Adding HCSR04 IP, switches, buttons, and leds, and AXI timers to Zynq system

Introduction

This lab guides you through making a custom HCSR04 IP. It also guides you through adding GPIO switches, GPIO buttons, GPIO 4 bit leds, and 2 axi timers to a Zynq system. It also guides you through pinning a HCSR04 module to a breadboard and connecting it to the zybo board.

Objectives

After completing this lab you will be able to,

- Build a simple breadboard circuit to interface with an HCR04 module
- Use the IP Packager feature of Vivado to create a custom hcsr04_ip peripheral
- Create hardware Design utilizing the hcsr04_ip module
- Create an application that utilizes the hcsr04_ip module drivers
- Measure distance

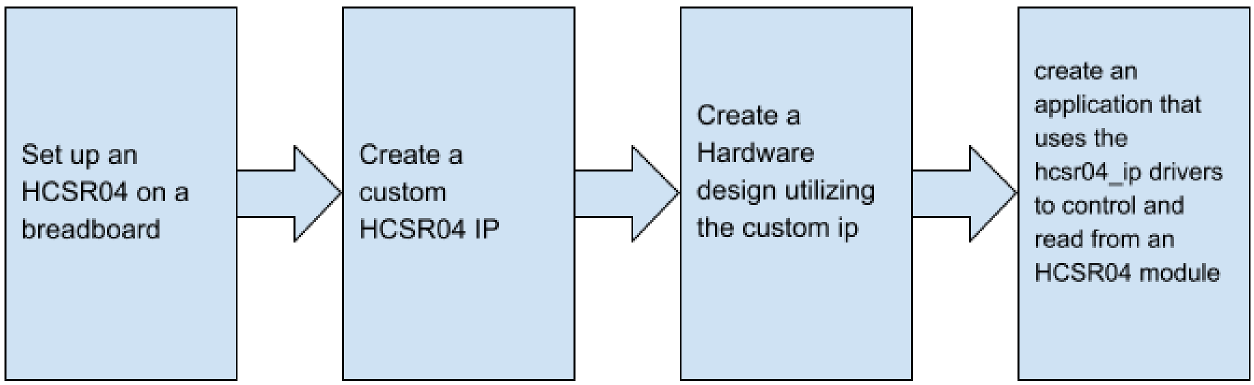
Procedure

This lab is separated into steps that consist of general overview statements that provide information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab. This lab comprises 4 primary steps: Set up an HCSR04 on a breadboard, Create a custom HCSR04 IP, Create a Hardware design utilizing the custom ip, and create an application that uses the hcsr04_ip drivers to control and read from an HCSR04 module

Design Description

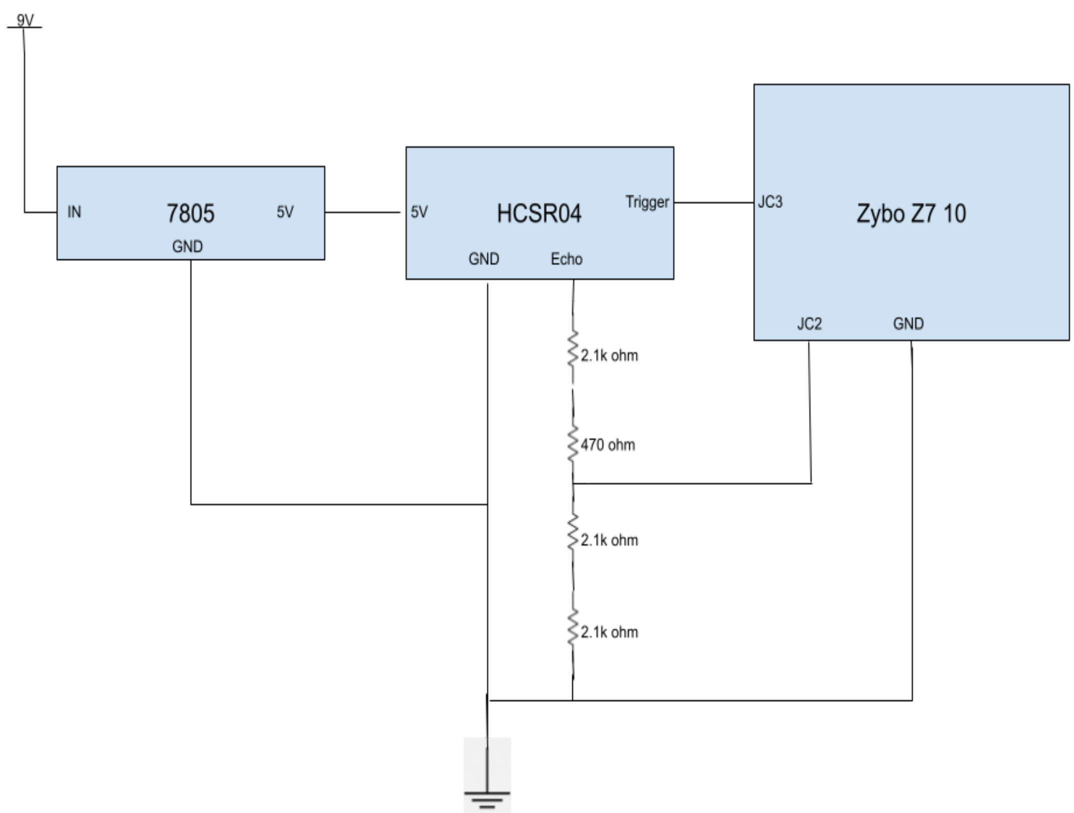
There will be 2 axi timers and 1 GPIO button whose interrupts will be connected to a Zynq processing system through a concat block. 2 more GPIO will be instantiated and connected to the boards switches and 4 bit mono color leds. An instance of the custom hcsr04_ip will be created and its echo and trigger ports will be made external and connected to the board's JC PMOD ports.

General Flow



Setting up HCSR04 on breadboard **Step 1**

1-1 Connect the HCSR04 and 7805 on a breadboard. Connect that to the Zybo z7 10



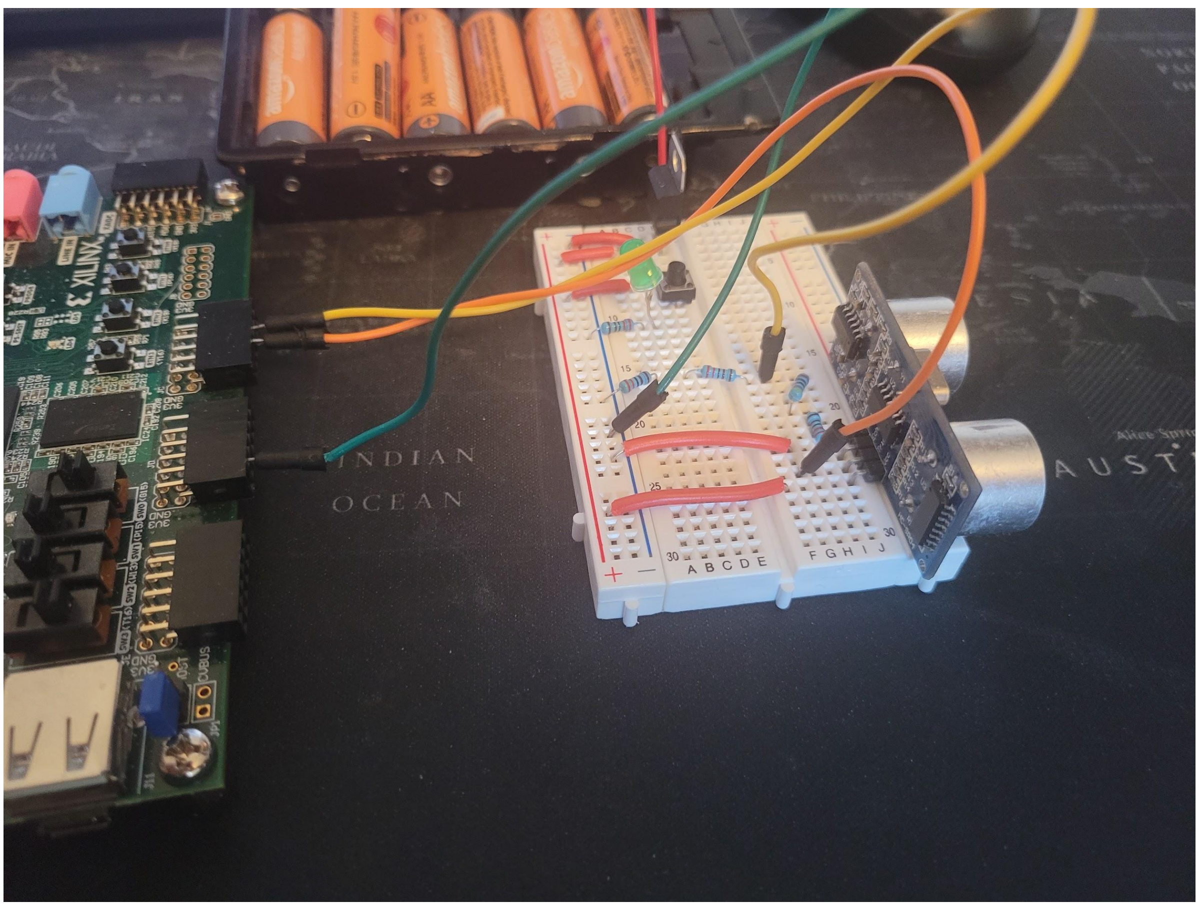
1-1-1 Get a power source that can deliver over 9v of power.

1-1-2 Connect a 7805 to 3 rows of a breadboard.

1-1-3 Connect the 9v into the input of the 7805. Connect the ground of the 7805 to the ground of the bread board. Connect the 5v output to the power strip on the breadboard.

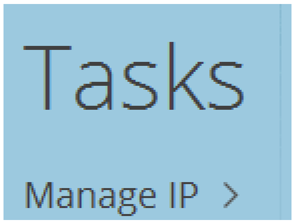
1-1-4 Connect the HCSR04 onto any 4 rows of a breadboard. Connect the Vcc pin to the powerstrip of the bread board. Connect the ground to the ground of the breadboard. Connect The trigger pin to the JC3 port of the Zybo board. Connect the echo pin to a 2.1kohm resistor to a 470 ohm resistor to a 2.1kohm resistor to a 2.1kohm resistor to ground. Connect a wire from between the two pairs of resistors to the JC2 port of the Zybo board.

1-1-5 Connect the Zybo Z7 10 ground pin to the ground of the breadboard.

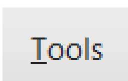


2-1 Go to manage ip and create an AXI Lite HCSR04 custom IP with Trigger output and Echo input

2-1-1 Open Vivado then click on Manage Ip under tasks. Then click on new IP location.



2-1-2A vivado window will open. Click on tools, then click on Create and Package new IP. Save everything into a well named folder



2-1-3 Name the IP HCSR04 and create it as an AXI lite. Edit it instead of closing.

