ENGG*3050
RCS Project Report

December 2nd, 2024

Mini Project: Dual Microblaze XADC Temperature reader

GROUP 17
Charlie Magri (1194399),
Yasir Al-Obaidi (1145544),
Jovan Gradojevic (1195257),
Haroon Shahbaz (1174177)

Table of Contents

Contents

List of Figures

Introduction:

This project demonstrates the communication and coordination of two MicroBlaze processors on an FPGA platform (NEXYS-A7 or ZedBoard) to solve a computational problem collectively. Our project allows the user to have access to the temperature value of the FPGA which solves the problems of FPGA's overheating and reaching dangerous temperatures as the user will be able to identify potentially harming temperature values before the damage to the board is done. Using many processors on an FPGA increases computing efficiency, especially for applications that can be parallelized. This project is inspired by the need for scalable, low cost, high-performance solutions in embedded systems that require resource sharing and inter-processor communication. This project also provides a real-life application of how FPGA technology can be used in a commercial product

Background/Lit Review:

For the methodologies we provided, one was done using Vivado HLS to develop a hardware software co-design implementation of the project. This methodology used a SDK and a c program to compute the task of constantly accessing the temperature of the board. In addition, a Vivado application programmed with pure VHDL was created to simulate access to the temperature value without a hardware software co-design approach. When it comes to the Software design of our block diagram methodology, the programming to monitor the essentials of the board were centered around the xsysmon.h library. In Xilinx, the xsysmon.h header file is part of the Xilinx System Monitor (Sysmon) library, which provides functions to interact with the System Monitor IP core integrated into Xilinx FPGAs. The Sysmon can measure various parameters such as temperature, voltage, and current. When working with the Nexys A7 board, the Sysmon IP core can be used to access the temperature readings from the on-chip temperature sensor. This allows for continuous monitoring of the temperature in the FPGA environment, which can be useful for system health monitoring, dynamic voltage and frequency scaling (DVFS), and other applications requiring temperature awareness. For this project, the lab resources from lab 1 assisted our group with resources in how to implement the 7-segment display of the Nexys A7. In addition, one of the tutorials provided in lab 5 gave us the resources to set up a single MicroBlaze block diagram design.

Main Body (Methodology):

Referring to figure 1 for the block diagram of the hardware software co-design for a visual depiction of our process. For a detailed description of our process: Using the Nexys A7 board as the hardware platform for the two Micro blazes as well as the built-in temperature sensor (XADC chip), we will simulate a sensor data acquisition system using two MicroBlaze processors on an FPGA. One MicroBlaze will act as the producer which will simulate a sensor reading data from a physical sensor such as a temperature sensor and write it to a shared memory system. The second MicroBlaze will read the sensor data, process it, and send the processed data to a UART terminal for monitoring by the user. This setup mimics a common real-world application where multiple processors manage different parts of a system (data acquisition, processing, and display). Communication between the processors happens through shared memory using AXI interconnect. The two Micro Blaze processors will communicate using an AXI interface with shared memory architecture. Each processor will have specified roles to ensure effective data sharing and reduce conflicts.

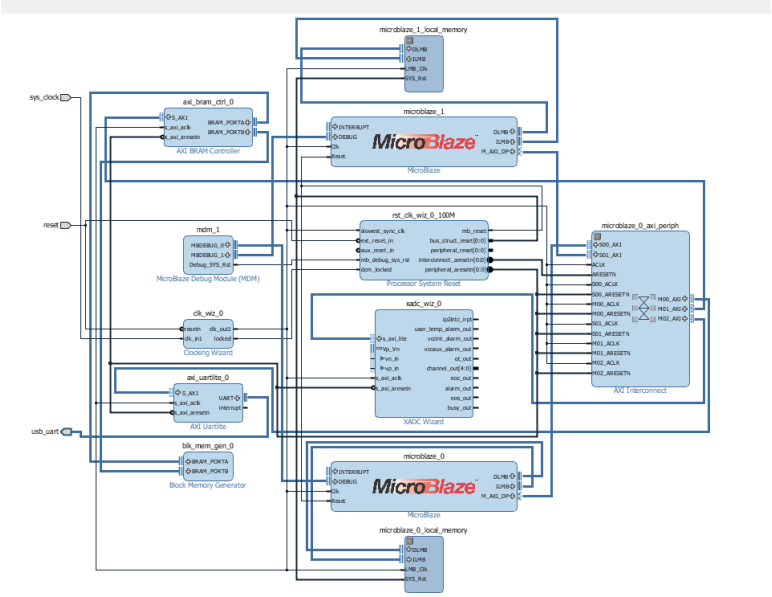

Figure 1: Block diagram of Master-Slave dual Microblaze system communicating with XADC Temperature sensor

Our process will also be implemented using state machines with pure VHDL codes as well and the results of the pure VHDL code implementation will be compared to the hardware system to review the advantages and disadvantages of the micro blazes possible applications. For the implementation using strictly hardware and VHDL, the code can be found in the appendix. Similarly to the hard/software co-design we began by instantiating a XADC block, however, instead of a direct AXI connection, we use the DRPs (Dynamic

