Laboratory 4 – Arrays and Pointers

I. Pointer Basics

Objectives

- To build applications using Arrays and Pointers
- To learn how to use pointers and Arrays in embedded projects
- To implement basic array operations and pointer manipulation
- To understand memory addressing in embedded systems

Activity Summary

1. Launch Vitis 2022.2 and create workspace
2. Import hardware platform (XSA file) from Vivado 2022.2
3. Create Pointer_basics applications
4. Implement array operations with pointers
5. Program the FPGA and run applications on Basys3

Introduction

What is pointer: int p; declares an integer variable that directly stores a value. int *p; declares a pointer to an integer: the asterisk (*) means p stores the address of an integer, which can be obtained with the address-of operator (&). For example, if int x = 5;, then p = &x; makes p point to x. int p[3]; declares an array of three integers, while int *p[3]; declares an array of three pointers to integers, where each element can store the address of a different integer. Ref: https://www.geeksforgeeks.org/c/array-of-pointers-in-c/

What is array: An array is a collection of elements of the same data type stored in contiguous memory locations. It has a fixed size defined at declaration, and elements are accessed using an index that starts at 0. For example, int numbers[5] = {10, 20, 30, 40, 50}; creates an array of five integers, where numbers[0] is 10 and numbers[2] is 30. Arrays are useful for storing and processing multiple related values under a single name. Ref: https://www.geeksforgeeks.org/c/difference-between-array-and-pointers/

Specifications

The application demonstrates fundamental pointer operations in C programming on Xilinx embedded platform through UART console output.

Functionality:

Basic Pointer Operations:

- Declares a char variable b initialized to 0
- Creates a pointer address pointing to variable b
- Displays initial value, variable address, and pointer value
- Modifies variable through pointer dereferencing (*address = 5)
- Verifies modification through both direct and pointer access

Multiple Pointer Management:

- Creates two integer variables: x = 100 and y = 200
- Assigns separate pointers ptr1 and ptr2 to each variable
- Displays values accessed through different pointers simultaneously

Pointer Arithmetic:

- Defines integer array: arr[5] = {10, 20, 30, 40, 50}
- Uses pointer to traverse array elements

- Compares array indexing arr[i] with pointer notation *(ptr+i)
- Iterates through all 5 elements displaying both access methods

Console Output:

Basic Pointer Operations:

- Initial value of b: 0
- Address of b: 0x[hexadecimal address]
- Value of address pointer: 0x[hexadecimal address]
- After modification: New value of b: 5
- Value accessed through pointer: 5

Multiple Pointers:

- x = 100, y = 200
- *ptr1 = 100, *ptr2 = 200

Pointer Arithmetic:

- For each array element (i = 0 to 4):
- arr[i] = [value], *(ptr+i) = [value]

Activities

1. **Open and Export Hardware Design**

   Follow the procedure described in Lab 1 instructions to unzip and open **Lab 4 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.

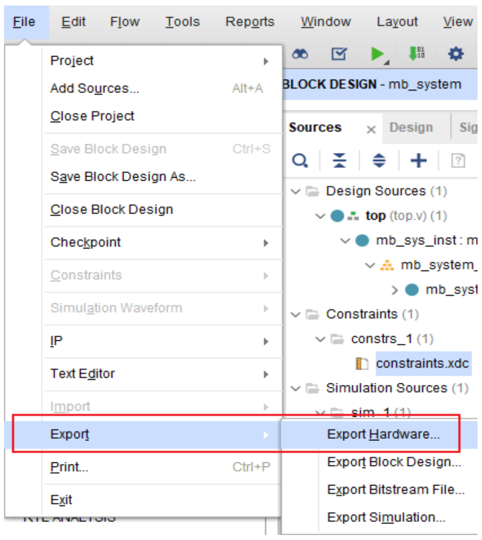
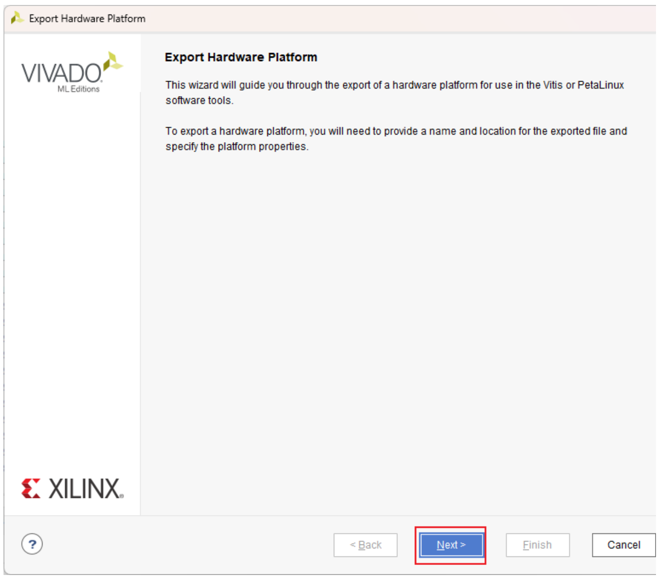


Figure 1(a)

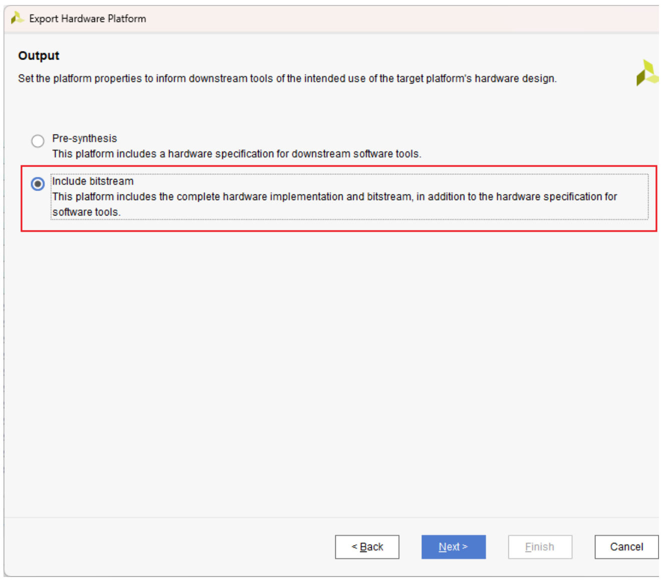

Export Hardware Platform

Export Hardware Platform

This wizard will guide you through the export of a hardware platform for use in the Vitis or PetaLinux software tools.

To export a hardware platform, you will need to provide a name and location for the exported file and specify the platform properties.

Figure 1(b)


Export Hardware Platform

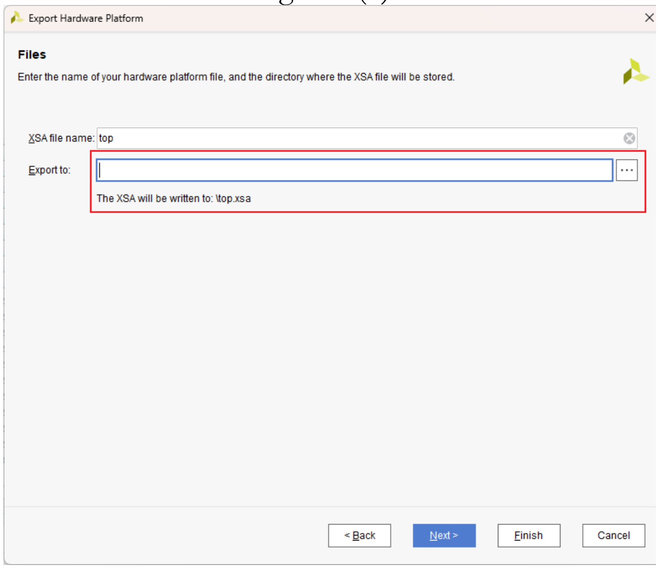
Output

Set the platform properties to inform downstream tools of the intended use of the target platform's hardware design.

( ) Pre-synthesis
This platform includes a hardware specification for downstream software tools.

[✓ Include bitstream]
This platform includes the complete hardware implementation and bitstream, in addition to the hardware specification for software tools.

Figure 1(c)


Export Hardware Platform

Files

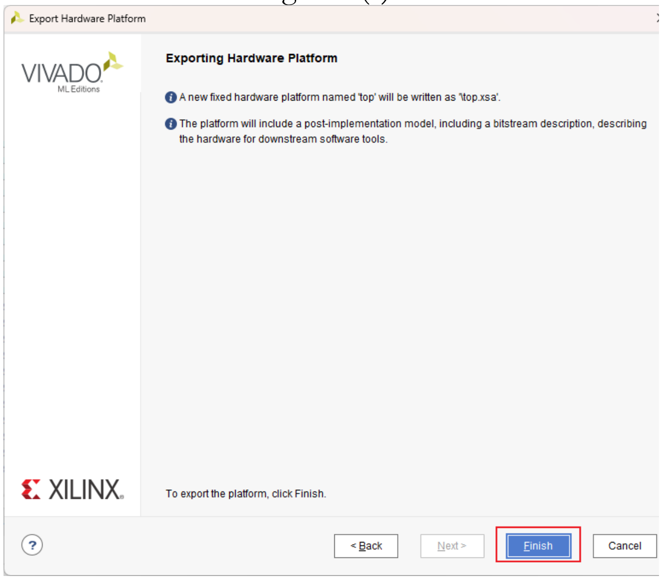
Enter the name of your hardware platform file, and the directory where the XSA file will be stored.

XSA file name: top

Export to:

The XSA will be written to: \top.xsa

Figure 1(d)


Export Hardware Platform

Exporting Hardware Platform

A new fixed hardware platform named 'top' will be written as '\top.xsa'.

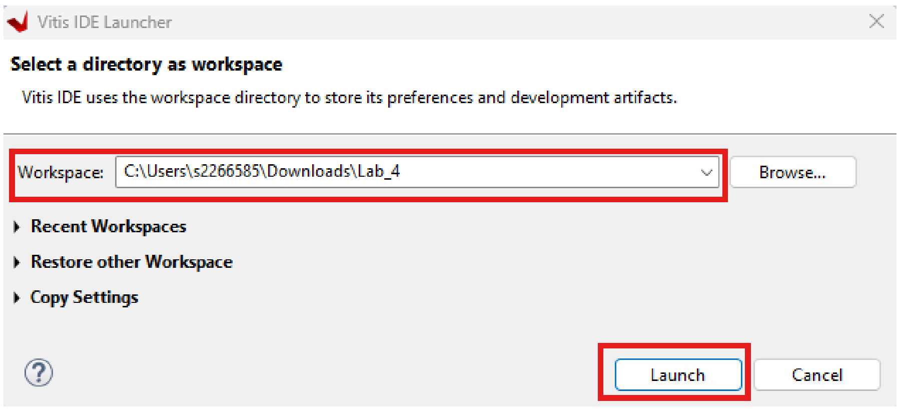
The platform will include a post-implementation model, including a bitstream description, describing the hardware for downstream software tools.

To export the platform, click Finish.

Figure 1(e)

2. **Launch Vitis**

Create the new folder in the PC, named Lab4, then open Vitis and select the file path of Lab4 as the workspace, and select **Launch**. **(Note: If you use Lab PC, please do not create the workspace on the uni-cloud, such as Desktop)**


Vitis IDE Launcher

Select a directory as workspace

Vitis IDE uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\Users\s2266585\Downloads\Lab_4

Figure 2

3. Create a New Application Project

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 3a), 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 3b), 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 3c).

