Self-Regulating Water Heater

Made by:
Raphael Enrico Catapang
Andreas Josef Diaz

For
ENGG 113.02
ENGG 181.31 / 181.32

In collaboration with:
Marc Jefferson Obeles
Justin Gabriel Sy

For
ENGG 113.02
ENGG 183.31/183.32

December 11, 2023

Table of Contents

Project Description

This project was inspired by Cucio et al.'s thesis proposal to create a PID-controlled water heater incubator for water quality testing. The overall goal was to create a system that allows users to change the temperature setpoint and the gain coefficients of the PID controller. This project was conducted under ENGG 113.02, ENGG 181.31/32, and ENGG 183.31/32. It consists of two parts: the FPGA side and the ESP32 side.

FPGA Goals:

- Implement a tunable PID controller to correct the error between current temperature and setpoint temperature.
- Interface a temperature sensor directly with the FPGA.
- Connect with a device capable of transmitting data to the cloud (ESP32).
    - The FPGA should be able to send and receive data simultaneously with the connected device.
- Interface the FPGA with the system's actuator, a 220 V AC resistive heater.

Hardware Used:

- Tang Nano 9k (FPGA)
- DS18B20 Temperature Sensor
- Fotek Solid State Relay
- Resistive Water Heater (220 V AC)
- NodeMCU ESP32 (x2)
- 16x2 LCD I2C
- MCP9808 Air Temperature Sensor
- Altera DE1 Board

Software Used:

- Arduino IDE
- Quartus (For using Altera DE1)
- Gowin (For using Tang Nano 9k)
- Firebase (Cloud)

System Overview:

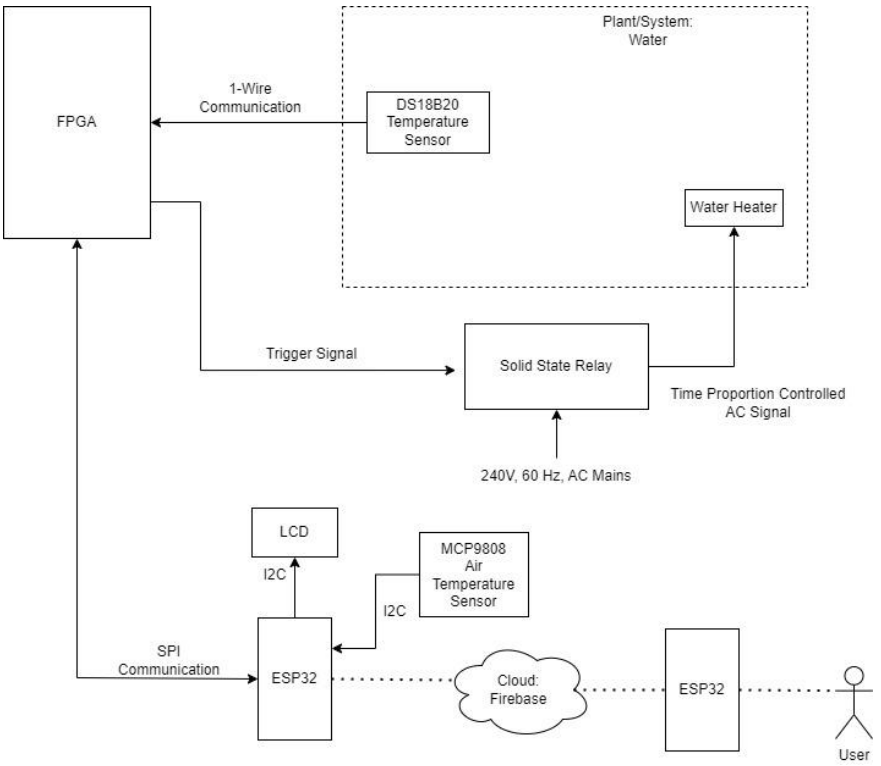

Fig. 1. Self-Regulating Water Heater Overview

Hardware Connections:

Altera DE1 Board

Data Sheet (Page 35)

