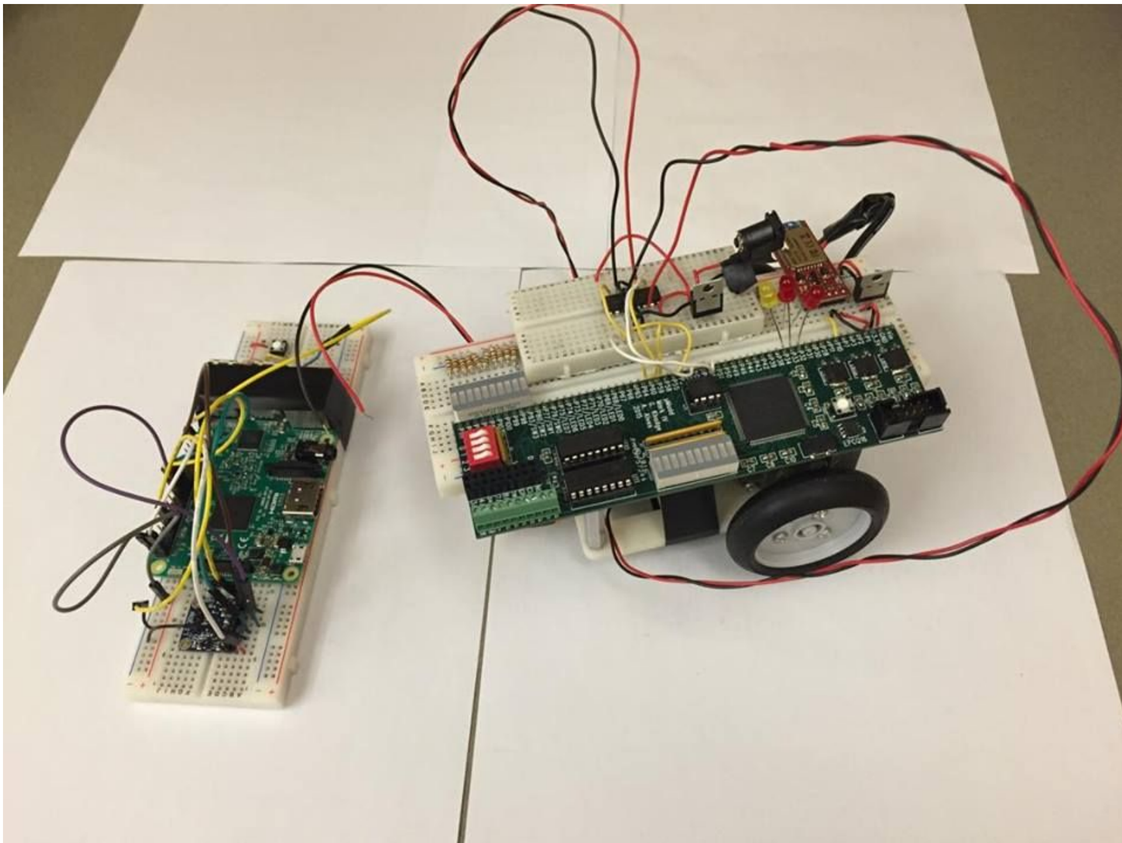
E155 - Microprocessor Systems and Design Final Project Report

Justin Lauw, Yashas Hegde

December 9, 2016

Tilt-Motion Controlled Vehicle



Abstract: The goal of this project was to build a car that varies its speed based on the pitch of a remote controller and the tightness of its turn based on the roll of the controller. The controller is designed using a Raspberry Pi 3 that takes in 3-axes acceleration readings from an LIS3DH accelerometer through SPI and converts them into pitch and roll of the controller. The Pi then calculates the left and right motor speed of the car based on pitch and roll. The left and right motor speed are each encoded into an 8-bit number that is sent over Bluetooth to a Bluesmirf module attached on the car in 2 data packets. The FPGA on the car reads in the left and right motor speed through UART and decodes them into control signals and PWM signals for the H-Bridge that controls the DC motors on the car.

Introduction

The motivation for this project was to make a fun and interesting toy that built on the concept of a Wii Remote for video games and translated it into real life. The goal of the project was to a build a tilt-motion remote controlled toy vehicle. The speed of the vehicle should correspond to the pitch (forward-backward axis) angle and the turning of the vehicle should correspond to the roll (left-right axis) angle. The vehicle should have a user controlled braking functionality and an emergency brake due to loss of communication between the remote and the vehicle. Finally, the motion of the user should not affect the motion of the vehicle.

System Partition:

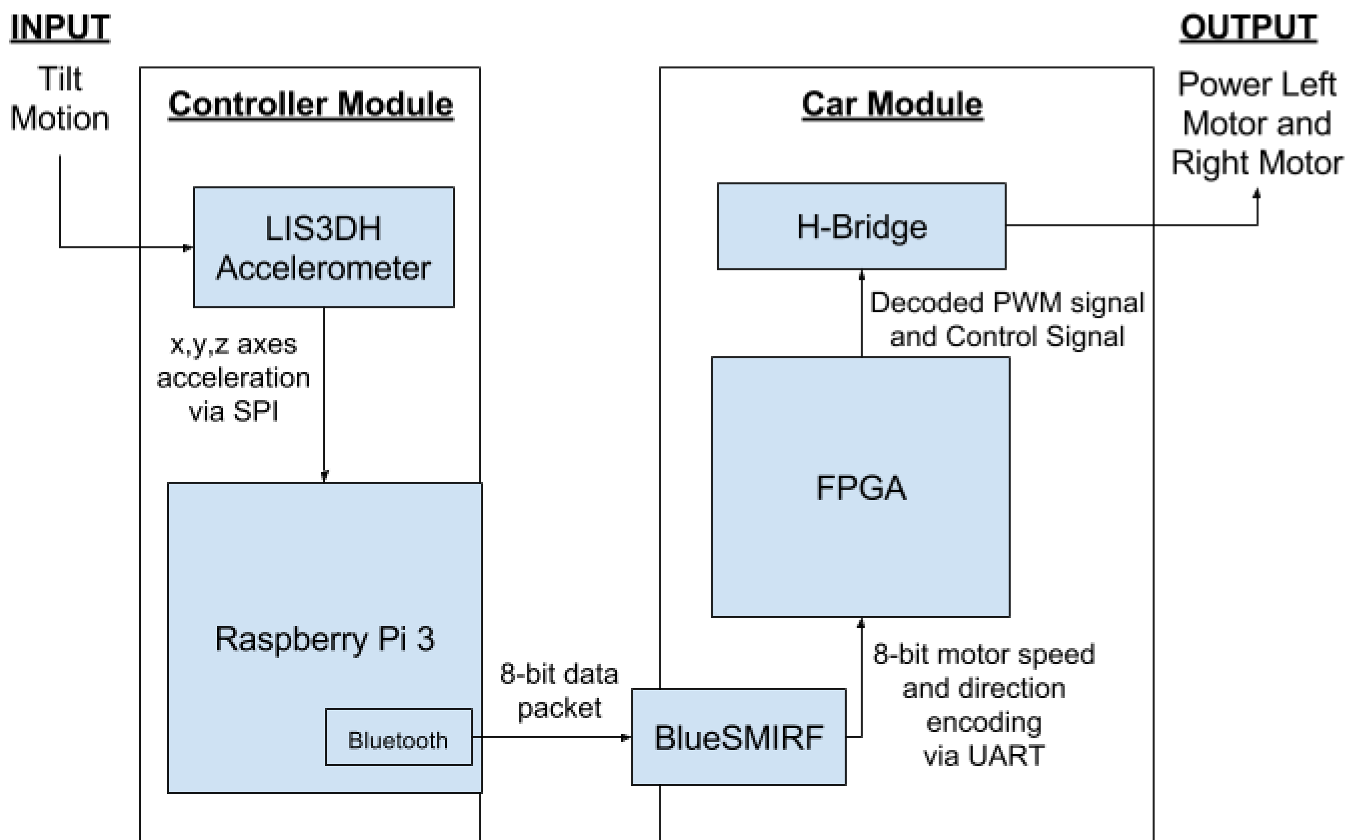


Figure 1: Block Diagram of Tilt-Motion Controlled Car System

The system consists of two main modules - the controller and the car. The controller is a breadboard used as the remote upon which a LIS3DH digital accelerometer and Raspberry Pi 3 were placed in order to measure its roll and pitch angle. The car consists of a gearbox mounted on a chassis with two motors connected to wheels in the back and a ball bearing in the front to provide stability. A breadboard with the MuddPi Mark IV board having an FPGA on it is mounted on top of the vehicle.

The Raspberry Pi 3 was chosen to be used on the controller since it was easy to use it as an SPI master to communicate with the accelerometer due to the provided functions in the EasyPIO.h header file [1]. Once the Pi collects the acceleration readings of all 3 axes from the accelerometer, it calculates the roll and pitch angle of the remote. It then uses these angles to calculate the left and right motor speed and direction of rotation and transmits them over Bluetooth to a Bluesmirf modem on the vehicle that acts as a serial pipeline.

The Field-programmable Gate Array (FPGA) on the vehicle reads the motor speed outputs of the Bluesmirf which come in the form of the Universal Asynchronous Receiver Transmitter (UART) communication protocol. It then stores these motor speeds and uses them to generate the appropriate Pulse Width Modulation (PWM) signal to control speed and H-bridge control signals to control motor direction. Finally, these signals are sent to an H-bridge that drives the two motors.

New Hardware

This project made use of the LIS3DH digital accelerometer manufactured by STMicroelectronics. The LIS3DH functions as an SPI slave and its x, y and z axis acceleration readings can be read from its registers using SPI communication, and its settings can be adjusted by using SPI to write to its registers. The timing diagram [2] below shows how this SPI communication takes place:

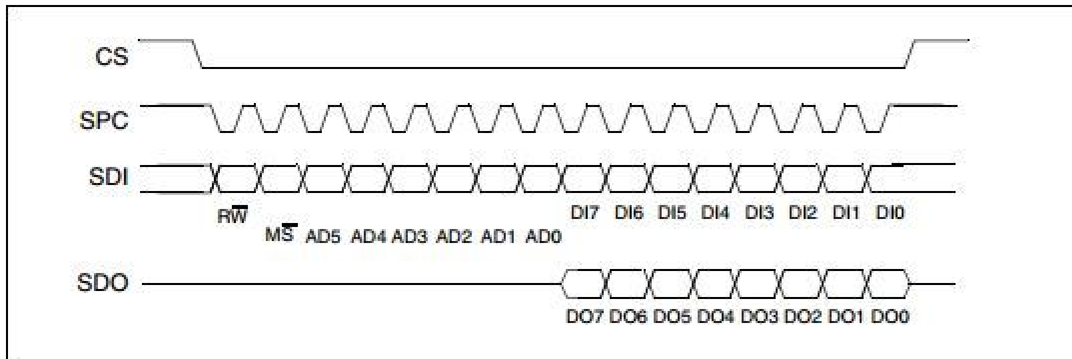


Figure 2: SPI timing diagram for LIS3DH [2]

The CS signal is the chip select signal that lets the LIS3DH know when a transmission is starting and ending. SPC is the SPI clock, and data should be clocked in from the master on the falling edge. SDI is the input from the master and SDO is the LIS3DH output. The RW bit should be 0 for a read and 1 for a write. If the function being executed is a read, the bits AD5-AD0 specify the address of the register which needs to be read from, and the D17-D10 bits do not matter. If the function is a write, AD5-AD0 is the address of the register being written to and D17-D10 are the data bits to be written to the register.

