ECE 551
Project Specification

Spring ’17
Line Following Robot
Revision 4
4/26/2017

Grading Criteria

- Project demo: 80%
  - DUT code review: 10%
    - Comment quality, conformity to Cummings' guidelines, etc.
  - Testbench code review: 15%
    - Comment quality, coverage, etc.
  - Reference testbench results: 20% (= FunctionalityScore)
    - Synthesis script review: 5%
    - Post-synthesis test results: 13%
    - Run of the robot on the track: 15%
    - Use of a version control system: 2%
- Design area: 20%
  - $\text{YourScore} = \text{YourArea} / \text{BestArea} \times \text{FunctionalityScore}$

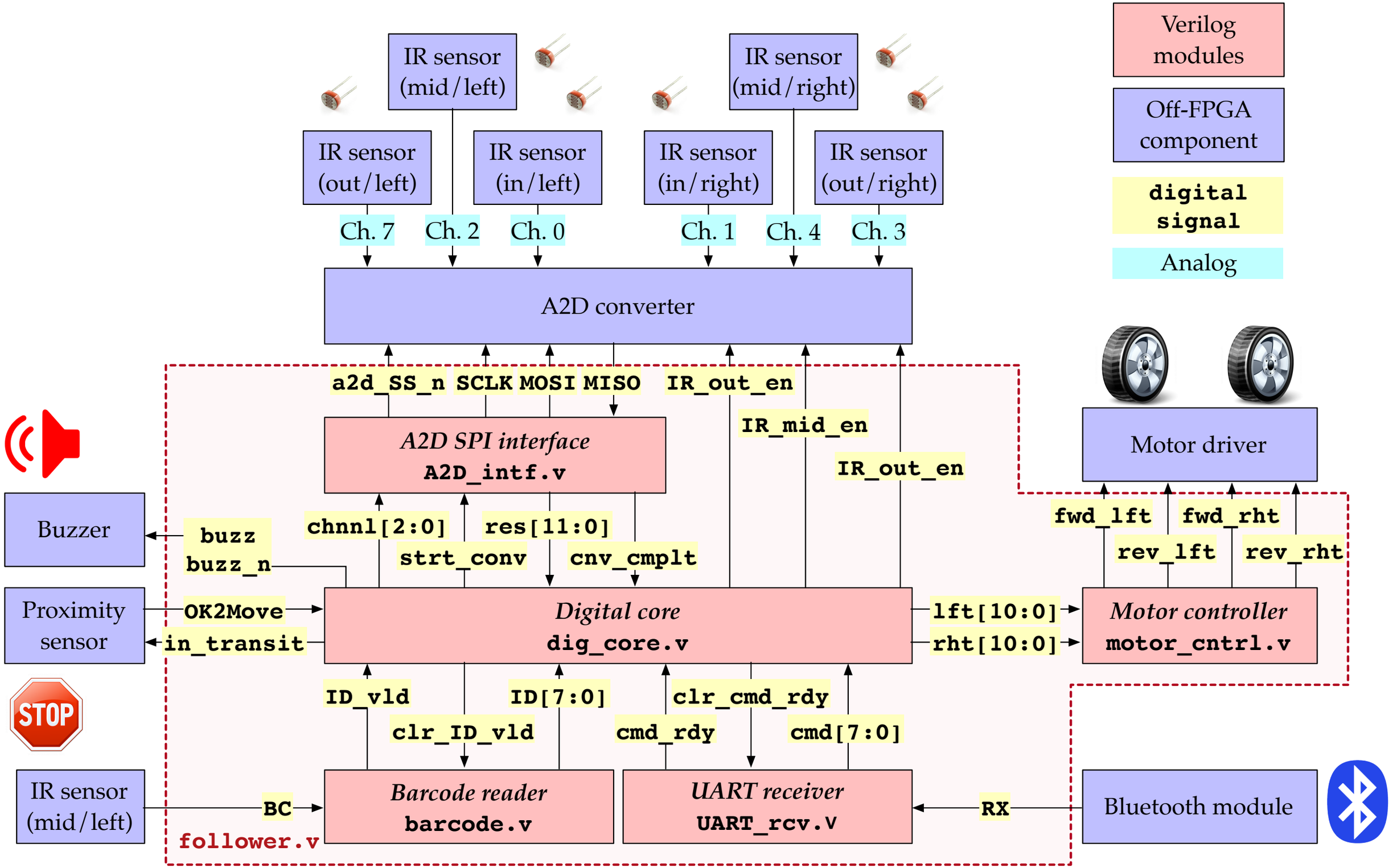
Project Demo (Tentative)

- Location: B555
- Date: 5/4 (Thu) and 5/5 (Fri)
  - 1.25% extra credit for demoing on Thursday
  - Time to be announced

Incentives for Contributions

- 1.25% extra credit for making significant contributions on the Moodle Discussion Forum
- This specification is incomplete or even incorrect
  - In practice, you are never given a perfect specification for a new design
- Make contributions
  - Make definitions more clear
  - Correct inconsistencies
    - e.g., 15-bit output connected to 16-bit input
  - Share testbenches
    - e.g., Verilog code for testbenches, input and output memory dump files
    - Share only after the testbench submission due +2 days has passed
  - Participate in discussions. Share ideas and information. Thumb up.

Top-Level Block Diagram



Digital Core (`dig_core.v`)

- *Command & control* module (`cmd_cntrl.v`)
  - Receives commands, reads barcodes and proximity sensor
  - Determines go/stop, alarms if blocked
- *Motion controller* module (`motion_cntrl.v`)
  - Reads IR sensors
  - Receives go/stop signal from ***command & control*** module
  - Determines left/right motor speed using ALU (`alu.v`)

Other Modules

- *A2D SPI interface* module **(a2d_intf.v)**
  - Reads A2D conversion results from SPI-based A2D converter
  - Sends A2D conversion results to ***digital core***
- *Barcode reader* module **(barcode.v)**
  - Reads barcodes along the line using another IR sensor
  - Sends current location to ***digital core***
- *UART receiver* module **(uart_rcv.v)**
  - Receives commands from smartphone through Bluetooth module
  - Sends received commands to ***digital core***
- *Motor controller* module **(motor_cntrl.v)**
  - Receives left/right motor speed from ***digital core***
  - Generates PWM signals to drive left/right motors

Off-Chip Components

- A2D converter
  - Performs A2D conversion of six IR sensors one by one
- Proximity sensor
  - IR sensor to detect obstacles
- Buzzer
  - Sounds if obstacle is detected
  - Driven by differential signal at 4 kHz
- Motor driver
  - H-bridge motor driver driven by forward/reverse PWM signals
- Bluetooth module
  - Receive commands from smartphone and converts them to UART