Step 3: Configure Platform On the platform selection page (Figure 3d), 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 3e), enter the application project name (such as " **Pointer_basics** "). The system will automatically create an associated system project " **Pointer_basics**_system" with target processor shown as "microblaze_0".

Step 5: Select Domain On the domain selection page (Figure 3f), 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 3g), 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 Lab4 embedded application.

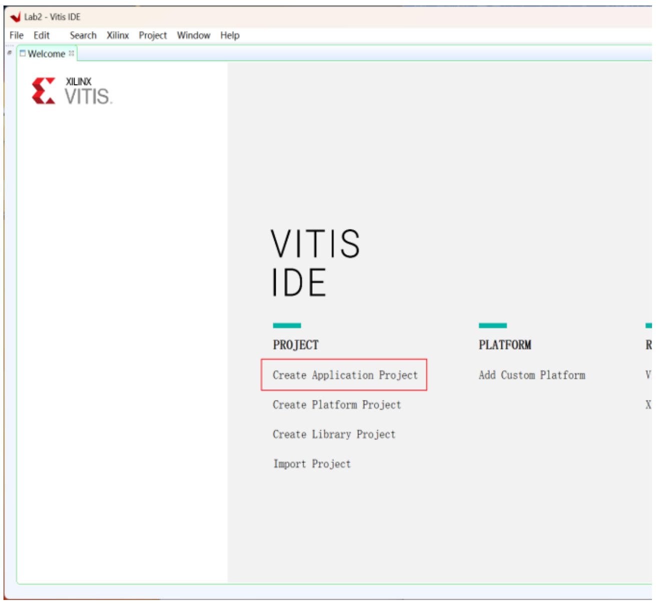
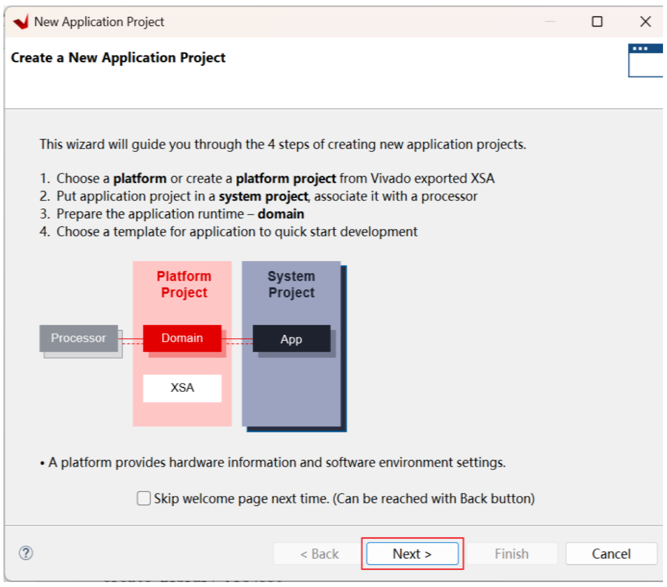


Figure 3(a)



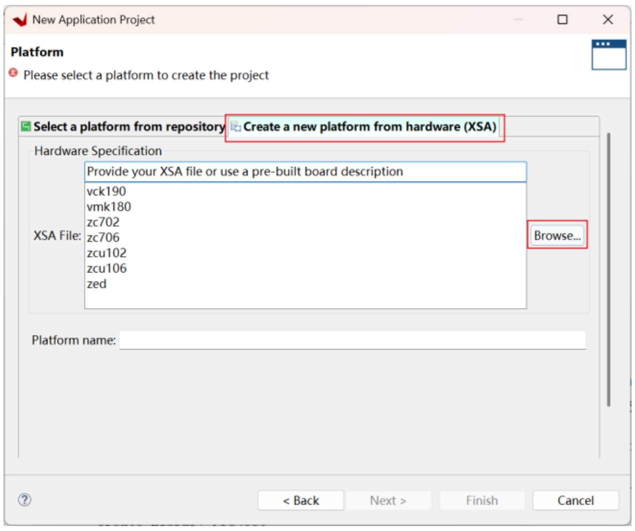
This wizard will guide you through the 4 steps of creating new application projects.

1. Choose a **platform** or create a **platform project** from Vivado exported XSA
2. Put application project in a **system project**, associate it with a processor
3. Prepare the application runtime – **domain**
4. Choose a template for application to quick start development

• A platform provides hardware information and software environment settings.

☐ Skip welcome page next time. (Can be reached with Back button)

Figure 3(b)


New Application Project
Select a platform from repository
Create a new platform from hardware (XSA)
Browse...
< Back
Next >
Finish
Cancel

Platform

Please select a platform to create the project

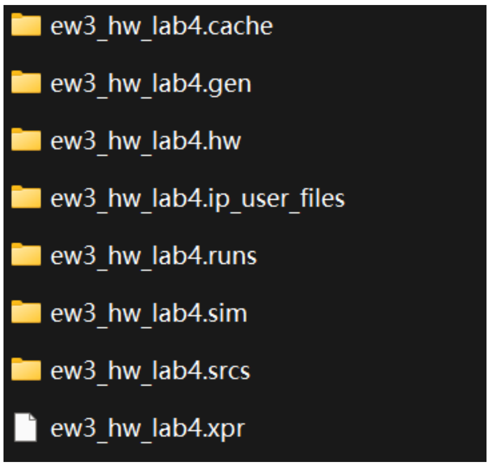
Hardware Specification

XSA File: Provide your XSA file or use a pre-built board description

vck190
vmk180
zc702
zc706
zcu102
zcu106
zed

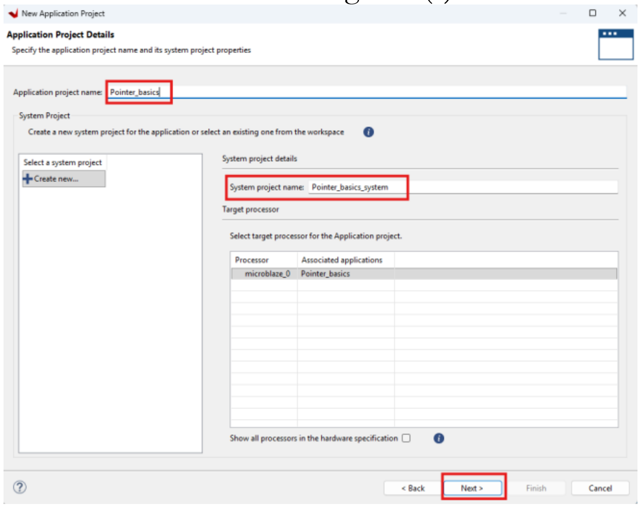
Platform name:

Figure 3(c)



ew3_hw_lab4.cache
ew3_hw_lab4.gen
ew3_hw_lab4.hw
ew3_hw_lab4.ip_user_files
ew3_hw_lab4.runs
ew3_hw_lab4.sim
ew3_hw_lab4.srcs
ew3_hw_lab4.xpr

Figure 3(d)



Application Project Details

Specify the application project name and its system project properties

Application project name: Pointer_basics

System Project

Create a new system project for the application or select an existing one from the workspace

System project details

System project name: Pointer_basics_system

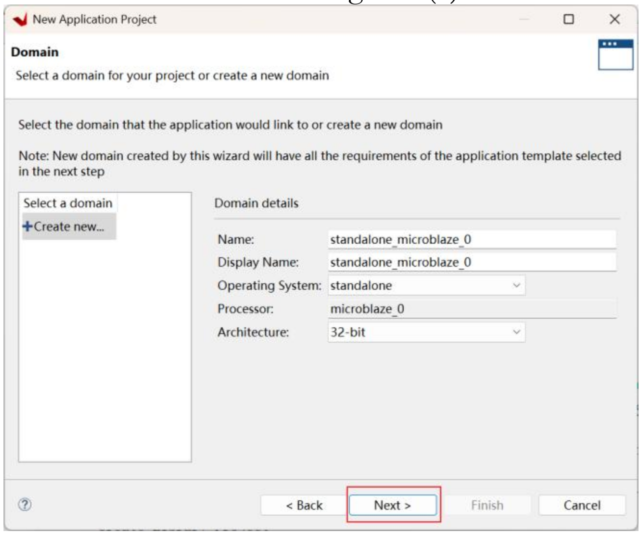
Target processor

Select target processor for the Application project.

Processor	Associated applications
microblaze_0	Pointer_basics

Show all processors in the hardware specification ☐

Figure 3(e)



Domain

Select a domain for your project or create a new domain

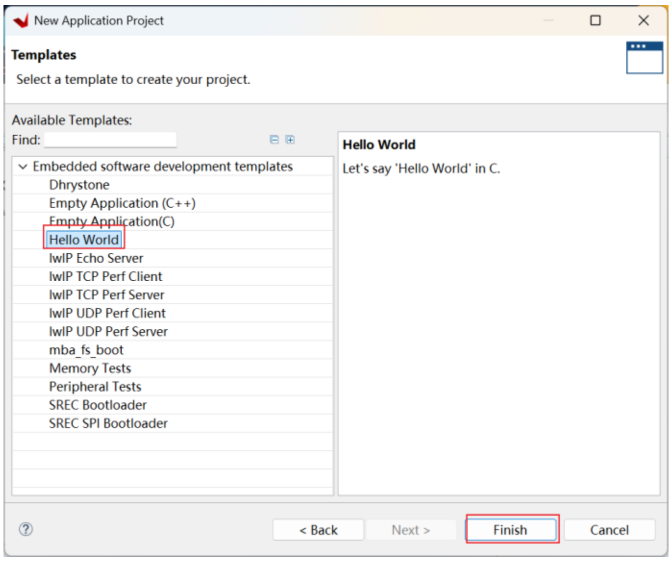
Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

Domain details

Name:	standalone_microblaze_0
Display Name:	standalone_microblaze_0
Operating System:	standalone
Processor:	microblaze_0
Architecture:	32-bit

Figure 3(f)



Templates

Select a template to create your project.

Available Templates:

Find:

Embedded software development templates
Dhrystone
Empty Application (C++)
Empty Application(C)
Hello World
lwIP Echo Server
lwIP TCP Perf Client
lwIP TCP Perf Server
lwIP UDP Perf Client
lwIP UDP Perf Server
mba_fs_boot
Memory Tests
Peripheral Tests
SREC Bootloader
SREC SPI Bootloader

Hello World

Let's say 'Hello World' in C.

Figure 3(g)

4. Create and Rename the Source File

- As shown in the Figure 4, in **Project Explorer**, expand the **Pointer_basics** → **src** folder.
- Right-click on helloworld.c, select **Rename…**, and rename the file to Pointer_basics.c.