Reconfiguration Ports) which allow a direct connection through VHDL. This replaces the flexibility and scalability of the AXI protocols with a deterministic and hardware-drive signal flow. It is worth noting that due to the usage of a necessary hardware component, we cannot simulate this design, instead we will create a separate implementation that will simulate temperature readings through a tightly coupled state machine. To create the pure VHDL version, first we will port map each of the necessary ports for the XADC block to function such as di_in, daddr_in, den_in, dwe_in, drdy_out, and do_out. We set the daddr_in which is the temperature register to the predefined constant "00000" which specifically targets the temperature reading channel. Next, we use den_in to enable the DRP for reading purposes in tandem with the drdy_out signal which will notify us when the data is ready in the 16-bit do_out bus. Now, the do_out data here is in its raw ADC format which means we will use the formula provided by Xilinx to extract a meaningful Celsius value. We will round the results of the following parameters due to the significant overhead effects if done with decimals. For example, the following equation will be rounded to 12.

$$T_{Celsius} = \left(\frac{Raw\ ADC\ Value \times Scale\ Factor}{Full\ Scale}\right) - Offset$$

Finally, we can extract the upper nibble (highBits) and lower nibble (lowBits) with the two seven-segment decoders, decode_low and decode_high. The digits will be toggled at a rate of 5ms to ensure there is no flickering and display alternates between both digits seamlessly. Finally, we truncate the Celsius temperature data into a data_out signal which will be constrained so we can compare it to our other implementation.

The testbench code will create a realistic raw temperature value then follow the conversions used in the above equation. Then we will follow the states as usual and verify the waveform to see if it matches the expected temperature values. During testing, we used values that would not be feasible, thus the equation provided negative answers as it would not be possible to process these values. It follows the IDLE, WAIT_FOR_XADC and SEND_DATA data states. First, we will wait for the clk to indicate high so the XADC is ready, then it will move to the WAIT_FOR_XADC state until it can retrieve data. Finally, the data is sent to wherever necessary in the future.

Although the hardware will be the focus of this project, software will be used to begin activities and ease communication. After we will compare the results of pure VHDL code implementation with the results of co-design implementation and discuss the advantages and disadvantages and their possible applications.

Results/Analysis:

When it comes to the Hardware Software Co-design methodology of the project, Xilinx was the application used to launch our SDK. As seen in Figure 6, an output of the temperature value was displayed in the built-in terminal in the Xilinx application. As our project was done on the Nexys A7, profiling our co-design was not an option for us due to the complexity of profiling on a board without a central processing unit. However, Xilinx has a built-in design summary that provides with all the essential information needed to compare this methodology with our other methodologies. As seen in Figures 2, 3, 4, and 5, we were able to output a utilization report as well as all the timing constraints of the codesign aspect of the project. With only 1243 LUTs out of 63400 used, the utilization is very low. This is an expected value as a very simple task is being done and MicroBlaze cores are known for being resource efficient. Similarly, only 76 out of 126,800 flip-flops (FFs) are used, which is minimal. This indicates that the design doesn't heavily rely on sequential logic. 24 out of 210 IOs are used. This is a moderate utilization and reflects the communication requirements between the MicroBlaze cores and other peripherals or external systems. In addition, the design meets all timing constraints (setup, hold, and pulse width) with no violations.

Figure 2: Setup Timing constraint of block diagram

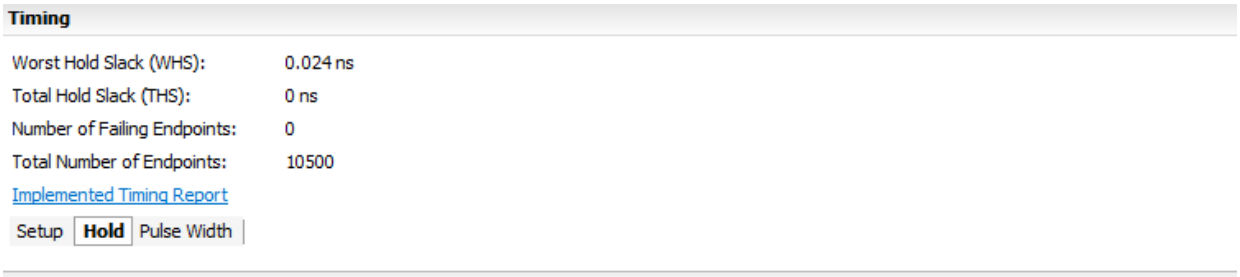

Figure 3: Hold Timing constraint of block diagram

Timing

Worst Pulse Width Slack (WPWS):	3 ns
Total Pulse Width Negative Slack (TPWS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	3297

