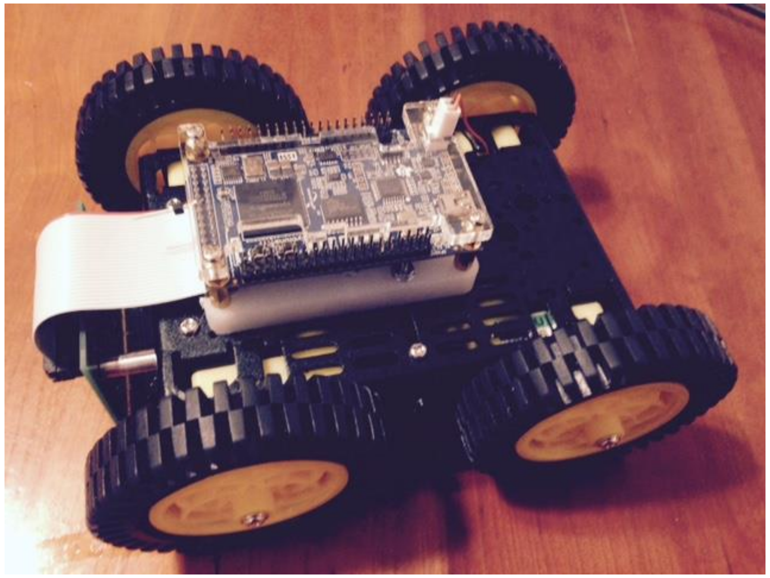
ECE 551

Project Spec



Spring ‘15

Line Following Robot

Grading Criteria: (Project is 30% of final grade)

- Project Grading Criteria:
  - Quantitative Element 20%

(yes this could result in extra credit)

$$Quantitative = \frac{EricKushagra\_ProjectArea}{YourSynthesizedArea}$$

Note: The design has to be functionally correct for this to apply

  - Project Demo (80%)
    - ✓ Code Review (10%)
    - ✓ Testbench Method/Completeness (12.5%)
    - ✓ Synthesis Script review (10%)
    - ✓ Post-synthesis Test run results (12.5%)
    - ✓ Results when placed in EricKushagra Testbench (20%)
    - ✓ Run of the robot on the track (15%)

Extra Credit Opportunity:

Appendix C of ModelSim tutorial instructs you how to run code coverage

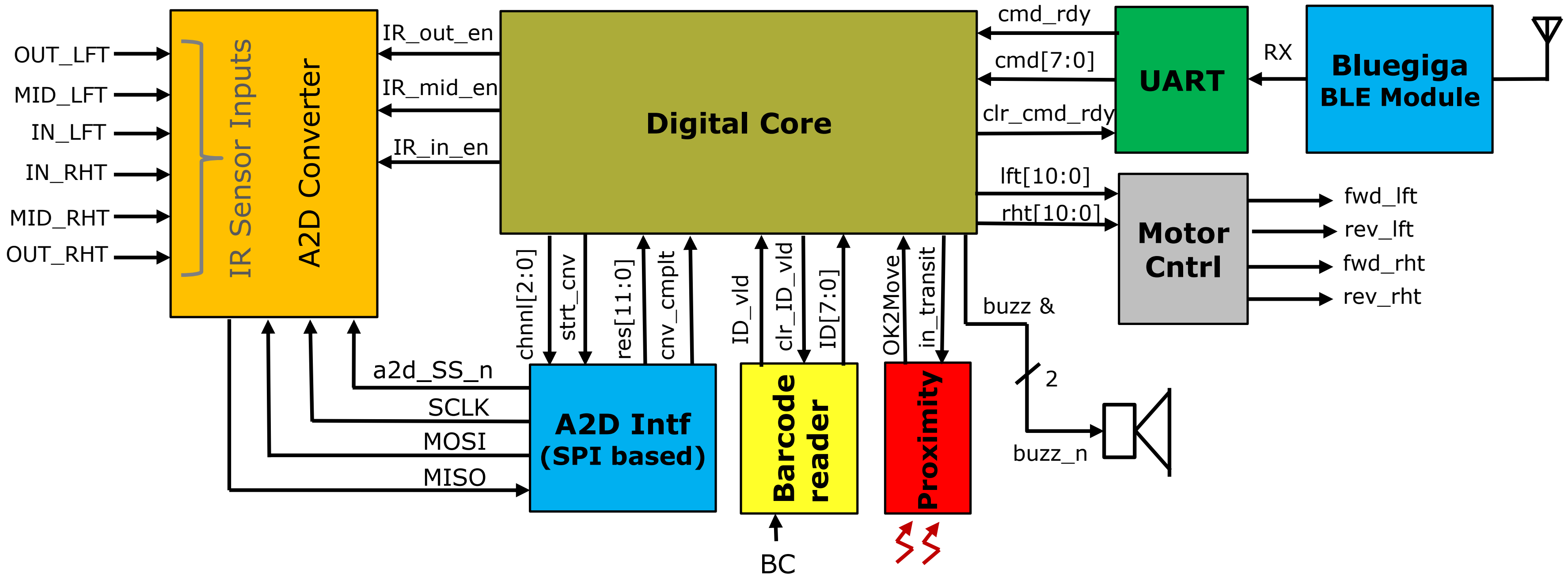
- Run code coverage on a single test and get 1.25% extra credit
- Run code coverage across your test suite and get a cumulative number and get 2.5% extra credit.
- Run code coverage across your test suite and give **concrete** example of how you used the results to improve your test suite and get 3.75% extra credit.

Project Due Date

- Project Demos will be held in B555:
  - Wednesday (5/6/15) from 1:00PM till evening.
    - ✓ 1.25% Extra Credit for demoing on Wednesday
  - Friday (5/8/15) from 1:00PM till evening.

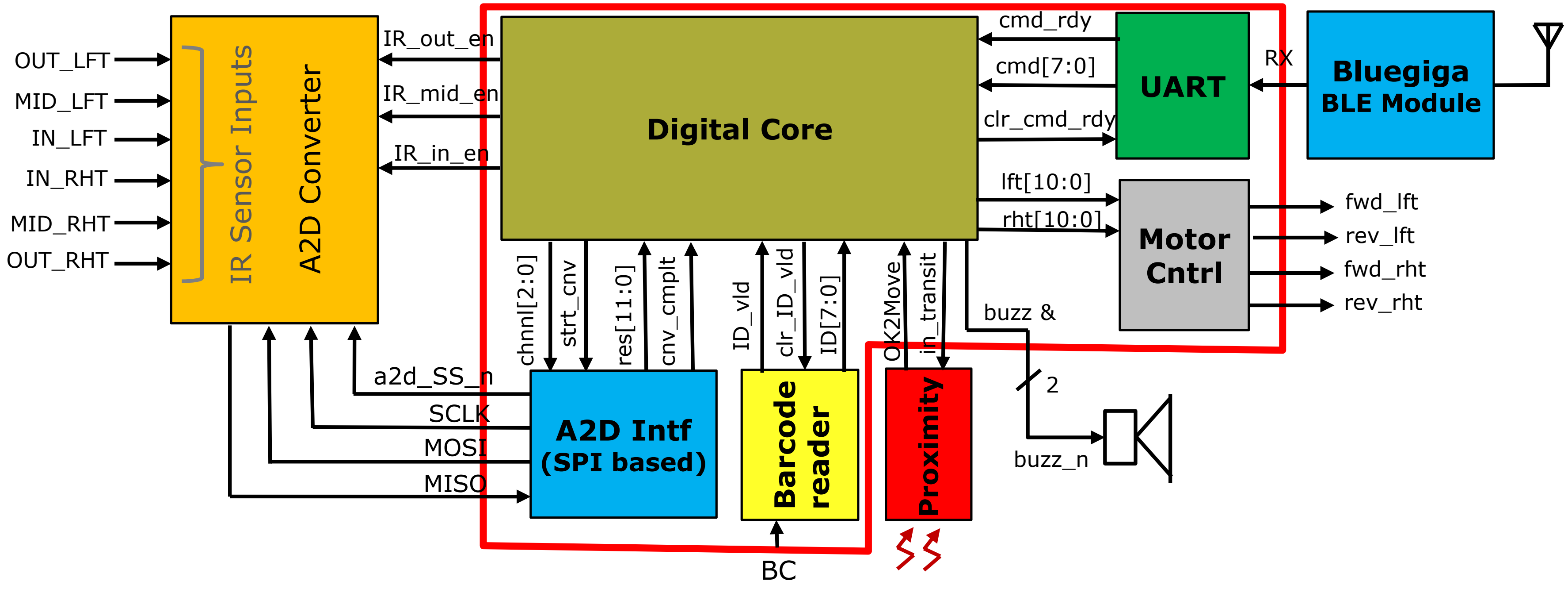
- Project Demo Involves:
    - ✓ Code Review
    - ✓ Testbench Method/Completeness
    - ✓ Synthesis Script & Results review
    - ✓ Post-synthesis Test run results
    - ✓ Results when placed in EricKushagra testbench
    - ✓ Run of your code in the Bot on the "track of death"

Block Diagram of Digital Portion



An array of six IR sensors will be digitized by the A2D converter and the readings will be mathematically combined to form a signed error signal *(Positive if too far left of the line, negative if too far right of the line)*. This error signal will be used in a PID control algorithm to determine the drive to each motor pair (left/right) to steer the follower. The digital core provides an 11-bit signed number for each motor pair *(left/right)* that represents the magnitude and direction *(forward/reverse).* The Motor Cntrl block converts these numbers into PWM signals that then drive the motors *(through an external driver chip)*. There are stations along the path of the line that will have "barcodes". A 7th IR sensor *(that sits to the extreme left of the follower)* is configured to give a serial bit stream as the follower drives over a station ID barcode. The follower will receive a command from the Bluetooth module *(sent via UART)* to go to a specific station and stop there. There is also a forward looking proximity sensor *(looks 10cm ahead).* If the path is clear it asserts OK2Move. If this signal falls the follower should hit the brakes and buzzes the piezo buzzer.

What is synthesized DUT vs modeled?

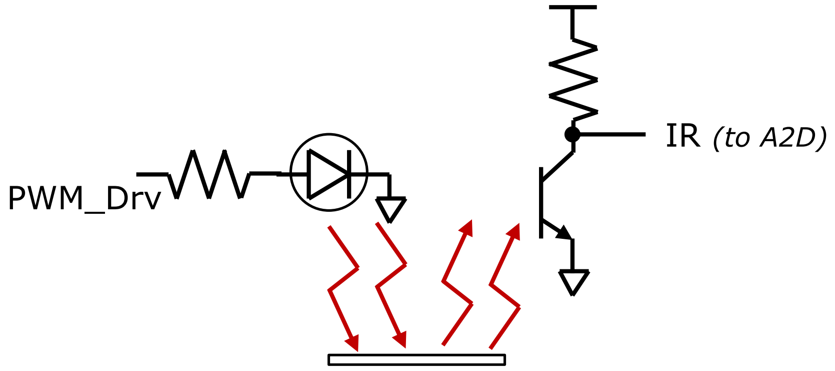


The blocks outlined in red above are pure digital blocks, and will be coded with the intent of being synthesized via Synopsys to our TSMC standard cell library. For practical purposes we will also map that logic to FPGA so we can run the demos.