To use the accelerometer in the context of an application where the aim is to constantly stream acceleration data, it is important to consider the data refresh rate of the LIS3DH (the rate at which it updates the data in its acceleration readings registers) and the rate at which the master is reading from it using SPI. It is ideal to avoid a situation where a read occurs when the accelerometer either hasn't updated from the last read or is still updating, which would cause an erroneous read. In order to make things simply, the CTRL_REG4 [2] can be written to to set the LIS3DH to block update mode, where it only updates values in its registers after a read has occurred.

There are two registers for each of the axes x, y and z containing acceleration data (OUT_XL, OUT_XH [2] and corresponding Y, Z registers). One contains the lower 8 bits of data and the other the upper 8 bits. Combining these gives us the final 16 bits of acceleration data. However, the OUT_L register's 4 lower bits are always 0, so the acceleration data actually has a 12 bit resolution encoded in 2's complement. The next section discusses interfacing the LIS3DH with a microcontroller SPI master.

Schematics

The schematic of the Controller System is below in Figure 3. The LIS3DH needs to be powered by a 3.3 V supply. Its SDA pin should be connected to MOSI, SDO to MISO and SCL to the SPI master clock pin. Finally, CS needs to be connected to the Chip Select output pin on the master. There is a push button switch that connects a GPIO pin to the 3.3V rail if pushed for the brake functionality. It is interfaced with a pulldown resistor to GND rail.

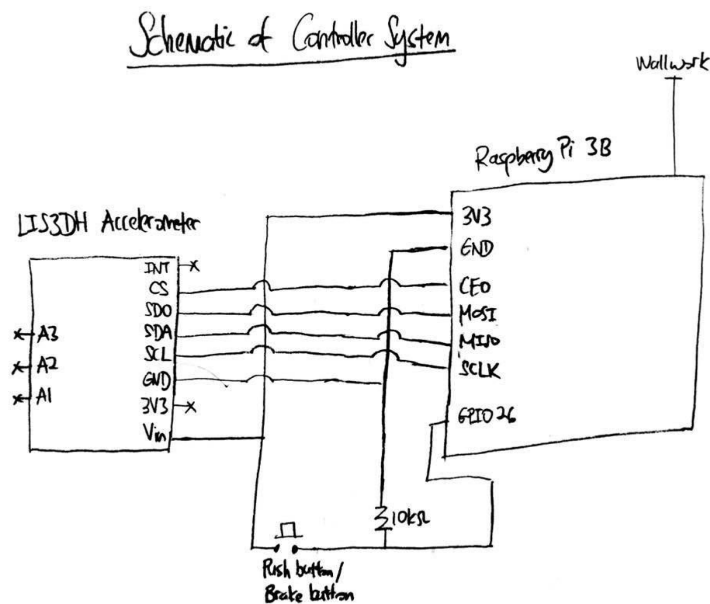


Figure 3: Remote Controller Schematic

The schematic of the Car System is below in Figure 4. Two different power supplies are used. The 9 V Battery is used to power the FPGA and LED arrays that indicate the sampling state and 8-bit Motor encodings that give the visual indication of the car speed and direction. The BlueSMIRF Module receives 3.3 V from the MuddPi. A 7.2 V NiMh Battery with an energy of 750 mAh is used to power the motors. The 7.2 V from the battery is stepped down to 5 V using a LM7805 Voltage regulator for logic voltage supply of the H-Bridge. The H-Bridge is connected to the respective pins on the MuddPi that outputs the control signals and PWM signal. The 1Y and 2Y H-bridge output pins are connected to the left motor, and 3Y and 4Y pins connected to right motor. The table of FPGA pin assignments is in Appendix H and Appendix I.

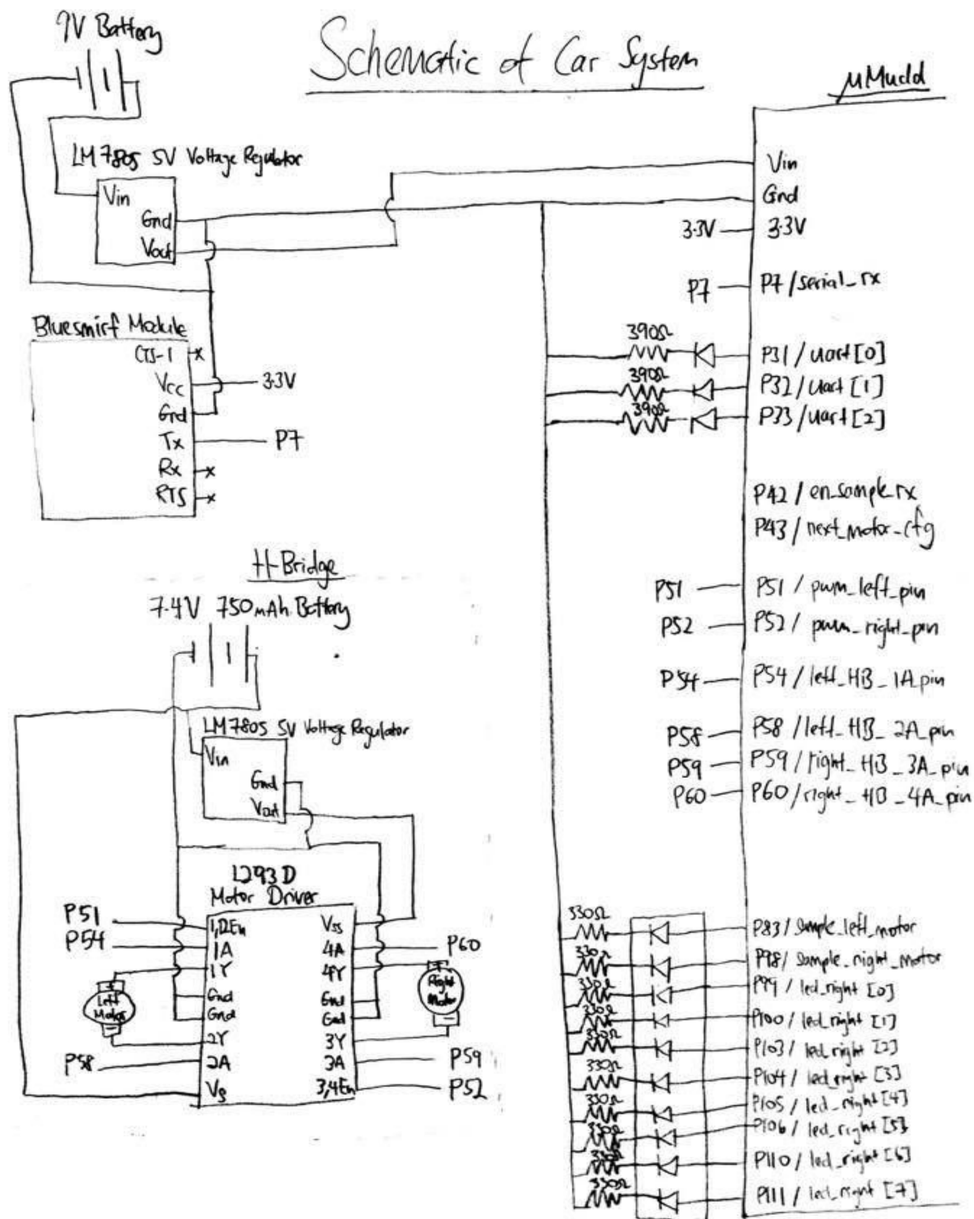


Figure 4: Car System Schematic

Microcontroller (Raspberry Pi) Design

The Raspberry Pi performs three main functions: 1) Read acceleration values and calculate pitch and roll from them, 2) Generate correctly encoded motor speeds based on pitch and roll, and 3) Send motor speeds over Bluetooth to the Bluesmirf.

1) Functions were written on the Pi to write to and read from the accelerometer using SPI. SPI was initialized using the SPIinit function from EasyPIO.h [1] with CS set to 0 and a clock frequency of 10 MHz. The read function was written to take in the address to be read from and it uses two instantiations of the SPISendReceive function in EasyPIO.h [1] in order to send the required 16 bits to the LIS3DH. The write function takes in an address to write to and the data to write and sends the appropriate bits in two bursts using SPISendRecevie twice. The specifics of the SPI format (what bits to send in order to read and write) are discussed in the section 'New Hardware'.

Once the Pi receives data back from the accelerometer it concatenates the two 8 bit data busses of the upper and lower registers for each axis into one 16 bit number. This number is then divided by 4 since the last 4 bits are 0 and not needed. It then uses the formulas shown below to calculate pitch and roll [3]:

$$\theta_x = \arctan\left(\frac{A_x}{\sqrt{A_y + A_z}}\right)$$
$$\theta_y = \arctan\left(\frac{A_y}{\sqrt{A_x + A_z}}\right)$$

Figure 5: Roll and Pitch Formulas [3]

2) The main code contains a function anglesToSpeeds that generates a pointer to a two char array which contains left and right motor speeds, calculated in the function and encoded such that the first bit indicates sign (0 for forward motor direction, 1 for backward) and the remaining 7 bits represent a decimal value between 0 and 127. The basic format for calculating the motor speeds is to first set both speeds proportional to the pitch, and then subtract a value proportional to the roll from the left motor to turn right, or subtract from the right motor to turn left based on whether roll is negative or positive. A pitch angle or roll angle of less than 10 degrees is considered to be 0 and 60 is considered to be the max angle (anything above 60 is treated as if it is 60). The sign bit is set to 0 if pitch is negative (indicating forward movement) and set to 1 if pitch is forward.

3) The Pi sends the two motor speeds over Bluetooth every 5000 runs of an infinite while loop in which acceleration readings are read and pitch, roll and motor speeds are calculated. Once the Pi is connected to the Bluesmirf, this consists simply of opening the

file dev/rfcomm0, writing the left motor speed char to it, closing the file and repeating the process for the right motor speed char. Finally, there is a case statement for when the brake button is pressed to send motor speeds of 0.

FPGA Design

The FPGA serves as a module that decodes the 8-bit data packets encoded by the Raspberry Pi into corresponding PWM signal of 1 kHz and Control Signals for the H-Bridge that controls the motors. The data is received at 115,200 baud rate from the BlueSMIRF.