2-1-4 In the sources tab click on the top level and add echo and trigger as an input and output

```
// Users to add ports here
input echo,
output trigger,
// User ports ends
```

2-1-5 At the bottom of the file port the signals into the lower level file that handles the AXI signals

```
hcsr04_ip_v1_0_S_AXI_inst (
    .echo(echo),
    .trigger(trigger),
```

2-1-6 In the sources tab click on the low level file that handles the AXI signals (ends with S_AXI_inst) and add the echo and trigger as an input and output.

```
// Do not modify the ports beyond this line
input echo,
output trigger,
```

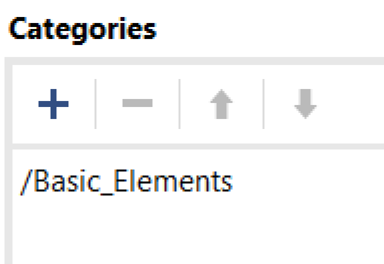
2-1-7 In order for us to read the echo we need to put the echo on the output. We can put it on any output but we will have it output when we read address offset 0

```
// Address decoding for reading registers
case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
  2'h0    : reg_data_out <= echo;
  2'h1    : reg_data_out <= slv_reg1;
  2'h2    : reg_data_out <= slv_reg2;
  2'h3    : reg_data_out <= slv_reg3;
  default : reg_data_out <= 0;
endcase
```

2-1-8 In order for us to send out a trigger signal we need to have our tigger variable receive data from a register that we have control over. We will use slave reg3 so that what ever we write on the lsb of slv_reg3 will become an output of our IP

```
// Add user logic here

assign trigger = slv_reg3[0];

// User logic ends
```

2-1-9 Now that we are finished with the logic we can start packaging the IP
In the Packager tab go to categories. Click on the plus and deselect everything. After that select Basic Elements.



2-1-10 In the Compatibility tab click on the plus and add zynq pre production.

Family | Simulator

Family	Life Cycle
zynq	Pre-Production

2-1-11 In the file groups tab click merge file changes and it should look like this after.

File Groups

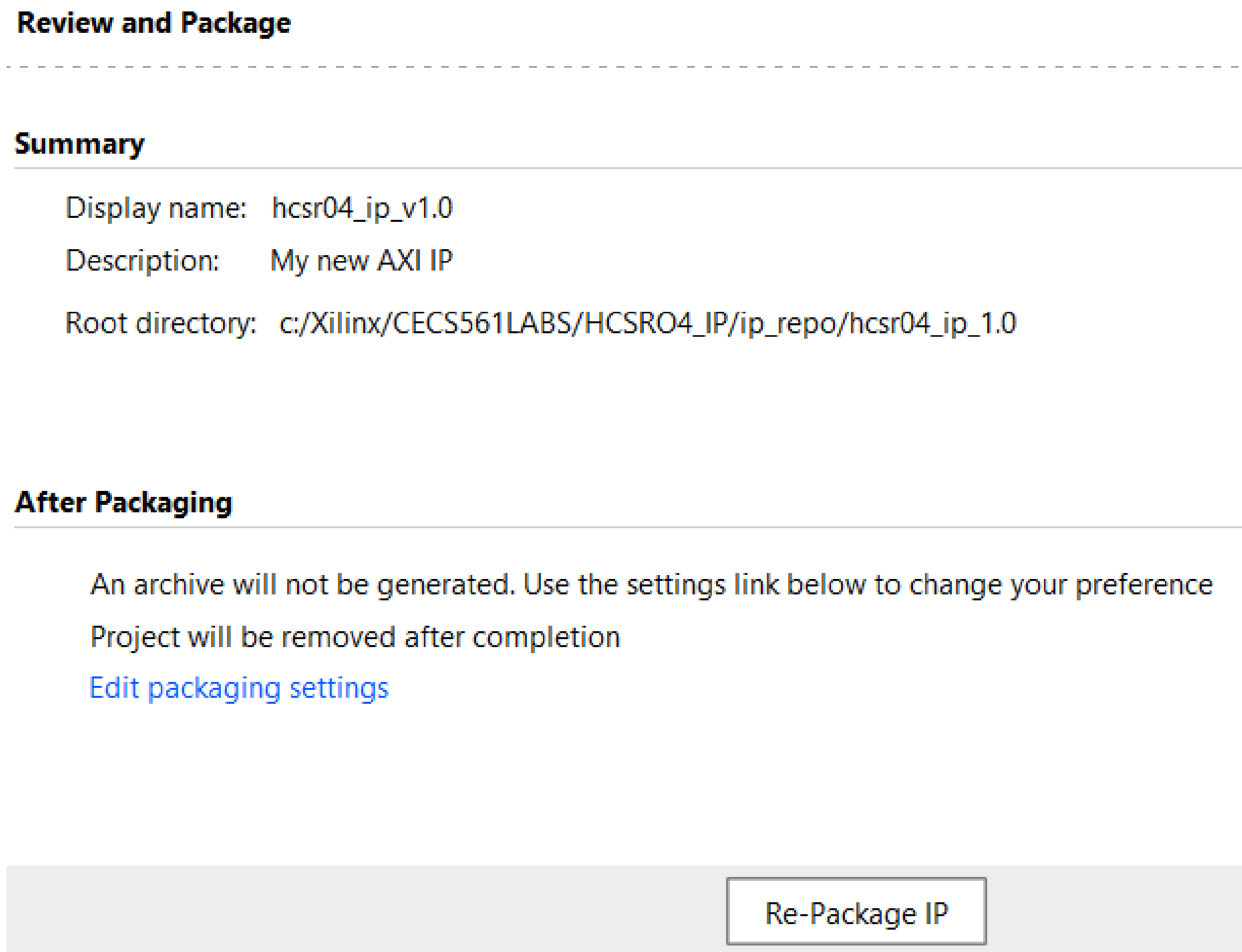
Name	Library Name	Type	Is Include	Used In Constant	File Group Name	Model Name
Standard			☐	☐		
Advanced			☐	☐		
Verilog Synthesis (2)			☐	☐		hcsr04_ip_v1_0
Verilog Simulation (2)			☐	☐		hcsr04_ip_v1_0
Software Driver (6)			☐	☐		
UI Layout (1)			☐	☐		
Block Diagram (1)			☐	☐		

2-1-12 In the customization parameters click on everything should look like this because no parameters were changed or added

Customization Parameters

Name	Description	Display Name	Value	Value Bit String Length
Customization Parameters				
C_S_AXI_DATA_WIDTH	Width of S_AXI data bus	C S AXI DATA WIDTH	32	0
C_S_AXI_ADDR_WIDTH	Width of S_AXI address bus	C S AXI ADDR WIDTH	4	0
C_S_AXI_BASEADDR		C S AXI BASEADDR	0xFFFFFFFF	32
C_S_AXI_HIGHADDR		C S AXI HIGHADDR	0x00000000	32

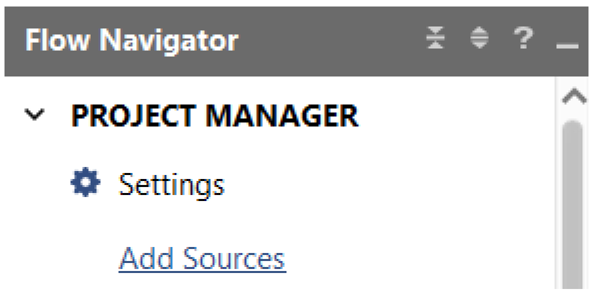
2-1-13 Click on review and package. Merge any changes. Then click on repackage at the bottom.



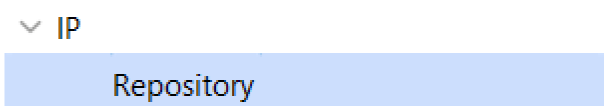
3-3 In vivado add 2 AXI timers, 3 gpio for button(with interrupt), switches, and leds. At a concat block for connecting the interrupts. Added the hcsr04_ip into the design.

3-1-1 Open vivado and create new project. Name it HCSR04_Project. Make sure everything is in verilog and the board is the Zybo Z7 10.

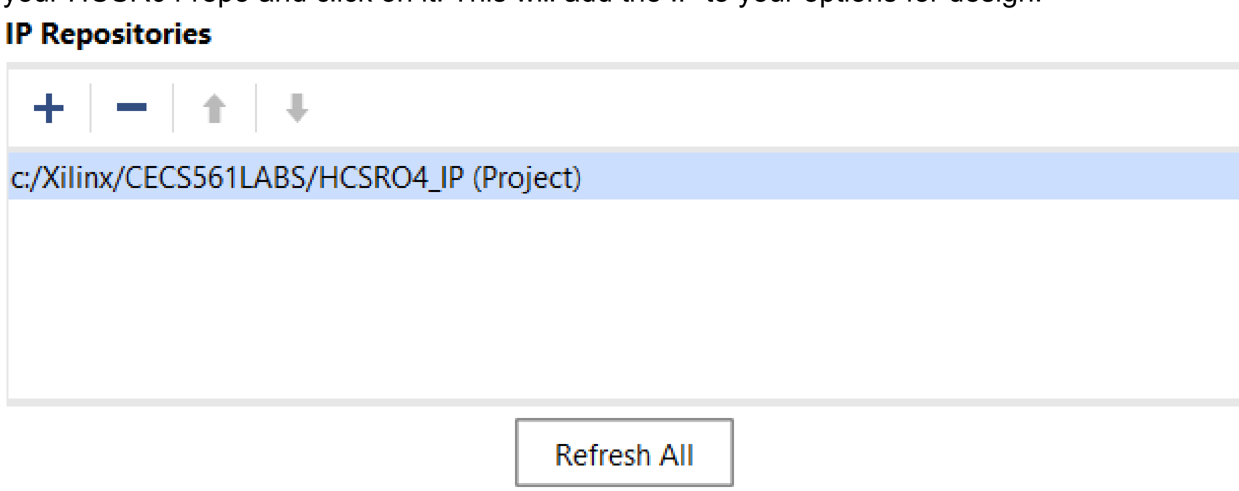
3-1-2 Once in the project go to settings in the flow navigator



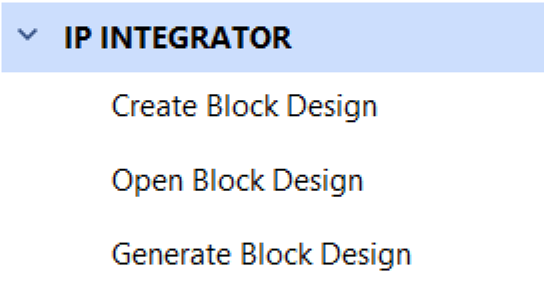
3-1-3 In the settings click on ip then click on repository



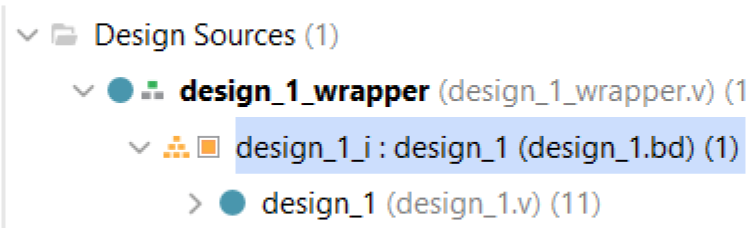
3-1-4 In the Ip Repositories window click on the plus sign. Navigate to the folder where you put your HCSR04 repo and click on it. This will add the IP to your options for design.



