

# Temperature Sensor - Report

By Alvin Grima and Ignacy Manturewicz

<b>Plagiarism Form.....</b>	<b>2</b>
<b>Aim.....</b>	<b>3</b>
<b>Methodology and Components.....</b>	<b>3</b>
Display Controller.....	3
Display.....	3
Switch.....	3
I2C Master.....	3
Sensor Controller.....	4
Divider.....	4
Top Level.....	4
Hierarchy of Modules.....	4
<b>Testing.....</b>	<b>4</b>
<b>Remarks and Conclusion.....</b>	<b>4</b>
<b>Code.....</b>	<b>6</b>
Display Controller.....	6
Analysis of Display Controller.....	7
Display.....	7
Analysis of Display.....	10
Switch.....	11
Analysis of Switch.....	11
I2C Master.....	12
Analysis of I2C Master Code.....	15
Entity i2c_master:.....	15
These are the output ports which were used:.....	15
A buffer port is used.....	16
Architecture 'logic' of the I2C master.....	16
Sensor Controller.....	23
Analysis of Sensor Controller.....	25
Entity temp_sensor.....	25
These are the output ports which were used:.....	26
Architecture 'behavior' of temp_sensor.....	26
The i2c master component is then created and it is instantiated.....	27
Divider.....	30
Analysis of Divider.....	30
Top Level.....	31
Constraint file.....	33

# Plagiarism Form

## FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

### Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own work, of work of another person, whether or not such work has been published” (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / We\*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our\* work, except where acknowledged and referenced.


I / We\* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

\* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Alvin Grima  
\_\_\_\_\_  
Student Name

  
\_\_\_\_\_  
Signature

Ignacy Manturewicz  
\_\_\_\_\_  
Student Name

  
\_\_\_\_\_  
Signature

\_\_\_\_\_  
Student Name

\_\_\_\_\_  
Signature

\_\_\_\_\_  
Student Name

\_\_\_\_\_  
Signature

MNE3002  
\_\_\_\_\_  
Course Code

FPGA Interfacing Project  
\_\_\_\_\_  
Title of work submitted

## Aim

This experiment aims to interface the onboard ADT7420 I<sup>2</sup>C *temperature sensor* with the 8-digit 7-segment *display* on the Nexys A7 FPGA Board, showing results in both Celsius and Fahrenheit scales.

## Methodology and Components

The temperature obtained by the *temperature sensor* will be displayed with the precision of two decimal places at the range from 0°C to 37°C (equivalently from 32°F to 99°F) on the right side of the *display*. It will be shown in either Celsius or Fahrenheit, depending on the on-board *switch*. The letter indicating which scale is being used will be displayed on the left side of the *display*.

### Display Controller

The Display Controller entity is responsible for manipulating the results obtained from the *Sensor Controller*. It takes the temperature reading (in degrees Celsius multiplied by one hundred) as an input. It outputs four decimal digits to be shown on the *display*. The scale on which the result is passed further depends on the *switch*.

[Display Controller code](#)

### Display

The Display entity takes the decimal values of each digit from the Display Controller and displays according to segments on the *Display*., just to resemble dividing by one hundred (inverting the previous multiplication that occurred in the Sensor Controller).

[Display code](#)

### Switch

The output of the Switch entity depends on the *switch*'s position. It is either 0 or 1: when it is 0, the temperature displayed will be in degrees of Celsius, when it is 1, the temperature displayed will be in degrees of Fahrenheit.

[Switch code](#)

### I<sup>2</sup>C Master

The I<sup>2</sup>C Master entity is responsible for handling the master-slave operations for the temperature sensor, it is used for serial communication. Its internal state machine starts with collecting the data to write and write the address of the slave to the bus. Later it proceeds to either write the obtained data or read the data from the slave and does this cycle. It operates on a previously generated, adjusted clock.

[I<sup>2</sup>C Master code](#)

## Sensor Controller

The Sensor Controller entity controls the I<sup>2</sup>C Master entity and parses its output to obtain the temperature in degrees Celsius multiplied by 100.

[Sensor\\_Controller\\_code](#)

## Divider

This entity divides the system clock so that it is easier to read correct data from the display.

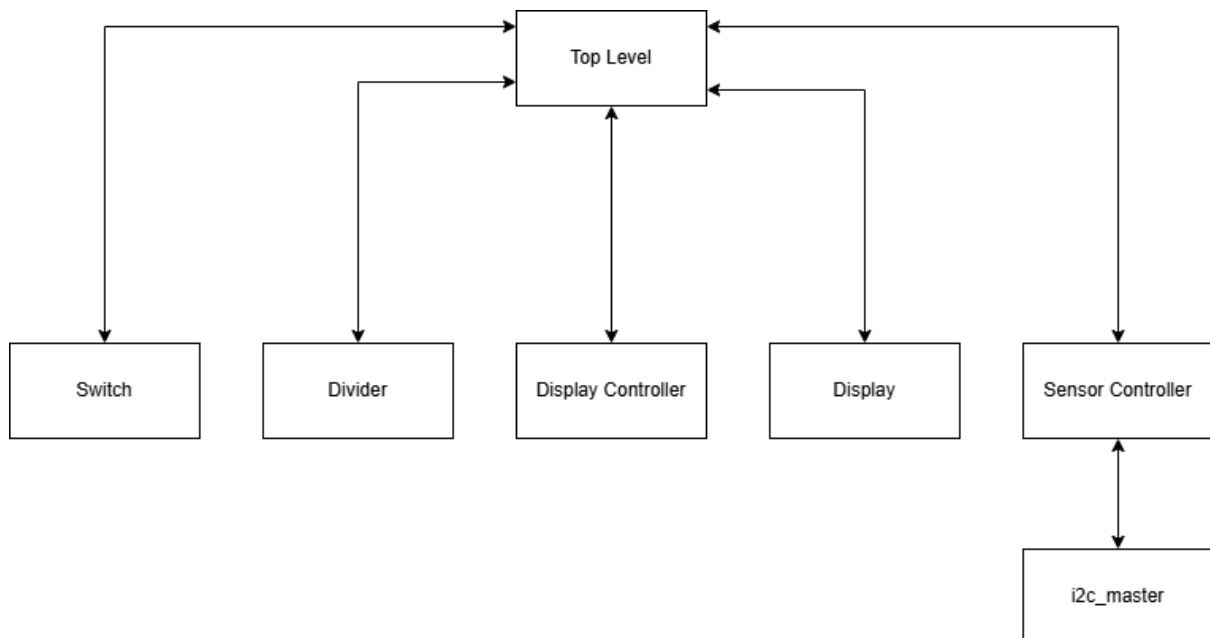
[Divider\\_code](#)

## Top Level

The Top Level connects other entities all together.

[Top\\_Level\\_code](#)

## Hierarchy of Modules



## Testing

The testing included two major phases. Firstly, the working of the display was verified by inputting preprogrammed arbitrary numbers. For example, for the given temperature signal value of 3660 the display showed 36.60. Secondly, the correctness of continuous temperature readings was checked by connecting the sensor to the display and breathing on it in order to imitate the changes in the temperature.

## Remarks and Conclusion

The experiment was successful. The hardest part in this project was dealing with the clock and setting accurate time delays. After completing it, the authors came to the conclusion that

*Temperature Sensor Report*

some entities (like the Switch) are a bit redundant and could be implemented differently without creating a whole module, however in the process of coding it, we opted to make it a separate module to keep each module as simple as possible.

