Exercise 21: Split Your Team (cmd_proc / cmd_proc_vld / charge)

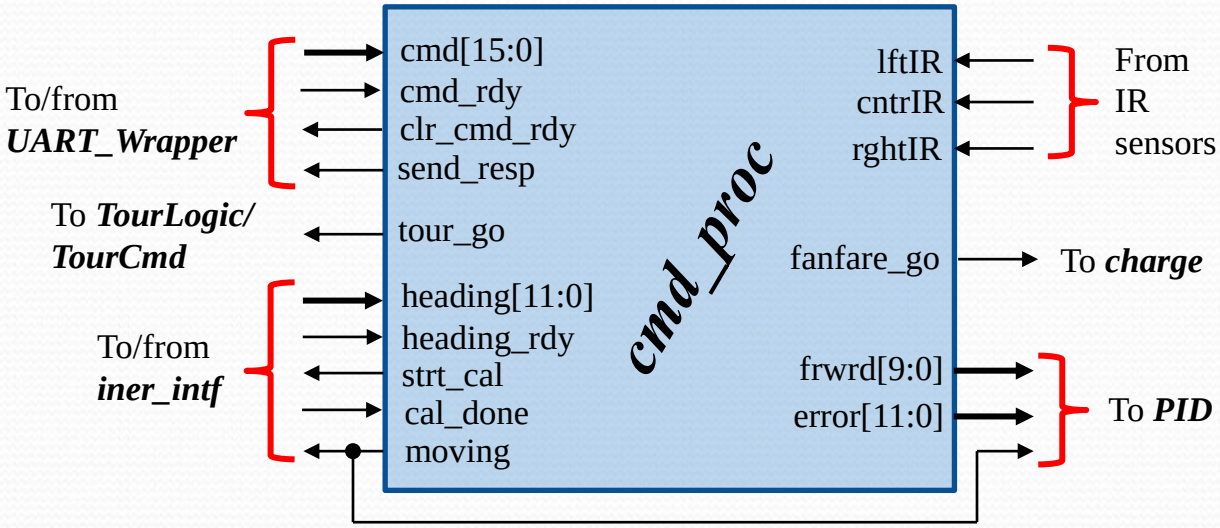
- Reccommended Split
  - 1 – 2 people on ***cmd_proc*** implementation.
  - 1 person gets started ***cmd_proc*** testbench
  - 1 person on ***charge*** fanfare

- For this exercise you will split your team and either work on cmd_proc & testing or on the unit to drive "Charge!" fanfare on the piezo buzzer.

- Decide how you are splitting roles and jump to the appropriate section of this document. ***cmd_proc*** is covered first.

Exercise 21: (cmd_proc)

The ***cmd_proc*** units main job is to process incoming commands from ***UART_Wrapper***. The command could be calibrate gyro, a movement command *(heading/num squares)*, or a complete tour command.

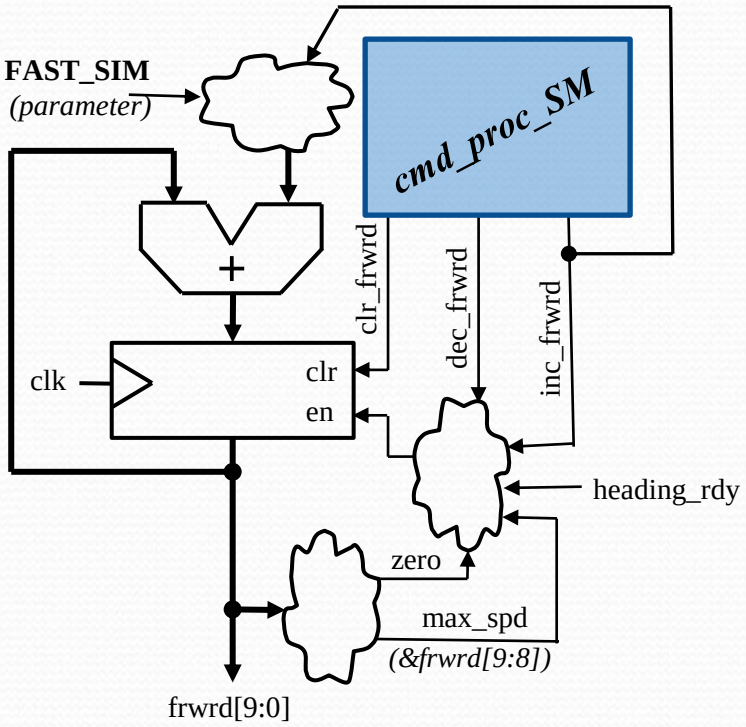


Since the ***cmd_proc*** knows the desired heading *(comes in via command)* and also knows the current **heading** *(via gyro)* it makes sense for it to generate the **error** term to the PID. Since it receives the movement commands it is also best suited to ramp up/dwn the **frwrd** speed register. We do not want jerky starts/stops, so the forward speed of the motors are ramped up and down *(controlled acceleration)* gradually.