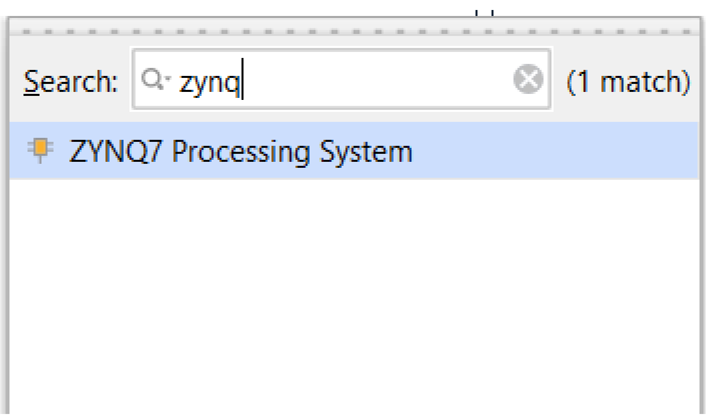
3-1-5 Go to the flow navigator and click on Create Block Design under IP Integrator. Name it whatever you like



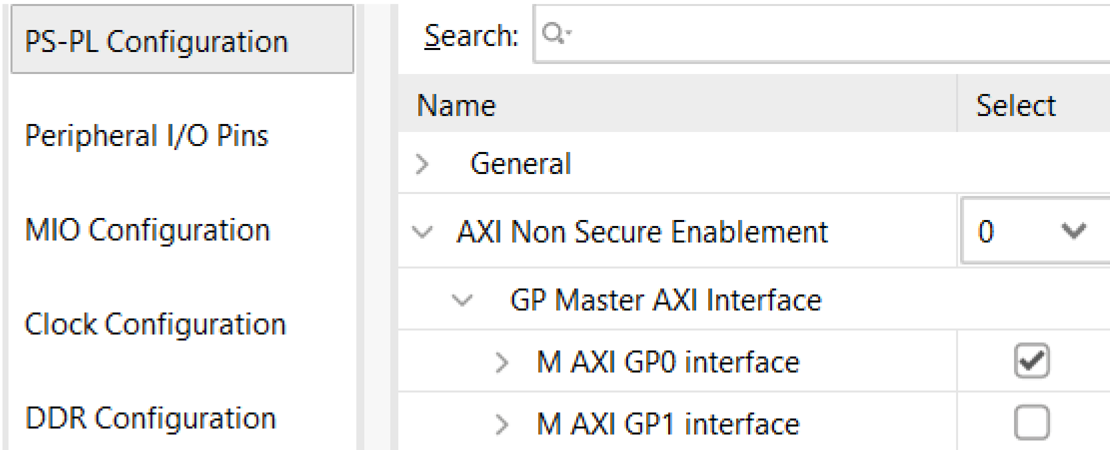
3-1-6 Go to the sources tab and click on the block design to open it. (ends in .bd)



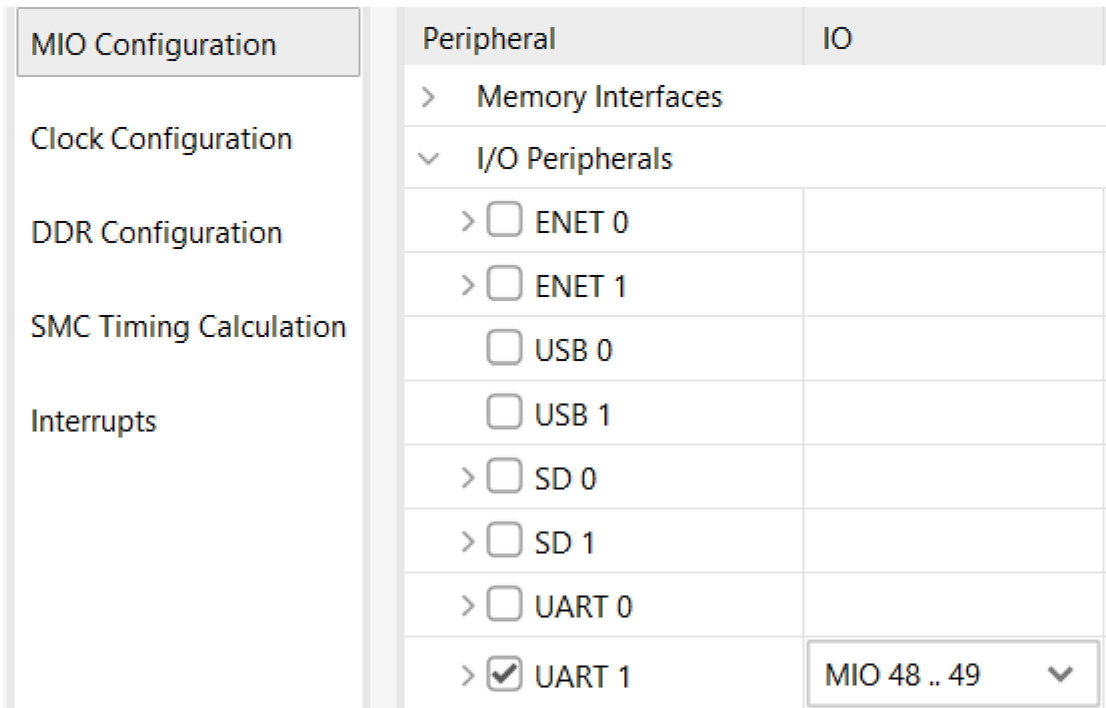
3-1-7 Click on the plus symbol and search for zynq then add the zynq processing system. Run connection automation.



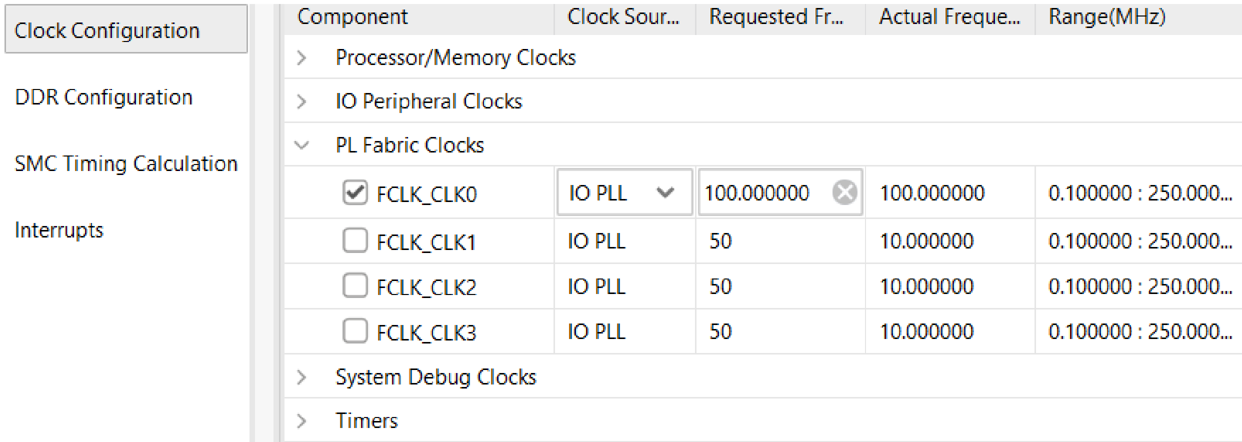
3-1-8 Open the zynq processing system. Disable everything. Enable GPIO Master 0



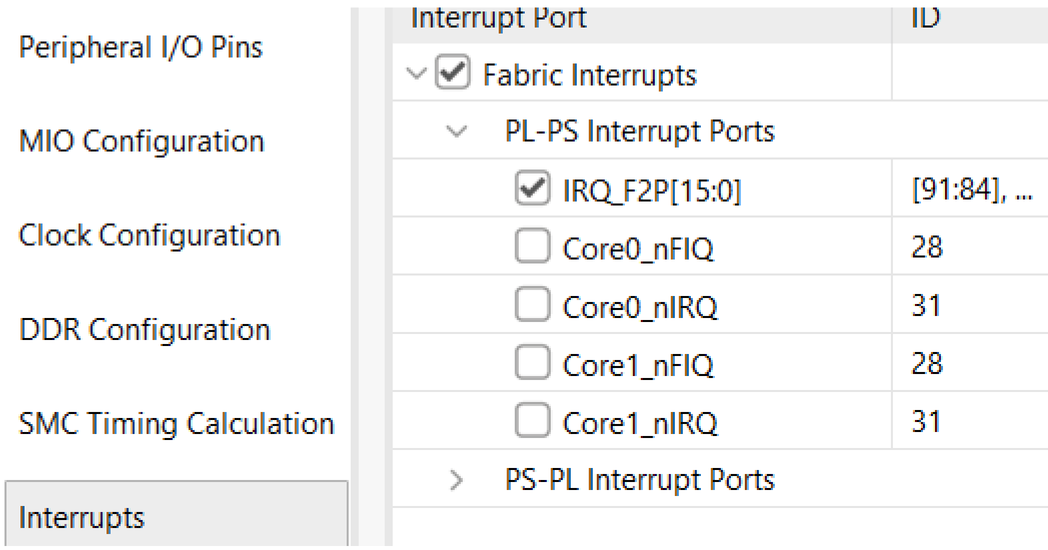
3-1-9 Go to MIO Configuration and disable everything. Enable Uart1.