You Must have a block called **follower.v** which is top level of what will be the synthesized DUT.

IR Sensor Theory of Operation:

The IR sensors being used are of the emitter/detector style.

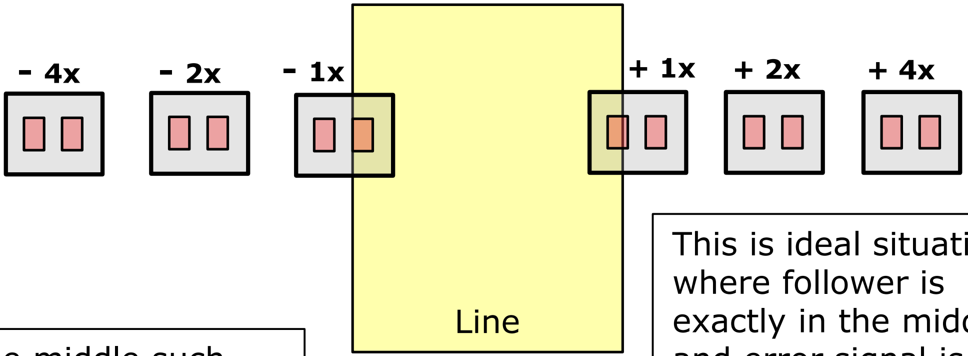


An LED emitter and detector come packaged in a single component. The infrared emitter *(basically a LED)* emits light. This light then reflects off a near by surface *(the white tape in our case)* and returns to the detector. The detector is a photo-transistor who's conductivity increases with the amount of photons incident on its base region. The photo-transistor is used in a pull-down roll, so the resulting signal is inversely proportional to the reflectivity of the surface. We will be using a white line on a non-reflective surface, so the line will give lower values from the A2D.

The strength of the return signal *(reflected infrared light)* is directly proportional to the amount of infrared light emitted by the emitter. This can be controlled by the resistor value, but can also be controlled by the duty cycle of the PWM waveform we drive the emitter with. This allows us to use a small value resistor, and then modulate the drive strength of the emitter with our code. Use of a smaller value resistor and PWM drive also lowers the overall power consumption of the IR circuit compared to just driving it 100% with a higher value limiting resistor.

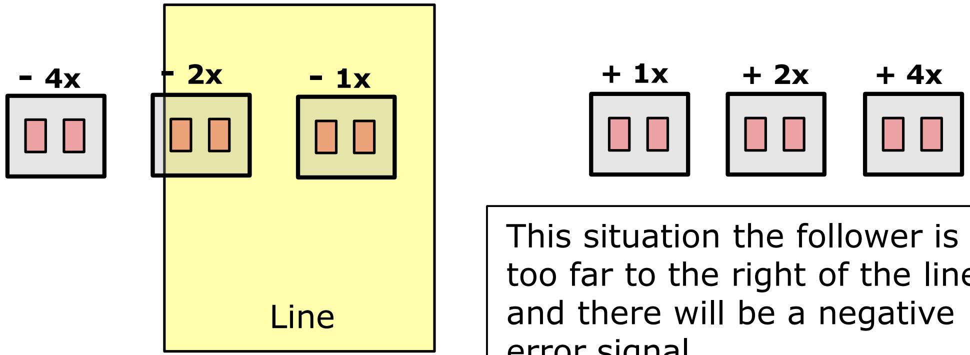
IR Sensor Array:

This is a view looking down at the follower as if it was transparent. We see the array of IR sensors, and our reflective line below that. When calculating our error signal **we invert our IR readings**. So the line gives high readings and the floor give zero readings.



This is ideal situation where follower is exactly in the middle and error signal is zero

Six sensors are arrayed out with a gap in the middle such that the line *(formed with 1/4 inch wide white tape)* is just wider than the spacing between the inner sensors.



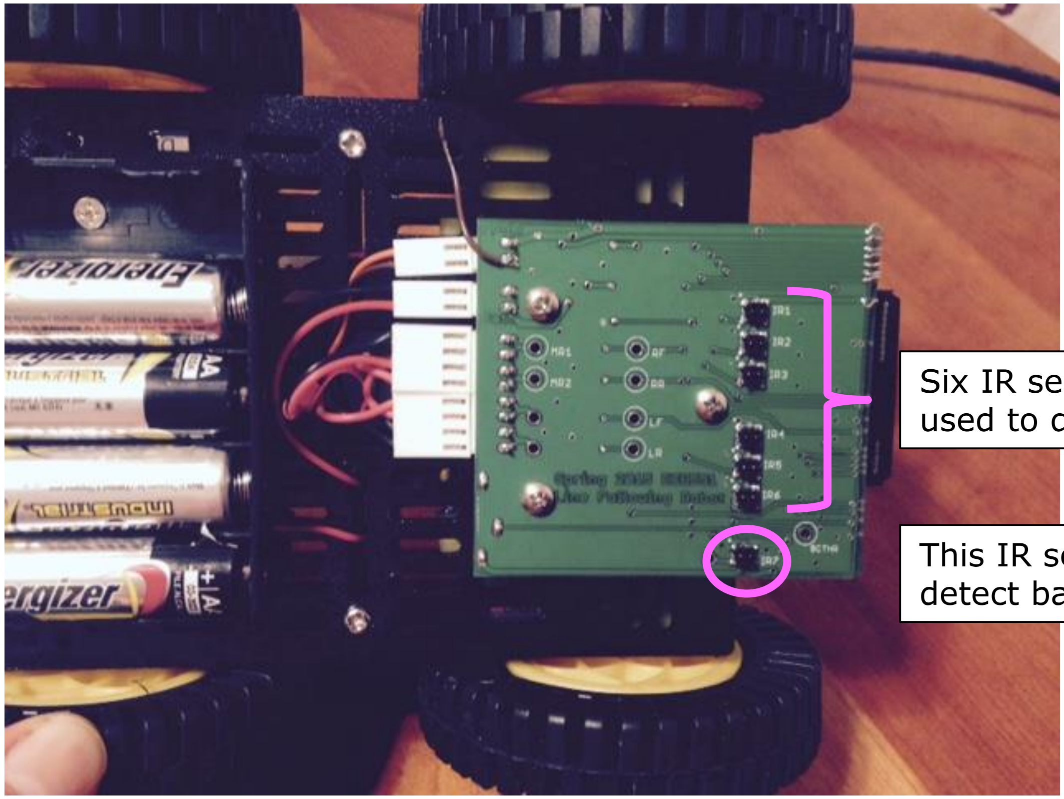
This situation the follower is too far to the right of the line and there will be a negative error signal

Note: the width of the line is such that at most only two sensors are over the line at any given time.

The left sensors are weighted negative and the right sensors are weighted positive. Their weights are by powers of two so the error signal increases in magnitude the further off course the follower is.

IR Sensor Array:

Bottom Side view of follower



Six IR sensors in an array used to detect line.

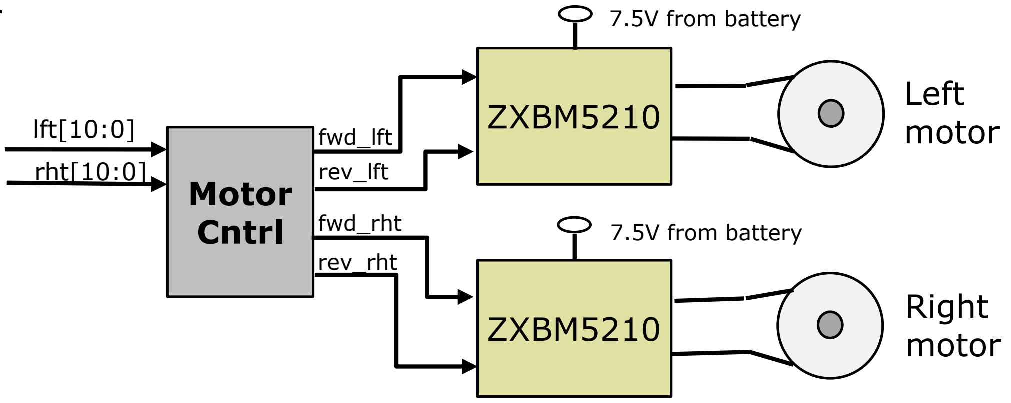
This IR sensor is used to detect bar code.

Motor Drive Controller

The motor controller receives a **signed** 11-bit number for both right and left motors. This will be converted in the motor controller to sign/magnitude and used to generate the PWM controls to the actual motor driver chips (ZXBM5210)

This implies (for both right and left):

- 1023 reverse speed settings
  - 0x400 maps same as 0x401 to a full reverse drive.
- 1023 forward speed settings
- Complete stop (0x000) brake mode

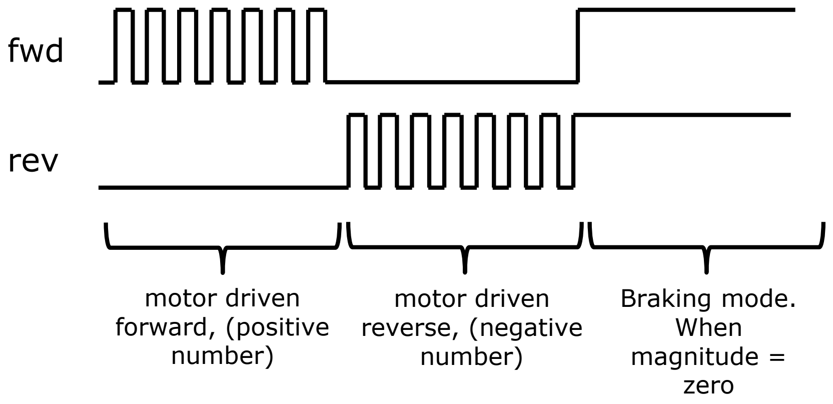


The motor controller produces a ***fwd*** and ***rev*** PWM signal for each motor.