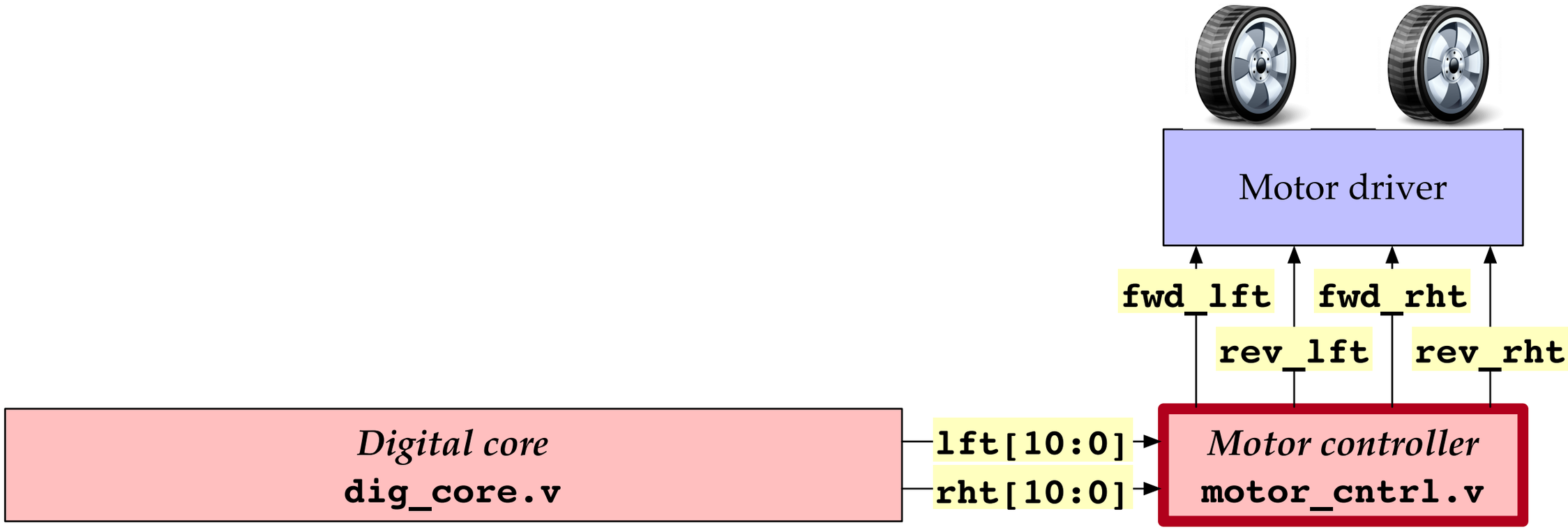
Note

- Clock **`clk`** and (active-low asynchronous) reset **`rst_n`** are not shown in the diagrams and interface tables, but they must be used by default.
- Reset all FFs (counters, shifters, output latches, etc.) must be initialized on reset.

Motor Controller

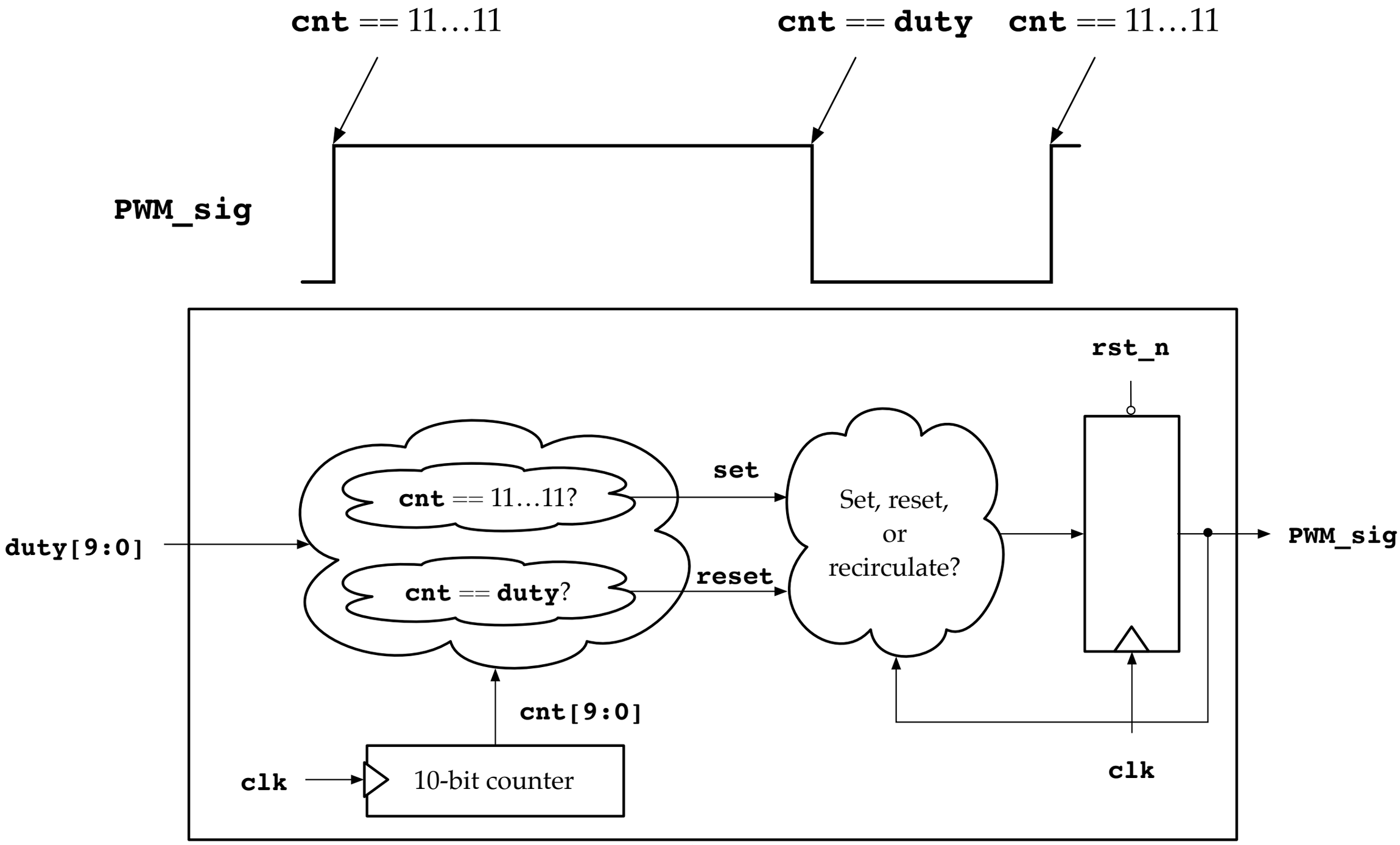
(motor_cntrl.v)