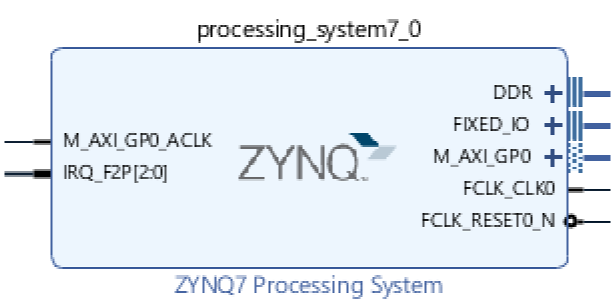
3-1-10 Go to Clock Configuration. Change FCLK_CLK0's Requested Frequency to 100.000000 under PL Fabric Clocks



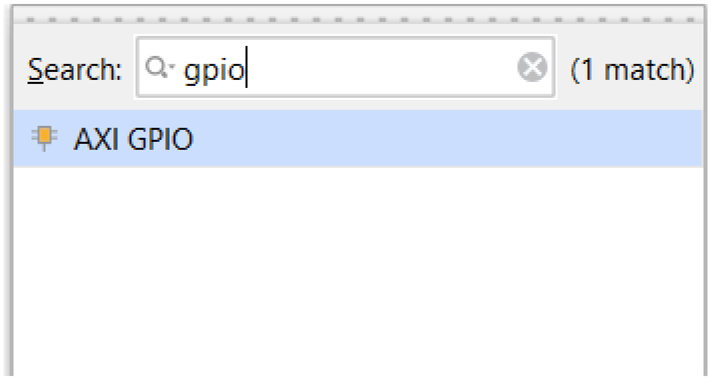
3-1-11 Go to Interrupts. Enable Fabric Interrupt. Under Fabric Interrupt under PL-PS Interrupt Ports enable IRQ_F2P



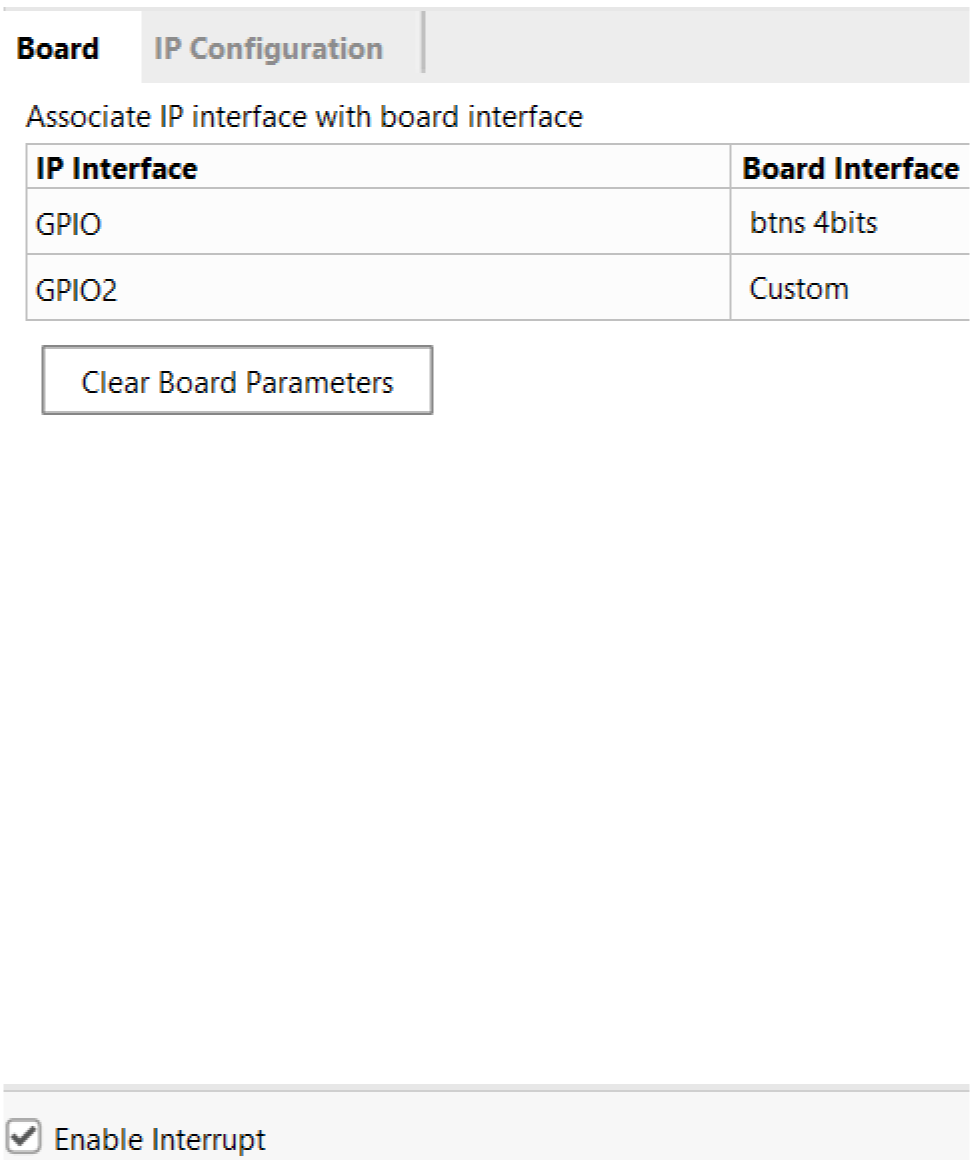
3-1-12 After doing all of that your Zynq Processing system should look like this without some of the lines.



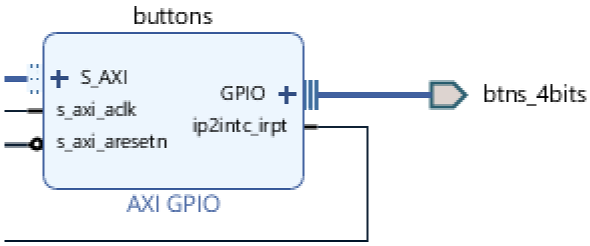
3-1-13 Go to the plus to add IP and search gpio. Add axi gpio 3 times to the project.



3-1-14 Name the first GPIO block buttons then open it for configuration. Click on btns 4bits under Board Interface then check the enable interrupts box at the bottom.



3-1-15 Then run connection automation with everything checked. After configuration the Buttons GPIO block should look like this minus the ip2 line.



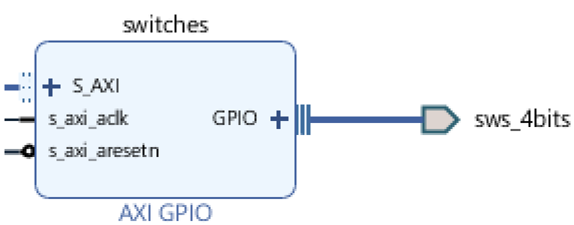
3-1-16 On the second gpio block name it switches then open it up for configuration. Under board interface in the GPIO row change it to sws 4bits.

Board IP Configuration

Associate IP interface with board interface

IP Interface	Board Interface
GPIO	sws 4bits
GPIO2	Custom

3-1-17 After that run connection automation and the block should look like this



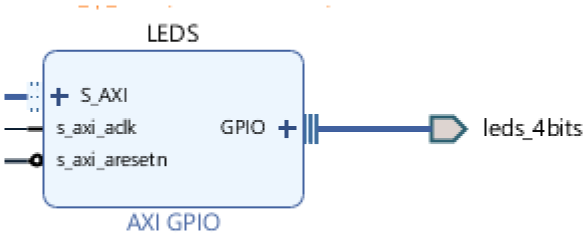
3-1-18 On the second gpio block name it switches then open it up for configuration. Under board interface in the GPIO row change it to leds 4bits.

Board IP Configuration

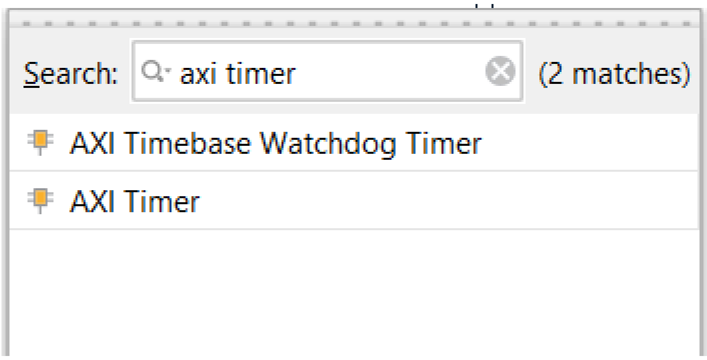
Associate IP interface with board interface

IP Interface	Board Interface
GPIO	leds 4bits
GPIO2	Custom

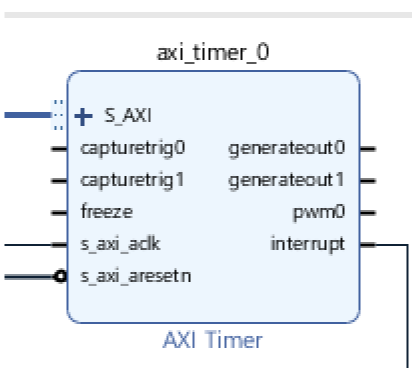
3-1-19 After that run connection automation and the block should look like this



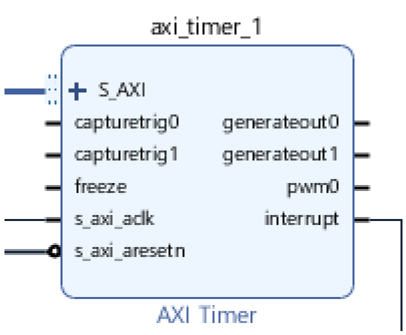
3-1-20 Click on the plus symbol and search for axi timer then add AXI timer to the project twice.



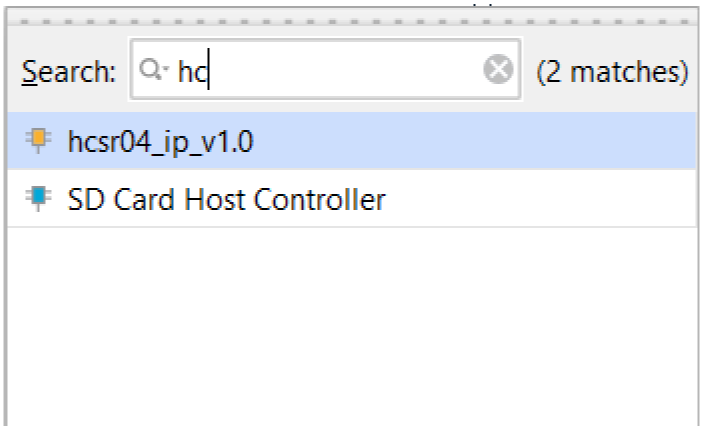
3-1-21 Timer0 should look like this minus the line from the interrupt



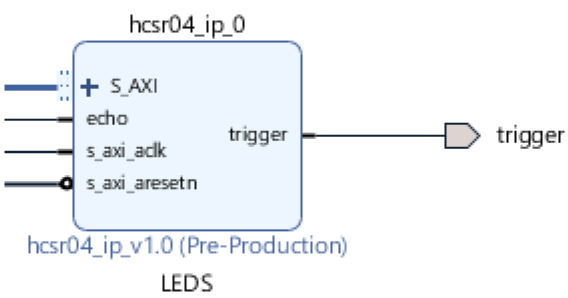
3-1-22 Timer1 should look like this minus the line from the interrupt



3-1-23 Go to the plus sign and search the name of the custom IP hcsr04 and add it to the block diagram.



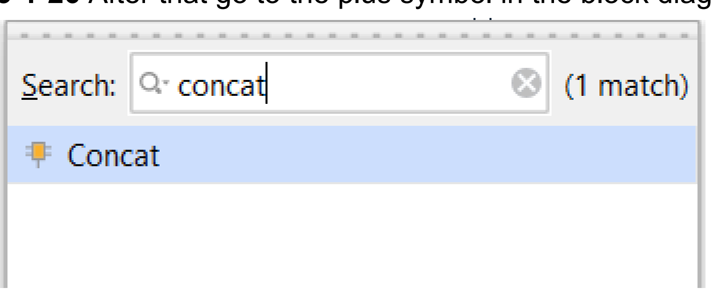
3-1-24 Run Connection Automation then Click on echo and click make external. Click on Trigger and click make external. The end result should look like this with echo as an input to the block diagram and trigger as an output.



