CS 684 :Embedded Systems, CSE Dept., IIT Bombay

Project Report

Spider Bot

Group 7

Team Leader :

Sanam Shakya (EE 2nd yr MTech, 123072001@iitb.ac.in)

Team Members :

Rahul Prajapat (EE 4th yr UG, 10d070039@iitb.ac.in)

Sanchar Sharma (EE 4th yr UG, 100070012@iitb.ac.in)

Ritubala Kachhwahe (CSE Mtech)

Supervisors :

Prof Kavi Arya

Prof Krithi Ramamritham

10 November, 2013

Abstract :

Our aim is to make a spider bot capable ofroaming around in a greenhouse 3 dimensionally. It will be used for monitoring purposes of plants in greenhouse, monitoring various parameters like temperature, humidity, video surveillance etc.

Content :

1. Introduction :

We are using Terasic's DE2i-150 FPGA development kit as a 'Brain' of our system. This kit consists of Intel atom N2600 processor and Altera cyclone series 4 FPGA. Atom is being used a process handler and FPGA as controller of our system.

We are using Foscam's IP camera model FI8910W for video surveillance purpose. This has inbuilt wifi module, a DSP processor and a controller. We are using IITB's wireless network as a hotspot and receiving video feed in lab by accessing camera's IP.

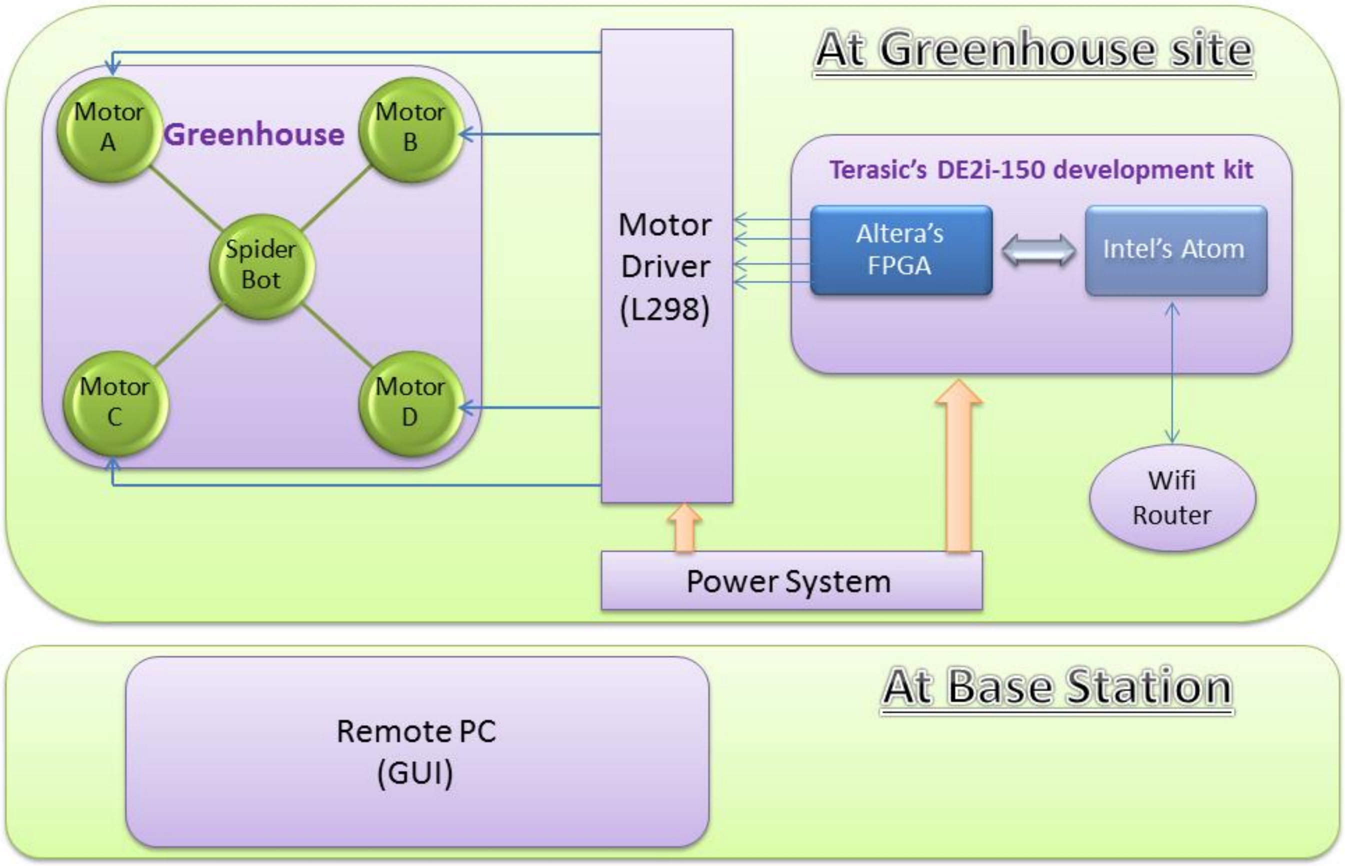
A web server has been made….

Spider bot has a chasis made of acrylic and is connected to four corners of greenhouse through nylon threads. Optimus[1] development board by Electronics Club, IIT Bombay is being used as controller sitting on spider chasis. An analog temperature sensor is connected to controller which is continuously giving temperature feed at various nodes in greenhouse. More sensors like humidity, gas sensors can be attached to this controller in near future.

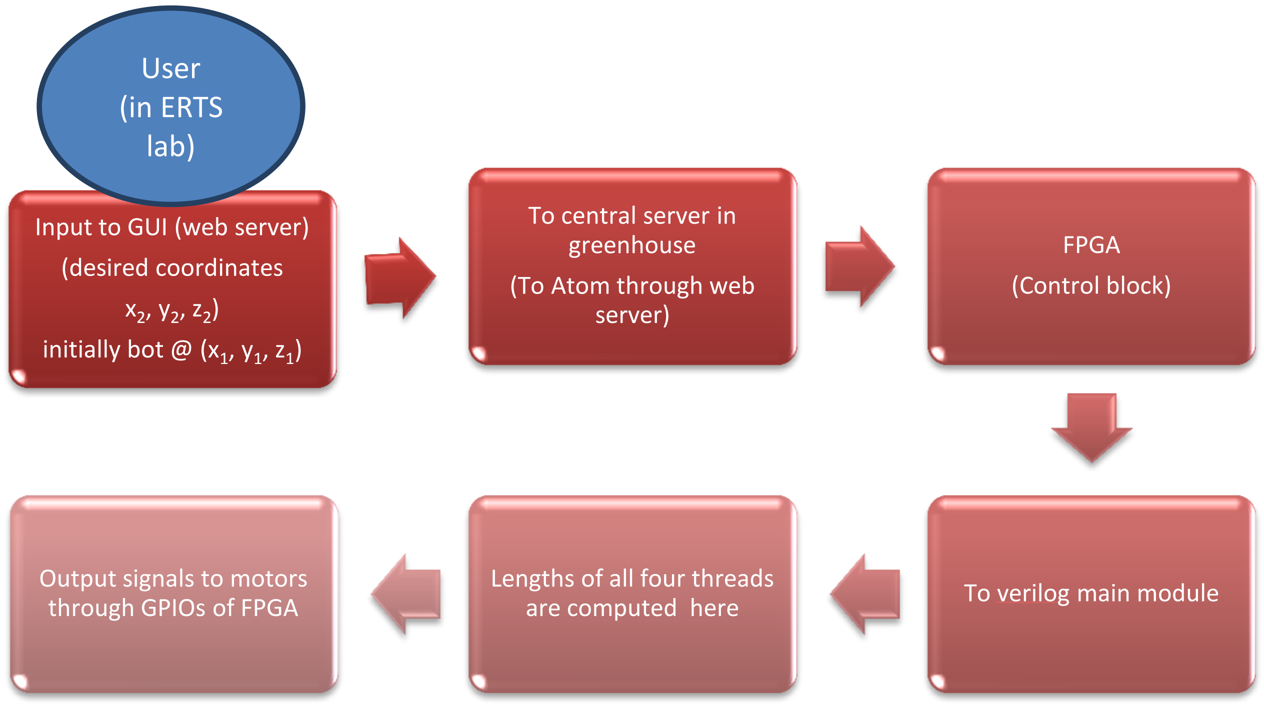
For communication between spider bot and central server (atom), we are using series 1 Xbees. Transmitter xbee is connected to a controller sitting on spider chasis and receiver xbee is connected to atom via usb.

For varying the lengths of the four threads, four high torque motors are established at four corners of the greenhouse. Motor drivers (L298) for these motors are placed just near the central board.

2. Product Design :



3. Process Flow :



4. Algorithms :

a) Navigation Algorithm :

Navigation system of spider bot was implemented in this way :

We can monitor certain plants planted in number of rows in greenhouse, so we can consider greenhouse as a 3D Cartesian plane of size p*q*r with center of greenhouse as (0,0,r) and plants at nodes (p,q,0) with $p \in (-n,n)$ where (2n+1) = number of plants in a row and $q \in (-m,m)$ where (2m+1) plant rows in greenhouse. In this way we can monitor plants robustly by navigating the spider bot periodically from one plant to another plant (node to node).

We can have a manual control as well which will be helpful in monitoring specific plant at any time we wish. Manual control input was the desired coordinates. Alternatively we can have a system with forward, backward, up and down functionality.

What we implemented is : The communication between motors and DE2i150 board is through Altera FPGA. A verilog code was written for the FPGA. The task of the code was to take the desired coordinates of the bot and then control the motor's speeds to navigate the bot to the desired location. The code will calculate the lengths of the ropes from each motor based on the given desired coordinates. Then it will provide the PWM output to each motor through a proportional controller for changing the lengths of the rope until the desired ropes' lengths' requirements are met. This can be understand properly in Verilog code section.

5. Hardware Design – Electronics Aspect :

a) Central Processing Unit

i) Terasic's DE2 i150 development board :

We issued the board from ERTS lab, CSE, IITB. Initially we thought to have a high end processor (Intel's Atom) for high end tasks and a microcontroller interfacing with outer world (motors, communication module). But since this board has an on board FPGA on it so we decided to use this powerful tool as our microcontroller.

User manuals and sample projects can be found in CD, which comes with this board.

(1) Intel Atom N2600 :

There are four folders in CD namely Demonstrations, Tools, Schematic, User manual. 'Getting started' doc for atom can be found at this location DE2_i150_CD\User manual\DE2i-150_QSG in CD.

We started exploring atom and found that there is a Yocto Linux OS running on it, a different Linux version say Ubuntu can also be installed on it, in fact we did it our later journey.

Initially we used Desktop for it, later we SSHed into it using through LAN port. IP of its LAN can be found using "avahi-browse –lar" system call. Later we were successful in SSHing into it wirelessly as well, DHCP enabled by default on its wireless adapter. One can use VNC remote server facility in Linux to see its file system.

The Atom processor on DE2i150 board was used as the user interface. It has two main interfaces from the outside world. One was the interface between Atom and FPGA. That is required because the desired coordinates of the bot has to be fed by the user to Atom and not directly to FPGA. Therefore, Atom needs to communicate the information to FPGA. Second interface is the wireless communication between IP Camera and Atom. Originally, we received the Atom board with Yoctolinux installed on it. However, we installed Ubuntu as there were software issues with Yocto pertaining to wireless.