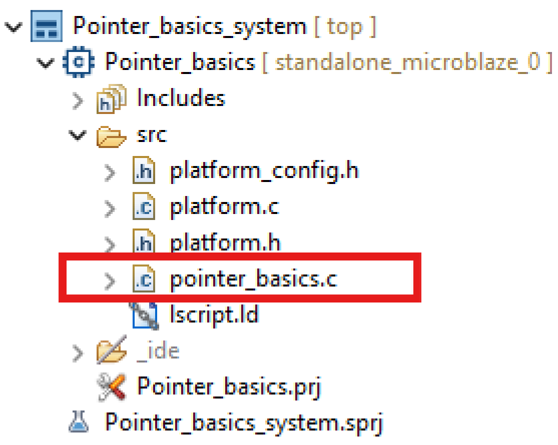

Pointer_basics_system [top]
Pointer_basics [standalone_microblaze_0]
Includes
src
platform_config.h
platform.c
platform.h
pointer_basics.c
lscript.ld
_ide
Pointer_basics.prj
Pointer_basics_system.sprj

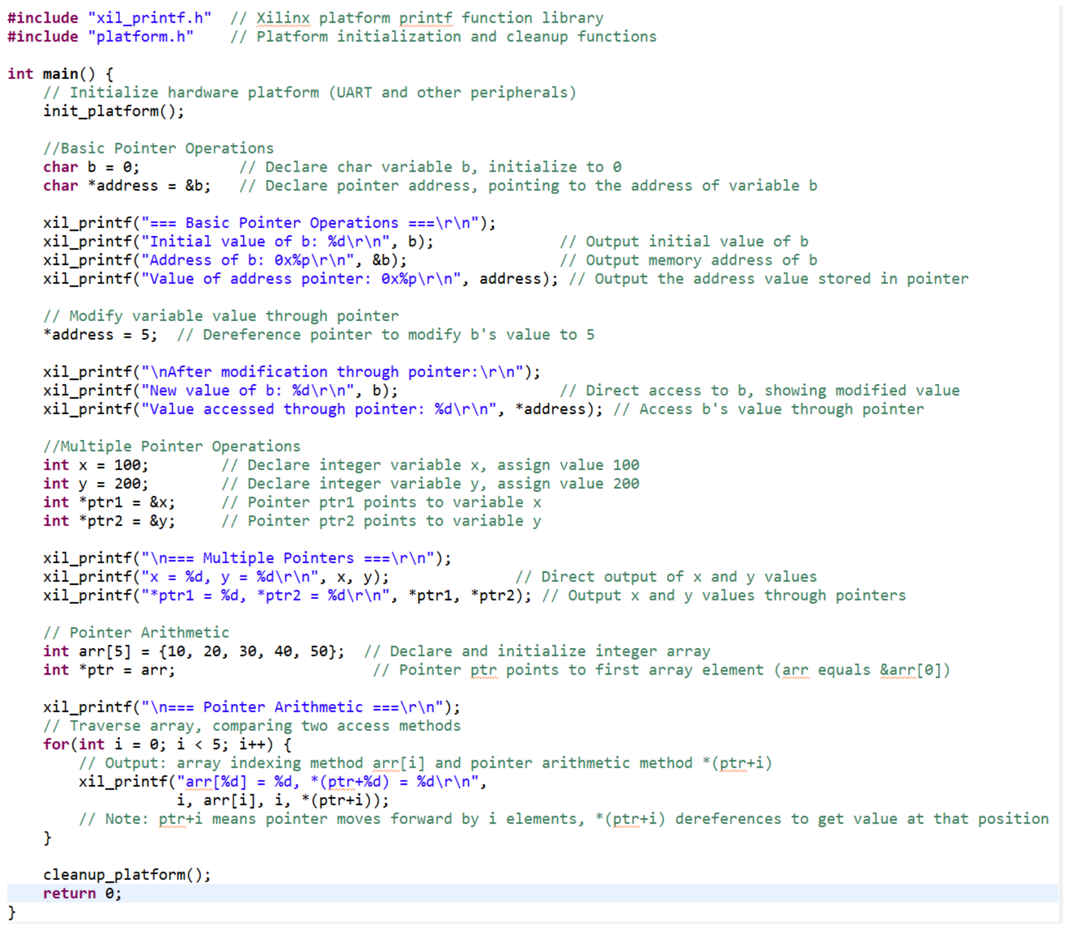
Figure 4.

5. Include Required Header Files

- Open Pointer_basics.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.



```
#include "xil_printf.h"  // Xilinx platform printf function library
#include "platform.h"    // Platform initialization and cleanup functions

int main() {
    // Initialize hardware platform (UART and other peripherals)
    init_platform();

    //Basic Pointer Operations
    char b = 0;           // Declare char variable b, initialize to 0
    char *address = &b;   // Declare pointer address, pointing to the address of variable b

    xil_printf("=== Basic Pointer Operations ===\r\n");
    xil_printf("Initial value of b: %d\r\n", b);              // Output initial value of b
    xil_printf("Address of b: 0x%p\r\n", &b);                 // Output memory address of b
    xil_printf("Value of address pointer: 0x%p\r\n", address); // Output the address value stored in pointer

    // Modify variable value through pointer
    *address = 5;  // Dereference pointer to modify b's value to 5

    xil_printf("\nAfter modification through pointer:\r\n");
    xil_printf("New value of b: %d\r\n", b);                  // Direct access to b, showing modified value
    xil_printf("Value accessed through pointer: %d\r\n", *address); // Access b's value through pointer

    //Multiple Pointer Operations
    int x = 100;        // Declare integer variable x, assign value 100
    int y = 200;        // Declare integer variable y, assign value 200
    int *ptr1 = &x;     // Pointer ptr1 points to variable x
    int *ptr2 = &y;     // Pointer ptr2 points to variable y

    xil_printf("\n=== Multiple Pointers ===\r\n");
    xil_printf("x = %d, y = %d\r\n", x, y);              // Direct output of x and y values
    xil_printf("*ptr1 = %d, *ptr2 = %d\r\n", *ptr1, *ptr2); // Output x and y values through pointers

    // Pointer Arithmetic
    int arr[5] = {10, 20, 30, 40, 50};  // Declare and initialize integer array
    int *ptr = arr;                      // Pointer ptr points to first array element (arr equals &arr[0])

    xil_printf("\n=== Pointer Arithmetic ===\r\n");
    // Traverse array, comparing two access methods
    for(int i = 0; i < 5; i++) {
        // Output: array indexing method arr[i] and pointer arithmetic method *(ptr+i)
        xil_printf("arr[%d] = %d, *(ptr+%d) = %d\r\n",
                   i, arr[i], i, *(ptr+i));
        // Note: ptr+i means pointer moves forward by i elements, *(ptr+i) dereferences to get value at that position
    }

    cleanup_platform();
    return 0;
}
```

Figure 5

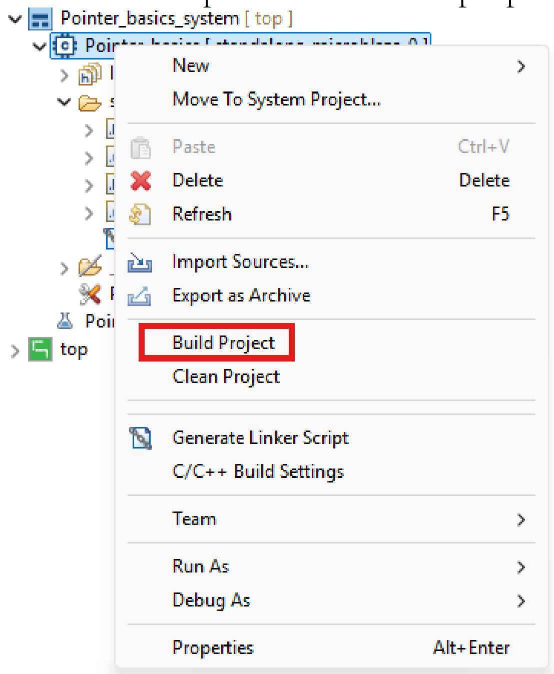
Note: This code demonstrates fundamental pointer concepts in embedded C programming using the

Xilinx platform, covering three essential topics: basic pointer operations, multiple pointer management, and pointer arithmetic. The program begins by initializing a char variable b to 0 and creating a pointer address to store the memory address of b. The code demonstrates that &b (the address-of operator) and address contain the same memory location, and by dereferencing the pointer with *address = 5, we can modify b indirectly. This illustrates the key concept that pointers allow indirect access to variables, meaning changes made through the pointer affect the original variable.

The second section demonstrates multiple pointer management by creating two separate integer variables, x and y, with different values (100 and 200 respectively). Each variable has its own pointer (ptr1 and ptr2) pointing to it, demonstrating that each pointer independently tracks its own memory location. This shows that multiple pointers can coexist in a program, each managing different memory locations without interfering with one another.

The final section illustrates pointer arithmetic using an integer array arr containing five elements (10, 20, 30, 40, 50). A pointer ptr is set to point to the first element of the array, taking advantage of the fact that in C, array names decay to pointers. The notation *(ptr+i) demonstrates pointer arithmetic, where adding i to the pointer moves it i elements forward in memory. The code shows that arr[i] and *(ptr+i) are equivalent ways to access array elements, with the pointer arithmetic automatically accounting for the data type size (4 bytes for int). The program prints addresses in hexadecimal and values in decimal throughout, allowing students to observe how pointer operations affect the underlying data.

6. Once you have completed writing the pointer_basics.c code, Vitis IDE will display the complete development environment. The project structure shows both the system project and application project. First, you need to build the application by right-clicking on the Pointer_basics project and selecting Build Project from the context menu (Figure 6a). This compiles the C source code into an executable file (.elf) that can run on the MicroBlaze processor. Next, to program the FPGA with the hardware design, right-click on the Pointer_basics project and select Program Device from the context menu (Figure 6b). This step configures the FPGA with the hardware bitstream containing the MicroBlaze processor and other peripherals.



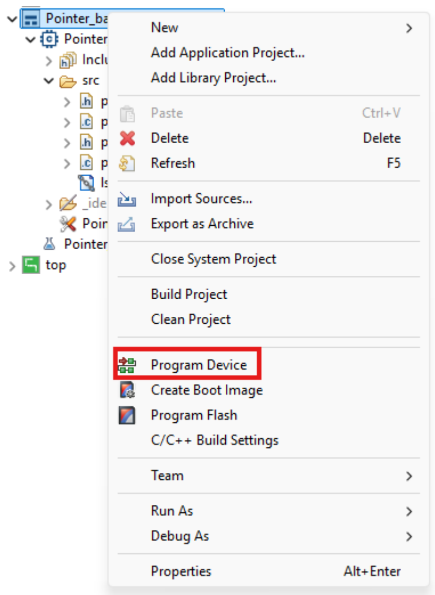


Figure 6(a) Figure 6(b)

7. To view the output and run the application, follow **Section 4 Steps 5-8** of Lab 1 instructions (also illustrated in **Figure 6(a)-(d)**): open the Terminal view, configure the Serial Terminal settings, and set up the run/debug configuration. **Figure 7** shows what the Console should look like when the application is successfully run.