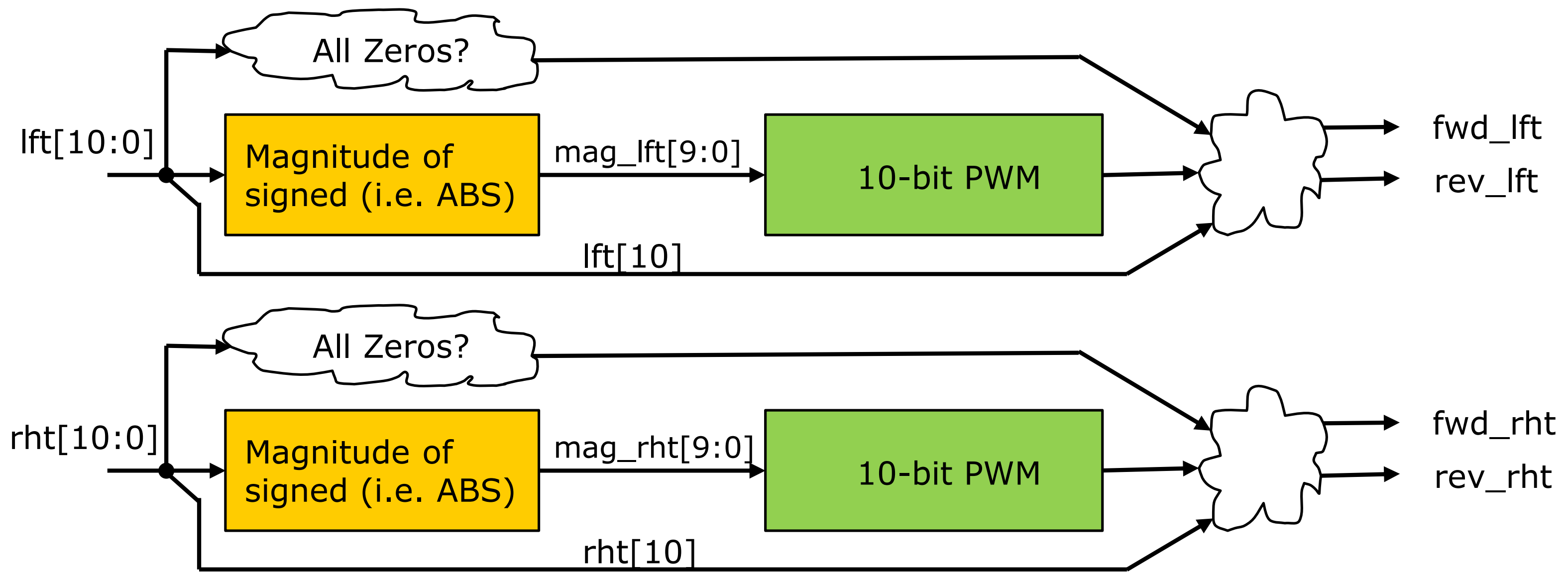
- To go forward, ***fwd*** is driven with a PWM signal with a specified duty cycle, and ***rev*** is driven low.
- To go in reverse, ***rev*** is driven with a PWM signal with a specified duty cycle and ***fwd*** is driven low.
- To hit the brakes *(which you do if the prox sensor deasserts* ***OK2Move****)* both ***fwd*** and ***rev*** are driven high.

The motor controller converts both ***rht[10:0]*** and ***lft[10:0]*** into their respective ***fwd*** and ***rev*** 10-bit PWM signals.

NOTE: The PWM module is not capable of producing zero duty cycle drive. If the magnitude is 0x000 then the PWM output is overridden and braking mode is used.


fwd
rev
motor driven forward, (positive number)
motor driven reverse, (negative number)
Braking mode. When magnitude = zero

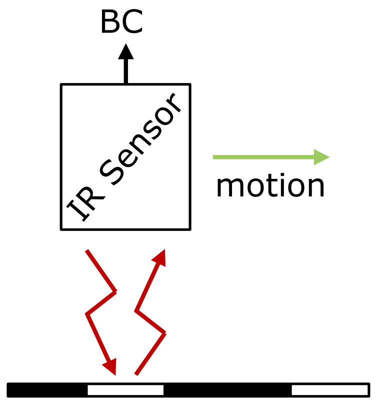
Possible Architecture of Motor Drive Controller



The inputs are 11-bit signed numbers. We need to just get the magnitude of the numbers and pipe that into a 10-bit PWM. The output of the 10-bit PWM will then be routed to **fwd_*** if the number was positive and to **rev_*** if the number was negative.

If the input was all zeros (this implies braking) then we want to make both **fwd_*** and **rev_*** go high.

Barcode Reader (Station ID)



An additional IR sensor is mounted on the bottom of the follower separated from the sensor array used for line following. This sensor's analog output runs into a comparator and forms a digital signal (***BC***) that will provide a barcode input.

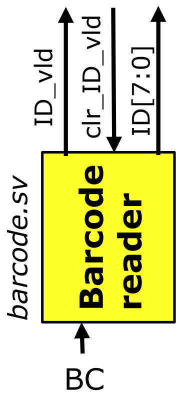
When the follower runs over a station ID (barcode) this signal will toggle in a pattern that follows the encoded station ID. This signal (***BC***) goes into a unit (*barcode.sv*) that will produce ***ID[7:0]*** and a signal called ***ID_vld*** from this signal. There is also an input to this module (***clr_ID_vld***) used to knock down the ***ID_vld*** output.

Of course the period of the pulses arriving on this BC signal will vary with the speed of the follower as it is passing over the station ID. Therefore the signaling protocol has to somehow encode timing information in with the data.

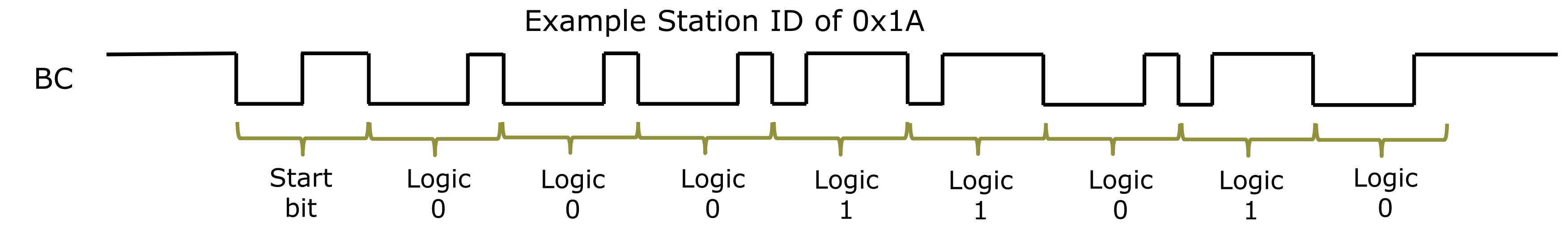
Station IDs are limited to straight sections of the course such that we know the follower is centered and traveling at a reasonably constant speed as it passes over the station ID.

Only the lower 6-bits of the ID are used as unique station ID identifiers. The upper 2-bits are used as an integrity check and **must be** 2'b00 for the ID to be considered valid.

Signal:	Dir:	Description:
BC	In	Signal from barcode IR sensor. Serial stream (8-bits in length) that has timing information encoded (see next slide).
ID_vld	Out	Asserted by barcode.sv when a full 8-bit station ID has been read, and the upper 2-bits are 2'b00. If upper 2'bits are not 2'b00 the barcode is assumed invalid.
ID[7:0]	Out	The 8-bit ID assembled by the unit, presented to the digital core
clr_ID_vld	In	Asserted by the digital core to knock down ID_vld. Digital core would assert after having grabbed the ID from this unit

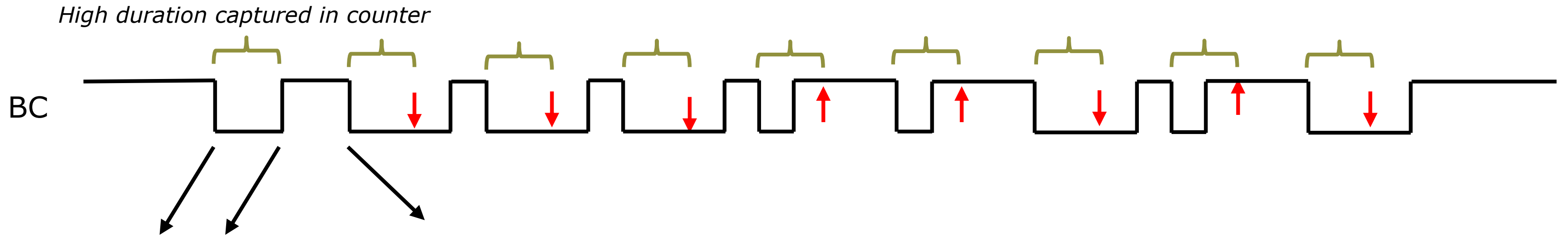


Barcode Reader (Station ID)



Timing information is built into the start bit. The signal has been high, then a start bit occurs. It is low for 50% of the duration of a bit period. The barcode unit (*barcode.sv*) times the duration of this low pulse of the start bit, and captures that value in a timer register.

Now for the next 8 subsequent falling edges of the BC signal the barcode unit will start a timer. When that timer matches the captured value of the start bit low period it will sample (and shift into a shift register) the value of the ***BC*** line. When finished the shift register will contain the 8-bit station ID. The MSB of the station ID is sent first (unlike UART protocol).



Barcode unit sees falling edge of start bit and starts a counter timing the low duration.

For the next 8 falling edges the barcode receiver will "see" the falling edge and start a counter. When the count value matches the captured low duration of the start bit it will shift the shift register, thus sampling the BC line value.

The red arrows indicate the times at which the BC line is (sampled) i.e. the shift register is shifted.

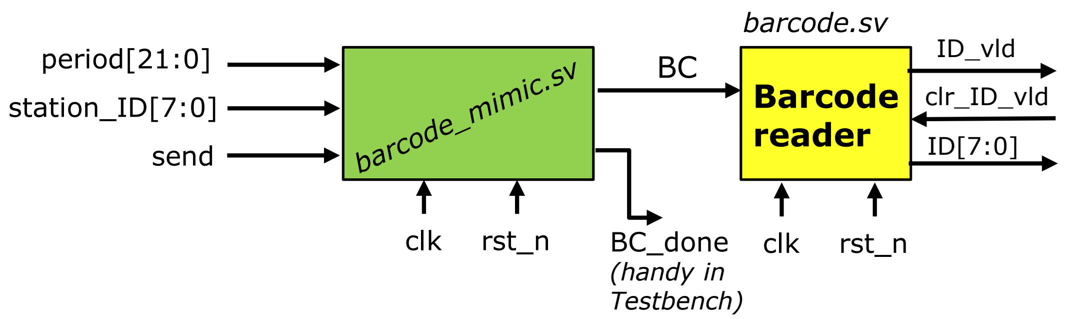
Barcode Reader

- How wide should the counters be in your barcode reader?