We also used Atom's USB peripheral to interface with Xbee USB adaptor board (used for communicating between spider bot and atom).

We had GUI running on it and web server as well which will be described later.

(2) Altera FPGA cyclone series 4 :

We started with a sample program given in CD for this FPGA, documentation of this first sample program can be located hereDE2_i150_CD\User manual\My_First_Fpga.

User manual of cyclone 4 can be located here DE2_i150_CD\User manual\DE2i-150_FPGA_System_manual, which can be used for seeing the articles like 'interfacing FPGA's GPIOs, LEDs, Switches' and similar things.

We wrote our navigation algorithmic code in Verilog only. In the mid demo we used firebird's microcontroller (atmega2560) to run this navigation algorithm. More on Verilog code in further sections.

(3) Communication between Atom and FPGA :

They are connected through PCIe line on board.We tried this communication, but didn't get success. Initially we tried DE2_i150_CD\Demonstrations\PCIe_SW_KIT tutorial to make it work, we installed the driver successfully on yocto running on Atom, then when we tried to run an application (DE2_i150_CD\Demonstrations\FPGA\PCIE_Fundamental) on this PCIe line but the problem we encountered that the .sof file generated through verilog code for FPGA side wasn't working, hence we weren't upload it on FPGA. So here the problem was from FPGA side, driver installation in Atom side was quite peacefully done.

We tried this tutorial as well, but no success http://rijndael.ece.vt.edu/de2i150/designs/hellopci.pdf

We got an idea to establish communication between these two hardware though normal serial ports, we got the serial (UART) application code for FPGA and implemented it successfully.

b) IP camera :

i) IP settings on camera through 'IP camera tool' :

The camera interface can be opened by typing in the IP address of the camera in the browser. Also remember to set the address to the no proxy list. In order to connect to a WLAN network, the settings including SSID and key should be set in the interface after connecting through ethernet. Upon opening the camera interface, a login interface is presented. If using FF or chrome, use the 'Click to login' button which presents a dialog box to input use name. The default username is 'admin' and password is blank.

ii) IP Camera GUI Settings :

Upon login there are 3 tabs on left. First one is device info, the second one is the live camera video and the last one is the device settings. The live video option opens an interface to watch the streaming video and an interface to control the camera. On clicking the last option, the settings, various options are presented on the left panel. The ones to notice are WLAN network and Network connection settings. In order to connect to the network, we need to input the ssid and key, if any into the WLAN network settings. There is also a checkbox which is to be checked to use WLAN connection. The SSID is the name of the wireless network to which the camera should be connected. Available wireless networks can be scanned by clicking the Scan button provided.

Once correct settings are input, clicking on submit reboots the camera and it will try to automatically connect to the network. The camera interface might not be opened if the IP settings are not correct. Generally, with a Infrastructure network, using DHCP settings would suffice. In case that does not work, then connect another device to the router to see what is the subnet mask and IP address range provided by the router and set accordingly.

iii) <u>Specifications of IP camera :</u>

- Model: FI8918W by FOSCAM
- Display Resolution 640x480 Pixels (300k Pixels
- Input Built-In Microphone, Output Built-In Speaker
- Audio Compression ADPCM
- Image Compression MJPEG
- Image Frame Rate 15fps(VGA), 30fps (QVGA).
- Wireless Standards IEEE 802.11 b/g, Data Rate 802.11 b/g: 11 MBPS (max), 802.11g: 54MBPS (max), Wireless Security WEP & WPA WPA2 Encryption.
- Pan/Tilt Angle Horizontal: 300°ree; & Vertical: 120°ree
- Infrared Light 11 IR, LEDs, Night visibility upto 8 meters
- Gross Weight 768g (Color box size: 200x124x189mm)
- Net Weight 418g (Accessories Included)
- CPU 2.0GHz, Memory Size 256 MB, Display Card 64M, Supported OS Microsoft Windows 2000 / XP / Vista / 7 / Mac, Browser IE 6.0, IE 7.0, Firefox, Safari (no sound), and other standard browser.

c) **Wifi Router :**

Earlier article on IP camera was with windows installed on a processor (laptop). In our case yoctolinux was there in Atom.

First we tried it in Ubuntu. In ubuntu, we tried to make a hotspot with WPA2 security protocol, but then we realized that ubuntu can create hotspot only with some WEP protocol. That's why we needed a router to create a hotspot with WPA2 security protocol.

d) **Controller on Spider Bot : Optimus**

This is normal atmega32u4 based development board being developed by Electronics Club, IIT Bombay. Documentation of the board can be found here : http://stab-iitb.org/wiki/Optimus

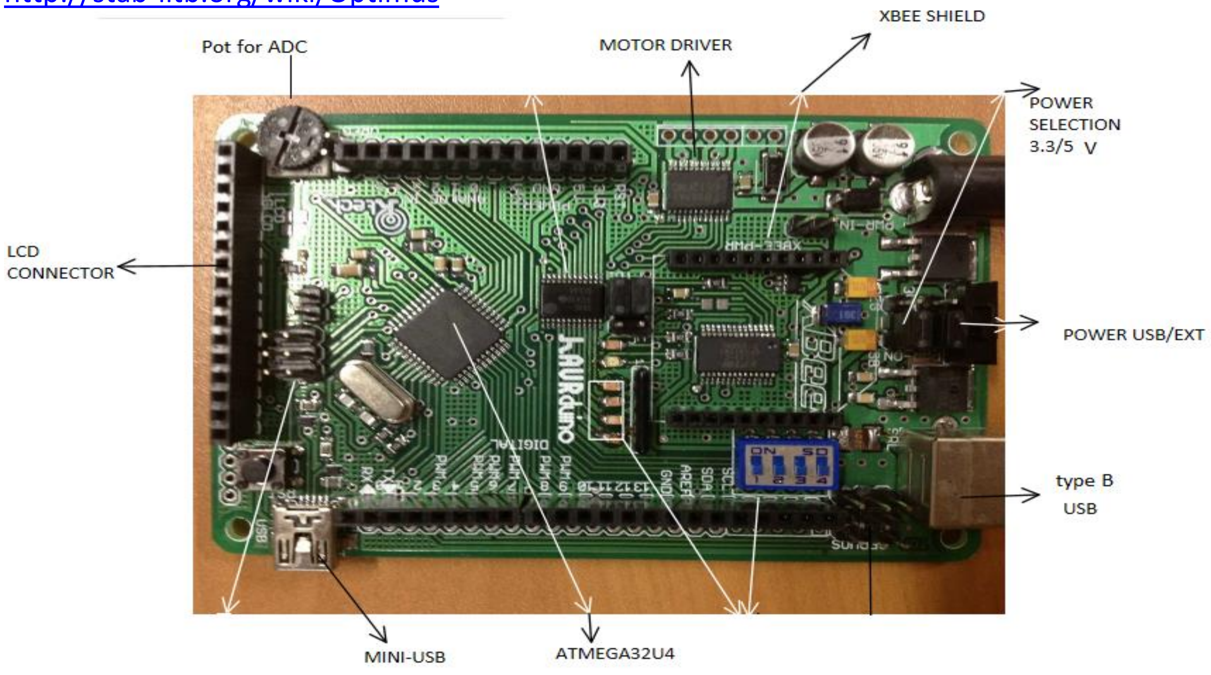
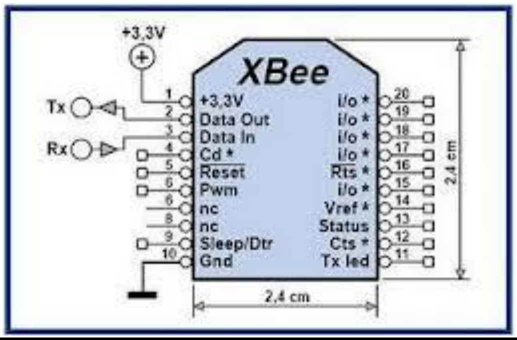
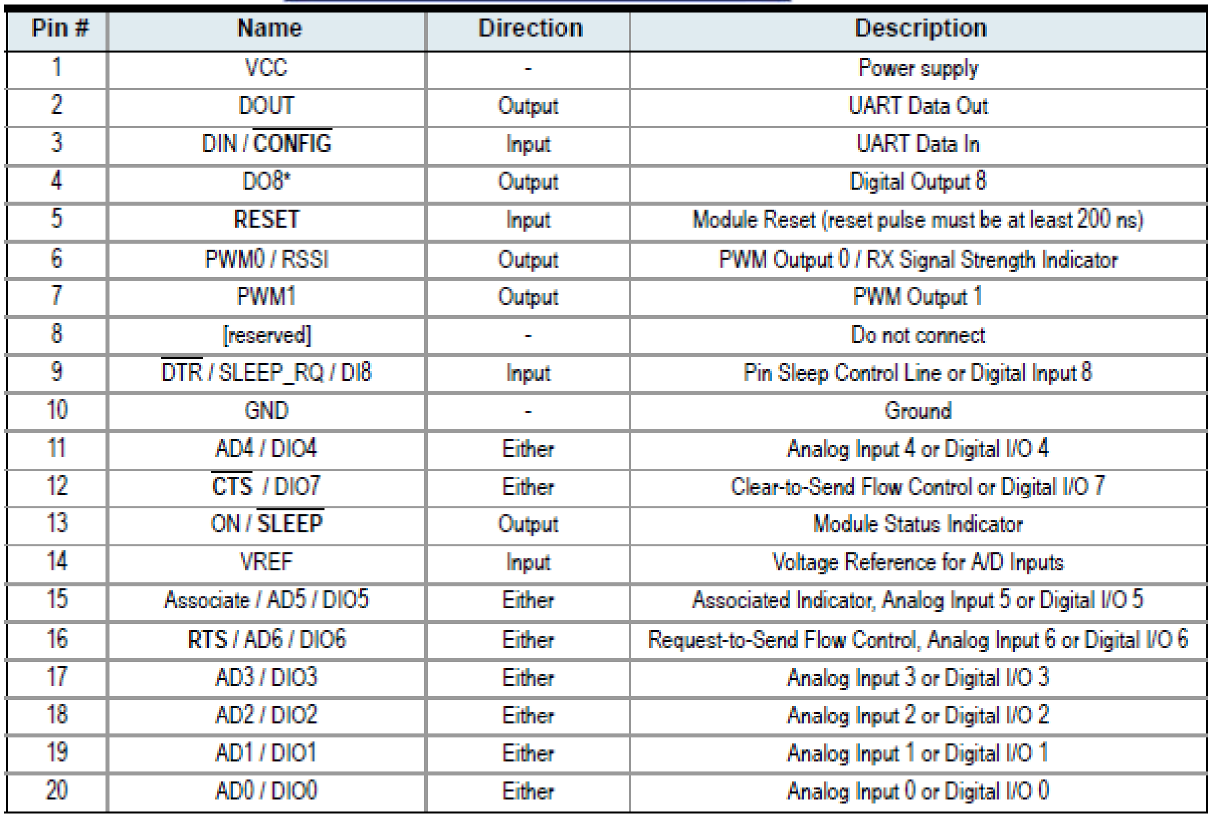


