Portland State
UNIVERSITY

ECE 544

PORTLAND STATE UNIVERSITY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ESD Design with FPGAs

PROJECT 2 REPORT

Prasanna Kulkarni
prasanna@pdx.edu

Atharva Mahindrakar
atharva2@pdx.edu

Date: May 20, 2019

Contents

List of Figures

1 Overview

The project is based on a Nexys DDR4 board, we are using the Microblaze soft-core IP as the core of our embedded system and the software code is written in C. The project uses many of the peripheral modules provided in the release folder. The basis of the prohect is implementing a Closed loop control system using PID control. The working can be seen by applying load to the shaft of the motor using a finger and the control system will try to maintain the speed.

2 Functional Specification :

The board takes the value of the rorary switch position from the PMOD Encoder. This value is used as the speed parameter for the motor. In terms of the PID control system we are using this point as the setpoint. The the PMOD Oled screen is used to display the values of the K_i, K_p, and K_d values.
The HB3 PMOD is being used as a driver to the motor. There is a direction input which can be used to change the rotation direction of the motor. The sa_input pin on the the driver gives out pulses based on the rotation of the motor. These pulses are detected in the hardware and are used to calculate the speed of the motor (rpm).
The board uses the Microblaze soft core IP using C. The function that drives the motor by generating the Pulse Width Modulation (PWM) and the speed calculation that is mentioned above are implemented in the hardware using a custom IP, PMOD_HB3

3 Work Distribution :

The majority of the work was broadly divided in the following manner.

- **Atharva :** Software
- **Prasanna :** Hardware
- The testing, debugging of the design as well as the graph calculation were done by both

4 Hardware :

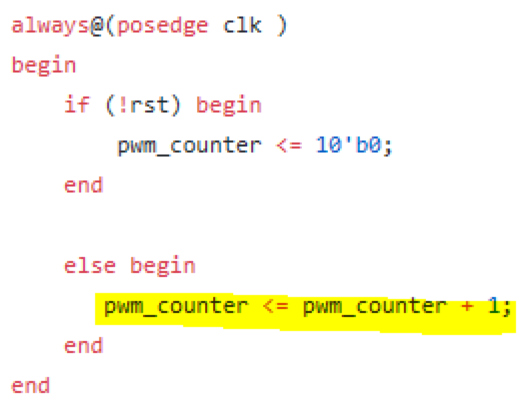
The biggest difference from previous work with similar environments and this project was that the part selection was done by using the board files that were provided. Doing this made some parts of the project easier, but figuring out how the constraints file and the wrapper worked together took a little bit of time.
There are three clocks generated which are 100Mhz, 50Mhz and 200Mhz. The 100Mhz clock is the system clock which is used to drive most of the system. The PMOD Oled screen works at a rate of 50Mhz but, a long part of time was spent on getting it to work properly and then a post on the discussion forum post mention trying to up the clock to a 100Mhz. We have used this ”hotfix” in the system. The 50

Mhz pin is left in the block design. The 200Mhz clock is used for the MIG 7 DDR2 SDRAM. There are two AXI Timers being used. One of them is used for the FreeRtos and another AXI Timer is being used as a clock source for the Nexys4IO IP block.
A big chunk of the development time was spent in deciding how to approach the PMOD_HB3 custom IP generation. The first approach was to use a AXI GPIO and a AXI Timer in order to save some time writing the HDL code for the custom IP. This was one of a couple of methods mentioned in the documentation. This method unfortunately was found not to work as expected. Thus, we decided to write the HDL code after all. Fortunately the rpm detection module was quite similar to what we had done in project 1.

4.1 PMOD_HB3 :

The PMOD_HB3 is made up of two major sections. These are the two major files that are used in making the IP. The `pwm_gen.v` is the hdl code that generates the PWM signal that is used to drive that motor. The `rpm_detect.v` is the file that detects the speed of the motor by using a simple edge detection algorithm on the sa_input pin.