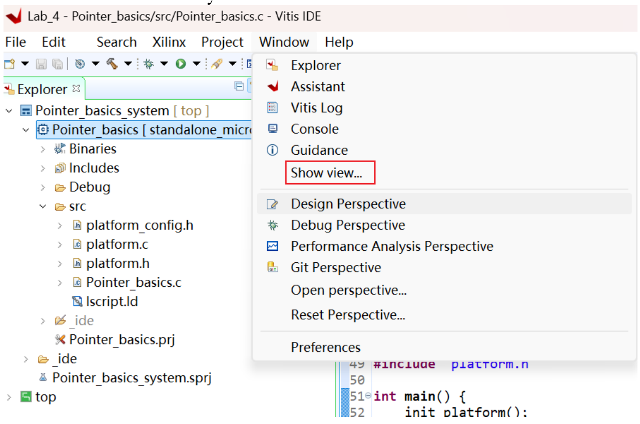


Figure 6(a)

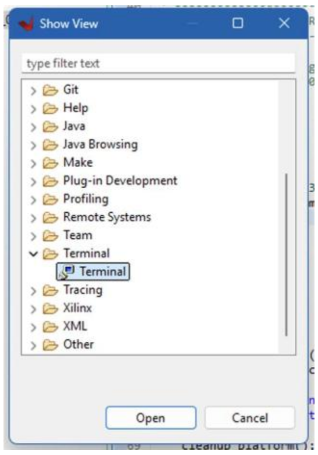
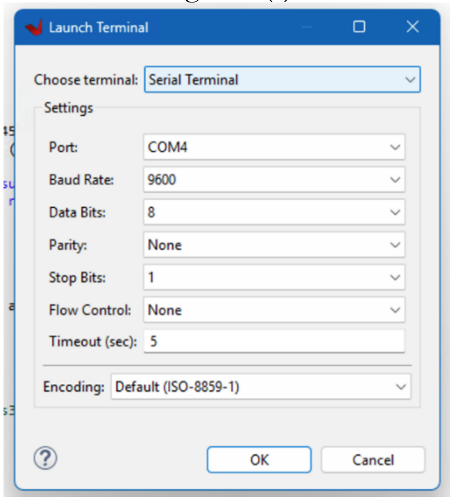


Figure 6(b)



Launch Terminal

Choose terminal: Serial Terminal

Settings

Port: COM4

Baud Rate: 9600

Data Bits: 8

Parity: None

Stop Bits: 1

Flow Control: None

Timeout (sec): 5

Encoding: Default (ISO-8859-1)

Figure 6(c)

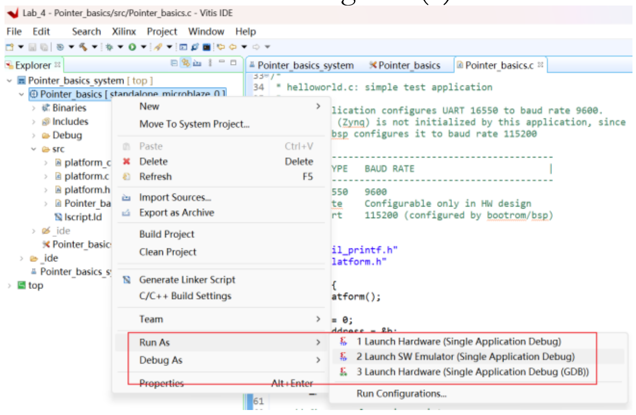
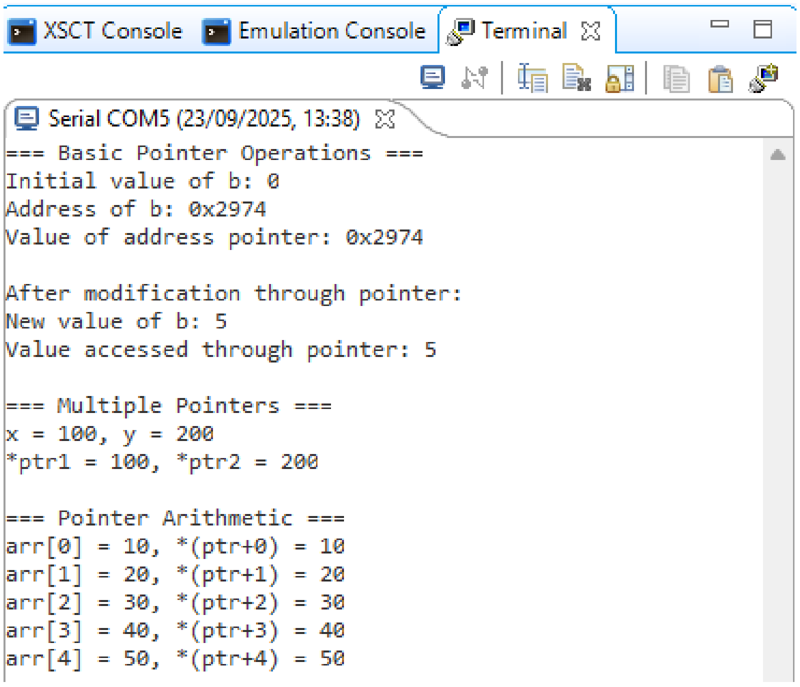

Lab_4 - Pointer_basics/src/Pointer_basics.c - Vitis IDE
File Edit Search Xilinx Project Window Help
Explorer
Pointer_basics_system [top]
Pointer_basics [standalone_microblaze_0]
New
Move To System Project...
Paste
Delete
Refresh
Import Sources...
Export as Archive
Build Project
Clean Project
Generate Linker Script
C/C++ Build Settings
Team
Run As
Debug As
Properties
1 Launch Hardware (Single Application Debug)
2 Launch SW Emulator (Single Application Debug)
3 Launch Hardware (Single Application Debug (GDB))
Run Configurations...

Figure 6(d)



```
=== Basic Pointer Operations ===
Initial value of b: 0
Address of b: 0x2974
Value of address pointer: 0x2974

After modification through pointer:
New value of b: 5
Value accessed through pointer: 5

=== Multiple Pointers ===
x = 100, y = 200
*ptr1 = 100, *ptr2 = 200

=== Pointer Arithmetic ===
arr[0] = 10, *(ptr+0) = 10
arr[1] = 20, *(ptr+1) = 20
arr[2] = 30, *(ptr+2) = 30
arr[3] = 40, *(ptr+3) = 40
arr[4] = 50, *(ptr+4) = 50
```

Figure 7

II. Swap Values

Objectives

- Implement value swapping using pointers
- Display results on LEDs
- Compare different swap methods

Activity Summary

1. Create a new application project called swap_values
2. Program the FPGA, setup Run Configuration, and run the application

Specifications

The application demonstrates three methods to use swapping two integers with LED and serial console output.

Functionality:

Method 1: Swap with Temporary Variable

- Swaps x=15 and y=27
- Uses temporary variable to store intermediate value
- Displays values on LEDs before and after swap (2 seconds each)

Method 2: XOR Swap

- Swaps a=100 and b=200
- Uses XOR bitwise operations, no temporary variable needed
- Includes safety check to prevent same-address issues

Method 3: Arithmetic Swap

- Swaps p=50 and q=75
- Uses addition/subtraction operations, no temporary variable needed
- Includes safety check

Continuous Display

- Infinite loop alternating between x and y values
- Each value displayed for 1 second

Console Output:

LED Display:

- x=15 → Binary 0000 1111 (LEDs 0-3 ON)
- y=27 → Binary 0001 1011 (LEDs 0,1,3,4 ON)
- Swapped values displayed through LED binary patterns

Example:

- Method 1: x=15, y=27 → x=27, y=15
- Method 2: a=100, b=200 → a=200, b=100
- Method 3: p=50, q=75 → p=75, q=50

Activities

The activities here involve swapping values using three different methods, i.e. using pointers, using XOR and using arithmetic operations. The output of swapping values using pointers is displayed using LEDs on the Basys3 board. Follow the steps below:

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

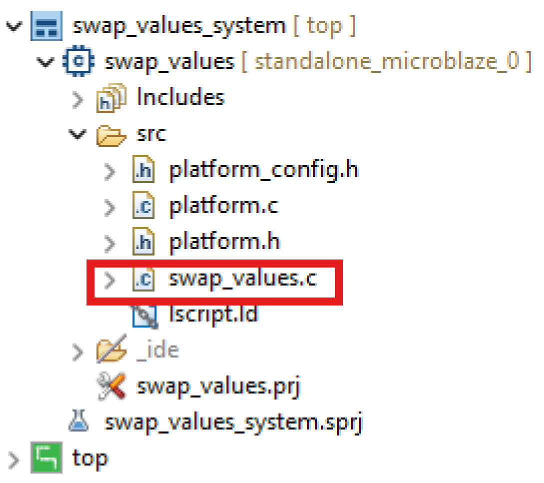

swap_values_system [top]
swap_values [standalone_microblaze_0]
Includes
src
platform_config.h
platform.c
platform.h
swap_values.c
lscript.ld
_ide
swap_values.prj
swap_values_system.sprj
top