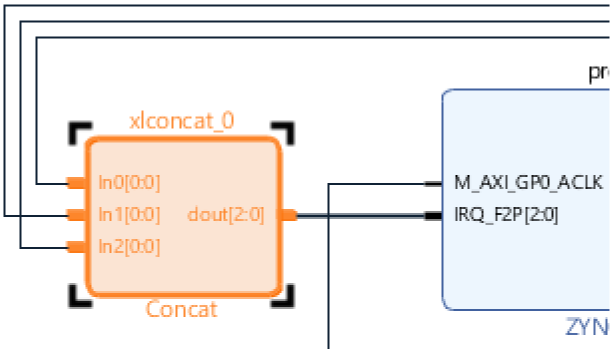
3-1-25 After adding the Zybo Z7 xdc constraint file to the project go into the file and uncomment jc pmod jc2 and jc3. Replace jc[2] and jc[3] with echo and trigger respectively.

```
##Pmod Header JC
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { jc[0] }]; #IO_L10P_T1_34 Sch=jc_p[1]
#set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports { jc[1] }]; #IO_L10N_T1_34 Sch=jc_n[1]
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { echo }]; #IO_L1P_T0_34 Sch=jc_p[2]
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { trigger }]; #IO_L1N_T0_34 Sch=jc_n[2]
```

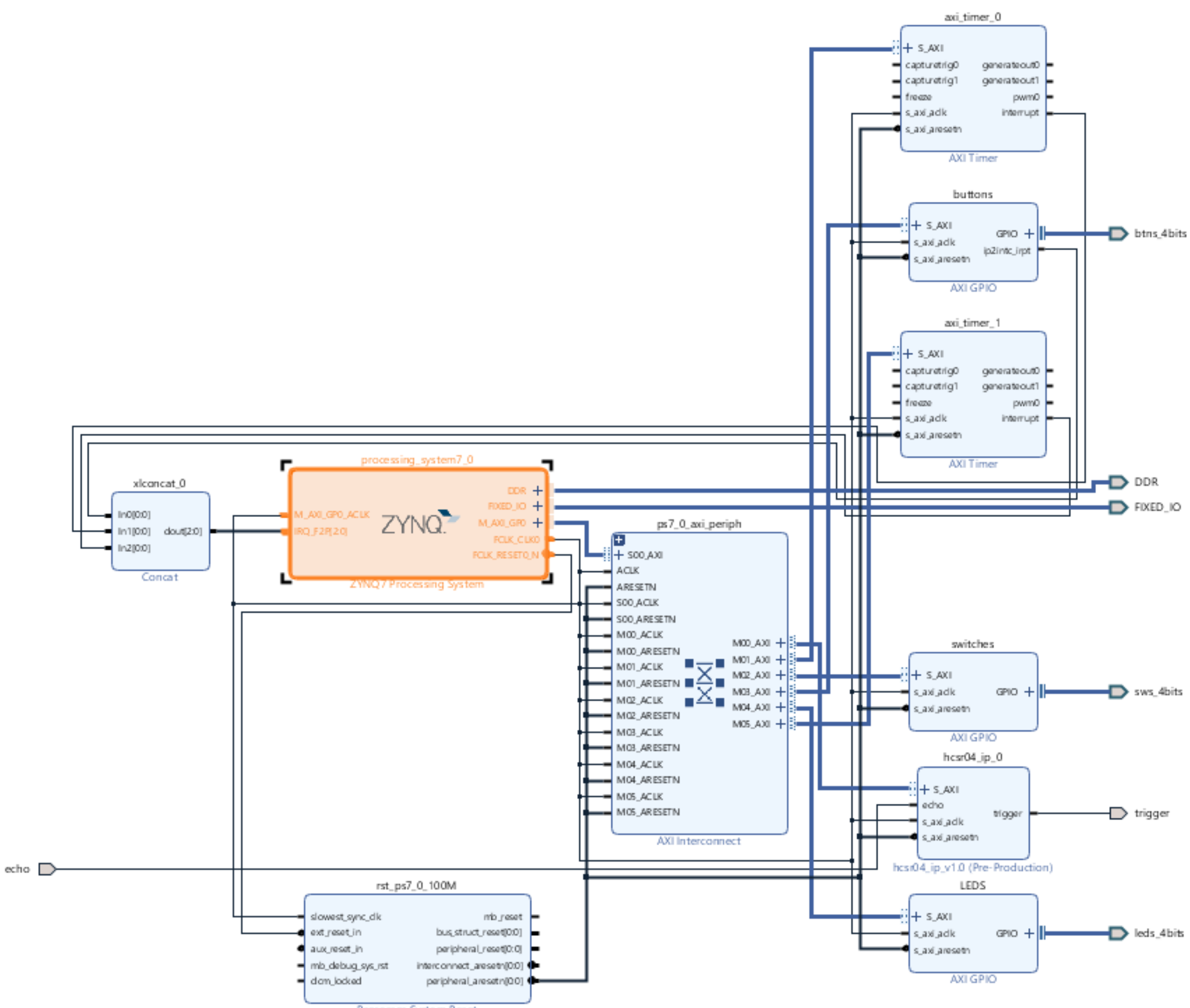
3-1-26 After that go to the plus symbol in the block diagram and search concat then add concat.



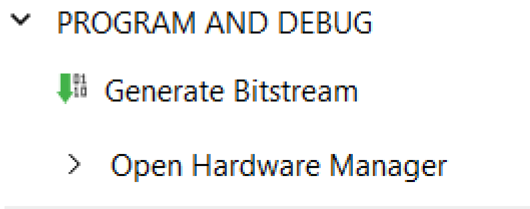
3-1-27 Click on concat and configure it for 3 inputs. After that connect Timer0 Interrupt, Timer1 Interrupt and the Button Interrupt line to the concat block. Connect the output to the zynq IRQ input. The block should look like this.



3-1-28 The whole system should look similar to this



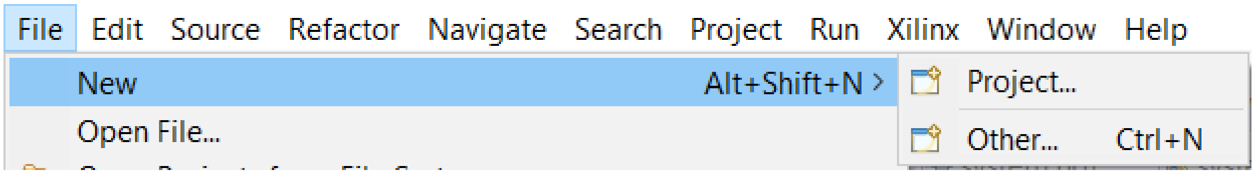
3-1-29 After that go to Generate Bit stream under Program and Debug under Flow Navigator then click ok on everything.



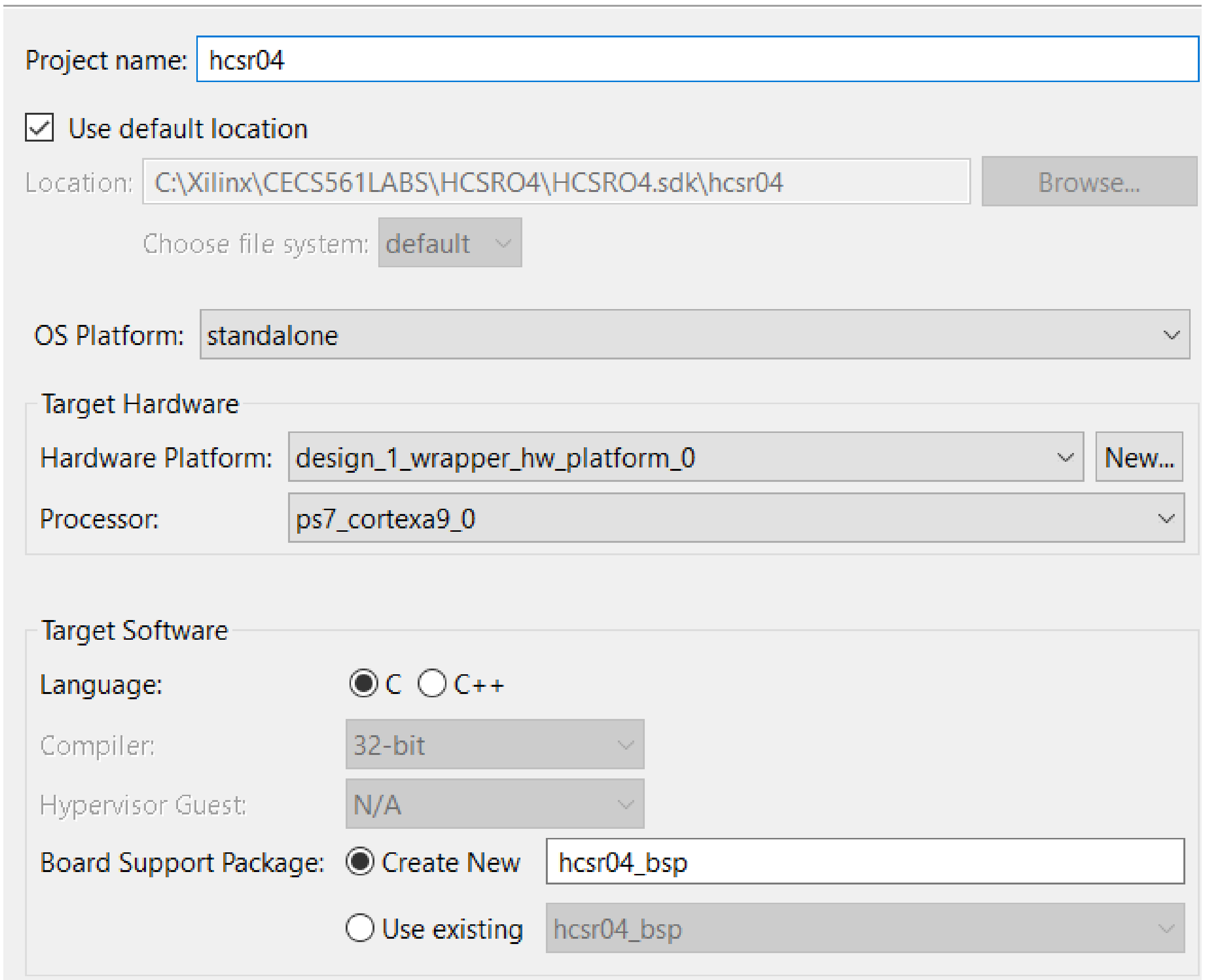
3-1-30 After bitstream is done click on export hardware under the vivado file tab. Then export the hardware with bitstream included.
After that click on launch SDK under the vivado file tab. Launch the SDK.