The FPGA consists of 4 main modules: 1) UART Module, 2) Multi-packet Management Module, 3) Motor Module, 4) Check Bluetooth Connection Module. The interactions of the major modules are described in Figure 6 block diagram below. Asynchronous data coming in from BlueSMIRF is passed through the synchronizer first before going to UART Module.

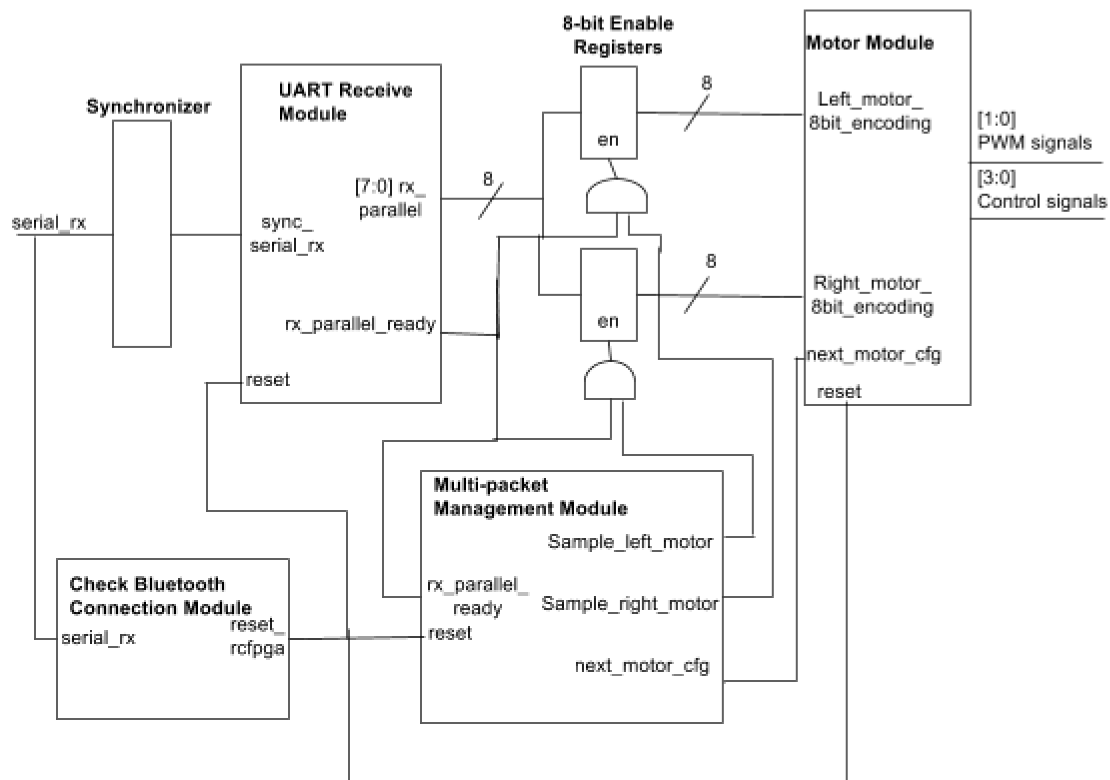


Figure 6: Simplified Block Diagram of FPGA Modules

UART Receive Module

The UART receive module uses the oversampling scheme to sample at the middle of the received bits that comes in serially and uses a serial in parallel out (SIPO) shift registers to generate an 8-bit motor encoding that is passed to the motor module to decode. The UART Module behaves according to the State Machine in Figure 7. The Block Diagram of the UART Receive Module is in Appendix D.

A slow-counter is used to track the sampling counts and number of bits that have been sampled. The slow-counter module involves two counters - a main counter that counts up at every clock cycle and another counter that runs on the main clock but only counts up when the main counter has reached an assigned divisor number. The slow-counters used to track sampling counts and number of bits are sample_counter and bits_counter respectively in Appendix D. The divisor for sample_counter is $40MHz / (115200\ baud\ rate \times 16\ samples) = 21.7 \approx 22$ The divisor for the sample_counter is $40MHz / 115200\ baud\ rate = 347.2 \approx 347$.

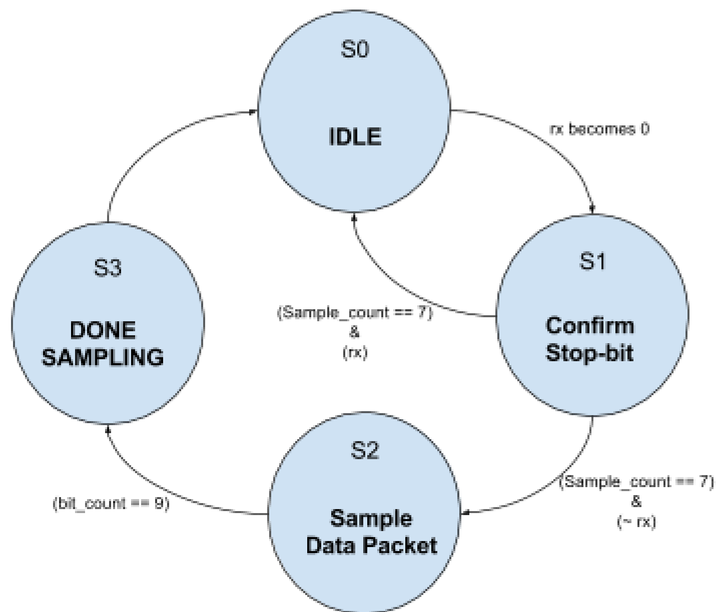


Figure 7: UART Receive Module State Machine

When a data packet comes in, the UART Module remains at S0 and waits until the incoming signal (rx) becomes 0, the beginning of the start bit, to move into S1, and then

starts the sample_counter and bit_counter. When the sample_count reaches 7, rx will be at the middle of the start bit. At this point, if the incoming rx is still 0, this confirms the start bit. The sample_counter will restart itself and the State Machine moves into S2 where the UART Module will begin sampling. At S2, whenever the sample_count reaches 15, the State Machine will output signal en_sample which enables the SIPO register to shift in the incoming signal. The State Machine will transition from S2 to S3 when bit_count reaches 9. This indicates that 8-bits of information has come into the UART Module. The state machine remains in S3 for 1 clock cycle. At S3, the state machine outputs a signal rx_parallel_ready that will enable the 8-bit register placed after the UART Receive Module to shift out the 8-bit motor encoding that has been sampled.

Multi-Packet Management Module

Each data transfer from the Pi to the BlueSMIRF is done in 2 8-bit data packets. The first packet is encoding for the left motor, second packet is encoding for the right motor. Thus a Multi-Packet Management Module is designed to manage this 2 packet transfer. The module is made up of a state machine described in Figure 8.

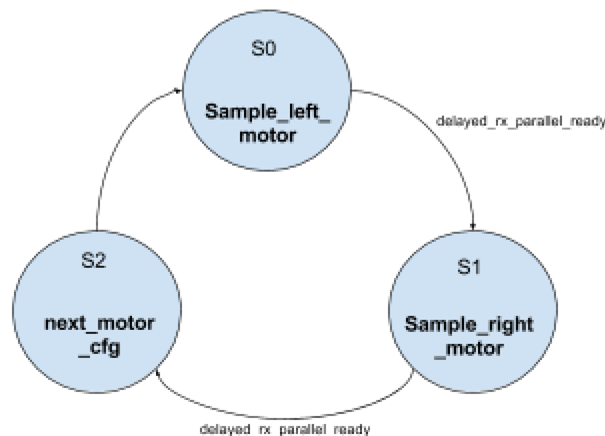


Figure 8: Multi-Packet Management State Machine

The Module outputs sample_left_motor and sample_right_motor to indicate that the left motor encoding and right motor encoding respectively are being sampled. The state machine transitions from S0 to S1 when rx_parallel_ready delayed by 1 clk cycle is triggered. This design decision is made because rx_parallel_ready and sample_left_motor

are required to enable the register to shift the newly sampled 8-bit encoding to the motor module as shown in Figure 6. At S1, the state machine samples the packet for the right motor and moves into S2 when delayed_rx_parallel_ready is triggered high. The module stays in S2 for 1 clock cycle where it generates next_motor_cfg signal that goes to the motor module to indicate that the new left and right motor encodings are ready for decoding.

Motor Module

The Motor Module decodes the most-significant bit (MSB) of the encoding into the control signal. The MSB of the left motor encoding is decoded to control signal 1A and 2A for the H-Bridge. The MSB of the right motor encoding is decoded to control signal 3A and 4A for the H-Bridge. The Motor Module Block Diagram is in Appendix F.

The least significant (LSB) 7 bits are used to determine the duty cycle of the PWM signal for respective motors. The PWM signal is generated based on a 7-bit slow_counter that takes 1 / 1kHz = 1 ms to count up to 2^7-1 = 127 and overflows after 127 counts. The design of the slow_counter is similar to that in UART module. The PWM signal will be high whenever the slow_count is less than the PWM duty cycle encoded. The PWM generator Module Block Diagram is in Appendix G.

Check Bluetooth Connection Module

This module is a safety feature that will reset the entire car system when bluetooth connection is lost. This module leverages on the fact that a data transfer occurs multiple times within a second. Hence, if the Tx pin on the BlueSMIRF idles high for more than 1 second, this indicates that Bluetooth connection is lost. A counter that counts up whenever the incoming signal is high is used. If the counter counts up to more than 40,000,000 which is the number of counts in 1 second for a clock speed of 40 MHz, the module will output rcfpga_reset that resets the entire system because Bluetooth connection is lost.

Results

The final product functions according to specifications - its forward and backward speeds are proportional to the tilt of the remote in the forward and backward directions, and its

turning radius corresponds to the left-right tilt of the remote. A push button switch on the remote brakes the car, and it stops upon loss of Bluetooth connection. However, if a user holds the remote level and walks around at a fast speed, it can cause the vehicle to move slightly forward or backward due to miscalculation of pitch resulting from acceleration in the x axis direction. Possible ways to solve this could be to implement a low-pass filter to filter out high frequency movements, or to possibly check the difference between successive x-axis readings and take action when it is too high (indicating that the user is lurching around).

