Zynq Book Tutorials
Zynq Processor System. The second half of the ECE3622 course will consider System-on-Chip (SoC) design for the processor System (PS) using the C language and the AXI/AMBA bus interface to the Programmable Logic (PL).
Zynq Processor System. Xilinx Zynq SoC has a dual core ARM Cortex A9 processor, the AMBA bus, connection devices (SPI, I2C, CAN, UART, GPIO, DMA), flash and DRAM memory controllers, security and programmable logic (PL) with DSP and RAM.
Zynq Processor System. The Verilog RTL projects in the first half of the ECE3622 course have introduced you to the Xilinx Vivado electronic design automation (EDA).

The ECE 3623 laboratory projects will now utilize the Zynq PS and a block-oriented design environment.
Zynq Processor System. The ECE3623 laboratory projects will include Xilinx IP blocks and various familiar design-reuse Pmod peripheral modules but now as IP blocks in the Vivado EDA.
Zynq Processor System. The Zynq Book Tutorials are posted on Blackboard. The Zynq Book Tutorials source codes in ZIP compressed format is also posted on Blackboard.
Zynq Processor System. The Zynq Book is posted on Blackboard and is a complete reference for the Zynq PS and AXI bus.
Zynq Processor System. The Zynq Book Tutorials first project implements a General Purpose Input/Output (GPIO) controller in the PL of the Zynq for the LEDs.

Exercise 1A Creating a First IP Integrator Design
Zynq Processor System. The GPIO controller connects to the LEDs to the Zynq PS with an AXI bus connection, allowing the LEDs to be controlled by a software application written in C.
Zynq Processor System. The template project begins with Exercise 1B: *Creating a Zynq System in Vivado* of with a block design as IP.
Zynq Processor System. An initial IP block is the ZYNQ7 Processing System:

Exercise 1B  Creating a Zynq System in Vivado
Zynq Processor System. The rst (reset) processing system, the AXI peripheral and the AXI GPIO template connected to the Zybo Board LEDs complete the block design.

Exercise 1B Creating a Zynq System in Vivado
Zynq Processor System. The template project continues with Exercise 1C: *Creating a Software Application with the SDK*. Here the C file is imported:
Zynq Processor System. The template project continues with Exercise 1C: *Creating a Software Application with the SDK*. The C file as a project on SDK is built in the Console:

```
16:47:37 **** Auto Build of configuration Debug for project LED_test ****
make all
'Building file: ../src/LED_test_tut_1C.c'
'Invoking: ARM gcc compiler'
arm-xilinx-ebi-gcc -Wall -O0 -g3 -c -fmessage-length=0 -MT"src/LED_test_tut_1C.o" -I../..//LED_test_bsp/ps7_cortexa9_0/include
'Finished building: ../src/LED_test_tut_1C.c'

'Building target: LED_test.elf'
'Invoking: ARM gcc linker'
arm-xilinx-ebi-gcc -Wl, -T -Wl,.../src/1script.1d -L../..//LED_test_bsp/ps7_cortexa9_0/lib -o "LED_test.elf" .../src/LED_test_tut
'Finished building target: LED_test.elf'

'Invoking: ARM Print Size'
arm-xilinx-ebi-size LED_test.elf | tee "LED_test.elf.size"
text  data   bss  dec  hex filename
 23684   1168  22580  47432   b948 LED_test.elf
'Finished building: LED_test.elf.size'

16:47:38 Build finished (took 820ms)
Zynq processor System. The C source for the initial project is `LED_test_tut_1C.c`.

/* GPIO driver initialization */
Status = XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}

The GPIO device driver is in `xgpio.h` and initialized with the unique designation `GPIO_DEVICE_ID`

#define GPIO_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID
/* GPIO device that LEDs are connected to */
Zynq processor System. The C source for the initial project is `LED_test_tut_1C.c`. Bits set to 0 are outputs.