Figure 8

2. Write the code as shown in Figure 9(a)-(d).

Figure 9(a): Header files and function declarations

Figure 9(b-c): Main function

Figure 9(d): Function implementations

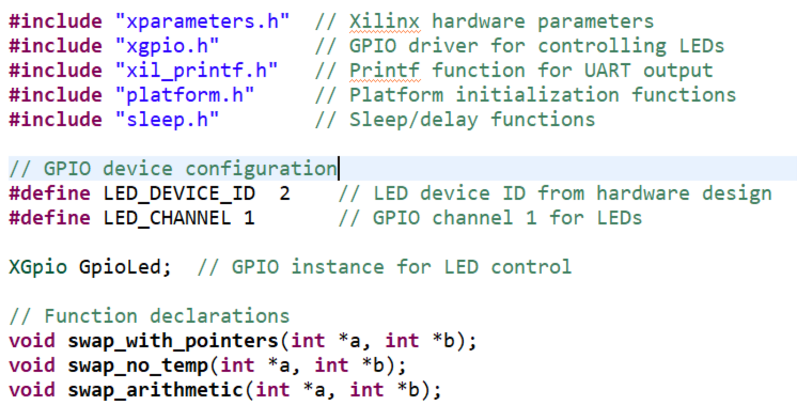
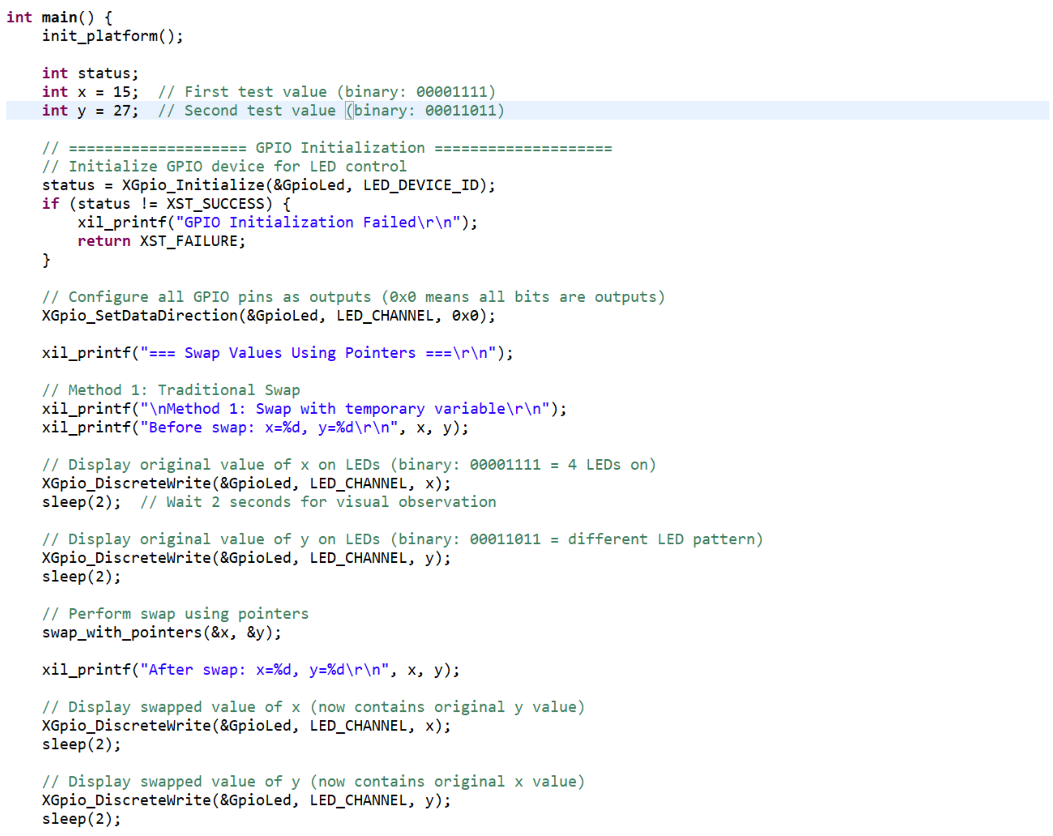
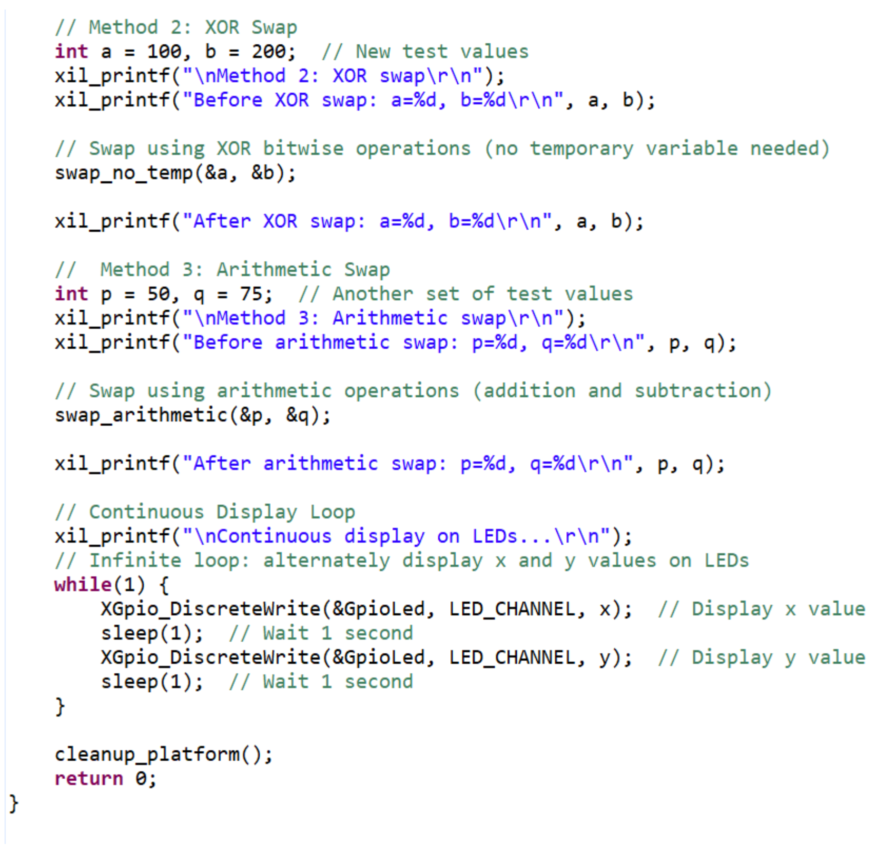

```
#include "xparameters.h"  // Xilinx hardware parameters
#include "xgpio.h"        // GPIO driver for controlling LEDs
#include "xil_printf.h"   // Printf function for UART output
#include "platform.h"     // Platform initialization functions
#include "sleep.h"        // Sleep/delay functions

// GPIO device configuration
#define LED_DEVICE_ID  2    // LED device ID from hardware design
#define LED_CHANNEL 1       // GPIO channel 1 for LEDs

XGpio GpioLed;  // GPIO instance for LED control

// Function declarations
void swap_with_pointers(int *a, int *b);
void swap_no_temp(int *a, int *b);
void swap_arithmetic(int *a, int *b);
```

Figure 9(a)



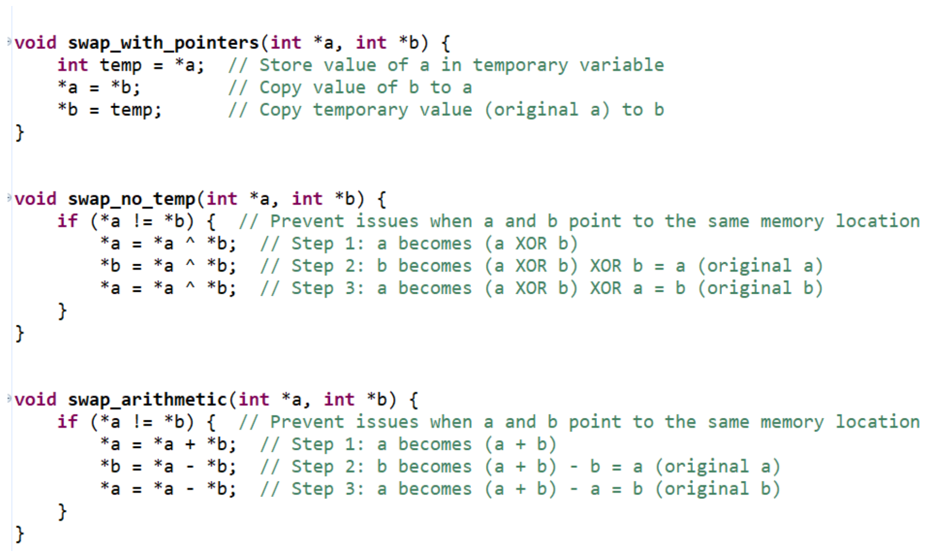
```c
int main() {
    init_platform();

    int status;
    int x = 15;  // First test value (binary: 00001111)
    int y = 27;  // Second test value (binary: 00011011)

    // ==================== GPIO Initialization ====================
    // Initialize GPIO device for LED control
    status = XGpio_Initialize(&GpioLed, LED_DEVICE_ID);
    if (status != XST_SUCCESS) {
        xil_printf("GPIO Initialization Failed\r\n");
        return XST_FAILURE;
    }

    // Configure all GPIO pins as outputs (0x0 means all bits are outputs)
    XGpio_SetDataDirection(&GpioLed, LED_CHANNEL, 0x0);

    xil_printf("=== Swap Values Using Pointers ===\r\n");

    // Method 1: Traditional Swap
    xil_printf("\nMethod 1: Swap with temporary variable\r\n");
    xil_printf("Before swap: x=%d, y=%d\r\n", x, y);

    // Display original value of x on LEDs (binary: 00001111 = 4 LEDs on)
    XGpio_DiscreteWrite(&GpioLed, LED_CHANNEL, x);
    sleep(2);  // Wait 2 seconds for visual observation

    // Display original value of y on LEDs (binary: 00011011 = different LED pattern)
    XGpio_DiscreteWrite(&GpioLed, LED_CHANNEL, y);
    sleep(2);

    // Perform swap using pointers
    swap_with_pointers(&x, &y);

    xil_printf("After swap: x=%d, y=%d\r\n", x, y);

    // Display swapped value of x (now contains original y value)
    XGpio_DiscreteWrite(&GpioLed, LED_CHANNEL, x);
    sleep(2);

    // Display swapped value of y (now contains original x value)
    XGpio_DiscreteWrite(&GpioLed, LED_CHANNEL, y);
    sleep(2);
```

Figure 9(b)



```c
    // Method 2: XOR Swap
    int a = 100, b = 200;  // New test values
    xil_printf("\nMethod 2: XOR swap\r\n");
    xil_printf("Before XOR swap: a=%d, b=%d\r\n", a, b);

    // Swap using XOR bitwise operations (no temporary variable needed)
    swap_no_temp(&a, &b);

    xil_printf("After XOR swap: a=%d, b=%d\r\n", a, b);

    //  Method 3: Arithmetic Swap
    int p = 50, q = 75;  // Another set of test values
    xil_printf("\nMethod 3: Arithmetic swap\r\n");
    xil_printf("Before arithmetic swap: p=%d, q=%d\r\n", p, q);

    // Swap using arithmetic operations (addition and subtraction)
    swap_arithmetic(&p, &q);

    xil_printf("After arithmetic swap: p=%d, q=%d\r\n", p, q);

    // Continuous Display Loop
    xil_printf("\nContinuous display on LEDs...\r\n");
    // Infinite loop: alternately display x and y values on LEDs
    while(1) {
        XGpio_DiscreteWrite(&GpioLed, LED_CHANNEL, x);  // Display x value
        sleep(1);  // Wait 1 second
        XGpio_DiscreteWrite(&GpioLed, LED_CHANNEL, y);  // Display y value
        sleep(1);  // Wait 1 second
    }

    cleanup_platform();
    return 0;
}
```

Figure 9(c)



```c
void swap_with_pointers(int *a, int *b) {
    int temp = *a;  // Store value of a in temporary variable
    *a = *b;        // Copy value of b to a
    *b = temp;      // Copy temporary value (original a) to b
}


void swap_no_temp(int *a, int *b) {
    if (*a != *b) {  // Prevent issues when a and b point to the same memory location
        *a = *a ^ *b;  // Step 1: a becomes (a XOR b)
        *b = *a ^ *b;  // Step 2: b becomes (a XOR b) XOR b = a (original a)
        *a = *a ^ *b;  // Step 3: a becomes (a XOR b) XOR a = b (original b)
    }
}


void swap_arithmetic(int *a, int *b) {
    if (*a != *b) {  // Prevent issues when a and b point to the same memory location
        *a = *a + *b;  // Step 1: a becomes (a + b)
        *b = *a - *b;  // Step 2: b becomes (a + b) - b = a (original a)
        *a = *a - *b;  // Step 3: a becomes (a + b) - a = b (original b)
    }
}
```

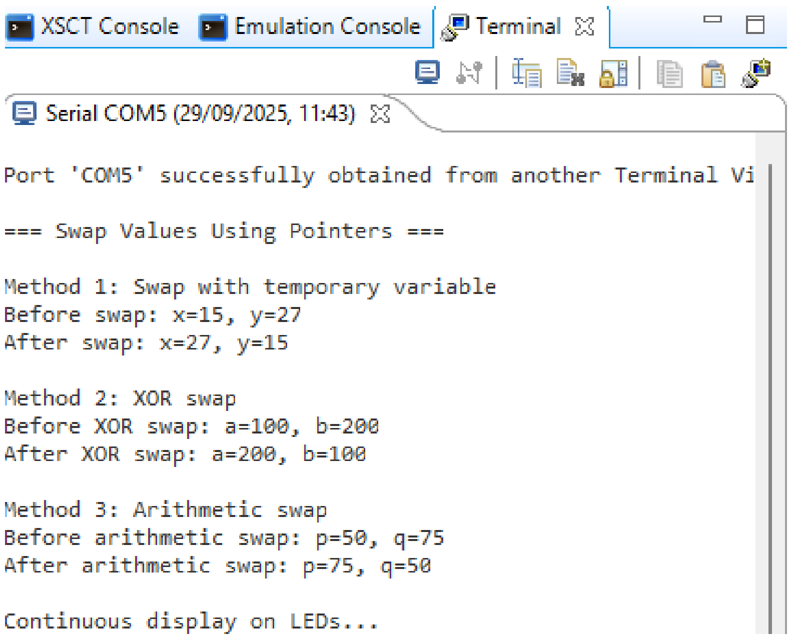
Figure 9(d)

Note: This program demonstrates three different methods of swapping two integer values using pointers on a Xilinx MicroBlaze embedded system, with visual feedback displayed on LEDs. The code includes several essential header files: xparameters.h for hardware parameters and configuration, xgpio.h for GPIO driver functions, xil_printf.h for serial output, platform.h for platform initialization, and sleep.h for delay functions. The program is configured to use LED_DEVICE_ID as 2 (which is hardware specific) and LED_CHANNEL as 1, with a GPIO instance called GpioLed for controlling the LEDs.

