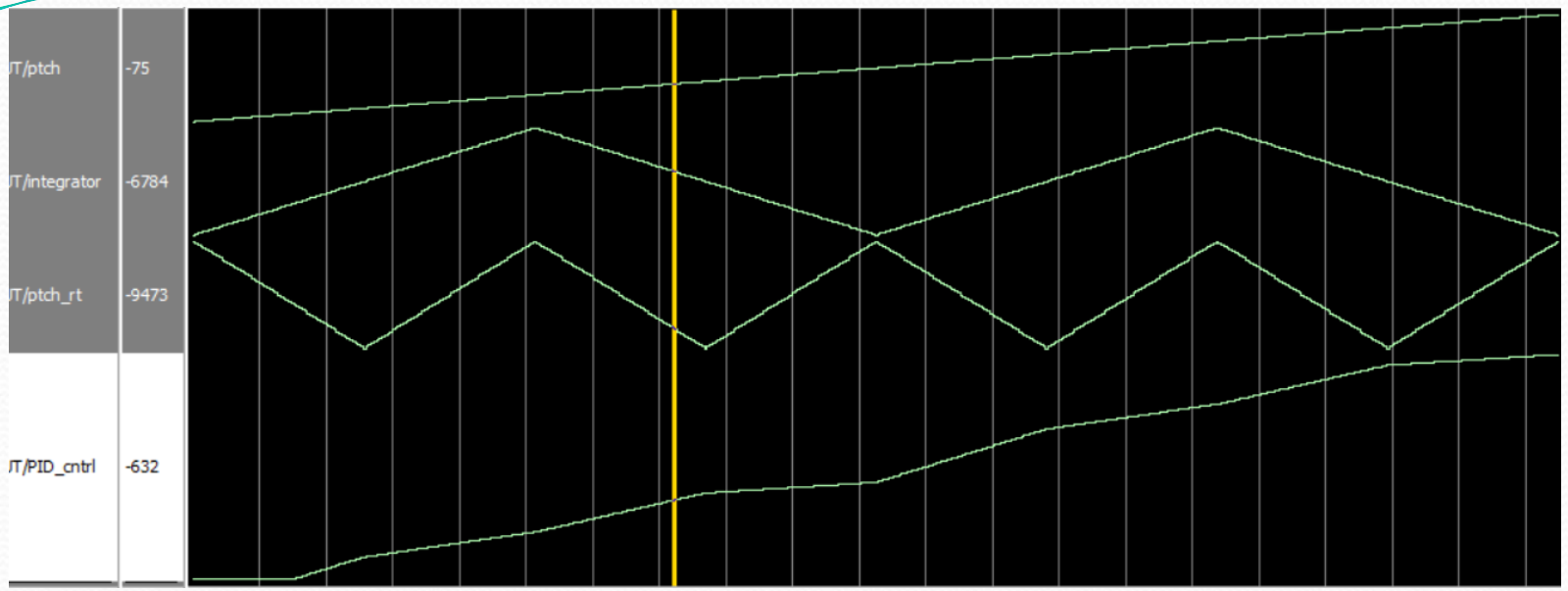
This is a simulation where **ptch** starts at 16'hFF00 (-256), ramps to 16'h0FF (+255). Simultaneously **integrator** is ramping up/down four times from 18'h3C000 to 18'h03FFF and then back down to 18'h3C000. While all that is occurring **ptch_rt** is ramping down/up eight times with extents of its ramp being 16'h0FFF to 16'hF000.



This is not a comprehensive self-checking test, but if you can duplicate the subtle shape and rough magnitudes of **PID_cntrl** you are probably in good shape. We will revisit validating PID later in the course. **NOTE**: The format of **PID_cntrl** is not shown as *"Analog (automatic)"* but rather as *"Analog (custom)"* with 150px high.

Submit what you have (**PID_Math.sv** & **PID_Math_tb.sv**) by the end of class. You need to finish this for HW2.