# Code

## Display Controller

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity display_control is
    Port (
        CLK_in: in std_logic;
        F_in: in std_logic;
        temperatura_in: in integer range 0 to 9999;
        th_out: out integer range 0 to 9;
        rd_out: out integer range 0 to 9;
        nd_out: out integer range 0 to 9;
        st_out: out integer range 0 to 9
    );
end display_control;

architecture disp_cntrl_arch of display_control is
    signal temperaturef: integer range 0 to 9999;
begin
    TempChange: process(Clk_in)
    begin
        if rising_edge(Clk_in) then
            temperaturef <= temperatura_in * 9 / 5 + 3200;
        end if;
    end process TempChange;

    Control: process(CLK_in)
        variable temp1: integer;
        variable temp2: integer;
    begin
        if rising_edge(CLK_in) then
            if F_in = '0' then
                th_out <= temperatura_in / 1000;
                temp1 := temperatura_in - (1000 * (temperatura_in/1000));
                rd_out <= temp1 / 100;
                temp2 := temp1 - (100 * (temp1/100));
                nd_out <= temp2 / 10;
                st_out <= temp2 - (10 * (temp2/10));
            else
                th_out <= temperaturef / 1000;
                temp1 := temperaturef - (1000 * (temperaturef/1000));
                rd_out <= temp1 / 100;
                temp2 := temp1 - (100 * (temp1/100));
                nd_out <= temp2 / 10;
                st_out <= temp2 - (10 * (temp2/10));
            end if;
        end if;
    end process Control;
end disp_cntrl_arch;
```

## Analysis of Display Controller

Input Ports Table:

CLK_in	Clock signal	std_logic
F_in	Signal from the Switch entity indicating whether to pass further the temperature in degrees of Celsius (0) or Fahrenheit (1)	std_logic
temperatura_in	Raw integer temperature reading from the Sensor Controller ranged from 0 to 9999 (degrees of Celsius multiplied by 100)	integer (0-9999)

Output Ports Table:

th_out	Represents the tens value of the temperature to pass	integer (0-9)
rd_out	Represents the units value of the temperature to pass	integer (0-9)
nd_out	Represents the decimal part of the temperature to pass	integer (0-9)
st_out	Represents the hundredth part of the temperature to pass	integer (0-9)

The architecture of the Display Controller consists of 2 processes:

1. The TempChange process converts the input temperature from Celsius to Fahrenheit.
2. The Control process converts the raw temperature reading into its scaled form. To achieve it, a simple mathematical digit extraction by integer division is enough (e.g. to obtain the most significant digit from the integer signal that is at maximum 9999, the value of the signal can be integer divided by 1000 and the result will carry the wanted digit). This operation is performed for all 4 expected digits, decrementing the raw temperature value accordingly.

## Display

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity display is
    Port (
        CLK_in: in std_logic;
        Rst_in: in std_logic;
        th_in: in integer range 0 to 9;
        rd_in: in integer range 0 to 9;
        nd_in: in integer range 0 to 9;
        st_in: in integer range 0 to 9;
        F_in: in std_logic;
        DIGITS_out: out std_logic_vector (7 downto 0) := "00000000";
        DP_out: out std_logic;
        CA_out: out std_logic;
        CB_out: out std_logic;
        CC_out: out std_logic;
        CD_out: out std_logic;
        CE_out: out std_logic;
        CF_out: out std_logic;
        CG_out: out std_logic
    );
end display;

architecture display_arch of display is
begin
    Display: process(Clk_in, Rst_in)
        variable temp: integer range 1 to 5 := 5;
    begin
        if Rst_in = '1' then
            DIGITS_out <= "11111111";
        elsif rising_edge(Clk_in) then
            if temp = 5 then
                DIGITS_out <= "01111111";
                if F_in = '0' then CA_out <= '0'; CB_out <= '1'; CC_out <= '1'; CD_out <=
'0'; CE_out <= '0'; CF_out <= '0'; CG_out <= '1'; DP_out <= '1';
                else CA_out <= '0'; CB_out <= '1'; CC_out <= '1'; CD_out <= '1'; CE_out <=
'0'; CF_out <= '0'; CG_out <= '0'; DP_out <= '1';
                end if;
            elsif temp = 4 then
                DIGITS_out <= "11110111";
                if th_in = 9 then CA_out <= '0'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'0'; CE_out <= '1'; CF_out <= '0'; CG_out <= '0'; DP_out <= '1';
                elsif th_in = 8 then CA_out <= '0'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'0'; CE_out <= '0'; CF_out <= '0'; CG_out <= '0'; DP_out <= '1';
                elsif th_in = 7 then CA_out <= '0'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'1'; CE_out <= '1'; CF_out <= '1'; CG_out <= '1'; DP_out <= '1';
                elsif th_in = 6 then CA_out <= '0'; CB_out <= '1'; CC_out <= '0'; CD_out <=
'0'; CE_out <= '0'; CF_out <= '0'; CG_out <= '0'; DP_out <= '1';
                elsif th_in = 5 then CA_out <= '0'; CB_out <= '1'; CC_out <= '0'; CD_out <=
'0'; CE_out <= '1'; CF_out <= '0'; CG_out <= '0'; DP_out <= '1';
                elsif th_in = 4 then CA_out <= '1'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'1'; CE_out <= '1'; CF_out <= '0'; CG_out <= '0'; DP_out <= '1';
                elsif th_in = 3 then CA_out <= '0'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'0'; CE_out <= '1'; CF_out <= '1'; CG_out <= '0'; DP_out <= '1';
                elsif th_in = 2 then CA_out <= '0'; CB_out <= '0'; CC_out <= '1'; CD_out <=
'0'; CE_out <= '0'; CF_out <= '1'; CG_out <= '0'; DP_out <= '1';
                elsif th_in = 1 then CA_out <= '1'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'1'; CE_out <= '1'; CF_out <= '1'; CG_out <= '1'; DP_out <= '1';
                else CA_out <= '0'; CB_out <= '0'; CC_out <= '0'; CD_out <= '0'; CE_out <=
'0'; CF_out <= '0'; CG_out <= '1'; DP_out <= '1';
                end if;
            elsif temp = 3 then
                DIGITS_out <= "11111011";
                if rd_in = 9 then CA_out <= '0'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'0'; CE_out <= '1'; CF_out <= '0'; CG_out <= '0'; DP_out <= '0';
                elsif rd_in = 8 then CA_out <= '0'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'0'; CE_out <= '0'; CF_out <= '0'; CG_out <= '0'; DP_out <= '0';
                elsif rd_in = 7 then CA_out <= '0'; CB_out <= '0'; CC_out <= '0'; CD_out <=
'1'; CE_out <= '1'; CF_out <= '1'; CG_out <= '1'; DP_out <= '0';
            end if;
        end if;
    end process;
end display_arch;
```





## Analysis of Display

Input Ports Table:

CLK_in	Clock signal	std_logic
Rst_in	Reset signal	std_logic
th_in	Represents the tens value of the temperature to display	integer (0-9)
rd_in	Represents the units value of the temperature to display	integer (0-9)
nd_in	Represents the decimal part of the temperature to display	integer (0-9)
st_in	Represents the hundredth part of the temperature to display	integer (0-9)
F_in	Signal from the Switch entity indicating whether to display the 'C' or 'F' letter on the left side of the display	std_logic

Output Ports Table:

DIGITS_out	Indicates which digit of the display is active at the moment	std_logic_vector (7-0)
DP_out	Indicates if the decimal point is lit at the active digit	std_logic
CA_out - CG_out	Indicate if the according segment is lit at the active digit	std_logics

Each clock cycle this entity outputs only one digit and then moves to the next one in the following cycle. Still, since the clock frequency is high the output values appear to be displayed simultaneously and the results are clear. The segments lit on the first digit depend on the *Switch*, it is either 'C' or 'F' exhibiting which scale is being used. Then, the 4 digits on the right side of the *display* are occupied by the reading of the temperature obtained from the Display Controller entity. The third digit from the left has an additional decimal point segment lit in order to make the reading clear.