Creating the Application Step 4

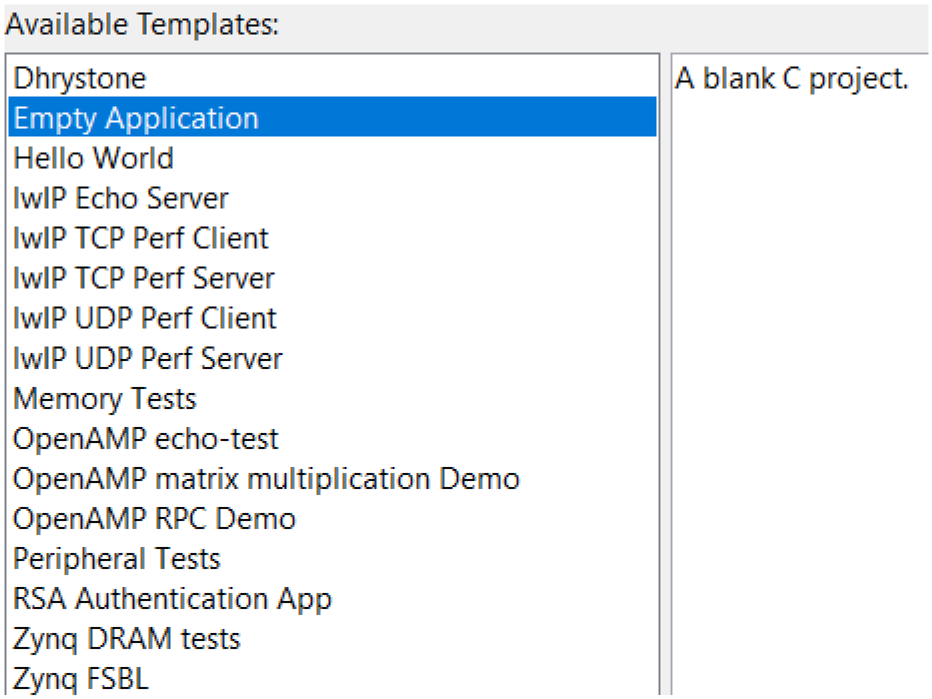
4-1 Create a new application. Include necessary files. Create useful definitions, variables, prototypes etc.. Create an application that uses interrupts to control leds and measure
In the sdk go to file then click on new. Go to Application Project and click on it.



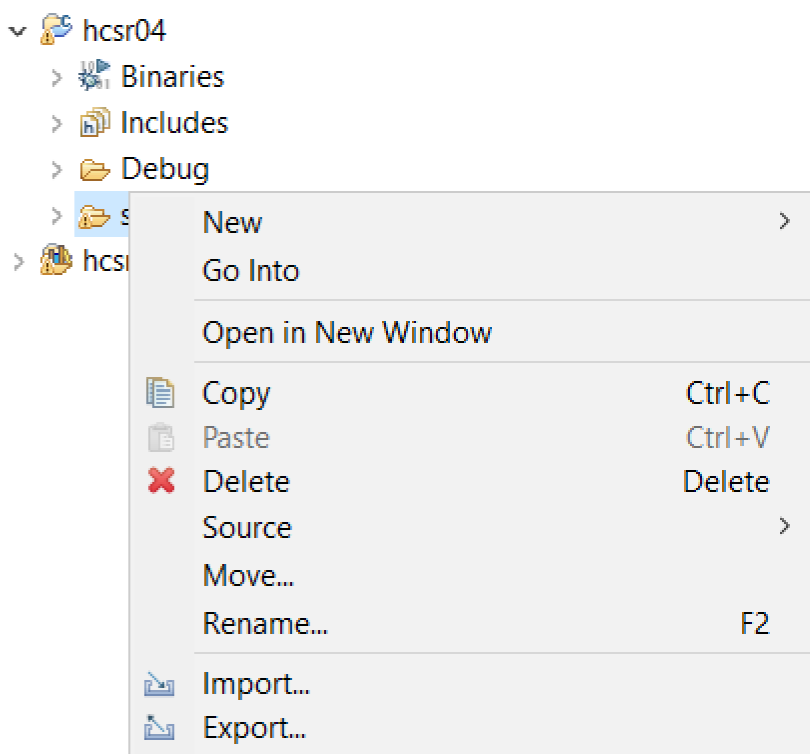
4-1-1 Once open name it to hcsr04 and click next



4-1-2 Create an empty project



4-1-3 Then add a new file. Name is hcsr04.c



4-1-4 After that open the file and make sure it includes xparameter.h to access important system definitions. Make sure that it has xgpio.h to access GPIO driver functions. Include xtmrctr.h for accessing Xilinx Axi timer driver functions. Include xscugic.h for accessing interrupt gic driver functions. Include xil_exception.h for interrupts and xil_printf.h for printing distance. Include hcsr04_ip.h for reading echo and setting trigger.

```
#include "xparameters.h" //Accessing Definitions
#include "xgpio.h"      //Accessing GPIO driver functions
#include "xtmrctr.h"    //Accessing Xilinx Axi timer driver functions
#include "xscugic.h"    //Accessing Xilinx Interrupt gic driver functions
#include "xil_exception.h"
#include "xil_printf.h"

#include "hcsr04_ip.h" //Accessing HCSR04 driving funcitons
```

4-1-5 Some definitions rename definitions from xparameters.h to make them more readable. The bottom definitions are BTN_INT for button channel 1 interrupt mask, TMR0_LOAD for 1 second timer0 interrupt. Tmr1_LOAD for 1 microsecond timer1 interrupt.

```
// Parameter definitions
#define INTC_DEVICE_ID       XPAR_PS7_SCUGIC_0_DEVICE_ID //Interrupt device ID
#define TMR0_DEVICE_ID       XPAR_TMRCTR_0_DEVICE_ID //Timer 0 device ID
#define TMR1_DEVICE_ID       XPAR_TMRCTR_1_DEVICE_ID//Timer 1 device ID
#define BTNS_DEVICE_ID       XPAR_BUTTONS_DEVICE_ID //Buttons device ID
#define SWITCH_DEVICE_ID     XPAR_SWITCHES_DEVICE_ID //Switches Device ID
#define LEDS_DEVICE_ID       XPAR_LEDS_DEVICE_ID // LEDs device ID
#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_BUTTONS_IP2INTC_IRPT_INTR //Button interrupt ID
#define INTC_TMR0_INTERRUPT_ID XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR //Timer 0 interrupt ID
#define INTC_TMR1_INTERRUPT_ID XPAR_FABRIC_AXI_TIMER_1_INTERRUPT_INTR //Timer 1 interrupt ID

#define BTN_INT              XGPIO_IR_CH1_MASK //Btn channel 1 interrupt mask
#define TMR0_LOAD            0x05F5E100 //100_000_000 = 1 second
#define ONE_MICRO            0x00000064 //1 micro second
#define TMR1_LOAD            ONE_MICRO //Timer 1 will be loaded with 1 micro second
```

4-1-6 We make driver instance variables. Three XGPIO for LEDs, Buttons, and Switches. One XScuGic for the gic. Two XtmrCtr for timer 0 and timer 1.

```
XGpio LEDInst, BTNInst, SWITCHInst;// Led, Button, and Switch gpio instance
XScuGic INTCInst; //Interrupt gic instance
XTmrCtr TMR0Inst,TMR1Inst; // Timer 0 and Timer 1 instance
```

4-1-7 We make led_data to hold the value that is written to the leds. Btn_value will read the button press value. The count direction variable will hold the value that determines the direction led_data counts. Timer0_toggle will hold the value that determines if timer0 is counting or stopped. Prev echo is used to hold the value of the previous echo value so that it can be used with the current echo value to detect an edge on the echo pin. Echo durations will hold the time between the positive and negative edge. Tmr1 will keep track of time passing. Distance will use echo_duration and edge detection to determine how far away an object is.

```
static int led_data; //Leds written with led_data
static int btn_value;// button press will be read and control Leds
static int count_direction = 1; //will be used to control the direction of Led data count
static int timer0_toggle = 0; //will be used to turn on or off Timer 0


int prev_echo = 0, //will be used with echo for edge detection
        echo = 0, //echo
        echo_duration = 0, //will be the duration between positive and negative edge detection
        tmr1_count = 0, //will keep track of amount of time passing as the hander gets called
        distance = 0; //used linear regression time to distance
```

These are the 5 functions that will be used in the program and defined later.

```
//----------------------------------------------------
// PROTOTYPE FUNCTIONS
//----------------------------------------------------
static void BTN_Intr_Handler(void *baseaddr_p);
static void TMR0_Intr_Handler(void *baseaddr_p);
static void TMR1_Intr_Handler(void *baseaddr_p);
static int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);
static int IntcInitFunction(u16 DeviceId, XTmrCtr *TMR0InstancePtr, XTmrCtr *TMR1InstancePtr, XGpio *GpioInstancePtr);
```

4-1-8 This is the button interrupt handler. The button interrupt handler is called when there is an interrupt caused by a button being pressed. The handler first disables interrupt while in the handler. The handler then takes in the value of the button pressed. After that if btn 3 is pressed then the led_data becomes 0. If btn 2 is pressed then the count direction is up and the led_data becomes the value of the switches. If btn 1 is pressed then the count direction is set as down and the led_data becomes the value read on the switches. If btn 0 is pressed then the timer0 toggle is flipped. It then checks the toggle value and either stops timer 0, or resets and starts timer0. Before exiting the handler the btn interrupt will be cleared and the interrupts will be enabled.

```
void BTN_Intr_Handler(void *InstancePtr)
{
    // Disable GPIO interrupts
    XGpio_InterruptDisable(&BTNInst, BTN_INT);
    // Ignore additional button presses
    if ((XGpio_InterruptGetStatus(&BTNInst) & BTN_INT) !=
            BTN_INT) {
            return;
        }
    btn_value = XGpio_DiscreteRead(&BTNInst, 1);
    if(btn_value == 0x08){//BTN 3 pressed
        led_data = 0; //Leds start counting from 0 in current direction
    }
    else if(btn_value == 0x04){//BTN 2 pressed
            count_direction = 1;
            led_data = XGpio_DiscreteRead(&SWITCHInst, 1);//leds count up from current led_data
        }
    else if(btn_value == 0x02){//BTN 1 pressed
            count_direction = 0;
            led_data = XGpio_DiscreteRead(&SWITCHInst, 1);//leds count down from current led_data
        }
    else if(btn_value == 0x01){//BTN 0 pressed
        timer0_toggle = (timer0_toggle == 1)? 0: 1; //turn timer 0 on or off

        if(timer0_toggle == 1){//if timer toggle is on stop mode
            XTmrCtr_Stop(&TMR0Inst,0);//stop timer
        }
        else{
            XTmrCtr_Reset(&TMR0Inst,0);//reset timer
            XTmrCtr_Start(&TMR0Inst,0);//start timer
        }
    }

    (void)XGpio_InterruptClear(&BTNInst, BTN_INT);//clear btn interrupt
    // Enable GPIO interrupts
    XGpio_InterruptEnable(&BTNInst, BTN_INT);
}
```

