

# LAB 4

## WRITING BASIC SOFTWARE APPLICATION

### AIM

To develop a **bare-metal software application** using **Vitis IDE** to access a custom AXI peripheral, execute the application from **BRAM**, and verify the functionality on the **Zybo (Zynq-7000) board**.

### OBJECTIVE

- Write a basic application to access an IP peripheral in Vitis IDE.
- Generate an ELF executable file
- Modify the linker script to execute code from BRAM
- Download the bitstream and application onto the Zynq board
- Verify correct hardware–software interaction

### HARDWARE AND SOFTWARE USED

Category	Description
FPGA Board	Xilinx Zybo (Zynq-7000)
Processor	ARM Cortex-A9
IDE	Vivado & Vitis
Language	C
Memory	BRAM and DDR

### THEORY

In Zynq-based embedded systems, software applications run on the **ARM Cortex-A9 processor** in the Processing System (PS). Custom peripherals implemented in the Programmable Logic (PL) are accessed through **AXI memory-mapped interfaces**.

In this lab, a **bare-metal application** is developed using **Vitis IDE** to control the custom AXI LED peripheral created in Lab 3. The application is configured to execute from **Block RAM (BRAM)** by modifying the **linker script**, enabling faster execution due to lower memory access latency compared to external DDR memory.

## PROCEDURE

1. The Vivado project created in **Lab 3** was opened and saved as a new project named **lab4** using the *Save As* option.
2. The hardware design was exported from Vivado by including the bitstream and launching **Vitis IDE**.
3. A new **Application Project** was created in Vitis using the exported **XSA file**, and a standalone bare-metal domain was selected.
4. An **Empty Application (C)** template was chosen, and the source file lab4.c was added to the project.
5. The GPIO driver APIs were studied using the **Board Support Package (BSP) documentation** to understand GPIO initialization and data access functions.
6. The xparameters.h file was examined to identify the base addresses and device IDs for **DIP switches, push buttons, and the custom LED IP**.
7. The C program was modified to initialize GPIO peripherals, read DIP switch and push button values, and display them on the serial terminal.
8. The DIP switch values were written to the custom LED AXI peripheral using the driver API `LED_IP_mWriteReg()`.
9. The application was built successfully, generating the **ELF executable file**.
10. The Zybo board was configured in **JTAG boot mode**, and the bitstream and application were downloaded to the hardware.
11. The program output was verified using the **Vitis Serial Terminal**, and the LED behavior was observed on the board.

## SOFTWARE DESCRIPTION

The software application is developed as a **bare-metal C program** using the **Vitis Unified IDE** to run on the ARM Cortex-A9 processor in the Processing System (PS) of the Zynq FPGA. The application utilizes the **Xilinx GPIO driver** to interface with on-board DIP switches and push buttons, and a **custom AXI LED IP driver** to control the LED outputs implemented in the Programmable Logic (PL).

The program begins by initializing the GPIO peripherals using device base addresses defined in the xparameters.h file. The GPIO data direction is configured such that all DIP switch and push-button signals act as inputs. During execution, the application continuously reads the status of the DIP switches and push buttons using discrete read operations.

The DIP switch value is then written to the custom LED AXI peripheral using the memory-mapped register write function `LED_IP_mWriteReg()`. This function generates an AXI write transaction that is detected by the custom hardware logic in the PL, which updates the LED output accordingly. Serial output messages are printed to the terminal for monitoring and verification.

A simple software delay is included to control the update rate of the LED display. The application runs in an infinite loop, thereby continuously demonstrating real-time interaction between the Processing System software and Programmable Logic hardware.

## CODE

```
#include "xparameters.h"
#include "xgpio.h"
#include "led_ip.h"

int main(void)
{
    XGpio dip, push;
    int i, psb_check, dip_check;

    xil_printf("-- Start of the Program --\r\n");

    /* Initialize DIP switches */
    XGpio_Initialize(&dip, XPAR_SWITCHES_BASEADDR);
    XGpio_SetDataDirection(&dip, 1, 0xFFFFFFFF);

    /* Initialize Push Buttons */
    XGpio_Initialize(&push, XPAR_BUTTONS_BASEADDR);
    XGpio_SetDataDirection(&push, 1, 0xFFFFFFFF);

    while (1)
    {
        /* Read Push Button status */
        psb_check = XGpio_DiscreteRead(&push, 1);
        xil_printf("Push Buttons Status %x\r\n", psb_check);

        /* Read DIP switch status */
        dip_check = XGpio_DiscreteRead(&dip, 1);
        xil_printf("DIP Switch Status %x\r\n", dip_check);

        /* Write DIP switch value to custom LED IP */
        LED_IP_mWriteReg(XPAR_LED_IP_BASEADDR, 0,
dip_check);

        /* Simple delay */
        for (i = 0; i < 9999999; i++);
    }

    return 0;
}
```

The software application initializes the GPIO peripherals to interface with the on-board DIP switches and push buttons. During execution, the program continuously reads the DIP switch values using the XGpio\_DiscreteRead() function. The obtained DIP switch value is

then written to the custom AXI LED IP using the `LED_IP_mWriteReg()` function. This written value is reflected on the LEDs through AXI-based memory-mapped communication between the Processing System and the Programmable Logic. Additionally, the serial terminal displays the DIP switch and push button status for real-time verification of system operation. The function `LED_IP_mWriteReg()` performs a memory-mapped AXI write transaction to the custom LED IP register, and the Verilog logic implemented in the Programmable Logic updates the LED outputs whenever a valid AXI write occurs at register offset 0.

