Arty MicroBlaze Soft Processing System Implementation Tutorial II

Daniel Wimberly, Sean Coss

***Abstract*—The Microblaze soft processor was set up on the Arty Artix-7 FPGA Evaluation board to read analog and digital signals using the Artix-7's built-in analog-to-digital converter (ADC) and its digital input pins. The Microblaze configuration was first set up using Xilinx's Vivado, then the C program to demonstrate the ADC and digital inputs was run using Xilinx's SDK. The output of each signal is sent through the USART port on the board to be displayed on a PC terminal.**

I. Arty Board and MicroBlaze

The board used for the project was the Arty board which is a development board built around the Artix-7 FPGA. This was developed by Xilinx for use with the MicroBlaze soft processor which is an HDL defined processor which can be written to the Artix-7 FPGA. The evaluation board provides connectivity such as switches, buttons, LEDs, RGB LEDs, Pmod connectors, shield connectors, USB, and Ethernet to work with HDL components and those defined through the MicroBlaze.

II. ADC Hardware Setup

This procedure builds on the implementation performed in the previous project report, titled Arty MicroBlaze Soft Processing System Implementation Tutorial. The process for the hardware implementation is highlighted below:

- Open the previous project from Vivado SDK
- Add the XADC Wizard block to the project
- Add an interrupt controller to the project
- Add an interrupt concatenation to the project
- Add GPIO blocks to the project
- Adjust added block settings
- Add GPIO pins to ADC block
- Create block diagram wrapper
- Create pin constrains for analog pins
- Generate and Export bitstream

The XADC Wizard IP was first added (Figure 1). This block is the ADC controller - it initializes the ADC and allows it to be used with internal and external analog inputs. Internal analog inputs include the internal temperature reference, internal voltage monitors, and several other system health monitors. External pins are also connected to the ADC, and these are what were used in the implementation.

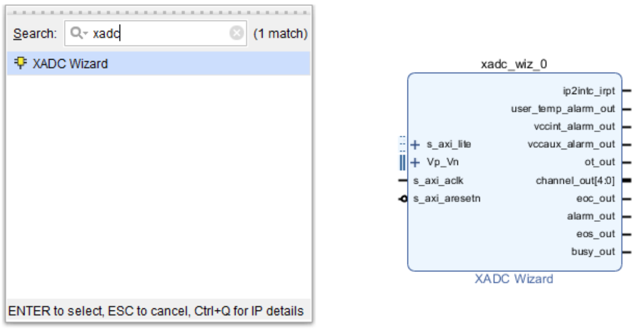
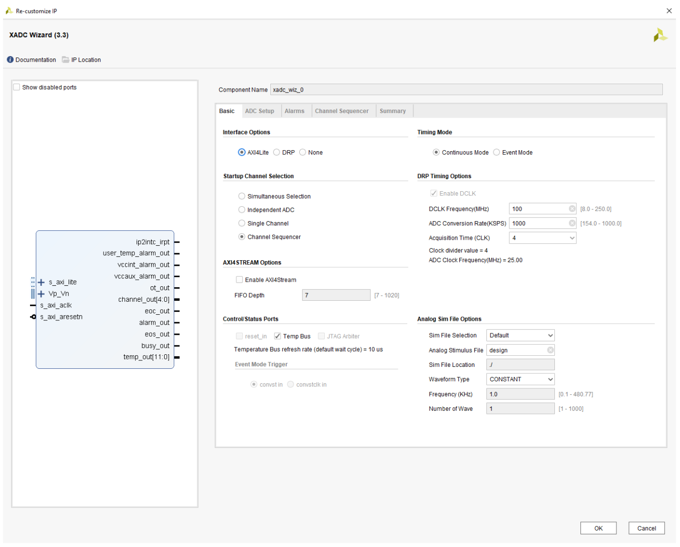


Fig. 1. Add XADC Wizard

The XADC Wizard has a large number of configuration options, which can be set up both on the hardware and software side. Here, the following settings were used in the wizard.

- **Startup Channel Selection:** Channel Sequencer
- **Control/Status Ports:** Temp Bus enabled
- **ADC Calibration:** No ADC Offset or Gain calibration
- **Channel Sequencer:** Check VP/VN, and vauxp/vauxn pins 0, 1, 2, 4, 5, 6, 7, 9, 10, 12 13, 14, 15

These settings can be seen in Figure 2 to Figure 5



Show disabled ports

Component Name: xadc_wiz_0

Interface Options: [✓ AXI4Lite] DRP None

Startup Channel Selection: Simultaneous Selection, Independent ADC, Single Channel, [✓ Channel Sequencer]

AXI4STREAM Options: Enable AXI4Stream; FIFO Depth 7 [7 - 1020]

Control/Status Ports: reset_in [✓ Temp Bus] JTAG Arbiter

Temperature Bus refresh rate (default wait cycle) = 10 us

Event Mode Trigger: [✓ convst in] convstclk in

Timing Mode: [✓ Continuous Mode] Event Mode

DRP Timing Options: [✓ Enable DCLK]

DCLK Frequency(MHz)	100	[8.0 - 250.0]
ADC Conversion Rate(KSPS)	1000	[154.0 - 1000.0]
Acquisition Time (CLK)	4	