4-1-9 The timer0 handler will be called every second. The handler disables timer0. It then checks to see if the count direction is up or down. If it is up then led_data increases by one. If it is down then the led_data will decrease by one. After that the data is written to the leds. Timer 0 is then reset and then started.

```
void TMR0_Intr_Handler(void *data)
{

    XTmrCtr_Stop(&TMR0Inst,0);//stop timer 0
    if(count_direction == 1){//if count direction indicates count up
        led_data++;//count up
    }
    else{//if count direction indicates count down
        led_data--;//count down
    }

    XGpio_DiscreteWrite(&LEDInst, 1, led_data);//update leds with current count
    XTmrCtr_Reset(&TMR0Inst,0);//reset timer 0
    XTmrCtr_Start(&TMR0Inst,0);//reset timer 1

    return;

}
```

4-1-10 The timer1 handler will be entered every microsecond. Upon entering the handler will disable timer1. The tmr1 count will be checked. If it is of 250k or .25 seconds then the tmr1 will reset its count back to 0. If the tmr1_count is less than 50k then the hcsr04 trigger echo routine may begin. If tmr1_count is 0 then the trigger will be started. If timr1_count is 10 then the trigger will end. Echo will be read from the hcsr04 driver function. If prev_echo and echo are 0 and 1 respectively then the echo_duration will begin. If prev_echo and echo are 1 and 0 respectively then the echo_duration will end. Echo duration will be equal to the time between the start and end of the echo. The distance will be calculated using a formula created by linear regression and data points. The distance will then be output to the terminal to be read. Prev_echo will be set to the current echo. Tmr1_count will be incremented. The timer1 will be reset and will start.

```
void TMR1_Intr_Handler(void *data)
{

    XTmrCtr_Stop(&TMR1Inst,0);//stop timer1 while in the handler
    tmr1_count = (tmr1_count == 250000)? 0: tmr1_count; // should restart trigger ever quarter second

    if(tmr1_count < 50000){

        if(tmr1_count == 0){ //beginning of trigger
            HCSR04_IP_mWriteReg(XPAR_HCSR04_IP_0_S_AXI_BASEADDR, HCSR04_IP_S_AXI_SLV_REG3_OFFSET, 0x01);
        }
        if(tmr1_count == 10){//end of trigger
            HCSR04_IP_mWriteReg(XPAR_HCSR04_IP_0_S_AXI_BASEADDR, HCSR04_IP_S_AXI_SLV_REG3_OFFSET, 0x00);

        }
        //read echo
        echo = HCSR04_IP_mReadReg(XPAR_HCSR04_IP_0_S_AXI_BASEADDR, HCSR04_IP_S_AXI_SLV_REG0_OFFSET);


        if((prev_echo == 0x00) & (echo == 0x01)){ // pos edge detect
            echo_duration = tmr1_count;
        }
        if((prev_echo == 0x01) & (echo == 0x00)){ //neg edge detect
            echo_duration = tmr1_count - echo_duration;//duration between posedge and negedge
            distance = (2024*tmr1_count) - 326130; //distance converstion * 100000
            distance /= 100000; // dividing by 100_000 to get it into inches
            xil_printf("Distance:\t %d inches\r\n", distance);//print distance to terminal
        }


        prev_echo = echo; // save current echo into prev echo for edge detection

    }

    tmr1_count += 1; //increment timer1 counter


    XTmrCtr_Reset(&TMR1Inst,0);//reset timer1
    XTmrCtr_Start(&TMR1Inst,0);//start timer1

    return;
}
```

4-1-11 This function taken from Ross elliot will setup the system for interrupts

```
int InterruptSystemSetup(XScuGic *XScuGicInstancePtr) //setup GIC
{
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                 (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                 XScuGicInstancePtr);
    Xil_ExceptionEnable();

    return XST_SUCCESS;
}
```

4-1-12 This function modified from a function Ross elliot made will take in 3 interrupts and initialize them with their handlers.

```
int IntcInitFunction(u16 DeviceId, XTmrCtr *TMR0InstancePtr, XTmrCtr *TMR1InstancePtr, XGpio *GpioInstancePtr)
{
    XScuGic_Config *IntcConfig;
    int status;
    // Interrupt controller initialisation
    IntcConfig = XScuGic_LookupConfig(DeviceId);
    status = XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig->CpuBaseAddress);
    if(status != XST_SUCCESS) return XST_FAILURE;
    // Call to interrupt setup
    status = InterruptSystemSetup(&INTCInst);
    if(status != XST_SUCCESS) return XST_FAILURE;
    // Connect GPIO interrupt to handler
    status = XScuGic_Connect(&INTCInst,
                             INTC_GPIO_INTERRUPT_ID,
                             (Xil_ExceptionHandler)BTN_Intr_Handler,
                             (void *)GpioInstancePtr);
    if(status != XST_SUCCESS) return XST_FAILURE;
    // Connect timer 0 interrupt to handler
    status = XScuGic_Connect(&INTCInst,
                             INTC_TMR0_INTERRUPT_ID,
                             (Xil_ExceptionHandler)TMR0_Intr_Handler,
                             (void *)TMR0InstancePtr);
    if(status != XST_SUCCESS) return XST_FAILURE;
    // Connect timer 1 interrupt to handler
    status = XScuGic_Connect(&INTCInst,
                             INTC_TMR1_INTERRUPT_ID,
                             (Xil_ExceptionHandler)TMR1_Intr_Handler,
                             (void *)TMR1InstancePtr);
    if(status != XST_SUCCESS) return XST_FAILURE;
    // Enable GPIO interrupts interrupt
    XGpio_InterruptEnable(GpioInstancePtr, 1);
    XGpio_InterruptGlobalEnable(GpioInstancePtr);
    // Enable GPIO and timer interrupts in the controller
    XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);
    XScuGic_Enable(&INTCInst, INTC_TMR0_INTERRUPT_ID);
    XScuGic_Enable(&INTCInst, INTC_TMR1_INTERRUPT_ID);
    return XST_SUCCESS;
}
```

4-1-13 This is the beginning of the main function where we will set up the GPIO Peripherals

```
//----------------------------------------------------
// MAIN FUNCTION
//----------------------------------------------------
int main (void)
{
    xil_printf("\r\nEntering main\r\n");

  int status;
  //----------------------------------------------------
  // INITIALIZE THE PERIPHERALS & SET DIRECTIONS OF GPIO
  //----------------------------------------------------
  // Initialise LEDs
  status = XGpio_Initialize(&LEDInst, LEDS_DEVICE_ID);
  if(status != XST_SUCCESS) return XST_FAILURE;
  // Initialise switches
  status = XGpio_Initialize(&SWITCHInst, SWITCH_DEVICE_ID);
  if(status != XST_SUCCESS) return XST_FAILURE;
  // Initialise Push Buttons
  status = XGpio_Initialize(&BTNInst, BTNS_DEVICE_ID);
  if(status != XST_SUCCESS) return XST_FAILURE;
  // Set LEDs direction to outputs
  XGpio_SetDataDirection(&LEDInst, 1, 0x00);
  // Set SWITCHES direction to outputs
  XGpio_SetDataDirection(&SWITCHInst, 1, 0x00);
  // Set all buttons direction to inputs
  XGpio_SetDataDirection(&BTNInst, 1, 0xFF);
```

4-1-14 This is the set up for timer0 with 1 second interval, interrupt, auto reload, and count down.

```
//----------------------------------------------------
// SETUP THE TIMER 0
//----------------------------------------------------
status = XTmrCtr_Initialize(&TMR0Inst, TMR0_DEVICE_ID); //Initialize timer 0
if(status != XST_SUCCESS) return XST_FAILURE; //check for failure
XTmrCtr_SetHandler(&TMR0Inst, TMR0_Intr_Handler, &TMR0Inst);//setup timer0 handler
XTmrCtr_SetResetValue(&TMR0Inst, 0, TMR0_LOAD);// set reset value
//set timer0 in interrupt mode, auto reload mode, and count down mode initially
XTmrCtr_SetOptions(&TMR0Inst, 0, XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION | XTC_DOWN_COUNT_OPTION);
```

4-1-15 This is the set up for timer1 with 1 microsecond interval, interrupt, auto reload, and count down.

```
//-----------------------------------------------------
// SETUP THE TIMER 1
//-----------------------------------------------------
status = XTmrCtr_Initialize(&TMR1Inst, TMR1_DEVICE_ID);//Initialize timer 1
if(status != XST_SUCCESS) return XST_FAILURE;//check for failure
XTmrCtr_SetHandler(&TMR1Inst, TMR1_Intr_Handler, &TMR1Inst);//setup timer1 handler
XTmrCtr_SetResetValue(&TMR1Inst, 0, TMR1_LOAD);//set reset value
//set timer1 in interrupt mode, auto reload mode, and count down mode initially
XTmrCtr_SetOptions(&TMR1Inst, 0, XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION | XTC_DOWN_COUNT_OPTION);
```

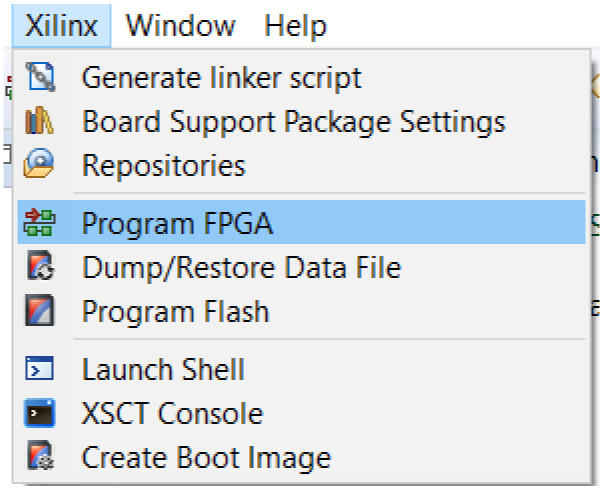
This is the interrupt setup for Timer0, Timer1, and the Buttons.

```
// Initialize interrupt controller
  //setup interrupt for Timer0, Timer1, and Buttons
status = IntcInitFunction(INTC_DEVICE_ID, &TMR0Inst, &TMR1Inst, &BTNInst);
if(status != XST_SUCCESS) return XST_FAILURE;//check for failure
```