References

[1] *E155: Microprocessor-Based Systems.* Harvey Mudd College, 2016. Web: http://pages.hmc.edu/harris/class/e155/

[2] " *LIS3DH, MEMS digital output motion sensor*". STMicroelectronics. May 2010.

http://www.st.com/content/ccc/resource/technical/document/datasheet/3c/ae/50/85/d6/b1/46/fe/CD00274221.pdf/files/CD00274221.pdf/jcr:content/translations/en.CD00274221.pdf

[3] "*Tilt measurement using a low-g 3-axis accelerometer".* STMicroelectronics. June 2014.

http://www.st.com/content/ccc/resource/technical/document/application_note/d2/d6/22/4f/b9/8f/45/59/DM00119046.pdf/files/DM00119046.pdf/jcr:content/translations/en.DM00119046.pdf

Parts List

Part	Source	Vendor Part #	Price
Adafruit LIS3DH Breakout board	Adafruit	LIS3DHTR	$11.95 + $10.47 Shipping = $22.42
Bluesmirf Silver	In possession //Sparkfun	WRL-12577 ROHS	$25
DC Motors	E11 Lab // Polulu	BR010077	----
9V Battery	Eng. Stockroom	PC1604BKD	----
Powerizer NiMh 7.2V Battery	E11 Lab	LC14500S2R1WR	----

Appendix A: Raspberry Pi 3B C code read_L1S3DH.c

```
// read_L1S3DH.c
// jlauw@g.hmc.edu, yhegde@g.hmc.edu, Dec 6, 2016
//
// Reads acceleration reading from L1S3DH Accelerometer through SPI
// Calculates pitch, roll, corresponding motor speeds and sends via BT

#include <stdio.h>
#include "EasyPIO.h"
#include <math.h>

// L1S3DH Accel Registers Address
#define OUT_X_L 0x28
#define OUT_X_H 0x29
#define OUT_Y_L 0x2A
#define OUT_Y_H 0x2B
#define OUT_Z_L 0x2C
#define OUT_Z_H 0x2D

// L1S3DH Read / Write Registers
#define READ_ACC  1
#define WRITE_ACC 0

// MACROS that sets and clears bits
#define SET_BIT(p,n) ((p) |= (1 << (n)))
#define CLR_BIT(p,n) ((p) &= (~(1) << (n)))

//pin for button to brake vehicle
#define BRAKE_PIN 26



////////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////////
unsigned char accSPI0read(unsigned char address);
void accSPI0write(unsigned char address,unsigned char data);

void send(char left, char right);
char *anglesToSpeeds(int pitch, int roll);



////////////////////////////////////////////////
// Main
////////////////////////////////////////////////

int main(void){
```

```
// Intitialize GPIO pins and SPI0 on Pi
  pioInit();
  spiInit(1000000, 0);

  // Initialize axes readings from accelerometer
  short x_axis;
  short y_axis;
  short z_axis;

  char *p;     //Initialize pointer to Motor Speed Array
  int  pressed; //Initialize boolean to check if brake button pressed

//Write To Settings Registers on the Accelerometer
  accSPI0write(0x20,0x77); // highest conversion rate, all axis on
  accSPI0write(0x23,0x84); // block update, and high resolution

  pinMode(BRAKE_PIN,INPUT);

  int count = 0;

  while(1){

    //check if Brake button pressed
    pressed = digitalRead(BRAKE_PIN);

    //read WHO_AM_I register to confirm SPI functionality
    int test = (int) accSPI0read(0x0F);
    if (test != 51){break;} //WHO_AM_I register should give 51

    //Read 3-axis readings
      // Receive Axes readings from
    x_axis =  accSPI0read(OUT_X_L) | (accSPI0read(OUT_X_H)<<8);
    y_axis =  accSPI0read(OUT_Y_L) | (accSPI0read(OUT_Y_H)<<8);
    z_axis =  accSPI0read(OUT_Z_L) | (accSPI0read(OUT_Z_H)<<8);

    //Level - offset axes acceleration readings
    //  Divide by 16 since last 4 digits of SPI are always 0
    x_axis = (x_axis/16) + 392;
    y_axis = (y_axis/16) + 416;
    z_axis = (z_axis/16) + 452;

    //Cast to acceleration readings double for accurate calculation
    double x = (double) x_axis;
    double y = (double) y_axis;
    double z = (double) z_axis;

    //Calculate pitch and roll in radians
    double pitch_rad = atan((x/(sqrt(y*y + z*z))));
    double roll_rad  = atan((y/(sqrt(x*x + z*z))));

    //Convert pitch and roll in radians to degrees
    int roll  = (int) (roll_rad*57.3);
    int pitch = (int) (pitch_rad*57.3);
```

```
    //Calls function to get Motor Encodings based on pitch and roll
    p = anglesToSpeeds(pitch,roll);

    //Dereference array pointer to get motor speeds
    char leftSpeed = *p;
    char rightSpeed = *(p+1);

    //send only after every 5000 iterations of while loop
    if(count == 5000){

      //if button pressed, send 0 speeds
      if(pressed){
                  char brakeSpeed = 0b00000000;
                  send(brakeSpeed,brakeSpeed);
                  count = 0;
                  printf("brake");
        }
      //else send calculated speeds
        else{
        printf("Roll:  %d\n",roll);
        printf("Pitch: %d\n",pitch);
        printf("left speed: %d\n, right speed:    %d\n\n",leftSpeed,rightSpeed);
        send(leftSpeed,rightSpeed);
        count = 0;
      }

    }
    count+=1;
}
}



////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////

//Function READS from L1S3DH Registers
//    Timing diagram for L1S3DH SPI Read in p.23 datasheet
 unsigned char accSPI0read(unsigned char address)
{
SPI0CSbits.TA = 1; // turn SPI on with the "transfer active" bit

unsigned char first8bits;                       //first8bits sent out
unsigned char last8bits = 0b11111111;           //last8bits garbage for READ
unsigned char rval;                                     //return value

  //Set first 8 bits sent over SPI
first8bits = (address & 0x3F);            //Set ADDRESS of register to read
from
first8bits = first8bits | (1<<7);   //Set first bit to READ

spiSendReceive(first8bits);         //sends MSB first
```

```
rval = spiSendReceive(last8bits);  //

SPI0CSbits.TA = 0; // turn off SPI

return rval;
}

//Function WRITES data to L1S3DH Registers
//  Timing diagram for L1S3DH SPI Write in p.23 datasheet
 void accSPI0write(unsigned char address,unsigned char data)
{
  SPI0CSbits.TA = 1;
  unsigned char first8bits;

  first8bits = (address & 0x3F);  //Set ADDRESS of register to Write to

  spiSendReceive(first8bits); //spiSendReceive sends MSB first
  spiSendReceive(data);       //send data to write to register

  SPI0CSbits.TA = 0; // turn off SPI
}

//Function Encodes of Motor Speed and Direction based on Pitch and Roll
char *anglesToSpeeds (int pitch, int roll)
{
  //Array that stores 2 chars data encoding for Left and Right Motor
  static char x[2];

  //Encoded Motor Speed and Direction sent to Car
  char leftMotor;
  char rightMotor;

  int Lmotor;
  int Rmotor;

  // Motor Speed variable to store calculations in floats
  float Lmotorf;
  float Rmotorf;

  // Cast int pitch & roll to floats
  float pitchf = (float) pitch;
  float rollf  = (float) roll;

  // Boolean used store motor direction
  int backward;
  int forward;

  ////////////////////////////////////////////////
// ********* If PITCH IN DEAD ZONE ********* //
  ////////////////////////////////////////////////
  if (abs(pitch)<10) {
            ,
    Lmotorf = 0.0;
    Rmotorf = 0.0;
```

```
//////////////////////////////
//****    CHECK ROLL    ****//
//////////////////////////////

// If in Dead Zone but turn left
if (roll<-10) { //Slight Left turn
  if(roll>-60){
    r = r*(-1);
    Lmotorf = (Lmotorf-(127.0*r/60.0));
    Rmotorf = (Rmotorf+(127.0*r/60.0));
  }
  else{ //Max Left Turn
     Lmotorf = Lmotorf-127.0;
    Rmotorf = Rmotorf + 127.0;
  }
}

  // IF in Dead zone but turn right
else if(roll>10) { //Slight Right turn
        if(roll<60){
    Lmotorf = (Lmotorf+(127.0*r/60.0));
    Rmotorf = (Rmotorf-(127.0*r/60.0));
  }
        else{  // Max Right turn
    Lmotorf = Lmotorf + 127.0;
    Rmotorf = Rmotorf - 127.0;
        }
}

// ELSE Roll and Pitch in DEAD Zone
else{//do nothing
}

////////////////////////////////
//**** ENCODE MOTOR SPEED ****//
////////////////////////////////
//bit manipulation to cast to 8 bit char, MSB being the direction

//Left Motor
if(Lmotorf<0.0){//Reverse
  Lmotor = (int) Lmotorf;
  Lmotor = Lmotor*(-1);
  leftMotor = (char) Lmotor;
  leftMotor |= 1<<7;
}
else{//Forward
  Lmotor = (int) Lmotorf;
  leftMotor = (char) Lmotor;
}

//Right motor
if(Rmotorf<0.0){//Reverse
  Rmotor = (int) Rmotorf;
```

```
      Rmotor = Rmotor*(-1);
      rightMotor = (char) Rmotor;
      rightMotor |= 1<<7;
    }
    else{//Forward
      Rmotor = (int) Rmotorf;
      rightMotor = (char) Rmotor;
    }

  }

/////////////////////////////////////////////////////
// ********* ELSE PITCH NOT IN DEAD ZONE ********* //
  /////////////////////////////////////////////////////
else {

      // If pitch NEGATIVE, CAR GOES FORWARD
      if (pitch<0){

      //Set direction booleans set to indicate forward
      forward = 1;
            backward = 0;

      pitchf = -1.0*pitchf; //Absolute Magnitude of pitch
            if(p>60.0) { //set to max speed, 100% PWM duty cycle encoded in
7-bits
                  Lmotorf = 127.0;
        Rmotorf = 127.0;
            }
            else{
        //Calculations ensures 40% < PWM duty cycle < 100% encoded in 7-bits
        Lmotorf = (40.0 + ((pitchf-10.0)*1.2))*1.28;
        Rmotorf = Lmotorf;
      }
      }

      // If pitch POSITIVE, CAR REVERSES
    else if(pitch>0){

      //Set direction booleans set to indicate backward
      forward = 0;
      backward = 1;

      ////////////////////////////////////////////
      //**** CHECK MAGNITUDE OF PITCH FIRST ****//
      ////////////////////////////////////////////

      if(pitch>60) { //set to max speed, 100% PWM duty cycle encoded in
7-bits
        Lmotorf = 127.0;
        Rmotorf = 127.0;
      }
      else{
        //Calculations ensures 40% < PWM duty cycle < 100% encoded in 7-bits
```

```
        Lmotorf = (40.0 + ((pitchf-10.0)*1.2))*1.28;
        Rmotorf = Lmotorf;
      }

      /////////////////////////////
      //**** THEN CHECK ROLL ****//
      /////////////////////////////

        //CAR Turns Left
        if(roll <-10){
        rollf = rollf*(-1.0); //Absolute Magnitude of Roll
        if(roll<(-60)){ //Max Roll
          Lmotorf = Lmotorf-(2.0*Lmotorf); //tightest turn possible; wheels go
in opp. dir
          }
        else{ //Not Max Roll
          Lmotorf = Lmotorf-(2.0*Lmotorf*rollf/60.0); //
              }
        }

      //CAR Turns Right
        else if(roll > 10){
        if(roll>(60)){ //Max Roll
          Rmotorf = Rmotorf-(2.0*Rmotorf);
        }
        else{ //Not Max Roll
          Rmotorf = Rmotorf-(2.0*Rmotorf*rollf/60.0);
            }
      }

      else{//do nothing
      }

    ////////////////////////////////
    //**** ENCODE MOTOR SPEED ****//
    ////////////////////////////////

    if(forward){  //CAR Moves FORWARD

      //LEFT Motor Speed and Direction Encoding
            if(Lmotorf<0.0){ //Left Motor Reverse
        Lmotor = (int) Lmotorf;
                  Lmotor = Lmotor*(-1);
        leftMotor = (char) Lmotor;
        leftMotor |= 1<<7; //Set MSB to 1 for reverse direction
      }
      else{ //Left Motor slightly Forward
        Lmotor = (int) Lmotorf;
        leftMotor = (char) Lmotor;
            }

      //RIGHT Motor Speed and Direction Encoding
            if(Rmotorf<0.0){ //Right Motor Reverse
        Rmotor = (int) Rmotorf;
```

```
        Rmotor = Rmotor*(-1);
        rightMotor = (char) Rmotor;
        rightMotor |= 1<<7;
      }
      else{ //Right Motor Forward
        Rmotor = (int) Rmotorf;
        rightMotor = (char) Rmotor;
      }
    }

    //CAR Moves BACKWARD
    else if(backward){

      //RIGHT Motor Speed and Direction Encoding
      if(Lmotorf<=0.0){
          Lmotor = (int) Lmotorf;
          Lmotor = Lmotor*(-1);
          leftMotor = (char) Lmotor;
      }
      else{
          Lmotor = (int) Lmotorf;
          leftMotor = (char) Lmotor;
          leftMotor |= 1<<7;
      }

      //LEFT Motor Speed and Direction Encoding
      if(Rmotorf<=0.0){
          Rmotor = (int) Rmotorf;
          Rmotor = Rmotor*(-1);
          rightMotor = (char) Rmotor;
      }
      else{
          Rmotor = (int) Rmotorf;
          rightMotor = (char) Rmotor;
          rightMotor |= 1<<7;
      }
    }
  }

  //Store leftMotor to 1st element, rightMotor to 2nd element
  x[0] = leftMotor;
  x[1] = rightMotor;

  return x;
}

//function to send 2 bluetooth packets for left and right motors
void send(char left, char right){

  FILE *fp;

  fp = fopen("/dev/rfcomm0", "w");
  fprintf(fp,"%c",left);
  fclose(fp);
```

```
  int count;

  fp = fopen("/dev/rfcomm0", "w");
  fprintf(fp, "%c",right);
  fclose(fp);
}
```