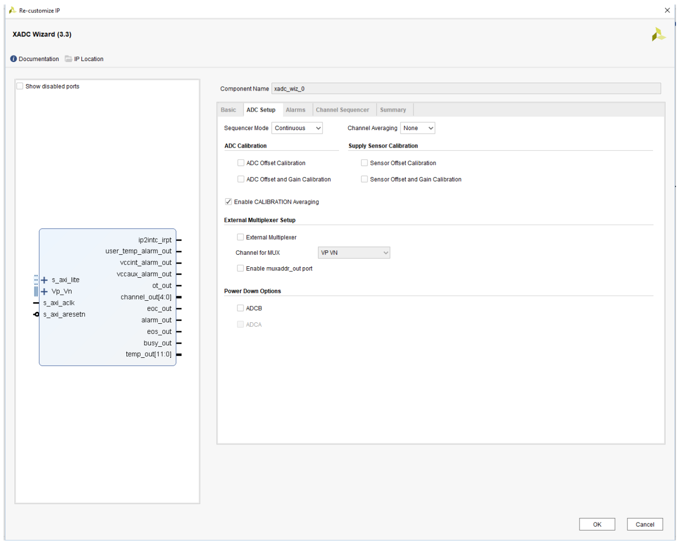
Clock divider value = 4

ADC Clock Frequency(MHz) = 25.00

Analog Sim File Options:

Sim File Selection	Default	
Analog Stimulus File	design	
Sim File Location	./	
Waveform Type	CONSTANT	
Frequency (KHz)	1.0	[0.1 - 480.77]
Number of Wave	1	[1 - 1000]

Fig. 2. XADC Wizard - Basic



Component Name xadc_wiz_0

Sequencer Mode: Continuous
Channel Averaging: None

ADC Calibration
- [ ] ADC Offset Calibration
- [ ] ADC Offset and Gain Calibration

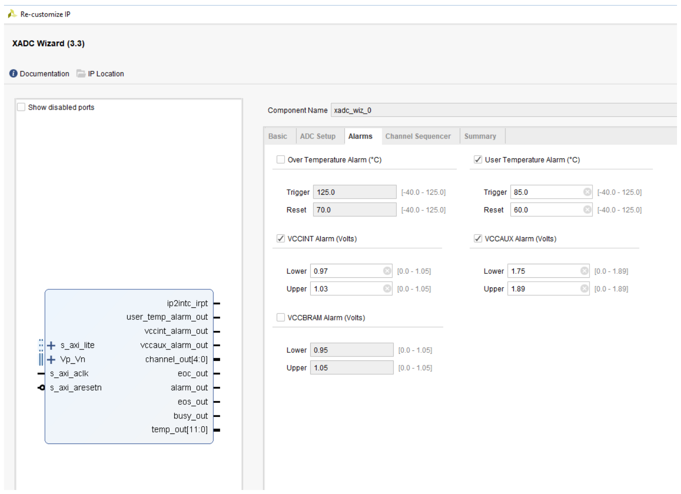
Supply Sensor Calibration
- [ ] Sensor Offset Calibration
- [ ] Sensor Offset and Gain Calibration

- [x] Enable CALIBRATION Averaging

External Multiplexer Setup
- [ ] External Multiplexer
- Channel for MUX: VP VN
- [ ] Enable muxaddr_out port

Power Down Options
- [ ] ADCB
- [ ] ADCA

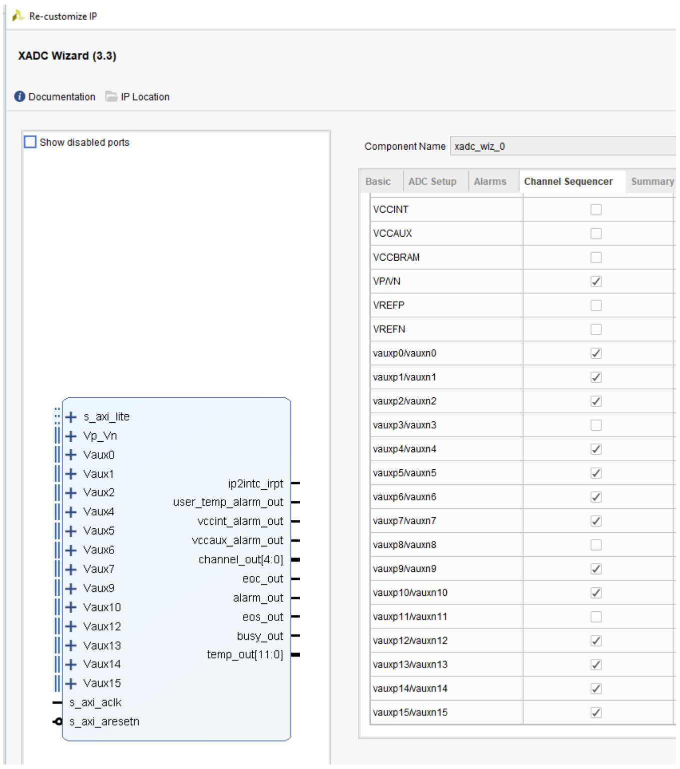
Fig. 3. XADC Wizard - ADC Setup



Component Name xadc_wiz_0

- [ ] Over Temperature Alarm (°C): Trigger 125.0 [-40.0 - 125.0]; Reset 70.0 [-40.0 - 125.0]
- [x] User Temperature Alarm (°C): Trigger 85.0 [-40.0 - 125.0]; Reset 60.0 [-40.0 - 125.0]
- [x] VCCINT Alarm (Volts): Lower 0.97 [0.0 - 1.05]; Upper 1.03 [0.0 - 1.05]
- [x] VCCAUX Alarm (Volts): Lower 1.75 [0.0 - 1.89]; Upper 1.89 [0.0 - 1.89]
- [ ] VCCBRAM Alarm (Volts): Lower 0.95 [0.0 - 1.05]; Upper 1.05 [0.0 - 1.05]