The duration of these signals depend on the speed of the follower as it passes over the station ID. We will assume a minimum speed of 0.5m/sec for the follower, and bit encodings of 2.5cm on the barcode. With the system running at 50MHz clock a safe value is 22-bits wide for the counters needed in your barcode receiver.

- How to test the barcode reader?

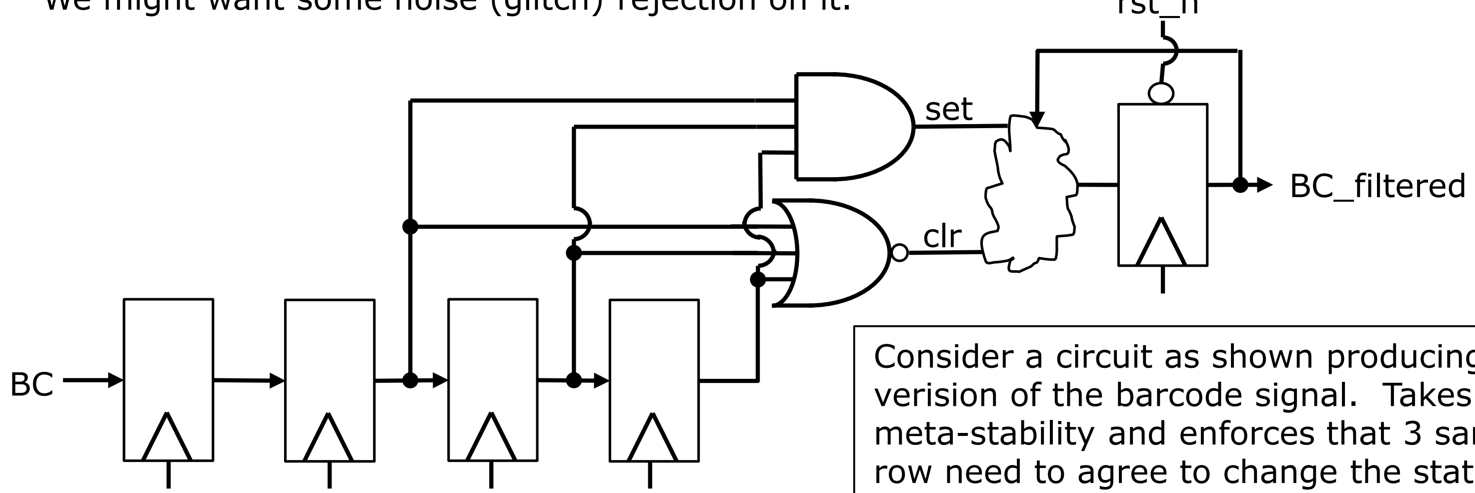
To facilitate testing of the barcode reader a module is provided (barcode_mimic.sv). This unit can be downloaded from the project area of the webpage. If you provide it with an 8-bit station ID and a 22-bit number indicating duration of pulses, then feed it a "send" pulse it will generate a BC bitstream that you can use to test your barcode reader.



Pick a value for ***period[21:0]*** (almost any value above 512 should work). Apply an 8-bit value to ***station_ID***, and hit barcode_mimic.sv with a pulse on ***send***. Once ***BC_done*** goes high the value on ***ID[7:0]*** should match the value placed on ***station_ID[7:0]*** and ***ID_vld*** should be high.

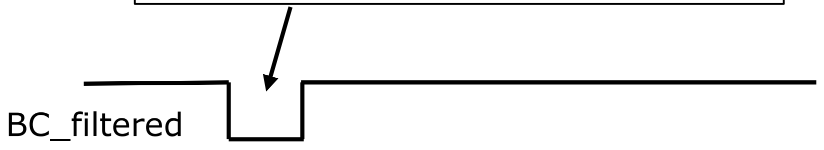
Barcode Reader…making it more robust

- Of course we know that BC is asynch to our clock domain so we should at a minimum double flop before use.
- However, there is also the matter that the BC signal is coming from an analog comparator output and has rather long rise/fall times (slew rate) relative to our 50MHz sampling period. We might want some noise (glitch) rejection on it.



Consider a circuit as shown producing a filtered verision of the barcode signal. Takes care of meta-stability and enforces that 3 sample in a row need to agree to change the state of the filtered signal.

Barcode reader gets a false pulse low



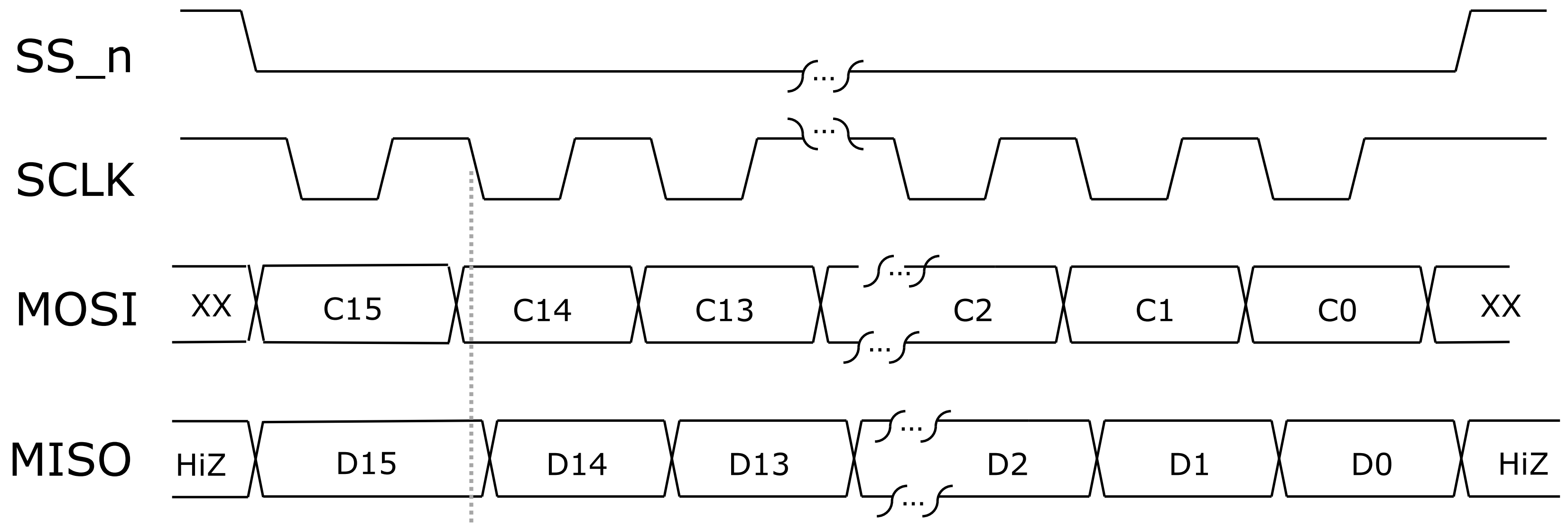
Another thing to consider would be the possibility that the barcode reader could pick up a false pulse low that it thinks is a start bit. This scenario should not get it stuck looking forever for the next falling edge. You should implement a timeout to return to IDLE if a subsequent falling edge does not occur in a “reasonable” amount of time. You can “reuse” your timer used for strobing mid bit. If it gets full you time out

What is SPI?

- Simple uni-directional serial interface (Motorola long long ago)
  - **S**erial **P**eripheral **I**nterconnect (very popular physical interface)
  - 4-wires for full duplex
    - ✓ MOSI (Master Out Slave In) (we drive this to A2D to inform what channel to read)
    - ✓ MISO (Master In Slave Out) (A2D sends data back over this line)
    - ✓ SCLK (Serial Clock)
    - ✓ SS_n (Active low Slave Select) (Our system only has a single slave, but in a system with multiple slaves this acts as a one hot selector of the active slave)
  - There are many different variants
    - ✓ MOSI Sampled on clock low vs clock high
    - ✓ SCLK normally high vs normally low
    - ✓ Widths of packets can vary from application to applications
    - ✓ Really is a very loose standard (barely a standard at all)
  - We will use the variant used by the A2D on the DE0_nano board.
    - ✓ MOSI changes on SCLK fall, and MISO is sampled on SCLK fall.

    (actually your SPI master should shift its shift register one system clock prior to SCLK fall)
    - ✓ SCLK normally high
    - ✓ 16-bit packets

SPI Packets

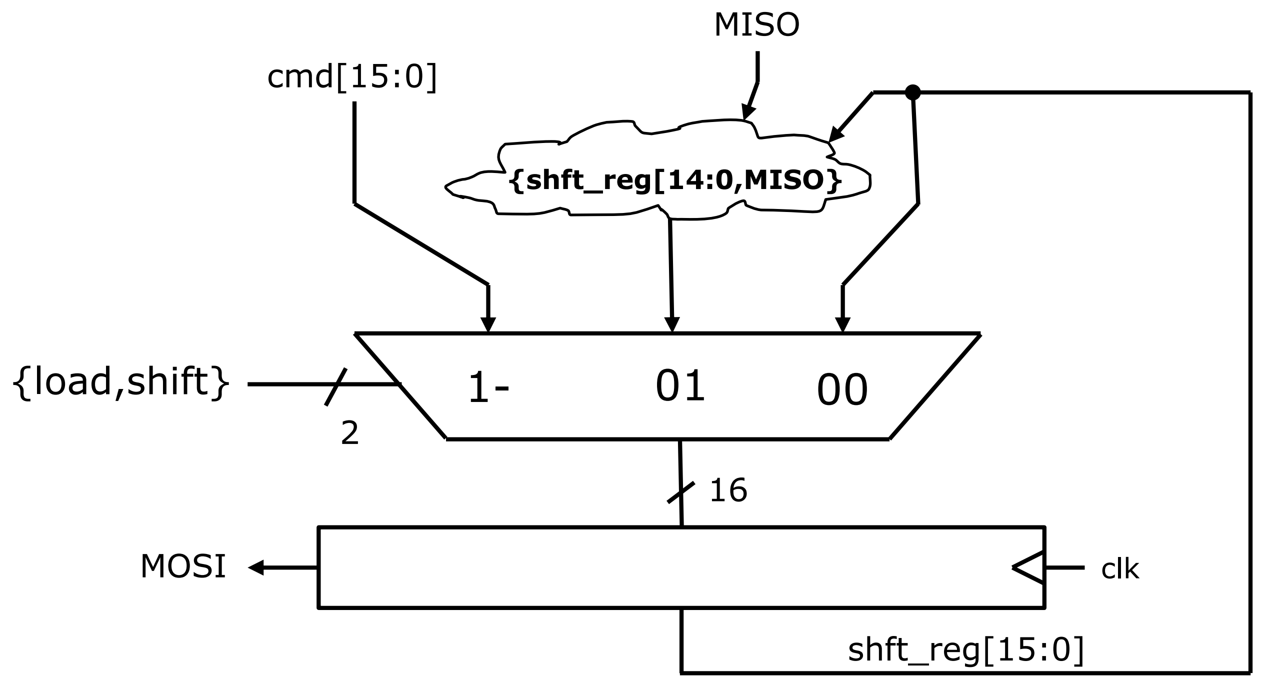


A SPI packet inherently involves a send and receive (full duplex). The full duplex packet is always initiated by the master. Master controls SCLK, SS_n, and MOSI. The slave drives MISO if it is selected. If the slave is not selected it should leave MISO high impedance. The A2D converter is the only SPI peripheral we have in the system.