Appendix B: Car System FPGA SystemVerilog Code rcfpga.sv

```
/////////////////////////////////////////////
// rcfpga.sv
// HMC E155 December 5 2016
// jlauw@hmc.edu, yhegde@hmc.edu
/////////////////////////////////////////////


//////////////////////////////////////////////////////
////////    !!!!!!!! FPGA MAIN !!!!!!!!!       //////
////////              RCFPGA                   //////
////////                                       //////
////////          clk_period:       25 ns      //////
////////          clk_frequency: 40 MHz        //////
//////////////////////////////////////////////////////

module rcfpga(input logic clk, reset,
              input logic bt_serial_rx,
            output logic left_HB_1A_pin, left_HB_2A_pin,
            output logic right_HB_3A_pin, right_HB_4A_pin,
            output logic pwm_left_pin, pwm_right_pin,
            output logic sample_left_motor, sample_right_motor,
            output logic [7:0] led_left, led_right,
            output logic [3:0] uart_state,
            output logic en_sample_rx,
            output logic next_motor_cfg);

//Synchronizer for Asynchronous Bluetooth rx
logic n1, sync_rx;
always_ff @(posedge clk)
      begin
            n1 <= bt_serial_rx;
            sync_rx <= n1;
      end

logic reset_rcfpga_main, reset_rcfpga;
assign reset_rcfpga = reset_rcfpga_main | reset;

//Main Module for RC Car
rcfpga_main rcfpga(clk, reset_rcfpga,
                        sync_rx,
                        left_HB_1A_pin, left_HB_2A_pin,
                        right_HB_3A_pin, right_HB_4A_pin,
                        pwm_left_pin, pwm_right_pin,
                        sample_left_motor, sample_right_motor,
                        led_left, led_right,
                        uart_state,
                        en_sample_rx,
                        next_motor_cfg);

//Module that Resets rcfpga_main when bluetooth
//is disconnected
```

```
check_BT_connect check_BT_connect(clk, reset,
                                  sync_rx,
                                  reset_rcfpga_main);
endmodule

/////////////////////////////////////////////////
//rcfpga_main
//   RC Car Top level module with UART interface
//      and Motor Speed Management
/////////////////////////////////////////////////
module rcfpga_main(input logic clk, reset,
            input logic serial_rx,
            output logic left_HB_1A_pin, left_HB_2A_pin,
            output logic right_HB_3A_pin, right_HB_4A_pin,
            output logic pwm_left_pin, pwm_right_pin,
            output logic sample_left_motor, sample_right_motor,
            output logic [7:0] led_left, led_right,
            output logic [3:0] uart_state,
            output logic en_sample_rx,
            output logic next_motor_cfg);

//Receive Bluetooth UART Module
logic [7:0] rx_parallel_out;
logic rx_parallel_ready, start_rx;

uart_rx uart_rx(clk, reset,
                  serial_rx,
                  rx_parallel_out,
                  rx_parallel_ready, start_rx,
                  uart_state,
                  en_sample_rx);

//Multipkt management Module
logic reset_multipkt_fsm;

multipkt_fsm multipkt_fsm(clk, reset_multipkt_fsm,
                          rx_parallel_ready,
                          sample_left_motor,
                          sample_right_motor,
                          next_motor_cfg);

logic exceed_lmtr_samptime;
assign reset_multipkt_fsm = reset | exceed_lmtr_samptime;

chk_rmtr_samptime chk_rmtr_samptime(clk, reset,

sample_right_motor,

exceed_lmtr_samptime);


// Flop enable to manage rx_parallel_out to Motor Module
logic [7:0] parallel_rx_left;
logic [7:0] parallel_rx_right;
```

```
always_ff @(posedge clk)
      if (reset) parallel_rx_left <= 8'b0;
      else if (sample_left_motor & rx_parallel_ready) parallel_rx_left <=
                                                            rx_parallel_out;

always_ff @(posedge clk)
      if (reset) parallel_rx_right <= 8'b0;
      else if (sample_right_motor & rx_parallel_ready) parallel_rx_right <=
                                                            rx_parallel_out;

//Motor Module
motor motor(clk, reset,
            parallel_rx_left,
            parallel_rx_right,
            next_motor_cfg,
            left_HB_1A_pin, left_HB_2A_pin,
            right_HB_3A_pin, right_HB_4A_pin,
            pwm_left_pin, pwm_right_pin);

// LED Displays speed of PWM for Right motor
flopenr8 flopenr_left(clk, reset, next_motor_cfg,
                     parallel_rx_left[7:0],
                     led_left[7:0]);

flopenr8 flopenr_right(clk, reset, next_motor_cfg,
                      parallel_rx_right[7:0],
                      led_right[7:0]);

endmodule

/////////////////////////////////////////////////
//check_rightMotor_sampleTime

//   Module that checks if sample_right_motor
//      high for longer than expected 86.8 microseconds
//      for 115200 baudrate
//      In this case, checks if sample_right_motor
//      high exceeds 100 microseconds which is
//      3472 clk cycles
/////////////////////////////////////////////////
module chk_rmtr_samptime(input logic clk, reset,
                        input logic sample_right_motor,
                        output logic exceed_lmtr_samptime);
logic reset_counter;

//resets counter whenever not sampling right motor
assign reset_counter = ~sample_right_motor | reset;

logic [31:0] counts_high;
counter32bit check_counter(clk, reset_counter,
                          counts_high);

//resets rcfpga when sampling right motor high
```

```
//for 100 microseconds or 3472 clk cycles
assign exceed_lmtr_samptime = (counts_high == 32'd3472);

endmodule

//////////////////////////////////////////////////
//check_BT_connect
//   Module that checks if Bluetooth lost
//      connection for 1 second
//////////////////////////////////////////////////
module check_BT_connect(input logic clk, reset,
                        input logic rx,
                        output logic reset_rcfpga);

logic reset_counter;

// resets counter when rx low, a.k.a start_bit
assign reset_counter = ~rx | reset;

logic [31:0] counts_high;
counter32bit check_counter(clk, reset_counter,
                          counts_high);

//resets rcfpga when bluetooth Tx remains high for 1 second
//          40,000,000 clk cycles == 1 clk cycle
assign reset_rcfpga = (counts_high == 32'd40000000);

endmodule



//////////////////////////////////////////////
// UART Receive Module
//   UART Receive Interface. Shifts in data packets
//      from bluetooth module serially and sends out
//      data packets to motor module in parallel
//      start-bit, 1 byte data, stop-bit per packet
//////////////////////////////////////////////

//////////////////////////////////////////////
// uart_rx
//      rx is synchronized in the Top Module
//   UART receive module
//      CLKS_per_Bit = 40MHz / (115200baud) = 347
//////////////////////////////////////////////
module uart_rx(input logic clk, reset,
              input logic rx,
              output logic [7:0] rx_out,
              output logic rx_parallel_ready,
              output logic start_rx,
              output logic [3:0] uart_state,
              output logic en_sample_rx);

//*** start_rx identifies beginning of start bit ***//
```

```
//logic start_rx;
//rx delayed by 1 clk cycle is sampled because of how counter resets
logic rx_delay;

always_ff @(posedge clk)
      rx_delay <= rx;

assign start_rx = ~rx & rx_delay & uart_state[0];

//*** COUNTERS to track sampling of rx bits ***//

///**************************************************************///
/*
// 40MHz / (115200*16) = 21.7
*/

//bitSampleTracker: runs on clk and resets after every
//                                    (CLKSPEED / (BAUDRATE * SamplingRate))
//                                    SamplingRate = 16
//counterBitSample: counts up when enabled after some clk cycle interval
//**Purpose: Bit Sample ensures that bits on rx line is sampled at the
//**               middle.

logic [8:0] bitSampleTrack;
logic rst_bitSampleTracker;

counter9bit bitSampleTracker(clk, rst_bitSampleTracker, bitSampleTrack);

//reset bit sample tracker when bit sample counter enabled
logic rst_bitSampleT;
assign rst_bitSampleT = (bitSampleTrack == 9'd22); //22, 21 in sim
assign rst_bitSampleTracker = reset | start_rx | rst_bitSampleT;

//bit sample counter counts up after 2 clk cycle
logic en_counter_bitsample;
assign en_counter_bitsample = (bitSampleTrack == 9'd21); //22-1

logic [3:0] bitsample_q;
logic rst_countBitSample;     //controlled by FSM
counter4bit_en counterBitSample(clk, en_counter_bitsample,
                               rst_countBitSample,
                               bitsample_q);

//overflows after 16 counts
///**************************************************************///

///**************************************************************///
/*
// 40MHz / 115200 = 347.2
*/

//numBitsTracker: runs on clk and resets after every
//                                  (CLKSPEED / BAUDRATE )
//counterNumBits: counts up when enabled after some clk cycle interval
```

```
//**Purpose: Number of Bits tracks the Bit Number that has been
//**               transferred in the datapacket.

logic [8:0] numBitsTrack;
logic rst_numBitsTracker;

//reset tracker when counter_numbits enabled
counter9bit numBitsTracker(clk, rst_numBitsTracker, numBitsTrack);

logic rst_numBitsT;
assign rst_numBitsT = (numBitsTrack == 9'd347); //rst after 347 clk
assign rst_numBitsTracker = reset | start_rx | rst_numBitsT;

logic en_counter_numbits;
assign en_counter_numbits = (numBitsTrack == 9'd346); //347-1

logic [3:0] numbits_q;
logic rst_numbits; //controlled by FSM
counter4bit_en counterNumBits(clk, en_counter_numbits,

rst_numbits,

numbits_q);
///*****************************************************************///

//*** FSM: uart_rx behavior ***//
uart_rx_fsm rx_fsm(clk, reset,
                   rx_delay,
                   start_rx,
                   bitsample_q, numbits_q,
                   rst_countBitSample, rst_numbits, en_sample_rx,
                   rx_parallel_ready,
                   uart_state);

//*** SIPO: uart_rx shift register ***//
logic [7:0] rx_outf;
uart_rx_sipo rx_sipo(clk, reset,
                     en_sample_rx, rx_delay,
                     rx_outf);

assign rx_out[7:0] = {rx_outf[0],rx_outf[1],rx_outf[2],rx_outf[3],

rx_outf[4],rx_outf[5],rx_outf[6],rx_outf[7]};

endmodule

//////////////////////////////////////////////
// uart_rx_fsm
//   FSM that controls the uart_rx module
//////////////////////////////////////////////
module uart_rx_fsm(input logic clk, reset,
                   input logic rx_delay,
                   input logic start_rx,
                   input logic [3:0] sample_count, numbits_count,
```

```
                  output logic rst_countBitSample,
                  output logic rst_numbits,
                  output logic en_sample_rx,
                  output logic ready_rx,
                  output logic [3:0] uart_state);

typedef enum logic [2:0] {S0, S1, S2, S3} statetype;
statetype state, nextstate;

always_ff @(posedge clk, posedge reset)
      if(reset) state <= S0;
      else    state <= nextstate;

//next state logic
always_comb
      case(state)

            //IDLE STATE
            S0: if(start_rx)  nextstate <= S1;
                  else       nextstate <= S0;

            //CONFIRM START STATE
            S1: if((sample_count == 7) & (~rx_delay))       nextstate <= S2;
                  else if((sample_count == 7) & (rx_delay)) nextstate <= S0;
                  else                                      nextstate <= S1;

            //SAMPLING STATE
            S2: if(numbits_count == 9) nextstate <= S3;
                  else                 nextstate <= S2;

            //STOP STATE
            S3: nextstate <= S0;

            default: nextstate <= S0;

      endcase

//Combinational logic to manage tracking of sample counts
logic sample14, sample14_delay;
logic sample15;
assign sample14 = (sample_count == 4'd14);
assign sample15 = (sample_count == 4'd15);
always_ff @(posedge clk)
      sample14_delay <= sample14;

logic ready_sample;
assign ready_sample = sample15 & sample14_delay;

//Output logic
assign rst_countBitSample = (state == S0) | ((sample_count == 8) &

(numbits_count == 4'b0));
assign rst_numbits = (state == S0);
assign en_sample_rx = (ready_sample & (state == S2));
```

```
assign ready_rx = (state == S3);

assign uart_state[0] = (state == S0);
assign uart_state[1] = (state == S1);
assign uart_state[2] = (state == S2);
assign uart_state[3] = (state == S3);

endmodule

/////////////////////////////////////////////
// uart_sipo
//   SIPO shift registers for uart_rx
/////////////////////////////////////////////
module uart_rx_sipo(input logic clk,
                        input logic reset,
                        input logic en,
                        input logic rx,
                        output logic [7:0] rx_out);

always_ff @(posedge clk)
      if (reset)   rx_out[7:0] <= 8'b0;
      else if (en) rx_out[7:0] <= {rx_out[6:0], rx};

endmodule

/////////////////////////////////////////////
// Counters
//          UART uses to track baud-rate
/////////////////////////////////////////////

// 9-bit counter used as dividers with enable
module counter9bit(input logic clk,
                  input logic reset,
                  output logic [8:0] q);

always_ff @(posedge clk)
      if(reset) q <= 0;
      else         q <= q+1'b1;

endmodule


//4-bit counter used for bit sampling
module counter4bit_en(input logic clk,
                     input logic en,
                     input logic reset,
                     output logic [3:0] q);

always_ff @(posedge clk)
      if(reset)    q <= 0;
      else if (en) q <= q+1'b1;

endmodule
```

```
// 8-BIT Enable Register
module flopenr8(input logic clk, reset, en,
               input logic [7:0] d,
               output logic [7:0] q);

always_ff @(posedge clk)
      if (reset)        q <= 0;
      else if (en)      q <= d;

endmodule



////////////////////////////////////////////
// multipkt_fsm
//   multipkt_fsm Module
//      Manages the multipkt data transfer over
//      Bluesmirf for controlling motor
////////////////////////////////////////////
module multipkt_fsm(input logic clk, reset,
                   input logic rx_parallel_ready,
                   output logic sample_left_motor,
                   output logic sample_right_motor,
                   output logic next_motor_cfg);

typedef enum logic [1:0] {S0, S1, S2} statetype;
statetype state, nextstate;

always_ff @(posedge clk, posedge reset)
      if(reset) state <= S0;
      else    state <= nextstate;

//delay state transition to next clk cycle
logic delayed_rx_parallel_ready;
always_ff @(posedge clk)
      if(reset) delayed_rx_parallel_ready <= 1'b0;
      else         delayed_rx_parallel_ready <= rx_parallel_ready;

//next state logic
always_comb
      case(state)

            //SAMPLING LEFT MOTOR STATE
            // S0 -> S1 when first data sample for left motor done
            S0: if(delayed_rx_parallel_ready) nextstate <= S1;
                  else                        nextstate <= S0;

            //SAMPLING RIGHT MOTOR STATE
            S1: if(delayed_rx_parallel_ready) nextstate <= S2;
                  else                        nextstate <= S1;

            //COMPLETE STATE for 1 clk cycle
            S2: nextstate <= S0;
```