Fig. 4. XADC Wizard - Alarms

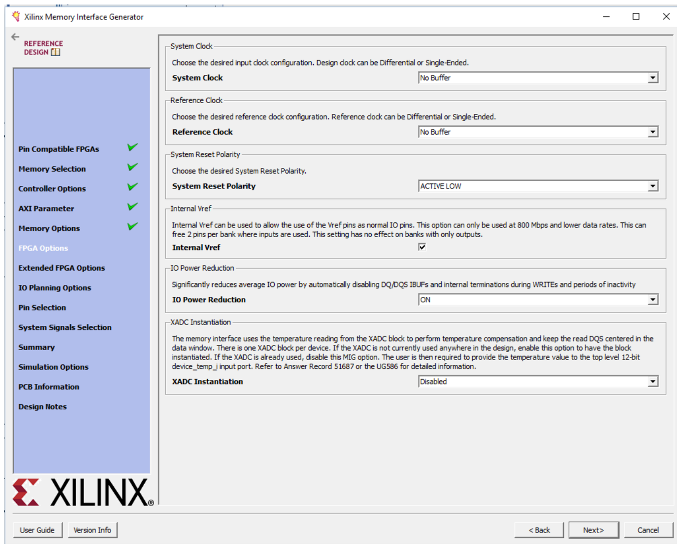


Component Name xadc_wiz_0

VCCINT	☐
VCCAUX	☐
VCCBRAM	☐
VP/VN	✓
VREFP	☐
VREFN	☐
vauxp0/vauxn0	✓
vauxp1/vauxn1	✓
vauxp2/vauxn2	✓
vauxp3/vauxn3	☐
vauxp4/vauxn4	✓
vauxp5/vauxn5	✓
vauxp6/vauxn6	✓
vauxp7/vauxn7	✓
vauxp8/vauxn8	☐
vauxp9/vauxn9	✓
vauxp10/vauxn10	✓
vauxp11/vauxn11	☐
vauxp12/vauxn12	✓
vauxp13/vauxn13	✓
vauxp14/vauxn14	✓
vauxp15/vauxn15	✓

Fig. 5. XADC Wizard - Channel Sequencer

The memory interface generator was previously set up to use the ADC temperature readings for temperature compensation, so it instantiated thee ADC. It was necessary to disable this instantiation in the memory interface generator, then to route the temperature output of the XADC Wizard block to the memory interface generator.



System Clock
Choose the desired input clock configuration. Design clock can be Differential or Single-Ended.
System Clock: No Buffer

Reference Clock
Choose the desired reference clock configuration. Reference clock can be Differential or Single-Ended.
Reference Clock: No Buffer

System Reset Polarity
Choose the desired System Reset Polarity.
System Reset Polarity: ACTIVE LOW

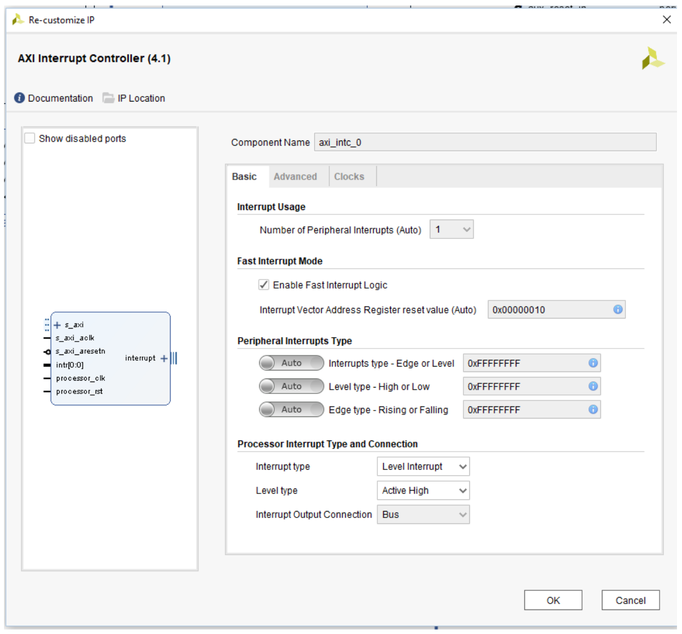
Internal Vref
Internal Vref can be used to allow the use of the Vref pins as normal IO pins. This option can only be used at 800 Mbps and lower data rates. This can free 2 pins per bank where inputs are used. This setting has no effect on banks with only outputs.
Internal Vref: ✓

IO Power Reduction
Significantly reduces average IO power by automatically disabling DQ/DQS IBUFs and internal terminations during WRITEs and periods of inactivity
IO Power Reduction: ON

XADC Instantiation
The memory interface uses the temperature reading from the XADC block to perform temperature compensation and keep the read DQS centered in the data window. There is one XADC block per device. If the XADC is not currently used anywhere in the design, enable this option to have the block instantiated. If the XADC is already used, disable this MIG option. The user is then required to provide the temperature value to the top level 12-bit device_temp_i input port. Refer to Answer Record 51687 or the UG586 for detailed information.
XADC Instantiation: Disabled

Fig. 6. Memory Interface Generator - Disable XADC Instantiation

After this step, the AXI Interrupt Controller was added, and its settings were adjusted according to Figure 7 to allow fast interrupt logic.



Re-customize IP

AXI Interrupt Controller (4.1)

Documentation IP Location

Show disabled ports

Component Name axi_intc_0

Basic Advanced Clocks

Interrupt Usage

Number of Peripheral Interrupts (Auto) 1

Fast Interrupt Mode

[✓] Enable Fast Interrupt Logic

Interrupt Vector Address Register reset value (Auto) 0x00000010

Peripheral Interrupts Type

Auto	Interrupts type - Edge or Level	0xFFFFFFFF
Auto	Level type - High or Low	0xFFFFFFFF
Auto	Edge type - Rising or Falling	0xFFFFFFFF

Processor Interrupt Type and Connection

Interrupt type	Level Interrupt
Level type	Active High
Interrupt Output Connection	Bus

OK Cancel

Fig. 7. AXI Interrupt Controller Settings

Next, the shield pins 0-19 and 26-41 were added to the block diagram from the board tab. Both shield interfaces were added to a single AXI GPIO IP. Finally, a Concat IP was added to concatenate the ADC interrupts with any future interrupts that may be added. Its concatenation input number was set to 1 in its settings.

