Laboratory 6 – 7-Segment Applications

I. 7-Segment Display

Objectives

- To build an application to control the hardware 7-segment display on the Basys3 FPGA board
- To learn how to organize source and header files of a project

Activity Summary

1. Launch Vitis 2022.2 after opening and exporting the hardware platform
2. Create the 7-SegDisplay application
3. Program FPGA, setup Run Configuration, and run the application
4. Repeat activity 3 for changes made to the application source file

Introduction

What is 7-segment Display?

A 7-segment display is a widely used electronic display device designed to show decimal digits and some alphanumeric characters. It consists of seven individual light-emitting segments labeled A through G, arranged in a figure-eight pattern, plus an optional decimal point (DP).

By illuminating different combinations of segments, the display can represent numbers 0-9 and selected letters or symbols. For instance, displaying the digit "1" requires only segments B and C to be lit, while displaying "8" requires all seven segments to be activated. There are two main types of 7-segment displays: Common Anode, where all segment anodes are tied together and a LOW signal illuminates the segment, and Common Cathode, where all segment cathodes are tied together and a HIGH signal illuminates the segment. Due to their simplicity, low cost, high readability, and ease of interfacing with microcontrollers, 7-segment displays remain prevalent in embedded systems and digital circuit design.

Ref Electronics Tutorials. "BCD to 7-Segment Display Decoder."

https://www.electronics-tutorials.ws/combination/comb_6.html

Digilent Reference. "Basys 3 FPGA Board Reference Manual." https://digilent.com/reference/programmable-logic/basys-3/reference-manual

Hardware Description: Basys3 7-Segment Display Circuit

In this section you are required to complete an application that drives a 7-segment display in software. The circuit description of the 7-segment display on the Basys3 board and the expected illumination pattern are shown in Figure 1.

- There are four digits and each digit has 7 segments (A to G) and a decimal point (DP).
- All the **cathodes** of each segment are tied together. As a result a '**0**' on any segment's **anode** turns that segment **ON** while a '**1**' turns it **OFF**.
- The 7 segments pins (CA to CG) and the DP pin are shared by all the digits.
- To select which of the digits uses the segments and DP pins, a multiplexing mechanism is used. Transistors **controlled by** AN0 to AN3 are used to multiplex the digits' access to the segment and DP pins.
- A '**0**' on AN0 will select the **rightmost digit** (digit position 1).

- Note that only one digit **should** be selected at a time **to display the intended number correctly**.

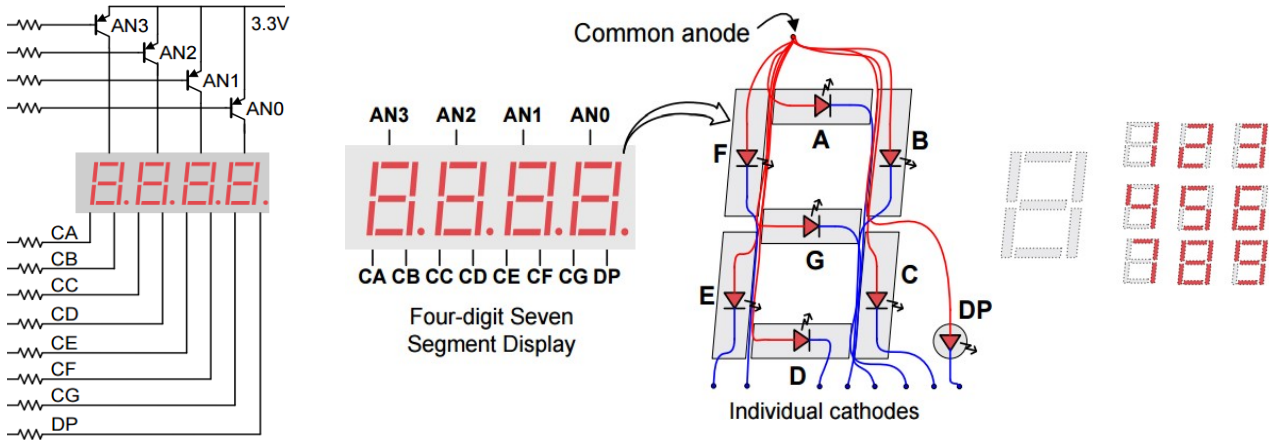

Figure 1: The 7-segment common-anode circuit and illumination pattern

7-Segment Display Pinout Description

The pin description for connecting to the 7-segment hardware is given in Tables 1 and 2. To write a binary-coded decimal (BCD) to any digit, the XGpio variable SEG7_HEX_OUT is used and the corresponding digit is selected by writing to the XGpio variable SEG7_SEL_OUT. These variables have been declared in the gpio_init.h file and initialized in the gpio_init.c file. Look inside the folder **Lab6 - Codes folder** for these files.

Table 1: Segment selection pin out

Segment Selection	DP	CG	CF	CE	CD	CC	CB	CA
SEG7_HEX_OUT	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

Table 2: Digit selection pin out

Digit Selection	AN3	AN2	AN1	AN0
SEG7_SEL_OUT	Bit3	Bit2	Bit1	Bit0

Refresh Frequency

Since all the anodes of the display are tied together, a multiplexer is used to select which digit is activated at a time. This requires that for all the digits to be lit, a time slice has to be given to each one at a frequency fast enough to avoid flickering. The recommended configuration uses a refresh period of 20 ms (50 Hz refresh frequency) with a per-digit display time of 5 ms, achieved by setting the FIT Timer to 500000 clock cycles at 100 MHz system clock. While the theoretical minimum refresh period is 16 ms (4 ms per digit), practical testing has shown that a 20 ms refresh period provides better visual stability and eliminates flickering across different lighting conditions and viewing angles, as the slightly longer display time ensures adequate LED brightness and allows margin for interrupt processing overhead. Alternative configurations range from a minimum of 400000 cycles (4 ms per digit, 62.5 Hz) which may cause flickering, to a maximum of 1000000 cycles (10 ms per digit, 25 Hz) with lower refresh rate, though the 500000 cycles configuration remains the optimal choice for most applications.

Hardware Timer ISR

For this lab, the hardware timer has been redesigned to generate an interrupt every **5 ms (0.005 seconds)**. The interrupt service routine will be used to drive each digit. See pages 15 to 17 of the "Basys3™ FPGA Board Reference Manual" for more information on how to drive the 7-segment display.

Binary-Coded Decimal for Driving the Display

How do we write numbers to the 7-segment display? Let us assume for instance that we want to write the number '1' to the **rightmost digit** of the display:

- The number in its original binary form is 0001.
- We have to write this to the 7 segments and DP of the digit.
- To do so, we need to encode the decimal number into an 8-bit binary and make a selection of AN0 as shown in Figure 2.
- Remember that a **'0'** turns **ON** a segment while a **'1'** turns it **OFF** (common anode configuration).
- Therefore, this encoding gives us the 8-bit binary number **11111001** for the segments and 4-bit number **0111** for the digit selection. Note that the DP is turned off.
- As a result we would write **11111001** (0xF9) to SEG7_HEX_OUT and **0111** (0b0111) to SEG7_SEL_OUT.
- Table 3 shows the BCD codes for some of the other values that can be written to a digit. Note that in all these codes the decimal point is not used.