3.3 Vcc = Connected to Vcc of DS18B20
GND = Connected to Ground of DS18B20, ESP32, and Relay
GPIO_1[0] = Connected to Data wire of DS18B20
GPIO_1[1] = Connected to Pin 18 of ESP32
GPIO_1[3] = Connected to Pin 19 of ESP32
GPIO_1[5] = Connected to Pin 23 of ESP32
GPIO_1[7] = Connected to Pin 5 of ESP32
GPIO_1[9] = Connected to Vcc of Relay

Tang Nano 9K

Data Sheet (Section 3.2: Pin Map)

3.3 Vcc = Connected to Vcc of DS18B20
GND = Connected to Ground of DS18B20, ESP32, and Relay
Pin 25 = Connected to Data wire of DS18B20
Pin 26 = Connected to Pin 18 of ESP32
Pin 27 = Connected to Pin 19 of ESP32
Pin 28 = Connected to Pin 23 of ESP32
Pin 29 = Connected to Pin 5 of ESP32
Pin 30 = Connected to Vcc of Relay

Results:

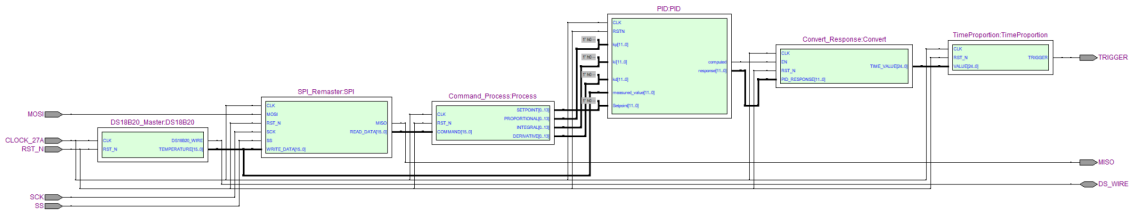

Fig. 2: Resulting Wrapper Circuit

Flow Status	Successful - Sat Dec 09 10:44:56 2023
Quartus II Version	9.0 Build 132 02/25/2009 SJ Web Edition
Revision Name	WaterHeater
Top-level Entity Name	WaterHeater
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Met timing requirements	Yes
Total logic elements	607 / 18,752 (3 %)
Total combinational functions	515 / 18,752 (3 %)
Dedicated logic registers	419 / 18,752 (2 %)
Total registers	419
Total pins	8 / 315 (3 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	6 / 52 (12 %)
Total PLLs	0 / 4 (0 %)

Fig. 3: Compilation Results in Quartus

Resource Utilization Summary

Resource	Usage	Utilization
Logic	576(494 LUT, 82 ALU) / 8640	7%
Register	385 / 6693	6%
--Register as Latch	0 / 6693	0%
--Register as FF	385 / 6693	6%
BSRAM	0 / 26	0%

Fig. 4: Compilation Results in Gowin FPGA Designer

HDL Modules (Verilog)

DS18B20 Master
File: DS18B20_Master.v
Description: This module provides an interface between the FPGA and the DS18B20 temperature sensor. It adheres to the timing requirements specified by the DS18B20 manufacturer. For this purpose, the following state machine was developed and implemented:

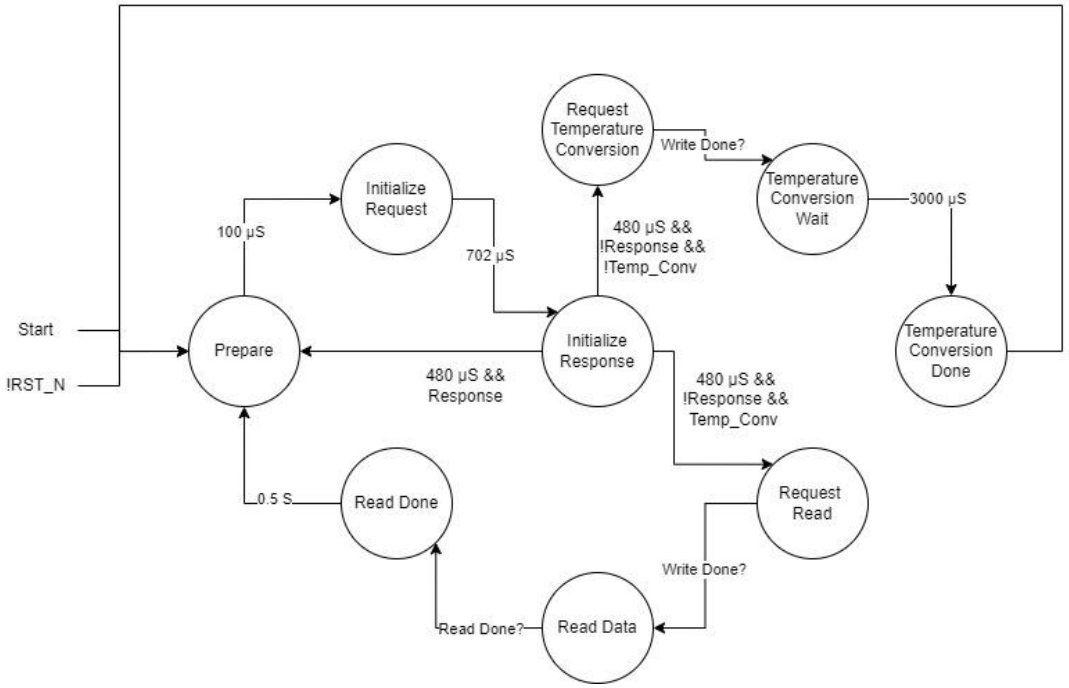

Fig. 5: Finite State Machine for DS18B20 Master Module

Reference Utilized: [DS18B20 - Programmable Resolution 1-Wire Digital Thermometer]
Inputs:

- Master Clock (CLK)
- Master Reset (RST_N)

Outputs:

- Temperature (16-Bit)

Input/Output:

- DS18B20 Data Wire

Notes:

- The temperature data provided by the DS18B20 sensor is 16 bits long, but only bits 0 through 11 hold the actual temperature information. These 12 bits represent an unsigned integer that corresponds to the temperature in degrees Celsius. See Fig. 6 for the format.
- The DS18B20 temperature sensor requires an external pull-up resistor (5 kOhm) connected to its data pin to function properly. This is because the data pin is a 3-state pin.

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
LS BYTE	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	BIT 15	**BIT 14**	**BIT 13**	**BIT 12**	**BIT 11**	**BIT 10**	**BIT 9**	**BIT 8**
MS BYTE	S	S	S	S	S	2^6	2^5	2^4

S = SIGN

Fig. 6: Temperature Data Format

Tests Done:

- Testbench is not possible due to the specific timing requirements of the DS18B20
- Debugging was done by using the module and checking the current state and the output.
    - State Testing was done by assigning each state (9 states in total) to its own LED in the Altera DE1 board.
        - State testing pinpoints where the FSM becomes stuck.
    - Output checking was done through the 7 segment displays.
        - Checked the whole number value of the resulting temperature data (Bits 4 to 10) and compared with Arduino serial monitor output.

Recommendations/Improvements:

- The current implementation employs numerous state-specific counters, each incrementing only when the system resides within its corresponding state. This approach presents inefficiencies. A more optimal solution would utilize a single counter whose maximum or minimum value dynamically adjusts based on the current state.
- While the current implementation prioritizes simplicity by solely requesting and reading temperature values, the DS18B20 datasheet offers a broader range of functionalities. Through additional commands, the master can leverage various peripherals within the DS18B20, all via the same data line. Future iterations should explore the possibility of utilizing multiple DS18B20 sensors on a single data line, unlocking greater potential and maximizing resource utilization.

SPI Slave
File: SPI.v
Description: There are four pins that are implemented for the communication between the master and slave: CLK, MISO, MOSI, and CS0. The CLK is what controls the rate of data exchange and this is sent by the master to the slave. The CS0, or the chip select, is a normally low signal that is switched to LOW by the master to indicate that data exchange will commence. In implementations where there are multiple slaves, this pin is important in selecting which slave is going to send data, or from which slave the data will be coming from. The MOSI, or the Master Output Slave Input, is the pin for sending data from the master to the slave. In this bus, the data being sent starts with the MSB and the slave turns this into one multiple-bit data. The MISO, or the Master Input Slave Output, is the pin for sending data from the slave to the master and this data transfer starts with the LSB.

Process:
1. Master sends CLK signal
2. Master sets the CS0 to LOW
3. Data can be sent from master to slave and slave to master at the same time.