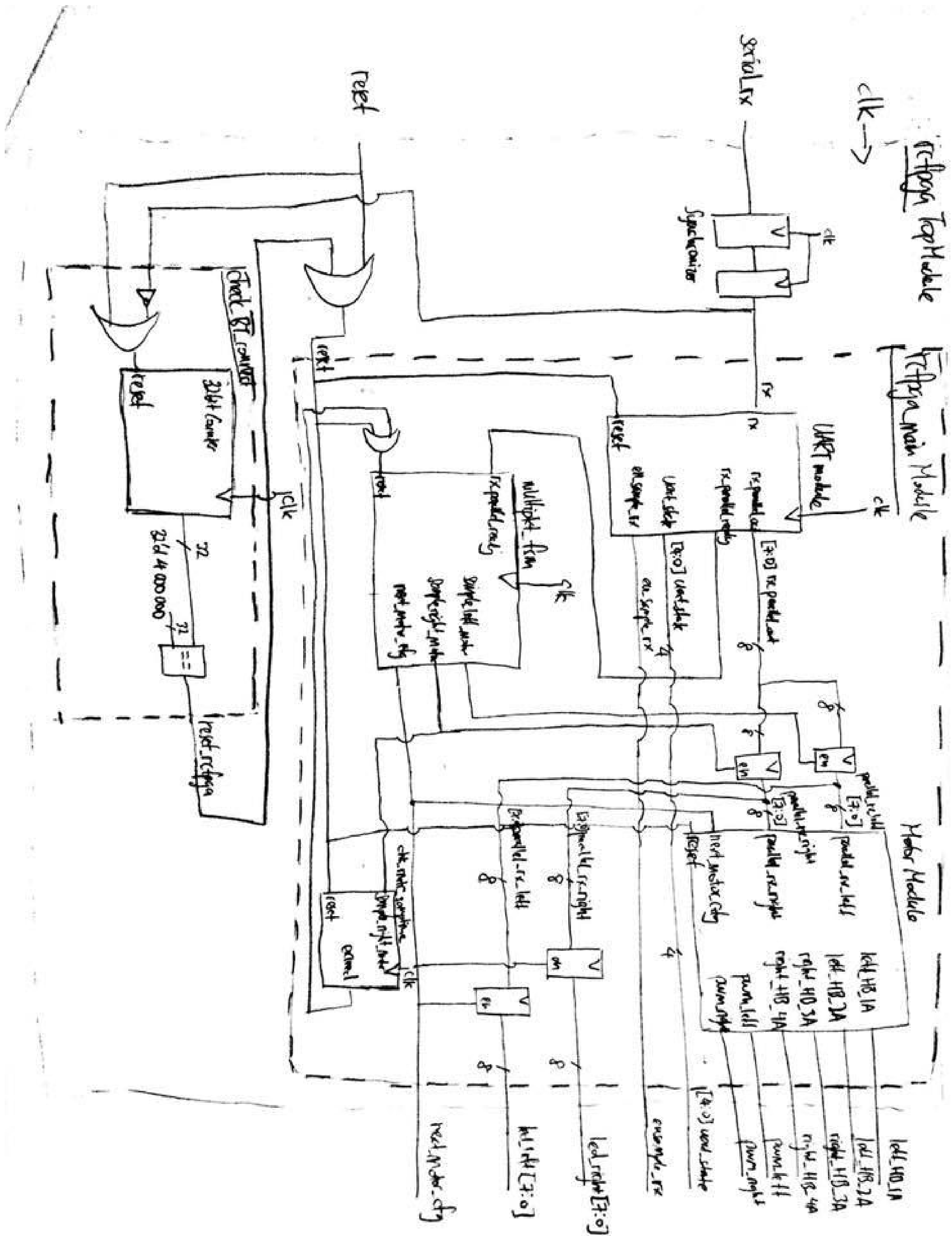
```
            default: nextstate <= S0;
      endcase

//output logic
assign sample_left_motor = (state == S0);
assign sample_right_motor = (state == S1);
assign next_motor_cfg = (state == S2);

endmodule



/////////////////////////////////////////////
// Motor Module
//   Controls the speed of the motor
//      based on data packet received from
//      UART Module
/////////////////////////////////////////////

/////////////////////////////////////////////
// motor
//   Motor Control module
//      Based on input from the remote control,
//      decides PWM and signal input to H-Bridge
//
//          motor direction: 1 = forward, 0 = reverse
/////////////////////////////////////////////
module motor(input logic clk, reset,
            input logic [7:0] parallel_rx_left,
            input logic [7:0] parallel_rx_right,
            input logic next_motor_cfg,
            output logic left_HB_1A, left_HB_2A,
            output logic right_HB_3A, right_HB_4A,
            output pwm_left, pwm_right);

// *** Control Logic *** //

//left Motor
always_ff @(posedge clk)
      if (reset) left_HB_1A <= 1'b0;
      else if (next_motor_cfg) left_HB_1A <= parallel_rx_left[7];

always_ff @(posedge clk)
      if (reset) left_HB_2A <= 1'b0;
      else if (next_motor_cfg) left_HB_2A <= ~parallel_rx_left[7];

//right Motor
always_ff @(posedge clk)
      if (reset) right_HB_3A <= 1'b0;
      else if (next_motor_cfg) right_HB_3A <= parallel_rx_right[7];

always_ff @(posedge clk)
      if (reset) right_HB_4A <= 1'b0;
      else if (next_motor_cfg) right_HB_4A <= ~parallel_rx_right[7];
```

```
// *** PWM *** //

//Counter divider is the count at which the main counter will
//reset itself
//Desired PWM_freq: 1000 Hz
//           clk_freq: 40 MHz
//          Period clk = 25 ns
//          Period PWM = 1 ms
//
//          Resolution for 7-bit = 1 ms / 2^7 = 7812.5 ns
//
~= 7813 ns
//
//          Counter_divider = 7813 ns / 25 ns ~= 313 clk cycles

logic [31:0] counter_divider;
assign counter_divider = 32'd313; //

logic enable_nextPWM_left, enable_nextPWM_right;
assign enable_nextPWM_left = next_motor_cfg;
assign enable_nextPWM_right = next_motor_cfg;

logic [6:0] OT_left, OT_right;
assign OT_left = parallel_rx_left[6:0];
assign OT_right = parallel_rx_right[6:0];

PWMgen PWM_left(clk, reset,
               counter_divider,
               enable_nextPWM_left,
               OT_left,
               pwm_left);

PWMgen PWM_right(clk, reset,
                counter_divider,
                enable_nextPWM_right,
                OT_right,
                pwm_right);

endmodule

/////////////////////////////////////////////
// motor_PWM
//   Pulse-Width Modulation module
//    <<Inputs>>
//          OT: Time high in terms of clk cycles,
//                 7-bit Resolution
/////////////////////////////////////////////
module PWMgen(input logic clk, reset,
             input logic [31:0] counter_div,
             input logic enable_nextPWM,
             input logic [6:0] OT,
             output logic PWMout);
```