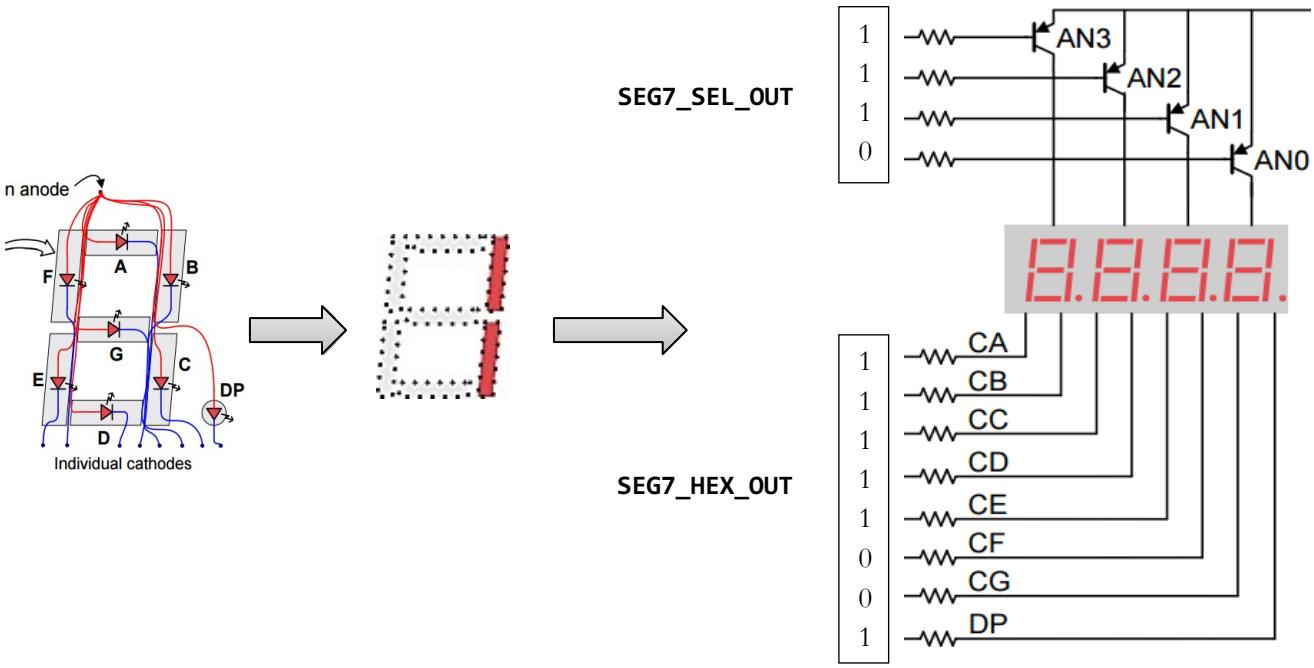

Figure 2: Segment and digit selection for writing a '1' to the 7-segment display

Table 3: Digit display BCD codes

Digit Value	SEG7_HEX_OUT (Hex)
Blank	1111 1111 (0xFF)
0	1100 0000 (0xC0)
1	1111 1001 (0xF9)
2	1010 0100 (0xA4)
3	1011 0000 (0xB0)
4	1001 1001 (0x99)
5	1001 0010 (0x92)
6	1000 0010 (0x82)
7	1111 1000 (0xF8)
8	1000 0000 (0x80)
9	1001 0000 (0x90)
Dash	1011 1111 (0xBF)

Project Files Organization

C applications are often organized in different files. All the macros, global variables and header file inclusions are usually placed in one or more header files (**.h files**), and functions are grouped in one or more source files (**.c files**). The main routine itself is placed in a file named **main.c**. The header files are then included in all the necessary source files. This structure has been used in the 7-segment applications and the subsequent ones.

Specifications

Implement a basic 7-segment display driver that continuously displays a fixed 4-digit number (1234) on the Basys3 FPGA board using interrupt-driven multiplexing.

Functionality

Hardware Initialization

- Initializes GPIO peripherals for 7-segment display control (SEG7_HEX_OUT, SEG7_SEL_OUT)
- Configures interrupt system to connect hardware timer ISR
- Handles initialization failures with error messages via UART console
- **Note**: FIT timer is pre-configured to 500000 clock cycles (5 ms period @ 100 MHz)

Display Operation

- Displays fixed number **1234** on 4-digit 7-segment display
- Uses time-multiplexed scanning: each digit lit for 5 ms
- Achieves 50 Hz refresh rate (20 ms complete cycle)
- Interrupt Service Routine triggers every 5 ms to update active digit

Display Control Flow

- Converts 4-digit number into individual digits (1, 2, 3, 4)
- Sequentially activates each digit position (AN0-AN3)
- Writes appropriate segment pattern to SEG7_HEX_OUT
- Selects corresponding digit via SEG7_SEL_OUT

Activities

1. Open and Export Hardware Design

Follow the procedure described in Lab 1 instructions to unzip and open **Lab 6 hardware Files** and export the hardware design files **Top.xsa (including bitstream)**.

As we mentioned in Lab 1 instructions to export the hardware platform and create the XSA file.

1. Click **File → Export → Export Hardware** (Figure 1a)
2. The Export Hardware Platform wizard opens (Figure 1b). Click **Next** to start the process.
3. In the Output window (Figure 1c), select **Include bitstream** to ensure the complete hardware implementation is included, then click **Next**.
4. In the Files window (Figure 1d), specify the XSA filename (default: "top") and export location. The system shows "The XSA will be written to: top.xsa". Click **Next**.
5. Review the summary (Figure 1e) showing that a hardware platform named 'top' will be created as top.xsa with post-implementation model and bitstream. Click **Finish** to complete the export.

The XSA file is now ready for use in Vitis IDE.

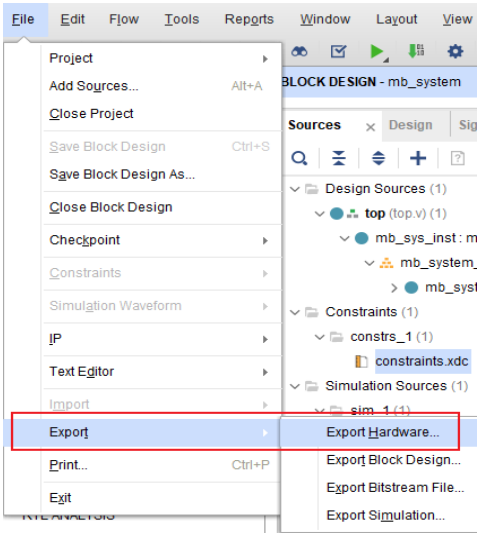

Figure 1(a)

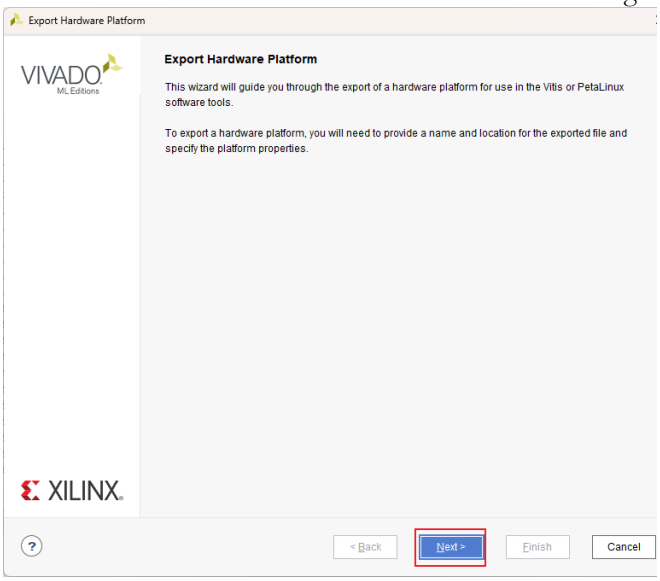

Figure 1(b)