Now that all IPs were added to the board, the board was wired appropriately. On the XADC Wizard, each pin was first set to be external, as shown in Figure 8. Then, the board was wired as shown in Figure 9 (see appendices for larger version).

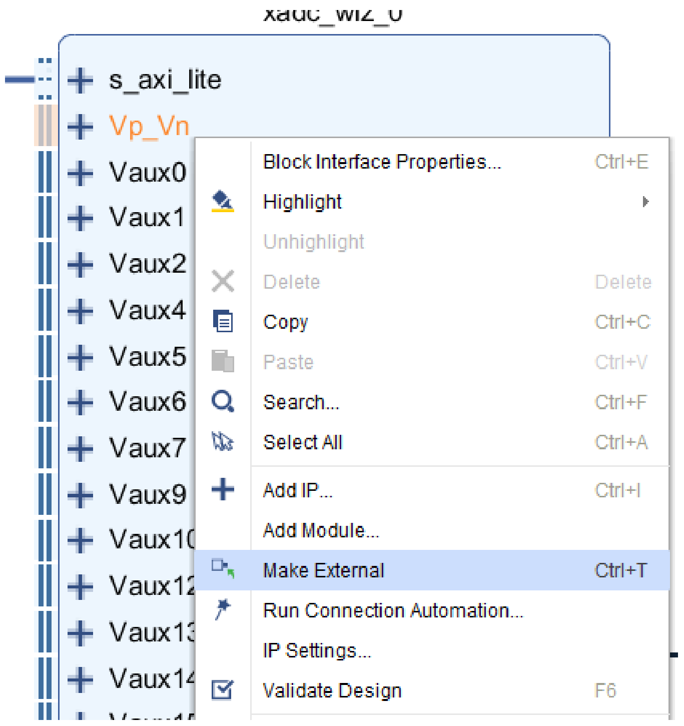


Fig. 8. Make XADC Pins External

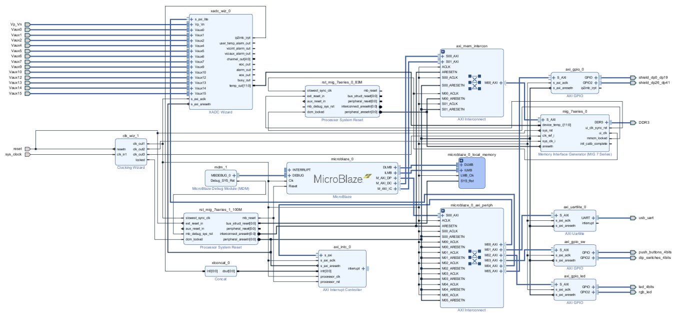
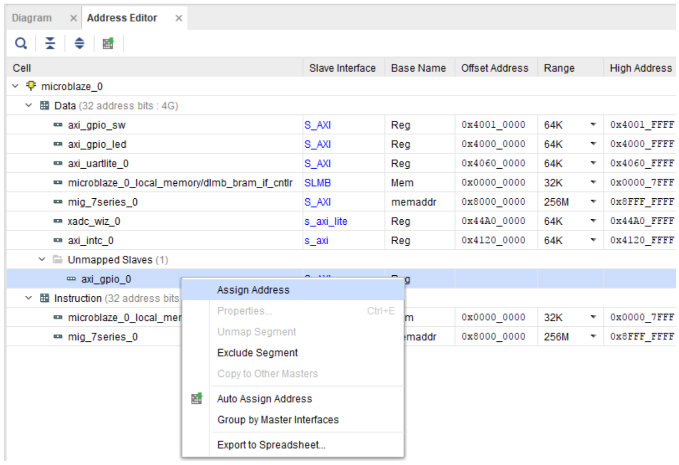


Fig. 9. System Block Diagram

Now, the Address Editor was used to assign addresses to all of the new slave devices, as shown in Figure 10

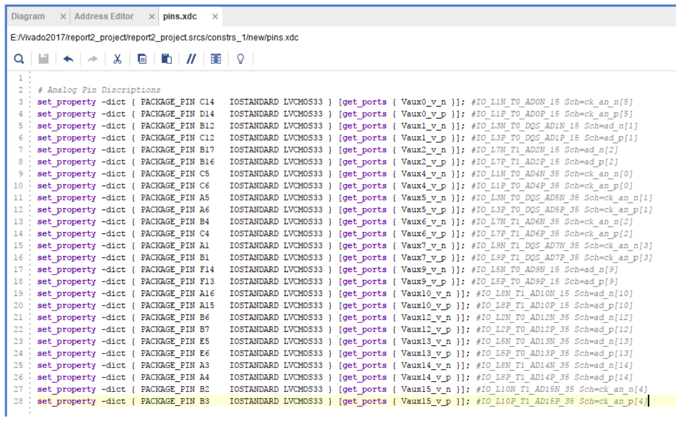


Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
axi_gpio_sw	S_AXI	Reg	0x4001_0000	64K	0x4001_FFFF
axi_gpio_led	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	256M	0x8FFF_FFFF
xadc_wiz_0	s_axi_lite	Reg	0x44A0_0000	64K	0x44A0_FFFF
axi_intc_0	s_axi	Reg	0x4120_0000	64K	0x4120_FFFF
Unmapped Slaves (1)					
axi_gpio_0	[illegible]	[illegible]g			
Instruction (32 address bits					
microblaze_0_local_mer		[illegible]m	0x0000_0000	32K	0x0000_7FFF
mig_7series_0		[illegible]maddr	0x8000_0000	256M	0x8FFF_FFFF

Fig. 10. Assigning New Slave Addresses

Next, the hdl wrapper was created for the block diagram. Creating this wrapper auto-validates the design to ensure no errors exist.