## Switch

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.numeric_std.all;

entity switch is
    Port (
        CLK_in: in std_logic;
        SW_in: in std_logic;
        F_out: out std_logic
    );
end switch;

architecture switch_arch of switch is
begin
    CzynFahrenheit: process(Clk_in)
    begin
        if rising_edge(Clk_in) then
            if SW_in = '0' then
                F_out <= '0';
            else
                F_out <= '1';
            end if;
        end if;
    end process CzynFahrenheit;
end switch_arch;
```

### Analysis of Switch

Input Ports Table:

CLK_in	Clock signal	std_logic
SW_in	Signal from the switch	std_logic

Output Ports Table:

F_out	Indicates later Fahrenheit consequences	std_logic
-------	---	-----------

The architecture of the Switch consists of passing the value of the system input SW\_in onto the internal signal F\_out.

## I<sup>2</sup>C Master

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY i2c_master IS
  GENERIC(
    input_clk : INTEGER := 50_000_000; --input clock speed from user logic in Hz
    bus_clk   : INTEGER := 400_000;    --speed the i2c bus (scl) will run at in Hz
  )
  PORT(
    clk      : IN      STD_LOGIC;      --system clock
    rst_n    : IN      STD_LOGIC;      --active low reset
    ena      : IN      STD_LOGIC;      --latch in command
    addr     : IN      STD_LOGIC_VECTOR(6 DOWNTO 0); --address of target slave
    rw       : IN      STD_LOGIC;      --'0' is write, '1' is read
    data_wr   : IN      STD_LOGIC_VECTOR(7 DOWNTO 0); --data to write to slave
    busy     : OUT     STD_LOGIC;      --indicates transaction in progress
    data_rd   : OUT     STD_LOGIC_VECTOR(7 DOWNTO 0); --data read from slave
    ack_error : BUFFER STD_LOGIC;      --flag if improper acknowledge from
    slave
    sda      : INOUT   STD_LOGIC;      --serial data output of i2c bus
    scl      : INOUT   STD_LOGIC;      --serial clock output of i2c bus
  )
END i2c_master;

ARCHITECTURE logic OF i2c_master IS
  CONSTANT divider : INTEGER := (input_clk/bus_clk)/4; --number of clocks in 1/4 cycle of
  scl
  TYPE machine IS(readystate, startstate, command, slv_ack1, write, read, slv_ack2,
  mstr_ack, stopstate); --needed states
  SIGNAL state      : machine;          --state machine
  SIGNAL dataclk    : STD_LOGIC;        --data clock for sda
  SIGNAL dataclkprev : STD_LOGIC;       --data clock during previous system
  clock
  SIGNAL scl_clk    : STD_LOGIC;        --constantly running internal scl
  SIGNAL scl_ena    : STD_LOGIC := '0'; --enables internal scl to output
  SIGNAL sda_int    : STD_LOGIC := '1'; --internal sda
  SIGNAL sda_ena_n  : STD_LOGIC;        --enables internal sda to output
  SIGNAL addr_rw    : STD_LOGIC_VECTOR(7 DOWNTO 0); --latched in address and read/write
  SIGNAL data_tx    : STD_LOGIC_VECTOR(7 DOWNTO 0); --latched in data to write to slave
  SIGNAL data_rx    : STD_LOGIC_VECTOR(7 DOWNTO 0); --data received from slave
  SIGNAL bit_cnt    : INTEGER RANGE 0 TO 7 := 7;   --tracks bit number in transaction
  SIGNAL stretch   : STD_LOGIC := '0';          --identifies if slave is
  stretching scl
BEGIN

  --generate the timing for the bus clock (scl_clk) and the data clock
  PROCESS(clk, rst_n)
    VARIABLE count : INTEGER RANGE 0 TO divider*4; --timing for clock generation
  BEGIN
    IF(rst_n = '0') THEN --reset asserted
      stretch <= '0';
      count := 0;
    ELSIF(clk'EVENT AND clk = '1') THEN
      dataclkprev <= dataclk; --store previous value of data clock
      IF(count = divider*4-1) THEN --end of timing cycle
        count := 0; --reset timer
      ELSIF(stretch = '0') THEN --clock stretching from slave not detected
        count := count + 1; --continue clock generation timing
      END IF;
      CASE count IS
        WHEN 0 TO divider-1 => --first 1/4 cycle of clocking
          scl_clk <= '0';
          dataclk <= '0';
        WHEN divider TO divider*2-1 => --second 1/4 cycle of clocking
          scl_clk <= '0';
          dataclk <= '1';
        WHEN divider*2 TO divider*3-1 => --third 1/4 cycle of clocking
          scl_clk <= '1';
          IF(scl = '0') THEN --detect if slave is stretching clock
            stretch <= '1';
          ELSE
            stretch <= '0';
          END IF;
        WHEN divider*3 TO divider*4-1 => --fourth 1/4 cycle of clocking
          scl_clk <= '1';
          dataclk <= '0';
        WHEN OTHERS =>
          scl_clk <= '0';
          dataclk <= '0';
      END CASE;
    END IF;
  END PROCESS;
END logic;
```

Temperature Sensor Report

```

        END IF;
        dataclk <= '1';
    WHEN OTHERS =>                                --last 1/4 cycle of clocking
        scl_clk <= '1';
        dataclk <= '0';
    END CASE;
END IF;
END PROCESS;

--state machine and writing to sda during scl low (data_clk rising edge)
PROCESS(clk, rst_n)
BEGIN
    IF(rst_n = '0') THEN                            --reset asserted
        state <= readystate;                        --return to initial state
        busy <= '1';                                --indicate not available
        scl_ena <= '0';                             --sets scl high impedance
        sda_int <= '1';                             --sets sda high impedance
        ack_error <= '0';                          --clear acknowledge error flag
        bit_cnt <= 7;                                --restarts data bit counter
        data_rd <= "00000000";                      --clear data read port
    ELSIF(clk'EVENT AND clk = '1') THEN
        IF(dataclk = '1' AND dataclkprev = '0') THEN --data clock rising edge
            CASE state IS
                WHEN readystate =>                  --idle state
                    IF(ena = '1') THEN              --transaction requested
                        busy <= '1';                --flag busy
                        addrw <= addr & rw;          --collect requested slave address and command
                        datatx <= data_wr;           --collect requested data to write
                        state <= startstate;
                    ELSE
                        busy <= '0';
                        state <= readystate;
                    END IF;
                WHEN startstate =>                    --start bit of transaction
                    busy <= '1';                    --resume busy if continuous mode
                    sda_int <= addrw(bit_cnt);      --set first address bit to bus
                    state <= command;
                WHEN command =>                      --address and command byte of transaction
                    IF(bit_cnt = 0) THEN             --command transmit finished
                        sda_int <= '1';             --release sda for slave acknowledge
                        bit_cnt <= 7;               --reset bit counter for "byte" states
                        state <= slv_ack1;          --go to slave acknowledge (command)
                    ELSE
                        bit_cnt <= bit_cnt - 1;      --next clock cycle of command state
                        sda_int <= addrw(bit_cnt-1); --keep track of transaction bits
                        state <= command;           --write address/command bit to bus
                    END IF;
                WHEN slv_ack1 =>                     --slave acknowledge bit (command)
                    IF(addrw(0) = '0') THEN          --write first bit of data
                        sda_int <= datatx(bit_cnt);
                        state <= write;
                    ELSE
                        sda_int <= '1';             --read command
                        state <= read;              --release sda from incoming data
                    END IF;
                WHEN write =>                         --write byte of transaction
                    busy <= '1';                    --resume busy if continuous mode
                    IF(bit_cnt = 0) THEN             --write byte transmit finished
                        sda_int <= '1';             --release sda for slave acknowledge
                        bit_cnt <= 7;               --reset bit counter for "byte" states
                        state <= slv_ack2;          --go to slave acknowledge (write)
                    ELSE
                        bit_cnt <= bit_cnt - 1;      --next clock cycle of write state
                        sda_int <= datatx(bit_cnt-1); --keep track of transaction bits
                        state <= write;           --write next bit to bus
                    END IF;
                WHEN read =>                         --read byte of transaction
                    busy <= '1';                    --resume busy if continuous mode
                    IF(bit_cnt = 0) THEN             --read byte receive finished
                        IF(ena = '1' AND addrw = addr & rw) THEN --continuing with another read at
same address
                            sda_int <= '0';         --acknowledge the byte has been received
                        ELSE
                            sda_int <= '1';         --stopping or continuing with a write
                        END IF;
                    END IF;
                WHEN repeated_start =>              --send a no-acknowledge (before stop or