4.1.1 `pwm_gen.v` :

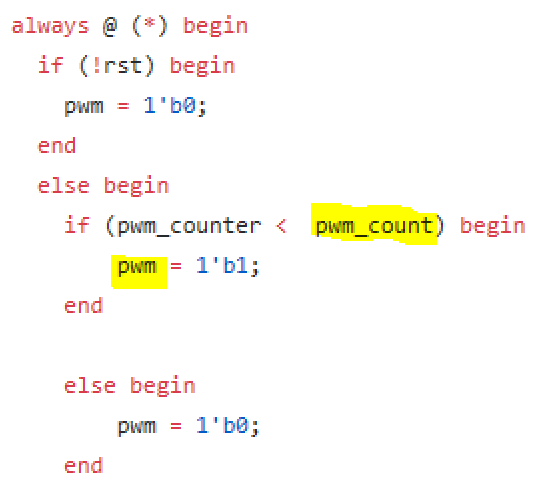


```
always@(posedge clk )
begin
    if (!rst) begin
        pwm_counter <= 10'b0;
    end

    else begin
       pwm_counter <= pwm_counter + 1;
    end
end
```

Figure 1: `pwm_gen.v` counter snippet

The basic idea with the `pwm_gen` code is to have a counters. This counter is the `pwm_counter`. In order to improve the resolution of the PID control we have scaled up the PWM to a 11 bit value. This means that it will range from 0-2047. The counter simply increments the count on every clock cycle unless we encounter a reset.

The idea here is quite simple, the signal `pwm` goes high if the count is below a certain value and remains low other wise. A point of interest here is the `pwm_count`. This variable is user given the requested pwm of the system. In order to get this variable into the system we are using a register inside the PMOD HB3 driver. The way we



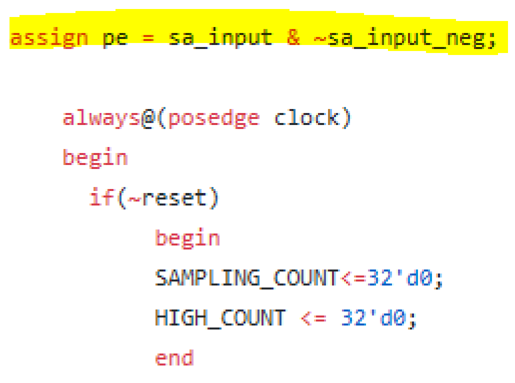
```
always @ (*) begin
  if (!rst) begin
    pwm = 1'b0;
  end
  else begin
    if (pwm_counter <  pwm_count) begin
        pwm = 1'b1;
    end

    else begin
        pwm = 1'b0;
    end
```

Figure 2: `pwm_gen.v` pwm snippet

have done this is by editing the hdl code inside the AXI Bus instantiation of the peripheral. Something that was extremely useful in doing this was the RojoBot project from ECE540.

4.1.2 `rpm_detect.v` :

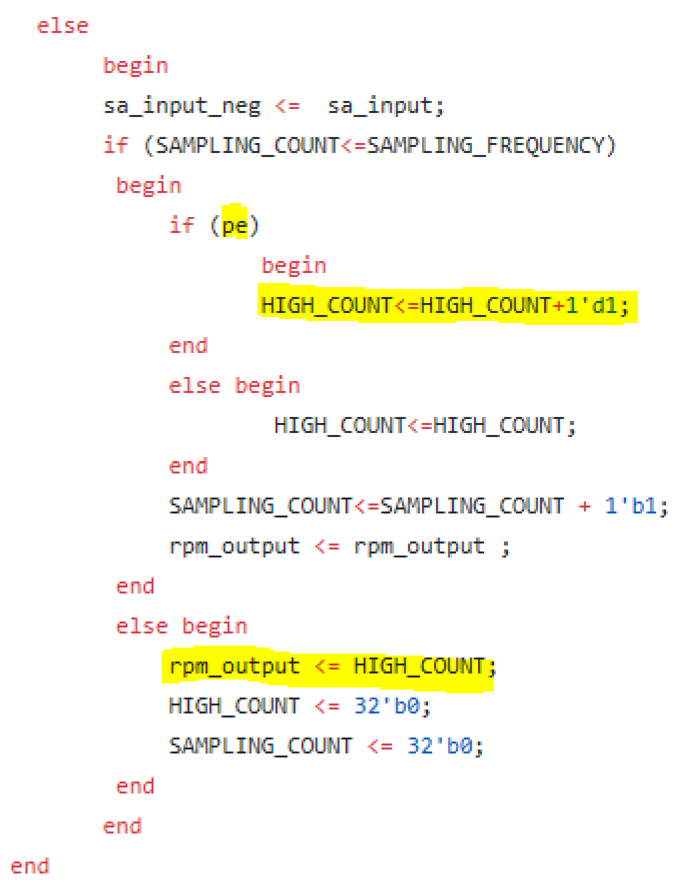
This is the more complex of the two hdl codes written for the IP. But, it helps that the concept is very similar to what we have already done in the first assignment fro pwm detection. What we have done is, compare the signal with a other "known" signal which generated by the `SAMPLING FREQUENCY` signal.



```
assign pe = sa_input & ~sa_input_neg;

    always@(posedge clock)
    begin
      if(~reset)
           begin
           SAMPLING_COUNT<=32'd0;
           HIGH_COUNT <= 32'd0;
           end
```