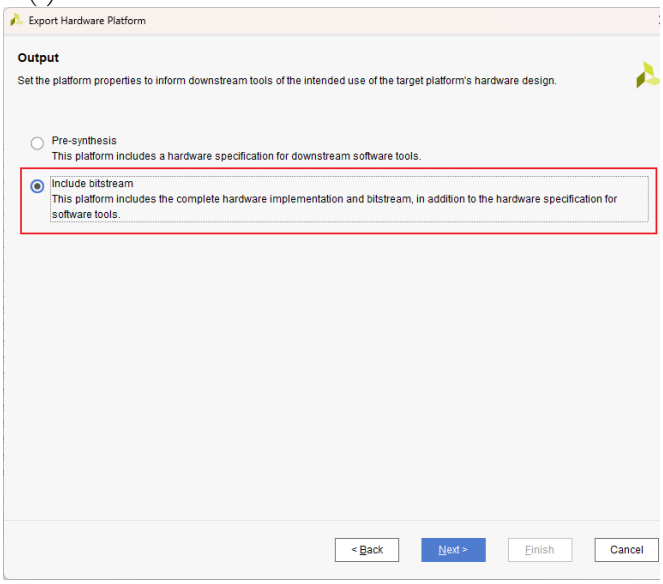

Figure 1(c)

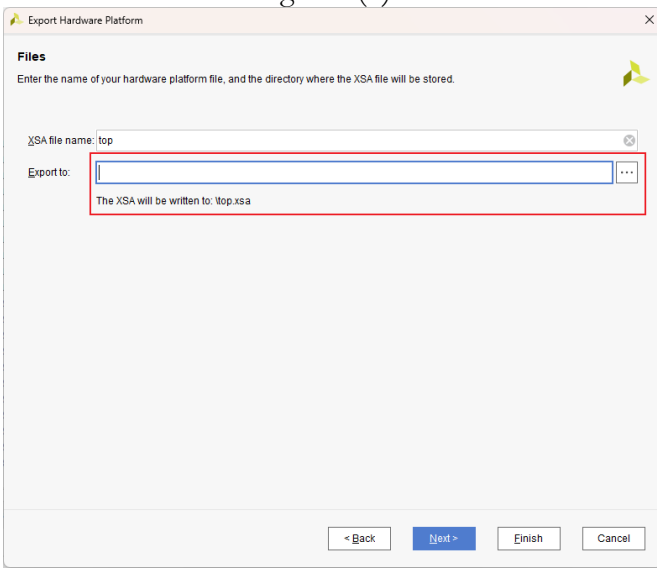

Figure 1(d)

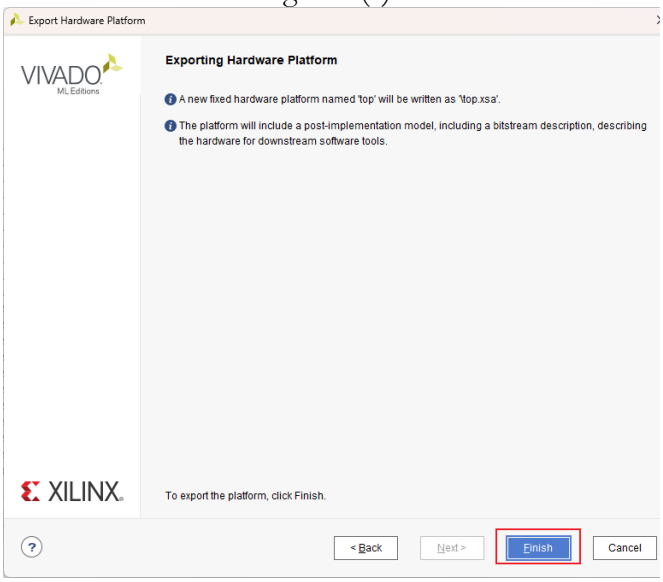

Figure 1(e)

2. Launch Vitis

Create the new folder in the PC, named Lab6, then open Vitis and select the file path of Lab6 as the workspace, and select **Launch**. **(Note: If you use Lab PC, please do not create the workspace on**

the uni-cloud)

3. **Create a New Application Project** as demonstrated in previous lab exercises.

Based on your images, here's the revised description for creating an application project:

Create a New Application Project

Launch Vitis IDE and follow these steps to create an application project:

Step 1: Launch Vitis IDE The Vitis IDE main interface will open (Figure 2a), displaying the welcome page. In the PROJECT section, click **Create Application Project** to create an application project.

Step 2: Select Platform In the New Application Project wizard (Figure 2b), select **Create a new platform from hardware (XSA)**, then click the **Browse...** button to locate the XSA file exported earlier from Vivado (the top.xsa file as shown in Figure 2c).

Step 3: Configure Platform On the platform selection page (Figure 2d), browse and select the top.xsa file from your project directory. The XSA file contains the hardware specification exported from Vivado.

Step 4: Configure Application Project On the Application Project Details page (Figure 2e), enter the application project name (such as " **7_Segment_Display** "). The system will automatically create an associated system project " 7_Segment_Display _system" with target processor shown as "microblaze_0".

Step 5: Select Domain On the domain selection page (Figure 2f), the system displays details for the "standalone_microblaze_0" domain, including processor microblaze_0 and 32-bit architecture.

Step 6: Choose Template On the template selection page (Figure 2g), select the **Hello World** template from the embedded software development templates. This template creates a "Hello World" program written in C.

Step 7: Complete Creation Click **Finish** to complete the application project creation. Vitis IDE will create the application project and open the complete development environment, displaying the project structure in the Explorer.

Once created, you can begin developing and debugging your Lab5 embedded application.

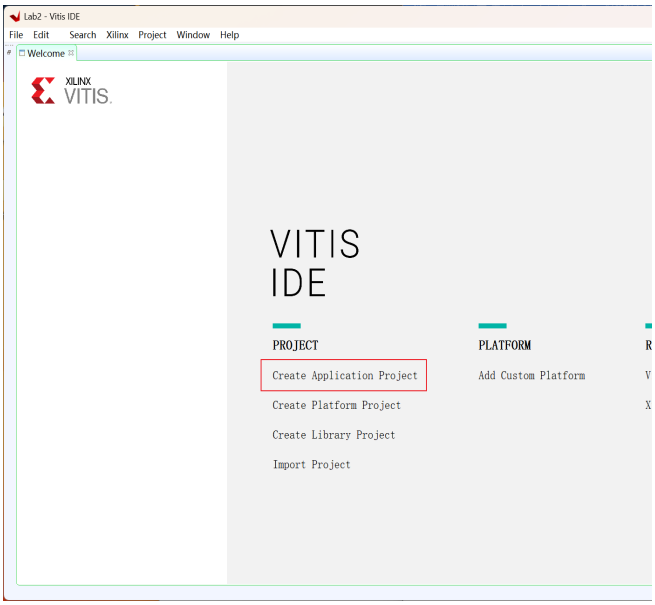

Figure 2(a)

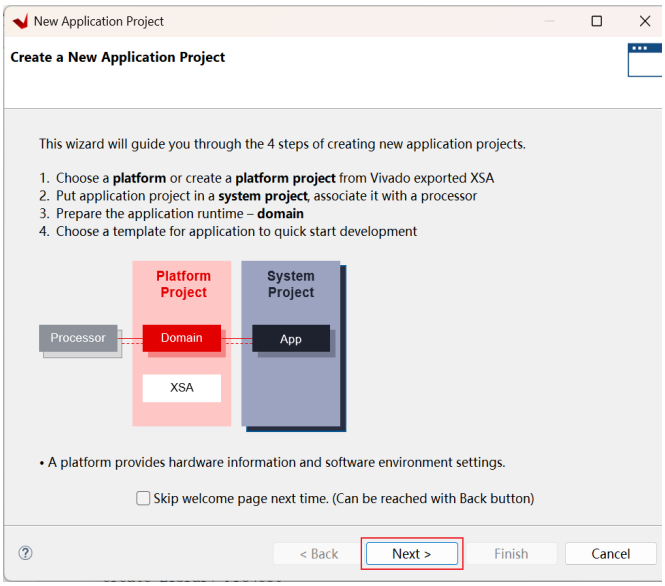

Figure 2(b)