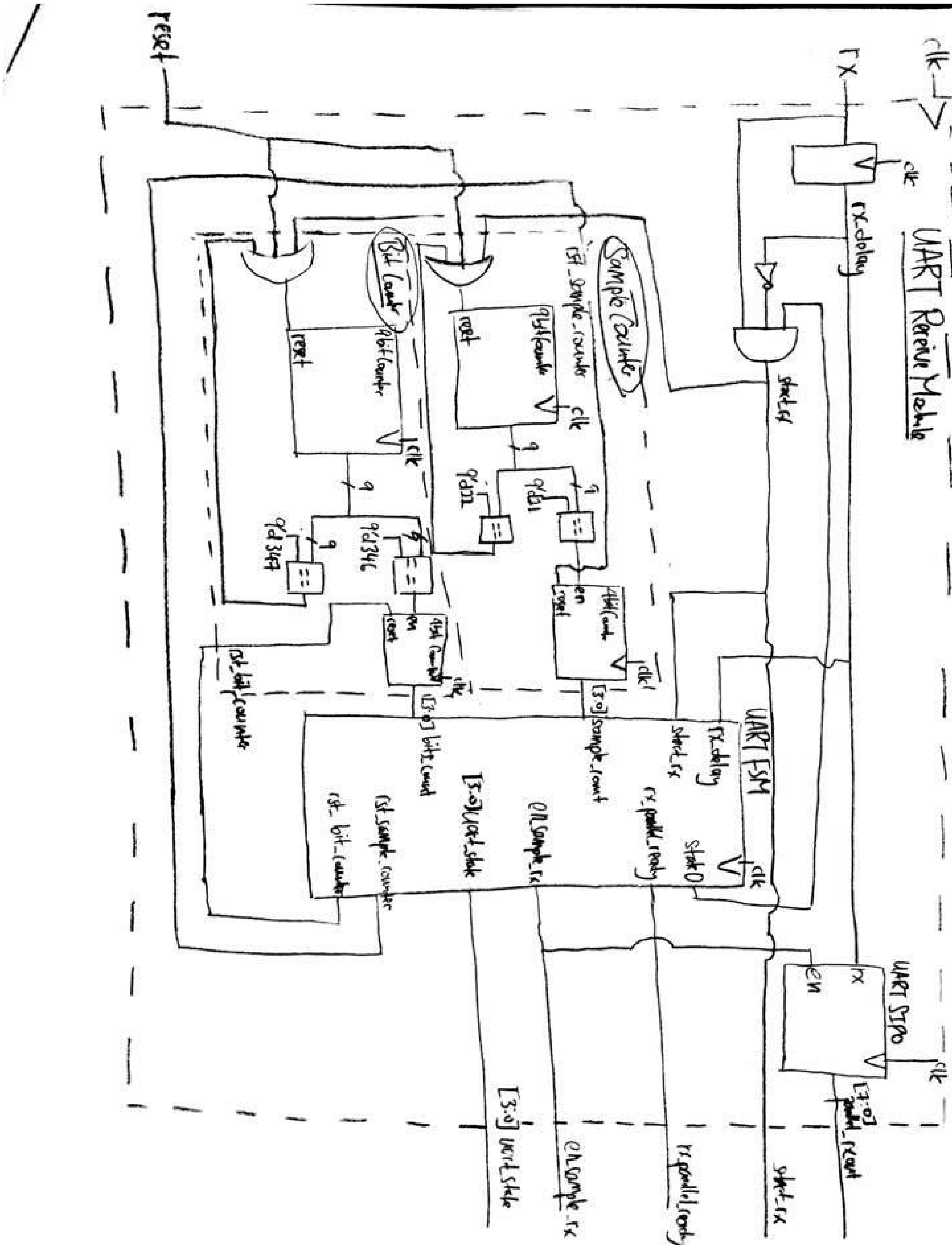
```
//Period of PWMgen is the maximum count of the
//counter_slow_en when the pwmCounter overflows

logic [6:0] OT_q;

always_ff @(posedge clk)
      if (reset)               OT_q <= 7'b0;
      else if (enable_nextPWM) OT_q <= OT;

logic [6:0] counts;
logic resetCounter;

//resets counter when new signal comes in
assign resetCounter = enable_nextPWM | reset;

counter_slow_en7bit pwmCounter(clk, resetCounter,
                                                       counter_div, counts);

assign PWMout = ((counts < OT_q) & ~reset);

endmodule

/////////////////////////////////////////////
// Main Counter
//          Motor uses to generate PWM signal
/////////////////////////////////////////////

// 32-bit counter run on clk
module counter32bit(input logic clk,
                   input logic reset,
                   output logic [31:0] q);

always_ff @(posedge clk)
      if(reset) q <= 0;
      else         q <= q+1'b1;

endmodule

/////////////////////////////////////////////
// Counter
//          Motor uses to generate PWM signal
/////////////////////////////////////////////

// Slow 7-bit Counter
module counter_slow_en7bit(input logic clk,
                          input logic reset,
                          input logic [31:0] counter_div,
                          output logic [6:0] q);

logic [31:0] main_count;
logic slow_counter_en;

counter32bit counter_main(clk, reset_mainCounter, main_count);
```

```
assign slow_counter_en = (main_count == counter_div);
assign reset_mainCounter = slow_counter_en | reset;

counter7bit counter_slow(clk, reset, slow_counter_en, q);

endmodule

// 7-bit counter used as dividers with enable
module counter7bit(input logic clk,
                   input logic reset,
                   input logic add_en,
                   output logic [6:0] q);

always_ff @(posedge clk)
      if (reset)              q <= 0;
      else if (add_en)  q <= q+1'b1;

endmodule
```

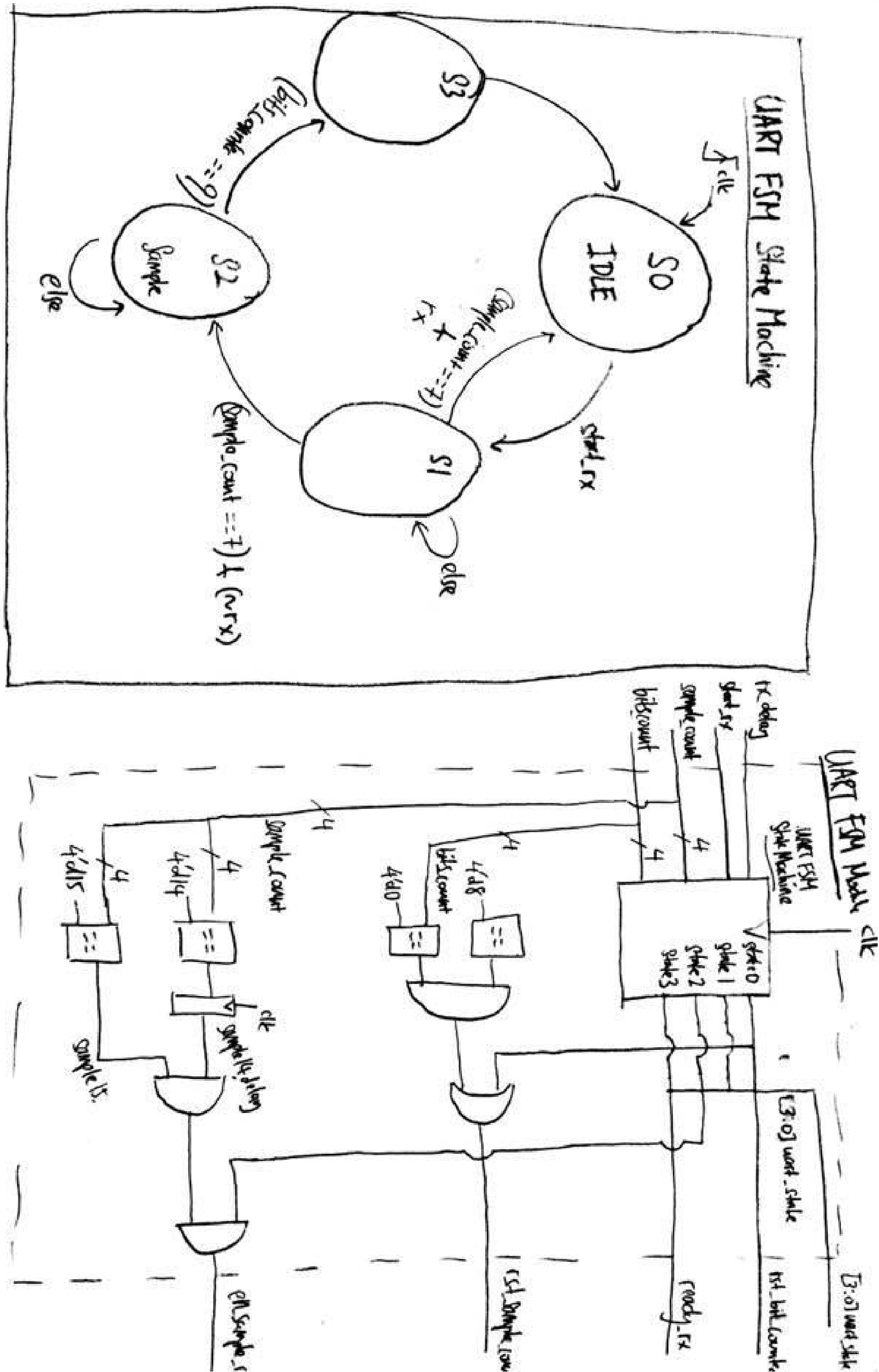
Appendix C: rcfpga Top Module Block Diagram