Figure 3: `rpm_detect.v` pulse edge snippet

The important part here is the `pe` signal which we have used for detection of an edge. Thus, we are measuring the number of edges in a "known" time and by doing that we can calculate the speed of the motor.



```
  else
       begin
       sa_input_neg <=  sa_input;
       if (SAMPLING_COUNT<=SAMPLING_FREQUENCY)
        begin
            if (pe)
                   begin
                   HIGH_COUNT<=HIGH_COUNT+1'd1;
            end
            else begin
                    HIGH_COUNT<=HIGH_COUNT;
            end
            SAMPLING_COUNT<=SAMPLING_COUNT + 1'b1;
            rpm_output <= rpm_output ;
        end
        else begin
            rpm_output <= HIGH_COUNT;
            HIGH_COUNT <= 32'b0;
            SAMPLING_COUNT <= 32'b0;
        end
       end
end
```

Figure 4: `rpm_detect.v` high count snippet

The other important signal in the hdl code is the `HIGH COUNT`. This variables stores the rpm output that we are displaying. To do this we have also made the rpm output as a register for our IP.

5 Software :

For the PID implementation for motor control, we have develop RTOS application using FreeRTOS platform on Xilinx Nexys4 board. For this application we created 3 threads with 1 helper function for PID algorithm.

5.1 Threading Model :

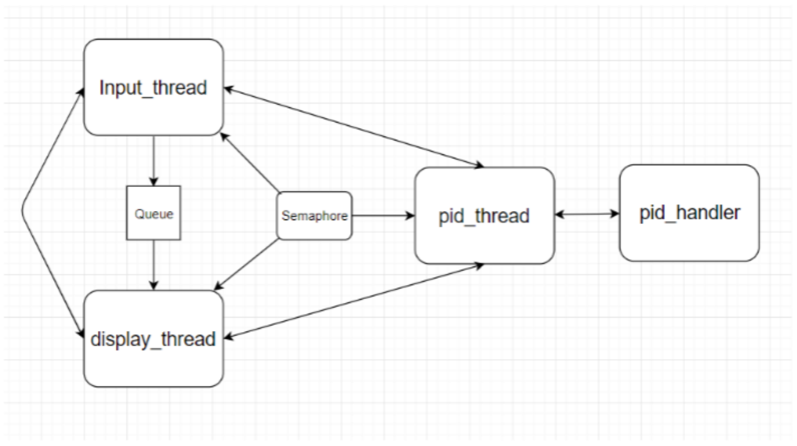


Figure 5: Threading model

For this application we created three threads with set priorities then these threads were scheduled to meet our applications requirements. Along with these threads a helper function is also created to help with PID thread. The thread generation and scheduling is achieved by using FreeRTOS library functions.
Following are the list of threads created -

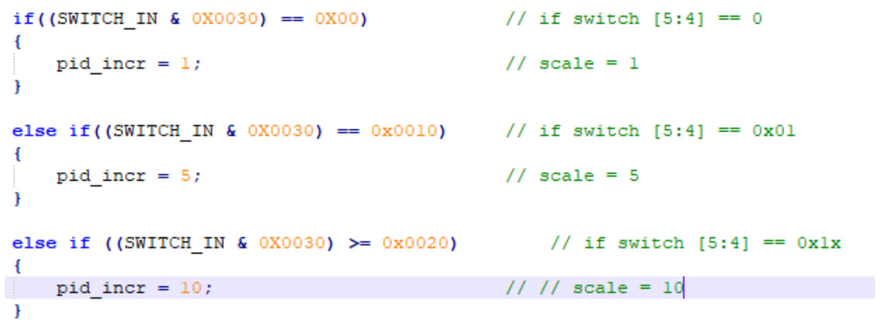
- Input Thread
- Display Thread
- PID thread

5.1.1 Input Thread :

In this thread, four different input operations are performed.