Figure 8 : Atmega32u4 based central processing unit

e) Communication Module – Xbees





Pin #	Name	Direction	Description
1	VCC	-	Power supply
2	DOUT	Output	UART Data Out
3	DIN / $\overline{\text{CONFIG}}$	Input	UART Data In
4	DO8*	Output	Digital Output 8
5	$\overline{\text{RESET}}$	Input	Module Reset (reset pulse must be at least 200 ns)
6	PWM0 / RSSI	Output	PWM Output 0 / RX Signal Strength Indicator
7	PWM1	Output	PWM Output 1
8	[reserved]	-	Do not connect
9	$\overline{\text{DTR}}$ / SLEEP_RQ / DI8	Input	Pin Sleep Control Line or Digital Input 8
10	GND	-	Ground
11	AD4 / DIO4	Either	Analog Input 4 or Digital I/O 4
12	$\overline{\text{CTS}}$ / DIO7	Either	Clear-to-Send Flow Control or Digital I/O 7
13	ON / $\overline{\text{SLEEP}}$	Output	Module Status Indicator
14	VREF	Input	Voltage Reference for A/D Inputs
15	Associate / AD5 / DIO5	Either	Associated Indicator, Analog Input 5 or Digital I/O 5
16	RTS / AD6 / DIO6	Either	Request-to-Send Flow Control, Analog Input 6 or Digital I/O 6
17	AD3 / DIO3	Either	Analog Input 3 or Digital I/O 3
18	AD2 / DIO2	Either	Analog Input 2 or Digital I/O 2
19	AD1 / DIO1	Either	Analog Input 1 or Digital I/O 1
20	AD0 / DIO0	Either	Analog Input 0 or Digital I/O 0

Figure 5 : Xbee pin configuration

f) Power system

i) Battery

(1) Calculation :

- **On spider bot**
  - (i) IP Camera - works at 5V, max 1A.
  - (ii) Microcontroller – max 300mA @5V
  - (iii) Xbee – <1A @3.3V

ii) Voltage regulator

We used LM7805.

g) Motor driver

We used L298 for four motors.

6. Hardware Design – Mechanical Aspect

a) Chasis

Four chasis platform were developed which consists of motor frame and support system, cable spool and encoders. These frames were kept at four corners of room on plywood platform.

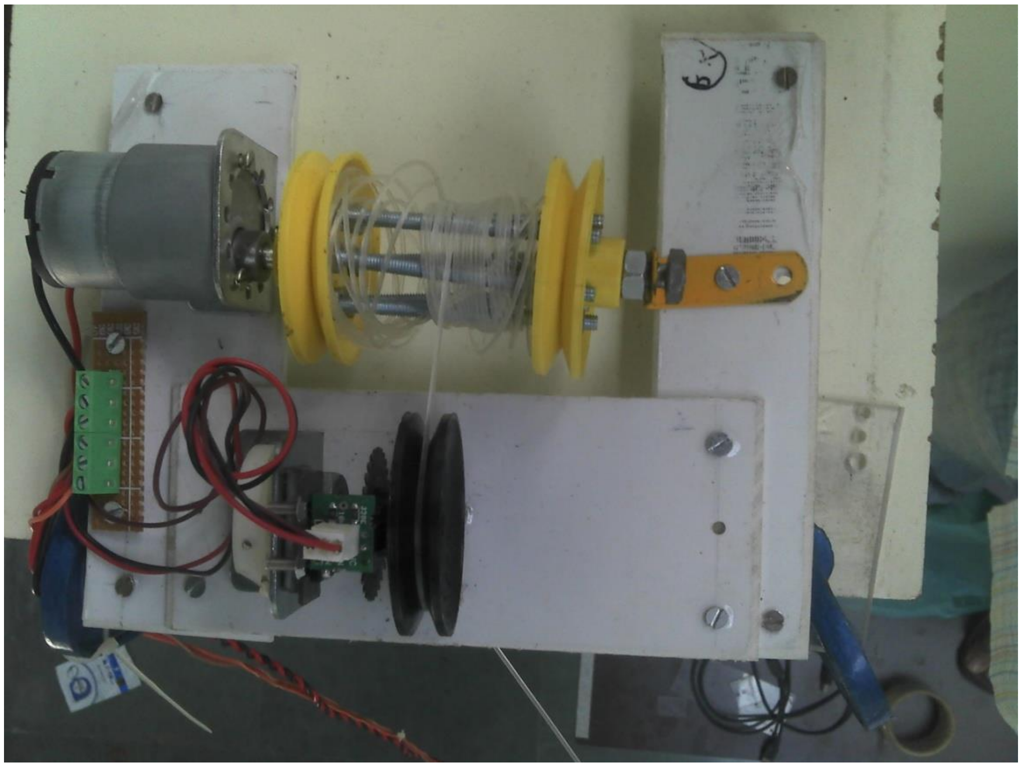


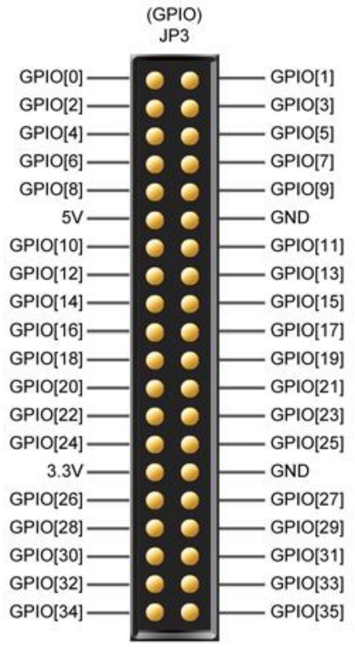
Fig. Motor chasis

b) Motors

On the above mentioned chasis 60 rpm motor were mounted which was coupled to the spool.

c) Cables

All the four motors power supply were given from 2 L298 board which was then controlled by the FPGA. Apart from that encoders power and signal from each motor frames were also connected to GPIO of FPGA.



GPIO pins	Functions
0	Encoder 1
1	Encoder 2
2	Encoder 3
3	Encoder 4
4	PWM1

5	PWM2
6	PWM3
7	PWM4
8	+5V
10-13	A1,B1,C1,D1
16-19	A2,B2,C2,D2

7. Software Design

a. Controller Programming

i. Verilog Coding on FPGA : Simple Version

For simple control it consists of three main programs:

1. Uart.v – which provides the serial communication for FPGA side.
2. PWM_8bit.v – which helps to provide Pulse Width Modulated signals for controlling the speed of the motor and direction control.
3. Simplemotion.v- which helps to connect above modules.

Uart.v-code

```
`timescale 1ns / 1ps
// Documented Verilog UART
// Copyright (C) 2010 Timothy Goddard (tim@goddard.net.nz)
// Distributed under the MIT licence.
//
// Permission is hereby granted, free of charge, to any person
obtaining a copy
// of this software and associated documentation files (the
"Software"), to deal
// in the Software without restriction, including without limitation
the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell
// copies of the Software, and to permit persons to whom the Software
is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be
included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT
SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN
```

```
// THE SOFTWARE.
//
module uart(
    input clk, // The master clock for this module
    input rst, // Synchronous reset.
    input rx, // Incoming serial line
    output tx, // Outgoing serial line
    input transmit, // Signal to transmit
    input [7:0] tx_byte, // Byte to transmit
    output received, // Indicated that a byte has been received.
    output [7:0] rx_byte, // Byte received
    output is_receiving, // Low when receive line is idle.
    output is_transmitting, // Low when transmit line is idle.
    output recv_error // Indicates error in receiving packet.
    );

parameter CLOCK_DIVIDE = 1302; // clock rate (50Mhz) / (baud rate
(9600) * 4)

// States for the receiving state machine.
// These are just constants, not parameters to override.
parameter RX_IDLE = 0;
parameter RX_CHECK_START = 1;
parameter RX_READ_BITS = 2;
parameter RX_CHECK_STOP = 3;
parameter RX_DELAY_RESTART = 4;
parameter RX_ERROR = 5;
parameter RX_RECEIVED = 6;

// States for the transmitting state machine.
// Constants - do not override.
parameter TX_IDLE = 0;
parameter TX_SENDING = 1;
parameter TX_DELAY_RESTART = 2;

reg [10:0] rx_clk_divider = CLOCK_DIVIDE;
reg [10:0] tx_clk_divider = CLOCK_DIVIDE;

reg [2:0] recv_state = RX_IDLE;
reg [5:0] rx_countdown;
reg [3:0] rx_bits_remaining;
reg [7:0] rx_data;

reg tx_out = 1'b1;
reg [1:0] tx_state = TX_IDLE;
reg [5:0] tx_countdown;
reg [3:0] tx_bits_remaining;
reg [7:0] tx_data;

assign received = recv_state == RX_RECEIVED;
assign recv_error = recv_state == RX_ERROR;
assign is_receiving = recv_state != RX_IDLE;
assign rx_byte = rx_data;

assign tx = tx_out;
assign is_transmitting = tx_state != TX_IDLE;
```

```
always @(posedge clk) begin
    if (rst) begin
        recv_state = RX_IDLE;
        tx_state = TX_IDLE;
    end

    // The clk_divider counter counts down from
    // the CLOCK_DIVIDE constant. Whenever it
    // reaches 0, 1/16 of the bit period has elapsed.
   // Countdown timers for the receiving and transmitting
    // state machines are decremented.
    rx_clk_divider = rx_clk_divider - 1;
    if (!rx_clk_divider) begin
        rx_clk_divider = CLOCK_DIVIDE;
        rx_countdown = rx_countdown - 1;
    end
    tx_clk_divider = tx_clk_divider - 1;
    if (!tx_clk_divider) begin
        tx_clk_divider = CLOCK_DIVIDE;
        tx_countdown = tx_countdown - 1;
    end

    // Receive state machine
    case (recv_state)
        RX_IDLE: begin
            // A low pulse on the receive line indicates the
            // start of data.
            if (!rx) begin
                // Wait half the period - should resume in the
                // middle of this first pulse.
                rx_clk_divider = CLOCK_DIVIDE;
                rx_countdown = 2;
                recv_state = RX_CHECK_START;
            end
        end
        RX_CHECK_START: begin
            if (!rx_countdown) begin
                // Check the pulse is still there
                if (!rx) begin
                    // Pulse still there - good
                    // Wait the bit period to resume half-way
                    // through the first bit.
                    rx_countdown = 4;
                    rx_bits_remaining = 8;
                    recv_state = RX_READ_BITS;
                end else begin
                    // Pulse lasted less than half the period -
                    // not a valid transmission.
                    recv_state = RX_ERROR;
                end
            end
        end
        RX_READ_BITS: begin
            if (!rx_countdown) begin
                // Should be half-way through a bit pulse here.
                // Read this bit in, wait for the next if we
                // have more to get.
                rx_data = {rx, rx_data[7:1]};
                rx_countdown = 4;
                rx_bits_remaining = rx_bits_remaining - 1;
                recv_state = rx_bits_remaining ? RX_READ_BITS :
RX_CHECK_STOP;
            end
        end
        RX_CHECK_STOP: begin
            if (!rx_countdown) begin
                // Should resume half-way through the stop bit
                // This should be high - if not, reject the
                // transmission and signal an error.
                recv_state = rx ? RX_RECEIVED : RX_ERROR;
            end
        end
        RX_DELAY_RESTART: begin
            // Waits a set number of cycles before accepting
            // another transmission.
            recv_state = rx_countdown ? RX_DELAY_RESTART : RX_IDLE;
        end
        RX_ERROR: begin
            // There was an error receiving.
            // Raises the recv_error flag for one clock
            // cycle while in this state and then waits
            // 2 bit periods before accepting another
            // transmission.
            rx_countdown = 8;
            recv_state = RX_DELAY_RESTART;
        end
        RX_RECEIVED: begin
            // Successfully received a byte.
            // Raises the received flag for one clock
            // cycle while in this state.
            recv_state = RX_IDLE;
        end
    endcase

    // Transmit state machine
    case (tx_state)
        TX_IDLE: begin
            if (transmit) begin
                // If the transmit flag is raised in the idle
                // state, start transmitting the current content
                // of the tx_byte input.
                tx_data = tx_byte;
                // Send the initial, low pulse of 1 bit period
                // to signal the start, followed by the data
                tx_clk_divider = CLOCK_DIVIDE;
                tx_countdown = 4;
                tx_out = 0;
                tx_bits_remaining = 8;
                tx_state = TX_SENDING;
            end
        end
        TX_SENDING: begin
            if (!tx_countdown) begin
                if (tx_bits_remaining) begin
```

```
                    tx_bits_remaining = tx_bits_remaining - 1;
                    tx_out = tx_data[0];
                    tx_data = {1'b0, tx_data[7:1]};
                    tx_countdown = 4;
                    tx_state = TX_SENDING;
                end else begin
                    // Set delay to send out 1 stop bit
                    tx_out = 1;
                    tx_countdown = 4;
                    tx_state = TX_DELAY_RESTART;
                end
            end
        end
        TX_DELAY_RESTART: begin
            // Wait until tx_countdown reaches the end before
            // we send another transmission. This covers the
            // "stop bit" delay.
            tx_state = tx_countdown ? TX_DELAY_RESTART : TX_IDLE;
        end
    endcase
end

endmodule
```