The program implements three distinct swapping methods. The first method, swap_with_pointers, uses the traditional approach with a temporary variable. It works by storing the value pointed to by 'a' in a temporary variable, then copying the value from 'b' to 'a', and finally copying the temporary value to 'b'. This method is simple, reliable, and readable, though it requires extra memory for the temporary variable. The second method uses XOR bitwise operations to swap values without a temporary variable. It performs three XOR operations: first 'a' becomes 'a XOR b', then 'b' becomes the result XORed with 'b' (yielding original a), and finally 'a' becomes the result XORed with the new 'b' (yielding original b). This method includes a safety check (if *a != *b) to prevent issues when both pointers point to the same memory location. While it doesn't need extra memory, it's less readable and can have aliasing issues. The third method, uses arithmetic operations by first setting 'a' to the sum of both values, then subtracting the new 'a' from 'b' to get the original 'a' value, and finally subtracting the new 'b' from 'a' to get the original 'b' value. Like the XOR method, it also includes a safety check to avoid issues when pointers are aliased.

The main function begins by initializing the platform and declaring two integers, x=15 and y=27, which will be swapped. It then initializes the GPIO for the LEDs using XGpio_Initialize and checks if the initialization was successful; if it fails, it prints an error message and returns XST_FAILURE. The GPIO data direction is set to output (0x0) for LED control. The program then demonstrates Method 1 by printing the values before the swap, displaying them on the LEDs with 2-second delays between displays (x shows as binary 00001111 and y as 00011011), performing the swap using pointers, printing the results, and displaying the swapped values on the LEDs. Next, it demonstrates Method 2 by creating new variables a=100 and b=200, performing the XOR swap, and printing the results. Finally, it demonstrates Method 3 with variables p=50 and q=75, performing the arithmetic swap and printing the results. After all demonstrations, the program enters an infinite loop that continuously alternates between displaying x and y on the LEDs every 1 second, providing ongoing visual feedback of the swapped values. The cleanup_platform function at the end (which is never reached due to the infinite loop) would normally clean up system resources.

3. Upon successful implementation of this code the output shown in Figure 10 would be observed on the console.



```
Port 'COM5' successfully obtained from another Terminal Vi

=== Swap Values Using Pointers ===

Method 1: Swap with temporary variable
Before swap: x=15, y=27
After swap: x=27, y=15

Method 2: XOR swap
Before XOR swap: a=100, b=200
After XOR swap: a=200, b=100

Method 3: Arithmetic swap
Before arithmetic swap: p=50, q=75
After arithmetic swap: p=75, q=50

Continuous display on LEDs...
```

Figure 10

4. The LED output shown in Figures 11(a) and 11(b) would be seen at every 1 second interval on the BASYS 3 board.

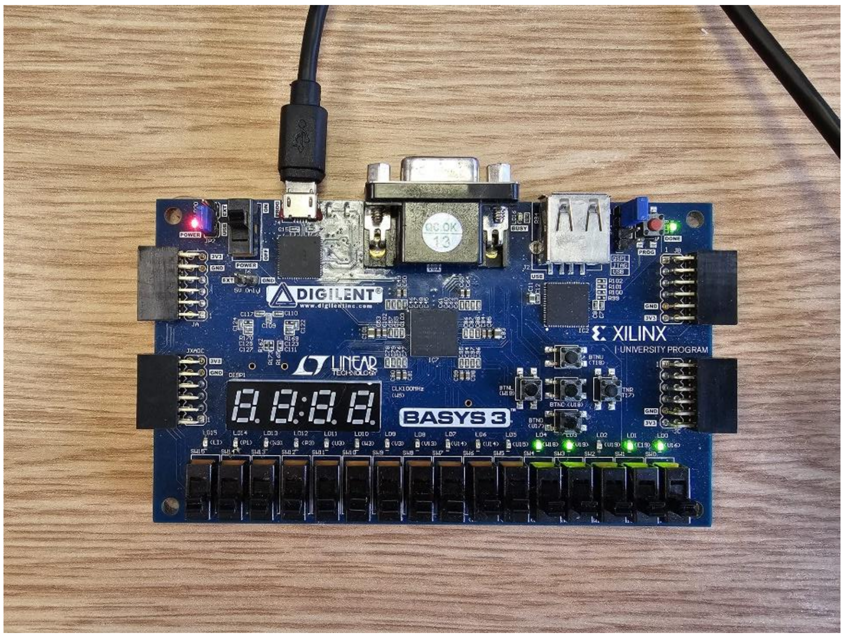


Figure 11(a)

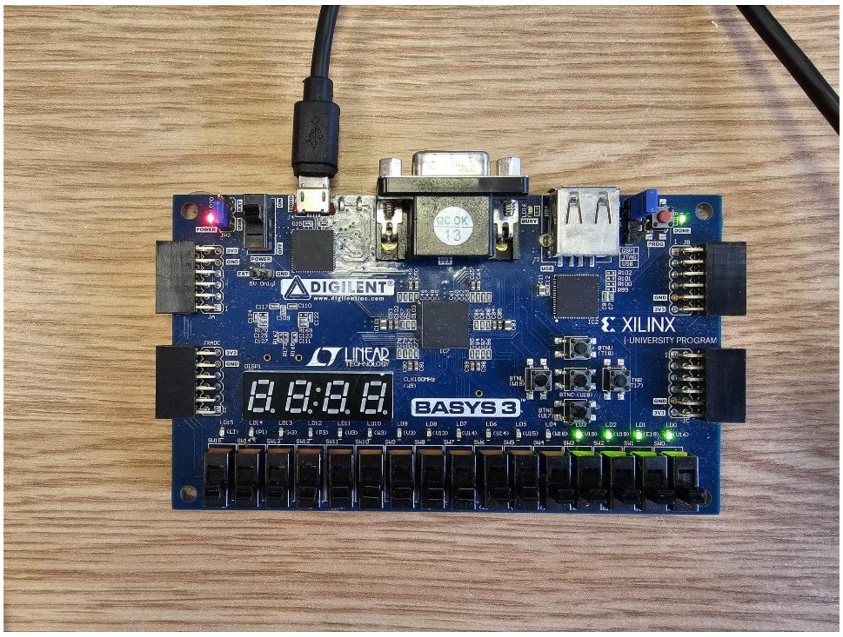

DIGILENT
LINEAR TECHNOLOGY
XILINX
UNIVERSITY PROGRAM
BASYS 3

Figure 11(b)

III. Array Operations with Pointers

Objectives

- Manipulate arrays using pointers
- Implement basic sorting algorithm
- Use buttons and switches for input

Activity Summary

1. Create a new application named **array_operations**
2. Program the FPGA, setup Run Configuration, and run the application

Specifications

The application demonstrates array operations using pointers on Xilinx embedded platform, performing maximum value detection and descending sort with visual feedback on LEDs.

Functionality:

Predefined Array:

- Fixed array with 6 elements: {45, 23, 67, 12, 89, 34}
- All operations performed using pointer arithmetic
- Array stored in memory at program initialization

Function 1: Find Maximum Value

- Traverses array using pointer arithmetic *(arr + i)
- Compares all elements to find the largest value
- Returns maximum value: 89
- Displays result on LEDs in binary format for 3 seconds

Function 2: Sort Array (Descending Order)

- Implements bubble sort algorithm using pointers
- Sorts from largest to smallest: 89 → 67 → 45 → 34 → 23 → 12
- Uses pointer notation *(arr+j) for element access and swapping
- Each sorted value displayed on LEDs for 2 seconds sequentially

Continuous Display:

- Infinite loop cycling through all sorted values
- Each value displayed for 2 seconds
- Repeats indefinitely after operations complete

Hardware Requirements:

- **LEDs**: 16 LEDs connected to GPIO Device ID 2, Channel 1
- **Display Format**: Binary representation (each LED = one bit)

Console Output:

```
=== Array Operations with Pointers ===
Note: Values are displayed on LEDs in binary format

Original array:
Array contents: 45 23 67 12 89 34

--- Function 1: Find Maximum Value ---
Maximum value in array: 89
Binary: 00000000 01011001 (LEDs show lower bits)
Displaying maximum value (89) on LEDs for 3 seconds...

--- Function 2: Sort Array (Descending Order) ---
Sorting array from largest to smallest...
```

```
Sorted array (descending):
Array contents: 89 67 45 34 23 12

Displaying each sorted value on LEDs (2 seconds each):
[1] Value: 89
Binary: 00000000 01011001
[2] Value: 67
Binary: 00000000 01000011
[3] Value: 45
Binary: 00000000 00101101
[4] Value: 34
Binary: 00000000 00100010
[5] Value: 23
Binary: 00000000 00010111
[6] Value: 12
Binary: 00000000 00001100

=== Operations Complete ===
Continuously cycling through sorted values on LEDs...
```

LED Display Pattern:
Maximum Value (89):

- Binary: 0101 1001
- LEDs ON: 0, 3, 4, 6
- Duration: 3 seconds

Sorted Values Display (2 seconds each):

1. **89** → Binary: 0101 1001 → LEDs 0,3,4,6 ON
2. **67** → Binary: 0100 0011 → LEDs 0,1,6 ON
3. **45** → Binary: 0010 1101 → LEDs 0,2,3,5 ON
4. **34** → Binary: 0010 0010 → LEDs 1,5 ON
5. **23** → Binary: 0001 0111 → LEDs 0,1,2,4 ON
6. **12** → Binary: 0000 1100 → LEDs 2,3 ON

Continuous Loop:

- Cycles through all 6 sorted values repeatedly
- Each value displayed for 2 seconds
- Loop continues indefinitely

Example:

- **Original array**: 45, 23, 67, 12, 89, 34
- **Maximum found**: 89
- **After sorting**: 89, 67, 45, 34, 23, 12 (descending order)
- **LED display**: Binary representation of each number visible as lit LEDs