Reference Utilized: [Circuit Basics - Basics of the SPI Communication]

Inputs:
- Master Clock (CLK)
- Master Output Slave Input (MOSI)
- Chip Select (CS0)
- MISOflag
- DATA_MISO

Outputs:
- Master Input Slave Output (MISO)
- DATA_MOSI
- dflag

Notes:
- MISO and MOSI are independent from one another.
- The amount of pins is not constrained to 4.
    - There are only 4 pins for master-to-slave connectivity but there are multiple pins for data transmission between the slave and the IC it is connected to since they are parallel.

Tests Done:
- **Testbench:** tb_SPI.v
- Tests two sets of data transmission in both MOSI and MISO
    - First Data sent through MOSI: 101010010101
    - Second Data sent through MOSI: 111110000111
    - First Data sent through MISO: 010011001100
    - Second Data sent through MISO: 110101101011
- Fig. 7 shows the results of the testbench.

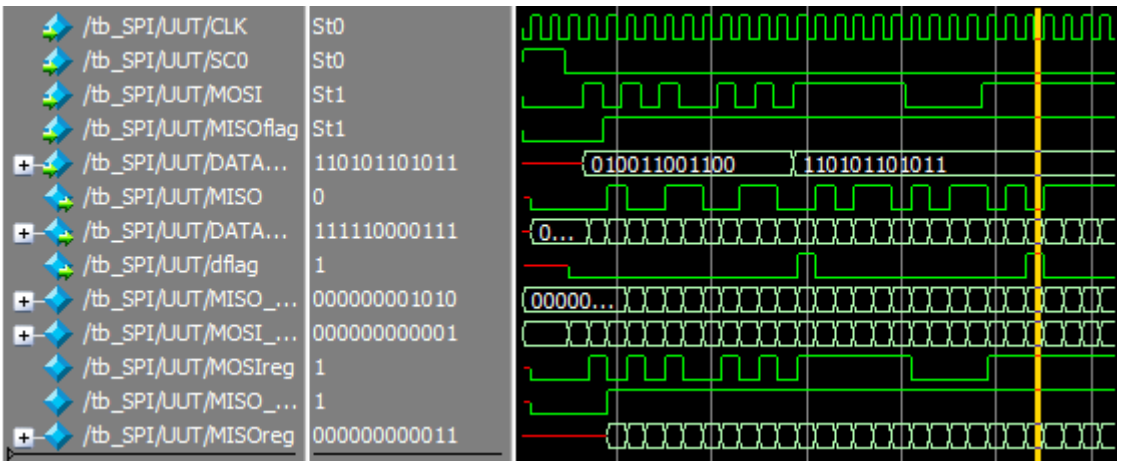

Fig. 7: Results of Testbench

SPI Slave Remaster
File: SPI_Remaster.v
Description: Improves or remasters the former SPI Slave implementation. Changes are mainly:

- Make the module synchronous (Following system or master clock).
    - Since the module is synchronous, edge detection techniques are utilized for the SPI signals to allow communication to function as expected.
- Follow the master reset.

Reference Utilized: [SPI 2 - A simple implementation]

Inputs:

- Master Clock (CLK)
- Master Reset (RST_N)
- Serial Clock (SCK)
- Master Out, Slave In (MOSI)
- Chip Select/Slave Select (SS)
- Write Data (16-Bit)

Outputs:

- Master In, Slave Out (MOSI)
- Read Data (16-Bit)

Notes:

- The serial clock should be four times less than the FPGA or system clock. In this case, it is recommended to set the SPI clock to 1 MHz in Arduino.
- Write data would always be the temperature output of the DS18B20 Master module
    - However, this SPI Slave module can be used for other purposes.
- Read data would always be the commands sent by the ESP32.
    - Again, this SPI slave module can be utilized for other purposes.
- It is assumed that 2 byte transactions are always done, hence the write and read data ports are 16 bits wide.
    - The Arduino IDE and the SPI library has the ability to send 2 bytes of data each transaction.
- The module can als do 1 byte transactions.
    - However, Write Data and Read Data is expected to be placed on bits 8 to 15.
- It is important that the connection between FPGA and ESP32 has a common ground.
    - Noisy data for MISO and MOSI will be apparent if no common ground is made.