PWM_8bit.v

```
//Avoid giving speed as h10 for the sake of motor
//Here en is different than multiplication. When en=0, outputs are
reset to zero. When en=1, normal PWM signal.
//clk defines the time period of the PWM signal
//speed is signed. So if speed[4]=1, the actual speed is negative with
value being two's complement
//For the sake of positive and negative speeds, two outputs are
provided, namely outp and outn
module PWM_timer(input clk, input [8:0] speed, input en, output out,
output dirp, dirn);

    reg [8:0] speed_copy; initial speed_copy<=0;

    reg outlogic; initial outlogic<=0;//This is the output of the logic
circuit. It has to be multiplexed with 0 depending on en
    assign out = (en ? outlogic : 0); //If en is off, outp=0

    reg [7:0] count; initial count<=0;//Keeping a count of the cycles
elapsed

    always @(negedge count[7]) speed_copy <= (speed[8] ? ~speed[7:0]+1
: speed[7:0]); //Latching speed

    always @(negedge clk) begin
        count<=count+1;
        if (count==0 && speed_copy[7:0]!=0) begin
            outlogic<=1;
        end else if (count==speed_copy[7:0]) begin
            outlogic<=0;
        end
    end

    assign dirp = (~speed_copy[8]);
    assign dirn = (speed_copy[8]);

endmodule
```

simple_motion.v

```
module simple_move (input clk, input pwm_clk,input en_PWM, input rx,
output tx, output rx_flag_out, output [7:0] rx_reg_out, output [3:0]
sig_to_mot, dirp, dirn);

    parameter FOR="f"; parameter BCK="b"; parameter RHT="r"; parameter
LFT="l";
    parameter UPW = "u"; parameter DNW = "d"; parameter STOP = "s";
    parameter SPEED_MAX_PULL = 9'h02; parameter SPEED_MAX_PUSH = 9'h80;
    parameter SPEED_UP = 9'h1F; parameter SPEED_DN = 9'hFF;
    wire clk_PWM, clk_uart;
    assign {clk_PWM, clk_uart} = {pwm_clk, clk};

    reg [8:0] speed0, speed1, speed2, speed3;
    PWM_timer mot0_PWM(clk_PWM, speed0, en_PWM, sig_to_mot[0], dirp[0],
dirn[0]);
    PWM_timer mot1_PWM(clk_PWM, speed1, en_PWM, sig_to_mot[1], dirp[1],
dirn[1]);
    PWM_timer mot2_PWM(clk_PWM, speed2, en_PWM, sig_to_mot[2], dirp[2],
dirn[2]);
    PWM_timer mot3_PWM(clk_PWM, speed3, en_PWM, sig_to_mot[3], dirp[3],
dirn[3]);

    wire rst, rx_flag, rx_busy, tx_busy, rx_error;
    reg [7:0] tx_reg; wire [7:0] rx_reg;
    reg tx_en;
    uart COM(clk_uart, rst, rx, tx, tx_en, tx_reg, rx_flag, rx_reg,
rx_busy, tx_busy, rx_error);

    assign {rx_flag_out, rx_reg_out}={rx_flag,rx_reg};

    always @(negedge rx_flag) begin

        case (rx_reg)
            "f": begin
                speed0<=SPEED_MAX_PULL; speed1<=SPEED_MAX_PULL;
                speed2<=SPEED_MAX_PUSH; speed3<=SPEED_MAX_PUSH;
            end
            "b": begin
                speed0<=SPEED_MAX_PUSH; speed1<=SPEED_MAX_PUSH;
                speed2<=SPEED_MAX_PULL; speed3<=SPEED_MAX_PULL;
            end
            "r": begin
                speed0<=SPEED_MAX_PUSH; speed1<=SPEED_MAX_PULL;
                speed2<=SPEED_MAX_PULL; speed3<=SPEED_MAX_PUSH;
            end
            "l": begin
                speed0<=SPEED_MAX_PULL; speed1<=SPEED_MAX_PUSH;
                speed2<=SPEED_MAX_PUSH; speed3<=SPEED_MAX_PULL;
            end
            "u": begin
                speed0<=SPEED_MAX_PULL; speed1<=SPEED_MAX_PULL;
                speed2<=SPEED_MAX_PULL; speed3<=SPEED_MAX_PULL;
            end
            "d": begin
                speed0<=SPEED_DN; speed1<=SPEED_DN;
                speed2<=SPEED_DN; speed3<=SPEED_DN;
            end
            "s": begin
                speed0<=0; speed1<=0;
                speed2<=0; speed3<=0;
            end
        endcase

    end


endmodule
```

