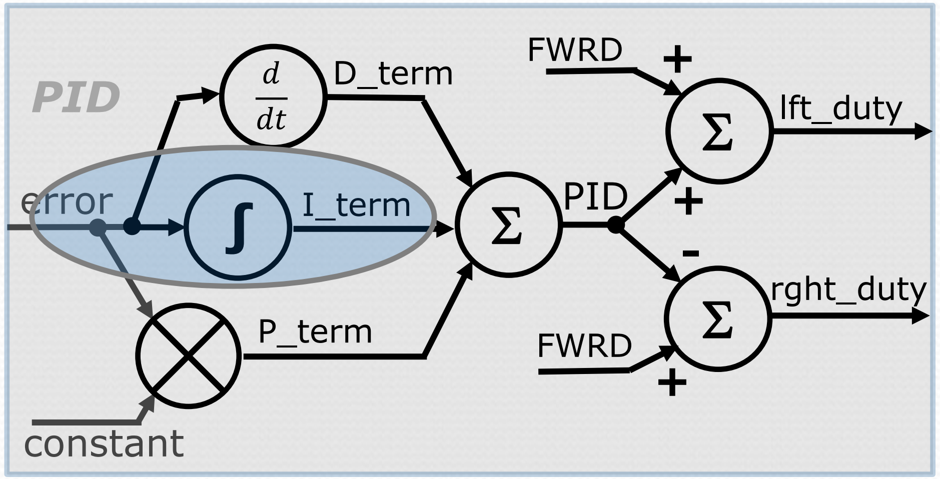
Exercise 09: I_term Datapath Design & Test



If **error** is positive it means our heading is too far CCW of desired so we should drive the left motor harder, and the right motor softer.

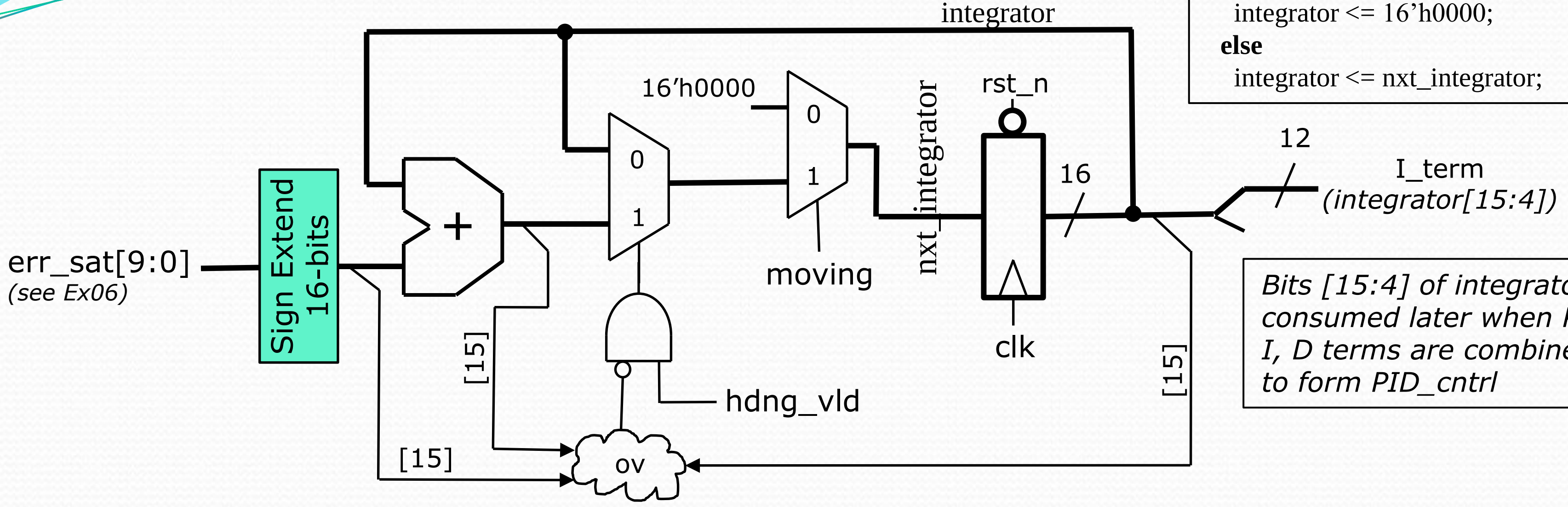
Therefore the PID control is added to the forward (FWRD) speed of the left motor, and subtracted from the forward speed of the right motor.