A final step was to add pin constraints that defined which physical FPGA pins the analog input pins were located at. To do this, a new constraints (.xdc) file was added, and the text shown in Figure 11. Note that in this file, the PACKAGE_PIN represents the physical FPGA pin, and the argument of the get_ports command is the analog pin name (Vauxi_v_n for negative, Vauxi_v_p for positive). To ensure the pin names are correct, it is recommended to open the hdl wrapper that was created earlier to see what names are given to those ports, and to make sure the pins.xdc file matches them.



E:/Vivado2017/report2_project/report2_project.srcs/constrs_1/new/pins.xdc

```

# Analog Pin Discriptions
set_property -dict { PACKAGE_PIN C14   IOSTANDARD LVCMOS33 } [get_ports { Vaux0_v_n }]; #IO_L1N_T0_AD0N_15 Sch=ck_an_n[5]
set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports { Vaux0_v_p }]; #IO_L1P_T0_AD0P_15 Sch=ck_an_p[5]
set_property -dict { PACKAGE_PIN B12   IOSTANDARD LVCMOS33 } [get_ports { Vaux1_v_n }]; #IO_L3N_T0_DQS_AD1N_15 Sch=ad_n[1]
set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { Vaux1_v_p }]; #IO_L3P_T0_DQS_AD1P_15 Sch=ad_p[1]
set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVCMOS33 } [get_ports { Vaux2_v_n }]; #IO_L7N_T1_AD2N_15 Sch=ad_n[2]
set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVCMOS33 } [get_ports { Vaux2_v_p }]; #IO_L7P_T1_AD2P_15 Sch=ad_p[2]
set_property -dict { PACKAGE_PIN C5    IOSTANDARD LVCMOS33 } [get_ports { Vaux4_v_n }]; #IO_L1N_T0_AD4N_35 Sch=ck_an_n[0]
set_property -dict { PACKAGE_PIN C6    IOSTANDARD LVCMOS33 } [get_ports { Vaux4_v_p }]; #IO_L1P_T0_AD4P_35 Sch=ck_an_p[0]
set_property -dict { PACKAGE_PIN A5    IOSTANDARD LVCMOS33 } [get_ports { Vaux5_v_n }]; #IO_L3N_T0_DQS_AD5N_35 Sch=ck_an_n[1]
set_property -dict { PACKAGE_PIN A6    IOSTANDARD LVCMOS33 } [get_ports { Vaux5_v_p }]; #IO_L3P_T0_DQS_AD5P_35 Sch=ck_an_p[1]
set_property -dict { PACKAGE_PIN B4    IOSTANDARD LVCMOS33 } [get_ports { Vaux6_v_n }]; #IO_L7N_T1_AD6N_35 Sch=ck_an_n[2]
set_property -dict { PACKAGE_PIN C4    IOSTANDARD LVCMOS33 } [get_ports { Vaux6_v_p }]; #IO_L7P_T1_AD6P_35 Sch=ck_an_p[2]
set_property -dict { PACKAGE_PIN A1    IOSTANDARD LVCMOS33 } [get_ports { Vaux7_v_n }]; #IO_L9N_T1_DQS_AD7N_35 Sch=ck_an_n[3]
set_property -dict { PACKAGE_PIN B1    IOSTANDARD LVCMOS33 } [get_ports { Vaux7_v_p }]; #IO_L9P_T1_DQS_AD7P_35 Sch=ck_an_p[3]
set_property -dict { PACKAGE_PIN F14   IOSTANDARD LVCMOS33 } [get_ports { Vaux9_v_n }]; #IO_L5N_T0_AD9N_15 Sch=ad_n[9]
set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 } [get_ports { Vaux9_v_p }]; #IO_L5P_T0_AD9P_15 Sch=ad_p[9]
set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports { Vaux10_v_n }]; #IO_L8N_T1_AD10N_15 Sch=ad_n[10]
set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33 } [get_ports { Vaux10_v_p }]; #IO_L8P_T1_AD10P_15 Sch=ad_p[10]
set_property -dict { PACKAGE_PIN B6    IOSTANDARD LVCMOS33 } [get_ports { Vaux12_v_n }]; #IO_L2N_T0_AD12N_35 Sch=ad_n[12]
set_property -dict { PACKAGE_PIN B7    IOSTANDARD LVCMOS33 } [get_ports { Vaux12_v_p }]; #IO_L2P_T0_AD12P_35 Sch=ad_p[12]
set_property -dict { PACKAGE_PIN E5    IOSTANDARD LVCMOS33 } [get_ports { Vaux13_v_n }]; #IO_L5N_T0_AD13N_35 Sch=ad_n[13]
set_property -dict { PACKAGE_PIN E6    IOSTANDARD LVCMOS33 } [get_ports { Vaux13_v_p }]; #IO_L5P_T0_AD13P_35 Sch=ad_p[13]
set_property -dict { PACKAGE_PIN A3    IOSTANDARD LVCMOS33 } [get_ports { Vaux14_v_n }]; #IO_L8N_T1_AD14N_35 Sch=ad_n[14]
set_property -dict { PACKAGE_PIN A4    IOSTANDARD LVCMOS33 } [get_ports { Vaux14_v_p }]; #IO_L8P_T1_AD14P_35 Sch=ad_p[14]
set_property -dict { PACKAGE_PIN B2    IOSTANDARD LVCMOS33 } [get_ports { Vaux15_v_n }]; #IO_L10N_T1_AD15N_35 Sch=ck_an_n[4]
set_property -dict { PACKAGE_PIN B3    IOSTANDARD LVCMOS33 } [get_ports { Vaux15_v_p }]; #IO_L10P_T1_AD15P_35 Sch=ck_an_p[4]
```

Fig. 11. Assigning Pin Constraints

At this point, the bitstream was created and exported to the SDK for software implementation.

III. Digital I/O Hardware Setup

The processes for setting up the Digital IØwas the exact same as the process described in the first tutorial project in which the AXI GPIO was set up. For this project a new AXI GPIO module was set up and then was connected to ”shield dp0 dp19 and shield dp26 dp41” so that the AXI GPIO module would connect to the shield pins. An image of this module is shown in Figure 12.

