Name: _____ ID: _____ Start Date: Wednesday, August 18, 2021
Name: _____ ID: _____ Due Date: Friday, August 20, 2021

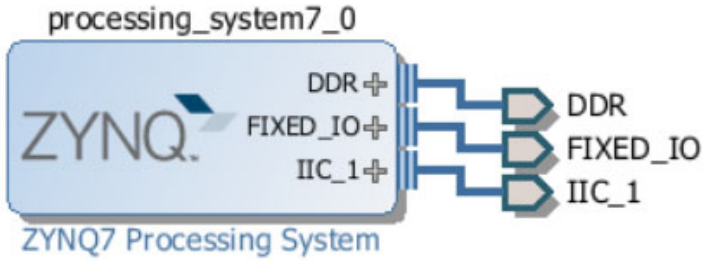
Software and Hardware Co-Design with Zybo, Summer 2021 HUST

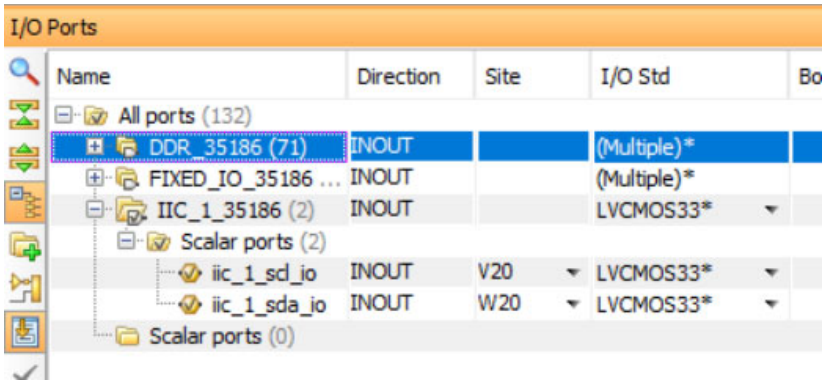
Lab #8: Reading Two TMP101 Temperature Sensors from MIO for One and SelectIO for the Other of Zybo

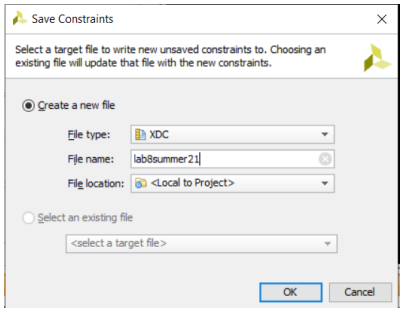
This is a group lab. Each group of two students should demonstrate your I2C TMP101 reading implementation on Zybo board and submit one pdf copy of report for this lab. Late lab submission will be accepted with a grade reduction of 10% for each day that it is late.

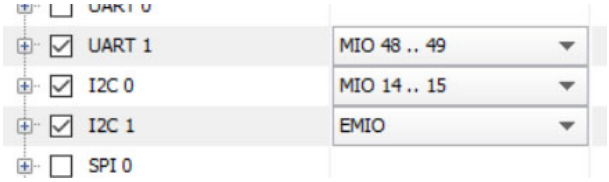
I. Objectives

(1) *Construct a Zynq 7010 circuit to implement the two I2C interfaces with the I2C controllers I2C 0 and I2C 1: I2C 0 on MIO pins 14 and 15 and I2C 1on SelectIO pins JB3 and JB4. Your block design should have just one I2C interface connection for I2C Module 1 of Zynq. This interface needs to be connected to two Select I/O pins on Zybo. The following example connects SCL pin to JB3 (pin V20) and SDA pin to JB4 (pin W20) of Pmod JB connector. You are free to use other Pmod connector pins manually through I/O Ports menu if you want to do so. A XDC file will need to be created to save this assignment.*
You should verify that your I2C Module 0 is connected to MIO pins 14 and 15, which are accessible from Pmod JF of Zybo.