The heading is not 100% accurate so there are guardrails formed by reflective tape and IR sensors. If a **lftIR** or **rghtIR** sensor throws a signal the course is nudged a bit. Also a **cntrIR** *(center)* gives 2 pulses for every square of movement, so this is used to determine when "The Knight" has moved the requested number of squares. When not **moving** neither the ***inertial_integrator*** or I_term in ***PID*** should integrate. Movements can *(optionally)* end with a fanfare played on the piezo

If the command is to complete the tour then **tour_go** is pulsed and commands come in from ***TourCmd*** instead of ***UART_Wrapper*** *(Bluetooth interface)*

Exercise 21: (cmd_proc) (frwrd register)



The **frwrd** speed register goes to the ***PID*** block and is added to the PID steering controls to determine the overall forward speed of the motors. The **frwrd** speed of the motors is never changed abruptly. Is is always ramped up to max_spd *(when the two MSB's of frwrd are 1)* and ramped back down to zero.

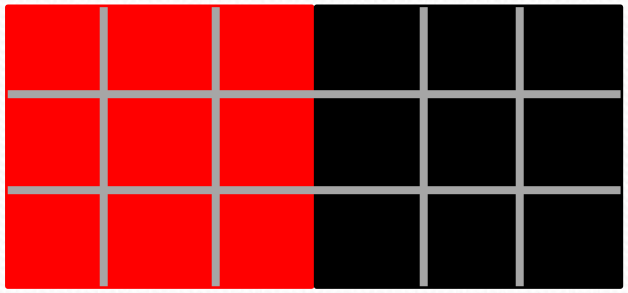
A move command should be thought of as 3 steps:

1. Set new heading and wait for error of heading to be small *(pointed right way)*
2. Ramp up frwrd till saturated at max_spd and count squares.
3. Ramp down frwrd register and come to stop *(optionally play "charge" fanfare).*

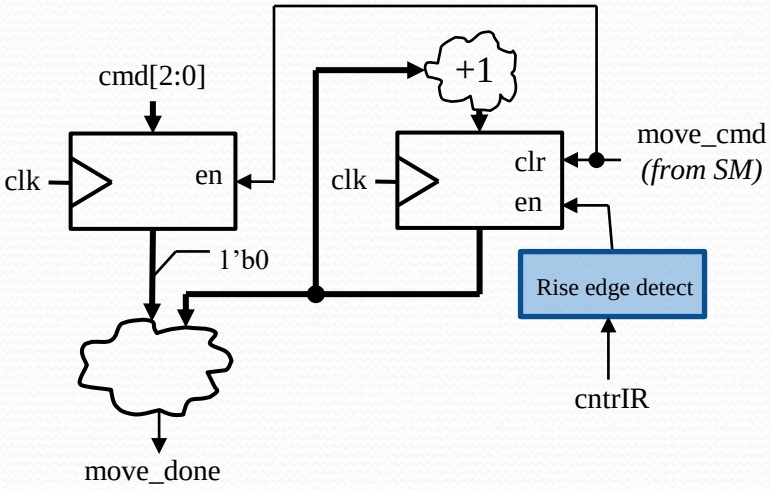
To speed up simulation ***cmd_proc*** should have a parameter ***(FAST_SIM)*** that is defaulted to 1. When **FAST_SIM** is a 1 then the increment amount of **frwrd** should be 0x20. When **FAST_SIM** is 0 then the increment amount of **frwrd** should be 0x04. The **frwrd** register is only incremented or decremented when there is a new **heading_rdy**. Also…the decrement amount should be twice the increment amount *(regardless of FAST_SIM we slow down twice as fast as we speed up).*

Although not shown here the **frwrd** register should have an asynch reset.