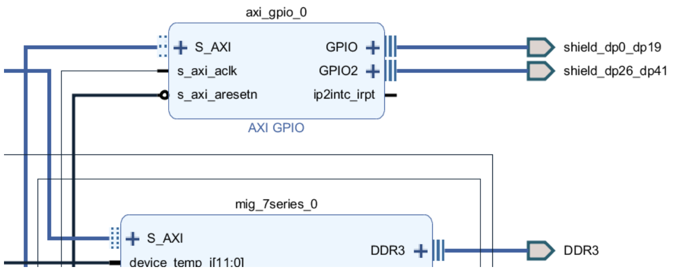


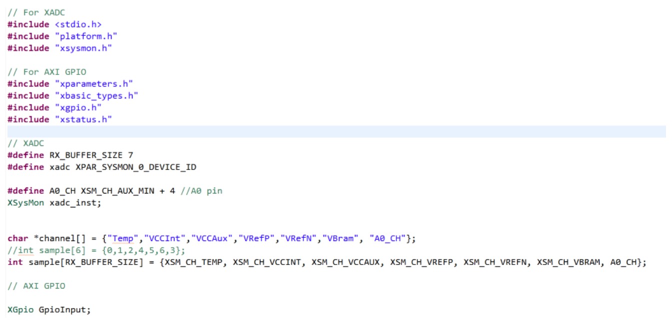
Fig. 12. Digital I/O Hardware

IV. SDK and Analog and Digital Source Implementation

The SDK implementation of the project involved the following steps:

- Create new application project using the latest design wrapper and the ”hello world” template
- Use the xsysmon library for the ADC
- Use the gpio library for digital reads
- Setup ADC and GPIO ports
- Reading ADC values and GPIO values
- Print results over serial connection to serial monitor

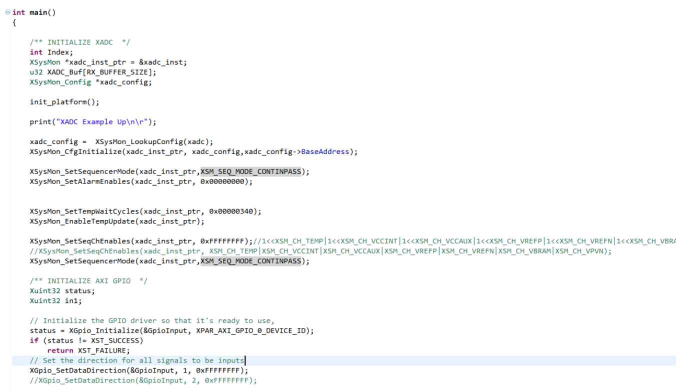
To set up the code for analog and digital input, first the right libraries had to be included. For the XADC portion ”xsysmon.h” was used and for the GPIO portion ”xgpio.h” was used. The ”xsysmon.h” contained all necessary addressing information to select the proper registers when trying to read ADC values. After this variables for controlling the channels were set up. For analog input, it was desired to use the A0 pin. In order to do this the analog ADC AUX min channel value from ”xsysmon.h” was shifted by 4 which would start the addressing at the start of channel A0. This portion of the code is shown in Figure 13.



```
// For XADC
#include <stdio.h>
#include "platform.h"
#include "xsysmon.h"

// For AXI GPIO
#include "xparameters.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "xstatus.h"

// XADC
#define RX_BUFFER_SIZE 7
#define xadc XPAR_SYSMON_0_DEVICE_ID

#define A0_CH XSM_CH_AUX_MIN + 4 //A0 pin
XSysMon xadc_inst;


char *channel[] = {"Temp","VCCInt","VCCAux","VRefP","VRefN","VBram", "A0_CH"};
//int sample[6] = {0,1,2,4,5,6,3};
int sample[RX_BUFFER_SIZE] = {XSM_CH_TEMP, XSM_CH_VCCINT, XSM_CH_VCCAUX, XSM_CH_VREFP, XSM_CH_VREFN, XSM_CH_VBRAM, A0_CH};

// AXI GPIO

XGpio GpioInput;
```