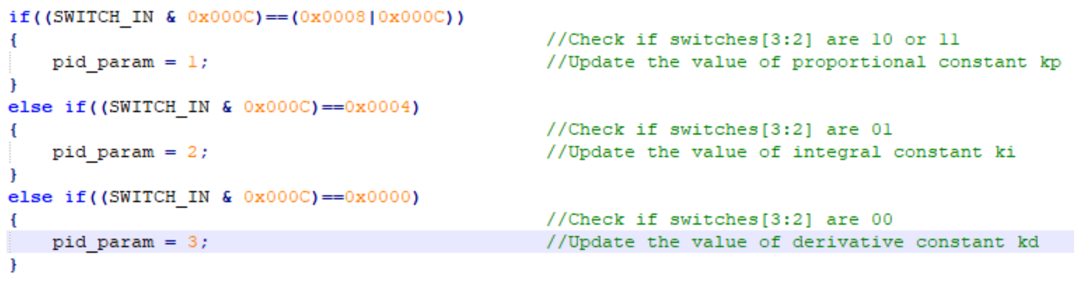
Reading Input Switches :

Depending upon the switches states, different combinations of tasks are performed. Using switches [5:4] increment/ decrement scale of the Kp, Ki, Kd parameters is decided.
Same is the case for the rotary encoders, as it depends upon switches [1:0] states for getting there increment/ decrement scale. Also switches [3:2] are used to get select one of the PID parameters of which value needs to be changed.



```
if((SWITCH_IN & 0X0030) == 0X00)            // if switch [5:4] == 0
{
    pid_incr = 1;                           // scale = 1
}

else if((SWITCH_IN & 0X0030) == 0x0010)     // if switch [5:4] == 0x01
{
    pid_incr = 5;                           // scale = 5
}

else if ((SWITCH_IN & 0X0030) >= 0x0020)        // if switch [5:4] == 0x1x
{
    pid_incr = 10;                          // // scale = 10
}
```

Figure 6: software switch input snippet



```
if((SWITCH_IN & 0x000C)==(0x0008|0x000C))
{                                               //Check if switches[3:2] are 10 or 11
    pid_param = 1;                              //Update the value of proportional constant kp
}
else if((SWITCH_IN & 0x000C)==0x0004)
{                                               //Check if switches[3:2] are 01
    pid_param = 2;                              //Update the value of integral constant ki
}
else if((SWITCH_IN & 0x000C)==0x0000)
{                                               //Check if switches[3:2] are 00
    pid_param = 3;                              //Update the value of derivative constant kd
}
```

Figure 7: software switch pid snippet

Reading Push Button inputs :

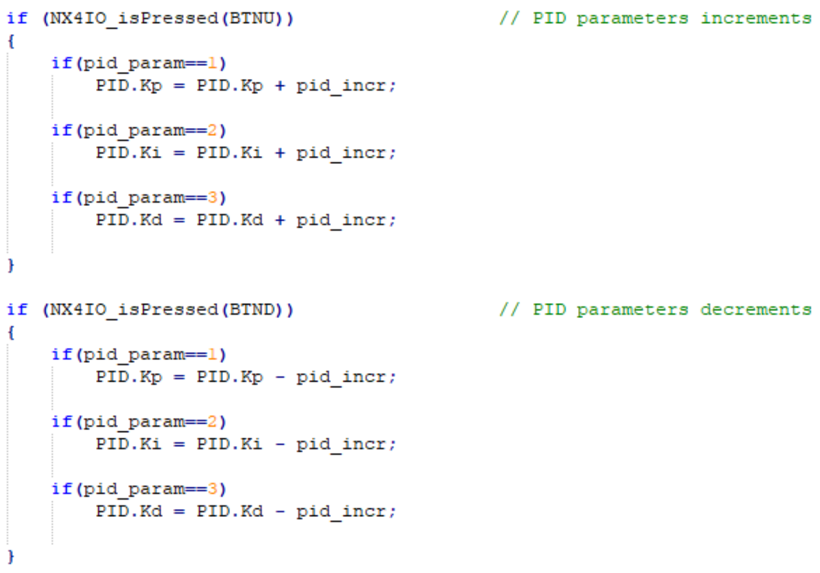
Push buttons C resets all the PID parameters as well the target RPM count. Push buttons U and D are used for increment and decrement of PID parameters with set scale by the switches as shown above.

