Class Report 5

Alec Roessler

December 1 2025

1 Introduction

The objective of this report is to successfully perform Experiment 17.8.1 from Pong Chu's textbook. The experiment outlines a project utilizing I2C protocol to interface with the boards built in ADT7420 temperature sensor. Chu's code base for both software and hardware interfacing should be extended to prompt the sensor for its temperature reading, convert to a human readable temperature, and then display it on the boards built in seven segment displays. The temperature format should be in the form "25.12 C" and able to be switched to Fahrenheit with a users input such as a switch or button press.

2 Interfacing with ADT7420 Temperature Sensor

The ADT7420 sensor includes a built in ADC (analog to digital converter) which converts the analog temperature reading into a digitized value. This value is stored in the chips on board registers and can be accessed by using I2C.

2.1 I2C Protocol

The I2C protocol is known for its hardware simplicity requiring just two wires, SCL and SDA. SCL is the serial clock line which supplies the clock ticks to all peripheral devices while the SDA, or serial data line supplies the data transmission. Importantly, this line is bi directional meaning it can both transmit and receive data. Because we are interested in reading from the temperature sensor, we will focus our attention on the reading "mode". Below is a diagram depicting the required transmissions from the controller and peripheral devices to transfer information from the peripheral to the controller.

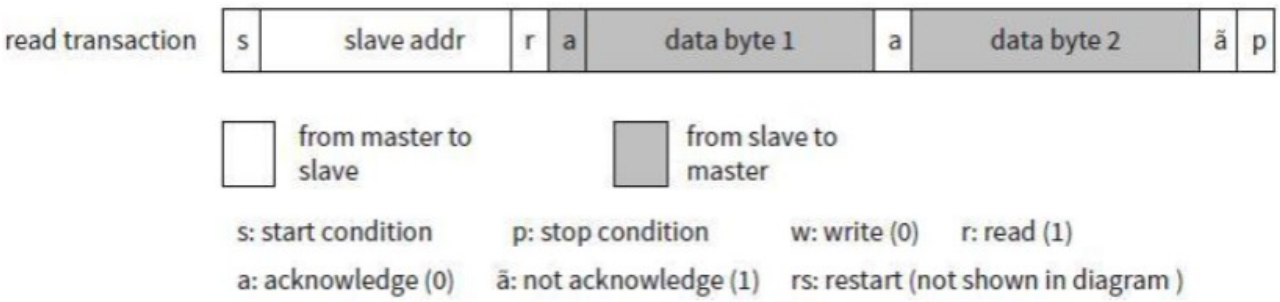

Figure 1: I2C Read Sequence

The controller first sends a start bit telling the peripherals that it wants to begin an operation. After this, it sends a byte containing the peripherals address in the 7 most significant bits of the transmitted byte. The last bit tells the peripheral if it is a read or write operation. Note that there can be many different peripherals connected to one controller via the same two wires and the controller can "talk" to each specific one simply by transmitting the corresponding address. After this byte is transmitted, the peripheral will respond with an acknowledge bit and then the peripteral transmits two data bytes with an acknowledge bit after each followed by a final "not acknowledge" and stop bits by the controller.

The timing of this process is dependent on the SCL line and the start condition is formed by the SCL line being low and SDA line transitions from high to low.

2.1.1 Hardware and Software Implementation

The hardware synthesized for this operation is that of a finite state machine to handle the logic associated with the I2C protocol. This is divided into 12 states corresponding to idle, hold, start1, start2, data1, data2, data3, data4, data_end, restart, stop1, and stop2. The clock used for the SCL line needs to be created for the hardware. The frequency of this clock can be anywhere from 100Khz to 3.4Mhz depending on the I2C mode being used. For each control condition, there are 2 slowed I2C clock cycles for every system cycle and 4 for each data cycle. The timing diagram is shown below in Figure 2. The number of cycles can then be calculated by dividing these numbers by the system clock frequency and is then used by a counter in the state machine.