```

```

        END IF;
        bit_cnt <= 7;
        data_rd <= datarx;
        state <= mstr_ack;
    ELSE
        bit_cnt <= bit_cnt - 1;
        state <= read;
    END IF;
    WHEN slv_ack2 =>
        IF(ena = '1') THEN
            busy <= '0';
            addrw <= addr & rw;
            datatx <= data_wr;
            IF(addrw = addr & rw) THEN
                sda_int <= data_wr(bit_cnt); --write first bit of data
                state <= write;
            ELSE
                --continue transaction with a read or new
slave
                state <= startstate;
            END IF;
        ELSE
            --complete transaction
            state <= stopstate;
        END IF;
    WHEN mstr_ack =>
        IF(ena = '1') THEN
            busy <= '0';
            --continue is accepted and data received is
available on bus
            addrw <= addr & rw;
            datatx <= data_wr;
            IF(addrw = addr & rw) THEN
                sda_int <= '1';
                state <= read;
            ELSE
                --continue transaction with a write or new
slave
                state <= startstate;
            END IF;
        ELSE
            --complete transaction
            state <= stopstate;
        END IF;
    WHEN stopstate =>
        busy <= '0';
        state <= readystate;
        --stop bit of transaction
        --unflag busy
    END CASE;
    ELSIF(dataclk = '0' AND dataclkprev = '1') THEN --data clock falling edge
        CASE state IS
            WHEN startstate =>
                IF(scl_ena = '0') THEN
                    scl_ena <= '1';
                    ack_error <= '0';
                END IF;
                --starting new transaction
                --enable scl output
                --reset acknowledge error output
            WHEN slv_ack1 =>
                IF(sda /= '0' OR ack_error = '1') THEN
                    --receiving slave acknowledge (command)
                    --no-acknowledge or previous
no-acknowledge
                    ack_error <= '1';
                END IF;
                --set error output if no-acknowledge
            WHEN read =>
                datarx(bit_cnt) <= sda;
                --receiving slave data
                --receive current slave data bit
            WHEN slv_ack2 =>
                IF(sda /= '0' OR ack_error = '1') THEN
                    --receiving slave acknowledge (write)
                    --no-acknowledge or previous
no-acknowledge
                    ack_error <= '1';
                END IF;
                --set error output if no-acknowledge
            WHEN stopstate =>
                scl_ena <= '0';
                --disable scl
            WHEN OTHERS =>
                NULL;
        END CASE;
    END IF;
END IF;
END PROCESS;

--set sda output
WITH state SELECT
    sda_ena_n <= dataclkprev WHEN startstate, --generate start condition
    NOT dataclkprev WHEN stopstate, --generate stop condition

```

```

        sda_int WHEN OTHERS;           --set to internal sda signal

--set scl and sda outputs
scl <= '0' WHEN (scl_ena = '1' AND scl_clk = '0') ELSE 'Z';
sda <= '0' WHEN sda_ena_n = '0' ELSE 'Z';

END logic;

```

## Analysis of I<sup>2</sup>C Master Code

### Entity i2c\_master:

Generics is defined by the entity like ports however unlike ports they can have default values and are used to define configurable parameters rather than external connections.

Generics	Description	Data Type
input_clk	System clock frequency already pre-defined as 50MHz	INTEGER
bus_clk	The i2c bus clock frequency already pre-defined as 400kHz	INTEGER

These are the input ports which were used:

Ports	Description	Data Type
clk	System Clock Input	STD_LOGIC
rst_n	Active Low reset signal	STD_LOGIC
ena	Enable signal to start a transaction	STD_LOGIC
addr	7-bit address of the target I2c Slave	STD_LOGIC_VECTOR(6 DOWNTO 0)
rw	Read/Write control bit (0 for write, 1 for read)	STD_LOGIC

These are the output ports which were used:

Ports	Description	Data Type
data_wr	8-bit data to write to the slave	STD_LOGIC_VECTOR(7 DOWNTO 0)
busy	Indicates whether the master is busy with a	STD_LOGIC

	transaction	
data_rd	8-bit data read from the slave	STD_LOGIC_VECTOR(7 DOWNTO 0)

A buffer port is used

The buffer port is similar to an INOUT port as it is written to and read from, however it is only driven by the master and does not need to be driven by an external entity hence an INOUT port is not appropriate.

Port	Description	Data Type
ack_error	Flag indicating if the slave failed to acknowledge	STD_LOGIC

These are the INOUT ports which were used:

Port	Description	Data Type
sda	Bidirectional serial data line	STD_LOGIC
scl	Bidirectional serial clock line	STD_LOGIC

### Architecture 'logic' of the I2C master

These are the Constants which were used:

Constants	Description	Data Type
divider	<p>Calculates the number of clock cycles to generate the I2C clock</p> <p>The equation is:  <math>(input\_clk/bus\_clk)/4</math> which depicts the number of clock cycles which correspond to <math>\frac{1}{4}</math> of the I2C clock cycle.</p>	INTEGER

These are the signals which were used:



Signals	Description	Data Type
state	represents the current state of the state machine	machine
dataclk dataclkprev	<p>These signals are used to control the timing of data transitions on the sda line</p> <p>dataclk specifically is a clock signal which determines when data (sda) should be changed</p> <p>dataclkprev stores the previous value of dataclk to detect rising or falling edges.</p>	STD_LOGIC STD_LOGIC
scl_clk	is the internal I2C clock signal	STD_LOGIC
scl_ena	<p>enables or disables the I2C clock output.</p> <p>when scl_ena is 1, the scl_clk signal is driven onto the scl line</p> <p>when scl_ena is 0, the scl line is set to high impedance allowing the slave device to control the clock</p>	STD_LOGIC
sda_int	represents the internal SDA signal, the signal holds the value of the data SDA that is being transmitted or received	STD_LOGIC
sda_ena_n	<p>Controls the SDA output when sda_ena_n is 0, the value of sda_int is driven onto the sda line.</p> <p>When sda_ena_n is 1, the sda line is set to high impedance (Z) allowing the slave device to control the data line</p>	STD_LOGIC
addrw	Stores the 7-bit slave address and the read/write	STD_LOGIC_VECTOR(7 DOWNTO 0)