Reading Rotary encoder data :

The rotary encoder is utilized in the same way as implemented in project 1, it is used to set target RPM for the PID application. The increment / decrement scale on each tick of the encoder depends upon the scale set by the switches [1:0].

PmodHB3 Encoder reading :

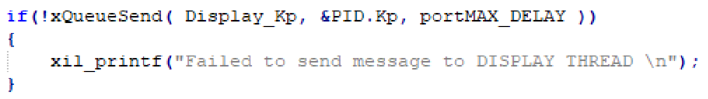
As we have developed our custom IP for PmodHB3 as described in hardware section. The Xilinx tool has generated a C driver to read / write the register using an AXI peripheral. We have assigned slv_reg1 for setting up the direction (0 or 1), slv_reg2 for giving PWM count as an input to the IP amp; slv_reg3 is used to read back the encoder output given by the motor. The function PMOD_HB3_mWriteReg is used to write data to these assigned registers over the AXI bus. For writing the motor direction, as we are using slv_reg1, we need to give an offset of 4 to the base address as shown below PMOD_HB3_mWriteReg(PMODHB3_BASEADDR, 4, 1);
As for sending PWM duty cycle count, we are using slv_reg3, hence we need to give an offset of 12 to write the PWM duty cycle count, as shown below



```
if (NX4IO_isPressed(BTNU))              // PID parameters increments
{
    if(pid_param==1)
        PID.Kp = PID.Kp + pid_incr;

    if(pid_param==2)
        PID.Ki = PID.Ki + pid_incr;

    if(pid_param==3)
        PID.Kd = PID.Kd + pid_incr;

}

if (NX4IO_isPressed(BTND))              // PID parameters decrements
{
    if(pid_param==1)
        PID.Kp = PID.Kp - pid_incr;

    if(pid_param==2)
        PID.Ki = PID.Ki - pid_incr;

    if(pid_param==3)
        PID.Kd = PID.Kd - pid_incr;

}
```

Figure 8: software change pid snippet

PMOD_HB3_mWriteReg(PMODHB3_BASEADDR, 12, tempTargetRPM); For reading the encoders output given by our custom made IP block, we need to follow following method RPM_detect = PMOD_HB3_mReadReg(PMODHB3_BASEADDR, 8); This function returns the data saved onto the sl_reg2 register.
After reading all the inputs, the PID parameters like Kp, Ki, Kd, target amp; current RPM to the struct we created at top of the code. The structure is created to keep these values accessible in any threads.
Also after reading these input data, it is sent to the display thread via Queue.



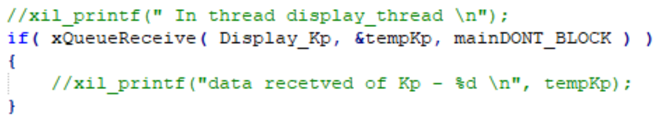
```
if(!xQueueSend( Display_Kp, &PID.Kp, portMAX_DELAY ))
{
    xil_printf("Failed to send message to DISPLAY THREAD \n");
}
```

Figure 9: send queue snippet

The same procedure is followed to send the rest of PID parameters as well detected RPM to the Display thread.

5.1.2 Display Thread :

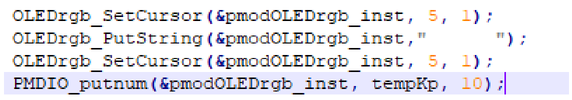
In this thread, all the PID parameters data sent from the input thread is displayed on the OLED display.



```
//xil_printf(" In thread display_thread \n");
if( xQueueReceive( Display_Kp, &tempKp, mainDONT_BLOCK ) )
{
    //xil_printf("data recetved of Kp - %d \n", tempKp);
}
```

Figure 10: receive queue snippet

In above example, the data sent by the input thread is stored in a variable name tempKp and this variable is further used in this thread to display the Kp data on the OLED screen.