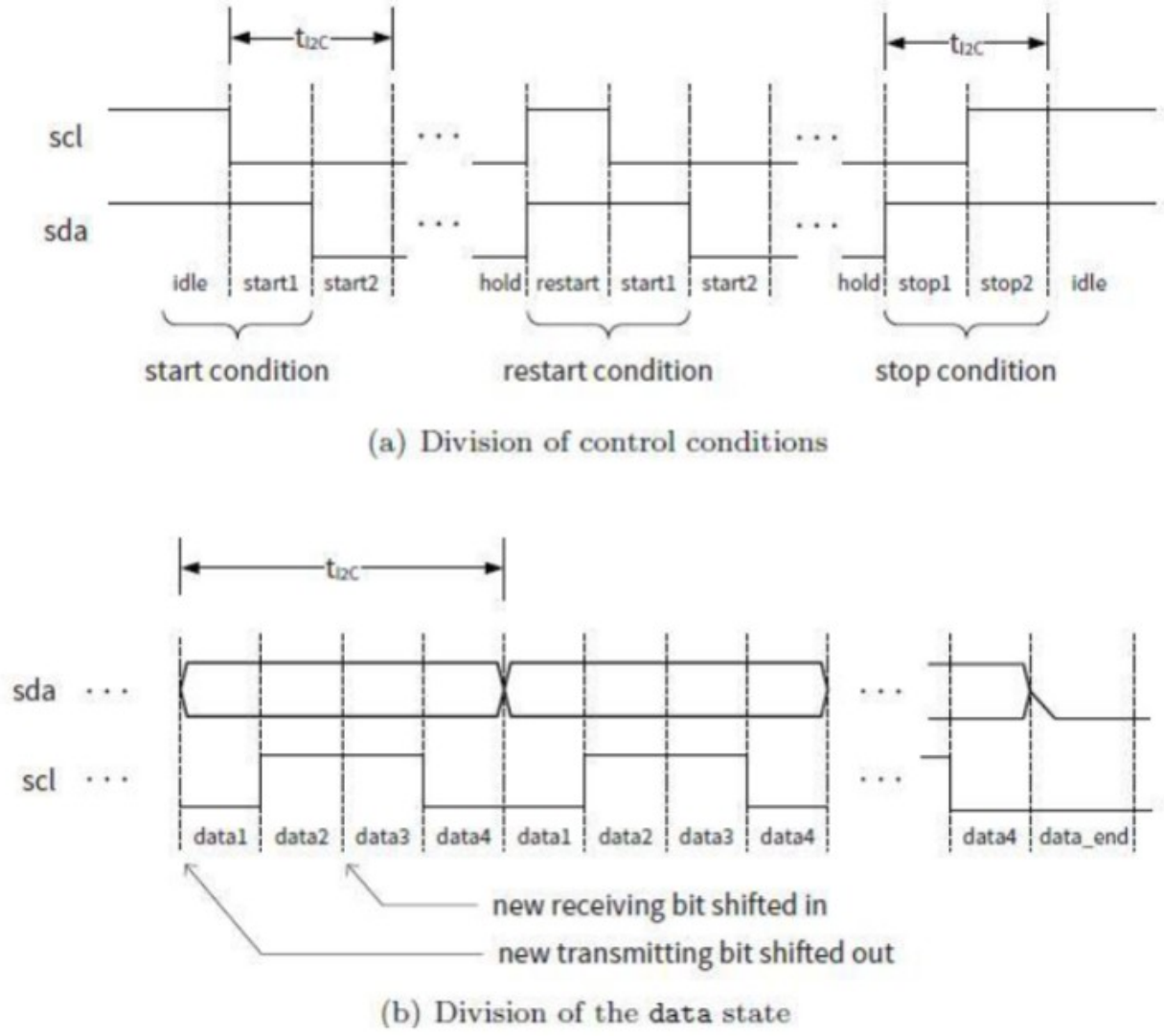

Figure 2: I2C Timing

The dvsr register is written to by the software core and sets the needed quarter period for the I2C clock rate. This is done in Chu's i2c_core.cpp with a SYS_CLK_FREQ of 100Mhz for the Nexys4 DDR board.

```
dvsr = (uint32_t) (SYS_CLK_FREQ * 1000000 / freq / 4);
```

The I2C core is then used to interact with the hardware for the read register and two write registers. The DVSR is written to the first write register while the other write register is used for writing data to the peripheral. This core is then integrated into slot 10. The i2c_core.cpp file is completed by writing functions to specify restart, ready, start, stop, write, and read to interface with the hardware.