	<p>bit that are transmitted during the address phase of the i2c transaction</p> <p>format is addrw(7 DOWNT0 1) for the address and the addrw(0) for the read/write bit.</p>	
datatx	<p>holds the data to be written to the slave</p> <p>the signal holds 8-bit data that is transmitted to the slave during a write operation</p> <p>The data is shifted out bit by bit onto the sda line during the write phase</p>	STD_LOGIC_VECTOR(7 DOWNT0 0)
datarx	<p>Stores the data received from the slave</p> <p>The signal holds 8-bit data that is received from the slave during a read operation.</p> <p>The data is shifted in bit by bit from the sda line during the read phase.</p>	STD_LOGIC_VECTOR(7 DOWNT0 0)
bit_cnt	<p>Tracks the current bit position in a transaction</p> <p>This signal is used to count the bits transmitted or received during a transaction.</p> <p>It starts at 7 for the MSB and decrements to 0 LSB.</p>	INTEGER RANGE 0 TO 7
stretch	<p>Detects if the slave is stretching the clock</p> <p>Clock stretching is a feature of I2C protocol that allows the slave device to pause the transaction by holding the SCL line low.</p>	STD_LOGIC

	This signal ensures proper synchronisation between the master and slave.	
--	--	--

The states are the following:

readystate:

This is the initial state where the I2C master waits for a transaction to begin.

Behavior:

- The busy signal is set to 0, indicating that the master is idle and ready to accept a new command.
- The master continuously monitors the ena (enable) signal. When ena is asserted (ena = '1'), the master latches the address (addr), read/write bit (rw), and data to write (data\_wr) into internal signals (addrw and datatx).
- The state transitions to startstate to initiate the transaction.

Key Signals:

- busy: Set to 0 (idle).
- addrw: Latches the 7-bit address and read/write bit.
- datatx: Latches the data to be written (if it's a write operation).

Why is this state useful?

This state ensures the I2C master is idle and ready to accept new commands. It monitors the ena signal to determine when to start a transaction, ensuring that the master does not initiate communication prematurely or miss a transaction request.

---

startstate (Start Condition)

Generates the I2C start condition, which signals the beginning of a transaction.

Behavior:

- The master pulls the sda line low while the scl line is high. This is the I2C start condition.
- The busy signal is set to 1, indicating that the master is now busy with a transaction.
- The state transitions to command to send the address and read/write bit.

Key Signals:

sda\_int: Driven low to generate the start condition.

scl\_ena: Enabled to start generating the I2C clock.

Why is this state useful?

This state generates the I2C start condition, which signals the beginning of a transaction. It ensures that the master and slave are synchronized before any data is transmitted, adhering to the I2C protocol and preventing communication errors.

---

command (Address and R/W Bit Transmission)

Transmits the 7-bit slave address and the read/write bit to the bus.

Behavior:

- The master sends the 7-bit address and the read/write bit (addrw) serially on the sda line, one bit per clock cycle.
- The bit\_cnt signal is used to track which bit is being transmitted (starting from bit 7 down to bit 0).

*Temperature Sensor Report*

- After transmitting all 8 bits (7 address bits + 1 R/W bit), the master releases the sda line and transitions to slv\_ack1 to wait for the slave's acknowledgment.

Key Signals:

addrw: The 8-bit value containing the address and R/W bit.

bit\_cnt: Decrementing after each bit is transmitted.

Why is this state useful?

This state transmits the 7-bit slave address and the read/write bit to the bus. It ensures that the correct slave device is addressed and informed about the type of operation (read or write), enabling proper communication between the master and the targeted slave.

slv\_ack1 (Slave Acknowledgment for Address/Command)

Waits for the slave to acknowledge the address and R/W bit.

Behavior:

- The master releases the sda line, allowing the slave to pull it low as an acknowledgment.
- If the slave does not pull sda low (no acknowledgment), the ack\_error flag is set.
- Depending on the R/W bit:
  - If it's a write operation (rw = '0'), the state transitions to write to send data.
  - If it's a read operation (rw = '1'), the state transitions to read to receive data.

Key Signals:

ack\_error: Set if the slave fails to acknowledge.

Why is this state useful?

This state waits for the slave's acknowledgment after the address and R/W bit have been transmitted. It ensures that the slave has received the address and command correctly, preventing the master from proceeding with a transaction if the slave is unresponsive or the address is incorrect.

write (Data Transmission to Slave)

Transmits data to the slave during a write operation.

Behavior:

- The master sends 8 bits of data (datatx) serially on the sda line, one bit per clock cycle.
- The bit\_cnt signal tracks the current bit being transmitted.
- After transmitting all 8 bits, the master releases the sda line and transitions to slv\_ack2 to wait for the slave's acknowledgment.

Key Signals:

datatx: The data to be written to the slave.

bit\_cnt: Decrementing after each bit is transmitted.

Why is this state useful?

This state transmits data to the slave during a write operation. It ensures that the data is sent bit by bit, adhering to the I2C protocol and enabling the master to write configuration settings or other data to the slave.

read (Data Reception from Slave)

Purpose: Receives data from the slave during a read operation.

Behavior:

- The master releases the sda line, allowing the slave to drive it.
- The master reads 8 bits of data from the sda line, one bit per clock cycle.
- The received data is stored in the datarx signal.
- After receiving all 8 bits, the master decides whether to acknowledge the data:
  - If continuing with another read operation, the master acknowledges by pulling

*Temperature Sensor Report*

sda low.

- If stopping or switching to a write operation, the master does not acknowledge (sda remains high)
- The state transitions to mstr\_ack to handle the acknowledgment.

Key Signals:

data\_rx: Stores the received data.

bit\_cnt: Decrement after each bit is received.

Why is this state useful?

This state receives data from the slave during a read operation. It ensures that the data is read bit by bit, adhering to the I2C protocol and enabling the master to retrieve information (e.g., sensor readings) from the slave.

---

slv\_ack2 (Slave Acknowledgment for Data)

Waits for the slave to acknowledge the data during a write operation.

Behavior:

- The master releases the sda line, allowing the slave to pull it low as an acknowledgment.
- If the slave does not pull sda low (no acknowledgment), the ack\_error flag is set.
- If the transaction is continuing (e.g., another write operation), the state transitions back to write.
- If the transaction is complete, the state transitions to stopstate.

Key Signals:

ack\_error: Set if the slave fails to acknowledge.

Why is this state useful?

This state waits for the slave's acknowledgment after data has been transmitted during a write operation. It ensures that the slave has received the data correctly, preventing the master from proceeding if there is a communication error or the slave is unable to process the data.

---

mstr\_ack (Master Acknowledgment for Data)

Handles the master's acknowledgment after receiving data during a read operation.

Behavior:

- If continuing with another read operation, the master acknowledges by pulling sda low.
- If stopping or switching to a write operation, the master does not acknowledge (sda remains high).
- The state transitions to stopstate if the transaction is complete, or back to read if continuing with another read operation.

Key Signals:

sda\_int: Controls the acknowledgment signal.

Why is this state useful?

This state handles the master's acknowledgment after receiving data during a read operation. It allows the master to control the flow of data by either acknowledging (ACK) to continue receiving more data or not acknowledging (NACK) to end the transaction, ensuring efficient and controlled communication.