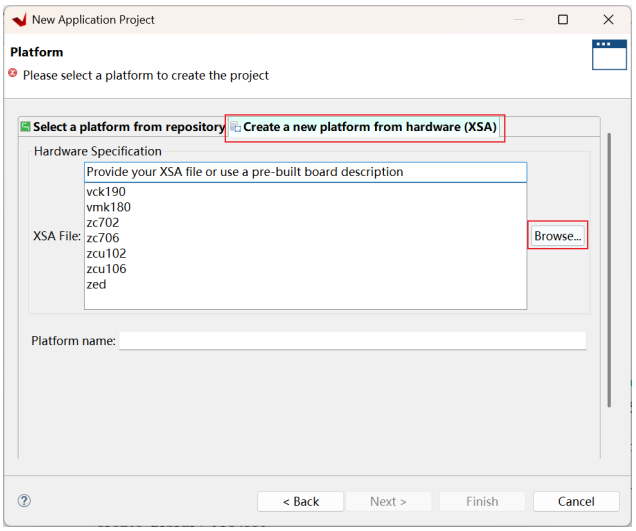

Figure 2(c)

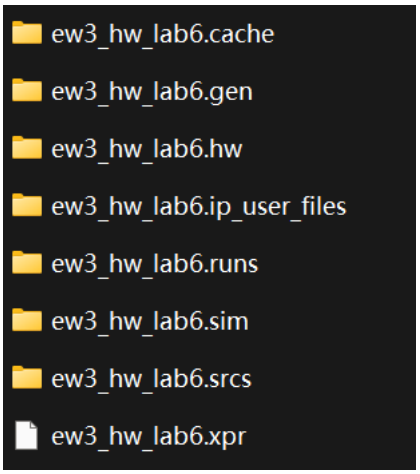

Figure 2(d)

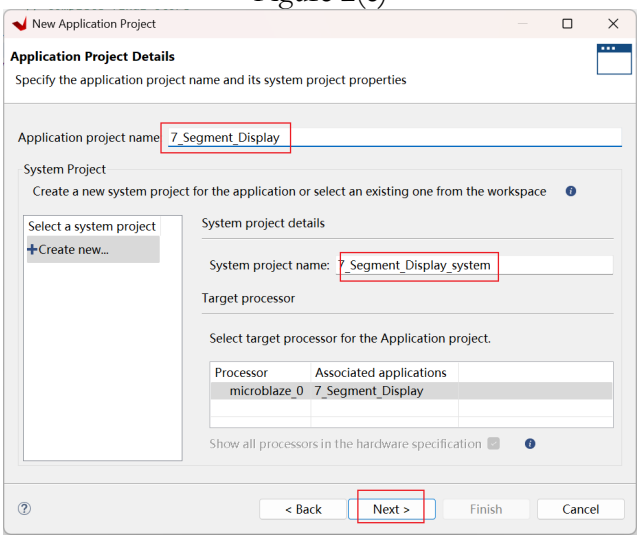

Figure 2(e)

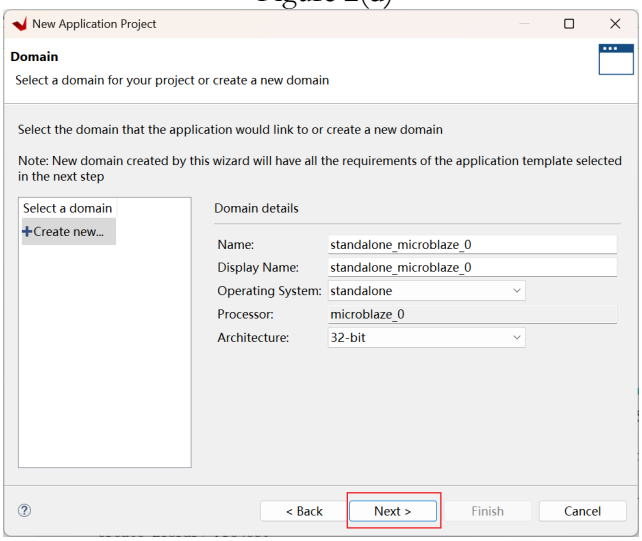

Figure 2(f)

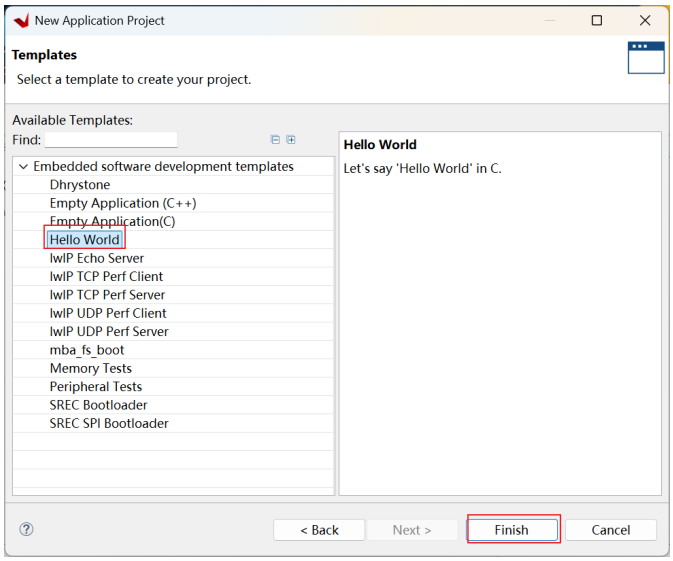

Figure 2(g)

4. Create and Rename the Source File

- As shown in the Figure 3, In **Project Explorer**, expand the 7_Segment_Display→ **src** folder.
- Right-click on helloworld.c, select **Rename…**, and rename the file to 7_Segment_Display.c.

- And add the gpio_init.c, gpio_init.h, seg7_display.c, seg7_display.h, xinterruptES3.c and timer_interrupt_func.c, which you can find at learn Lab 6 codes Folder.

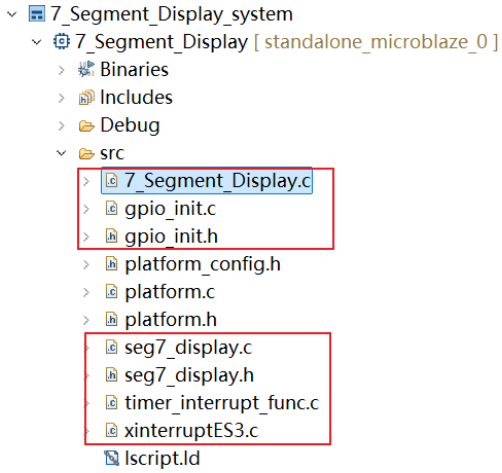

Figure 3

- The following section aims to elaborate on the functional implementation of the newly added source files in the project. For clarity, detailed explanations have been integrated directly into the code segments as comments.

  This application's display logic is built around four key methods, all declared in seg7_display.h and defined in seg7_display.c. The main function you'll call is displayNumber(u16 number), which prepares the data for the screen. Internally, it uses the calculateDigits(u16 number) helper function to break down the input into individual digits. Since the hardware has only four digits, it can only show numbers between 0 and 9999; you'll need to modify the displayNumber function to show dashes ("----") if the input is out of this range. The actual display update is driven by a timer interrupt: the hwTimerISR() method is called periodically, and its job is to execute displayDigit(), which handles the low-level task of lighting up the correct segments. To help you, much of this code is already written and well-commented, and you'll be guided to finish the implementation.