The SPI master will have a 16-bit shift register. The MSB of this shift register is MOSI. MISO will feed into the LSB of this shift register. The shift register should shift **one system clock prior** to SCLK fall, this eliminates any timing difficulties. The A2D on the DE0-Nano samples MOSI on the positive edge of SCLK, and changes MISO on the negative edge of SCLK. Of course all your flops are based purely on clk (system clock), not SCLK! SCLK is a signal output from your SPI master.

SCLK will be 1/32 of our system clock (50MHz/32 = 1.5625MHz

SPI Shift Register



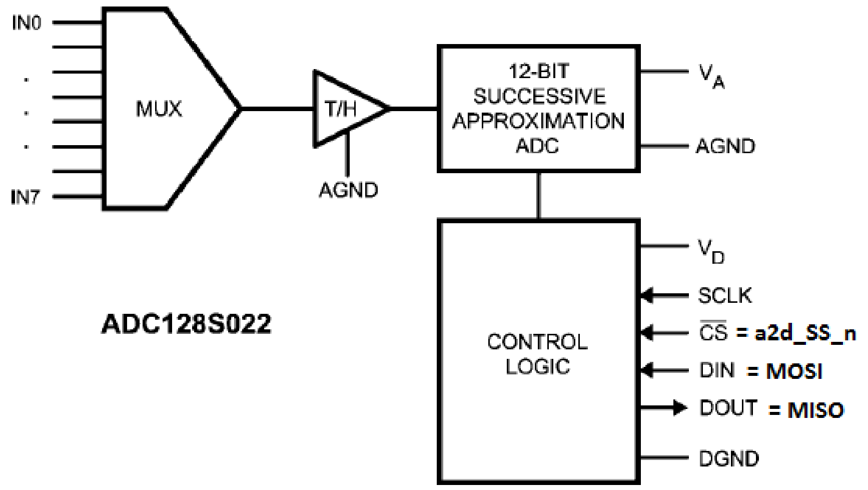
The main datapath of the SPI master consists of a 16-bit shift register. The MSB of this shift register provides MOSI. The shift register can be parallel loaded with the command to send to the A2D, or it can left shift one position taking MISO as the new LSB, or it can simply maintain.

Since the SPI master is also generating SCLK it can choose to shift this register in any relationship to SCLK that it desires. To alleviate timing difficulties it is best that the shift register is shifted one clock prior to SCLK fall.

A2D Converter (National Semi ADC128S022)

The ADC128S is a 12-bit eight channel A2D converter. Only one channel can be converted at a time. The A2D is read by via the SPI bus, and is used to convert the values of the six IR sensors.

ADC Channel:	IR Sensor:
000 = addr	IR_in_lft (inside left sensor)
001 = addr	IR_in_rht (inside left sensor)
010 = addr	IR_mid_lft (middle left sensor)
011= addr	IR_out_rht (outside right sensor)
100= addr	IR_mid_rht (middle right sensor)
111= addr	IR_out_lft (outside left sensor)



To read the A2D converter one sends the 16-bit packet {2'b00,addr,11'b000} twice via the SPI. So it is essentially a 32-bit SPI packet.

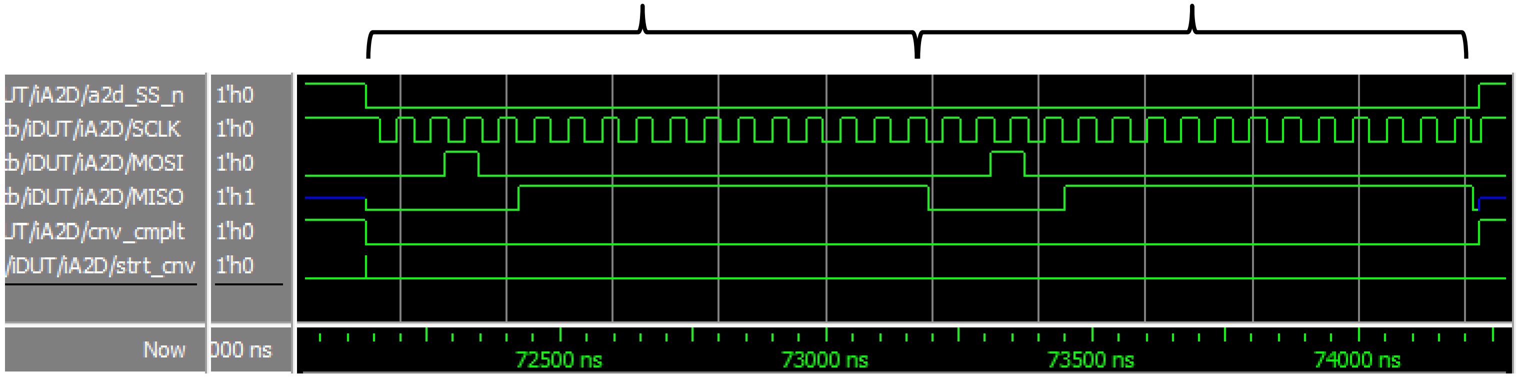
During the first 16-bits of this SPI transaction the value returned over MISO will be ignored. The first 16-bits are really setting up the channel we wish the A2D to convert. During the 2nd 16-bit packet the data returned on MISO will be the result of the conversion. Only the lower 12-bits are meaningful since it is a 12-bit A2D. The IR sensors will be read in a round robin fashion with the inside pair read first, the middle pair next, and finally the outside pair. After the IR sensors are read a new control value will be calculated and the PWM duty cycle to the right/left motors will be updated.

NOTE: The value your A2D_intf.sv should provide is the 1's complement (inversion) of what was actually read from the A2D over the SPI bus.

A2D Converter (Example SPI Read)

First 16-bit SPI transaction specifies The channel to perform conversion on. Data returned on MISO is junk.

Second 16-bit SPI transaction the data sent over MOSI does not really matter, just reading result over MISO.



Our use of the A2D converter will involve two 16-bit SPI transactions back to back (so it will look like one long 32 bit transaction).

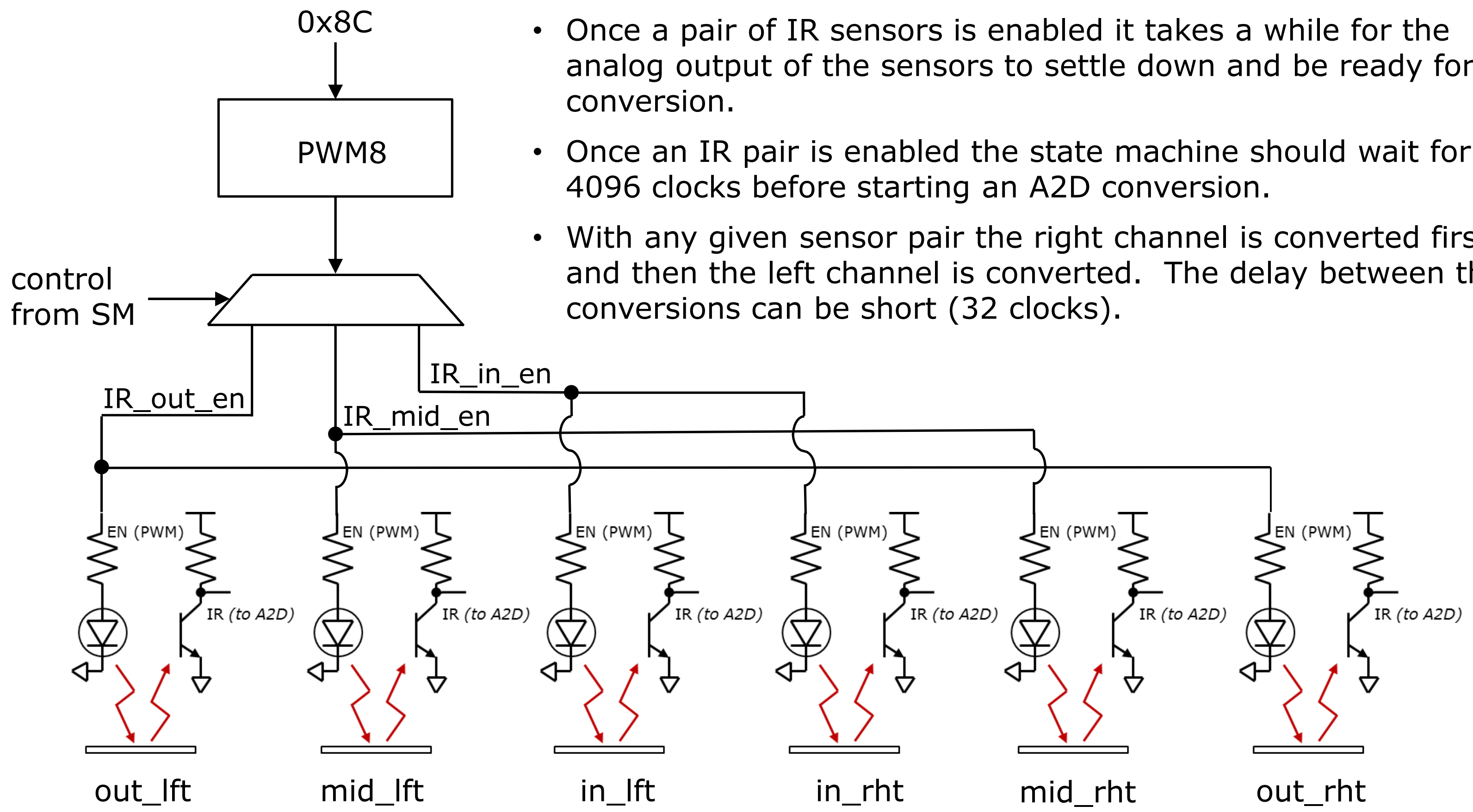
The first transaction here is sending a 0x2000 to the A2D over MISO. The command to request a conversion is {2'b00,channel[2:0],11'h000}. The upper 2-bits are always zero, the next 3-bits specify 1:8 A2D channels to convert, and the lower 11-bits of the command are zero. Therefore, the 0x2000 in this example represents a request for channel 4 conversion.