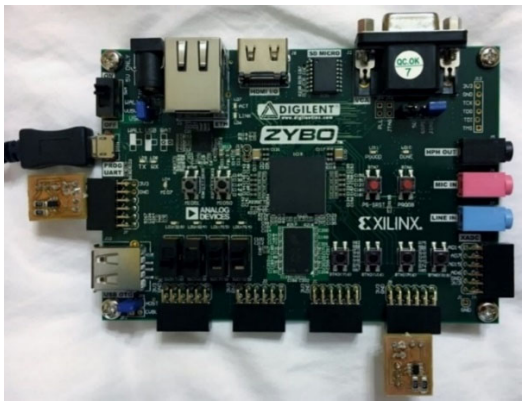


(1) *Connect two TMP101 breakout boards on Zybo so that they can be read by Zynq 7010 and their temperatures. You are free to set their addresses.*

(2) *Develop a C program for ARM I2C modules to configure the TMP101 temperature sensors, one in 10-bit resolution and the other in 12-bit resolution and to read the TMP101 temperatures and display their temperatures in both Celsius and Fahrenheit on the series terminal of the SDK with the correct resolutions. You will need to configure your TMP101 sensors to transmit two byte temperature values of 12 bits so that the temperatures will have four digits after decimal point.*

(3) *Use printf() for floating point display because xil_printf() does not support floating point. Format for floating point printing is 8.4f, where 8 is the total number of digits and 4 is the number of digits after decimal point. See an I2C example program from Xilinx in the appendix of this handout. This program can be used to test your hardware.*

(4) *Submit a memo to explain how you have implemented and verified your design and implementation. Show, with screen captures, which temperature sensor has 10-bit resolution and which one has 12-bit resolution. Include relevant screen captures and your source code in your memo.*



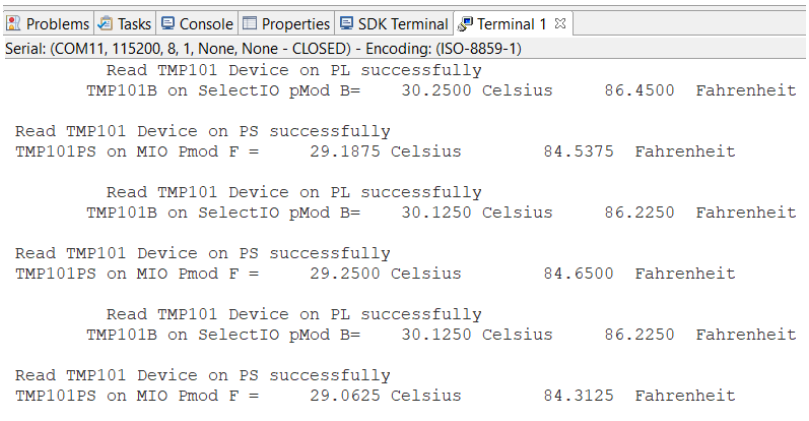
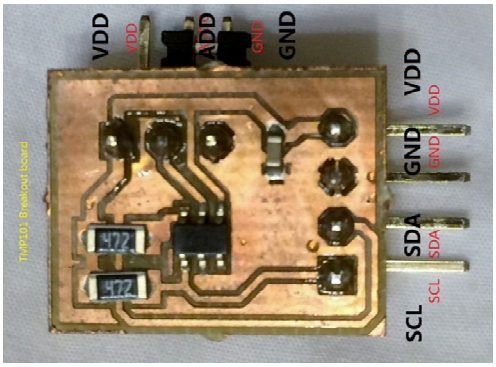


Figure 1. Zybo board with two TMP101 temperature sensors on Port F and Port B. UART 1 on Zynq must be selected to be on MIO 48 and 49.

The TMP101 breakout board in Port F is connected to the PS through MIO pins and is connected to Port F are as follows: SDK on JF9 (M14) and SDA on JF10 (M15). The TMP101 on Port B is connected to the PS through SelectIO pins and is connected so that JB3 (V20) is SCL and JB4 (W20) is SDA.