Verilog Coding on FPGA : Advanced Version –

It is fully modular and properly commented.

```
//clk is the input clock for running pipe
//sign = 1 means operands are signed and the output should also be signed
//oprnd1 and oprnd2 are inputs. They are latched at the beginning of calculations to
avoid dubious results.
//A positive edge on en is going to trigger a multiplication
//product is output. It is not driven through a flipflop and will contain intermediate
values until multiplication is over
//comp_sig=0 means the multiplier is busy. This can be used to know when the
multiplication is over.
module multiplier(input clk, input sign, input [31:0] oprnd1, oprnd2, input en, output
[63:0] product, output comp_sig);

        reg [63:0] product_temp; //internal calculations of the pipe

        reg [31:0] oprnd1_copy; //Latching oprnd1
        reg [63:0] oprnd2_copy; //Latching oprnd2

        //Finds the sign of output
        wire negative_output; assign negative_output = ( sign && ((oprnd1[31] &&
!oprnd2[31]) | | (!oprnd1[31] && oprnd2[31])) );
```

```
	reg [5:0] bit_consumed; //Stores the number of unprocessed bits

	assign comp_sig = &(~bit_consumed); //When all bits are processed, comp_sig
signals that multiplication is over

	//initial bit_consumed <= 0;

	assign product = ( (negative_output) ? ~product_temp + 1'b1 : product_temp );
	reg en_hist; //Useful for detecting positive edges of multiplication
	always @(negedge clk) begin
		en_hist<=en;
		if (en && (~en_hist)) begin
			//Initialisation
			bit_consumed <= 6'd32;
			product_temp <= 0;
			//Two's Complements if the oprnds are negative and sign bit is 1.
			//Also, extending one of the inputs to 64 bits for future use.
			oprnd2_copy <= ( (!sign || !oprnd2[31]) ? { 32'd0, oprnd2 } : {
32'd0, ~oprnd2 + 1'b1} );
			oprnd1_copy <= ( (!sign || !oprnd1[31]) ? oprnd1 : ~oprnd1 + 1'b1
);
		end else begin
			if (bit_consumed > 0) begin
				//Repeated addition
				if( oprnd1_copy[0] == 1'b1 ) product_temp <=
product_temp + oprnd2_copy;

				oprnd1_copy <= oprnd1_copy >> 1;
				oprnd2_copy <= oprnd2_copy << 1;
				bit_consumed <= bit_consumed - 1'b1;

		  end
		end
  end
endmodule

//clk is the input clock for running pipe
//sign = 1 means operands are signed and the output should also be signed
//dividend and divisor are inputs. They are latched at the beginning of calculations to
avoid dubious results.
//A positive edge on en is going to trigger a division
//quotient and remainder are outputs. They are not driven through a flipflop and will
contain intermediate values until division is over
//comp_sig=0 means the module is busy. This can be used to know when the division is
```

```
over.
module division(input clk, input sign, input [63:0] dividend, input [31:0] divisor, input en,
output [63:0] quotient, output [31:0] remainder, output comp_sig);

        reg [63:0] quotient_temp; //Internal calculations of the pipe
        reg [95:0] divisor_copy; wire [95:0] diff;
        reg [95:0] dividend_copy;
        wire negative_output;
        assign negative_output = ( sign && ((divisor[31] && !dividend[63]) ||
(!divisor[31] && dividend[63])) );

        assign remainder = (!negative_output) ? dividend_copy[31:0] :
~dividend_copy[31:0] + 1'b1;

        reg [6:0] bit_consumed; //Stores the number of unprocessed bits
        assign comp_sig = &(~bit_consumed); //When all bits are processed, comp_sig
signals that multiplication is over

        initial bit_consumed <= 0;

        assign quotient = (!negative_output) ? quotient_temp : ~quotient_temp + 1'b1;
        assign diff = dividend_copy - divisor_copy; //Used in line 87 for repeated
subtraction

        reg en_hist;
        always @(negedge clk) begin
                en_hist<=en;
                if (en && (~en_hist)) begin
                        //Initialisation
                        bit_consumed <= 7'd64;
                        quotient_temp <= 0;
                        //Latching the inputs dependent on the sign and whether sign bit
is 1 or 0.
                        //Also the length of operands are changed for future convenience.
                        dividend_copy <= (!sign || !dividend[63]) ? {32'd0,dividend} :
{32'd0,~dividend + 1'b1};
                        divisor_copy <= (!sign || !divisor[31]) ? {1'b0,divisor,63'd0} :
{1'b0,~divisor + 1'b1,63'd0};
                end else begin
                        if (bit_consumed > 0) begin

                                quotient_temp <= quotient_temp << 1;

                                if(!diff[95]) begin
```

```
                                        dividend_copy <= diff; //Repeated subtraction (see
line 71)
                                        quotient_temp[0] <= 1'd1;
                                end

                                divisor_copy <= divisor_copy >> 1;
                                bit_consumed <= bit_consumed - 1'b1;

                        end
                end
        end

endmodule

//Implements the Newton-Raphson method for square root
//clk is input clock to facilitate pipes
//oprnd is input and rootoprnd is output. Input is latched but output is not driven by a
flipflop. Output will contain dubious values as long as the calculations are going on.
//A positive edge on en is going to trigger a start
//comp_sig = 0 means the module is busy. This can be used to know when the
calculations are over
module squareroot(input clk, input [63:0] oprnd, output [31:0] rootoprnd, input en,
output reg comp_sig);

        reg [31:0] present_guess; initial present_guess<=0; //Current guess of the square
root
        assign rootoprnd = present_guess;
        reg count; //State variable. It will be used later
        reg [63:0] oprnd_reg; //Variable for latching inputs

        //Calculating next_guess
        wire [31:0] next_guess; wire [63:0] correction;
        assign next_guess = (present_guess + correction[31:0])>>1 ; //1/2*(x_n+V/x_n)
std way to calculate square root
        //Format for division is division(clk, sign, dividend, divider, quotient, remainder,
comp_sig)
        wire [31:0] remainder; wire div_comp; reg en_div; initial en_div<=0;
        division
corr_compute(clk,0,oprnd_reg,present_guess,en_div,correction,remainder,div_comp);

        reg en_hist;
        reg div_start; initial div_start<=0; //This is just for adding a delay in changing
state
        always @(negedge clk) begin
```

```
		en_hist<=en;
		if (en && (~en_hist)) begin
			//Initialization
			//present_guess<={oprnd[63],31'h0}; //Try to extract only the
even elements of the array for faster convergence.
			comp_sig<=0;
			count<=0;
			en_div<=1;
			oprnd_reg<=oprnd;
			div_start<=0;
		end else begin
			case (count)
				0: begin
					div_start<=1;
					if (div_comp && div_start) begin
						count<=count+1;
						comp_sig<=
(next_guess[31:1]==present_guess[31:1]); //Checking if the Newton-Raphson has
converged
						en_div<=0;
					end
				end
				1: begin
					//Starting the next division if Newton-Raphson has
not converged
					div_start<=0;
					present_guess<=next_guess;
					if (!comp_sig) begin
						en_div<=1;
						count<=count+1;
					end
				end
			endcase
		end
	end

endmodule

//This module takes in the coordinates of a motor and the object. It calculates the
distance between them. The distance will be used as length of rope.
//All coordinates are assumed to be positive
//clk, en and comp_sig are as in multiplication
//coor_mot_i is the ith component of the motor location.
//Restrict coor_obj_i to be 31 bits and NOT 32 bits. This is to remove the overflow
```

```
condition.
module len_calc(input clk, input en, input [31:0] coor_mot_x, coor_mot_y, coor_mot_z,
input [31:0] coor_obj_x, coor_obj_y, coor_obj_z, output [31:0] len, output reg
comp_sig);

        reg count; //State variable used later
        //Defining the enable pins and completion flags
        wire comp_x, comp_y, comp_z, sq_comp; reg en_mult, en_sq;
        initial begin
                en_mult<=0;
                en_sq<=0;
        end

        wire comp_mult = comp_x & comp_y & comp_z; //Completion flag of all
multiplications
        reg en_hist; //Used for detecting positive edges of en
        reg [2:0] clk_factor; initial clk_factor <= 0;
        reg clk_div; initial clk_div<=0; //This is a slowed down clock. It is required for
synchronization with squareroot
        always @(negedge clk) begin
                clk_factor<=clk_factor+1;
                if (clk_factor==0) clk_div<=~clk_div;
        end

        always @(negedge clk_div) begin
                en_hist<=en;
                if (en && (~en_hist)) begin
                        //Initialization
                        en_mult<=1;
                        count<=0;
                        comp_sig<=0;
                end else begin
                        case (count)
                                0: begin
                                        //Wait for the multiplication to get over.
                                        if (comp_mult) begin
                                                count<=count+1;
                                                en_sq<=1; //Trigger the squareroot
calculation
                                                en_mult<=0;
                                        end
                                end
                                1: begin
                                        //Wait for the squareroot calculation to get over.
                                        if (sq_comp) begin
                                                comp_sig<=1;
                                                en_sq<=0;
                                        end
                                end
                        endcase
                end
        end

        //Instantiating multiplier and squareroot modules
        wire [63:0] len_squared;
        wire [31:0] coor_diff_x; assign coor_diff_x = coor_obj_x - coor_mot_x;
        wire [31:0] coor_diff_y; assign coor_diff_y = coor_obj_y - coor_mot_y;
        wire [31:0] coor_diff_z; assign coor_diff_z = coor_obj_z - coor_mot_z;
        wire [63:0] coor_diff_x_squared; multiplier
multx(clk,1,coor_diff_x,coor_diff_x,en_mult,coor_diff_x_squared,comp_x);
        wire [63:0] coor_diff_y_squared; multiplier
multy(clk,1,coor_diff_y,coor_diff_y,en_mult,coor_diff_y_squared,comp_y);
        wire [63:0] coor_diff_z_squared; multiplier
multz(clk,1,coor_diff_z,coor_diff_z,en_mult,coor_diff_z_squared,comp_z);

        assign len_squared = coor_diff_x_squared + coor_diff_y_squared +
coor_diff_z_squared;

        squareroot sqlen(clk,len_squared,len,en_sq,sq_comp);

endmodule

//This is a PID controller. Currently we are using it only as P but it can be customized to
be used as PID
//speed is a signed variable. It is driven through a flipflop and hence never contains
intermediate values
//Here comp_sig represents zero error. This can be used as a flag to know when the
output has stabilised.
//clk and en are as in multiplication.
//des_rot_inp gives the magnitude or the two's complement of the magnitude which is
the desired output. Care should be taken as it is not latched.
//dir=1 means desired output is negative
//pulses is the input coming from the encoder. The module will count the pulses and
attempt to make it equal to des_rot_inp.
module controller(input clk, input en, input [31:0] des_rot_inp, input dir, input pulses,
output reg [4:0] speed, output comp_sig);

        initial speed<=0;
```

```
	wire [31:0] des_rot_mag; assign des_rot_mag = des_rot_inp;
	wire [31:0] des_rot; assign des_rot = (dir ? ~des_rot_mag + 1 : des_rot_mag);
//Finding the two's complement based on dir

	reg [31:0] count_pulse; initial count_pulse<=0;
	reg pulses_hist; reg en_hist;
	//assign count_pulse_out=count_pulse[8:0];

	//Debouncing by reducing the clock period
	reg clk_debounced; initial clk_debounced<=0;
	reg [15:0] clk_counter; initial clk_counter<=0;
	always @(negedge clk_debounced) begin
		clk_counter<=clk_counter+1;
		if (clk_counter==0) clk_debounced<=~clk_debounced;
	end

	//Calculating error and pulse count
	wire [31:0] error; assign error=des_rot-count_pulse;
	wire error_sign;
	assign error_sign = error[31];
	always @(negedge clk_debounced) begin
		en_hist<=en; pulses_hist<=pulses;
		if (en && (~en_hist)) count_pulse<=0; //At neg->pos transition reset
count_pulse.
		else if (pulses_hist && (~pulses))
count_pulse<=count_pulse+{{31{error_sign}},1'b1};
	end

	assign comp_sig = ( (error[31:1]==31'h0) | (error[31:1]==31'h7FFFFFFF) );
	//PID Constants
	wire [31:0] Kp; wire [31:0] Kd; wire [31:0] Ki; wire mult_comp;
	assign Kp=32'h1F222222; assign Kd=32'h0;

	reg mult_comp_reg; initial mult_comp_reg<=0; always @(negedge clk)
mult_comp_reg<=mult_comp;//Introducing a delay
	//Derivative
	reg [31:0] reg_error;
	always @(posedge mult_comp_reg) begin
		reg_error<=error;
	end
	wire [31:0] der_error; assign der_error=error-reg_error;

	//integral
	/*reg [31:0] accum;
	always @(posedge mult_comp) begin
		accum <= ( en ? accum+error : 0 ); //Resetting the value of accum once
the control is over.
	end*/

	wire [63:0] speed_exact_Kp, speed_exact_Kd, speed_exact;
	//Presently working with only proportional
	reg en_mult; initial en_mult<=0;
	wire mult_comp_Kp;
	//Generates a pulse whenever multipliers are ready
	assign mult_comp = mult_comp_Kp;
	always @(negedge clk) en_mult <= (en_mult ? ~mult_comp : 1);

	multiplier mult_Kp(clk,1,Kp,error,en_mult,speed_exact_Kp,mult_comp_Kp);
//Kp*error
	//multiplier
mult_Kd(clk,1,Kd,der_error,en_mult,speed_exact_Kd,mult_comp_Kd); //Kd*der_error

	//The following is PD
/*	wire [63:0] speed_sum; assign speed_sum = speed_exact_Kp + speed_exact_Kd;
	wire speed_overshoot; assign speed_overshoot = ( (
({speed_sum[63],speed_exact_Kp[63],speed_exact_Kd[63]}!=3'b111) &
({speed_sum[63],speed_exact_Kp[63],speed_exact_Kd[63]}!=3'b000) ) |
((speed_sum[63:33]!=31'h0) & (speed_sum[63:33]!=31'h7FFFFFFF)) );
	always @(posedge mult_comp_reg) speed[4:0]<=( en ?
		( speed_overshoot ? {speed_sum[63],{3{~speed_sum[63]}},1'b1} :
{speed_sum[63],speed_sum[32:29]} )
		: 5'b0 );*/

	//The following is only P
	wire [63:0] speed_sum; assign speed_sum = speed_exact_Kp;
	wire speed_overshoot; assign speed_overshoot = ((speed_sum[63:33]!=31'h0) &
(speed_sum[63:33]!=31'h7FFFFFFF));
	always @(posedge mult_comp_Kp) speed[4:0]<=( en ?
		( speed_overshoot ? {speed_sum[63],{3{~speed_sum[63]}},1'b1} :
{speed_sum[63],speed_sum[32:29]} )
		: 5'b0 );


endmodule

//Here en is different than multiplication. When en=0, outputs are reset to zero. When
en=1, normal PWM signal.
//clk defines the time period of the PWM signal.
```

```
//speed is signed. So if speed[4]=1, the actual speed is negative with value being two's
complement
//For the sake of positive and negative speeds, two outputs of dirp and dirn are
provided. dirp=dirn means that the motor is off. dirp=1 and dirn=0 means positive
rotation and vice versa.
//out is the PWM signal dependent only on the magnitude of the speed
//speed is latched to avoid dubious PWM output.
module PWM_timer(input clk, input [4:0] speed, input en, output out, output dirp, dirn);

        reg [4:0] speed_copy; initial speed_copy<=0;

        reg outlogic; initial outlogic<=0;//This is the output of the logic circuit. It has to
be multiplexed with 0 depending on en
        assign out = (en ? outlogic : 0); //If en is off, out=0

        reg [3:0] count; initial count<=0;//Keeping a count of the cycles elapsed

        always @(negedge count[3]) speed_copy <= (speed[4] ? ~speed[3:0]+1 :
speed[3:0]); //Latching speed

        always @(negedge clk) begin
                count<=count+1;
                if (count==0 && speed_copy[3:0]!=0) begin
                        outlogic<=1;
                end else if (count==speed_copy[3:0]) begin
                        outlogic<=0;
                end
        end

        //Direction of rotation depending on the sign of speed and whether speed is 0 or
not.
        assign dirp = ( (speed==0) ? 1'b1 : (~speed_copy[4]) );
        assign dirn = ( (speed==0) ? 1'b1 : (speed_copy[4]) );

endmodule

//This module is used for testing the connection of four motors
//des_roti are taken to be 3 bit for input convenience. The actual rotation values are
des_roti*8.
//dir[i] gives the direction of rotation of motor_i
//pulses[i] is the input from encoder values of motor_i
//PWM=0 stops all the motors
//sig_to_mot, dirp, dirn provides the PWM signals to motors
//Rest of the signals are only for debugging purposes and will be mentioned in the code
```

```
module fourmot_testing (input clk, input en, input [2:0] des_rot0, des_rot1, des_rot2,
des_rot3, input [3:0] dir, input [3:0] pulses, input PWM, output [3:0] sig_to_mot, dirp,
dirn, output [8:0] count_pulse, output reg [1:0] pulse_state, input pulse_change, output
reg [3:0] en_controller, output reg state);

        //Button pulse_change will increase pulse_state by 1