  The code shown in Figure 4(a)-(h):

  Figure 4(a): Seg7_display.h

  Figure 4(b): Seg7_display.c

  Figure 4(c): gpio_init.h

  Figure 4(d): gpio_init.c

  Figure 4(e): timer_interrupt_func.c

```c
#ifndef __SEG7_DISPLAY_H_ // Include Guard.
#define __SEG7_DISPLAY_H_

#include "xgpio.h" // Include GPIO driver header.

// --- MACRO DEFINITIONS ---

// Define the segment patterns for a Common Anode 7-segment display.
// A '0' bit turns a segment ON, and a '1' bit turns it OFF.
// The bits correspond to segments (dp, g, f, e, d, c, b, a).
#define DIGIT_BLANK     0xFF // All segments OFF.
#define DIGIT_ZERO      0xC0 // Pattern for '0'.
#define DIGIT_ONE       0xF9 // Pattern for '1'.
#define DIGIT_TWO       0xA4 // Pattern for '2'.
#define DIGIT_THREE     0xB0 // Pattern for '3'.
#define DIGIT_FOUR      0x99 // Pattern for '4'.
#define DIGIT_FIVE      0x92 // Pattern for '5'.
#define DIGIT_SIX       0x82 // Pattern for '6'.
#define DIGIT_SEVEN     0xF8 // Pattern for '7'.
#define DIGIT_EIGHT     0x80 // Pattern for '8'.
#define DIGIT_NINE      0x90 // Pattern for '9'.
#define DIGIT_DASH      0xBF // Pattern for '-'.

// Define logical numbers for the special characters for easier use in code.
#define NUMBER_BLANK    10
#define NUMBER_DASH     11

// Define the digit selection patterns. This is for an active-low common anode display.
// A '0' selects (enables) the corresponding digit.
#define EN_FIRST_SEG    0b0111 // Enable digit 1 (left-most).
#define EN_SECOND_SEG   0b1011 // Enable digit 2.
#define EN_THIRD_SEG    0b1101 // Enable digit 3.
#define EN_FOURTH_SEG   0b1110 // Enable digit 4 (right-most).

// --- FUNCTION PROTOTYPES ---

// Declares the function to set up the interrupt system (implementation likely in xinterruptES3.
int setUpInterruptSystem(void);

// Declares the Interrupt Service Routine (ISR) for the hardware timer.
void hwTimerISR(void *CallbackRef);

// Declares the function to display a 16-bit unsigned number.
void displayNumber(u16 number);

// Declares the function to split a number into its individual digits.
void calculateDigits(u16 number);

// Declares the function that drives the GPIOs to show a single digit.
void displayDigit(void);

#endif // End of include guard.
```

Figure 4(a)

```c
#include <stdio.h>
#include "xil_types.h"
#include "seg7_display.h"
#include "gpio_init.h"

// --- GLOBAL VARIABLES ---
u8 digitDisplayed = FALSE; // A flag for synchronization between the main loop and the ISR.
                           // The ISR sets it to TRUE after displaying a digit.
u8 digits[4];              // An array to store the individual digits of the number to be displayed.
u8 numOfDigits;            // The number of active digits in the number (e.g., 3 for the number 123).
u8 digitToDisplay;         // The actual digit value (0-9) to be shown in the current cycle.
u8 digitNumber;            // The position (1-4) of the digit to be displayed in the current cycle.

/**
 * @brief Displays a number up to 9999 on the 4-digit display.
 * @param number The 16-bit unsigned integer to display.
 */
void displayNumber(u16 number)
{
    u8 count;

    if (number <= 9999) // Check if the number is within the displayable range.
    {
        calculateDigits(number); // Split the number into individual digits and store them in the 'digits'
        count = 4; // Start displaying from the right-most digit (digit 4).

        // Loop to display each valid digit of the number.
        while (count > 4 - numOfDigits)
        {
            digitToDisplay = digits[count-1]; // Set the digit value to be displayed.
            digitNumber = count;              // Set the digit position to be enabled.
            count--;

            // This is a busy-wait loop. It waits for the ISR to finish displaying the previous digit.
            // The ISR will set digitDisplayed to TRUE.
            while (digitDisplayed == FALSE);

            // Reset the flag, indicating that we are now preparing the next digit.
            digitDisplayed = FALSE;
        }
    }
    else // If the number is out of range.
    {
        // Display "----" to indicate an overflow or error.
        count = 1;
        while (count < 5)
        {
            digitToDisplay = NUMBER_DASH; // Set the character to a dash.
            digitNumber = count;          // Select each digit position in turn.
            count++;
            while (digitDisplayed == FALSE); // Wait for the ISR.
            digitDisplayed = FALSE;          // Reset the flag.
        }
    }
}


   /**
    * @brief Splits a u16 number into four separate digits.
    * @param number The number to split (0-9999).
    */
   void calculateDigits(u16 number)
   {
       u8 fourthDigit, thirdDigit, secondDigit, firstDigit;

       // Use integer division and modulo arithmetic to extract each digit.
       if (number > 999) // 4-digit number.
       {
           numOfDigits = 4;
           fourthDigit  = number % 10;
           thirdDigit   = (number / 10) % 10;
           secondDigit  = (number / 100) % 10;
           firstDigit   = number / 1000;
       }
       else if (number > 99) // 3-digit number.
       {
           numOfDigits = 3;
           fourthDigit  = number % 10;
           thirdDigit   = (number / 10) % 10;
           secondDigit  = number / 100;
           firstDigit   = 0; // The most significant digit is not used.
       }
       else if (number > 9) // 2-digit number.
       {
           numOfDigits = 2;
           fourthDigit  = number % 10;
           thirdDigit   = number / 10;
           secondDigit  = 0;
           firstDigit   = 0;
       }
       else // 1-digit number.
       {
           numOfDigits = 1;
           fourthDigit  = number % 10;
           thirdDigit   = 0;
           secondDigit  = 0;
           firstDigit   = 0;
       }

       // Store the calculated digits in the global array.
       digits[0] = firstDigit;
       digits[1] = secondDigit;
       digits[2] = thirdDigit;
       digits[3] = fourthDigit;
   }
```

```c
/**
 * @brief Drives the GPIOs to display a single digit at its specified position.
 * This function is designed to be called rapidly by a timer interrupt.
 */
void displayDigit(void)
{
    // Use a switch statement to write the correct segment pattern to the segment GPIO port
    // based on the global 'digitToDisplay' variable.
    switch (digitToDisplay)
    {
        case NUMBER_BLANK:
            XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_BLANK);
            break;
        case 0: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_ZERO); break;
        case 1: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_ONE); break;
        case 2: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_TWO); break;
        case 3: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_THREE); break;
        case 4: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_FOUR); break;
        case 5: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_FIVE); break;
        case 6: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_SIX); break;
        case 7: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_SEVEN); break;
        case 8: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_EIGHT); break;
        case 9: XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_NINE); break;
        case NUMBER_DASH:
            XGpio_DiscreteWrite(&SEG7_HEX_OUT, 1, DIGIT_DASH);
            break;
        default:
            break;
    }

    // Write the correct selection pattern to the digit selector GPIO port
    // based on the global 'digitNumber' variable to enable the correct digit.
    if (digitNumber == 1)
        XGpio_DiscreteWrite(&SEG7_SEL_OUT, 1, EN_FIRST_SEG);
    else if (digitNumber == 2)
        XGpio_DiscreteWrite(&SEG7_SEL_OUT, 1, EN_SECOND_SEG);
    else if (digitNumber == 3)
        XGpio_DiscreteWrite(&SEG7_SEL_OUT, 1, EN_THIRD_SEG);
    else if (digitNumber == 4)
        XGpio_DiscreteWrite(&SEG7_SEL_OUT, 1, EN_FOURTH_SEG);

    // Set the flag to TRUE to signal to the main loop (in displayNumber)
    // that the display has been updated and it can prepare the next digit.
    digitDisplayed = TRUE;
}
```