Motor Controller



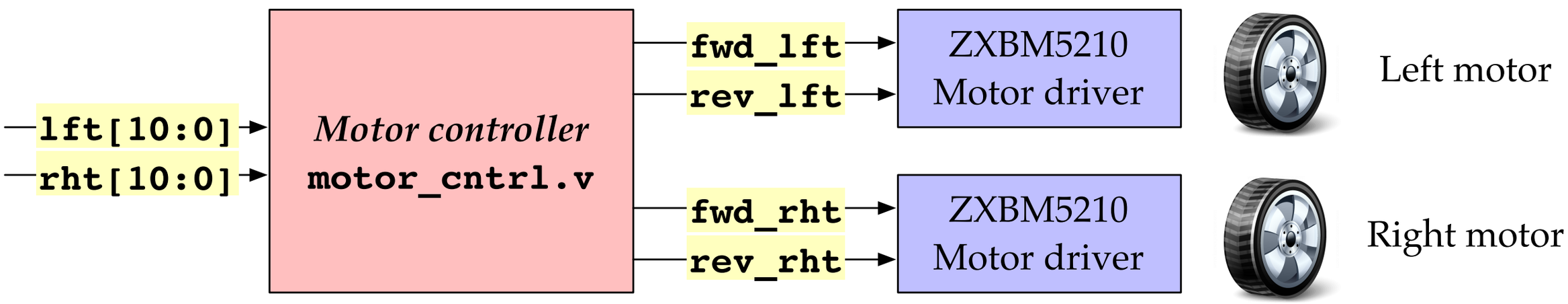
PWM Generator

- In Exercise 6, we made the PWM generator **`pwm.v`**



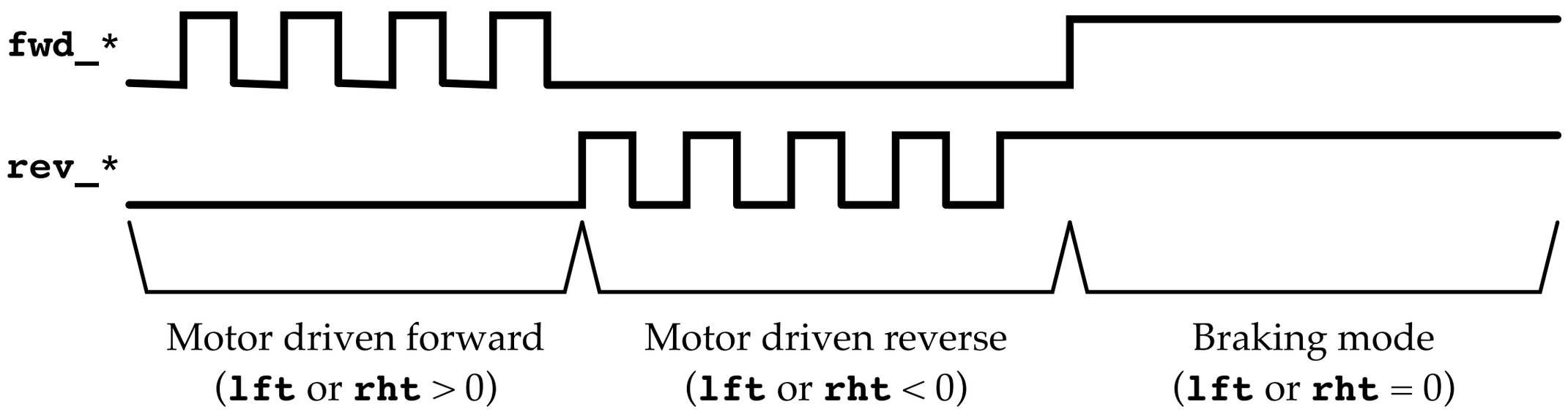
Motor Controller

- 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



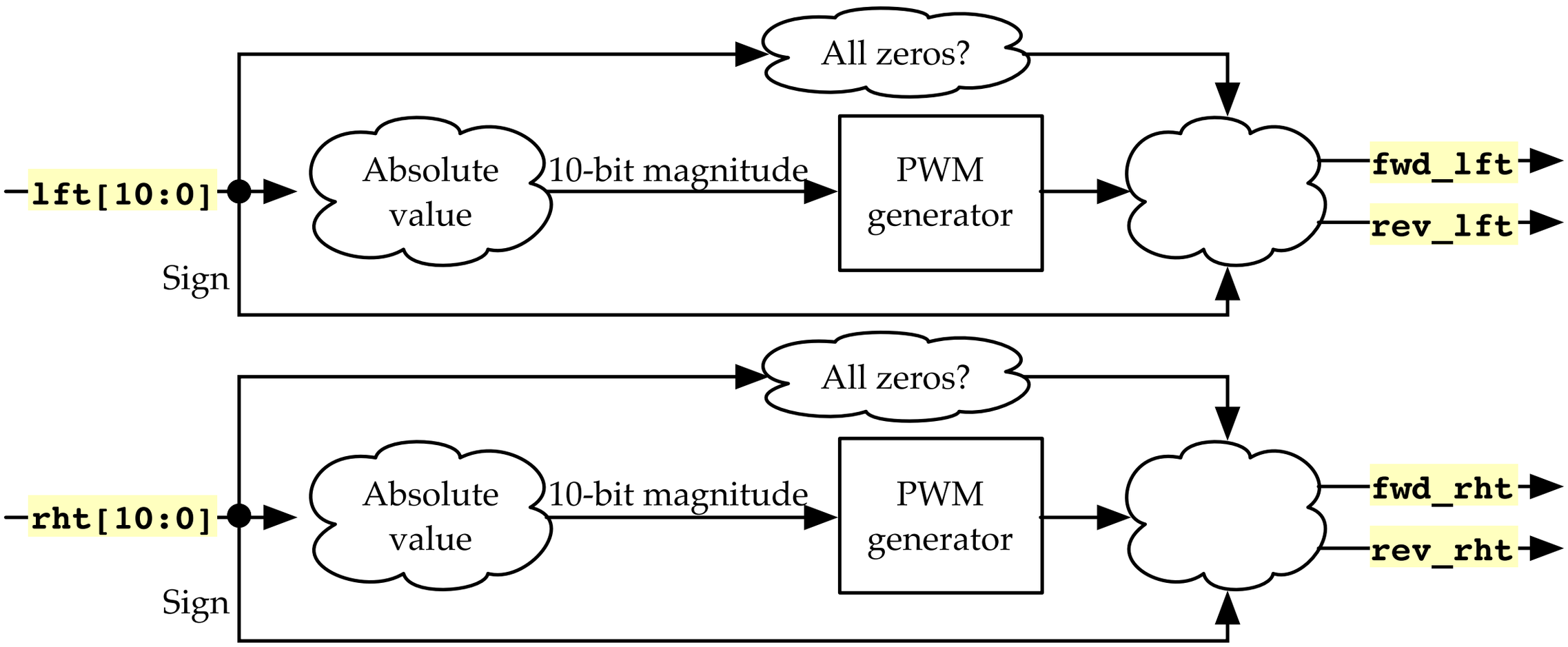
Motor Controller

- 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**.
- NOTE: The PWM module we design in Exercise 6 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.