Tests Done:

- No testbenches done, as it builds on the original SPI Slave module, which was tested.
- Tested directly with the ESP32 to check if data sent and data received are correct.
    - **Sketch utilized:** SPITest.ino
    - **Wrapper module used**: SPI_Wrapper.v
- Testing the MISO capability
    - Made Write Data a constant value (16'hdead, 16'hbeef, 16'hfafa)
    - Check the Arduino serial monitor to see if the given data in hexadecimal is the same.

- Testing the MOSI capability
    - Also made data sent by the ESP32 a constant value (16’hdead, 16’hbeef, 16’hfafa).
    - Interface Read Data port to the 7-segment display as hexadecimal.
    - Check if the Hex value on the 7-segment display is the same as the command given from ESP32.

Recommendations/Improvements:

- The module works if the serial clock is at 1 MHz.
    - Tests should be performed on higher serial clock speeds
    - Improvements or optimizations should also be made to allow faster serial clock speeds.

Command Processing
File: Command_Process.v
Description: This module receives 16-bit (2-byte) commands sent from an ESP32 to an FPGA and processes them. It analyzes the command based on the 14th and 15th bits to determine the intended action and subsequently adjusts the setpoint or PID gain coefficients accordingly.

TABLE I: List of Commands

Command[15:14]	Description of Command
2'b00	Change the Setpoint value to Command[13:0].
2'b01	Change the Proportional Gain value to Command[13:0].
2'b10	Change the Integral Gain value to Command[13:0].
2'b11	Change the Derivative Gain value to Command[13:0].

Additionally, the module enforces limitations on the acceptable input values. For setpoint commands, only values within the range of 27.5 to 100 degrees Celsius are permitted. Providing values in the Q7.7 fixed-point format outside this range will not result in any change to the setpoint. For all other commands, any value within the Q7.7 fixed-point format is accepted.

Reference Utilized: None

Inputs:
- Master Clock (CLK)
- Master Reset (RST_N)
- Command (16-Bit)

Outputs:
- Setpoint Value (14-Bit)
- Proportional Gain Value (14-Bit)
- Integral Gain Value (14-Bit)
- Derivative Gain Value (14-Bit)

Notes:
- The FPGA can perform the PID control loop independently of the ESP32 due to the availability of the setpoint and PID gain values in dedicated registers. This allows the FPGA to directly access these values without requiring communication with the ESP32.
    - If the ESP32 is removed while the FPGA is running, the last stored values would still be used
    - If the FPGA is reset, it would utilize the values hard-coded to it when reset.
- Although the outputs are provided in Q7.7 fixed-point format, the PID calculator only utilizes 11 bits, resulting in a Q7.4 fixed-point value.

Tests Done:

- **Testbench:** tb_Command_Process.v
- Tests each command and checks if the setpoint value changes if given value is within or outside the range.
- Fig. 8 shows the results of the testbench.

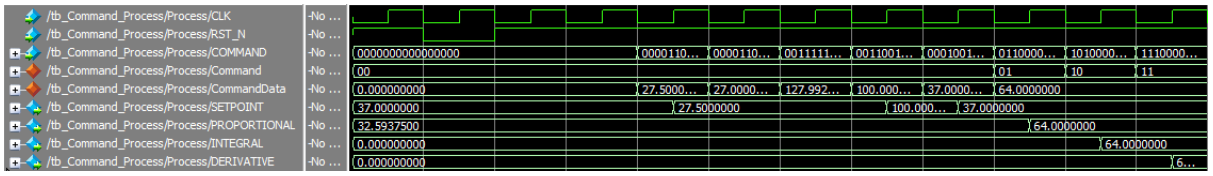

Fig. 8: Results of Testbench

PID Calculator
File: PID.v
Description: Takes the feedback signal and interprets the appropriate output to correct the error. The proportional function of the output is what brings back the response towards the setpoint. A higher P value can cause a larger overshoot and a lower one can make the error correction slower. The integral function is what lessens the steady-state error and the differential function is what lessens or eliminates the oscillation of the response. The output is the sum of all the PID portions.

Reference Utilized: [PID Arduino]
- The PID.v implementation is not a direct translation of the PID.h library but it was the main reference for implementing the response of each function.