Implemented Timing Report

Setup | Hold | **Pulse Width**

Figure 4: Pulse Width Timing constraint of block diagram

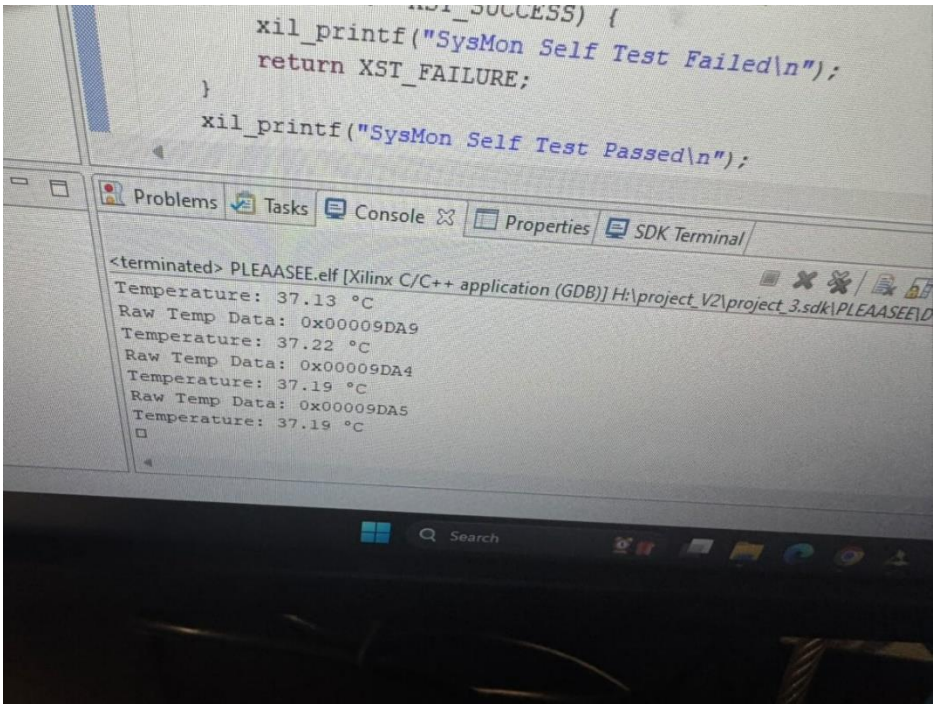

Figure 5: Example output of Hardware Software Co-Design SDK C file

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	2931	63400	4.62
LUTRAM	260	19000	1.37
FF	2681	126800	2.11
BRAM	66	135	48.89
IO	4	210	1.90
BUFG	3	32	9.38
MMCM	1	6	16.67

Figure 6: Post-Implementation Utilization of the Software/Hardware Co-design

In terms of the pure VHDL results, we can view the results of the utilization report in Figure 7. Compared to the hardware/software implementation the number of endpoints is much smaller using only a tiny fraction, 167 compared to 10,500 for the setup. This does not mean the software/hardware co-design is faster but does indicate that there is significantly more complexity compared to the pure VHDL version. Additionally, there are extreme setup timing violations which are due to the overhead of the temperature calculations, contrary to the H/S which has a built-in conversion for Celsius in Sysmon. In Figure 8, we can locate the exact issue in the temperature where the clock violation occurs. To fix this, the VHDL code must be further optimized, and extra variables would need to be initialized to hold the intermediary values between operations.