2.1.2 ADT7420 Temperature Sensor

There are 14 registers located inside the ADT7420 sensor. Register 11 is the value of the sensors address (0xcb in hex). The first and second registers, corresponding to offsets of 0 and 1 respectively are the data registers holding the current temperature value in Celcius. Register 0 holds the upper byte while register 1 holds the

lower byte. After reconstructing this into a single value, the 13 most significant bits represent the final Celcius temperature to be read from the sensor and used for the controller.

2.2 Project Setup and Basic Temperature Program

When tackling this project, I decided the best course of action would be to first set up a basic project in which I can use Chu's ADT7420 test function as part of his main sampler program to verify the correct reading of the temperature and debug via UART.

2.2.1 Project Setup

To set up the project, I first uploaded all needed files into my Github repository under a new folder for Experiment 17.8.1. These were all needed files including constraint, HDL MMIO and top level modules, and CPP main sampler, headers, and cpp core files. The repository was then cloned to my local computer and a new Vivado project was started. The project instruction located in canvas were followed and all HDL files were added to the project. The file mmio_sys_sampler.sv was modified to delete all unnecessary slot instantiations leaving only UART, I2C, SSEG, GPI, SYSTEMTIMER, and GPO (SLOTS 0-3, 8, 10). Then, the generate function was modified so as to reflect these changes when setting the unused read registers to zero as done so in the Verilog code below.

```
for (i=4; i<64; i=i+1) begin
    if ((i != 8) && (i != 10))
    assign rd_data_array[i] = 32'h0;
```

2.2.2 Basic Temperature Program

To test the basic system and confirm correct temperature readings, Chu's function adt7490_check was used. This function first confirms the peripherals address is the correct address (0xCB) is correct in register 11. Next, it performs the I2C read sequence by calling the drivers functions for write (specifying peripheral and that it wants to read) and read (actually reading the registers contents for temperature output). Next it assembles both registers contents and reconstructs the temperature reading (13 most significant bits). After this it handles negative values by the sign bit and sends the result via UART to be displayed on an external serial monitor.

After running the program, the following was seen on the serial monitor (MOBAXTERM) successfully verifying the peripheral address and temperature reading.

```
read ADT7420 id (should be 0xcb): cb
temperature (C): 26.625
```

The final read_temp function was written by incorporating Chu's existing adt7490_check function and deleting the portions pertaining to verifying the address, printing over UART, and illuminating LED's.

3 Seven Segment Display

As per the experiment requirements, the temperature should be displayed on the on board seven segment displays. To do this I choose to use Chu's sseg core and extend it to handle the temperature printing. To do this I integrated a new function called print_temp as part of the sseg class. This new function allows for the temperature value in Celsius to be passed in as well as a boolean variable for determining if it should print in Celsius or Fahrenheit.

The function works as follows:

- Convert to the correct temperature unit (C or F) using the input boolean flag.
- Extract the ones, tens, hundreds, and thousands place individually from the temperature value.
- Fill the display buffer with leading whitespace, digits, and the correct unit character.
- Use Chu's `h2s` function to convert digits and the unit to seven-segment encoding.
- Write the encoded data to hardware using Chu's `write_8pt` function.

The function can then be called by simply passing in the temperature value in Celsius and setting the Celsius boolean flag.

4 Farenheit and Celcius Selection

This flag needs to be set by the user which necessitates the need for some sort of interaction with the physical world. My first thought was to use a button since we have a debouncing function already in Chu's main sampler. However, after thourough testing and timing debugging I was not able to get this to work.

4.1 Switch

As a fall back, I decided to incorporate the use of a swtich (namely switch 0). I decided this was actually a better course of action as the temperature unit could be set permently instead of the user having to hold down the button. To use the switch, I modified the main function to poll the switch's state every 100 ms and update the boolean is_celsius. The temperature is then read from the read_temp function and passed along with the is_celsius variable to print_temp.

```
int main() {
    //uint8_t id, ;
    float temp;
    bool is_celsius = true;
    int switch_state = 0;
    while (1) {
        switch_state = sw.read();
        if (switch_state & 0x01) {
            is_celsius = false;
         } else {
            is_celsius = true;
         }
        temp = temp_read(&adt7420, &led);
        sseg.print_temp(temp, is_celsius);
        sleep_ms(100);
    } //while
} //main
```

5 Submission Information

The Github repository with the files mentioned in this report can be found at
https://github.com/alecroessler/Softcore-SoC/tree/main/Experiment_17.8.1
Youtube video demonstration:
https://www.youtube.com/shorts/lR8BBTQ-FWk