Motor 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.



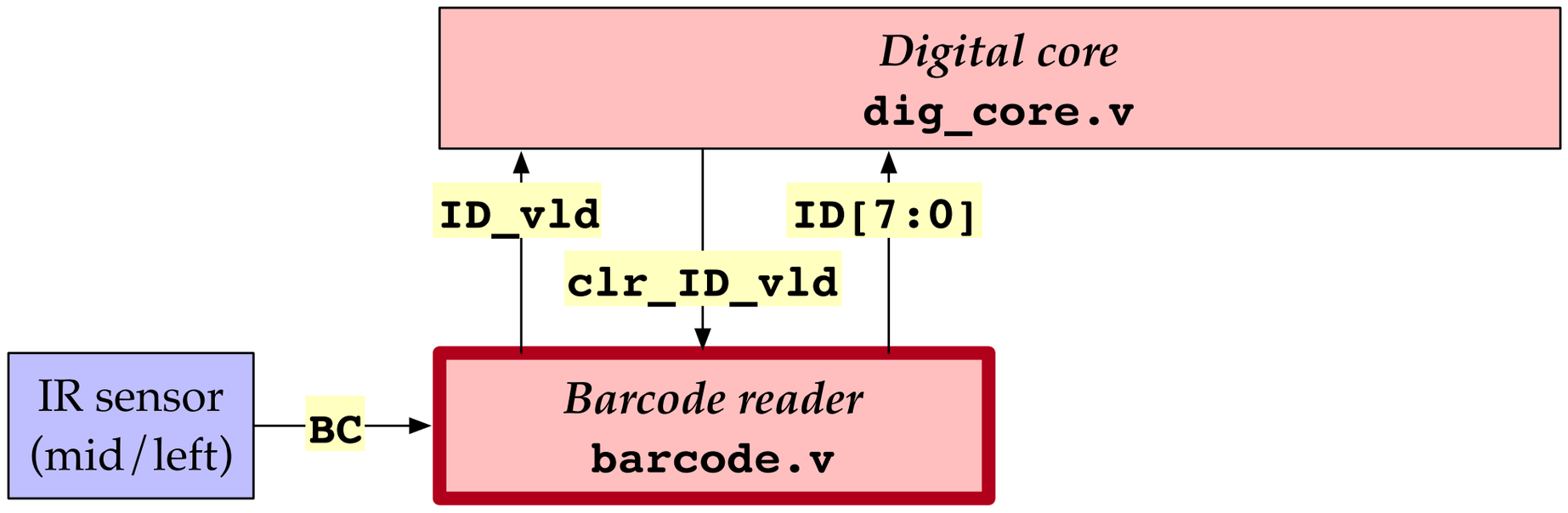
Motor Controller Interface

Signal	Dir.	Width	Description
`lft`	in	11	Signed speed of left motor
`rht`	in	11	Signed speed of right motor
`fwd_lft`	out	1	Forward PWM signal of left motor
`rev_lft`	out	1	Reverse PWM signal of left motor
`fwd_rht`	out	1	Forward PWM signal of right motor
`rev_rht`	out	1	Reverse PWM signal of right motor

Barcode Reader

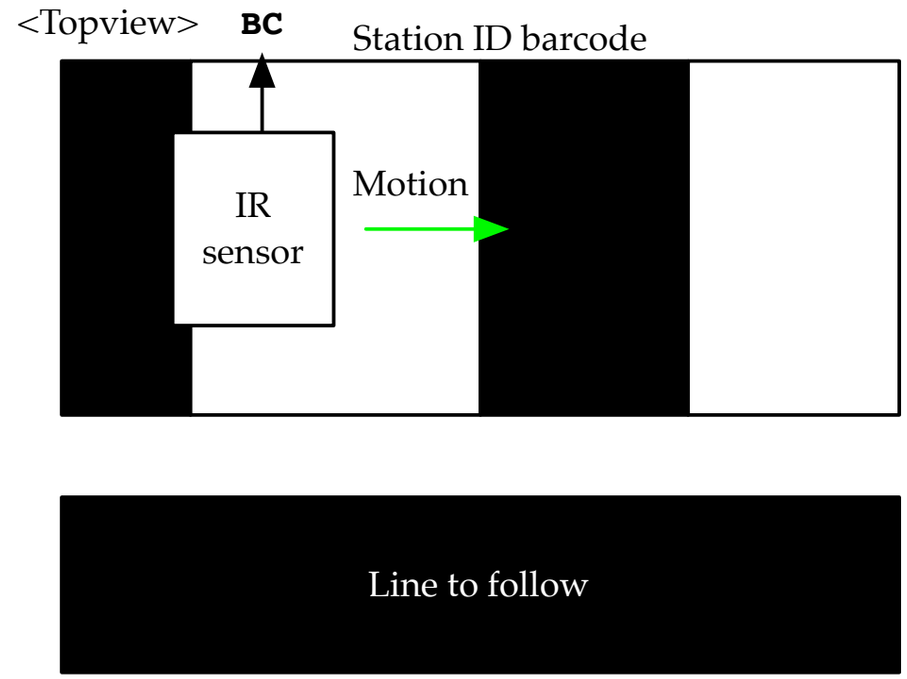
(barcode.v)