Figure 4(b)

```c
#ifndef __GPIO_INIT_H_ // Include Guard to prevent this header from being included multiple tim
#define __GPIO_INIT_H_

#include "xgpio.h" // Include the header file for the Xilinx GPIO driver library.

// Function prototype for the GPIO initialization function.
// It is defined in gpio_init.c.
XStatus initGpio(void);

// Use the 'extern' keyword to declare global variables.
// These variables are defined in gpio_init.c, and 'extern' makes them
// accessible to any other source file that includes this header.
// Each variable is an instance of the XGpio driver struct.

extern XGpio SEG7_HEX_OUT;  // For the GPIO connected to the 7-segment display's segment lines
extern XGpio SEG7_SEL_OUT;  // For the GPIO that selects which of the 4 digits is active.
extern XGpio P_BTN_LEFT;    // For the GPIO connected to the left push-button.
extern XGpio P_BTN_RIGHT;   // For the GPIO connected to the right push-button.
extern XGpio LED_OUT;       // For the GPIO connected to the LEDs.
extern XGpio SLIDE_SWITCHES;// For the GPIO connected to the slide switches.

#endif // End of the include guard.
```

Figure 4(c)

```c
#include "gpio_init.h" // Include the corresponding header file to get the variable declarations.

// Define the global GPIO driver instances that were declared with 'extern' in the header file.
// Memory is allocated for these structures here.
XGpio SEG7_HEX_OUT;
XGpio SEG7_SEL_OUT;
XGpio P_BTN_LEFT;
XGpio P_BTN_RIGHT;
XGpio LED_OUT;
XGpio SLIDE_SWITCHES;

XStatus initGpio(void)
{
    XStatus status; // Variable to hold the return status of function calls for error checking.

    // Initialize the GPIO for the 7-segment display's segment data lines.
    status = XGpio_Initialize(&SEG7_HEX_OUT, 0);
    if (status != XST_SUCCESS) return XST_FAILURE; // If initialization fails, return immediately.

    // Initialize the GPIO for the 7-segment display's digit selection lines. Device ID is 1.
    status = XGpio_Initialize(&SEG7_SEL_OUT, 1);
    if (status != XST_SUCCESS) return XST_FAILURE;

    // Initialize the GPIO for the LEDs. Device ID is 2.
    status = XGpio_Initialize(&LED_OUT, 2);
    if (status != XST_SUCCESS) return XST_FAILURE;

    // Initialize the GPIO for the left push-button. Device ID is 4.
    status = XGpio_Initialize(&P_BTN_LEFT, 4);
    if (status != XST_SUCCESS) return XST_FAILURE;

    // Initialize the GPIO for the right push-button. Device ID is 5.
    status = XGpio_Initialize(&P_BTN_RIGHT, 5);
    if (status != XST_SUCCESS) return XST_FAILURE;

    // Initialize the GPIO for the slide switches. Device ID is 7.
    status = XGpio_Initialize(&SLIDE_SWITCHES, 7);
    if (status != XST_SUCCESS) return XST_FAILURE;

    // If all initializations were successful, return success.
    return XST_SUCCESS;
}
```

Figure 4(d)

```c
#include "seg7_display.h" // Include header to get the prototype for displayDigit().

/**
 * @brief The Interrupt Service Routine (ISR) for the hardware timer.
 * @param CallbackRef A reference passed during interrupt setup, not used here.
 *
 * This function is automatically called by the processor whenever the timer interrupt occurs.
 * It should be as short and fast as possible.
 */
void hwTimerISR(void *CallbackRef)
{
    // Each time the timer interrupt fires, call the displayDigit() function.
    // This refreshes one digit on the 7-segment display.
    displayDigit();
}
```

Figure 4(e)

5. Include Required Header Files

- Open 7_Segment_Display.c
- Remove the following two lines of code:

  print("Hello World\n\r");

  print("Successfully ran Hello World application");

- Add the code shown in Figure 5(a)-(h).

Figure 5(a): Header files and variable declarations

Figure 5(b): Main function

```
#include <stdio.h>
#include "platform.h"
#include "xil_types.h"      // Added for integer type definitions
#include "seg7_display.h"
#include "gpio_init.h" // Added for 7-segment definitions


u16 slideSwitchIn = 0;
```

Figure 5(a)

```
int main()
{
    init_platform();
    int status;

    // Initialize the GPIOs
    status = initGpio();
    if (status != XST_SUCCESS) {
        print("GPIOs initialization failed!\n\r");
        cleanup_platform();
        return 0;
    }

    // Setup the Interrupt System
    status = setUpInterruptSystem();
    if (status != XST_SUCCESS) {
        print("Interrupt system setup failed!\n\r");
        cleanup_platform();
        return 0;
    }

    while (1)
    {
        // Call the method to display number
        displayNumber(1234);

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

Figure 5(b)

6. **Now that you have completed the code**

   (1) **Program the FPGA**,
   (2) **Setup Run Configurations**, and
   (3) **Run the application**.

   The number 1234 should be seen on the display. This is because the `displayNumber(1234)` is called in the 7_Segment_Display.c file.

   As a simple exercise, you can change the number to be displayed and rerun the application.

7. The 7-segment output shown in Figures 6:

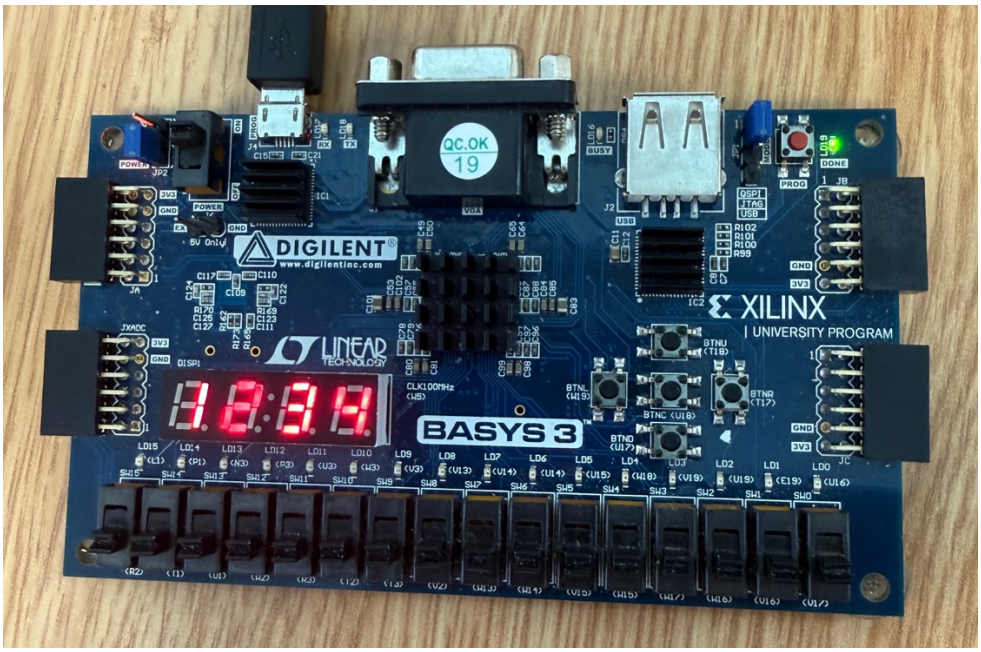

Figure 6

II. Button Counter

Objectives

- To learn how to detect whether a button is pressed
- To build an application to display how many times a button is pressed

Activity Summary

1. Create an application named ButtonCounter
2. Write the code for the application
3. Program the FPGA (if necessary), setup Run Configuration, and run the application

Specifications

The application demonstrates a simple digital up-counter controlled by a push-button, with the output shown on a 4-digit 7-segment display.

Functionality

- **Digital Counter**
  Implements a digital counter that starts at 0.
  The counter value is stored in a 16-bit unsigned integer (u16).
- **User Input**
  Uses the left push-button (P_BTN_LEFT) as the sole input to increment the counter.
  The counter increments only after a full **press-and-release** cycle of the button, ensuring one increment per press.
- **System Initialization**
  Initializes all required GPIO peripherals for the button and the 7-segment display.
  Sets up a timer-based interrupt system to handle the display refresh logic.

7-Segment Display Output

- The current value of the counter is continuously displayed on the 4-digit 7-segment display.
- The display is driven by the displayNumber() function, which is updated in the main loop.
- The underlying interrupt system ensures the display is refreshed dynamically for a stable, flicker-free output.

Example User Flow

1. **Initial State:** On power-up, the counter is 0, and the display show 0.
2. **First Press:** The user presses and releases the left push-button. The counter increments to 1, and the display updates to show 1.
3. **Second Press:** The user presses and releases the button again. The counter increments to 2, and the display shows 2.
4. **Holding Press:** If the user presses and holds the button down, the counter value does **not** change. It will only increment after the button is released.

Activities

The activities here involve counting the number of times a button on the Basys3 board has been pressed and displaying this on the 7-segment display. Create the ButtonCounter application. The file structure here is similar to that of the 7-segment display application and the files needed are already included in the "Lab6 - Codes folder". The following steps have already been implemented for you. Just follow them to understand what has been done.

8. As shown in Figure 7, Create a new application with the name **Button_Counter** and rename **helloworld.c** to **Button_Counter.c**

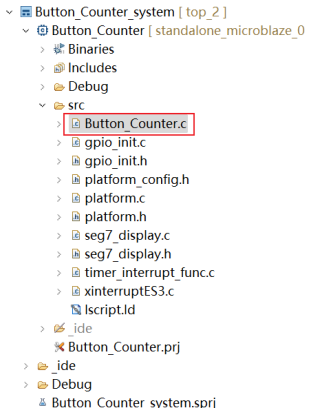

Figure 7

9. Write the code as shown in Figure 8(a)-(b).

Figure 8(a): Header files and function declarations

Figure 8(b): Main function