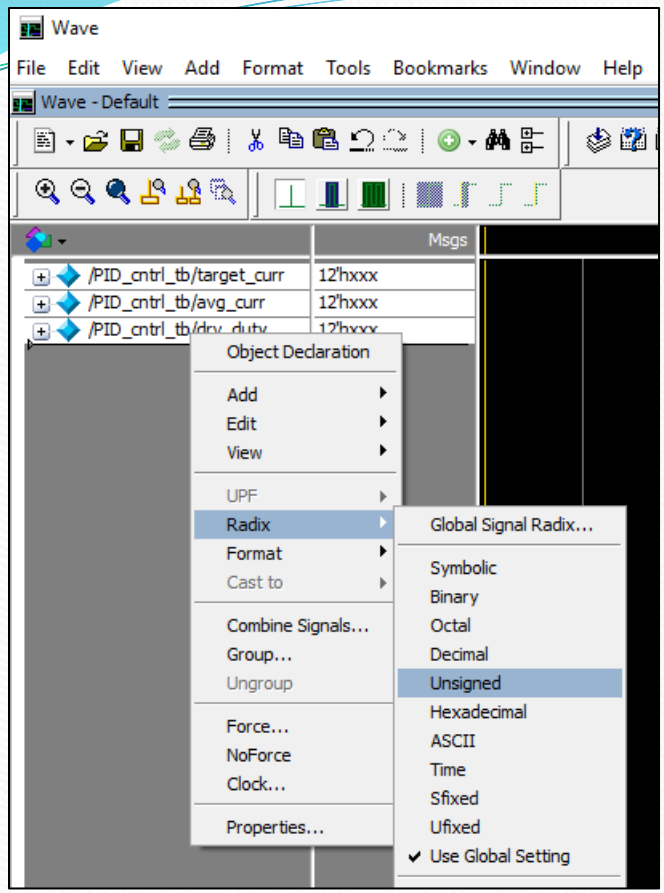
Exercise 7 PID Math: (Testing it…test bench implementation)



There are several ways you can accomplish generating this stimulus. I chose to break it into 8 repeat loops each with 64 iterations *(so 512 in all)*. **ptch** starts at -256 and during these iterations **ptch** is always increasing by 1 *(so it will increase by 512 in the end)*. During the first two loops **integrator** is increasing by 18'h00080. During the first loop **ptch_rt** is decreasing by 16'h0100. You can figure out the rest from there, or do it a completely different way.

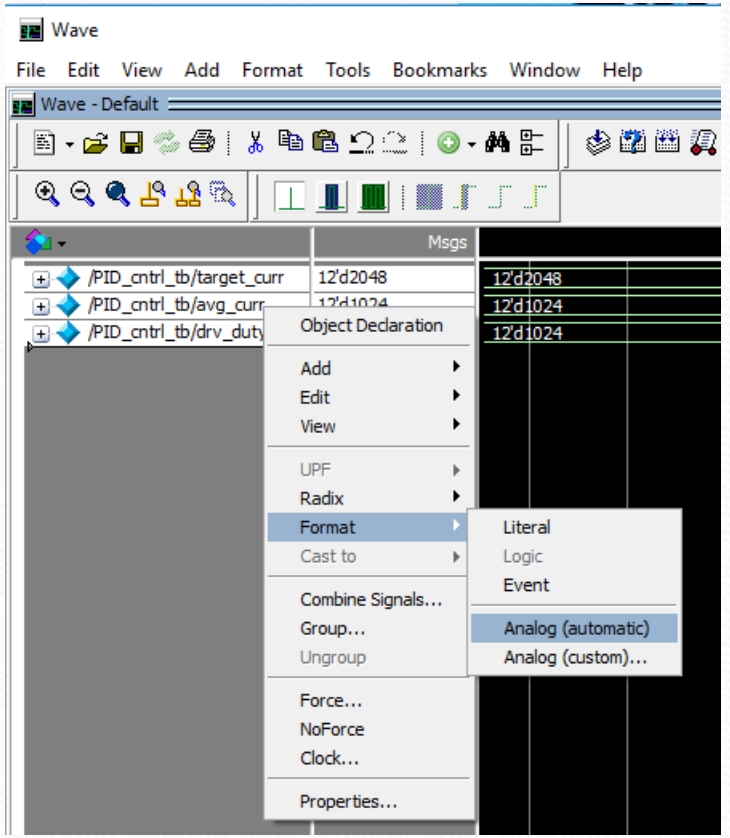
Submit what you have (**PID_Math.sv** & **PID_Math_tb.sv**) by the end of class. You need to finish this for HW2.

Viewing waveforms as analog



First select the signals of interest and change the radix (right click). If the signals are signed you would choose **Decimal**. If the signals are unsigned choose **Unsigned**.

If you have not already…run the simulation



Finally right click on the signals again and change the display format to “Analog (automatic)” It can’t automatically scale signals for which it does not have data, so you have to have run the simulation to do this.