## OBSERVATION

- The hardware platform was successfully imported into Vitis IDE.
- The application compiled without errors and the **ELF file was generated successfully.**
- The serial terminal displayed the program start message indicating correct execution.
- DIP switch and push button values were continuously read and displayed on the serial monitor.
- The DIP switch values were correctly written to the **custom AXI LED IP**, and the LED output reflected the switch status.
- The application demonstrated correct interaction between **Processing System software** and **Programmable Logic hardware**.

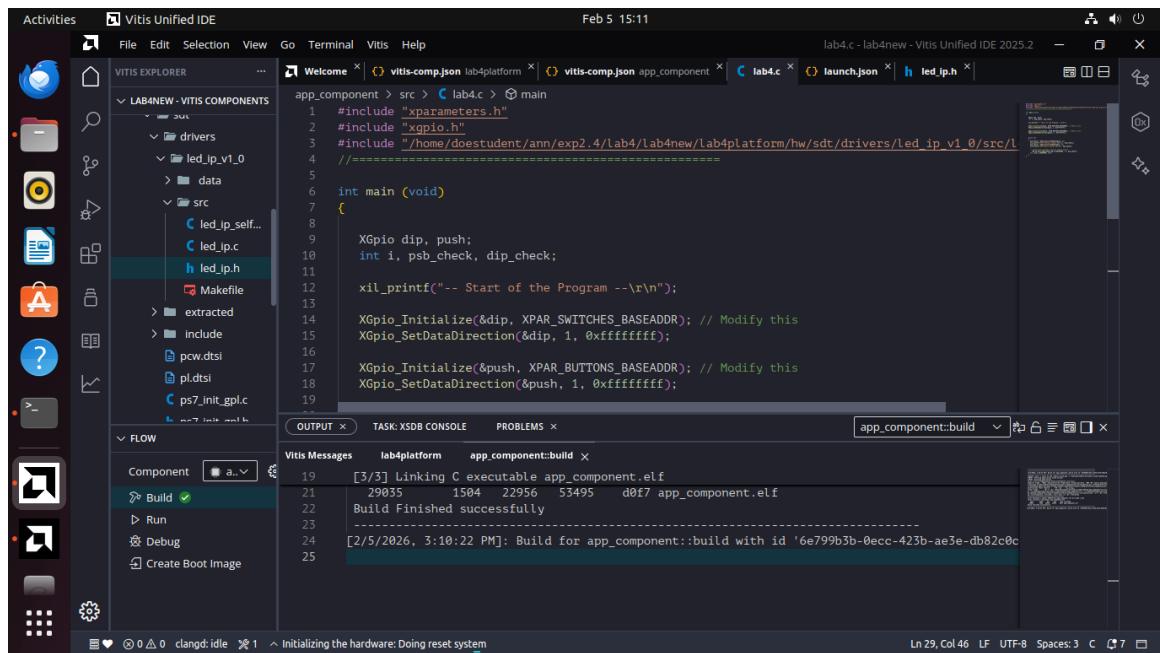


Fig 1: Vitis IDE showing C source code for accessing GPIO inputs and writing values to custom AXI LED IP

The screenshot shows the Vitis Unified IDE interface. The left sidebar contains a 'VITIS EXPLORER' tree with 'LAB4NEW - VITIS COMPONENTS' expanded, showing sub-folders like 'drivers', 'led\_ip\_v1\_0', and 'src'. The 'src' folder contains files such as 'led\_ip\_self...', 'led\_ip.c', 'led\_ip.h', and 'Makefile'. The main workspace displays a C code editor with the file 'lab4.c' open. The code includes XGpio\_SetDataDirection and XGpio\_DiscreteRead functions. Below the editor is an 'OUTPUT' tab showing build logs, which indicate a successful build of 'app\_component' with ID '6e799b3b-0ecc-423b-ae3e-db82c0c'. The bottom status bar shows the date and time as 'Feb 5 15:11'.

Fig 2: Vitis IDE showing C source code for accessing GPIO inputs and writing values to custom AXI LED IP

This screenshot shows the Vitis Unified IDE with the 'FLOW' tab selected. The 'Run' option is highlighted. The main workspace displays the same 'lab4.c' code as in Fig 2. The 'OUTPUT' tab is active, showing a serial terminal window titled '/DEV/TTYUSB1 (DIGILENT)'. The terminal output shows the start of the program and various GPIO status readings. A message at the bottom right of the terminal window states 'Build for app\_component:build finished successfully.' The bottom status bar shows the date and time as 'Feb 5 15:16'.