Fig. 13. Code Preamble Section

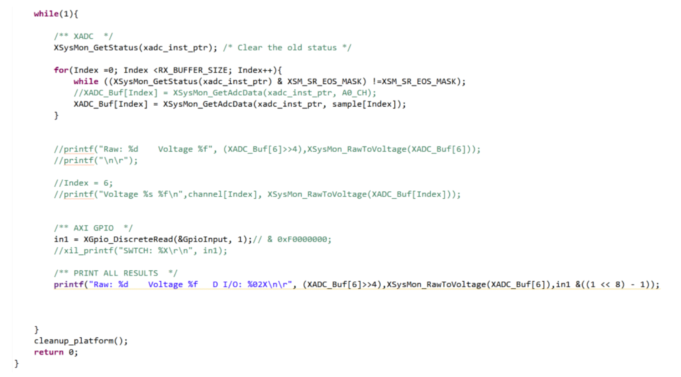
After this the main section of the code was set up. First to be set up in this section was the initialization for all of the hardware. For the ADC this involved getting the configuration, setting the sequencer mode, disabling alarms, and channel enables. It should be noted that one issue that occurred in this project was caused by the sequencer mode. Example code had this set to ”safe mode” such that only internal ADC measurements could be made. Setting this to ”CONTINPASS” allowed the ADC to take external measurements and thus allowed the project to work. For the GPIO initialization involved initializing the GPIO device associated with the shield pins and then setting the pins to be inputs. Only I/O pins 0 to 19 were set as inputs since pins 0 through 7 were actually used for digital input. The code for this section can be seen in Figure 14.



```
int main()
{

    /** INITIALIZE XADC  */
    int Index;
    XSysMon *xadc_inst_ptr = &xadc_inst;
    u32 XADC_Buf[RX_BUFFER_SIZE];
    XSysMon_Config *xadc_config;

    init_platform();

    print("XADC Example Up\n\r");

    xadc_config =  XSysMon_LookupConfig(xadc);
    XSysMon_CfgInitialize(xadc_inst_ptr, xadc_config,xadc_config->BaseAddress);

    XSysMon_SetSequencerMode(xadc_inst_ptr,XSM_SEQ_MODE_CONTINPASS);
    XSysMon_SetAlarmEnables(xadc_inst_ptr, 0x00000000);


    XSysMon_SetTempWaitCycles(xadc_inst_ptr, 0x00000340);
    XSysMon_EnableTempUpdate(xadc_inst_ptr);

    XSysMon_SetSeqChEnables(xadc_inst_ptr, 0xFFFFFFFF);//1<<XSM_CH_TEMP|1<<XSM_CH_VCCINT|1<<XSM_CH_VCCAUX|1<<XSM_CH_VREFP|1<<XSM_CH_VREFN|1<<XSM_CH_VBRA
    //XSysMon_SetSeqChEnables(xadc_inst_ptr, XSM_CH_TEMP|XSM_CH_VCCINT|XSM_CH_VCCAUX|XSM_CH_VREFP|XSM_CH_VREFN|XSM_CH_VBRAM|XSM_CH_VPVN);
    XSysMon_SetSequencerMode(xadc_inst_ptr,XSM_SEQ_MODE_CONTINPASS);

    /** INITIALIZE AXI GPIO  */
    Xuint32 status;
    Xuint32 in1;

    // Initialize the GPIO driver so that it's ready to use,
    status = XGpio_Initialize(&GpioInput, XPAR_AXI_GPIO_0_DEVICE_ID);
    if (status != XST_SUCCESS)
        return XST_FAILURE;
    // Set the direction for all signals to be inputs
    XGpio_SetDataDirection(&GpioInput, 1, 0xFFFFFFFF);
    //XGpio_SetDataDirection(&GpioInput, 2, 0xFFFFFFFF);
```

Fig. 14. Code Initialization Section

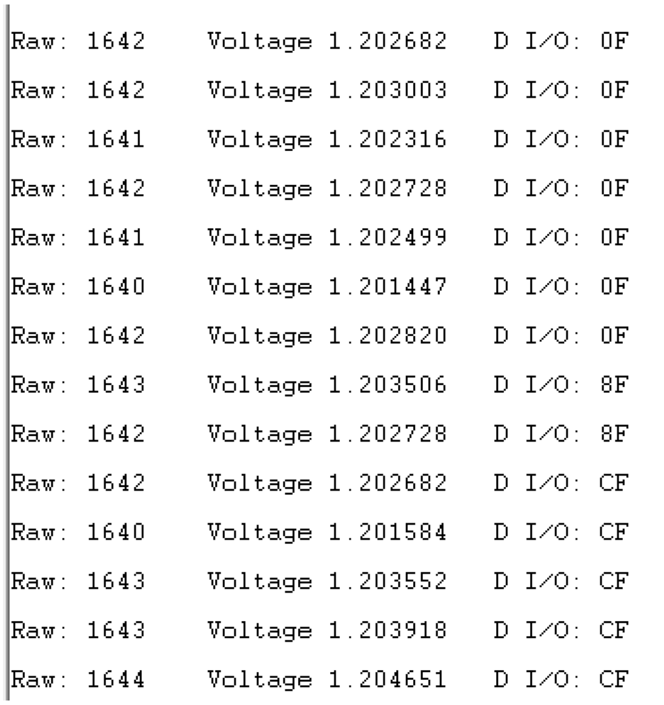
The final section of the code involved reading the ADC values and converting to voltages as well as reading the digital input. All of these inputs that were read in were then printed to the serial port in a form such that the raw ADC, voltage value, and digital value could all be read and continuously updated. This was all done in an infinite loop so that measurements were always being made and then reported. This section of the code is shown in Figure 15.



```
    while(1){

        /** XADC  */
        XSysMon_GetStatus(xadc_inst_ptr); /* Clear the old status */

        for(Index =0; Index <RX_BUFFER_SIZE; Index++){
            while ((XSysMon_GetStatus(xadc_inst_ptr) & XSM_SR_EOS_MASK) !=XSM_SR_EOS_MASK);
            //XADC_Buf[Index] = XSysMon_GetAdcData(xadc_inst_ptr, A0_CH);
            XADC_Buf[Index] = XSysMon_GetAdcData(xadc_inst_ptr, sample[Index]);
        }


        //printf("Raw: %d    Voltage %f", (XADC_Buf[6]>>4),XSysMon_RawToVoltage(XADC_Buf[6]));
        //printf("\n\r");

        //Index = 6;
        //printf("Voltage %s %f\n",channel[Index], XSysMon_RawToVoltage(XADC_Buf[Index]));


        /** AXI GPIO  */
        in1 = XGpio_DiscreteRead(&GpioInput, 1);// & 0xF0000000;
        //xil_printf("SWTCH: %X\r\n", in1);

        /** PRINT ALL RESULTS  */
        printf("Raw: %d    Voltage %f   D I/O: %02X\n\r", (XADC_Buf[6]>>4),XSysMon_RawToVoltage(XADC_Buf[6]),in1 &((1 << 8) - 1));



    }
    cleanup_platform();
    return 0;
}
```