        initial pulse_state<=0;
        wire [8:0] pulse01, pulse23, count_pulse_NE, count_pulse_NW, count_pulse_SE,
count_pulse_SW;
        //Implementing a 4x1 mux
        assign count_pulse = (pulse_state[1] ? pulse23 : pulse01);
        assign pulse01 = (pulse_state[0] ? count_pulse_NW : count_pulse_NE);
        assign pulse23 = (pulse_state[0] ? count_pulse_SW : count_pulse_SE);
        always @(posedge pulse_change) pulse_state<=pulse_state+1;


        /*reg [3:0] en_controller;*/ initial en_controller<=0;
        reg [3:0] en_timer; initial en_timer<=0;
        wire clk_calc, clk_PWM; assign clk_calc = clk; assign clk_PWM = clk;
        wire [4:0] speed0, speed1, speed2, speed3; wire [3:0] comp_cont;

        wire [31:0] des_rot0_inp, des_rot1_inp, des_rot2_inp, des_rot3_inp;
        assign des_rot0_inp = {24'h0,des_rot0,3'h0};
        assign des_rot1_inp = {24'h0,des_rot1,3'h0};
        assign des_rot2_inp = {24'h0,des_rot2,3'h0};
        assign des_rot3_inp = {24'h0,des_rot3,3'h0};
        //controller gives the speed of the rope given the measured and desired lengths.
        controller mot_control_mod_NE(clk_calc, en_controller[0], des_rot0_inp, dir[0],
pulses[0], speed0, comp_cont[0], count_pulse_NE);
        controller mot_control_mod_NW(clk_calc, en_controller[1], des_rot1_inp,
dir[1], pulses[1], speed1, comp_cont[1], count_pulse_NW);
        controller mot_control_mod_SE(clk_calc, en_controller[2], des_rot2_inp, dir[2],
pulses[2], speed2, comp_cont[2], count_pulse_SE);
        controller mot_control_mod_SW(clk_calc, en_controller[3], des_rot3_inp, dir[3],
pulses[3], speed3, comp_cont[3], count_pulse_SW);

        //PWM_timer converts the speed variable into PWM signal to be sent to motors
        PWM_timer mot_timer_mod_NE(clk_PWM, speed0, (en_timer[0] & PWM),
sig_to_mot[0], dirp[0], dirn[0]);
        PWM_timer mot_timer_mod_NW(clk_PWM, speed1, (en_timer[1] & PWM),
sig_to_mot[1], dirp[1], dirn[1]);
        PWM_timer mot_timer_mod_SE(clk_PWM, speed2, (en_timer[2] & PWM),
sig_to_mot[2], dirp[2], dirn[2]);
        PWM_timer mot_timer_mod_SW(clk_PWM, speed3, (en_timer[3] & PWM),
sig_to_mot[3], dirp[3], dirn[3]);

        //reg state;
        initial state<=0;
        wire [3:0] len_diff = dir;
        reg state_en; initial state_en<=0;
        reg en_hist; initial en_hist<=0;
        always @(negedge clk) begin
                en_hist<=en;
                case (state)
                        0: begin
                                en_controller<=4'h0; en_timer<=4'h0;
                                if (en && ~en_hist) begin
                                        state<=1;
                                end
                        end
                        1: begin
                                en_controller<=4'hF; en_timer<=4'hF;
                                if (&comp_cont | ~en) state<=0;
                        end
                endcase
        end

endmodule

//This is the main moduel which is used for motion
module motion (input clk, input en, input [3:0] pulses, input [31:0] obj_x, obj_y, obj_z,
input PWM_en, output [3:0] sig_to_mot, dirp, dirn);

        reg [3:0] en_controller; reg [3:0] en_len; reg [3:0] en_timer;
        initial begin en_controller<=0; en_len<=0; en_timer<=0; end

        //These variables can be used if different clocks for different modules is desired
        wire clk_calc, clk_PWM, clk_len; assign clk_calc = clk; assign clk_PWM = clk;
assign clk_len = clk;
        wire [4:0] speed0, speed1, speed2, speed3; wire [3:0] comp_cont;
        wire [31:0] len0, len1, len2, len3; wire [3:0] comp_len;

        //Storing the previous values of lengths
        reg [31:0] len_hist0, len_hist1, len_hist2, len_hist3;
        initial begin
                len_hist0=32'd259; len_hist1<=32'd346; len_hist2<=32'd346;
len_hist3<=32'd259;
        end
        //assign len0_out = len0[9:6]; assign len1_out = len1[9:6];
        //assign len2_out = len2[9:6]; assign len3_out = len3[9:6];

        //Finding the amount of rotation of each motor based on previous lengths and
the new desired lengths
        wire [31:0] des_rot0_sign, des_rot1_sign, des_rot2_sign, des_rot3_sign;
        assign des_rot0_sign = len0 - len_hist0; assign des_rot1_sign = len1 - len_hist1;
        assign des_rot2_sign = len2 - len_hist2; assign des_rot3_sign = len3 - len_hist3;
        wire [3:0] dir;
        assign dir = {des_rot0_sign[31], des_rot1_sign[31], des_rot2_sign[31],
des_rot3_sign[31]};

        wire [31:0] des_rot0, des_rot1, des_rot2, des_rot3;
        assign des_rot0 = (des_rot0_sign[31] ? ~des_rot0_sign + 1 : des_rot0_sign);
        assign des_rot1 = (des_rot1_sign[31] ? ~des_rot1_sign + 1 : des_rot1_sign);
        assign des_rot2 = (des_rot2_sign[31] ? ~des_rot2_sign + 1 : des_rot2_sign);
        assign des_rot3 = (des_rot3_sign[31] ? ~des_rot3_sign + 1 : des_rot3_sign);

        //The coordinates of motor. They are calculated based on the experimental
setup
        wire [31:0] mot_NE_x, mot_NE_y, mot_NE_z;        wire [31:0] mot_NW_x,
mot_NW_y, mot_NW_z;
        wire [31:0] mot_SW_x, mot_SW_y, mot_SW_z;        wire [31:0] mot_SE_x,
mot_SE_y, mot_SE_z;
        assign {mot_NE_x,mot_NE_y,mot_NE_z} = {32'h0,32'h3D4,32'h1AC};
        assign {mot_NW_x,mot_NW_y,mot_NW_z} = {32'h25D,32'h3D4,32'h1A9};
        assign {mot_SW_x,mot_SW_y,mot_SW_z} = {32'h244,32'h0,32'h18E};
        assign {mot_SE_x,mot_SE_y,mot_SE_z} = {32'h0,32'h0,32'h190};

        //Instantiating the modules which will calculate the lengths of each rope
        len_calc mot_len_NE(clk_len, en_len[0], mot_NE_x, mot_NE_y, mot_NE_z,
obj_x, obj_y, obj_z, len0, comp_len[0]);
        len_calc mot_len_NW(clk_len, en_len[1], mot_NW_x, mot_NW_y, mot_NW_z,
obj_x, obj_y, obj_z, len1, comp_len[1]);
        len_calc mot_len_SW(clk_len, en_len[2], mot_SW_x, mot_SW_y, mot_SW_z,
obj_x, obj_y, obj_z, len2, comp_len[2]);
        len_calc mot_len_SE(clk_len, en_len[3], mot_SE_x, mot_SE_y, mot_SE_z, obj_x,
obj_y, obj_z, len3, comp_len[3]);

        //Proportional controller
        controller mot_control_mod_NE(clk_calc, en_controller[0], des_rot0, dir[0],
pulses[0], speed0, comp_cont[0]);
        controller mot_control_mod_NW(clk_calc, en_controller[1], des_rot1, dir[1],
pulses[1], speed1, comp_cont[1]);
	controller mot_control_mod_SW(clk_calc, en_controller[2], des_rot2, dir[2],
pulses[2], speed2, comp_cont[2]);
	controller mot_control_mod_SE(clk_calc, en_controller[3], des_rot3, dir[3],
pulses[3], speed3, comp_cont[3]);

	//PWM_timer converts the speed variable into PWM signal to be sent to motors
	PWM_timer mot_timer_mod_NE(clk_PWM, speed0, (en_timer[0] & PWM_en),
sig_to_mot[0], dirp[0], dirn[0]);
	PWM_timer mot_timer_mod_NW(clk_PWM, speed1, (en_timer[1] & PWM_en),
sig_to_mot[1], dirp[1], dirn[1]);
	PWM_timer mot_timer_mod_SW(clk_PWM, speed2, (en_timer[2] & PWM_en),
sig_to_mot[2], dirp[2], dirn[2]);
	PWM_timer mot_timer_mod_SE(clk_PWM, speed3, (en_timer[3] & PWM_en),
sig_to_mot[3], dirp[3], dirn[3]);

	reg [1:0] state;
	initial state<=0;
	wire [3:0] len_diff = dir;
	reg state_en; initial state_en<=0;
	reg en_hist; initial en_hist<=1;
	reg [1:0] clk_fac0; initial clk_fac0<=0;
	reg clk_div0; initial clk_div0<=0;

	always @(negedge clk) begin
		clk_fac0<=clk_fac0+1;
		if (clk_fac0==0) clk_div0<=~clk_div0;
	end

	//FSM to order the events
	always @(negedge clk_div0) begin
		en_hist<=en;
		if (en) begin
			case (state)
				0: begin
					en_controller<=0; en_timer<=0;
					if (en & ~en_hist) begin//At positive edge,
initializing
						en_len<=4'hF;
						state_en<=1;
					end else en_len<=4'h0;
					if (state_en==1 & ~|comp_len) state<=1; //Waiting
for the length calculations to begin
				end
```

```
                                        1: begin
                                                  if (&comp_len) begin //At the completion of length
calculations
                                                            en_controller<=4'hF; en_timer<=4'hF;
en_len<=4'h0; //Start controller and PWM output
                                                            state_en<=0;
                                                  end
                                                  if (state_en==0 & ~|comp_cont) state<=2;
//Waiting for controller to begin
                                        end
                                        2: begin
                                                  if (&comp_cont) begin //At the end of controller
                                                            state<=0;
                                                            len_hist0<=len0; len_hist1<=len1;
len_hist2<=len2; len_hist3<=len3;
                                                  end
                                        end
                              endcase
                    end else begin
                              en_controller<=4'h0; en_len<=4'h0; en_timer<=4'h0; state<=0;
state_en<=0; //Ground state of the system
                    end
          end

endmodule

//This is the module which was used in final demonstration
module main (input clk, input rx, input [3:0] pulses, output tx, input PWM_en, output
[3:0] sig_to_mot, dirp, dirn);

          uart comm(clk, rst, rx, tx, tx_en, tx_reg, rx_flag, rx_reg, rx_busy, tx_busy,
rx_error);

endmodule

module uart(
   input clk, // The master clock for this module
   input rst, // Synchronous reset.
   input rx, // Incoming serial line
   output tx, // Outgoing serial line
   input transmit, // Signal to transmit
   input [7:0] tx_byte, // Byte to transmit
   output received, // Indicated that a byte has been received.
   output [7:0] rx_byte, // Byte received
```

```
    output is_receiving, // Low when receive line is idle.
    output is_transmitting, // Low when transmit line is idle.
    output recv_error // Indicates error in receiving packet.
    );

parameter CLOCK_DIVIDE = 1302; // clock rate (50Mhz) / (baud rate (9600) * 4)

// States for the receiving state machine.
// These are just constants, not parameters to override.
parameter RX_IDLE = 0;
parameter RX_CHECK_START = 1;
parameter RX_READ_BITS = 2;
parameter RX_CHECK_STOP = 3;
parameter RX_DELAY_RESTART = 4;
parameter RX_ERROR = 5;
parameter RX_RECEIVED = 6;

// States for the transmitting state machine.
// Constants - do not override.
parameter TX_IDLE = 0;
parameter TX_SENDING = 1;
parameter TX_DELAY_RESTART = 2;

reg [10:0] rx_clk_divider = CLOCK_DIVIDE;
reg [10:0] tx_clk_divider = CLOCK_DIVIDE;

reg [2:0] recv_state = RX_IDLE;
reg [5:0] rx_countdown;
reg [3:0] rx_bits_remaining;
reg [7:0] rx_data;

reg tx_out = 1'b1;
reg [1:0] tx_state = TX_IDLE;
reg [5:0] tx_countdown;
reg [3:0] tx_bits_remaining;
reg [7:0] tx_data;

assign received = recv_state == RX_RECEIVED;
assign recv_error = recv_state == RX_ERROR;
assign is_receiving = recv_state != RX_IDLE;
assign rx_byte = rx_data;

assign tx = tx_out;
assign is_transmitting = tx_state != TX_IDLE;
```

```
always @(posedge clk) begin
        if (rst) begin
                recv_state = RX_IDLE;
                tx_state = TX_IDLE;
        end

        // The clk_divider counter counts down from
        // the CLOCK_DIVIDE constant. Whenever it
        // reaches 0, 1/16 of the bit period has elapsed.
 // Countdown timers for the receiving and transmitting
        // state machines are decremented.
        rx_clk_divider = rx_clk_divider - 1;
        if (!rx_clk_divider) begin
                rx_clk_divider = CLOCK_DIVIDE;
                rx_countdown = rx_countdown - 1;
        end
        tx_clk_divider = tx_clk_divider - 1;
        if (!tx_clk_divider) begin
                tx_clk_divider = CLOCK_DIVIDE;
                tx_countdown = tx_countdown - 1;
        end

        // Receive state machine
        case (recv_state)
                RX_IDLE: begin
                        // A low pulse on the receive line indicates the
                        // start of data.
                        if (!rx) begin
                                // Wait half the period - should resume in the
                                // middle of this first pulse.
                                rx_clk_divider = CLOCK_DIVIDE;
                                rx_countdown = 2;
                                recv_state = RX_CHECK_START;
                        end
                end
                RX_CHECK_START: begin
                        if (!rx_countdown) begin
                                // Check the pulse is still there
                                if (!rx) begin
                                        // Pulse still there - good
                                        // Wait the bit period to resume half-way
                                        // through the first bit.
                                        rx_countdown = 4;
```

```
                    rx_bits_remaining = 8;
                    recv_state = RX_READ_BITS;
                end else begin
                    // Pulse lasted less than half the period -
                    // not a valid transmission.
                    recv_state = RX_ERROR;
                end
            end
        end
        RX_READ_BITS: begin
            if (!rx_countdown) begin
                // Should be half-way through a bit pulse here.
                // Read this bit in, wait for the next if we
                // have more to get.
                rx_data = {rx, rx_data[7:1]};
                rx_countdown = 4;
                rx_bits_remaining = rx_bits_remaining - 1;
                recv_state = rx_bits_remaining ? RX_READ_BITS :
RX_CHECK_STOP;
            end
        end
        RX_CHECK_STOP: begin
            if (!rx_countdown) begin
                // Should resume half-way through the stop bit
                // This should be high - if not, reject the
                // transmission and signal an error.
                recv_state = rx ? RX_RECEIVED : RX_ERROR;
            end
        end
        RX_DELAY_RESTART: begin
            // Waits a set number of cycles before accepting
            // another transmission.
            recv_state = rx_countdown ? RX_DELAY_RESTART : RX_IDLE;
        end
        RX_ERROR: begin
            // There was an error receiving.
            // Raises the recv_error flag for one clock
            // cycle while in this state and then waits
            // 2 bit periods before accepting another
            // transmission.
            rx_countdown = 8;
            recv_state = RX_DELAY_RESTART;
        end
        RX_RECEIVED: begin
```

```
                    // Successfully received a byte.
                    // Raises the received flag for one clock
                    // cycle while in this state.
                    recv_state = RX_IDLE;
                end
        endcase

        // Transmit state machine
        case (tx_state)
                TX_IDLE: begin
                        if (transmit) begin
                                // If the transmit flag is raised in the idle
                                // state, start transmitting the current content
                                // of the tx_byte input.
                                tx_data = tx_byte;
                                // Send the initial, low pulse of 1 bit period
                                // to signal the start, followed by the data
                                tx_clk_divider = CLOCK_DIVIDE;
                                tx_countdown = 4;
                                tx_out = 0;
                                tx_bits_remaining = 8;
                                tx_state = TX_SENDING;
                        end
                end
                TX_SENDING: begin
                        if (!tx_countdown) begin
                                if (tx_bits_remaining) begin
                                        tx_bits_remaining = tx_bits_remaining - 1;
                                        tx_out = tx_data[0];
                                        tx_data = {1'b0, tx_data[7:1]};
                                        tx_countdown = 4;
                                        tx_state = TX_SENDING;
                                end else begin
                                        // Set delay to send out 1 stop bit.
                                        tx_out = 1;
                                        tx_countdown = 4;
                                        tx_state = TX_DELAY_RESTART;
                                end
                        end
                end
                TX_DELAY_RESTART: begin
                        // Wait until tx_countdown reaches the end before
                        // we send another transmission. This covers the
                        // "stop bit" delay.
```

```
                              tx_state = tx_countdown ? TX_DELAY_RESTART : TX_IDLE;
                    end
          endcase
end
endmodule

dm
```