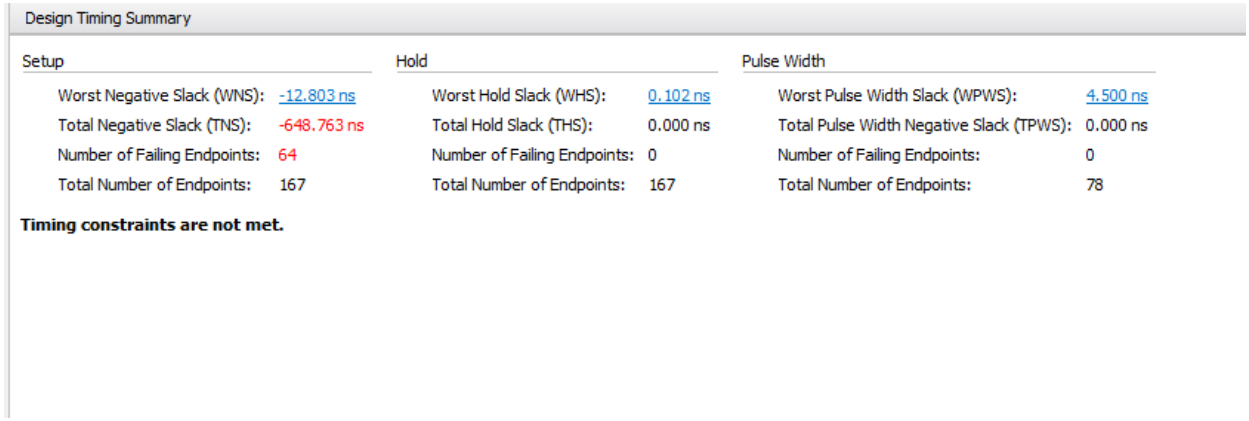

Figure 7: Timing constraints of pure VHDL implementation

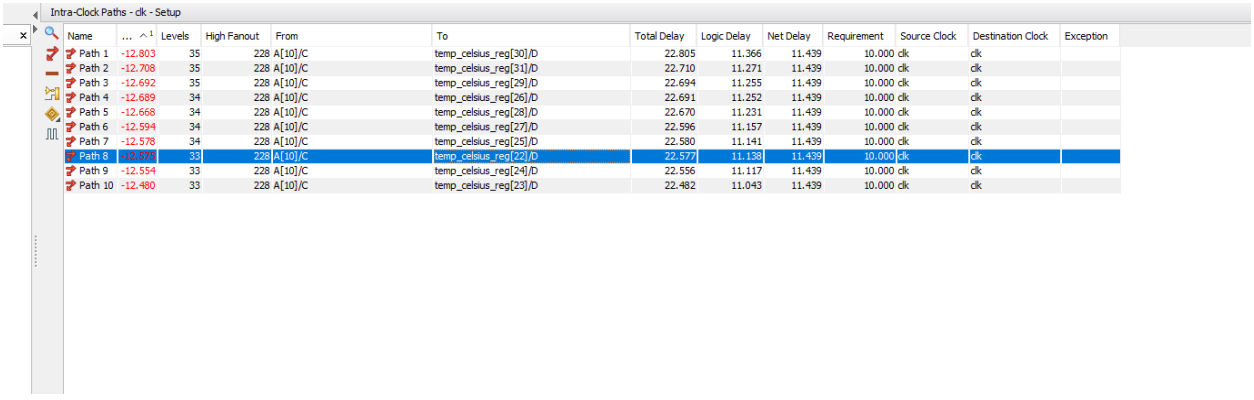

Figure 8: Bottlenecks of pure VHDL implementation

Otherwise, due to the simplicity of the program there will be half as many LUTs (Figure 9: 1243 vs. 2931). In addition, there is no need to use BRAM which takes up a lot of utilization (49%) and MMCM (17%) when only using VHDL. We see similar results to the hardware implementation and the temperature is accurate with minor rounding errors in Figure 10.

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	1243	63400	1.96
FF	76	126800	0.06
IO	24	210	11.43
BUFG	1	32	3.13

Figure 9: Post-Implementation Utilization of the Pure VHDL design

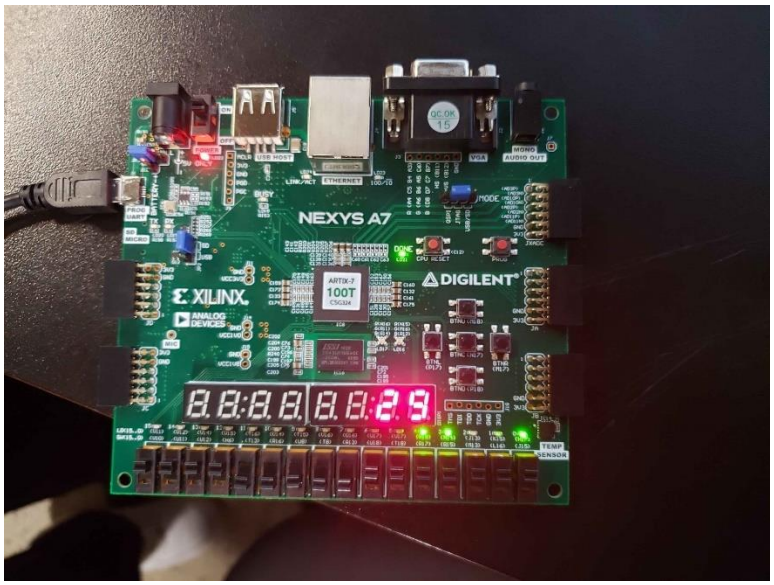

Figure 10: Pure VHDL output on the Nexys 7

Additionally, we can verify that the correct temperature readings are calculated in our state machine testbench.

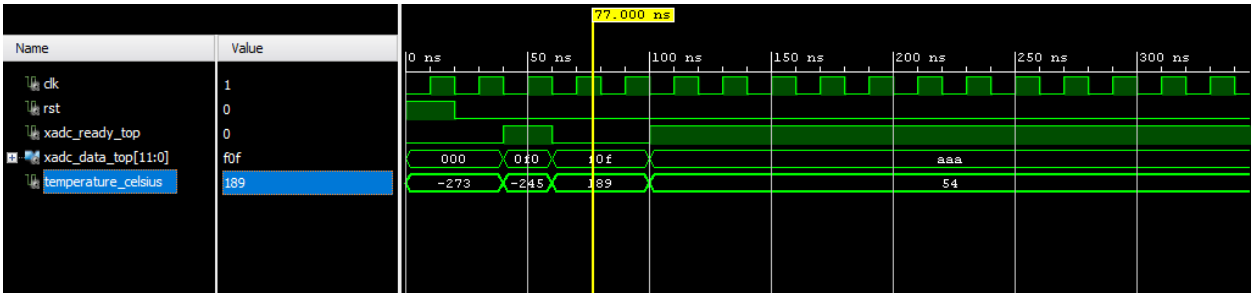

Figure 11: Simulation Waveform of Testbench

As seen in Figure 11, the reset value is -273 °C, which aligns with our expected value with minimal error. The next value is completely infeasible as a temperature therefore we end up with a negative value, however the next two indicate that the

temperature value aligns with our input. Afterwards, we tested the temperature of 36 °C in raw ADC and instead our result was 26 °C. We tweaked the scale factor from 12 to 13 and got around 48 °C. Once again, the scale factor would normally be decimal of around 12.6 so we can interpolate that our values are still correct with minimal rounding error.