Exercise 21: (cmd_proc) (counting squares)

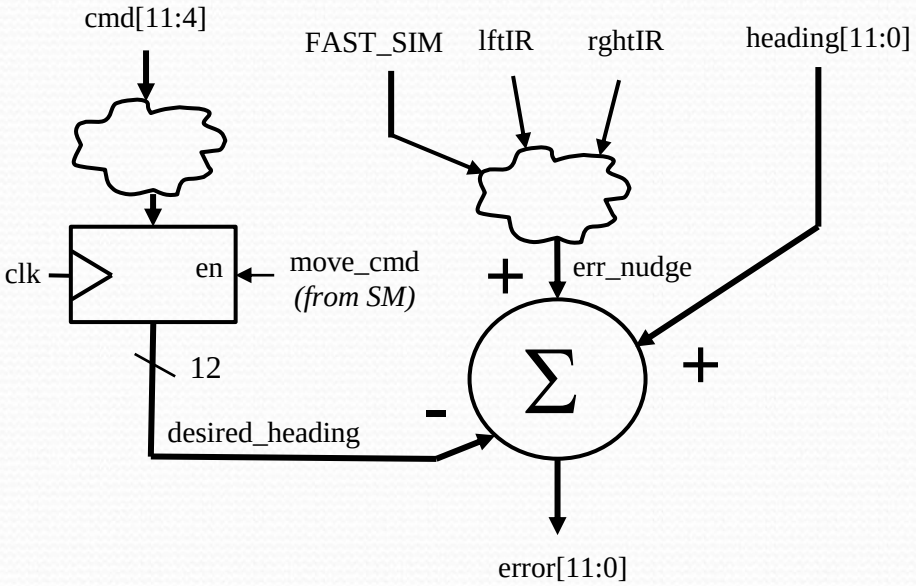


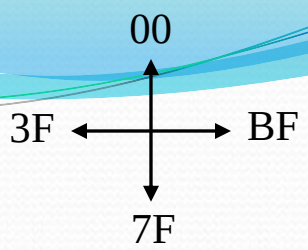
The squares of the board have reflective tape on them forming "guardrails" There are 3 IR sensors on the front of the robot (**lftIR**, **cntrIR**, **rghtIR**). The left & right IR's are used for course correction *(gyro is not perfectly accurate)*. When a left/right IR "sees" a line it nudges the robot back on course. The center IR sensor will see two pulses for every square it moves. Assume "The Knight" is currently in the middle of the red square, and is asked to move to the black square. The center IR will see a pulse as it leaves the red square, and another as it enters the center of the black square. Our deceleration *(ramp down of **frwrd**)* is quick enough that the robot will stop right in the middle of the black square if we start the ramp the instant we see the 2nd pulse on **cntrIR**.



When the "move" command is issued the number of squares to move is in **cmd**[2:0]. This should be captured and the center line counter should be cleared. The **cntrIR** signal will be present for multiple clock cycles, but we only want to increment once per line so put a rise edge detector on it. Remember you are comparing 2x number of squares to the center line counter to know if **move_done**.

Exercise 21: (cmd_proc) (PID interface)





When a move command comes in, **cmd**[11:4] specifies the heading. However, our heading is really a 12-bit value. If the desired heading is non-zero we will promote it 4-bits and append an F. For example if cmd[11:4] was 8'h00 then desired_heading would become 12'h000, however, if cmd[11:4] was 8'h3F then desired heading would become 12'h3FF.

If "The Knight" is wandering a bit too left and the **lftIR** guardrail signal fires we will nudge the robot a bit to the right by adding an additional 12'h05F when forming the **error** term. If "The Knight" is wandering a bit too right and the **rghtIR** guardrail signal fires we will nudge the robot a bit to the left by adding an additional 12'hFA1 *(adding negative value)* when forming the **error** term. **NOTE:** the nudge term is also subject to **FAST_SIM**. When **FAST_SIM**==1 **err_nudge** should be 12'h1FF or 12'hE00