Inputs:
- Master Clock (CLK)
- Master Reset (RST_N)
- PID constants (k_P, k_I, k_d)
- measured_value
- Setpoint

Outputs:
- Response (12-Bit)
- Valid response flag (computed)

Notes:
- Total clock cycles: 5
- response 12-bit number
  - response[11] = sign bit
  - response[10:4] = whole number
  - response[3:0] = decimal number
- Constants can be changed through the k_P, k_I, k_d pins.

Tests Done:
- **Testbench:** tb_PID.v
- Tests the individual outputs of the PID responses when encountering a decrease in error from a large starting measured value.
- Fig. 9 to 11 shows the results of the testbench.

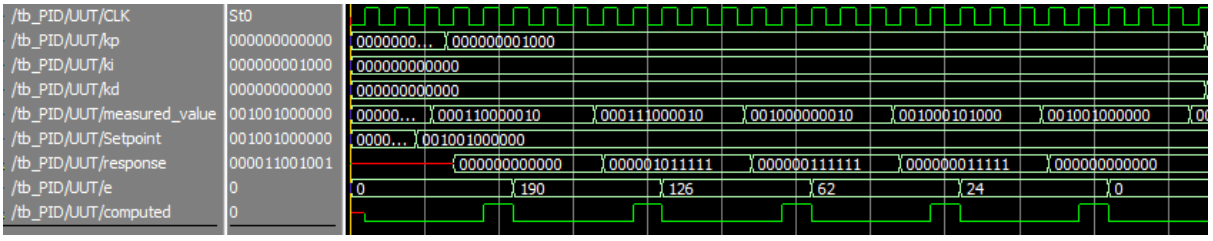

Fig. 9: Results of PID Testbench with Isolated P Response

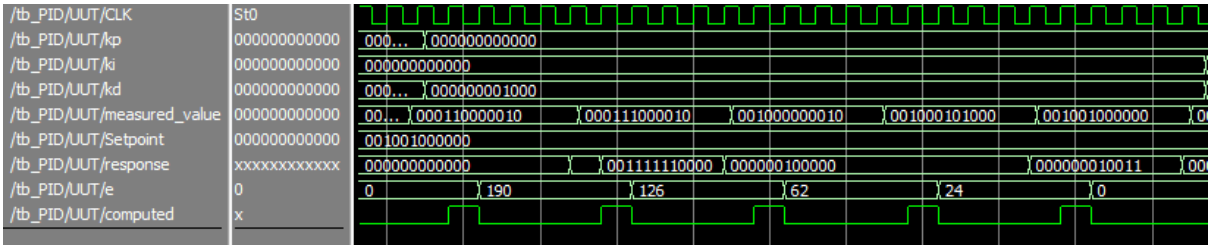

Fig. 10: Results of PID Testbench with Isolated I Response

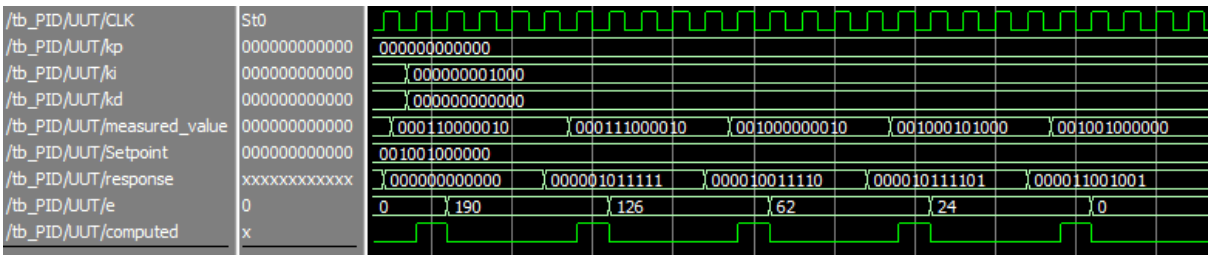

Fig. 11: Results of PID Testbench with Isolated D Response

Recommendations/Improvements:

- The amount of decimal bits could be improved by making it dynamic to make room for more accurate small number calculations and responses.
- If possible, reduce the number of clock cycles

Convert PID Response
File: Convert_Response.v
Description: This module converts the result of the PID calculation into an equivalent time-proportional value. It expects an input value between 0 and 63, which corresponds to the desired step values. The resolution of the resulting trigger time value is the following:

$$Resolution = \frac{1.0667\ Seconds}{63 - 0} \approx 0.0169\ second$$

Turning this into a 25-bit value in a 27 MHz system clock,

$$Resolution = 25'd457158$$

The time value is calculated to be

$$Trigger\ Time\ Value = Input * 25'd457158$$

Thus, the range of the trigger time value would be between 0 second to 1.0667 seconds with a 0.0169 second step size.