For the next 16-bit transaction the data sent over MOSI to the A2D does not matter that much. We are really just trying to get the data back from the A2D over the MISO line.

Note the timing of data vs SCLK edges. Note the behavior of SS_n. Note SCLK is normally high.

A2D Converter (IR enabling and settling time)

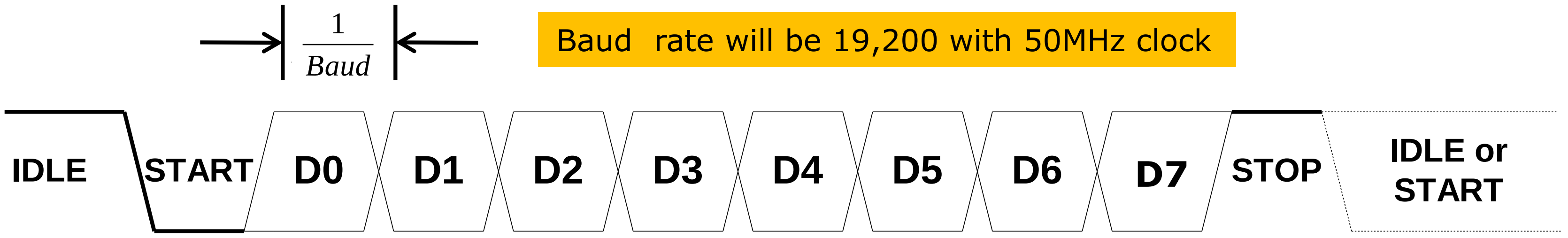
- The IR sensors are enabled in three pairs, inner, middle, and outer.
- The enables to these sensors should be driven with a PWM signal coming from an 8-bit PWM. The duty cycle of this signal will be a constant and is currently to be set to 0x8C.
- Creating an 8-bit PWM module should be trivial by modifing the 10-bit PWM in motor controller.



- Once a pair of IR sensors is enabled it takes a while for the analog output of the sensors to settle down and be ready for conversion.
- Once an IR pair is enabled the state machine should wait for 4096 clocks before starting an A2D conversion.
- With any given sensor pair the right channel is converted first, and then the left channel is converted. The delay between the conversions can be short (32 clocks).

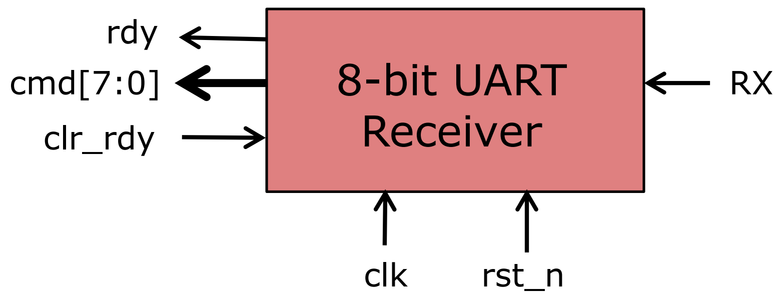
What is UART (RS-232)

- RS-232 signal phases
  - Idle
  - Start bit
  - Data (8-data for our project)
  - Parity (no parity for our project)
  - Stop bit – channel returns to idle condition
  - Idle or Start next frame



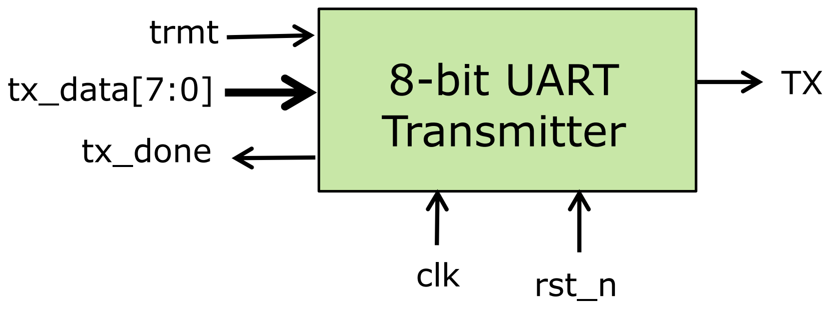
- Receiver monitors for falling edge of Start bit. Counts off 1.5 bit times and starts shifting (right shifting since LSB is first) data into a register.
- Transmitter sits idle till told to transmit. Then will shift out a 9-bit (start bit appended) register at the baud rate interval.

UART Receiver/Transmitter



Signal:	Dir:	Description
clk,rst_n	in	50MHz system clock & active low reset
RX	in	Serial data carrying command from BLE112 module
rdy	out	Asserted when a byte has been received
cmd[7:0]	out	Byte received (serves as command to follower)
clr_rdy	in	Asserted to knock down the cmd_rdy signal.

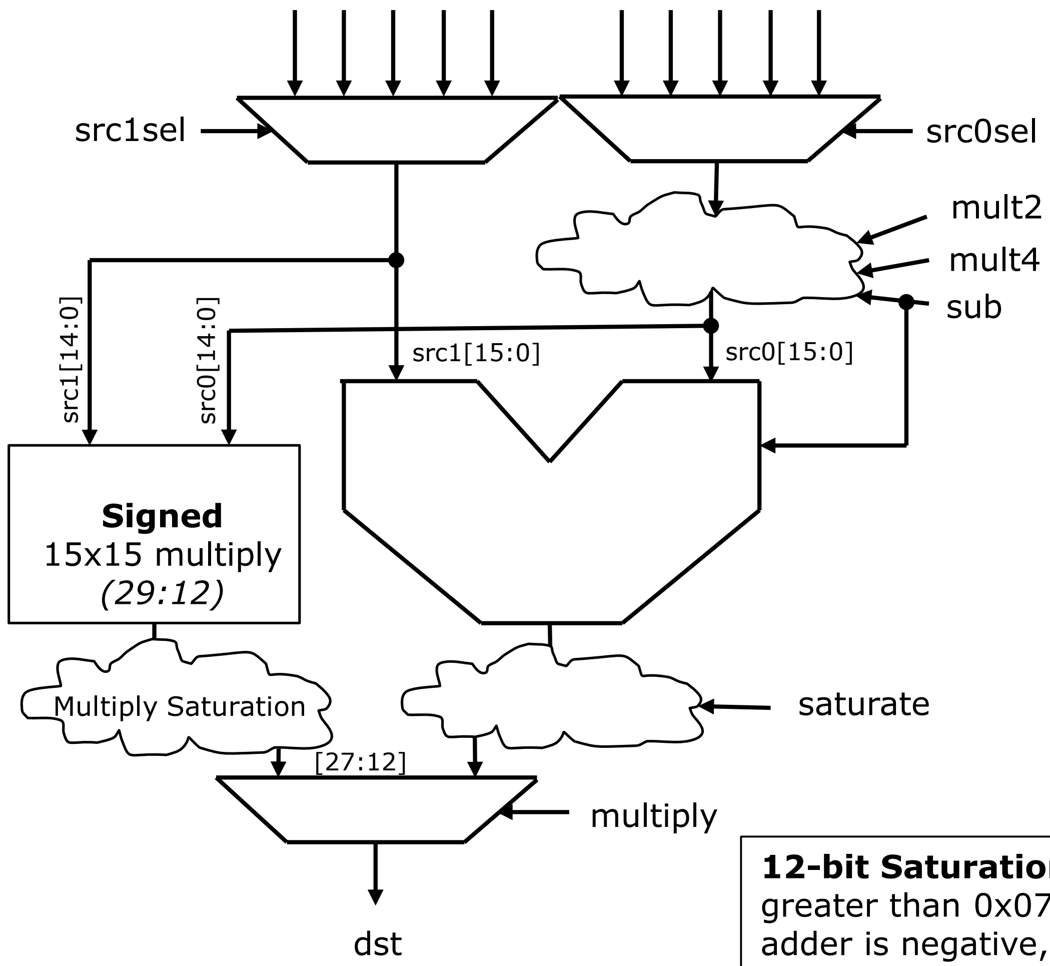
A Bluetooth Low Energy module (BLE112) on the follower will receive a command over the air and send it to the follower via RS232 protocol.



Signal:	Dir:	Description
clk,rst_n	in	50MHz system clock & active low reset
TX	out	Serial data output
trmt	in	Asserted for 1 clock to initiate transmission
tx_data[7:0]	in	Byte to transmit
tx_done	out	Asserted when byte is done transmitting. Stays high till next byte transmitted.

The follower design does not need a transmitter, however your testbench would be well served to have a transmitter so you can test your receiver.

ALU *(needed to perform PI math)*



Signal(s):	Width	Dir:	Note:
Accum, Pcomp	16	in	Full width
Pterm,	14	in	Unsigned
Fwd, A2D_res	12	in	Unsigned
Error,Intgrl, Icomp,Iterm	12	in	Signed
src1sel, src2sel	3	in	Use local params
multiply, sub,mult2, mult4, saturate	1	in	Control signals used to affect calculations
dst	16	Out	The result

12-bit Saturation Logic: If result of adder is positive, but greater than 0x07FF, then saturate to 0x07FF. If result of adder is negative, but less than 0xF800 then saturate to 0xF800.

ALU

The ALU will be used to calculate the Error term from the IR readings.

It will also be used to calculate the PI results and add/subtract that value from the **Fwd** speed register.

15x15 bit multiplier yields a 30-bit product. Use bits 27:12 as result, however, we want to saturate to a 15-bit result. So we look at bits 29:26 to determine if we should saturate to 0x3FFF if positive, or to 0xC000 if negative.

To perform the calculations you will need certain operands on certain sides of the ALU. See the table below:

SRC1[15:0]	SRC0[15:0]
Accum	{4'b0000,a2d_res}
{4'b0000,Iterm}	{{4{Intgrl[11]}},Intgrl}
{{4{Error[11]}},Error}	{{4{Icomp[11]}},Icomp}
{{8{Error[11]}},Error[11:4]}	Pcomp
{4'b0000,Fwd}	{{2b00,Pterm}

Use a 16-bit ALU although many values will be saturated to signed 12-bit value.

SRC0 value has possibility of being left shifted by 1 or 2 bits (mult2 or mult4)

SRC0 value also has a 1's complementor which when used in conjunction with a Cin provides subtraction ability.

Make your test bench count. Finding bugs in this unit in the fullchip simulation is not the way to go. It will cost you much grief late in the semester when you can least afford it. You want to find your bugs now.