ii. C coding on Optimus

```
#include<avr/io.h>
#include<avr/interrupt.h>
#include<stdint.h>
#include<util/delay.h>
#define sbi(x,y) x |= y

#define USART_BAUDRATE 76800
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)

void USART_Init( void );
void USART_Transmit( unsigned char data );
unsigned char USART_Receive( void );
void USART_word_transmit(char*);

void USART_Init( void )
{
          //sbi(UCSR1A,_BV(U2X1));     /* Set baud rate */
          //UBRR1H = 0 ;
          //UBRR1L = 12;

          UCSR1A = 0;
          UCSR1B = (1<<RXEN1) | (1<<TXEN1);
          UCSR1C = (1<<UCSZ10) | (1<<UCSZ11);

          UBRR1L = BAUD_PRESCALE; // Load lower 8-bits of the baud rate value into the
low byte of the UBRR register
   UBRR1H = (BAUD_PRESCALE >> 8); // Load upper 8-bits of the baud rate value into the
high byte of the UBRR register

          /* Enable receiver and transmitter */
          UCSR1A &= ~(1 << U2X1);                                        //*

}
```

```
void USART_Transmit( unsigned char data )
{
	/* Wait for empty transmit buffer */
	while ( !( UCSR1A & (1<<UDRE1)) );

	/* Put data into buffer, sends the data */
	UDR1 = data;

}

unsigned char USART_Receive( void )
{
	/* Wait for data to be received */
	while ( !(UCSR1A & (1<<RXC1)) );

	/* Get and return received data from buffer */
	return UDR1;
}


void USART_word_transmit(char *ptr){
	for (inti=0; i<20; i++){
		char j = ptr[i];
		USART_Transmit(j);
	}
}



int main( void )
{
	USART_Init();
	DDRB = 0xFF;
	PORTB = 0x00;
	DDRF = 0xFF;
	PORTF = 0x00;

	unsigned char receivedData = USART_Receive();

	while(1)
	{

		unsigned char receivedData = USART_Receive();
		//USART_Transmit('b');
		if (receivedData == 'a'){
			char str[] = "20* celcius        ";
			USART_word_transmit(str);
			/*
			USART_Transmit('2');
			USART_Transmit('0');
			USART_Transmit('*');
			USART_Transmit(' ');
			USART_Transmit('c');
			*/

			PORTB = ~PORTB;
		}

		if (receivedData == 'b'){
			char str[] = "21* celcius        ";
			USART_word_transmit(str);
			PORTB = ~PORTB;
		}

		if (receivedData == 'c'){
			char str[] = "22* celcius        ";
			PORTB = ~PORTB;
		}

		if (receivedData == 'd'){
			USART_Transmit('d');
			PORTB = ~PORTB;
		}

		//_delay_ms(1);

		/*
		if (receivedData == 'a')USART_Transmit('20* cel');
		if (receivedData == 'b')USART_Transmit('21* cel');
		if (receivedData == 'c')USART_Transmit('22* cel');
		if (receivedData == 'd')USART_Transmit('23* cel');
		*/

	}

	return 0;
}
```

b. Web Server on Intel atom :

In Intel atom board of Terasic development board web server were created. For creating web server following things were done:

1. Installation of Ubuntu 10.04 operating system was done.
2. Then after setting the internet settings for Ubuntu apache2 web server and php5 package was downloaded from repository according to steps in following website https://help.ubuntu.com/community/ApacheMySQLPHP
3. Then all the html and php files are kept into /var/www which is the default directory for the web server.