---

stopstate (Stop Condition)

Purpose: Generates the I2C stop condition, which signals the end of a transaction.

Behavior:

- The master pulls the sda line high while the scl line is high. This is the I2C stop condition.

*Temperature Sensor Report*

- The busy signal is set to 0, indicating that the master is idle and ready for a new transaction.
- The state transitions back to readystate.

Key Signals:

sda\_int: Driven high to generate the stop condition.

scl\_ena: Disabled to stop generating the I2C clock.

Why is this state useful?

This state generates the I2C stop condition, which signals the end of a transaction. It ensures that the bus is released and ready for the next transaction, adhering to the I2C protocol and preventing bus contention or communication errors.

## Sensor Controller

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY temp_sensor IS
  GENERIC(
    sys_clk_freq      : INTEGER := 50_000_000;           -- System clock
    frequency in Hz
    temp_sensor_addr : STD_LOGIC_VECTOR(6 DOWNTO 0) := "1001011" -- I2C address of the
    temperature sensor
  );
  PORT(
    clk      : IN      STD_LOGIC;           -- System clock
    rst_n    : IN      STD_LOGIC;           -- Active-low reset
    scl      : INOUT   STD_LOGIC;           -- I2C serial clock
    sda      : INOUT   STD_LOGIC;           -- I2C serial data
    --i2c_ack_err : OUT   STD_LOGIC;           -- I2C acknowledge
    error_flag (small change with Alvin's)
    temp      : OUT     integer range 0 to 10000      -- Temperature value
    (small change with Alvin's)
  );
END temp_sensor;

ARCHITECTURE behavior OF temp_sensor IS
  -- Define the states of the state machine
  TYPE machine IS (startstate, setresolution, pausestate, read_data, output_result);
  SIGNAL state      : machine;           -- State machine signal
  SIGNAL i2cena      : STD_LOGIC;         -- I2C enable signal
  SIGNAL i2caddr     : STD_LOGIC_VECTOR(6 DOWNTO 0); -- I2C target address
  SIGNAL i2crw       : STD_LOGIC;         -- I2C read/write command
  SIGNAL i2cdatawr   : STD_LOGIC_VECTOR(7 DOWNTO 0); -- I2C write data
  SIGNAL i2cdatard   : STD_LOGIC_VECTOR(7 DOWNTO 0); -- I2C read data
  SIGNAL i2cbusy     : STD_LOGIC;         -- I2C busy signal
  SIGNAL busyprev    : STD_LOGIC;         -- Previous value of I2C busy
  signal
  SIGNAL tempdata    : STD_LOGIC_VECTOR(15 DOWNTO 0); -- Buffer for raw temperature
  data
  SIGNAL temperatureraw : UNSIGNED(12 DOWNTO 0);         -- Raw temperature after
  processing
  SIGNAL temperaturescaled : UNSIGNED(15 DOWNTO 0);     -- Scaled temperature value
  signal i2c_ack_err : STD_LOGIC; --small change with Alvin's

  -- I2C Master component declaration
  COMPONENT i2c_master IS
    GENERIC(
      input_clk : INTEGER; -- Input clock frequency in Hz
      bus_clk   : INTEGER  -- I2C bus clock frequency in Hz
    );
    PORT(
      clk      : IN      STD_LOGIC;           -- System clock
      rst_n    : IN      STD_LOGIC;           -- Active-low reset
      ena      : IN      STD_LOGIC;           -- Enable transaction
      addr     : IN      STD_LOGIC_VECTOR(6 DOWNTO 0); -- Target slave address
      rw       : IN      STD_LOGIC;           -- Read/Write control ('0' for
  write, '1' for read)
      data_wr  : IN      STD_LOGIC_VECTOR(7 DOWNTO 0); -- Data to write
      busy     : OUT     STD_LOGIC;           -- Indicates ongoing transaction
      data_rd  : OUT     STD_LOGIC_VECTOR(7 DOWNTO 0); -- Data read from slave
      ack_error : BUFFER STD_LOGIC;           -- Acknowledge error flag
      sda      : INOUT   STD_LOGIC;           -- I2C data line
      scl      : INOUT   STD_LOGIC;           -- I2C clock line
    );
  END COMPONENT;

BEGIN
  -- Instantiate the I2C Master
  i2c_master_0: i2c_master
    GENERIC MAP(input_clk => sys_clk_freq, bus_clk => 400_000)
    PORT MAP(
      clk => clk, rst_n => rst_n, ena => i2cena, addr => i2caddr,
      rw => i2crw, data_wr => i2cdatawr, busy => i2cbusy,
```

*Temperature Sensor Report*

```

    data_rd => i2cdatar, ack_error => i2c_ack_err, sda => sda, scl => scl
);

PROCESS(clk, rst_n)
    VARIABLE busycnt : INTEGER RANGE 0 TO 3 := 0;           -- Counts I2C busy transitions
    VARIABLE counter : INTEGER RANGE 0 TO sys_clk_freq/10 := 0; -- 100ms delay counter
BEGIN
    IF (rst_n = '0') THEN
        -- Reset logic
        counter := 0;
        i2cena <= '0';
        busycnt := 0;
        temp <= 0;
        state <= startstate;
    ELSIF (clk'EVENT AND clk = '1') THEN
        -- State machine logic
        CASE state IS
            WHEN startstate =>
                -- Wait 100ms before starting communication
                IF (counter < sys_clk_freq/10) THEN
                    counter := counter + 1;
                ELSE
                    counter := 0;
                    state <= setresolution;
                END IF;
            WHEN setresolution =>
                -- Configure the temperature sensor
                busyprev <= i2cbusy;
                IF (busyprev = '0' AND i2cbusy = '1') THEN
                    busycnt := busycnt + 1;
                END IF;
                CASE busycnt IS
                    WHEN 0 =>
                        i2cena <= '1';
                        i2caddr <= temp_sensor_addr;
                        i2crw <= '0';
                        i2cdatarw <= "00000011"; -- Address of Configuration Register
                    WHEN 1 =>
                        i2cdatarw <= "10000000"; -- Set 16-bit resolution
                    WHEN 2 =>
                        i2cena <= '0';
                        IF (i2cbusy = '0') THEN
                            busycnt := 0;
                            state <= pausestate;
                        END IF;
                    WHEN OTHERS => NULL;
                END CASE;
            WHEN pausestate =>
                -- Pause between transactions
                IF (counter < sys_clk_freq/769_000) THEN
                    counter := counter + 1;
                ELSE
                    counter := 0;
                    state <= read_data;
                END IF;
            WHEN read_data =>
                -- Read temperature data
                busyprev <= i2cbusy;
                IF (busyprev = '0' AND i2cbusy = '1') THEN
                    busycnt := busycnt + 1;
                END IF;
                CASE busycnt IS
                    WHEN 0 =>
                        i2cena <= '1';
                        i2caddr <= temp_sensor_addr;
                        i2crw <= '0';
                        i2cdatarw <= "00000000"; -- Address of Temperature Value MSB Register
                    WHEN 1 =>
                        i2crw <= '1';
                    WHEN 2 =>
                        IF (i2cbusy = '0') THEN
                            tempdata(15 DOWNT0 8) <= i2cdatar; -- MSB data
                        END IF;
                    WHEN OTHERS => NULL;
                END CASE;
            WHEN OTHERS => NULL;
        END CASE;
    END IF;
END PROCESS;

```

### Temperature Sensor Report