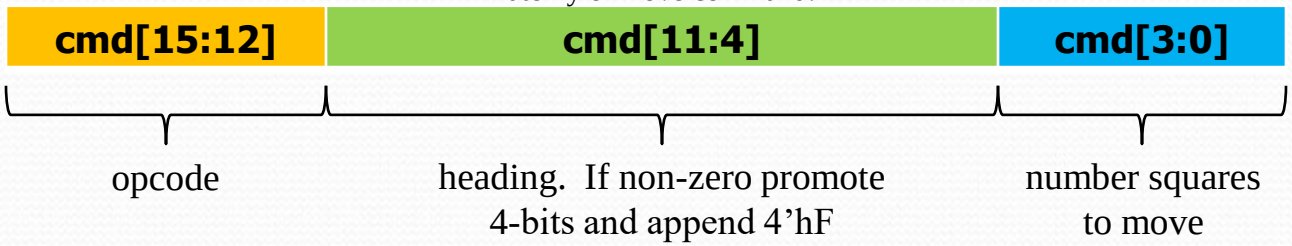
The **error** term then in general is: **error** = **heading** – **desired_heading** + **err_nudge** *(all are 12-bit values)*

Exercise 21: (cmd_proc) (command processing)

The main job of ***cmd_proc*** is to interpret incoming commands. The command is The command opcode is contained in **cmd**[15:12] there are currently only 4 commands "The Knight" can interpret.

Cmd[15:12]:	Description:
4'b0000	Calibrate command (kick off **strt_cal** and wait for **cal_done**). **send_resp**
4'b0010	Move command: bits[11:4] specify heading, bits [2:0] specify number of squares. A "move" is comprised of 3 steps: 1.) update desired_heading and wait for error<± threshold *(± 12'h030)* 2.) ramp up speed *(**frwrd** register)* and count squares till **move_done** 3.) ramp down speed and **send_resp**
4'b0011	Move with fanfare: Same as above except also initiate fanfare *("Charge" theme (assert **fanfare_go**))* when **move_done**.
4'b0100	Start Tour command: bits[7:4] specify the starting X and bits[3:0] specify the starting Y. Assert **tour_go** and return to ***IDLE***. Futher commands will now be sent by ***TourCmd*** instead of UART_Wrapper.

Anatomy of move command:

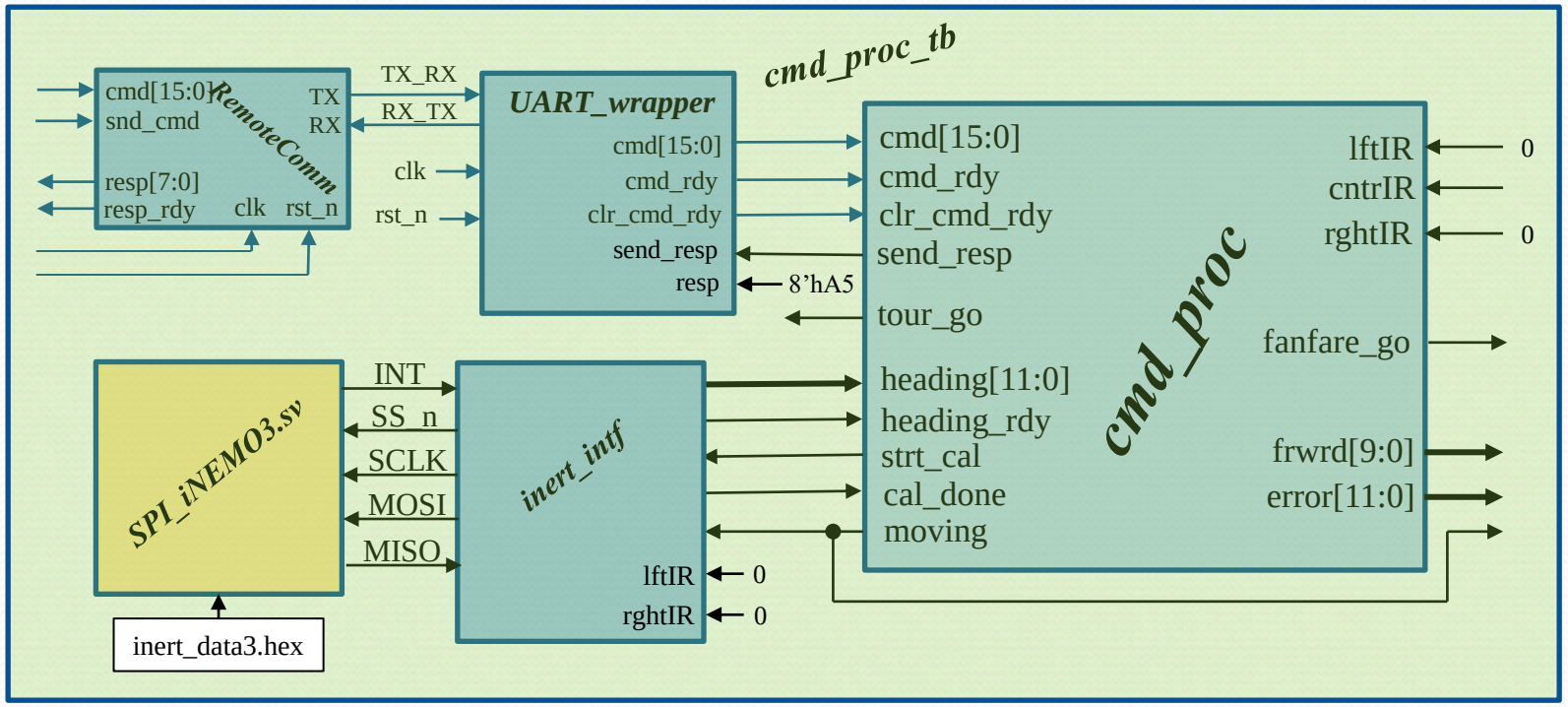


Exercise 21: (cmd_proc) (SM Control)

Extract the needed states from this description:

- The machine sits in ***IDLE*** until there is a **cmd_rdy.** Then it should dispatch to the appropriate state based on the opcode (**cmd**[15:12]). It should immediately **clr_cmd_rdy,**
- If the command is to calibrate is simply asserts **strt_cal** and waits for **cal_done.** It should also **send_resp** *(0xA5 positive ack)* when **cal_done**.
- If the command is to move it must capture the desired heading and then wait for the ***PID*** to do its job. Once the error *(a signed quantity)* falls below ± 12'h030 then our heading is "close enough" that we can start to ramp up **frwrd** speed. Adjusting our heading qualifies as **moving**.
- Ramping up can be the same state as traveling at max speed since the **frwrd** register saturates. We count the squares as we are **moving** to determine **move_done**
- Once the appropriate number of squares have been traveled we ramp down speed and initiate a fanfare if the move was with fanfare. When the **frwrd** speed reaches zero we **send_resp.** The robot is still considered to be **moving** during the ramp down phase.
- If the command is start tour we simply switch command control to ***TourCmd*** by asserting **tour_go**, **send_resp** should **not** be sent.

Exercise 21: (cmd_proc) (Testing it)



Since ***cmd_proc*** is so central to the design it has an extensive interface. It is often easier to test blocks like this in the greater context of their environment. For this reason, I recommend creating a testbench with the support blocks shown above. Think of it as a start to your "fullchip" testbench.

You can apply a 16-bit command to **cmd[15:0]** into ***RemoteComm*** and then assert **snd_cmd** for one clock. This will send the command and you can check for its proper operation.

Exercise 21: (cmd_proc) (Testing it)

There is a lot to test with ***cmd_proc***, and some of it might be better tested at the actual "fullchip" level when a physics model of "The Knight" is in the loop. However, the following are tests you should consider performing at this level:

Sequence:	Description:
1	Establish all inputs to known values and assert/deassert reset
2	Send Calibrate command (0x0000). Wait for **cal_done** or timeout if it does not occur Wait for **resp_rdy** or timeout if it does not occur
3	Send command to move "north" 1 square (0x2001) • Wait for **cmd_sent**….then check if **frwrd**==10'h000 (it should) • Wait for 10 positive edges of **heading_rdy**…**frwrd** should now be 10'h120 (or possibly 10'h140). (increments every **heading_rdy**) • Check that **moving** signal is asserted at this time • Now wait for 20 or more positive edges of **heading_rdy**. **frwrd** should be saturated at max speed by now. • Now give it a pulse on **cntrIR** (like it crossed a line). Wait a bit. **frwrd** should remain saturated at max speed • Now give it a 2nd pulse on **cntrIR** (it crossed a 2nd line) **frwrd** should start ramping down at twice the rate it ramped up. • Move should eventually end when **frwrd** hits zero. Check for **resp_rdy** or timeout if it never occurs.