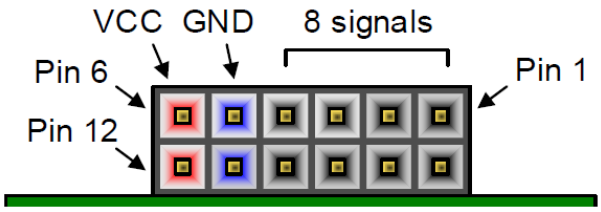
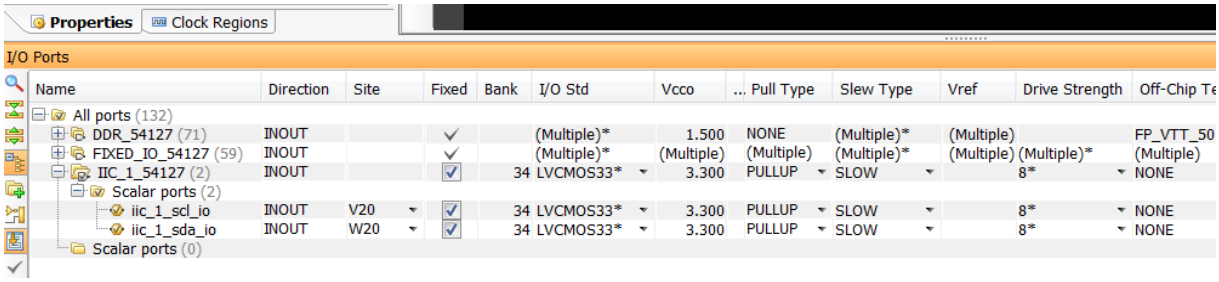


Figure 16. Pmod diagram.

Pmod JA (XADC)	Pmod JB (Hi-Speed)	Pmod JC (Hi-Speed)	Pmod JD (Hi-Speed)	Pmod JE (Std.)	Pmod JF (MIO)
JA1: N15	JB1: T20	JC1: V15	JD1: T14	JE1: V12	JF1: MIO-13
JA2: L14	JB2: U20	JC2: W15	JD2: T15	JE2: W16	JF2: MIO-10
JA3: K16	JB3: V20	JC3: T11	JD3: P14	JE3: J15	JF3: MIO-11
JA4: K14	JB4: W20	JC4: T10	JD4: R14	JE4: H15	JF4: MIO-12
JA7: N16	JB7: Y18	JC7: W14	JD7: U14	JE7: V13	JF7: MIO-0
JA8: L15	JB8: Y19	JC8: Y14	JD8: U15	JE8: U17	JF8: MIO-9
JA9: J16	JB9: W18	JC9: T12	JD9: V17	JE9: T17	JF9: MIO-14
JA10: J14	JB10: W19	JC10: U12	JD10: V18	JE10: Y17	JF10: MIO-15

Assign Pin V20 to I2C 1 clock pin and Pin W20 to I2C 1 data pin.



II. Software Development

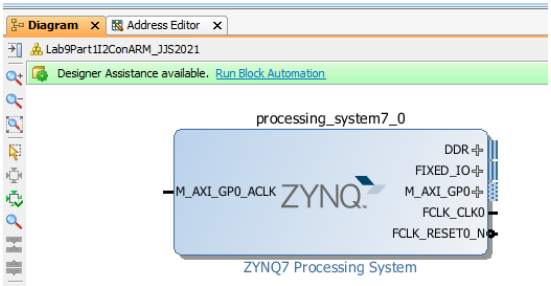
Use the I2C example program in the appendix as a starting point to read two I2C TMP101 sensors with different addresses. One sensor is configured to have 12-bit resolution and the other 10-bit resolution. Display their temperatures in both Celsius and Fahrenheit are displayed on the series terminal of the SDK.

I. Good Block Design Practice

Project file name should not be more than 256 characters and should not contain any special characters such as space, dash, etc.