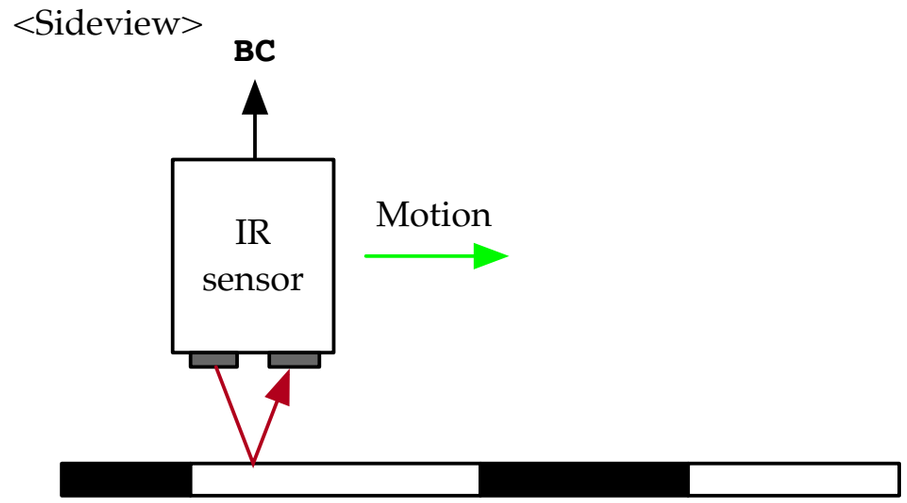
Barcode Reader



Barcode Reader

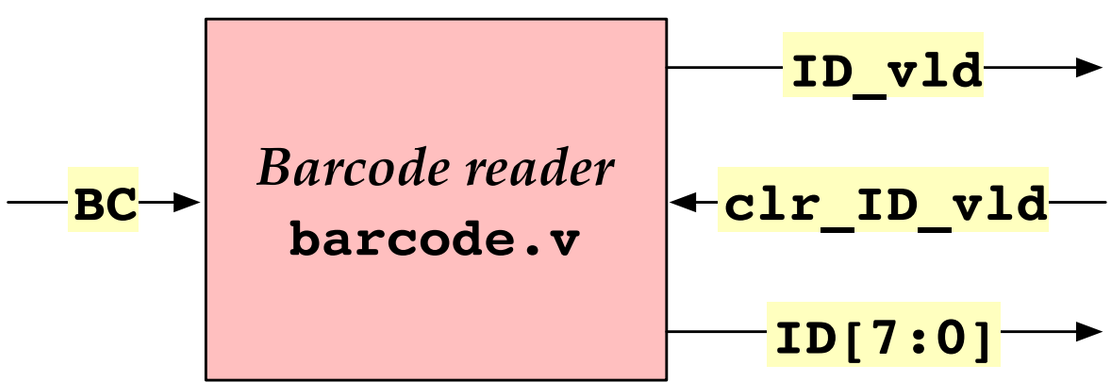
- An IR sensor is mounted on the bottom of the follower separated from the sensor array used for line following. This sensor's analog output runs into a comparator and forms a digital signal (**BC**) that will provide a barcode input.
- Station ID barcode will be marked using black lines along the line that the robot follows. When the IR sensor is above the black lines, **BC** is low, otherwise, **BC** is high.
- 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.





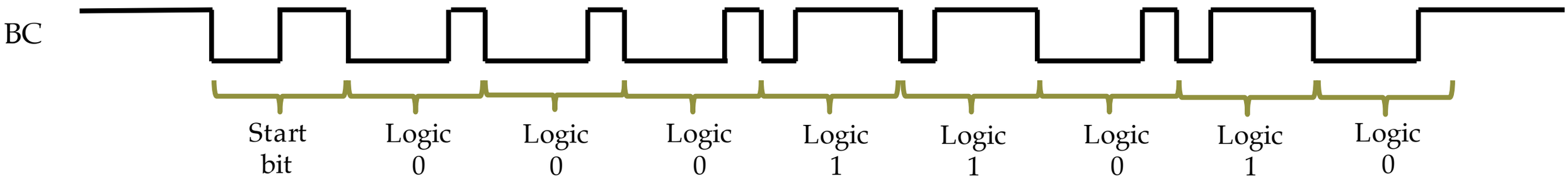
Barcode Reader Interface

Signal	Dir.	Width	Description
BC	In	1	Signal from barcode IR sensor. Serial stream (8-bits in length) that has timing information encoded (see next slide).
ID_vld	Out	1	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	Out	8	The 8-bit ID assembled by the unit, presented to the digital core.
clr_ID_vld	In	1	Asserted by the digital core to knock down **ID_vld**. Digital core would assert after having grabbed the ID from this unit.

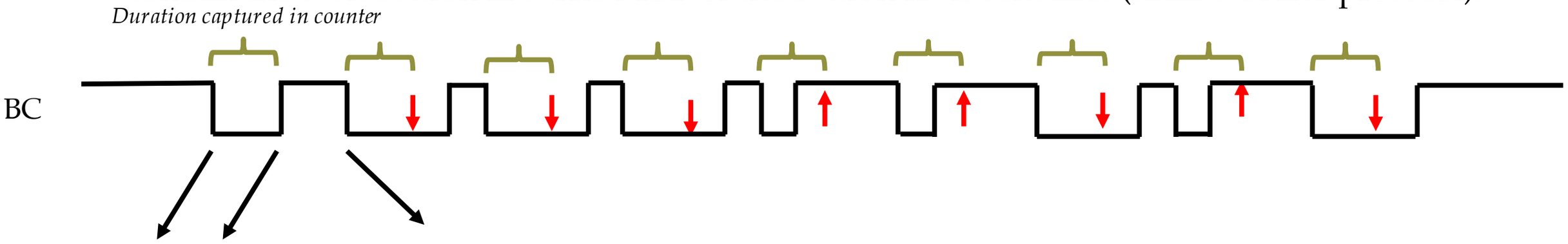


Decoding the BC Signal

Example Station ID of 0x1A



- 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).