Exercise 21: (cmd_proc) (Testing it continued)

The previous table outlined what I would consider to be a minimum test of ***cmd_proc***. A further test could include:

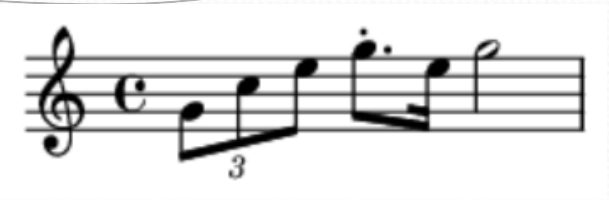
Sequence:	Description:
4	Send another move "north" 1 square command • Wait for it to be up to speed • Once it is moving hit it with either a **lftIR** or **rghtIR** pulse (many clocks in width). Do you see a significant disturbance in **error**? (you should)

Due to limitations in our model at this time (will be corrected when we get to "fullchip" testing) we cannot test moves to any direction other than "north".

The ***cmd_proc*** SM has not been fully tested.

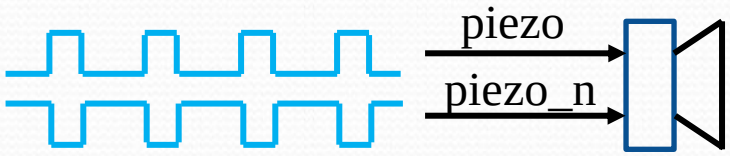
More thorough testing will occur when we test at the fullchip level with a complete physics model of "The Knight" and the chess board.

Exercise 21: (Charge! Fanfare)



- At the end of every "L" move "The Knight" will play the "Charge!" fanfare on the piezo buzzer.

Note:	Freq:	Duration:
G6 (G in octave 6)	1568	2^{23} clocks
C7 (C in octave 7)	2093	2^{23} clocks
E7	2637	2^{23} clocks
G7	3136	$2^{23} + 2^{22}$ clocks
E7	2637	2^{22} clocks
G7	3136	2^{24} clocks



A piezo bender is a "speaker" that can be driven with the GPIO's of our FPGA. We simply drive with a square wave of the frequency we want to generate a tone for.

The duty cycle is not so important (anything from 20% to 80% will do). We will drive differentially for increased amplitude.

The interface of the block should be as shown in the table. The implementation will require a couple of counter/timers (frequency & duration) and a controlling SM.

charge.sv Interface:

Signal:	Dir:	Description:
clk,rst_n	in	50MHz clk
go	in	initiates "tune"
piezo, piezo_n	out	Differential piezo drive

Exercise 21: (Charge! Fanfare)

- The duration of the notes (2^{23} clock cycles) is quite long for simulation purposes *(especially when we get to fullchip simulations where **charge.sv** will be one of many blocks being simulated).*

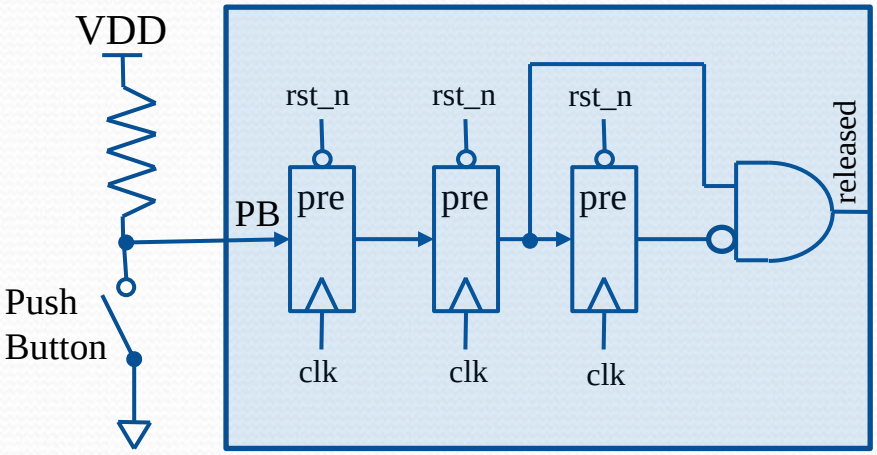
- We will need a method to speed up simulations. ***charge.sv*** should employ a parameter (called **FAST_SIM**). **FAST_SIM** should be defaulted true. When **FAST_SIM** is passed a 0 then durations should be as specified. When **FAST_SIM** is true then the duration of notes should be 1/16 their normal length. **HINT**: use a **generate if** statement to create an amount by which you increment your duration counter. Incement by 1 or by 16 depending on **FAST_SIM**.