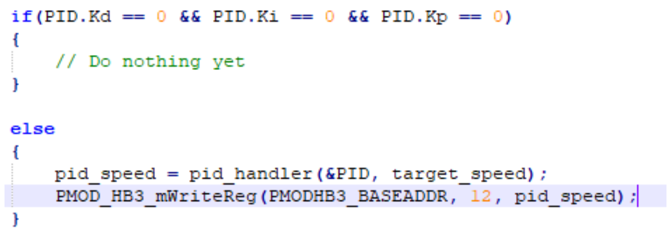


```
OLEDrgb_SetCursor(&pmodOLEDrgb_inst, 5, 1);
OLEDrgb_PutString(&pmodOLEDrgb_inst,"      ");
OLEDrgb_SetCursor(&pmodOLEDrgb_inst, 5, 1);
PMDIO_putnum(&pmodOLEDrgb_inst, tempKp, 10);
```

Figure 11: print OLED snippet

5.1.3 PID Thread :

In this thread, the PID parameters are accessed through the declared structure not by the queue messaging. At first, it is checked, if any of the PID parameters is not zero. If all the parameters are zero, then PID threads just stalls and gives up the semaphore, and if one of the PID parameter is non zero, then this threads call helper function pid_handler which is used for performing the PID calculations as well the returning the required PWM signal value required for that particular situation.



```
if(PID.Kd == 0 && PID.Ki == 0 && PID.Kp == 0)
{
    // Do nothing yet
}

else
{
    pid_speed = pid_handler(&PID, target_speed);
    PMOD_HB3_mWriteReg(PMODHB3_BASEADDR, 12, pid_speed);
}
```

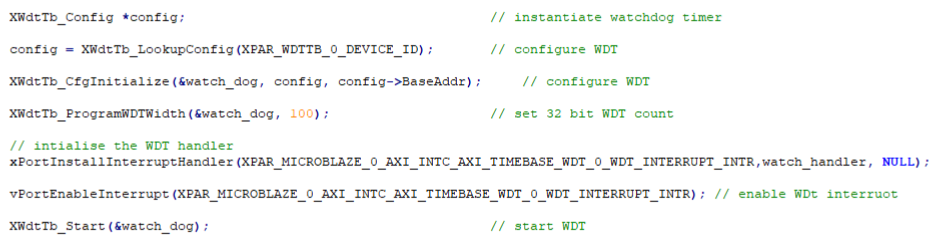
Figure 12: write register snippet

PID Handler :

To this function, the reference of the structure of the PID parameters is parsed. The steps are followed as required by the PID algorithm to calculate the required PWM duty cycle. This calculated PWM value is then returned to the thread, where it gets sent to the PMOD_HB3_mWriteReg function to rotate the motor.

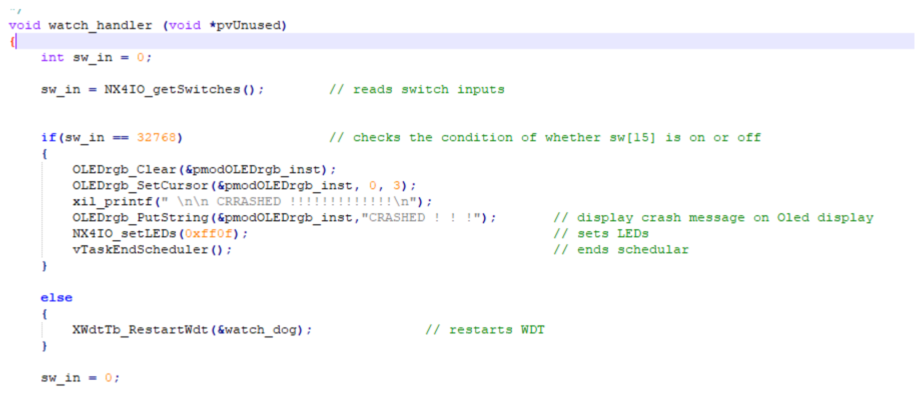
5.1.4 Watchdog Timer :