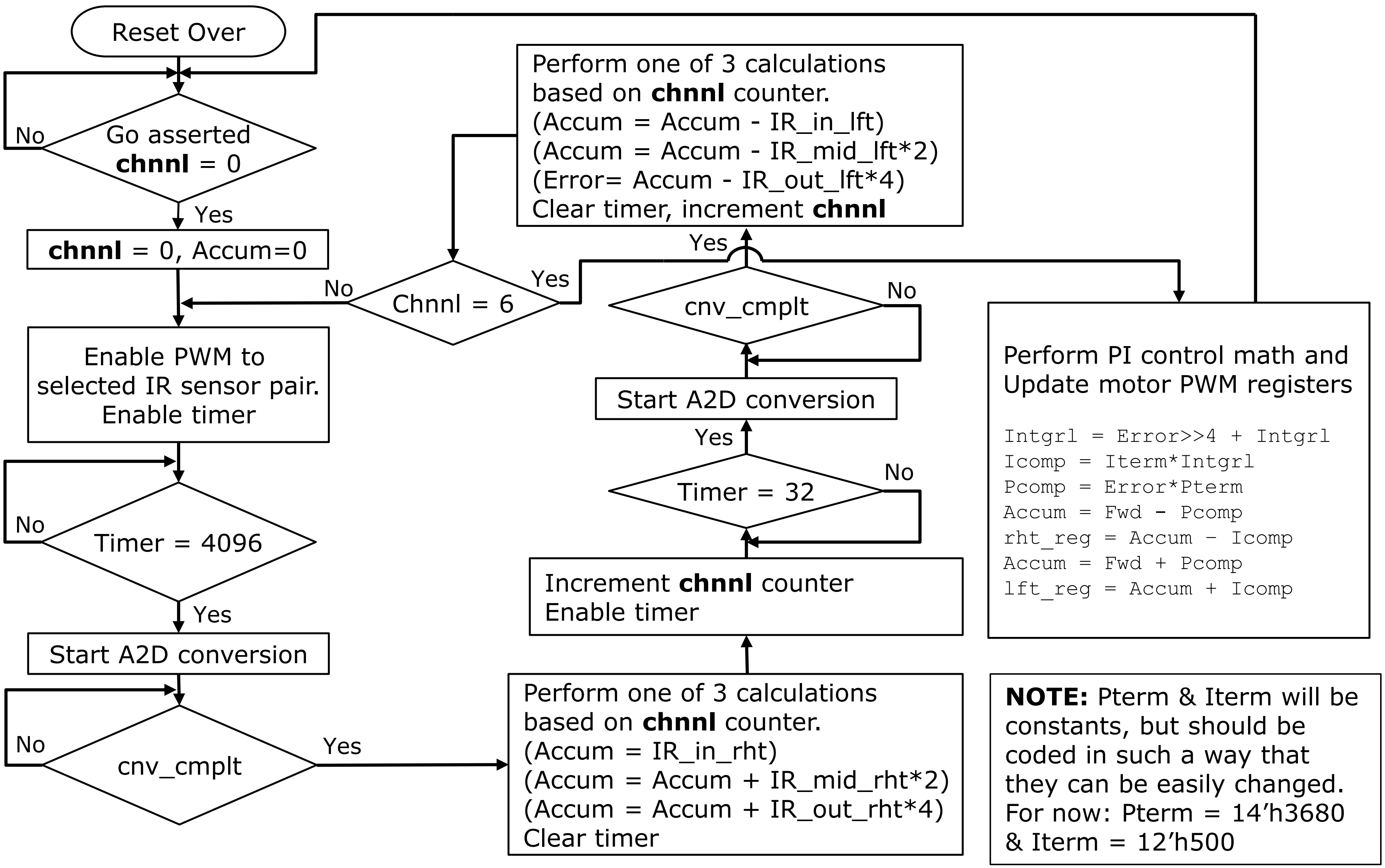
Control Math (Classic PI control)

```
1. Accum = 0
2. Accum = Accum + IR_in_rht
3. Accum = Accum – IR_in_lft
4. Accum = Accum + IR_mid_rht*2
5. Accum = Accum – IR_mid_lft*2
6. Accum = Accum + IR_out_rht*4
7. Error = saturate(Accum – IR_out_lft*4)
8. Intgrl = saturate(Error>>4 + Intgrl)
9. Icomp = Iterm*Intgrl
10. Pcomp = Error*Pterm
11. Accum = Fwd - Pcomp
12. rht_reg = saturate(Accum – Icomp)
13. Accum = Fwd + Pcomp
14. lft_reg = saturate(Accum + Icomp)
```

rht_reg => upper 11-bits of this register form the PWM duty cycle and direction that is sent to the right motor driver. **lft_reg** is the same for the left motor

Fwd => 12-bit register that holds the forward speed. This register should ramp up from 0x000 to a maximum of 0x700. It should be incremented once per PI calculation loop (not once per clock).

Digital Core Functionality (IR Sensors & PI Math)

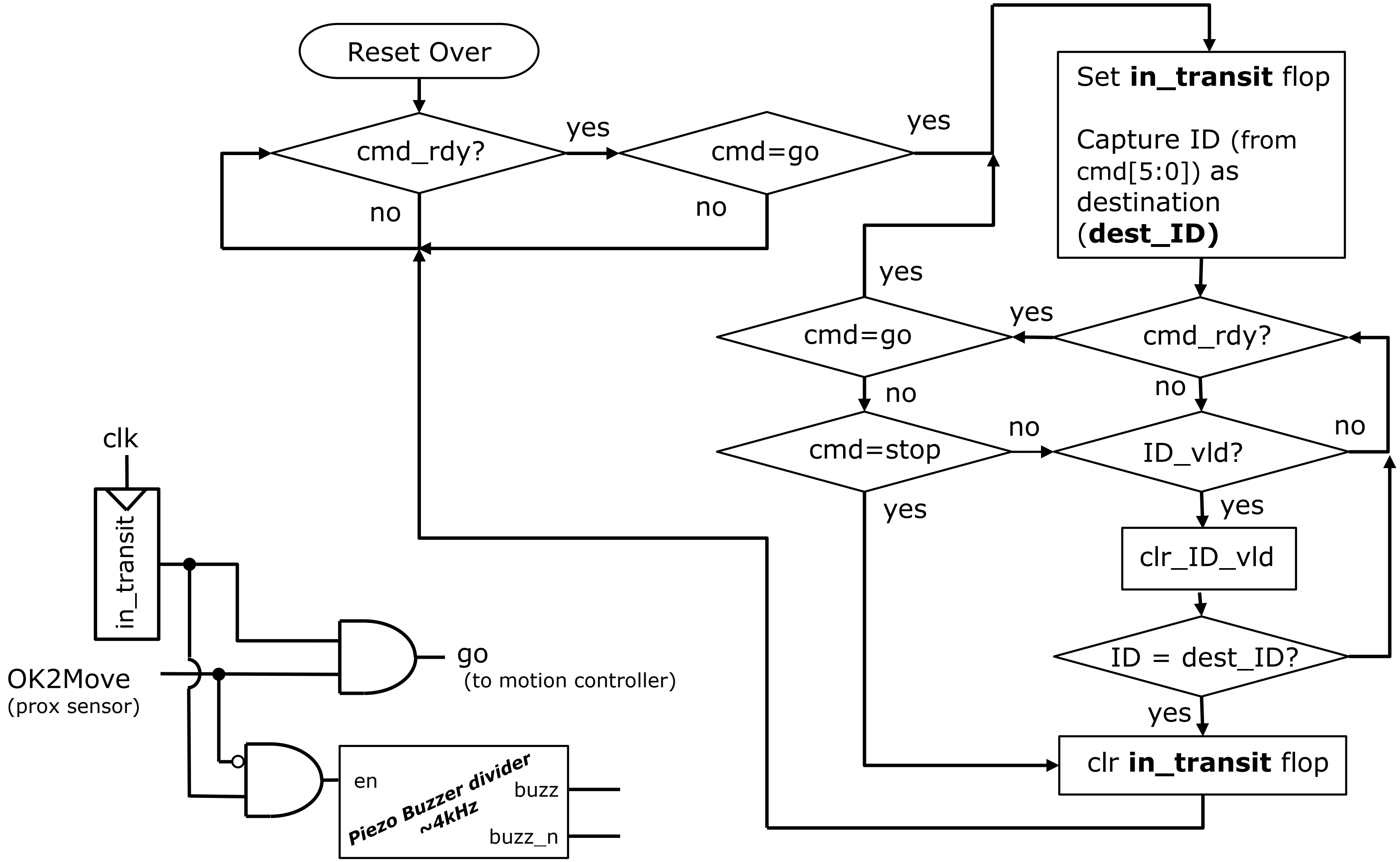


Reset Over
Go asserted chnnl = 0
No
Yes
chnnl = 0, Accum=0
Chnnl = 6
No
Yes
Enable PWM to selected IR sensor pair. Enable timer
Timer = 4096
No
Yes
Start A2D conversion
cnv_cmplt
No
Yes
Perform one of 3 calculations based on chnnl counter.
(Accum = Accum - IR_in_lft)
(Accum = Accum - IR_mid_lft*2)
(Error= Accum - IR_out_lft*4)
Clear timer, increment chnnl
Yes
cnv_cmplt
No
Start A2D conversion
Yes
Timer = 32
No
Increment chnnl counter
Enable timer
Perform one of 3 calculations based on chnnl counter.
(Accum = IR_in_rht)
(Accum = Accum + IR_mid_rht*2)
(Accum = Accum + IR_out_rht*4)
Clear timer
Perform PI control math and Update motor PWM registers

```
Intgrl = Error>>4 + Intgrl
Icomp = Iterm*Intgrl
Pcomp = Error*Pterm
Accum = Fwd - Pcomp
rht_reg = Accum – Icomp
Accum = Fwd + Pcomp
lft_reg = Accum + Icomp
```

NOTE: Pterm & Iterm will be constants, but should be coded in such a way that they can be easily changed. For now: Pterm = 14’h3680 & Iterm = 12’h500

Digital Core Functionality (command processing)



Small Details to Note:

- The value your A2D_intf.sv should provide is the 1's complement (inversion) of what was actually read from the A2D over the SPI bus.