- Create a simple testbench (***charge_tb.sv***) that simply instantiates the DUT, applies clock and reset and asserts **go**. Note **piezo** and **piezo_n** should not be toggling before **go** is asserted or after the "tune" has completed.

- Once your testbench is passing (visual inspection of **piezo/piezo_n**) move on to the next portion (testing with DE0).

Exercise 21: (Charge! Fanfare…testing on DE0)

We need a block to synchronize a push button switch and then perform rising edge detection on it.



PB_release.sv

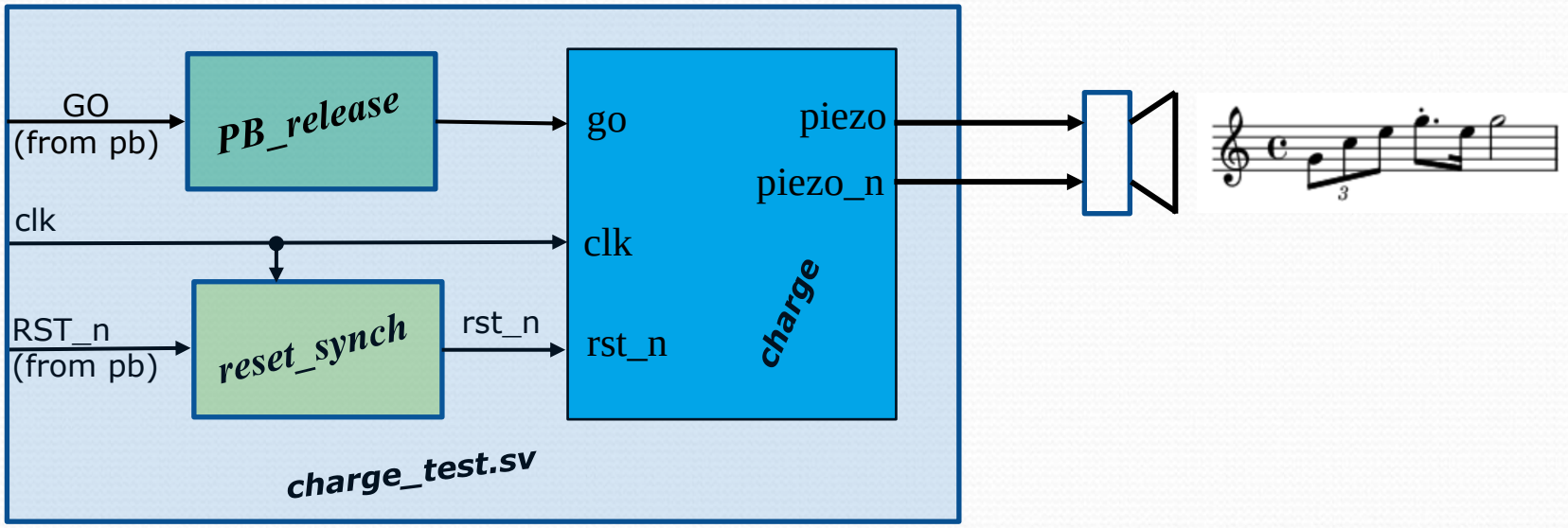
NOTE: these flops are asynch preset not reset

Study this…does it make sense?

- The first two flops are just double flopping for meta-stability purposes. PB is asych input right?
- Third flop is used to implement a rising edge detector. If the input to 3rd flop is high, but it output is low then a rising edge must be coming through (i.e. the release of the button.)

Implement this in system Verilog. Call it: **PB_release.sv**

Exercise 21: (Charge! Fanfare…testing on DE0)



- Create **charge_test.sv** to map your **charge.sv** to the DE0 and test in "real life". Don't forget to pass a FAST_SIM parameter of 0 to charge since we are testing it on real HW now.

Signal:	Dir:	Description:
clk	in	50MHz clock
RST_n	in	Unsynchronized input from push button
GO	in	Raw PB version of go
piezo/piezo_n	out	Differential drive of piezo bender

Exercise 21: (Charge! Fanfare…testing on DE0)

- There are Quartus project file and settings file available for download: (**charge_test.qpf, charge_test.qsf**).
- Open the .qpf and **ensure you add** all necessary files to the project.
- Ensure the project builds in Quartus with no errors
- Once it does call Eric or Tommy over for a demo with the DE0.
- All groups must demo a working **charge.sv** block