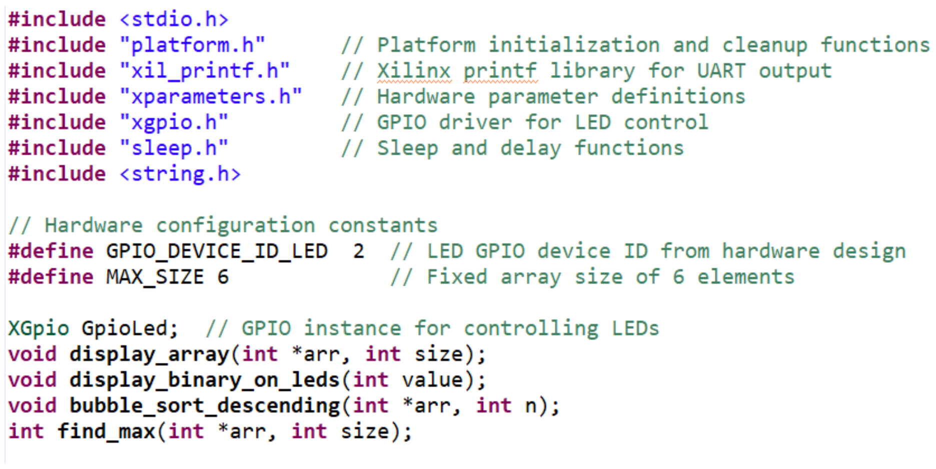
Activities

1. Write the code as shown in Figure 12(a)-(d).

   `Figure 12(a): Header files and function declarations`

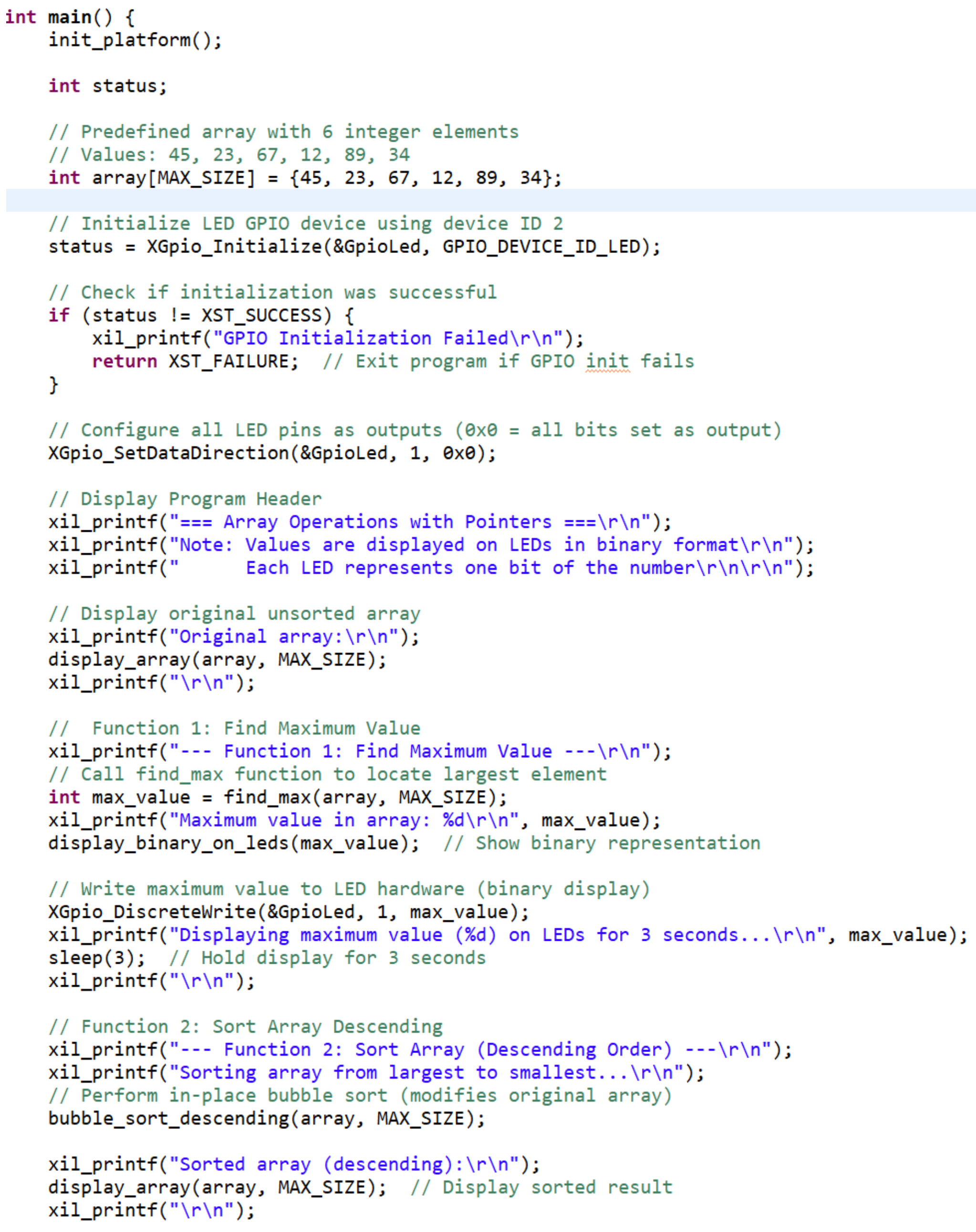
   `Figure 12(b-c): Main function`

   `Figure 12(d): Function implementations`



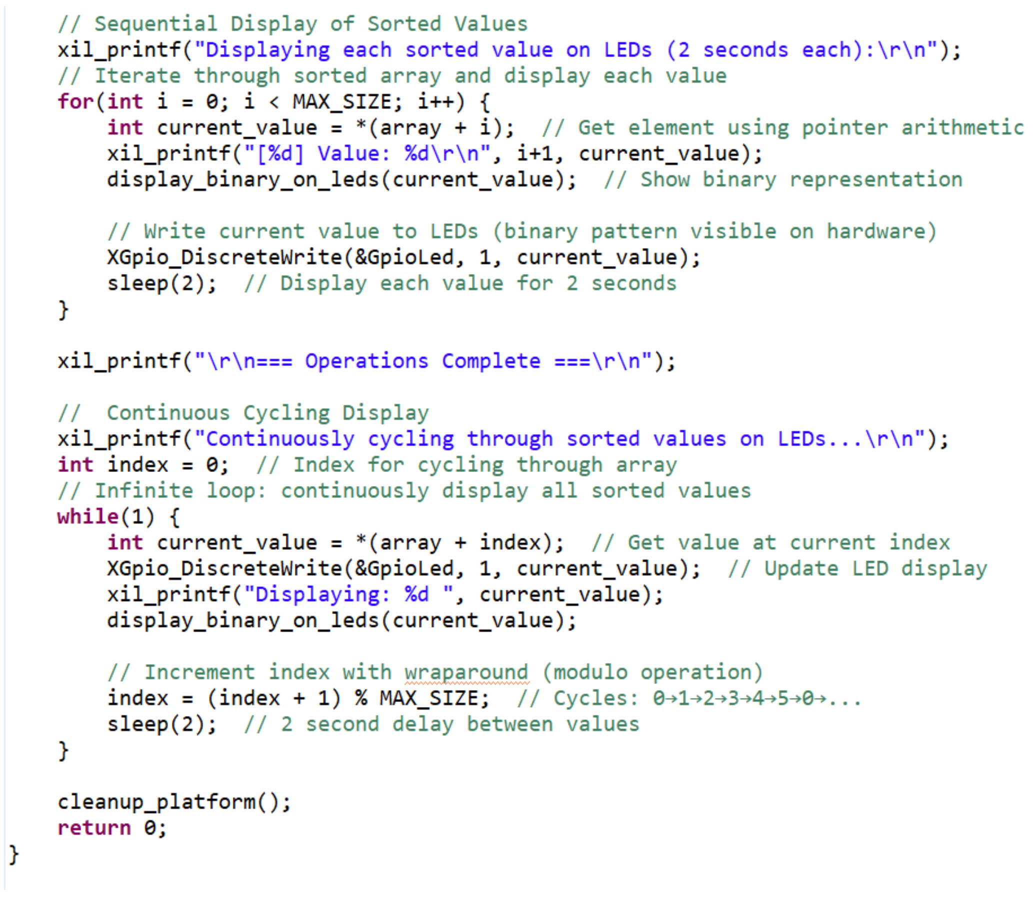
```c
#include <stdio.h>
#include "platform.h"      // Platform initialization and cleanup functions
#include "xil_printf.h"    // Xilinx printf library for UART output
#include "xparameters.h"   // Hardware parameter definitions
#include "xgpio.h"         // GPIO driver for LED control
#include "sleep.h"         // Sleep and delay functions
#include <string.h>

// Hardware configuration constants
#define GPIO_DEVICE_ID_LED  2  // LED GPIO device ID from hardware design
#define MAX_SIZE 6             // Fixed array size of 6 elements

XGpio GpioLed;  // GPIO instance for controlling LEDs
void display_array(int *arr, int size);
void display_binary_on_leds(int value);
void bubble_sort_descending(int *arr, int n);
int find_max(int *arr, int size);
```

Figure 12(a)



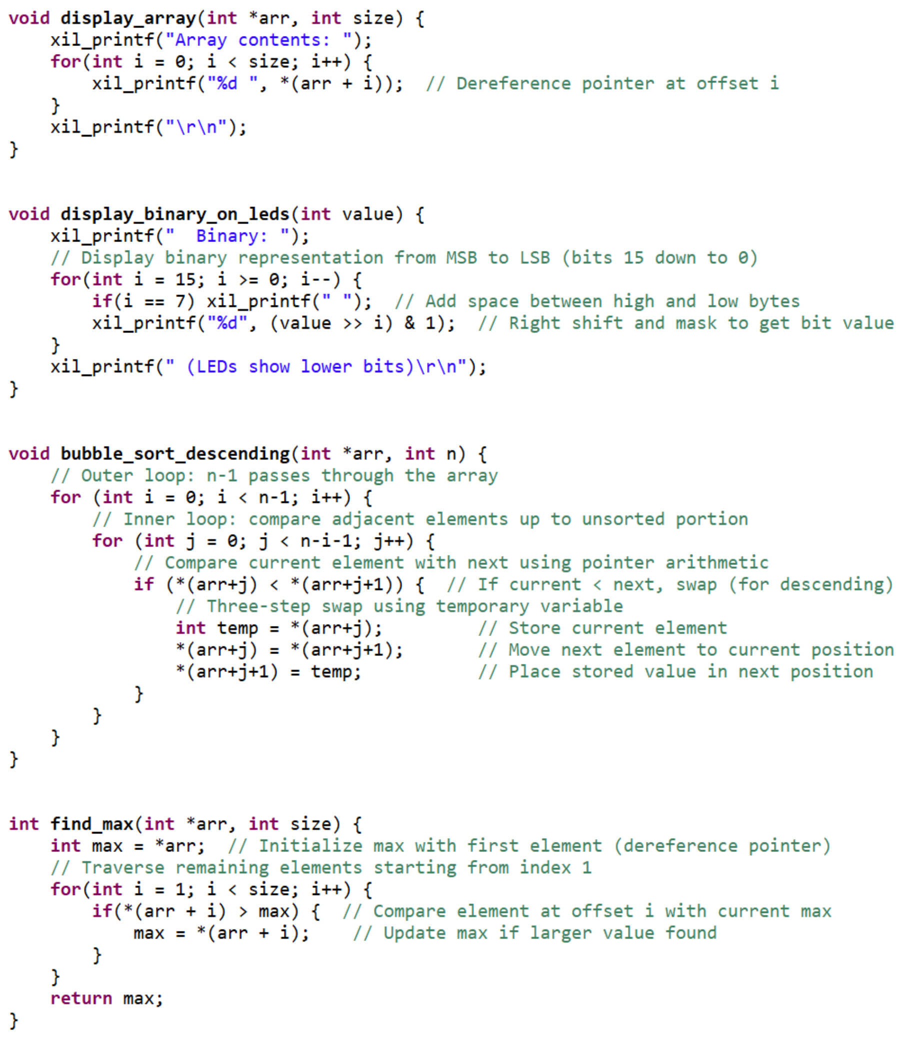
```c
int main() {
    init_platform();

    int status;

    // Predefined array with 6 integer elements
    // Values: 45, 23, 67, 12, 89, 34
    int array[MAX_SIZE] = {45, 23, 67, 12, 89, 34};

    // Initialize LED GPIO device using device ID 2
    status = XGpio_Initialize(&GpioLed, GPIO_DEVICE_ID_LED);

    // Check if initialization was successful
    if (status != XST_SUCCESS) {
        xil_printf("GPIO Initialization Failed\r\n");
        return XST_FAILURE;  // Exit program if GPIO init fails
    }

    // Configure all LED pins as outputs (0x0 = all bits set as output)
    XGpio_SetDataDirection(&GpioLed, 1, 0x0);

    // Display Program Header
    xil_printf("=== Array Operations with Pointers ===\r\n");
    xil_printf("Note: Values are displayed on LEDs in binary format\r\n");
    xil_printf("      Each LED represents one bit of the number\r\n\r\n");

    // Display original unsorted array
    xil_printf("Original array:\r\n");
    display_array(array, MAX_SIZE);
    xil_printf("\r\n");

    //  Function 1: Find Maximum Value
    xil_printf("--- Function 1: Find Maximum Value ---\r\n");
    // Call find_max function to locate largest element
    int max_value = find_max(array, MAX_SIZE);
    xil_printf("Maximum value in array: %d\r\n", max_value);
    display_binary_on_leds(max_value);  // Show binary representation

    // Write maximum value to LED hardware (binary display)
    XGpio_DiscreteWrite(&GpioLed, 1, max_value);
    xil_printf("Displaying maximum value (%d) on LEDs for 3 seconds...\r\n", max_value);
    sleep(3);  // Hold display for 3 seconds
    xil_printf("\r\n");

    // Function 2: Sort Array Descending
    xil_printf("--- Function 2: Sort Array (Descending Order) ---\r\n");
    xil_printf("Sorting array from largest to smallest...\r\n");
    // Perform in-place bubble sort (modifies original array)
    bubble_sort_descending(array, MAX_SIZE);

    xil_printf("Sorted array (descending):\r\n");
    display_array(array, MAX_SIZE);  // Display sorted result
    xil_printf("\r\n");
```