```

        END IF;
    WHEN 3 =>
        i2cena <= '0';
        IF (i2cbusy = '0') THEN
            tempdata(7 DOWNT0 0) <= i2cdatarad; -- LSB data
            busycnt := 0;
            state <= output_result;
        END IF;
    WHEN OTHERS => NULL;
END CASE;

WHEN output_result =>
    -- Process and output temperature data
    temperatureraw <= unsigned(tempdata(15 DOWNT0 3));
    temperaturescaled <= resize((temperatureraw * 625) / 100, 16);
    temp <= to_integer(signed(std_logic_vector(temperaturescaled)));
    state <= pausestate;
WHEN OTHERS =>
    state <= startstate;
END CASE;
END IF;
END PROCESS;
END behavior;

```

## Analysis of Sensor Controller

### Entity temp\_sensor

Generics is defined by the entity like ports however unlike ports they can have default values and are used to define configurable parameters rather than external connections.

Generics	Description	Data Type
sys_clk_freq	System clock frequency already pre-defined as 50MHz	INTEGER
temp_sensor_addr	7-bit I2C address of the temperature sensor pre-defined as "1001011"	STD_LOGIC_VECTOR(6 DOWNT0 0)

These are the input ports which were used:

Ports	Description	Data Type
clk	System Clock Input	STD_LOGIC
rst_n	Active Low reset signal	STD_LOGIC

These are the INOUT ports which were used:

Port	Description	Data Type
sda	Bidirectional serial data line	STD_LOGIC

scl	Bidirectional serial clock line	STD_LOGIC
-----	---------------------------------	-----------

These are the output ports which were used:

Ports	Description	Data Type
i2c_ack_error	Acknowledge error flag from the I2C master	STD_LOGIC
temp	Output signal representing the scaled temperature value range from 0 to 10000	INTEGER

### Architecture 'behavior' of temp\_sensor

These are the signals which were used:

Signals	Description	Data Type
state	represents the current state of the state machine	machine
i2cena	Enable the signal for the i2c master	
i2caddr	7-bit I2C address of the ADT7420	
12crw	Read/Write control bit for the I2C master	
i2cdatawr	Data written to ADT7420 such as resolution setting and configuration of register address	
i2cdatar	Data read from the ADT7420	
i2cbusy	Indicates whether the I2C master is busy	
busyprev	holds the previous value of the i2cbusy for edge detection	
tempdata	Buffer for raw temperature data	
temperatureraw	Raw temperature value after processing (13 bits)	
temperaturescaled	Scaled temperature value from 13 bits to 16 bits	

i2c_ack_err	Acknowledge error flag from the I2C master	
-------------	--	--

The i2c master component is then created and it is instantiated

The states are the following:

startstate (Initialization State):

This is the initial state where the system waits for a fixed delay (100 ms) after reset to ensure the temperature sensor is powered up and ready for communication.

Behavior:

- A counter is used to create a 100 ms delay. This delay is necessary because many I2C devices require a stabilization period after power-up before they can respond to commands.
- The counter increments on each clock cycle until it reaches the value corresponding to 100 ms ( $\text{sys\_clk\_freq}/10$ ).
- Once the delay is complete, the state transitions to setresolution to begin configuring the sensor.

Key Signals:

- counter: Used to count clock cycles for the 100 ms delay.
- state: Transitions to setresolution after the delay.

Why is this state useful?

This state ensures the temperature sensor is ready to accept commands.

Prevents communication errors that could occur if the sensor is accessed too quickly after power-up.

---

setresolution (Configuration State)

This is the state which configures the temperature sensor to use 16-bit resolution for temperature readings. This ensures the sensor provides high-precision temperature data.

Behavior:

- The I2C master is enabled ( $\text{i2cena} = '1'$ ), and the sensor's configuration register is addressed.
- The following steps are performed:
  - Write the configuration register address ( $\text{i2cdatawr} = "00000011"$ ) to the sensor.
  - Write the resolution setting ( $\text{i2cdatawr} = "10000000"$ ) to the configuration register. This sets the sensor to 16-bit resolution.
- The busycnt variable is used to track the progress of these steps:
  - busycnt = 0: Enable I2C and send the configuration register address.
  - busycnt = 1: Send the resolution setting.

*Temperature Sensor Report*

- busycnt = 2: Disable I2C and wait for the transaction to complete.

Once the configuration is complete, the state transitions to pausestate.

Key Signals:

- i2cena: Enabled to start the I2C transaction.
- i2caddr: Set to the temperature sensor's I2C address (temp\_sensor\_addr).
- i2crw: Set to '0' (write operation).
- i2cdawr: Holds the data to be written (register address and resolution setting).
- busycnt: Tracks the progress of the configuration steps.

Why is this state useful?

This state ensures the sensor is configured to provide high-precision temperature data. and sets up the sensor for accurate readings.

#### pausestate (Pause State)

This state introduces a short pause between transactions to allow the sensor to process the configuration and prepare for the next operation.

Behavior:

- A counter is used to create a short delay. The delay duration is calculated as  $\text{sys\_clk\_freq}/769\_000$ , which corresponds to a small pause (e.g., a few microseconds).
- Once the delay is complete, the state transitions to read\_data to begin reading temperature data.

Key Signals:

- counter: Used to count clock cycles for the pause.
- state: Transitions to read\_data after the pause.

Why is this state useful?

This state provides a buffer between transactions to ensure the sensor is ready for the next operation hence it prevents overlapping commands that could cause communication errors.

#### read\_data (Data Read State)

This state reads the raw temperature data from the sensor. This involves reading two bytes (MSB and LSB) from the sensor's temperature register.

Behavior:

- The I2C master is enabled (i2cena = '1'), and the sensor's temperature register is addressed.
- The following steps are performed:
  - Write the temperature register address (i2cdawr = "00000000") to the sensor.
  - Switch to read mode (i2crw = '1') to read the temperature data.
  - Read the MSB of the temperature data and store it in tempdata(15 DOWNT0 8).
  - Read the LSB of the temperature data and store it in tempdata(7 DOWNT0 0).

- The busycnt variable is used to track the progress of these steps:
  - busycnt = 0: Enable I2C and send the temperature register address.
  - busycnt = 1: Switch to read mode.
  - busycnt = 2: Read the MSB.
  - busycnt = 3: Read the LSB and disable I2C.
- Once the data is read, the state transitions to output\_result.

#### Key Signals

- i2cena: Enabled to start the I2C transaction.
- i2caddr: Set to the temperature sensor's I2C address (temp\_sensor\_addr).
- i2crw: Switches between write ('0') and read ('1') modes.
- i2cdawr: Holds the temperature register address.
- i2cdatard: Holds the data read from the sensor.
- tempdata: Stores the raw temperature data (16 bits).
- busycnt: Tracks the progress of the read steps.

#### Why is this state useful?

This state retrieves the raw temperature data from the sensor and prepares the data for processing in the next state.

---

#### output\_result (Data Processing State)

This state processes the raw temperature data and outputs the scaled temperature value.

#### Behavior:

- The raw temperature data (tempdata) is processed to extract the 13-bit temperature value (temperatureraw). This is done by taking the upper 13 bits of the 16-bit raw data (tempdata(15 DOWNT0 3)).
- The raw temperature is scaled to a meaningful value. The scaling formula is:
  - $\text{temperaturescaled} = (\text{temperatureraw} * 625) / 100$
 This converts the raw data into a temperature value with a resolution of 0.0625°C (For accurate temperature as per the ADT7420 manual)
- The scaled temperature is converted to an integer and output on the temp signal.
- The state transitions back to pausestate to repeat the process.

#### Key Signals

- temperatureraw: Holds the 13-bit raw temperature value.
- temperaturescaled: Holds the scaled temperature value (16 bits).
- temp: Outputs the final temperature value as an integer.

#### Why is this state useful?

This state converts the raw sensor data into a usable temperature value and provides the final output for use by other parts of the system.

## Divider

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity divider is
    Port (
        CLK_in : in  STD_LOGIC;
        RST_in : in  STD_LOGIC;
        CLK_out : out STD_LOGIC);
end divider;

architecture divider_arch of divider is
begin
    process(CLK_in, RST_in)
        variable count: integer;
    begin
        if RST_in = '1' then
            count := 0;
            CLK_out <= '0';
        elsif rising_edge(CLK_in) then
            if count >= 0 and count < 50000 then
                CLK_out <= '1';
            elsif count >= 50000 and count < 100000 then
                CLK_out <= '0';
            elsif count = 100000 then
                count := 0;
            end if;
            count := count + 1;
        end if;
    end process;
end divider_arch;
```