```c
/* Set the direction for the LEDs to output. */
XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0x0);

/* Loop forever blinking the LED. */
while (1) {
    /* Write output to the LEDs. */
    XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, led);
    /* Flip LEDs. */
    led = ~led;
}
```

Exercise 1C Creating a Software Application in the SDK
Zynq Processor System.

Information and documentation for the peripheral drivers of the Zynq are found on the `system.mss` tab:
Zynq Processor System. The Exercise 1B and 1C results in a simple blinking LED task.
Expanding the Basic IP. The previous project in Vivado is augmented by adding additional GPIOs and configuring the system to utilize interrupts. Two GPIO controllers are used where one uses the push buttons to generate interrupts.

Exercise 2A Expanding the Basic IP Integrator Design
Expanding the Basic IP. The other GPIO controller will connect to the LEDs. Both will also be connected to the Zynq processor via an AXI bus connection, allowing the LEDs to be controlled by a software application.

Exercise 2B  Creating a Zynq System with Interrupts in Vivado
Expanding the Basic IP. Hardware interrupts from the push buttons trigger asynchronous functions in the Zynq PS. The GPIO interrupts must be enabled.

**Exercise 2B** Creating a Zynq System with Interrupts in Vivado
Expanding the Basic IP. A software application is created that utilizes hardware interrupts. The push buttons are used to increment a counter by different values. The count will be continuously displayed on the LEDs in binary form, where LED0 corresponds to the LSB and LED3 the MSB.
Expanding the Basic IP. The C source for the interrupt project is `interrupt_controller_tut_2B.c`.

Exercise 2C Creating a Software Application in the SDK
Expanding the Basic IP. Drivers and parameters configure and operate the GPIO. Detailed information for the drivers can be found in the `system.mss` file which explains the purpose of each function and the parameters passed to it.

Predesignated parameters can also be found in `xparameters.h`.

---

Exercise 2C Creating a Software Application in the SDK
Expanding the Basic IP. interrupt_controller_tut_2B.c.

#include "xparameters.h"
#include "xgpio.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_printf.h"

// Parameter definitions
#define INTC_DEVICE_ID XPAR_PS7_SCUGIC_0_DEVICE_ID
#define BTNS_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID
#define LEDS_DEVICE_ID XPAR_AXI_GPIO_1_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_Fabric_AXI_GPIO_0_IP2INTC_IRPT_INTR
Expanding the Basic IP. `interrupt_controller_tut_2B.c`

```c
#define BTN_INT XGPIO_IR_CH1_MASK

XGpio LEDInst, BTNInst;
XScuGic INTCInst;
static int led_data;
static int btn_value;
```
Expanding the Basic IP. *interrupt_controller_tut_2B.c.*

Interrupt handler:

```c
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;
    }
}
```
Expanding the Basic IP. *interrupt_controller_tut_2B.c.*

Interrupt handler:

```c
btn_value = XGpio_DiscreteRead(&BTNInst, 1);
// Increment counter based on button value
led_data = led_data + btn_value;

XGpio_DiscreteWrite(&LEDInst, 1, led_data);
(void)XGpio_InterruptClear(&BTNInst, BTN_INT);

// Enable GPIO interrupts
XGpio_InterruptEnable(&BTNInst, BTN_INT);
```
Expanding the Basic IP. *interrupt_controller_tut_2B.c.*

Prototype functions:

```c
static void BTN_Intr_Handler(void *baseaddr_p);
static int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);
static int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr);
```

---

**Exercise 2C** Creating a Software Application in the SDK

![Zynq Board Diagram]
Expanding the Basic IP. interrupt_controller_tut_2B.c.

Main:

```c
int main (void)
{
    int status;

    // Initialize LEDs
    status = XGpio_Initialize(&LEDInst, LEDSDEVICE_ID);
    if(status != XST_SUCCESS) return XST_FAILURE;
}
```
Expanding the Basic IP. interrupt_controller_tut_2B.c.

Main:

// Initialize 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 all buttons direction to inputs
XGpio_SetDataDirection(&BTNInst, 1, 0xFF);
Expanding the Basic IP. interrupt_controller_tut_2B.c.

Main:

```c
// Initialize interrupt controller
status = IntcInitFunction(INTC_DEVICE_ID, &BTNInst);
if(status != XST_SUCCESS) return XST_FAILURE;

while(1);
return 0;
```
Expanding the Basic IP. *interrupt_controller_tut_2B.c.*
Initial interrupt system setup:

```c
int InterruptSystemSetup(XScuGic *XScuGicInstancePtr) {
    // Enable interrupt
    XGpio_InterruptEnable(&BTNInst, BTN_INT);
    XGpio_InterruptGlobalEnable(&BTNInst);

    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IN
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                XScuGicInstancePtr);
    Xil_ExceptionEnable();

    return XST_SUCCESS;
}
```
Expanding the Basic IP. `interrupt_controller_tut_2B.c`. `InterruptSystemSetup(XScuGic *XScuGicInstancePtr);` initializes and configures the interrupt controller in the Zynq PS, connecting the interrupt handler to the interrupt source.