Fig. 15. Code Value Reading and Reporting Section

To set up an analog input, a voltage divider was set up using a potentiometer which was set to a 3.3V max voltage and then the middle wiper was connected to A0. A voltage of 3.3V was used because this was the maximum voltage that this ADC pin could handle. Analog input was also tested using a function generator and a sinusoidal input. A digital input was set up using an external 8-dip switch array with the 8 switched connected to pins IO0 through IO7. Output to the serial port was able to show that the values were being measured and reported properly. Voltages were in the expected range of 0 to 3.3V and raw values were in the expected range of 0 to 4095 for the 12 bits of storage for each. The digital values shown on the monitor matched the configuration of the dip switches. An example output of this is shown in Figure 16.



```
Raw: 1642    Voltage 1.202682    D I/O: 0F
Raw: 1642    Voltage 1.203003    D I/O: 0F
Raw: 1641    Voltage 1.202316    D I/O: 0F
Raw: 1642    Voltage 1.202728    D I/O: 0F
Raw: 1641    Voltage 1.202499    D I/O: 0F
Raw: 1640    Voltage 1.201447    D I/O: 0F
Raw: 1642    Voltage 1.202820    D I/O: 0F
Raw: 1643    Voltage 1.203506    D I/O: 8F
Raw: 1642    Voltage 1.202728    D I/O: 8F
Raw: 1642    Voltage 1.202682    D I/O: CF
Raw: 1640    Voltage 1.201584    D I/O: CF
Raw: 1643    Voltage 1.203552    D I/O: CF
Raw: 1643    Voltage 1.203918    D I/O: CF
Raw: 1644    Voltage 1.204651    D I/O: CF
```

Fig. 16. Code Value Reading and Reporting Section

V. Conclusion

In conclusion XADC and AXI GPIO modules were implemented on an Arty board along with the Microblaze soft processor. The steps for setting the both modules in the Vivado software were described. After the modules were set up and the hardware exported for working with the SDK, code was written to set up channels, initialize hardware, take measurements and report values from both the XADC and the AXI GPIO modules. The XADC was set up to take analog measurements on pin A0. The AXI GPIO was set up to make digital measurements on pins IO0 through IO7. Analog input was provided by a potentiometer voltage divider circuit and a function generator. Digital input was provided by an array of 8 dip switches. The ADC was shown to work as it was able to measure both the 0 to 3.3V voltage divider input and the varying sinusoidal input from the function generator with 12 bits of precision. The digital input was shown to work as the values of 0x00 to 0xFF could be reported as being measured as would match with the current combination of the dip switches. Though these two demonstrators it was determined that analog and digital input were possible and set up correctly.

APPENDIX A
FINAL BLOCK DIAGRAM

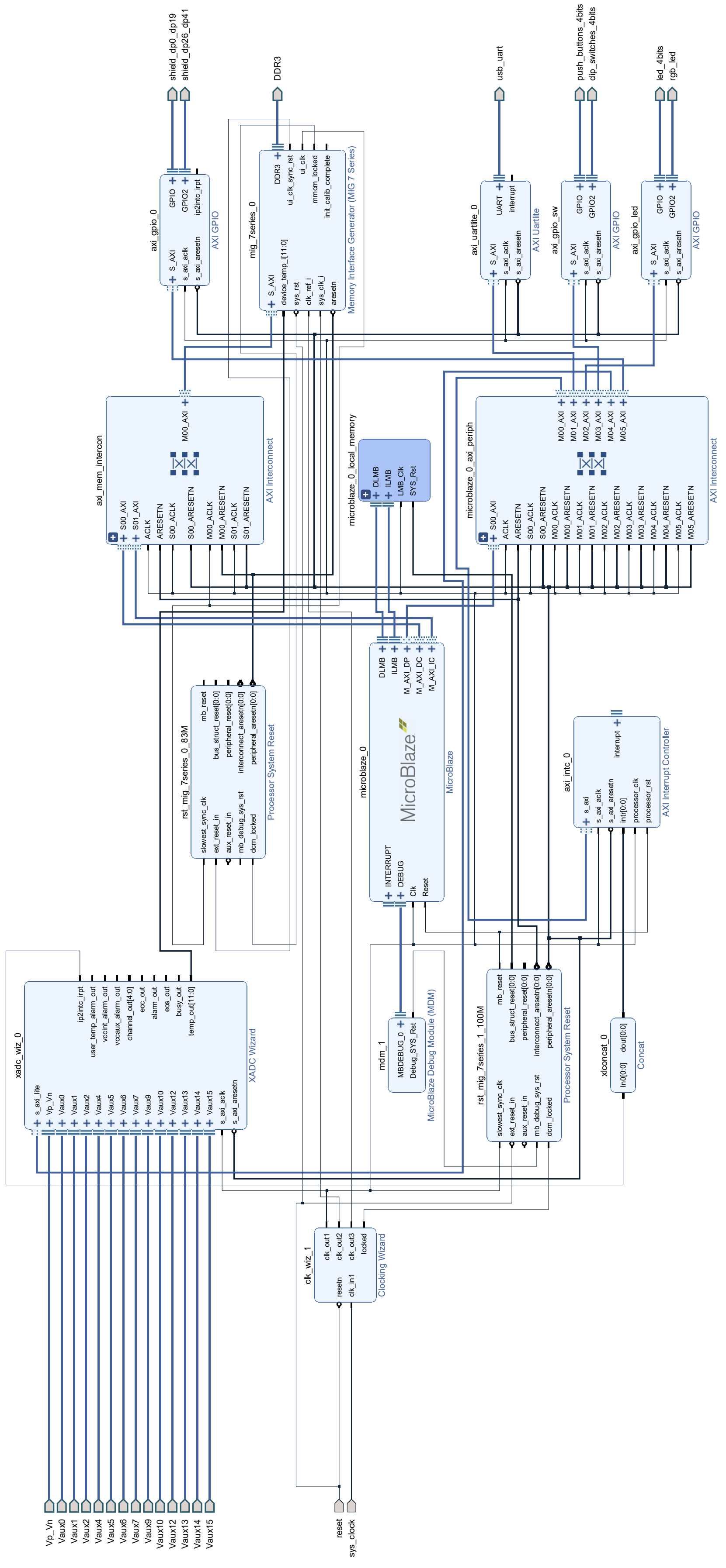


Fig. 17. Enlarged Block Diagram