Conclusion:

In conclusion, this project successfully demonstrates the ability of two MicroBlaze processors on an FPGA to collaboratively monitor and process temperature data, showcasing the effectiveness of hardware-software co-design in embedded systems. By leveraging the flexibility of the AXI interface and the deterministic nature of pure VHDL implementations, we explored the advantages and trade-offs between these approaches. The co-design approach provided a resource-efficient, scalable, and real-time monitoring system with low utilization rates and adherence to timing constraints. In contrast, the pure VHDL implementation highlighted a hardware-centric, deterministic alternative suitable for specific applications requiring minimal software intervention. Together, these methodologies emphasize the versatility of FPGA technology in creating innovative, high-performance embedded systems capable of addressing practical challenges like temperature monitoring and system safety.

References:

Sysmon:
[1] https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/sysmon/src/xsysmon.h
Lab 1:
[2] https://courselink.uoguelph.ca/d2l/le/content/895729/viewContent/3919723/View

Lab 5:
[3] https://courselink.uoguelph.ca/d2l/le/content/895729/viewContent/3923095/View

Xilinx:

[4] https://www.cmc.ca/xilinx-ise-vivado/

Microblaze:

[5] https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/microblaze.html

Appendices

MICROBLAZE C CODE:

```
#include "xparameters.h"
#include "xsysmon.h"
#include "xil_printf.h"
#include "xil_printf.h"
#define SYSMON_DEVICE_ID XPAR_SYSMON_0_DEVICE_ID
XSysMon SysMonInst;
int main() {
    int Status;
    float Temperature;
    XSysMon_Config *ConfigPtr;
    xil_printf("Starting SysMon Temperature Monitoring\n");
    ConfigPtr = XSysMon_LookupConfig(SYSMON_DEVICE_ID);
    if (ConfigPtr == NULL) {
        xil_printf("SysMon LookupConfig Failed\n");
        return XST_FAILURE;
    }
    Status = XSysMon_CfgInitialize(&SysMonInst, ConfigPtr, ConfigPtr->BaseAddress);
    if (Status != XST_SUCCESS) {
        xil_printf("SysMon Initialization Failed\n");
        return XST_FAILURE;
    }
    xil_printf("SysMon Initialized Successfully\n");
    Status = XSysMon_SelfTest(&SysMonInst);
    if (Status != XST_SUCCESS) {
        xil_printf("SysMon Self Test Failed\n");
        return XST_FAILURE;
    }
    xil_printf("SysMon Self Test Passed\n");
    XSysMon_SetSequencerMode(&SysMonInst, XSM_SEQ_MODE_SAFE);
    while (XSysMon_GetStatus(&SysMonInst) & XSM_SR_BUSY_MASK);
    xil_printf("SysMon Sequencer Mode Set\n");
    while (1) {
        u32 RawTempData = XSysMon_GetAdcData(&SysMonInst, XSM_CH_TEMP);
        xil_printf("Raw Temp Data: 0x%08X\n", RawTempData);
        Temperature = XSysMon_RawToTemperature(RawTempData);
        //xil_printf("Temperature: %0.2f °C\n", Temperature);
        int TempInt = (int)(Temperature * 100);  // Multiply by 100 for two decimal places
        xil_printf("Temperature: %d.%02d °C\n", TempInt / 100, TempInt % 100);
        for (volatile int i = 0; i < 10000000; i++);
    }
    return XST_SUCCESS;
}
```

PURE VHDL CODE XADC:

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

use IEEE.std_logic_unsigned.all;
use IEEE.NUMERIC_STD.ALL;

entity XADCDemo is
   Port (
        clk     : in  STD_LOGIC;
        data_out : out STD_LOGIC_VECTOR (7 downto 0);
        AN : out STD_LOGIC_VECTOR(7 downto 0);
        Cathode : out STD_LOGIC_VECTOR(6 downto 0)
       );
end XADCDemo;

architecture Behavioral of XADCDemo is

signal sum : STD_LOGIC_VECTOR(6 downto 0); --changed range
signal carry_out : STD_LOGIC;

signal lowBits : STD_LOGIC_VECTOR(3 downto 0);
signal highBits : STD_LOGIC_VECTOR(3 downto 0);

signal lowSegmentDis : STD_LOGIC_VECTOR (6 downto 0);
signal highSegmentDis : STD_LOGIC_VECTOR (6 downto 0);

signal num : STD_LOGIC := '0';
signal counter : integer range 0 to 32000 := 0;