```c
int InterruptSystemSetup(XScuGic *XScuGicInstancePtr)
{
...
}
```

---

**Exercise 2C** Creating a Software Application in the SDK
Expanding the Basic IP. `interrupt_controller_tut_2B.c.`

`InterruptSystemSetup(XScuGic *XScuGicInstancePtr);` also enables the interrupt sources and registers exceptions.

```c
int InterruptSystemSetup(XScuGic *XScuGicInstancePtr)
{
    ...
}
```
Expanding the Basic IP. \textit{interrupt\_controller\_tut\_2B\_c.}

Initial setup:

```c
int IntcInitFunction(u16 Deviceld, XGpio *GpioInstancePtr)
{
    XScuGic_Config *IntcConfig;
    int status;

    // Interrupt controller initialization
    IntcConfig = XScuGic_LookupConfig(Deviceld);
    status = XScuGic_CfgInitialize(&INTCInst, IntcConfig,
                                    IntcConfig->CpuBaseAddress);
    if(status != XST_SUCCESS) return XST_FAILURE;
}
```
Expanding the Basic IP. *interrupt_controller_tut_2B.c.* The interrupt controller *XScuGic INTCInst* handles all incoming interrupt requests passed to the PS and performs the function associated with each interrupt source.

```c
// Interrupt controller initialization
IntcConfig = XScuGic_LookupConfig(DeviceId);
status = XScuGic_CfgInitialize(&INTCInst, IntcConfig,
                               IntcConfig->CpuBaseAddress);
if(status != XST_SUCCESS) return XST_FAILURE;
```
Expanding the Basic IP. `interrupt_controller_tut_2B.c`. The interrupt controller `XScuGic INTCInst` is also capable of prioritizing multiple interrupt sources to the requirements of the application.

```c
// Interrupt controller initialization
IntcConfig = XScuGic_LookupConfig(DeviceId);
status = XScuGic_CfgInitialize(&INTCInst, IntcConfig,
                                IntcConfig->CpuBaseAddress);
if(status != XST_SUCCESS) return XST_FAILURE;
```
Expanding the Basic IP. *interrupt_controller_tut_2B.c.*

Initial setup:

```c
// 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);
```
Expanding the Basic IP. `interrupt_controller_tut_2B.c. BTN_Intr_Handler(void *InstancePtr);` is the interrupt handler function for the push buttons and is called every time an interrupt request from the push buttons in the PL is received in the PS.

```c
void BTN_Intr_Handler(void *InstancePtr)
{
    ...
}
```

---

**Exercise 2C** Creating a Software Application in the SDK

![Xilinx Vivado](image-url)
Expanding the Basic IP. *interrupt_controller_tut_2B.c.*

`BTN_Intr_Handler(void *InstancePtr)` performs a counter increment on each call and displays the value of the counter on the LEDs in binary.

```c
void BTN_Intr_Handler(void *InstancePtr)
{
    ...
}
```

**Exercise 2C** Creating a Software Application in the SDK
Expanding the Basic IP. *interrupt_controller_tut_2B.c*. Initial setup:

```c
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);