Figure 12(b)



```c
    // Sequential Display of Sorted Values
    xil_printf("Displaying each sorted value on LEDs (2 seconds each):\r\n");
    // Iterate through sorted array and display each value
    for(int i = 0; i < MAX_SIZE; i++) {
        int current_value = *(array + i);  // Get element using pointer arithmetic
        xil_printf("[%d] Value: %d\r\n", i+1, current_value);
        display_binary_on_leds(current_value);  // Show binary representation

        // Write current value to LEDs (binary pattern visible on hardware)
        XGpio_DiscreteWrite(&GpioLed, 1, current_value);
        sleep(2);  // Display each value for 2 seconds
    }

    xil_printf("\r\n=== Operations Complete ===\r\n");

    //  Continuous Cycling Display
    xil_printf("Continuously cycling through sorted values on LEDs...\r\n");
    int index = 0;  // Index for cycling through array
    // Infinite loop: continuously display all sorted values
    while(1) {
        int current_value = *(array + index);  // Get value at current index
        XGpio_DiscreteWrite(&GpioLed, 1, current_value);  // Update LED display
        xil_printf("Displaying: %d ", current_value);
        display_binary_on_leds(current_value);

        // Increment index with wraparound (modulo operation)
        index = (index + 1) % MAX_SIZE;  // Cycles: 0→1→2→3→4→5→0→...
        sleep(2);  // 2 second delay between values
    }

    cleanup_platform();
    return 0;
}
```

Figure 12(c)



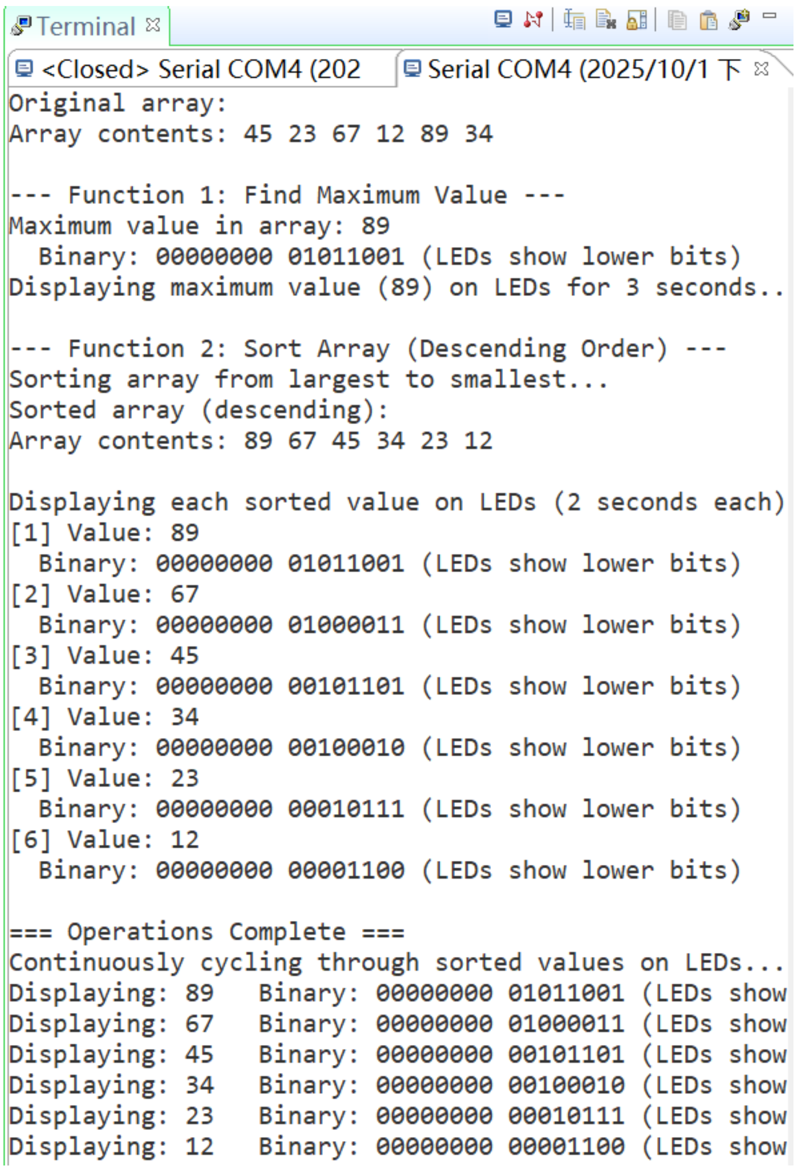
```c
void display_array(int *arr, int size) {
    xil_printf("Array contents: ");
    for(int i = 0; i < size; i++) {
        xil_printf("%d ", *(arr + i));  // Dereference pointer at offset i
    }
    xil_printf("\r\n");
}


void display_binary_on_leds(int value) {
    xil_printf("  Binary: ");
    // Display binary representation from MSB to LSB (bits 15 down to 0)
    for(int i = 15; i >= 0; i--) {
        if(i == 7) xil_printf(" ");  // Add space between high and low bytes
        xil_printf("%d", (value >> i) & 1);  // Right shift and mask to get bit value
    }
    xil_printf(" (LEDs show lower bits)\r\n");
}


void bubble_sort_descending(int *arr, int n) {
    // Outer loop: n-1 passes through the array
    for (int i = 0; i < n-1; i++) {
        // Inner loop: compare adjacent elements up to unsorted portion
        for (int j = 0; j < n-i-1; j++) {
            // Compare current element with next using pointer arithmetic
            if (*(arr+j) < *(arr+j+1)) {  // If current < next, swap (for descending)
                // Three-step swap using temporary variable
                int temp = *(arr+j);         // Store current element
                *(arr+j) = *(arr+j+1);       // Move next element to current position
                *(arr+j+1) = temp;           // Place stored value in next position
            }
        }
    }
}


int find_max(int *arr, int size) {
    int max = *arr;  // Initialize max with first element (dereference pointer)
    // Traverse remaining elements starting from index 1
    for(int i = 1; i < size; i++) {
        if(*(arr + i) > max) {  // Compare element at offset i with current max
            max = *(arr + i);    // Update max if larger value found
        }
    }
    return max;
}
```

Figure 12(d)

Note: This embedded C program demonstrates pointer-based array operations on a Xilinx platform with a predefined dataset. The program uses a single GPIO device (ID 2) to control LEDs for displaying numerical values in binary format. It initializes with a fixed array of six integers: {45, 23, 67, 12, 89, 34}. The program automatically executes two main functions without requiring user input: first, it finds the maximum value (89) using pointer traversal through the find_max() function and displays it on the LEDs for 3 seconds; second, it sorts the array in descending order using the bubble_sort_descending() function with pointer arithmetic, resulting in {89, 67, 45, 34, 23, 12}. After sorting, each value is sequentially displayed on the LEDs with 2-second delays. All array operations utilize pointer notation *(arr+i) rather than array indexing to demonstrate pointer manipulation techniques. The program concludes by entering an infinite loop that continuously cycles through all sorted values, displaying each for 2 seconds on the LEDs in binary format. Status messages and binary representations are output via UART using xil_printf to provide feedback about operations performed, with each LED representing one bit of the displayed integer value.

2. The program operates automatically without user input. It begins with a predefined array of six elements: {45, 23, 67, 12, 89, 34}. Upon initialization, the program automatically executes two core functions: first, it finds the maximum value (89) in the array using the find_max() function

and displays it on the LEDs in binary format for 3 seconds; second, it sorts the array in descending order (from highest to lowest) using the bubble_sort_descending() function, resulting in {89, 67, 45, 34, 23, 12}. After sorting, each value is sequentially displayed on the LEDs with 2-second intervals. The program then enters a continuous loop that cycles through all sorted values, displaying each on the LEDs in binary format. Sample output showing the original array, maximum value detection, and sorted results is displayed via UART terminal as shown in **Figure 13.**



```
Original array:
Array contents: 45 23 67 12 89 34

--- Function 1: Find Maximum Value ---
Maximum value in array: 89
  Binary: 00000000 01011001 (LEDs show lower bits)
Displaying maximum value (89) on LEDs for 3 seconds..

--- Function 2: Sort Array (Descending Order) ---
Sorting array from largest to smallest...
Sorted array (descending):
Array contents: 89 67 45 34 23 12

Displaying each sorted value on LEDs (2 seconds each)
[1] Value: 89
  Binary: 00000000 01011001 (LEDs show lower bits)
[2] Value: 67
  Binary: 00000000 01000011 (LEDs show lower bits)
[3] Value: 45
  Binary: 00000000 00101101 (LEDs show lower bits)
[4] Value: 34
  Binary: 00000000 00100010 (LEDs show lower bits)
[5] Value: 23
  Binary: 00000000 00010111 (LEDs show lower bits)
[6] Value: 12
  Binary: 00000000 00001100 (LEDs show lower bits)

=== Operations Complete ===
Continuously cycling through sorted values on LEDs...
Displaying: 89   Binary: 00000000 01011001 (LEDs show
Displaying: 67   Binary: 00000000 01000011 (LEDs show
Displaying: 45   Binary: 00000000 00101101 (LEDs show
Displaying: 34   Binary: 00000000 00100010 (LEDs show
Displaying: 23   Binary: 00000000 00010111 (LEDs show
Displaying: 12   Binary: 00000000 00001100 (LEDs show
```

Figure 13

References:

You can go to the lab4 reference in the learn section, or directly copy the link to your browser:

1. Array of Pointers in C https://www.geeksforgeeks.org/c/array-of-pointers-in-c/
2. Difference between Arrays and Pointers https://www.geeksforgeeks.org/c/difference-between-array-and-pointers/