COMPONENT xadc_wiz_0
  PORT (
    di_in      : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
    daddr_in   : IN  STD_LOGIC_VECTOR(6 DOWNTO 0);
    den_in     : IN  STD_LOGIC;
    dwe_in     : IN  STD_LOGIC;
    drdy_out   : OUT STD_LOGIC;
    do_out     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    dclk_in    : IN  STD_LOGIC;
    vp_in      : IN  STD_LOGIC;
    vn_in      : IN  STD_LOGIC;
```

```
   reset_in    : IN  STD_LOGIC;
   channel_out : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)
  );
END COMPONENT;

constant chan_temp_addr : std_logic_vector(4 downto 0) := "00000"; -- Temperature
channel address

signal di_in      : std_logic_vector(15 downto 0) := (others => '0');
signal daddr_in    : std_logic_vector(6 downto 0);
signal den_in      : std_logic := '0';
signal dwe_in      : std_logic := '0';
signal drdy_out    : std_logic;
signal do_out      : std_logic_vector(15 downto 0);
signal temp_data_reg: std_logic_vector(11 downto 0);
signal channel_out  : std_logic_vector(4 downto 0);

-- Signal for temperature in Celsius
signal temp_celsius : integer := 0;

begin

xadc_inst : xadc_wiz_0
  PORT MAP (
   di_in      => di_in,
   daddr_in   => daddr_in,
   den_in     => den_in,
   dwe_in     => dwe_in,
   drdy_out   => drdy_out,
   do_out     => do_out,
   dclk_in    => clk,
   vp_in      => '0',
   vn_in      => '0',
   reset_in   => '0',
   channel_out => channel_out
  );

process(clk)
   constant SCALE_FACTOR : integer := 12; -- (503.975 * 100) / 4096
   constant OFFSET_CELSIUS : integer := 273; -- 273.15 scaled as an integer
   variable adc_scaled : integer; -- Intermediate temperature calculation
   variable counter : integer range 0 to 5000 := 0; -- Counter for multiplexing display
begin
   if rising_edge(clk) then
```

```
      -- Enable XADC conversions
      den_in <= '1';

      -- Capture temperature data when ready
      if drdy_out = '1' then
        temp_data_reg <= do_out(15 downto 4); -- Capture the temperature data

        -- Convert ADC Code to Celsius
        adc_scaled := (to_integer(unsigned(temp_data_reg)) * SCALE_FACTOR) / 100; --
Scale down
        temp_celsius <= adc_scaled - OFFSET_CELSIUS;
      end if;

      -- Multiplexing logic for seven-segment display
      if counter = 5000 then -- 5 ms interval for digit switching
        if num = '0' then
          num <= '1';
          AN <= "11111110";  -- Activate rightmost digit
          Cathode <= lowSegmentDis;
        else
          num <= '0';
          AN <= "11111101";  -- Activate second rightmost digit
          Cathode <= highSegmentDis;
        end if;
        counter := 0;
      else
        counter := counter + 1;
      end if;
    end if;
end process;

-- Assign the converted temperature data to the seven-segment decoder inputs
lowBits <= std_logic_vector(to_unsigned(temp_celsius mod 10, 4));  -- Lower nibble
(ones place)
highBits <= std_logic_vector(to_unsigned(temp_celsius / 10, 4));  -- Upper nibble (tens
place)

-- Instantiate the seven-segment decoders
decode_low : entity work.BCD_7SegDecoder port map (
    Input_7SD => lowBits,
    a_to_g => lowSegmentDis
);

decode_high : entity work.BCD_7SegDecoder port map (
```

```
   Input_7SD => highBits,
   a_to_g => highSegmentDis
);

-- Address the temperature channel
daddr_in <= "00" & chan_temp_addr;

-- Output the temperature data (truncated to 8 bits for debugging or external use)
data_out <= std_logic_vector(to_unsigned(temp_celsius, 8));

end Behavioral;
```

CONSTRAINTS:

```
## Clock signal
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -add -name sys_clk_pin - period 10.00 -waveform {0 5} [get_ports { clk }];
## Add timing constraints if needed

set_property -dict { PACKAGE_PIN P18 IOSTANDARD LVCMOS33 } [get_ports { reset }];
set_property -dict { PACKAGE_PIN M18 IOSTANDARD LVCMOS33 } [get_ports { state }];
# Orginally H6

# data_out (data_outED[8:0])
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports
{ data_out[0] }];   # Orginally R18
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports
{ data_out[1] }];   # Orginally V17
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports
{ data_out[2] }];   # Orginally U17
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports
{ data_out[3] }];   # Orginally U16
set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports
{ data_out[4] }];   # Orginally V16
set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports
{ data_out[5] }];   # Orginally T15
set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports
{ data_out[6] }];   # Orginally U14
set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports
{ data_out[7] }];   # Orginally T16