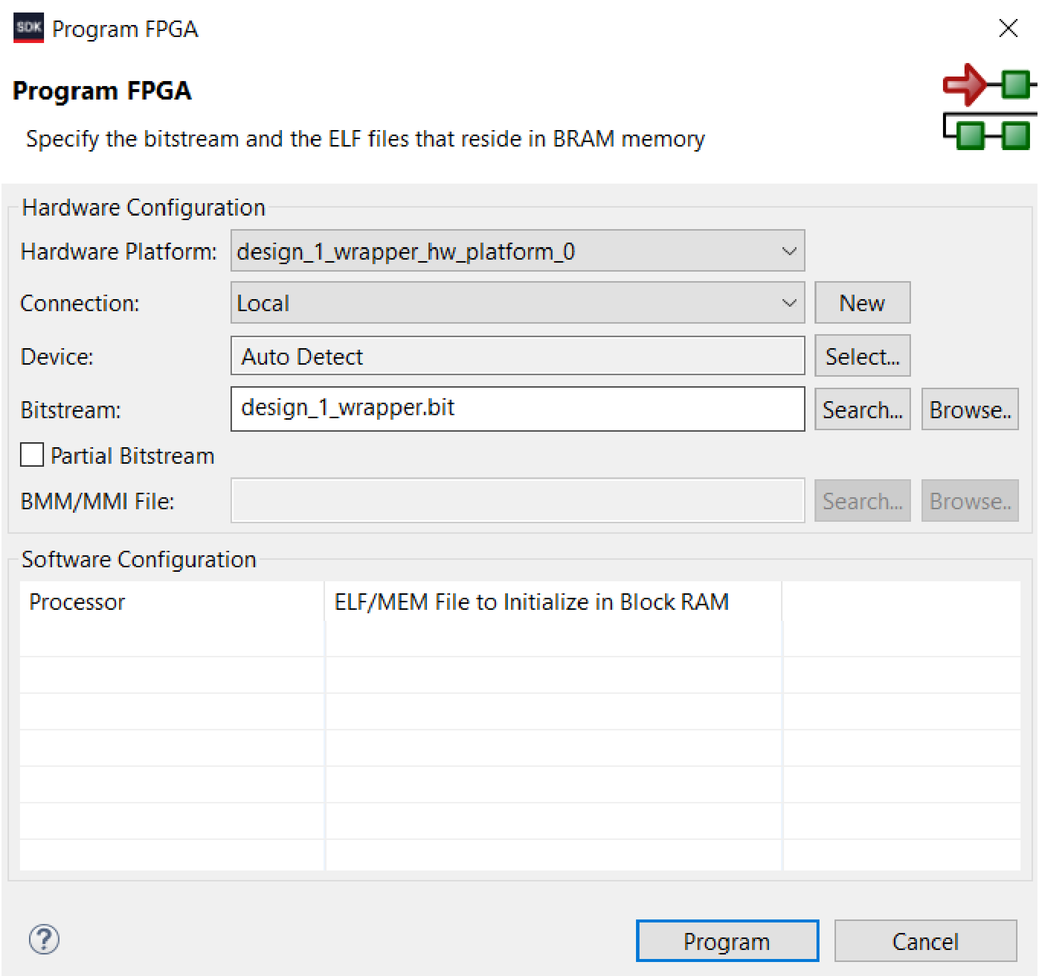
This is the end of the main function. The timers are started and there is an infinite while loop. The main function will do nothing more and the entire program will be run with timer and button interrupts.

```
  XTmrCtr_Start(&TMR0Inst, 0);//start timer 0
  XTmrCtr_Start(&TMR1Inst, 0);//start timer 1




  while(1);//inf loop

  return 0;
}
```

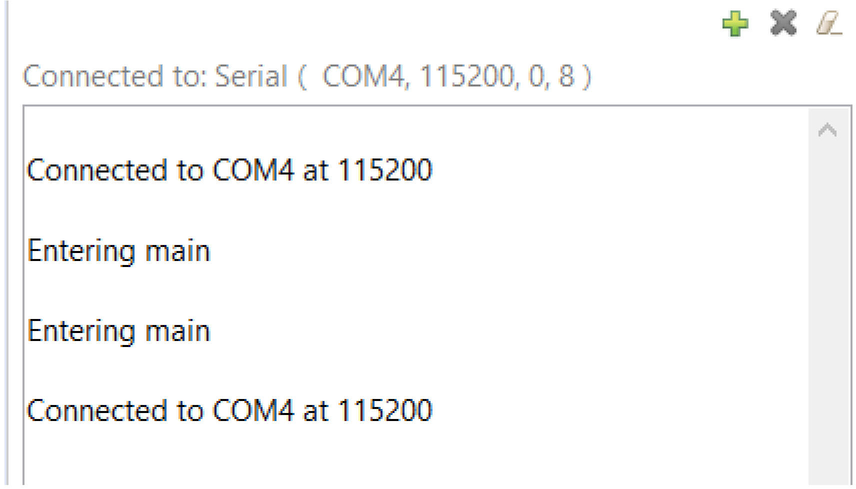
4-1-16 After everything the code is added the fpga can start getting ready to be programmed. Go to the xilinx tab and click on program fpga.



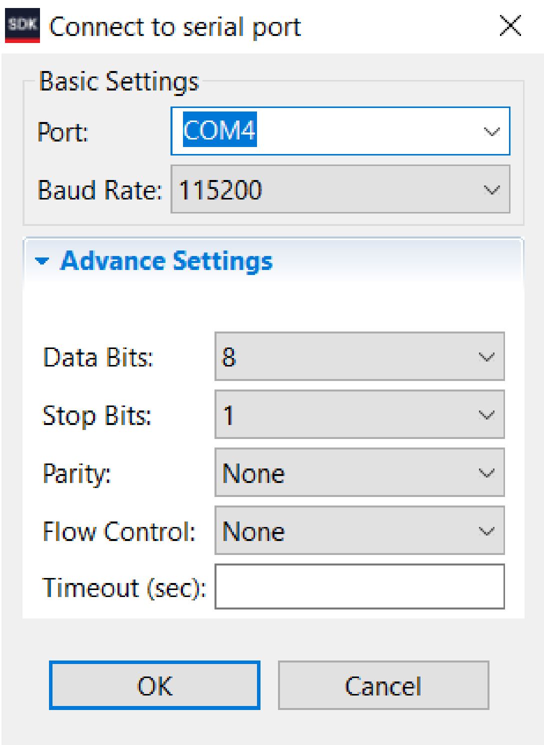
4-1-17 From there program the FPGA with the bit file



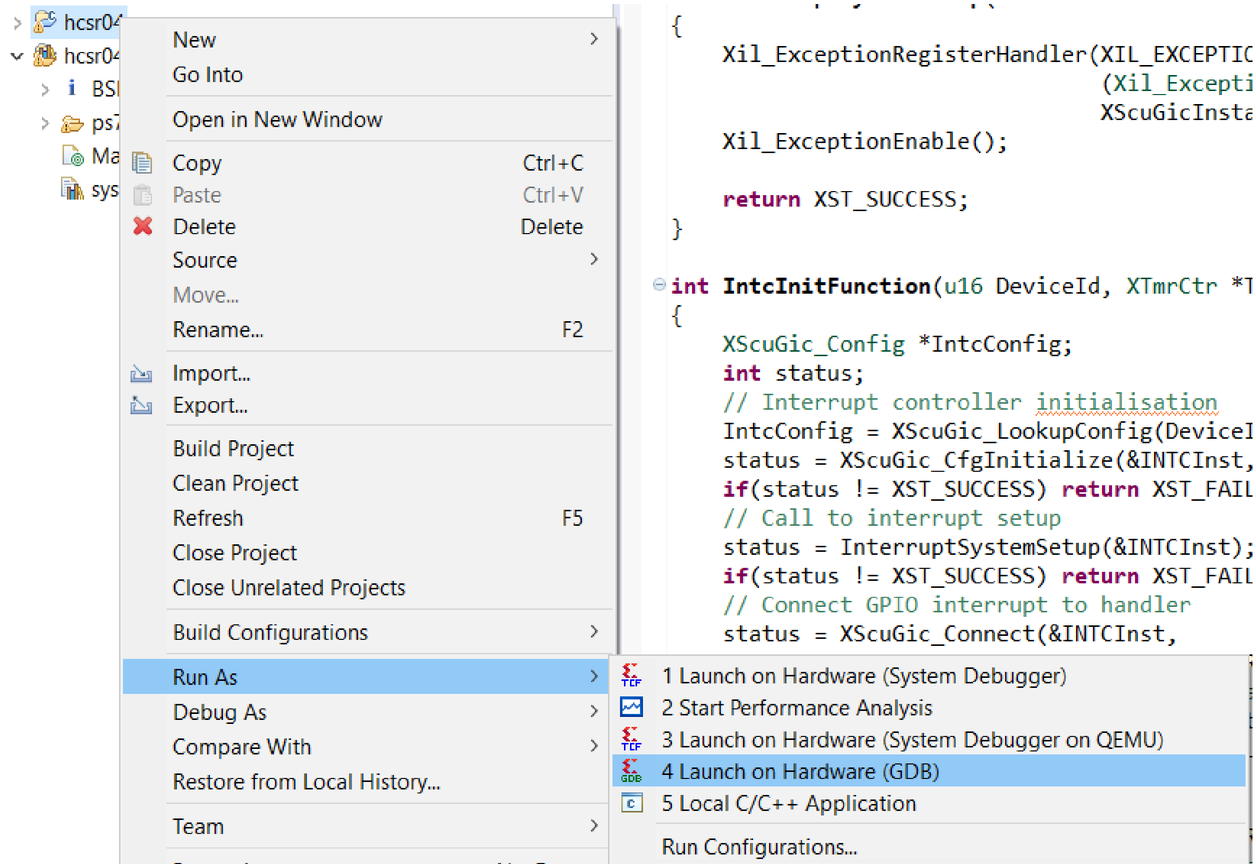
4-1-18 After that open the sdk terminal



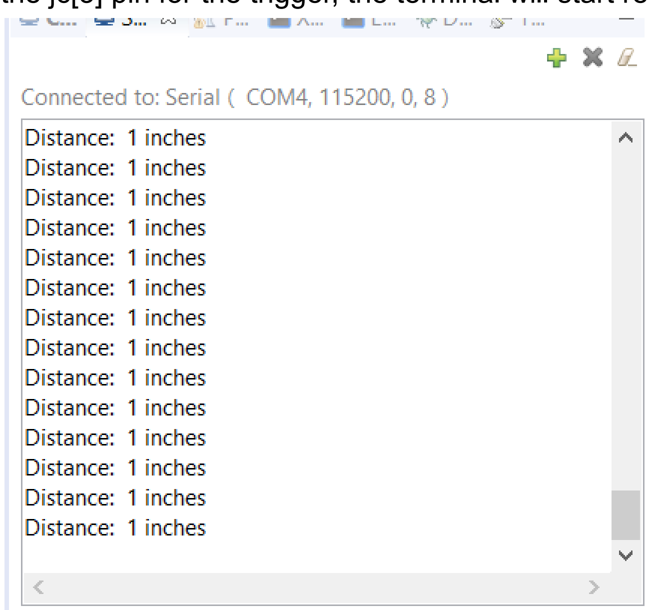
4-1-19 Click on the plus sign and connect the terminal to the FPGA for communication



4-1-20 Then go to the application project and right click. Go to run as and Launch on Hardware(GDB)



4-1-21 After all this and properly setting up an HCSR04 module on the jc[2] pin for the echo and the jc[3] pin for the trigger, the terminal will start receiving measurements from the sensor.



Conclusion

We create an HCSR04 custom ip is the IP packager with a trigger output signal and an echo input signal. We used a breadboard circuit to connect together a battery, 7805, HCSR04, and a Zybo board. We use Vivado to create a block design with our custom ip while enabling interrupts in the Zynq system. We then used the Vivado Sdk to create an application that uses the custom ip drivers alongside some timers to both control the led count on the board and measure distance with the sensor.