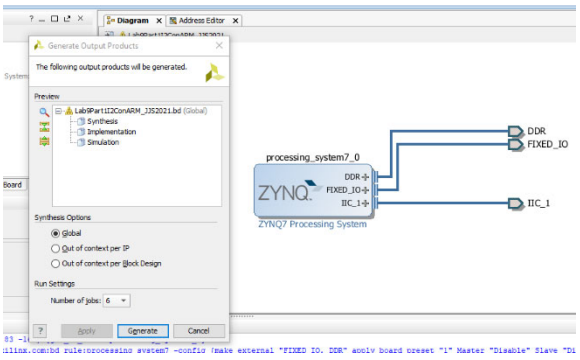
I.1 Run Block Automation Right Away

After adding a zynq block to your block design, always click Run Block Automation to make sure your UART 1 is configured correctly. Do not manually change UART 1 configuration in Peripheral I/O Pins configuration menu.



I.2 Run Generate Output Products after Block Design Change

Always run Generate Output Products after each block design change and before creating the HDL wrapper. Sometimes it is better to remove the HDL wrapper before running Generate Output Products. This is to make sure all software drivers are associated correctly with hardware IP components.



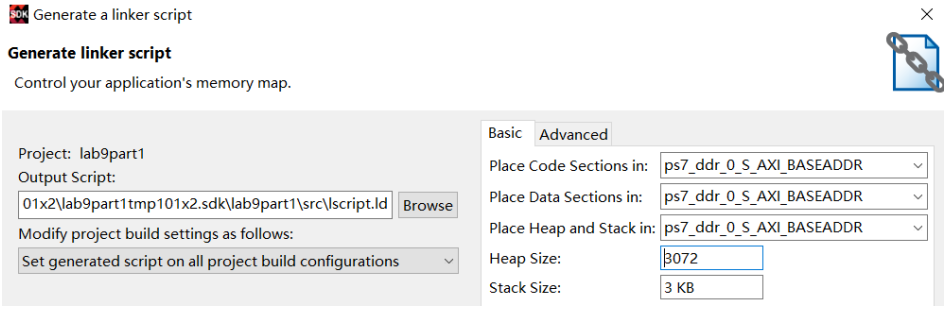
II. Possible Errors

II.1 printf() and xil_printf() should not be used together

printf() and xil_printf() are independent of each other. If they are used together, xil_printf() may display faster than printf() even if printf() is called first. Therefore, the text on display may seem to be out of order.

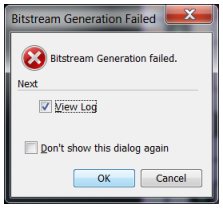
II.2 printf() errors

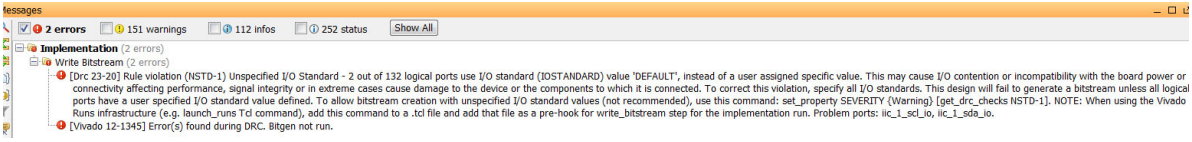
Printf() uses stack to operate. If the stack size is too small, printf() may work strangely. Make sure the stack size is at last 3KB.



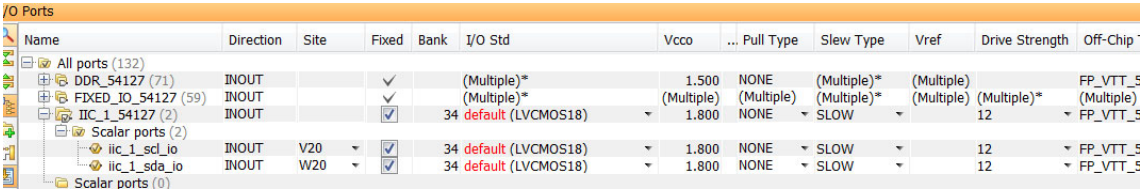
II.3 An Error on I/O Standard.