#7 segment display
set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports
{ Cathode[0] }];
```

```
set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports
{ Cathode[1] }];
set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports
{ Cathode[2] }];
set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports
{ Cathode[3] }];
set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports
{ Cathode[4] }];
set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports
{ Cathode[5] }];
set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports
{ Cathode[6] }];

set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { AN[0] }];
set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { AN[1] }];
set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { AN[2] }];
set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { AN[3] }];
set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { AN[4] }];
set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { AN[5] }];
set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports { AN[6] }];
set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { AN[7] }];

# Create a clock with a 100 MHz frequency (10 ns period)
create_clock -period 10.000 -name clk -waveform {0.000 5.000} [get_ports clk]

# Output delay constraints for data_out, AN, and Cathode
set_output_delay -clock [get_clocks clk] -min -add_delay 1.000 [get_ports {data_out[*]}]
set_output_delay -clock [get_clocks clk] -max -add_delay 4.000 [get_ports {data_out[*]}]
set_output_delay -clock [get_clocks clk] -min -add_delay 1.000 [get_ports {AN[*]}]
set_output_delay -clock [get_clocks clk] -max -add_delay 4.000 [get_ports {AN[*]}]
set_output_delay -clock [get_clocks clk] -min -add_delay 1.000 [get_ports {Cathode[*]}]
set_output_delay -clock [get_clocks clk] -max -add_delay 4.000 [get_ports {Cathode[*]}]

# Set the OFFCHIP_TERM property for all output ports
set_property OFFCHIP_TERM NONE [get_ports {data_out[*]}]
set_property OFFCHIP_TERM NONE [get_ports {AN[*]}]
set_property OFFCHIP_TERM NONE [get_ports {Cathode[*]}]
```

SEVEN SEGMENT DECODER:

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

entity BCD_7SegDecoder is
```

```
    Port ( Input_7SD : in STD_LOGIC_VECTOR (3 downto 0);
           a_to_g : out STD_LOGIC_VECTOR (6 downto 0));
end BCD_7SegDecoder;

architecture Behavioral of BCD_7SegDecoder is

begin
  process(Input_7SD)
  begin
    case Input_7SD is                       -- a,b,c,d,e,f,g
       when "0000" => a_to_g <= "1000000"; --0
       when "0001" => a_to_g <= "1111001"; --1      1001111       1111001
       when "0010" => a_to_g <= "0100100"; --2    0010010  0100100
       when "0011" => a_to_g <= "0110000"; --3
       when "0100" => a_to_g <= "0011001"; --4
       when "0101" => a_to_g <= "0010010"; --5
       when "0110" => a_to_g <= "0000010"; --6
       when "0111" => a_to_g <= "1011000"; --7
       when "1000" => a_to_g <= "0000000"; --8
       when "1001" => a_to_g <= "0010000"; --9
       when "1010" => a_to_g <= "0001000"; --A
       when "1011" => a_to_g <= "0000011"; --b
       when "1100" => a_to_g <= "1000110"; --C
       when "1101" => a_to_g <= "0100001"; --d
       when "1110" => a_to_g <= "0000110"; --E
       when "1111" => a_to_g <= "0001110"; --F
    end case;
  end process;
end Behavioral;
```

STATE MACHINE CODE:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 11/29/2024 02:41:22 PM
-- Design Name: XADC Top Level Simulation
-- Module Name: xadc_toplevel - Behavioral (No MicroBlaze)
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
```

```
--  This is a top-level module that integrates a XADC, a state machine, and
--  simulates the behavior of XADC data handling without involving MicroBlaze
processors.
-- Dependencies:
--  None
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------

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

-- Entity declaration for the top-level module without MicroBlaze
entity xadc_toplevel is
  Port (
    clk               : in STD_LOGIC;       -- Clock input
    rst               : in STD_LOGIC;       -- Reset signal
    -- XADC signals (read-only in the top-level module)
    xadc_ready_top        : in STD_LOGIC;       -- XADC data ready signal (input)
    xadc_data_top         : in STD_LOGIC_VECTOR(11 downto 0)  -- Fake XADC data (12-bit)
(input)
  );
end xadc_toplevel;

-- Architecture definition for the top-level module without MicroBlaze
architecture Behavioral of xadc_toplevel is

  -- State machine states
  type state_type is (IDLE, WAIT_FOR_XADC, SEND_DATA);
  signal current_state, next_state : state_type;

  -- Internal signal to hold data
  signal xadc_data_reg : STD_LOGIC_VECTOR(11 downto 0);

  -- XADC internal signals (simulated here)
  signal xadc_ready_internal : STD_LOGIC := '0';
  signal xadc_data_internal : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');

begin
```

```
-- XADC simulation logic (replacing the separate component)
process(clk, rst)
begin
 if rst = '1' then
  xadc_ready_internal <= '0';
  xadc_data_internal <= (others => '0');
 elsif rising_edge(clk) then
  -- Simulate XADC behavior
  if xadc_ready_top = '1' then
   xadc_data_internal <= xadc_data_top;  -- Read the XADC data
   xadc_ready_internal <= '1';         -- Indicate that the XADC data is ready
  else
   xadc_ready_internal <= '0';         -- Indicate that the XADC is not ready
  end if;
 end if;
end process;