Important files in web server at /var/www location were:

1. Index.php

   It consist the following code:

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="style.css" />

<!--function for IP camera pan tilt control-->
<script type="text/javascript">

var user="usr1";
var pwd="test";

function decoder_control_2(command)
{

action_zone.location='http://greenhouse.cse.iitb.ac.in:8090/decoder_co
ntrol.cgi?command='+command+'&usr='+user+'&pwd='+pwd;
}
</script>


</head>
<body>

<!--navigation bar-->
<div id="navbar">
    <div id="holder">
<ul>
<li><a href="#" id="onlink">Home</a></li>
<li><a href="auto2.php">Auto</a></li>
<li><a
href="..\google_surface_plot\temp_prof.php">TempProfile</a></li>

</ul>

</div>
</div>

<!-main wrapper which consists all div tags-->
```

```
<div id="wrapper">
<center><h1>Spyder GUI control</h1></center>
<!--camera image-->
<div id="camera">
<img
src="http://greenhouse.cse.iitb.ac.in:8090/videostream.cgi?user=user1&
pwd=test" width="700" height="450" id="pic">
</div>

<div id="map">
<canvas id="myCanvas" width="289" height="100" ></canvas>
    <script>
      var canvas = document.getElementById('myCanvas');
      var context = canvas.getContext('2d');
      var centerX = canvas.width / 2;
      var centerY = canvas.height / 2;
      var radius = 10;
      var imageObj = new Image();

      context.beginPath();
      context.arc(centerX, centerY, radius, 0, 2 * Math.PI, false);
      context.fillStyle = 'red';
      context.fill();
      context.lineWidth = 5;
      context.strokeStyle = '#003300';
      context.stroke();
    </script>
</div>


<!--buttons for platform motor control-->
<div id="controls">
<form method="post" action="<?php echo $_SERVER['PHP_SELF'];?>">

<div id="left">
<button type="submit"  value="Left" name="rcmd" style="border: 0;
background: transparent" title="left">
    <img src="button_left.png" width="90" height="90" alt="submit" />
</button>
</div>

<div id="right">
<button type="submit"  value="Right" name="rcmd" style="border: 0;
background: transparent" title="right">
    <img src="button_right.png" width="90" height="90" alt="submit" />
</button>
</div>

<div id="stop">
<button type="submit"  value="Stop" name="rcmd" style="border: 0;
background: transparent" title="stop">
    <img src="button_stop.png" width="90" height="90" alt="submit" />
</button>
</div>

<div id="forward">
<button type="submit"  value="Forward" name="rcmd" style="border: 0;
```

```
background: transparent" title="forward">
    <img src="button_forward.png" width="90" height="90" alt="submit"
/>
</button>
</div>

<div id="backward">
<button type="submit"  value="Backward" name="rcmd" style="border: 0;
background: transparent" title="backward">
    <img src="button_backward.png" width="90" height="90" alt="submit"
/>
</button>

</div>
<div id="up">
<input type="submit" value="Up" name="rcmd" title="up">
</div>

</form>

</div>

<!--buttons for camera pan tilt control-->
<div id="camera_control">
<iframe name="action_zone" style="DISPLAY: none" width="0" height="0">
</iframe>

<button type="button" onTouchStart="decoder_control_2(0)"
onTouchEnd="decoder_control_2(1)" onMouseDown="decoder_control_2(0)"
onMouseUp="decoder_control_2(1)" style="border: 0; background:
transparent" title="backward">
<img src="button_forward.png" width="90" height="90" alt="submit" />
</button>
<button type="button" onTouchStart="decoder_control_2(2)"
onTouchEnd="decoder_control_2(3)" onMouseDown="decoder_control_2(2)"
onMouseUp="decoder_control_2(3)"style="border: 0; background:
transparent" title="backward">
<img src="button_backward.png" width="90" height="90" alt="submit" />
</button>
<button type="button" onTouchStart="decoder_control_2(4)"
onTouchEnd="decoder_control_2(5)" onMouseDown="decoder_control_2(4)"
onMouseUp="decoder_control_2(5)"style="border: 0; background:
transparent" title="backward">
<img src="button_right.png" width="90" height="90" alt="submit" />
</button>
<button type="button" onTouchStart="decoder_control_2(6)"
onTouchEnd="decoder_control_2(7)" onMouseDown="decoder_control_2(6)"
onMouseUp="decoder_control_2(7)"style="border: 0; background:
transparent" title="backward">
<img src="button_left.png" width="90" height="90" alt="submit" />
</button>

</div>

</div>

<!--php code for serial communication and sending commands on button
```

```
press
<?php
$verz="1.0";
$comPort = "/dev/ttyACM1"; /*change to correct com port */

if (isset($_POST["rcmd"])) {
$rcmd = $_POST["rcmd"];
switch ($rcmd) {
     case 'Stop':
        echo "Stop";
       $fp =fopen($comPort, "w");
  fwrite($fp, 's'); /* this is the char that it will write */
  fclose($fp);
  break;
     case 'Forward':
        echo "Forward";
        $fp =fopen($comPort, "w");
  fwrite($fp, 'f'); /* this is the char that it will write */
  fclose($fp);
  break;
  case 'Backward':
        echo "Backward";
        $fp =fopen($comPort, "w");
  fwrite($fp, 'b'); /* this is the char that it will write */
  fclose($fp);
  break;
  case 'Up':
        echo "up";
        $fp =fopen($comPort, "w");
  fwrite($fp, 'u'); /* this is the char that it will write */
  fclose($fp);
  break;
case 'Right':
        echo "right";
        $fp =fopen($comPort, "w");
  fwrite($fp, 'r'); /* this is the char that it will write */
  fclose($fp);
  break;
case 'Left':
        echo "left";
        $fp =fopen($comPort, "w");
  fwrite($fp, 'l'); /* this is the char that it will write */
  fclose($fp);
  break;
default:
  die('Select the correct options');
}

}
?>
</body>
</html>
```

2. **Style .css**

   It provides style to the above webpage where each parts are created using ‘div’ tag.Code:

```
body {
background:url(naturewallpapers252872529.jpg);
}
#wrapper{
width:1180px;
background:rgba(0,153,0,0.5);

margin:25px 30px auto;
-webkit-border-bottom-left-radius:15px;
-webkit-border-bottom-right-radius:15px;
-webkit-border-top-right-radius:15px;
height:650px;


}
#navbar{
    width:660px;


}
#navbar #holder {
height:10px;
border-bottom:1px solid #0000;
width:630px;
padding-left:30px;
 }
#navbar #holder ul {
    list-style:none;
    margin:0;
    padding:0;



}

#navbar #holder ul li a{
text-decoration:none;
float:left;
margin-right:5px;
font-family:"Arial Black";
color:#000;
border:1px solid #000;
border-bottom:none;
padding:5px;
width:120px;
text-align:center;
display:block;
background:#69F;
-webkit-border-top-left-radius:15px;
-webkit-border-top-right-radius:15px;

}

#navbar #holder ul li  a:hover {
background: #F90;
```

```
color:#FFF;



}

#holder ul li a#onlink{
background:#FFF;
color:#000;
border-bottom:1px solid #FFF;

}
#hover ul li a#onlink:hover{
background:#FFF;
color:#449;
text-shadow:1px 1px 1px #000;
}

#camera{
postion:absolute;
top:10px;
left:50px;
}
#surfacePlotDiv{
width:500px;
height:500px
postion:absolute;
top:10px;
left:50px;
}

canvas {
 background:url(green_house2.jpg) ;
 }
 #map{
 position:absolute;
 top:100px;
 left:800px;
 }
 #map2{
 position:absolute;
 top:100px;
 left:800px;
 }
#controls {
background-color:#777;
width:200px;
height:200px;

margin:0 auto;
position:absolute;
top:350px;
left:850px;
-webkit-border-top-left-radius:15px;
-webkit-border-top-right-radius:15px;
-webkit-border-bottom-left-radius:15px;
```

```
-webkit-border-bottom-right-radius:15px;
}

#forward{
position:absolute;
top:-20px;
left:50px;
}

#backward{
position:absolute;
top:120px;
left:50px;

}

#left{
position:absolute;
top:50px;
left:-25px;
}
#right{
position:absolute;
top:50px;
left:120px;
}

#stop{
position:absolute;
top:50px;
left:50px;
}
```

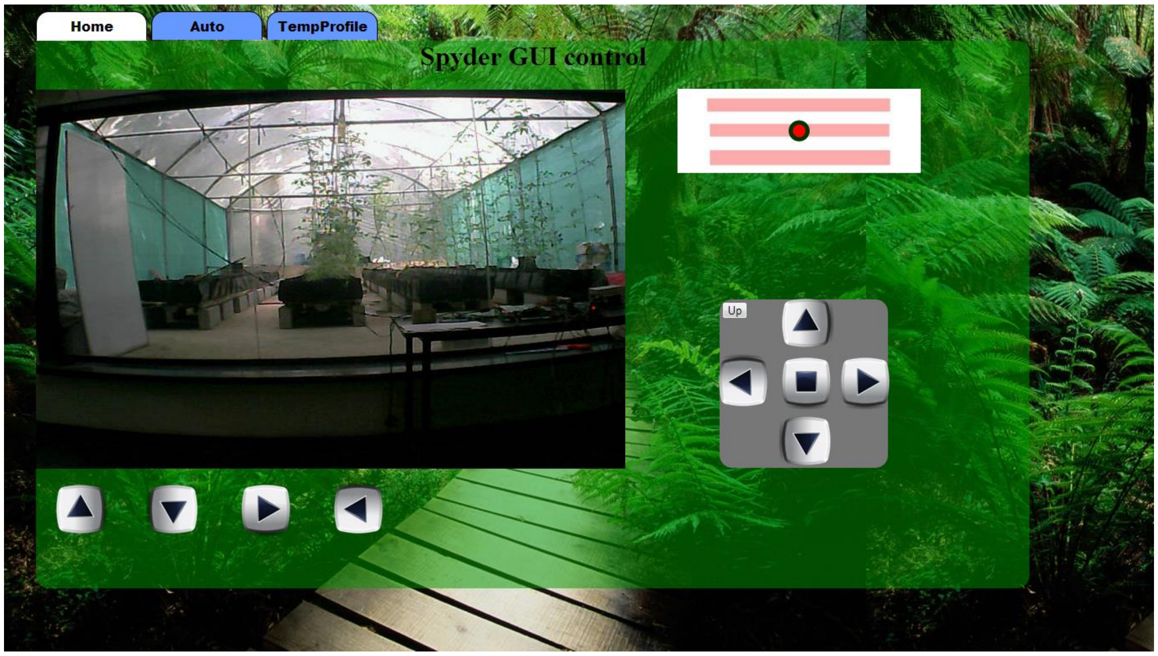


Fig. GUI for Spyder bot control

c. **Port forwarding setting in router**

Now for making server accessible throughout IITB intranet following port forwarding done in the router.

Router address 192.168.1.1

IP camera - 192.168.1.150:8090

Intel atom webserver-192.168.1.100:8095

Web server usrername- erts-1
Password - password

8. Challenges faced :

- Development of the required hardware
  - Four motor platforms located at four corners of room
  - Put a pulley system to fix encoder
- Learning the new development platform
  - Communication between Intel atom part and FPGA part
  - Developing the user interface to control the bot remotely
- Controlling the precise location of the bot due to non uniform speed of motors
- Developing the control algorithm for the precise control for the length of the cables
  - Independent control of ropes were note possible during motion always one of the rope become slack
- Developing the power supply for the IP camera module
- Because of dense wiring and big circuits in the electrical design, the debugging was challenging.
- Initially, working with Yocto project to use peripherals (WiFi) was a little difficult.
- Communication between FPGA and Atom was challenging.

9. Insights gained :

- Learnt a lot about our development board capability
- Learnt to create the remote web server for control applications
- Learnt to develop and use the Quartus feature for development and synthesis of FPGA hardware
  - Intellectual Property blocks(IP blocks) or Mega-function blocks

10. Future Scope :

- Develop the waypoint control for the bot
- Solve the problem of communication between FPGA and Intel atom board
- Develop the api for getting the sensor data