Due to the estimated 5-clock-cycle delay of the PID calculator module, a dedicated enable port was implemented. Upon receiving the "done" signal from the PID calculator, the "Convert PID Response" module automatically calculates the equivalent time value. If the module is not enabled, the last calculated equivalent time value remains the output.

Reference Utilized: None

Inputs:
- Master Clock (CLK)
- Master Reset (RST_N)
- Enable
- PID Response (Signed Q7.4)

Outputs:
- Trigger Time Value (25-Bit)

Notes:
- This module assumes that the system/master clock has a speed of 27 MHz, as the step value and the resulting trigger time value would be the equivalent to the time unit of seconds.
- Sign bit should not affect the results. Even though the PID output is expected to have no negative values, the module is designed to still reduce input errors.

Tests Done:
- **Testbench:** tb_Convert_Response.v
- Tests if enable works, if the sign bit and the fractional bits affect the results, and tests if the calculations are correct.
- Fig. 12 shows the results of the testbench.

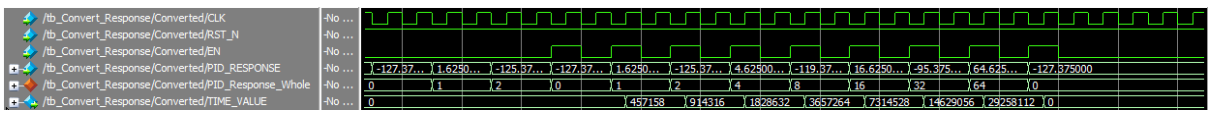

Fig. 12: Testbench Results of Convert Response Module

Time Proportioning Control
File: TimeProportion.v
Description: A counter with a period of 1.0667 seconds is integrated into a 27 MHz system clock. The module will output a high signal to the relay once the trigger time value becomes greater than the current count of the counter; otherwise, it will output a low signal to the relay.

Reference Utilized: None

Inputs:
- Master Clock (CLK)
- Master Reset (RST_N)
- Trigger Time Value (25-Bit)

Outputs:
- Trigger Signal to Relay

Notes:
- This module assumes the system/master clock has a speed of 27 MHz. As a result, the counter's maximum count is 28,800,900, which is equivalent to 1.0667 seconds.
- When the master reset button is pressed, the counter's current count is set to its maximum value. This prevents the relay from being triggered during the reset process.

Tests Done:
- No testbench was used.
- Tests were done by observing the relay LED while running the module.

Water Heater Wrapper
File: WaterHeater.v
Description: Wraps all the modules together and connects them to the FPGA pins.

Inputs:

- 27 MHz Clock (CLK)
- Pulled Up Pushbutton (RST_N)
- DS18B20 Data Wire (DS_WIRE)
- SPI Serial Clock (SCK)
- SPI MOSI
- SPI Chip Select (SS)
- SPI MISO

Outputs:

- Relay Trigger
- SPI MISO

ESP32 Sketches

SPI ESP32
File: spi.ino
Description:

- Receives Water Temperature Data from FPGA through SPI
- Sends command and command data to FPGA through SPI
- Gathers ambient air temperature through I2C (MCP9808 Sensor)
- Shows Setpoint and Current Water Temperature through I2C LCD
- Sends Current Water Temperature and Ambient Air Temperature to Firebase Cloud
- Receives changes in PID gain and Setpoint values from Firebase Cloud.

Dashboard ESP32
File: dashboard.ino
Description:

- Receives Current Water Temperature and Ambient Air Temperature from Firebase Cloud.
- Shows Current Water Temperature and Ambient Air Temperature values to the user through a dashboard.
- Allows users to change setpoint and PID gain values through the dashboard.
- Sends changes in setpoint and PID gain values to Firebase Cloud.

Project Video Demo

The project demo can be accessed in this link, which is also the project presentation video for ENGG 113.02.