Proportional **I**ntegral **D**erivative (PID) is a classic control scheme. If you are curious why/how it works research on your own, or take ECE332.

- Proportional is simply multiplication of the **error** by a constant (see Ex06)
- Integration is nothing more than summing over time, so this is implemented with an accumulator. This exercise will focus on developing the logic for the **I_term**
- A derivative can be approximated by how much a value changed over a given period of time, so this can be implemented by keeping track of previous values of **error**, and subtracting them from the current value of **error**. This will be done in a future exercise.

Exercise 09: I_term Datapath Design & Test

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    integrator <= 16'h0000;
  else
    integrator <= nxt_integrator;
```



Bits [15:4] of integrator consumed later when P, I, D terms are combined to form PID_cntrl

On every valid (**hdng_vld**) reading the saturated error (**err_sat**) is accumulated into a 16-bit accumulator register. We then use the upper bits ([15:4]) of this accumulator to form our I_{term} that summed with our P_{term} and D_{term} form **PID** control.

When the MazeRunner is not **moving** we don't want the integrator to get "wound up" so it is cleared whenever not **moving**. The MazeRunner cannot correct its error (by steering) if it is not **moving.**

We don't want to let the integrator roll over (either beyond its most positive number or it 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 MSB of 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.

The accumulator register should be inferred with a simple ***always_ff*** block that infers a 16-bit wide register that is asynchronously reset and takes on the value of **nxt_integrator**. **nxt_integrator** will be a 16-bit wide internal signal that you generate using ***assign*** statements.

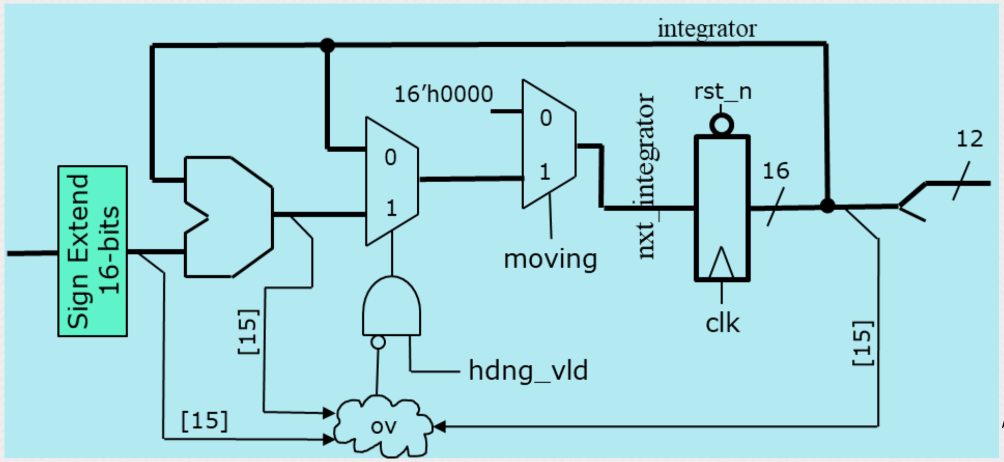
Exercise 09: I_term Datapath Design & Test

Signal:	Dir:	Description:
clk, rst_n	in	System clock and asynch active low reset
hdng_vld	in	A new **error** signal is valid and should be accumulated into I_term
moving	in	The MazeRunner is moving so PID should be active
err_sat[9:0]	in	The error of the course (heading – desired_heading)
I_term[11:0]	out	The I_term for eventual use in PID control

Implement ***I_term.sv*** with the above specified interface and the functionality outlined on the previous page.

The next page discusses testing it.

Exercise 09: I_term Datapath Design & Test



I_term_tb.sv

```
initial begin
  clk = 0;
  rst_n = 0;
  moving = 1;
  hdng_vld = 1;
  err_sat = 10'h1FF;
  @(negedge clk);
  rst_n = 1;   // deassert reset

  repeat (10) @(posedge clk);
  …
end

always
  #5 clk = ~ clk;
```

Build a testbench (***I_term_tb.sv***) and apply stimulus to **clk**, **rst_n**, **moving**, **hdng_vld**, and **err_sat**.

Initially assert **rst_n**, then deassert it *(goes high)* on the first negative edge of **clk.** *(see example code).*

Now test various scenarios including **moving** clearing the accumulator, **hdng_vld** not updating it, and freezing of value if **ov** occurs.