The watchdog timer is basically used to restart the embedded systems in times of sudden failure or malfunction. In this application, it is used to force the crash if the switch [15] is turned on. First the WDT is extentiated and assigned an interval at which it gets interrupts as well the interrupt handler is assigned.
Then in WDT handler, at each call the state of SW[15] is checked. If the switch is on, then program is given a forced crash otherwise WDT just gets restarted.
In this function, we are scaling down the 8 bit PID parameters from 0 255 scale to 0 1 scale and then further calculations are performed.



```
XWdtTb_Config *config;                                  // instantiate watchdog timer

config = XWdtTb_LookupConfig(XPAR_WDTTB_0_DEVICE_ID);   // configure WDT

XWdtTb_CfgInitialize(&watch_dog, config, config->BaseAddr);    // configure WDT

XWdtTb_ProgramWDTWidth(&watch_dog, 100);                // set 32 bit WDT count

// intialise the WDT handler
xPortInstallInterruptHandler(XPAR_MICROBLAZE_0_AXI_INTC_AXI_TIMEBASE_WDT_0_WDT_INTERRUPT_INTR,watch_handler, NULL);

vPortEnableInterrupt(XPAR_MICROBLAZE_0_AXI_INTC_AXI_TIMEBASE_WDT_0_WDT_INTERRUPT_INTR); // enable WDt interruot

XWdtTb_Start(&watch_dog);                               // start WDT
```

Figure 13: watchdog initiataion snippet



```
void watch_handler (void *pvUnused)
{
    int sw_in = 0;

    sw_in = NX4IO_getSwitches();        // reads switch inputs


    if(sw_in == 32768)                  // checks the condition of whether sw[15] is on or off
    {
        OLEDrgb_Clear(&pmodOLEDrgb_inst);
        OLEDrgb_SetCursor(&pmodOLEDrgb_inst, 0, 3);
        xil_printf(" \n\n CRRASHED !!!!!!!!!!!!!\n");
        OLEDrgb_PutString(&pmodOLEDrgb_inst,"CRASHED ! ! !");       // display crash message on Oled display
        NX4IO_setLEDs(0xff0f);                                      // sets LEDs
        vTaskEndScheduler();                                        // ends schedular
    }

    else
    {
        XWdtTb_RestartWdt(&watch_dog);              // restarts WDT
    }

    sw_in = 0;
```

Figure 14: watchdog handler snippet

6 Challenges Faced :

- Generating PMOD HB3 IP
- Assigning the salve registers correctly
- Getting FreeRtos to run properly with nexys4IO, this was fixed by allocating mor stack/heap in the linker script menu.

7 Results :

When PID parameters are set as shown in graph, after some time it gets stabalised to target RPM with +- 5% accuracy After motor reaches the target RPM, load is applied

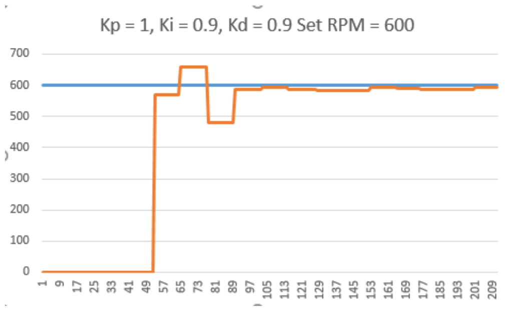


Figure 15: result 1

to the shaft, it results in sharp drop in RPM of motor, then again using PID it reaches stability after some time

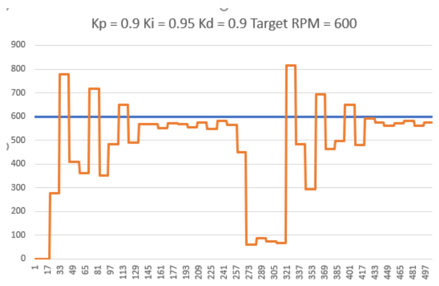


Figure 16: result 2

8 References

- Getting Started Guide provided by Prof Kravitz.
- The Project description in the Assignment release document.
- Diligent Documentation about Pmod Encoder and Pmod OLED
- Nexys4 DDR Device Manual