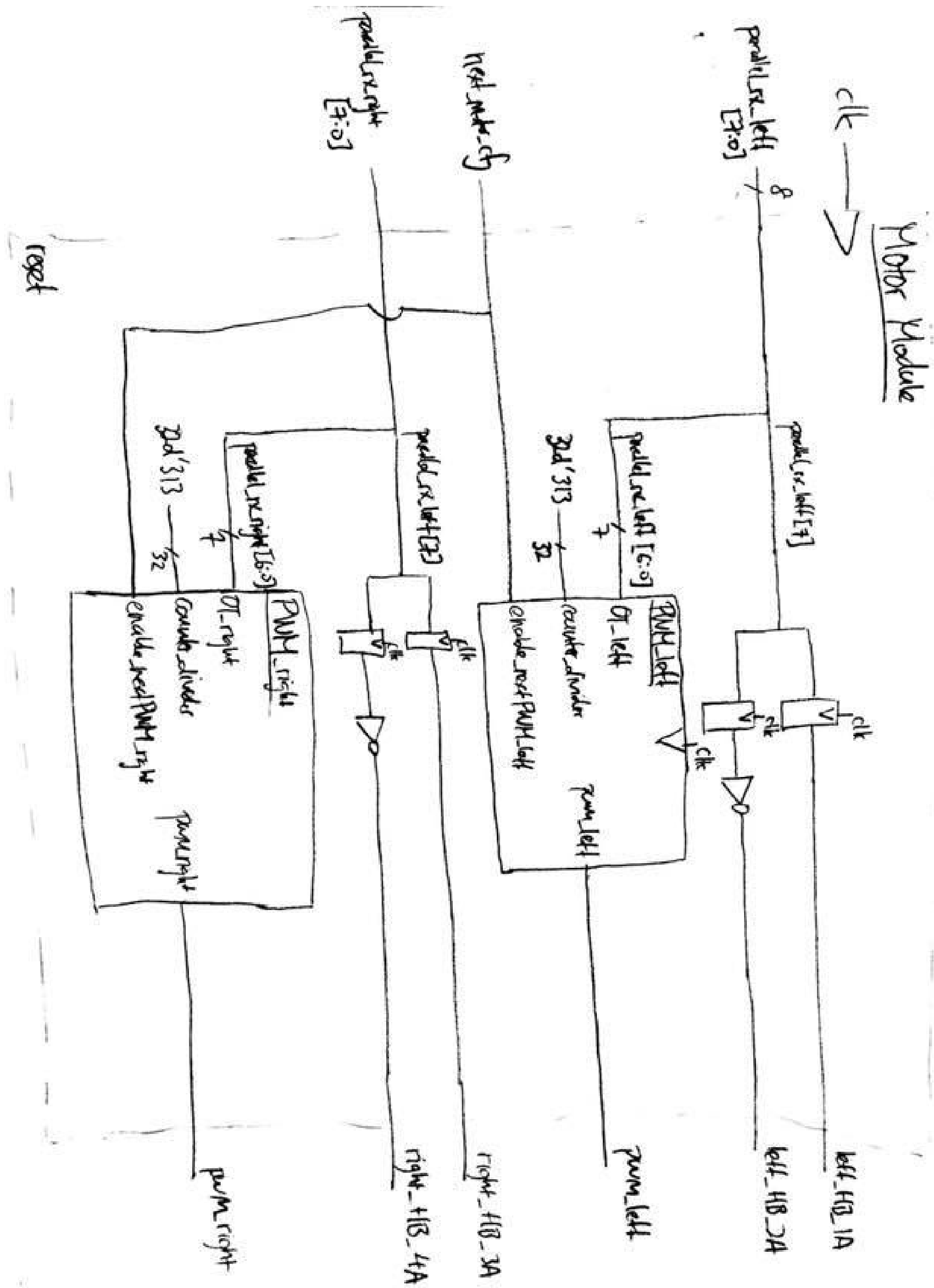
Appendix D: UART Receive Module Block Diagram



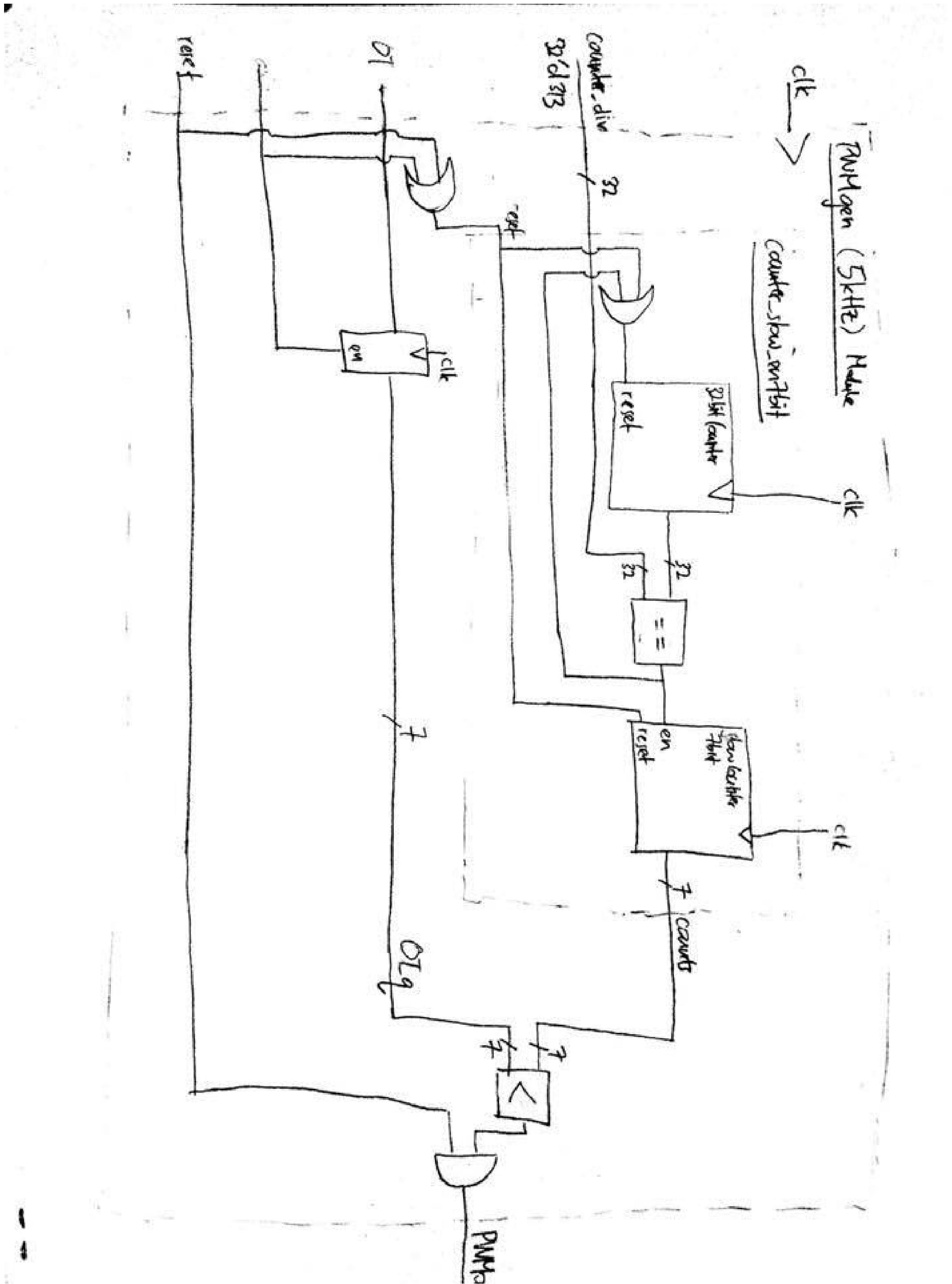
Appendix E: UART Receive Module FSM State Machine and Block Diagram



Appendix F: Motor Module Block Diagram



Appendix G: PWM generator Module Block Diagram



Appendix H: Pin Assignments for MuddPi Internal Circuit

rcfpga logic	FPGA pins
clk	Pin 88
led_left[7]	Pin 75 / led7
led_left[6]	Pin 74 / led6
led_left[5]	Pin 73 / led5
led_left[4]	Pin 72 / led4
led_left[3]	Pin 71 / led3
led_left[2]	Pin 70 / led2
led_left[1]	Pin 69 / led2
led_left[0]	Pin 68 / led10
reset	Pin 76 / SW[0]

Appendix I: Pin Assignments for MuddPi External Circuit

rcfpga logic	FPGA pins
serial_rx	Pin 7
uart[0]	Pin 31
uart[1]	Pin 32
uart[2]	Pin 33
en_sample_rx	Pin 42
next_motor_cfg	Pin 43
pwm_left_pin	Pin 51
pwm_right_pin	Pin 52
left_HB_1A_pin	Pin 54

left_HB_2A_pin	Pin 58
right_HB_3A_pin	Pin 59
right_HB_4A_pin	Pin 60
sample_left_motor	Pin 83
sample_right_motor	Pin 98
led_right[0]	Pin 99
led_right[1]	Pin 100
led_right[2]	Pin 103
led_right[3]	Pin 104
led_right[4]	Pin 105
led_right[5]	Pin 106
led_right[6]	Pin 110
led_right[7]	Pin 111