return XST_SUCCESS;
```
Expanding the Basic IP. interrupt_controller_tut_2B.c. The counter increments by different values based on the push button which is pressed.

You should be able to determine the value assigned to each of the push buttons (BTNU, BTND, BTNLS, BTNR and BTNC) On the Zybo Board.
Expanding the Basic IP. An AXI Timer is an additional source of interrupts that is added to the project created in Exercise 2B. The AXI Timer is in the IP Catalog.
Expanding the Basic IP. However, since there is already an interrupt connected to the input of the PS, an additional IP block is used to concatenate two interrupt requests into one interrupt signal.
Expanding the Basic IP. Right-click anywhere on the PS block design and select Add IP. Enter concat in the search field and add the IP Concat to the design.
Expanding the Basic IP. The Concat IP block is used to connect the two interrupt sources: the AXI Timer and the push button from the GPIO.

Exercise 2D Adding a Further Interrupt Source
Expanding the Basic IP. The complete design:

Exercise 2D Adding a Further Interrupt Source
IP Creation in HDL. Zynq devices have both a processing system (PS) and programmable logic (PL). Existing Verilog HDL datapath module for the Pmod peripherals are packaged with an IP interface that is compatible with the PS known as the AXI interface.
IP Creation in HDL. When creating IP in Verilog HDL, Vivado provides a set of AXI interface templates which can be created and customized via the *Create and Package IP Wizard*.
IP Creation in HDL. The *Create and Package New IP* facilitates two major functions:

- The packaging of existing source files into an IP package which is compatible with the IP Integrator tool.
- The creation of AXI4 IP peripherals.
IP Creation in HDL. An AXI4-Lite bus interconnection IP adds functionality to allow the LEDs on the Zybo Board to be controlled with software on the Zynq PS.
IP Creation in HDL. The Peripheral Details dialogue allows you to specify the Name and Version for the new peripheral with the IP Location as C:/Zynq_Book/ip_repro (the default location).

![Peripheral Details dialogue](image)
IP Creation in HDL. The *Add Interfaces* dialogue specifies the AXI4 interface in the custom peripheral as:

- Number of interfaces
- Interface type (AXI-Lite, AXI-Stream or AXI-Full)
- Interface mode (slave or master)
- Interface data width
IP Creation in HDL. The *Create Peripheral* dialogue details the output files which will be created. The *Edit IP* option creates the IP peripheral files and create a new Vivado project where the functionality of the peripheral can be modified in the source HDL code and then packaged.
IP Creation in HDL. If Verilog was specified the files are:

- `led_controller_v1_0.v` which instantiates the AXI-Lite interfaces.
- `led_controller_v1_0_S00_AXI.v` which contains the AXI4-Lite interface functionality for the interactions between the peripheral in the PL and the software running on the PS.
IP Creation in HDL. The IP Packager pane opens.
IP Creation in HDL. Functionality can now be added to the led_controller peripheral. A new output port connects the LED pins on the Zynq device and the value received from the Zynq PS is outputted to the new output port.

The `led_controller_v1_0_S00_AXI.v` is edited by double-clicking on it in the Sources pane.
IP Creation in HDL. The Zynq Book Tutorial describes the IP packaging and archiving of the led_controller:
IP Creation in HDL. The *led_controller_v1_0_S00_AXI.vhd* is described but can be easily converted to Verilog.

```vhdl
-- Users to add ports here
LEDs_out : out std_logic_vector(3 downto 0);

-- Add user logic here
LEDs_out <= slv_reg0(3 downto 0);
```

This is the entry port declaration.
IP Creation in HDL. The `led_controller_v1_0.vhd` is described but can be easily converted to Verilog.

```vhdl
-- Users to add ports here
LEDs_out : out std_logic_vector(3 downto 0);
-- component declaration
port ( 

LEDs_out : out std_logic_vector(3 downto 0);
```

Inside the port declaration add:
IP Creation in HDL. The *led_controller_v1_0.vhd* is described but can be easily converted to Verilog.

```
-- Instantiation of Axi Bus Interface S00_AXI

port map (    

LEDS_out => LEDs_out,  
```

Inside the component port map add:

This is the port mapping from the top module to the AXI4-Lite interface.
IP Creation in HDL. Next, return to the Package IP - led_controller and select Customization Parameters:

The Customization Parameters Wizard updates the IP Packager for the changes in the HDL source files.
IP Creation in HDL. When the IP peripheral is created a set of software driver files were generated. The SDK is set to those driver files by adding a new repository to the SDK project.
IP Creation in HDL. The *Zynq Book Tutorial* describes the SDK repository and drivers of the *led_controller*.
IP Creation in HDL.
After the driver and the software
`led_controller_test_tut_4A.c` is opened and compiled the
Exercise 4A is ready to run on the Zybo Board.
IP Creation in HDL. Exercise 4A utilizes the `xil_printf` compact diagnostic print routine and the SDK Terminal.

```c
xil_printf("led_controller IP test be,
xil_printf("---------------------

/* Loop forever */
while(1){
    while(led_val<=LED_LIMIT){
        /* Print value to terminal
        xil_printf("LED value: %d
        /* Write value to led con
```
IP Creation in HDL. The SDK Terminal displays the LED value.

```c
xil_printf("led_controller IP test begin.

/* Loop forever */
while(1){
    while(led_val<=LED_LIMIT){
        /* Print value to terminal */
        xil_printf("LED value: %d
        /* Write value to led_controller */
        LED_CONTROLLER_mWriteReg(led_val);
        /* increment LED value */
        led_val++;
```

Connect icon  Terminal tab
End of Zynq Book Tutorials