- In order for the integral term of the PI calculations to not "run away" and saturate too quickly, integration only occurs once every 4 calculation cycles.
    - ✓ Easiest way to accomplish this is to have a 2-bit counter that is incremented every time the integration math step is performed (`Intgrl = Error>>4 + Intgrl).` Lets call this counter int_dec[1:0] for integral decimator.
    - ✓ Your signal to enable write back to the Intgrl register (something like dst2intgrl) is then only asserted when the 2-bit counter is full (i.e. dst2intgrl = &int_dec)

- The forward register (**Fwd**) is needed to make the follower move forward. The PI calculations are to create a correction term to steer the follower and keep it on the line, but we need the follower to move forward in general. That is why in the final calculations you can see the results of the PI calculations are added/subtracted from the **Fwd** register. Now we can't have the **Fwd** register get up to full positive value because we need overhead for the steering to work, so the **Fwd** register will start at zero and ramp up (incrementing by 1 each we write to the integral term) till it saturates at 7/16 of full value. To make it clearer I will just show my code for **Fwd** on the next slide.

Small Details to Note: (continued)

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
              Fwd <= 12'h000;
  else if (~go)                               // if go deasserted Fwd knocked down so
              Fwd <= 12'b000;                 // we accelerate from zero on next start.
  else if (dst2intgrl & ~&Fwd[10:8])          // 43.75% full speed
              Fwd <= Fwd + 1'b1;              // only write back 1 of 4 calc cycles
```

- When go is deasserted (either due to proximity detector triggering, or stop command, or reaching your destination). We also need to ensure we zero the right and left motor drive registers. My code for the rht_reg is shown below:

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
              rht_reg <= 12'h000;
  else if (!go)
              rht_reg <= 12'h000;
  else if (dst2rht)
              rht_reg <= dst[11:0];
```

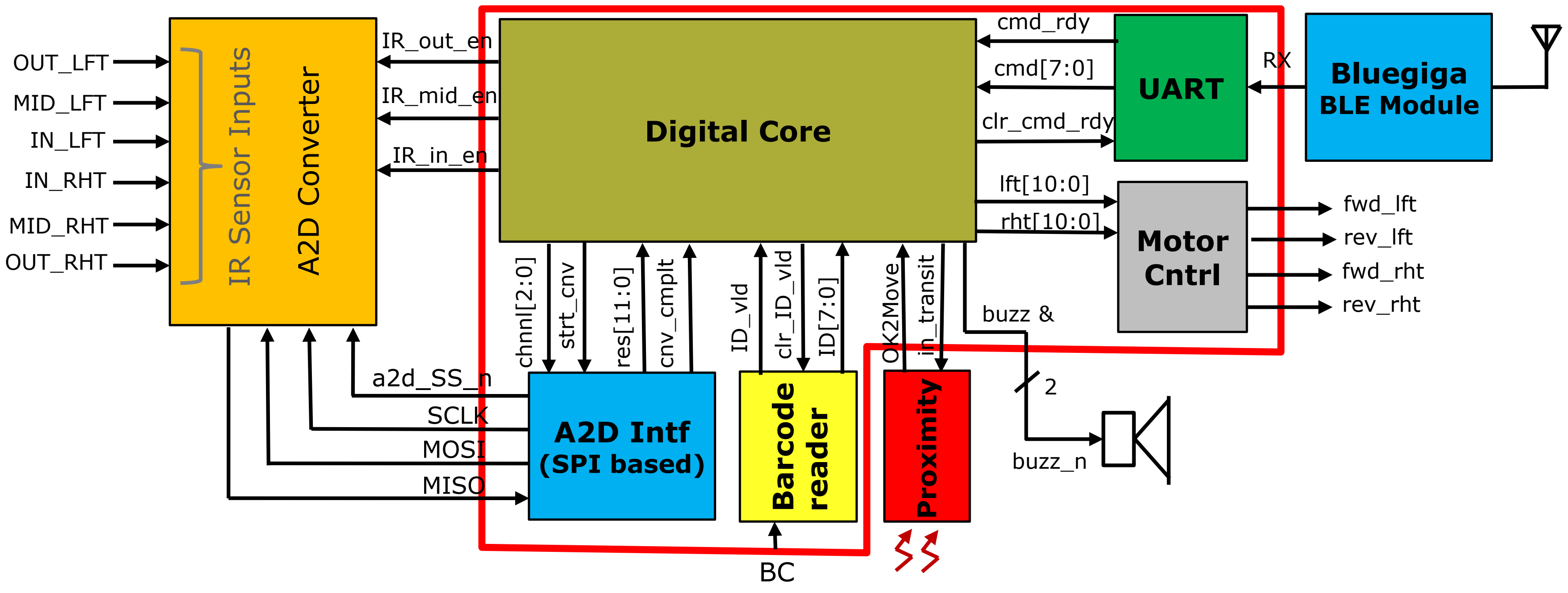
Commands

(Commands received over UART interface via Bluetooth)

Command Encoding: All commands are 8-bits. The upper 2-bits encode the command, and the lower 6-bits encode the station ID (if pertinent)

Bits[7:6]	Bits[5:0]	Description:
2'b00	6'hxx	Stop command. Follower will deassert "go" to the motion control unit and stop where it is.
2'b01	6'hID	Go command. Follower will flop the lower 6-bits as the destination station ID and will start moving along the line. When the barcode unit asserts a valid station ID (ID_vld) the ID just read will be compared to the station ID. If the ID matches the destination ID the follower stops. If not it keeps moving. A new go command can come in while the follower was still executing a prior go command. In this case the new destination ID overrides the prior one.
2'b1x	6'hxx	These two commands are reserved for future use.

Required Hierarchy & Interface



Your design will be placed in an "EricKushagra" testbench to validate its functionality. It must have a block called **follower.v** which is top level of what will be the synthesized DUT (shown outlined in red here)

The interface of follower.v **must match exactly** to our specified follower.v interface

Please download follower.v (interface skeleton) from the class webpage.

The hierarchy/partitioning of your design below **follower.v** is up to your team.

The hierarchy of your testbench above is up to your team.

follower Interface

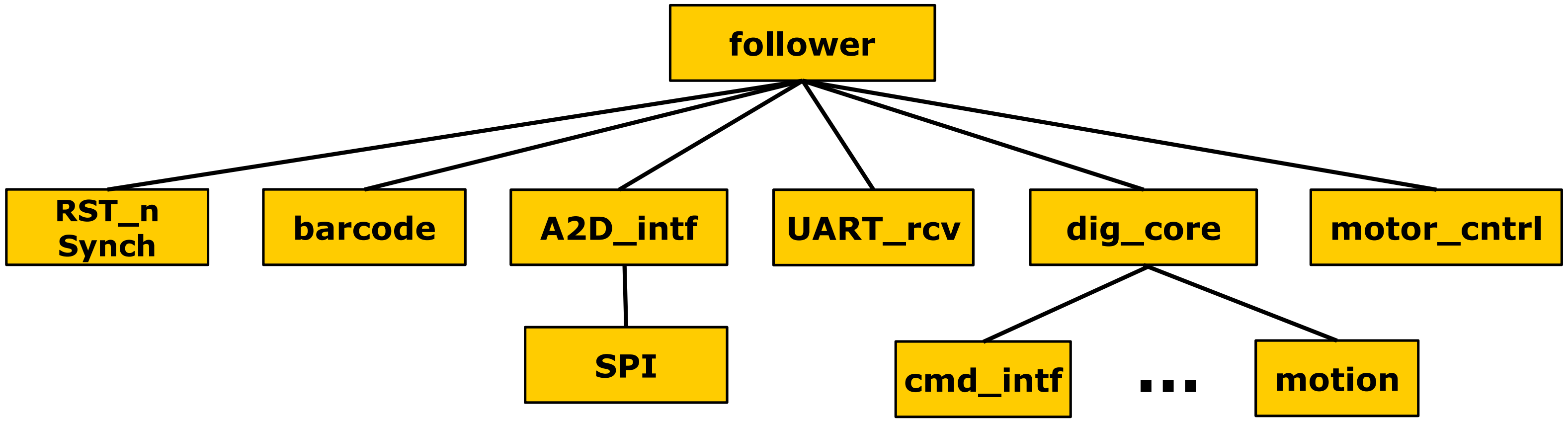
Signal Name:	Dir:	Description:
clk	in	Clock input (50MHz)
RST_n	in	Active low input from push button. Should be synchronized inside follower to produce **rst_n** which goes to all other units as the global reset.
MISO	in	Master In Slave Out from SPI bus of A2D converter
a2d_SS_n	out	Active low slave select to A2D SPI interface
SCLK	out	SPI bus clock (to A2D)
MOSI	out	MOSI to SPI bus of A2D converter
OK2Move	in	From forward looking proximity sensor. Indicates nothing in the way
in_transit	Out	Acts as enable to proximity sensor
buzz,buzz_n	out	Used to drive the piezo buzzer
RX	in	Serial (RS232) input from Bluetooth Low Energy module. Forms command.
led[7:0]	out	To the 8 active high LED outputs of the DE0 nano board. Use for what you want.
rev_rht,fwd_rht	out	PWM signals to control right motor pair
rev_lft,fwd_lft	out	PWM signals to control left motor pair
IR_in_en	out	Enables the inside pair of IR sensors
IR_mid_en	out	Enables the middle pair of IR sensors
IR_out_en	out	Enables the outside pair of IR sensors

Provided Modules & Files: (available on website under: Project)

File Name:	Description:
follower_tb.sv	**Optional** testbench template file.
follower.sv	**Requried** interface skeleton verilog file. **Copy this** and flush it out with your design
dig_core.sv	Optional digital core interface skeleton. Might be handy as starting point
ADC128s.sv	Models the A2D converter used to convert the IR sensor data
analog.dat	Text file that represents IR values. Read by ADC128s.sv model to model IR sensor readings.
barcode_mimic.sv	Model you can use when testing your **barcode.sv** unit.
check_math.pl	Perl program one can use to check the output of their PI calculations. It outputs many of the intermediate calculated values. Reads analog.dat, the P & I coefficients used are set near the top of the .pl file)

Synthesis:

- You have to be able to synthesize your design at the **follower** level of hierarchy.



- Your synthesis script should write out a gate level netlist of follower (follower.vg).
- You should be able to demonstrate at least one of your tests running on this post synthesis netlist successfully.
- Timing (400MHz) is mildly challenging. Your main objective is to minimize area.

Synthesis Constraints:

Contraint:	Value:
Clock frequency	400MHz (yes, I know the project spec speaks of 50MHz, but that is for the FPGA mapped version. The TSMC mapped version needs to hit 400MHz.
Input delay	0.5ns after clock rise for all inputs
Output delay	0.5ns prior to next clock rise for all outputs
Drive strength of inputs	Equivalent to a ND2D2BWP gate from our library
Output load	0.1pF on all outputs
Wireload model	TSMC32K_Lowk_Conservativve
Max transition time	0.15ns
Clock uncertainty	0.10ns

NOTE: Area should be taken after all hierarchy in the design has been smashed.