If you see the following error, change the I/O standard for pin V20 and W20 to LVCMOS33.

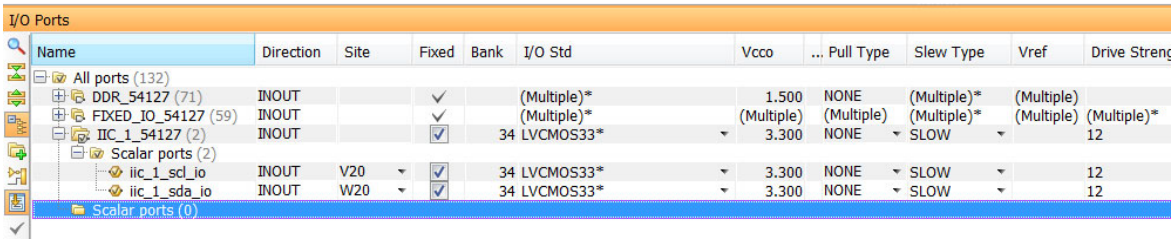




The default I/O standard is LVCMOS18 (1.8V) but the required I2C standard should be 3.3V.

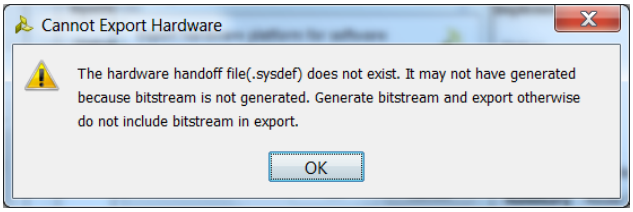


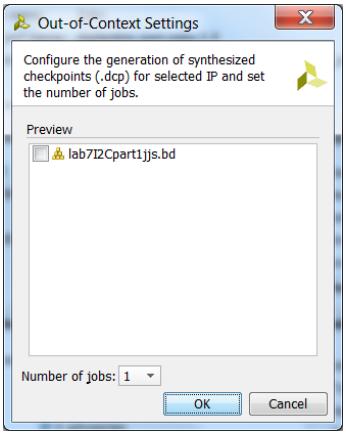
Here is the correct I/O standard: LVCOMS33



II.4 Out of Context settings error: the hardware handoff (.sysdef) does not exist

This error occurs when out of context settings is checked. Disabling “out of context settings” would fix this error.


Cannot Export Hardware
The hardware handoff file(.sysdef) does not exist. It may not have generated because bitstream is not generated. Generate bitstream and export otherwise do not include bitstream in export.
OK


Out-of-Context Settings
Configure the generation of synthesized checkpoints (.dcp) for selected IP and set the number of jobs.
Preview
lab7I2Cpart1jjs.bd
Number of jobs: 1
OK
Cancel