Fig 3: Serial terminal output showing start of program execution

The screenshot shows the Vitis Unified IDE interface. In the center, there is a code editor window displaying C code for a main function. The code includes includes for xparameters.h and xgpio.h, and defines XGpio dip, push; int i, psb\_check, dip\_check;. It then prints a start message and initializes XGpio for dip and push buttons. Below the code editor is a terminal window titled "/DEV/TTYUSB1 (DIGILENT)". The terminal output shows the application reading DIP switch and push button statuses. The first few lines of the output are:

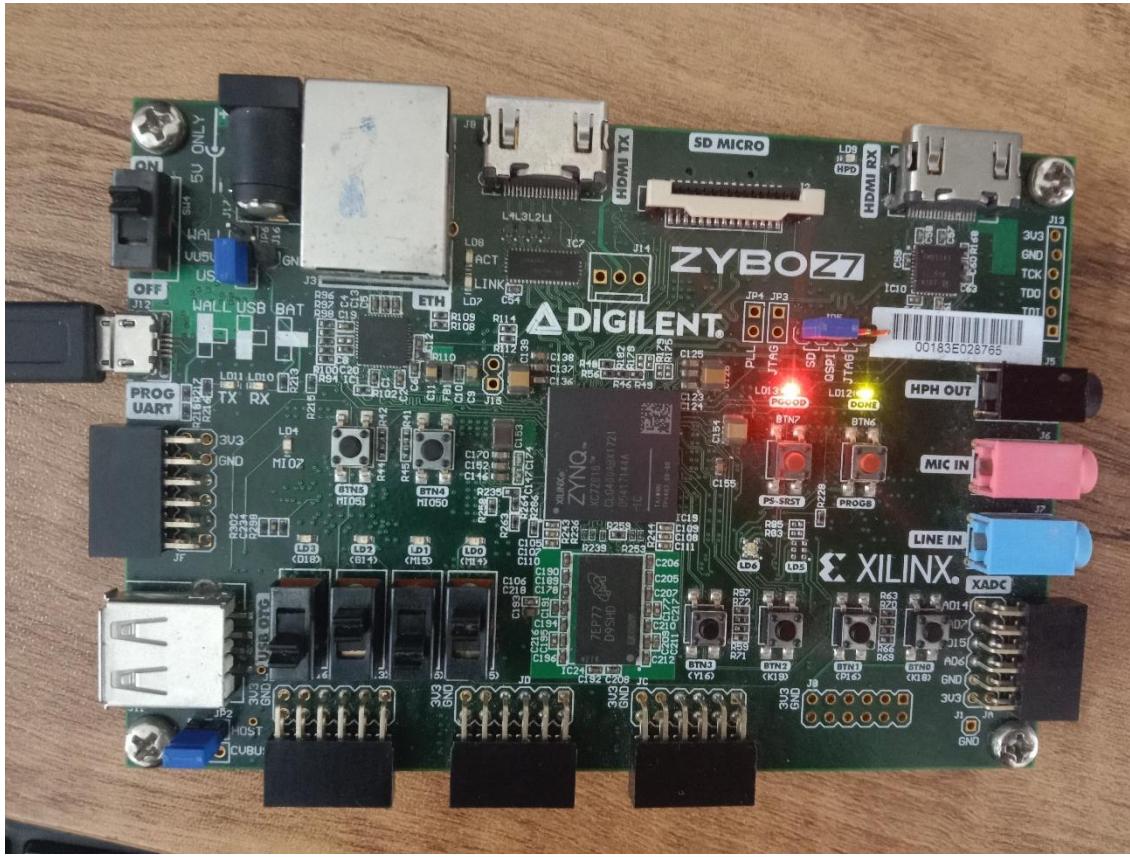
```
Push Buttons Status 0
DIP Switch Status 0
Push Buttons Status 0
```

Fig 4: Serial terminal output showing DIP switch and push button status read by the application

This screenshot is identical to Fig 4, showing the same Vitis Unified IDE interface and serial terminal output. The terminal window displays the application's read of DIP switch and push button statuses, with the first few lines being:

```
Push Buttons Status 6
DIP Switch Status 5
```

Fig 5 : Serial terminal output showing DIP switch and push button status read by the application



**Fig 6:** LED Output Corresponding to DIP Switch Value in Zybo board

## RESULTS

Thus, a bare-metal software application was successfully developed using Vitis IDE, executed from BRAM, and verified on the Zynq-7000 Zybo board by controlling the custom AXI LED peripheral.

## CONCLUSION

This lab demonstrated the complete software development flow for a Zynq-based embedded system. By executing the application from BRAM and accessing AXI-mapped peripherals, efficient hardware-software interaction was achieved. The experiment reinforced concepts of linker script modification, ELF generation, and embedded software execution on FPGA-based SoCs.

## REFERENCE

[https://xilinx.github.io/xup\\_embedded\\_system\\_design\\_flow/lab4.html](https://xilinx.github.io/xup_embedded_system_design_flow/lab4.html)