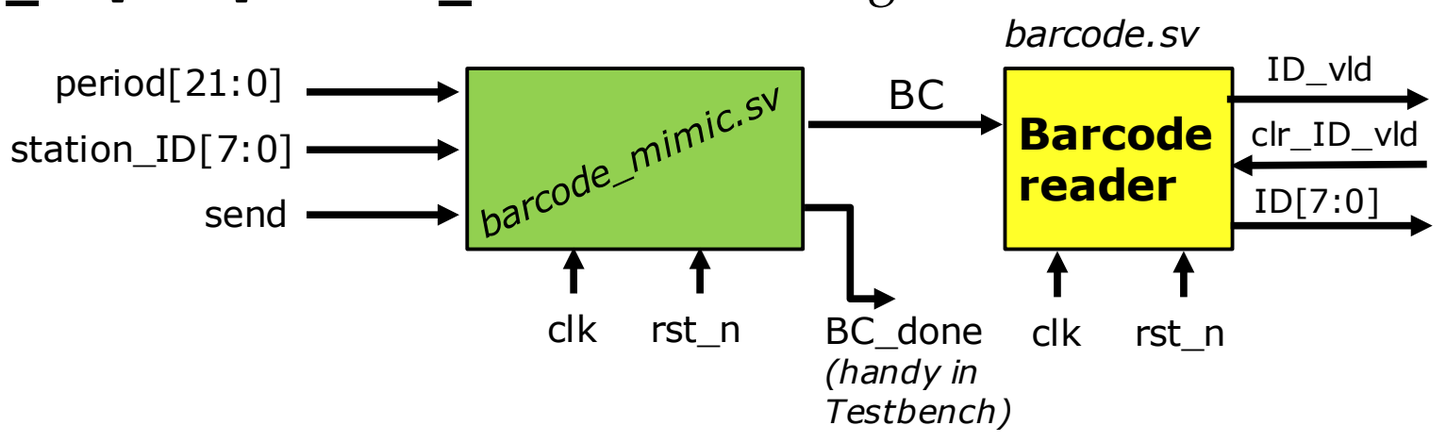
Barcode unit sees falling edge of start bit and starts a counter timing the low duration.

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.

Design and Test Hints

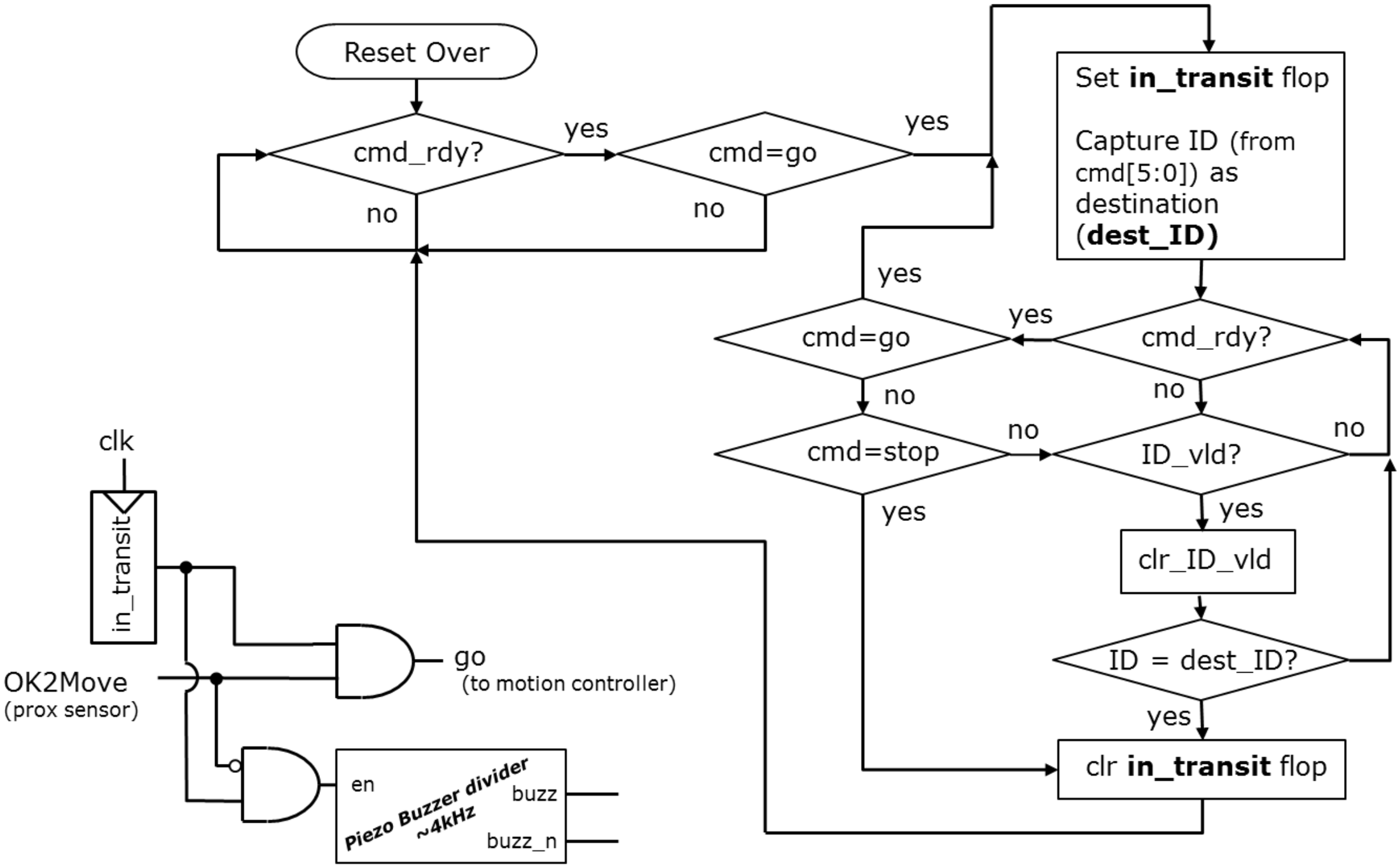
- How wide should the counters be in your barcode reader?
  - The duration of these signals depend on the speed of the follower as it passes over the station ID. We will assume a minimum speed of 0.5m/sec for the follower, and bit encodings of 2.5cm on the barcode. With the system running at 50MHz clock 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 Homework 4 folder. 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[7:0]`**, 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.



Command & Control

(cmd_contrl.v)

Command & Control Flow



Command & Control Interface

Signal	Dir.	Wid.	Connected to	Description
`cmd[7:0]`	in	8	**`UART_rcv`**	Command received
`cmd_rdy`	in	1	**`UART_rcv`**	Indicates command is ready
`clr_cmd_rdy`	out	1	**`UART_rcv`**	Clears **`cmd_rdy`**
`in_transit`	out	1	Off-chip proximity sensor	Froms enable to proximity sensor
`OK2Move`	in	1	Off-chip proximity sensor	Low if there's an obstacle and has to stop. (See diagram in previous page.)
`go`	out	1	**`motion_cntrl`**	Tells motion controller to move forward. (See diagram in previous page)
`buzz`	out	1	Off-chip buzzer	To piezo buzzer. 4KHz when obstacle detected. (See diagram in previous page)
`buzz_n`	out	1	Off-chip buzzer	Inversion of **`buzz`**. (Piezo buzzer is driven by a differential pair.)
`ID[7:0]`	in	8	**`barcode`**	Station ID
`ID_vld`	in	1	**`barcode`**	Indicates station ID is valid
`clr_ID_vld`	out	1	**`barcode`**	Clears **`ID_vld`**

Commands from UART Receiver

- 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 commands are reserved and not used.