-- State machine process
process(clk, rst)
begin
 if rst = '1' then
  current_state <= IDLE;
  xadc_data_reg <= (others => '0');
 elsif rising_edge(clk) then
  current_state <= next_state;
 end if;
end process;

-- State machine logic
process(current_state, xadc_ready_internal)
begin
 -- Default output values
 next_state <= current_state;

 case current_state is
  when IDLE =>
   -- Wait for XADC data to be ready
   if xadc_ready_internal = '1' then
    next_state <= WAIT_FOR_XADC;
   end if;

  when WAIT_FOR_XADC =>
   -- Store the XADC data when it's ready
```

```
      if xadc_ready_internal = '1' then
        xadc_data_reg <= xadc_data_internal;  -- Store the XADC data
        next_state <= SEND_DATA;
      end if;

    when SEND_DATA =>
      -- Output the stored XADC data or send it to uart
      next_state <= IDLE;

    when others =>
      next_state <= IDLE;
  end case;
 end process;

end Behavioral;
```

STATE MACHINE TESTBENCH:

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 11/29/2024 02:41:22 PM
-- Design Name: XADC Top Level Simulation Testbench
-- Module Name: xadc_toplevel_tb
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--  This is a testbench for the `xadc_toplevel` module. It simulates the XADC
--  behavior and tests the state machine logic without involving MicroBlaze processors.
--  This version includes temperature conversion from XADC data to Celsius.
-- Dependencies:
--  None
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;  -- Use NUMERIC_STD for unsigned conversion
```

```
entity xadc_toplevel_tb is
end xadc_toplevel_tb;

architecture Behavior of xadc_toplevel_tb is

  -- Signal declaration for the unit under test (UUT)
  signal clk               : STD_LOGIC := '0';  -- Clock signal
  signal rst               : STD_LOGIC := '0';  -- Reset signal
  signal xadc_ready_top        : STD_LOGIC := '0';  -- XADC ready signal
  signal xadc_data_top         : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');  --
XADC data (12-bit)

  -- Signal to hold the calculated temperature in Celsius (using integer for calculations)
  signal temperature_celsius    : integer := 0;

  -- Component declaration for the UUT (Unit Under Test)
  component xadc_toplevel
    Port (
      clk               : in STD_LOGIC;
      rst               : in STD_LOGIC;
      xadc_ready_top        : in STD_LOGIC;
      xadc_data_top         : in STD_LOGIC_VECTOR(11 downto 0)
    );
  end component;

begin

  -- Instantiate the Unit Under Test (UUT)
  uut: xadc_toplevel
    Port map (
      clk               => clk,
      rst               => rst,
      xadc_ready_top        => xadc_ready_top,
      xadc_data_top         => xadc_data_top
    );

  -- Clock generation process (50MHz clock)
  clk_process: process
  begin
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
```

```
end process;

-- Temperature conversion process
temperature_process: process(xadc_data_top)
  variable xadc_value : unsigned(11 downto 0);  -- Variable to store converted XADC data
  constant SCALE_FACTOR : integer := 12; -- (503.975 * 100) / 4096
  constant OFFSET_CELSIUS : integer := 273; -- 273.15 scaled as an integer
  variable adc_scaled : integer; -- Intermediate temperature calculation
begin
  -- Convert xadc_data_top (STD_LOGIC_VECTOR) to unsigned
  xadc_value := unsigned(xadc_data_top); -- Correct conversion to unsigned

  -- Scale the ADC value
  adc_scaled := (to_integer(xadc_value) * SCALE_FACTOR) / 100; -- Scale down

  -- Now subtract OFFSET_CELSIUS, ensuring both operands are integers
  temperature_celsius <= adc_scaled - OFFSET_CELSIUS;

  -- Output the temperature value for simulation purposes (optional)
  report "Temperature: " & integer'image(temperature_celsius) & " °C";
end process;

-- Stimulus process to apply signals to the UUT
stimulus_process: process
begin
  -- Apply reset
  rst <= '1';
  wait for 20 ns;
  rst <= '0';

  -- Simulate XADC data ready with some values
  wait for 20 ns;
  xadc_ready_top <= '1';  -- XADC is ready
  xadc_data_top <= "000011110000";

  wait for 20 ns;
  xadc_ready_top <= '0';  -- XADC is not ready
  xadc_data_top <= "111100001111";

  wait for 40 ns;
  xadc_ready_top <= '1';  -- XADC is ready again
  xadc_data_top <= "101010101010";

  -- End simulation after a while
```

```
    wait for 100 ns;

    -- Stop the simulation gracefully by waiting indefinitely
    wait;
  end process;

end Behavior;
```