```c
#include <stdio.h>
#include "platform.h"
#include "xil_types.h"      // Added for integer type definitions like u16.
#include "seg7_display.h"   // Added for 7-segment display function declarations.
#include "gpio_init.h"      // Added for GPIO setup function declarations.
```

Figure 8(a)

```c
int main()
{
    // Initialize the platform (enables caches, etc.).
    init_platform();
    int status; // Variable to hold function return status for error checking.

    status = initGpio();
    if (status != XST_SUCCESS) {
        print("GPIOs initialization failed!\n\r");
        cleanup_platform();
        return 0; // Exit if GPIOs fail to initialize.
    }

    // This is critical for the background timer that refreshes the 7-segment display.
    status = setUpInterruptSystem();
    if (status != XST_SUCCESS) {
        print("Interrupt system setup failed!\n\r");
        cleanup_platform();
        return 0; // Exit if interrupts fail to set up.
    }

    u16 pushBtnLeftIn = 0; // Variable to store the state of the left push-button.
    u16 counter = 0;       // The counter value that will be displayed.

    while (1)
    {
        // Continuously call displayNumber. This function prepares the digit data
        // that the background timer interrupt will use to refresh the display.
        displayNumber(counter);

        // Read the current state of the left push-button.
        // The function returns 1 if pressed, 0 otherwise.
        pushBtnLeftIn = XGpio_DiscreteRead(&P_BTN_LEFT, 1);

        // Check if the button has been pressed (a rising edge is detected).
        if (pushBtnLeftIn == 1) {
            // This inner loop waits for the button to be released.
            // It prevents the counter from incrementing multiple times if the button is held down.
            while (pushBtnLeftIn == 1)
            {
                // Continuously poll the button until it is released (returns to 0).
                pushBtnLeftIn = XGpio_DiscreteRead(&P_BTN_LEFT, 1);

                /* Call displayNumber() again inside this waiting loop.
                 * This ensures that the display-driving ISR always has the most
                 * current data, even while the main loop is "stuck" here.
                 */
                displayNumber(counter);
            }
            // Increment the counter ONLY AFTER the button has been released.
            counter++;
        }
    }

    // This part of the code is unreachable in a normal embedded system
    // due to the infinite while(1) loop.
    cleanup_platform();
    return 0;
}
```

Figure 8(b)

Note: This program demonstrates a simple digital up-counter on a Xilinx MicroBlaze embedded system. It uses a push-button for user input and a 4-digit 7-segment display for visual output. The code includes several essential header files: platform.h for basic system initialization, xil_types.h for integer definitions like u16, xgpio.h for the GPIO driver, and the custom headers gpio_init.h and seg7_display.h which contain the specific setup and logic for the project's hardware. The program's functionality is divided into two main parts: an interrupt-driven display system and a polling-based input handler. The display mechanism is managed by a background timer interrupt configured by the setUpInterruptSystem() function. This interrupt periodically calls an Interrupt Service Routine (ISR) that handles the rapid multiplexing of the four digits on the 7-segment display, creating a stable and flicker-free image. The main application loop interfaces with this system by calling the displayNumber() function, which simply prepares the numerical data for the ISR to use. The counter logic is implemented using a robust "press-detect, wait-for-release" method to handle user input. When the main loop detects that the left push-button has been pressed, it enters a nested while loop. This inner loop effectively pauses the main logic, continuously polling the button's state until it is released. The counter variable is incremented only after the button has been let go. This strategy ensures that one complete physical press results in exactly one increment, preventing the counter from increasing uncontrollably if the user holds the button down. The main function begins by initializing the platform, the GPIOs, and the critical interrupt system. Inside its infinite while(1) loop, it constantly calls displayNumber() to keep the display data current and checks the push-button's status. If a press is detected, it waits for the release and then increments the counter. This creates a simple yet reliable interactive counter, with the current value always visible on the 7-segment display.

III. Advanced Counter

Objectives

- To build a counter application for buttons and switches
- To learn how to detect the pressing of different buttons and switches

Activity Summary

1. Create a new application called AdvancedCounter
2. Write the application code
3. Program the FPGA (if necessary), setup Run Configuration, and run the application

Specifications

The application implements a dual-mode interactive counter system on the Basys3 board. It features two distinct counters whose values are manipulated and displayed based on user input from two push-buttons and a set of slide switches.

Functionality

- **Dual-Mode Operation**
  - The system maintains two independent 16-bit counters: a simple **incrementing counter** (counter) and a **switch accumulator** (switchCounter).
  - A displayMode flag determines which counter's value is currently shown on the 7-segment display.
  - Pressing a push-button performs an action on its corresponding counter and immediately switches the display to show the result.
- **Left Push-Button (P_BTN_LEFT)**
  - Controls the simple **incrementing counter**.
  - Each press-and-release cycle increments the counter by 1.
  - The counter wraps around from **9999 back to 0**.
  - Activates **Display Mode 0**, showing the counter value.
- **Right Push-Button (P_BTN_RIGHT) & Slide Switches**
  - Controls the **switch accumulator**.
  - On a press-and-release, the system reads the 4-bit value (0-15) from the slide switches (SLIDE_SWITCHES).
  - This value is added to the switchCounter. The total wraps around modulo 10000.
  - Activates **Display Mode 1**, showing the switchCounter value.
- **Input Handling**
  - Both buttons use a software delay for **debouncing** to prevent false triggers from mechanical noise.
  - The logic includes a "wait-for-release" loop to ensure that one physical press results in only **one action**.

7-Segment Display Output

- The 4-digit display shows the value of the counter corresponding to the currently active displayMode.
- The display is managed by an interrupt-driven system, ensuring a stable, flicker-free output while the main loop polls for user input.

Example User Flow

1. **Initial State:** The system starts in Mode 0. Both counters are 0, and the display shows 0.
2. **Press Left Button:** The user presses and releases the left button. The counter becomes 1, and the display updates to 1.
3. **Set Switches:** The user sets the slide switches to 5 (binary 0101).
4. **Press Right Button:** The user presses and releases the right button. The system reads 5 from the switches, adds it to switchCounter (now 5), and switches to Mode 1. The display updates to show 5.
5. **Press Left Button Again:** The user presses the left button. The counter increments to 2, and the system switches back to Mode 0. The display updates to show 2.

Activities

For this application, only the left (**BTNL**) and right (**BTNR**) buttons will be used for input, as the center button (**BTNC**) acts as a hardware reset for the MicroBlaze processor. The software should be designed to count the number of times **BTNL** is pressed and show this total on the 7-segment display. Additionally, when the right button (**BTNR**) is pressed, the program must display the current value of the slide switches instead. The following steps are suggested to achieve this.

1. As shown in Figure 9, Create a new application with the name **Advanced_Counter** and rename **helloworld.c** to **Advanced_Counter.c**

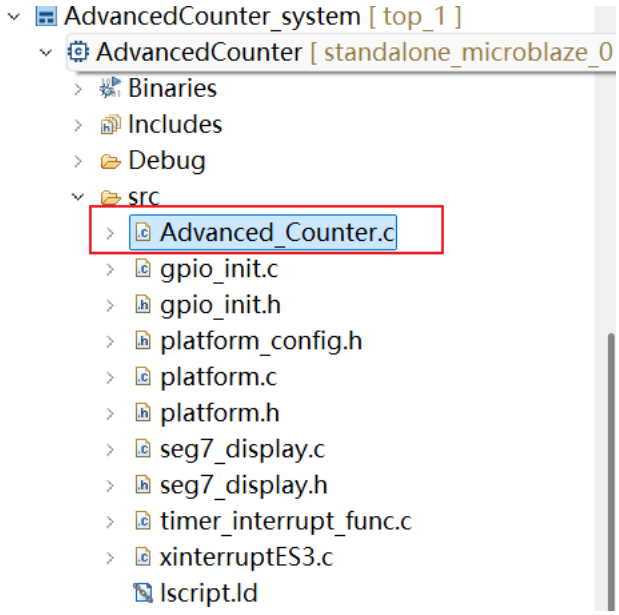

Figure 9

2. Write the code as shown in Figure 10(a)-(b).

Figure 9(a): Header files and function declarations

Figure 9(b): Main function