III. Appendix xiicps_polled_master_example.c

The following example, xiicps_polled_master_example.c is from C:\Xilinx\SDK\2016.2\data\embeddedsw\XilinxProcessorIPLib\drivers\iicps_v3_2\examples.

```
/* Copyright (C) 2010 - 2014 Xilinx, Inc.  All rights reserved. */
/**
* @file xiicps_polled_master_example.c
* A design example of using the device in polled mode as master.
* The example uses buffer size 132. Please set the send buffer of the
* Aardvark device to be continuous 64 bytes from 0x00 to 0x3F.
* <pre> MODIFICATION HISTORY:
* Ver   Who Date     Changes
* ----- --- -------- -----------------------------------------------
* 1.00a jz  01/30/10 First release
*/
#include "xparameters.h"
#include "xiicps.h"
#include "xil_printf.h"
/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user
 * can easily change all the needed parameters in one place.
 */
#define IIC_DEVICE_ID                   XPAR_XIICPS_0_DEVICE_ID

/* The slave address to send to and receive from.*/
#define IIC_SLAVE_ADDR                  0x55
#define IIC_SCLK_RATE                   100000

/*The following constant controls the length of the buffers to be sent
 * and received with the IIC.  */
#define TEST_BUFFER_SIZE    132

/****************** Function Prototypes ******************/

int IicPsMasterPolledExample(u16 DeviceId);
/******************** Variable Definitions ***************/

XIicPs Iic;                 /**< Instance of the IIC Device */

/* The following buffers are used in this example to send and
 * receive data with the IIC. */
u8 SendBuffer[TEST_BUFFER_SIZE];  /**< Buffer for Transmitting Data */
u8 RecvBuffer[TEST_BUFFER_SIZE];  /**< Buffer for Receiving Data */

/*****************************************************/
* Main function to call the polled master example.
* @param        None.
* @return       XST_SUCCESS if successful, XST_FAILURE if unsuccessful.
* @note                 None.
*****************************************************/
int main(void)
{
        int Status;

        xil_printf("IIC Master Polled Example Test \r\n");
```

```
/* Run the Iic polled example in master mode, specify the Device
 * ID that is specified in xparameters.h. */
	Status = IicPsMasterPolledExample(IIC_DEVICE_ID);
	if (Status != XST_SUCCESS) {
		xil_printf("IIC Master Polled Example Test Failed\r\n");
		return XST_FAILURE;
	}

xil_printf("Successfully ran IIC Master Polled Example Test\r\n");
	return XST_SUCCESS;
}

/*****************************************************/
* This function sends data and expects to receive data from
* slave as modular of 64.
* This function uses interrupt-driven mode of the device.
* @param	DeviceId is the Device ID of the IicPs Device and is the
*		XPAR_<IICPS_instance>_DEVICE_ID value from xparameters.h
* @return	XST_SUCCESS if successful, otherwise XST_FAILURE.
* @note		None.
*****************************************************/
int IicPsMasterPolledExample(u16 DeviceId)
{
	int Status;
	XIicPs_Config *Config;
	int Index;

/* Initialize the IIC driver so that it's ready to use
 * Look up the configuration in the config table, then initialize it*/
	Config = XIicPs_LookupConfig(DeviceId);
	if (NULL == Config) {
		return XST_FAILURE;
	}
	Status = XIicPs_CfgInitialize(&Iic, Config, Config->BaseAddress);
	if (Status != XST_SUCCESS) {
		return XST_FAILURE;
	}

/* Perform a self-test to ensure that the hardware
 * was built correctly. */
	Status = XIicPs_SelfTest(&Iic);
	if (Status != XST_SUCCESS) {
		return XST_FAILURE;
	}

/* Set the IIC serial clock rate. */
	XIicPs_SetSClk(&Iic, IIC_SCLK_RATE);

/* Initialize the send buffer bytes with a pattern to send and the
 * the receive buffer bytes to zero to allow the receive data to be
 * verified. */
	for (Index = 0; Index < TEST_BUFFER_SIZE; Index++) {
		SendBuffer[Index] = (Index % TEST_BUFFER_SIZE);
		RecvBuffer[Index] = 0;
	}
/* Send the buffer using the IIC and ignore the number of bytes sent
 * as the return value since we are using it in interrupt mode. */
	Status = XIicPs_MasterSendPolled(&Iic, SendBuffer,
```

```
                        TEST_BUFFER_SIZE, IIC_SLAVE_ADDR);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }
/* Wait until bus is idle to start another transfer. */
        while (XIicPs_BusIsBusy(&Iic)) {
                /* NOP */
        }
        Status = XIicPs_MasterRecvPolled(&Iic, RecvBuffer,
                        TEST_BUFFER_SIZE, IIC_SLAVE_ADDR);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }
/* Verify received data is correct. */
         for(Index = 0; Index < TEST_BUFFER_SIZE; Index ++) {

/* Aardvark as slave can only set 64 bytes for output */
                if (RecvBuffer[Index] != Index % 64) {
                        return XST_FAILURE;
                }
        }
        return XST_SUCCESS;
}
```