Motion Controller

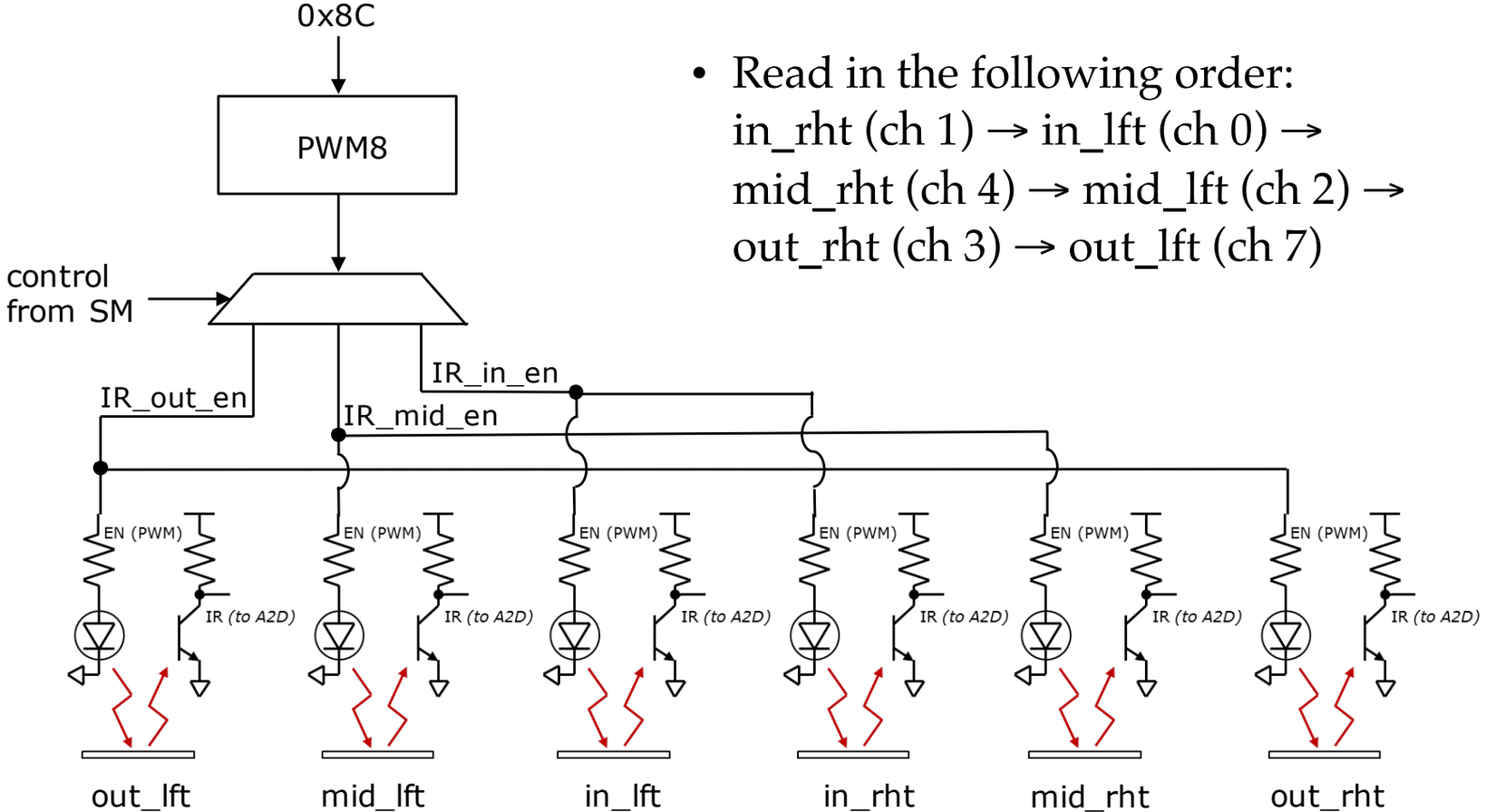
(motion_cntrl.v)