### Analysis of Divider

Input Ports Table:

CLK_in	Default clock signal	std_logic
RST_in	Reset signal	std_logic

Output Ports Table:

CLK_out	Slowed Clock	std_logic
---------	--------------	-----------

The architecture of the Divider consists of slowing down the clock so the digits on the display are better visible.

## Top Level

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.numeric_std.all;

entity Temperature_Sensor_tl is
  Port (
    CLK100MHZ : in STD_LOGIC;
    RST : in STD_LOGIC;
    SW: in std_logic;
    DIGITS: out std_logic_vector (7 downto 0);
    DP: out std_logic;
    CA: out std_logic;
    CB: out std_logic;
    CC: out std_logic;
    CD: out std_logic;
    CE: out std_logic;
    CF: out std_logic;
    CG: out std_logic;
    SDA : inout STD_LOGIC;
    SCL : inout STD_LOGIC
  );
end Temperature_Sensor_tl;

architecture TempSensTL_arch of Temperature_Sensor_tl is
  signal CLK: std_logic;
  signal F: std_logic;
  signal temperatura: integer range 0 to 10000 := 3660;
  signal th: integer range 0 to 10;
  signal rd: integer range 0 to 9;
  signal nd: integer range 0 to 9;
  signal st: integer range 0 to 9;

  component divider is
    Port (
      CLK_in : in STD_LOGIC;
      RST_in : in STD_LOGIC;
      CLK_out : out STD_LOGIC
    );
  end component;

  component display_control is
    Port (
      CLK_in: in std_logic;
      F_in: in std_logic;
      temperatura_in: in integer range 0 to 9999;
      th_out: out integer range 0 to 10;
      rd_out: out integer range 0 to 9;
      nd_out: out integer range 0 to 9;
      st_out: out integer range 0 to 9
    );
  end component;

  component display is
    Port (
      CLK_in: in std_logic;
      RST_in: in std_logic;
      th_in: in integer range 0 to 10;
      rd_in: in integer range 0 to 9;
      nd_in: in integer range 0 to 9;
      st_in: in integer range 0 to 9;
      F_in: in std_logic;
      DIGITS_out: out std_logic_vector (7 downto 0) := "00000000";
      DP_out: out std_logic;
      CA_out: out std_logic;
      CB_out: out std_logic;
      CC_out: out std_logic;
      CD_out: out std_logic;
      CE_out: out std_logic;
      CF_out: out std_logic;
      CG_out: out std_logic
    );
  end component;
```

*Temperature Sensor Report*



```

);
end component;

component switch is
Port (
    CLK_in: in std_logic;
    SW_in: in std_logic;
    F_out: out std_logic
);
end component;
begin
divider_1: divider
port map(
    CLK_in => CLK100MHZ,
    RST_in => RST,
    CLK_out => CLK
);

display_control_1: display_control
port map(
    CLK_in => CLK,
    F_in => F,
    temperatura_in => temperatura,
    th_out => th,
    rd_out => rd,
    nd_out => nd,
    st_out => st
);

display_1: display
port map(
    CLK_in => CLK,
    RST_in => RST,
    th_in => th,
    rd_in => rd,
    nd_in => nd,
    st_in => st,
    F_in => F,
    DIGITS_out => DIGITS,
    DP_out => DP,
    CA_out => CA,
    CB_out => CB,
    CC_out => CC,
    CD_out => CD,
    CE_out => CE,
    CF_out => CF,
    CG_out => CG
);

switch_1: switch
port map(
    CLK_in => CLK,
    SW_in => SW,
    F_out => F
);

sensor_controller_1: entity work.sensor_controller
port map (
    clk => CLK100MHZ,
    rst_n => '1', -- Active-low reset
    scl => SCL,
    sda => SDA,
    temp => temperatura
);
end TempSensTL_arch;

```

The Top Level consists of copying the ports of all components previously analyzed and then assigning the signals of each entity to the common top level and system signals. The system signals enabled can be found in the constraint file below.

## Constraint file

```
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];

set_property -dict { PACKAGE_PIN V10     IOSTANDARD LVCMOS33 } [get_ports { SW }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

set_property -dict { PACKAGE_PIN T10     IOSTANDARD LVCMOS33 } [get_ports { CA }];
#IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10     IOSTANDARD LVCMOS33 } [get_ports { CB }]; #IO_25_14
Sch=cb
set_property -dict { PACKAGE_PIN K16     IOSTANDARD LVCMOS33 } [get_ports { CC }]; #IO_25_15
Sch=cc
set_property -dict { PACKAGE_PIN K13     IOSTANDARD LVCMOS33 } [get_ports { CD }];
#IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15     IOSTANDARD LVCMOS33 } [get_ports { CE }];
#IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11     IOSTANDARD LVCMOS33 } [get_ports { CF }];
#IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18     IOSTANDARD LVCMOS33 } [get_ports { CG }];
#IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15     IOSTANDARD LVCMOS33 } [get_ports { DP }];
#IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17     IOSTANDARD LVCMOS33 } [get_ports { DIGITS[0] }];
#IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18     IOSTANDARD LVCMOS33 } [get_ports { DIGITS[1] }];
#IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9      IOSTANDARD LVCMOS33 } [get_ports { DIGITS[2] }];
#IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14     IOSTANDARD LVCMOS33 } [get_ports { DIGITS[3] }];
#IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14     IOSTANDARD LVCMOS33 } [get_ports { DIGITS[4] }];
#IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14     IOSTANDARD LVCMOS33 } [get_ports { DIGITS[5] }];
#IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2      IOSTANDARD LVCMOS33 } [get_ports { DIGITS[6] }];
#IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13     IOSTANDARD LVCMOS33 } [get_ports { DIGITS[7] }];
#IO_L23N_T3_A02_D18_14 Sch=an[7]

set_property -dict { PACKAGE_PIN N17     IOSTANDARD LVCMOS33 } [get_ports { RST }];
#IO_L9P_T1_DQS_14 Sch=btnc

set_property -dict { PACKAGE_PIN C14     IOSTANDARD LVCMOS33 } [get_ports { SCL }];
#IO_L1N_T0_AD0N_15 Sch=tmp_scl
set_property -dict { PACKAGE_PIN C15     IOSTANDARD LVCMOS33 } [get_ports { SDA }];
#IO_L12N_T1_MRCC_15 Sch=tmp_sda
```