```
#include <stdio.h>
#include "platform.h"
#include "xil_types.h"
#include "seg7_display.h"
#include "gpio_init.h"
```

Figure 10(a)

```c
int main(void)
{
    int status;                // To store return values from functions for error checking.
    u16 counter = 0;           // A simple incrementing counter controlled by the left button.
    u16 switchCounter = 0;     // A counter that accumulates values from the slide switches.
    u8 displayMode = 0;        // A flag to control what is displayed: 0 for 'counter', 1 for 'switchCour

    init_platform();           // Initialize the platform (e.g., enable caches).

    // Initialize all GPIO peripherals.
    status = initGpio();
    if (status != XST_SUCCESS) {
        print("GPIO init failed!\n\r");
        return XST_FAILURE;
    }
    // Set up the interrupt system, which is essential for the 7-segment display driver.
    status = setUpInterruptSystem();
    if (status != XST_SUCCESS) {
        print("Interrupt setup failed!\n\r");
        return XST_FAILURE;
    }
    print("System ready!\n\r");

    // --- Main Application Loop ---
    while (1)
    {
        if (displayMode == 0) {
            // Mode 0: Display the value of the simple 'counter'.
            displayNumber(counter);
        } else {
            // Mode 1: Display the value of the 'switchCounter'.
            displayNumber(switchCounter);
        }

        if (XGpio_DiscreteRead(&P_BTN_LEFT, 1) == 1)
        {
            // A simple software delay loop for button debouncing.
            // 'volatile' prevents the compiler from optimizing this loop away.
            for (volatile int i = 0; i < 100000; i++);

            // After the delay, check the button again to confirm it's a valid press, not just noise.
            if (XGpio_DiscreteRead(&P_BTN_LEFT, 1) == 1) {
                // Action: If counter is 9999, wrap it around to 0; otherwise, increment.
                counter = (counter >= 9999) ? 0 : counter + 1;
                // Switch the display mode to show the 'counter'.
                displayMode = 0;
                // Wait here until the button is released. This ensures one action per press.
                while (XGpio_DiscreteRead(&P_BTN_LEFT, 1) == 1);
            }
        }
    if (XGpio_DiscreteRead(&P_BTN_RIGHT, 1) == 1)
    {
        // Same debouncing delay.
        for (volatile int i = 0; i < 100000; i++);

        // Re-check the button state to confirm the press.
        if (XGpio_DiscreteRead(&P_BTN_RIGHT, 1) == 1) {
            // Read the slide switches and apply a bitmask '& 0x0F'.
            // This keeps only the lower 4 bits, resulting in a value from 0 to 15.
            u16 switchVal = XGpio_DiscreteRead(&SLIDE_SWITCHES, 1) & 0x0F;

            // Add the switch value to the switchCounter and use the modulo operator (%)
            // to ensure the result stays within the 0-9999 range.
            switchCounter = (switchCounter + switchVal) % 10000;

            // Switch the display mode to show the 'switchCounter'.
            displayMode = 1;

            // Wait here until the button is released to prevent multiple additions.
            while (XGpio_DiscreteRead(&P_BTN_RIGHT, 1) == 1);
        }
    }
}

// Code below this loop is unreachable in a typical embedded system.
cleanup_platform();
return 0;
}
```

Figure 10(b)

Note: This program demonstrates a dual-mode interactive counter on a Xilinx MicroBlaze embedded system, utilizing two push-buttons and the slide switches for input, with output directed to a 4-digit 7-segment display. The code includes several standard headers: platform.h for system initialization, xil_types.h for specific integer types like u16, and xgpio.h for the GPIO driver. It also uses the custom headers gpio_init.h and seg7_display.h to manage the hardware-specific setup and display logic.

The program's core functionality is built around a **state machine** controlled by a displayMode flag, which dictates the system's behavior. In **Mode 0**, the application functions as a simple up-counter, incrementing a counter variable each time the left push-button is pressed. In **Mode 1**, it acts as an accumulator, reading a 4-bit value from the slide switches and adding it to a switchCounter variable each time the right push-button is pressed. Pressing either button not only performs the associated action but also immediately switches the system into that button's corresponding mode, ensuring the display always shows the most recently updated value.
The program's main function begins by initializing the platform, GPIOs, and the critical interrupt system that drives the 7-segment display in the background. Inside its infinite while(1) loop, it first updates the display based on the current displayMode. It then polls the left and right push-buttons. Both button handlers use a robust **debouce and wait-for-release** mechanism. This involves a short software delay to filter out mechanical noise, followed by a loop that pauses execution until the button is let go. This ensures that a single, continuous press is registered as only one event, making the counter's operation precise and predictable.

References:

You can go to the lab6 reference in the learn section, or directly copy the link to your browser:

1. Guide to 7-Segment Display Multiplexing https://learn.sparkfun.com/tutorials/7-segment-display-multiplexing-with-arduino-and-shift-registers/all
2. A Guide to Debouncing Buttons https://www.maximintegrated.com/en/design/technical-documents/tutorials/6/6953.html