Read ALU (Exercise 4) Specification

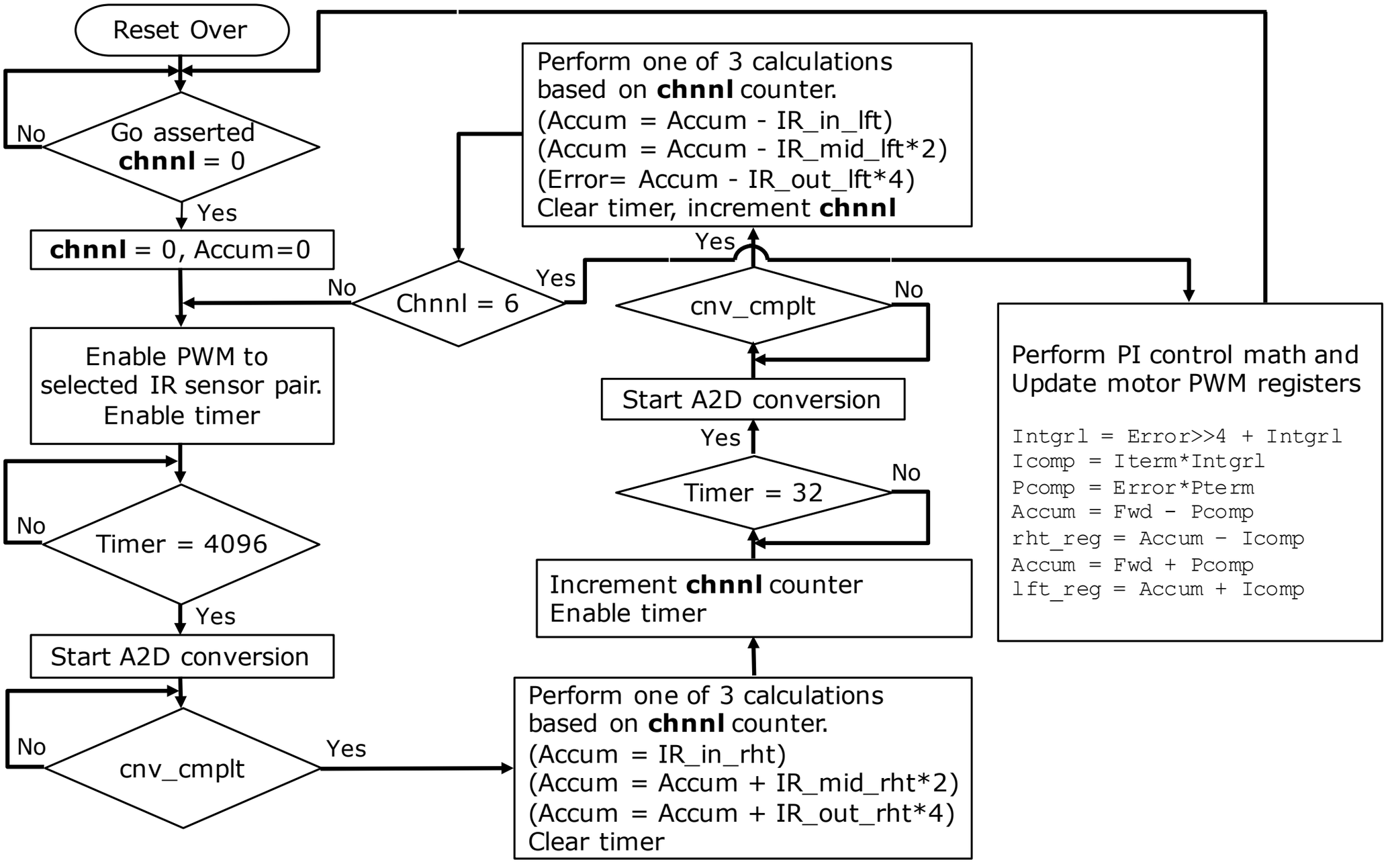
- The ALU will be instantiated in the motion controller
- Your SM needs to generate control signals to steer the sources and destination from/to a proper register
- Understand the math
- Understand the input and output ports

Reading IR Sensors

- The IR sensors are enabled in three pairs, inner, middle, and outer, in order to prevent interference.
- 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 modifying 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).
- Read in the following order:
  in_rht (ch 1) → in_lft (ch 0) →
  mid_rht (ch 4) → mid_lft (ch 2) →
  out_rht (ch 3) → out_lft (ch 7)



Motion Control Flow



- 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
- Multiplication takes two cycles. Wait for one more cycle before you proceed.

Motion Controller Interface

Signal	Dir.	Wid.	Connected to	Description
`go`	in	1	**`cmd_cntrl`**	We can move forward if high
`start_conv`	out	1	**`A2D_intf`**	Initiates a conversion
`chnnl[2:0]`	out	3	**`A2D_intf`**	A2D channel number
`cnv_cmplt`	in	1	**`A2D_intf`**	Indicates conversion is complete
`A2D_res[11:0]`	in	12	**`A2D_intf`**	Unsigned 12-bit result from A2D
`IR_in_en`	out	1	Off-chip A2D	PWM based enable to inner IR sensors
`IR_mid_en`	out	1	Off-chip A2D	PWM based enable to middle IR sensors
`IR_out_en`	out	1	Off-chip A2D	PWM based enable to outer IR sensors
`LEDs[7:0]`	out	8	Off-chip LEDs	Upper bits of **`error`**
`lft[10:0]`	out	11	**`motor_cntrl`**	Direction/speed of left motor
`rht[10:0]`	out	11	**`motor_cntrl`**	Direction/speed of right motor

Motion Controller Registers (not complete)

Register	Width	Description
`Accum`	16	Accum register used when developing Error term
`Error`	12	Error register used in PI control
`Pcomp`	16	Stores the final P component of PI control
`Intgrl`	12	Integrator used to integrate Error term in PI control
`Icomp`	12	Stores the final I component of PI control
`lft_reg`	12	12-bit result for left motor drive
`rht_reg`	12	12-bit result for right motor drive
`Fwd`	12	Determines FWD speed. Saturates at 43.75%

- Again, see and compare with the ALU specification.

Motion Controller Control Signals (not complete)

Register	Width	Description
`src1sel,src0sel`	3	ALU source selection
`sub,multiply,mult2,mult4,saturate`	1	ALU controls from SM
`dst2Accum,dst2Err,dst2Int, dst2Icmp,dst2Pcmp,dst2lft,dst2rht`	1	Controls from SM that determine where ALU's **`dst`** results go

- Again, see and compare with the ALU specification.

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.
  - White line will return a high value from A2D but should be interpreted as a low error.
  - Black background will return a low value from A2D but should be interpreted as a high error.
- 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`**). Let's call this counter **`int_dec[1:0]`** for integral decimator.
  - Your signal to enable write back to the **`Intgrl`** register (something like **`dst2Int`**) is then only asserted when the 2-bit counter is full (i.e. **`dst2Int`** = &**`int_dec`**)

Small Details to Note

- The forward register (**`Fwd`**) is the forward speed and 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`**.

```
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;
```

Small Details to Note

- 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];
```

Test

Testbench Set

- Note: For using this testbench set, you should have implemented single-cycle A2D read (https://ay16-17.moodle.wisc.edu/prod/mod/forum/discuss.php?d=14015)
- Test1: Sends GO command to station1. Checks that the in_transit signal is set.
- Test2: Sends GO command to station1. Then sends a station2 barcode, follower should not stop. Then sends a station 1 barcode. Follower should stop and braking controls should be applied.
- Test3: Sends GO command to station1. Checks that the in_transit signal is set. Then deasserts OK2Move. Checks that the buzzer signals start toggling.
- Test4: Sends Go command to station1. Uses analog.dat and checks that first calc is zero, and 2nd is full left
- Test5: Uses a 2nd version of analog.dat (analog2.dat) Checks 2nd and 4th values for non-railed PWM output
- Test6: Runs for a long time with a DC error term. Used to check if Iterm is having an effect.
- Test7: Sends GO command to station1. Then sends a go command to station3. then sends a station1 barcode, follower should not stop. Then sends station1 barcode...should stop.

Check the Math using the Perl Code

- The Perl code will do the same math that the Follower verilog should do
- You need analog.dat to run the script
- Compare the outputs (detailed_calcs.txt and summary_calcs.txt) with your simulation result

Synthesis

Synthesize Your Design

- 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 mode	TSMC32K_Lowk_Conservative
Max transition time	0.15ns
Clock uncertainty	0.10ns

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