Laboratory 7 – VGA Applications

I. Colour the World

Objectives

- To build an application with VGA display
- To learn how to perform hardware and software combined design flow with Xilinx Vitis
- To use the slide switches input to control VGA display
- To understand how to call slide switches and control them

Activity Summary

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

Introduction

1. Integrating Hardware and Software for MicroBlaze on FPGA

The Figure 1. below illustrates the standard design flow for the **MicroBlaze soft processor**. The paths highlighted in **red boxes** represent the specific steps followed in our laboratory sessions.

The complete design flow for working with the MicroBlaze soft processor on an FPGA can be divided into two major phases: the hardware design phase in Vivado and the software development phase in Vitis.

In the hardware design phase, the process begins by adding the MicroBlaze processor core into the block design in Vivado, together with the necessary peripherals such as UART for serial communication, GPIO for input and output control, and timers for event scheduling. Once the system is defined, the design is synthesized, meaning that the HDL description is translated into a gate-level representation of the circuit. The implementation step then performs placement and routing, which maps the synthesized logic elements onto the physical FPGA resources. After this, a configuration file known as a bitstream (.bit) is generated, which can be used to program the FPGA hardware. Finally, Vivado exports a hardware platform description file (.xml), which contains all the essential information about the processor, its connected peripherals, memory mapping, and address configuration. This exported file provides the bridge between the hardware design in Vivado and the software development in Vitis.

In the software development phase, the exported hardware platform is imported into Xilinx Vitis. Based on this platform, a new software project is created, for example an LED control program or a VGA display program. The software is written in C or C++, compiled, and built into an executable file called program.elf. This file contains the machine code that will run on the MicroBlaze processor. The executable is then associated with the hardware bitstream so that both the FPGA configuration and the software application can work together. Once linked, both the bitstream and the ELF file are downloaded onto the FPGA board. At this point, the system can be tested, executed in real-time, or debugged interactively inside Vitis.

In laboratory sessions, students typically follow only the essential path through this flow, which is also highlighted in red in the reference flowchart. These steps include: first, adding the MicroBlaze processor in Vivado, then synthesizing, implementing, and generating the FPGA bitstream; second, exporting the hardware platform from Vivado into Vitis; and third, creating a software application in Vitis, compiling it to produce the ELF file, and finally downloading it together with the bitstream onto the FPGA for execution. This streamlined sequence ensures that students focus on the critical stages of hardware and software integration while avoiding unnecessary complexity.

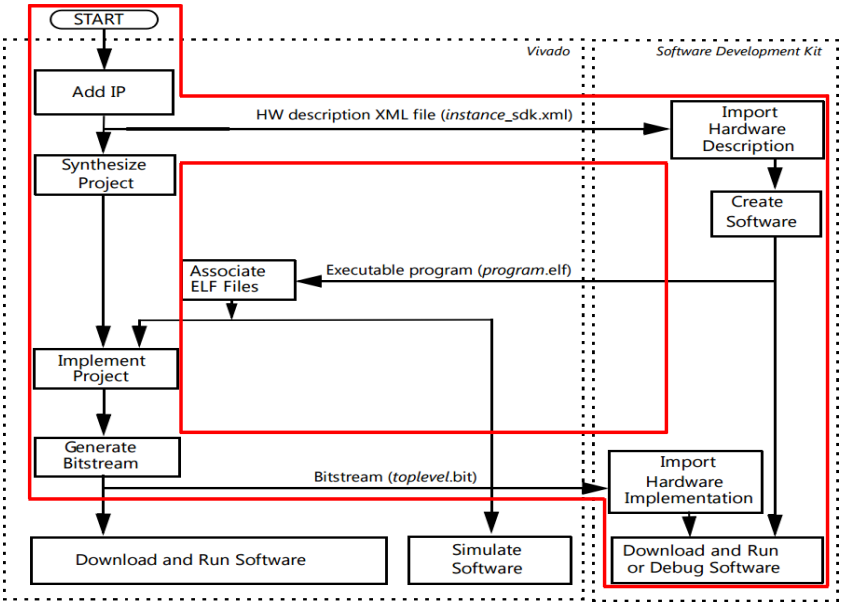

Figure 1: The generic Vivado tool flow for MicroBlaze

2. What is VGA

The Video Graphics Array (VGA) is a widely used display standard that was originally developed by IBM in 1987. Although it has since been surpassed by more modern digital interfaces such as HDMI and DisplayPort, VGA remains common in educational and embedded systems because of its simplicity and ease of implementation on FPGA boards like the Basys3. A VGA interface transfers video signals to a monitor using analog voltage levels, where the intensity of each primary colour (Red, Green, and Blue) is controlled by digital values provided by the FPGA and converted into corresponding voltage levels on the VGA connector pins[1].

A VGA signal is made up of several key components. First, there are the colour signals: red, green, and blue (RGB). These three signals are combined in different intensities to form any desired colour. For example, full red with zero green and zero blue produces a pure red pixel, while full intensities of all three channels produce white. On the Basys3 board, each colour channel is represented with 4 bits, giving 16 intensity levels per colour. Since there are three channels, this allows a total of 4096 unique colours ($16 \times 16 \times 16$). The FPGA drives these colour signals through dedicated pins on the VGA port, which the monitor interprets as pixel brightness values.

In addition to the colour channels, VGA also requires two synchronization signals: horizontal sync (HS) and vertical sync (VS). These signals act like timing markers that tell the monitor when to start a new line of pixels (horizontal sync) and when to start a new frame (vertical sync). By carefully controlling these signals, the FPGA ensures that the pixels are displayed in the correct positions on the screen. A standard VGA resolution, such as 640×480 at 60 Hz, means that the FPGA must generate the correct sync pulses and pixel data at a precise rate so that the monitor can render the image smoothly.

Specifications

In this lab session, a VGA driver has already been implemented for you in hardware, and the corresponding bitstream file has been generated. The MicroBlaze processor has also been inserted into the design using a block design in Vivado. Therefore, you will not need to write any Verilog or VHDL

hardware description code.

The main goal of this session is to demonstrate how to display colours on a VGA monitor in hardware using the Basys3 FPGA board. Instead of writing HDL code, you will control the display directly from software, using registers that have already been mapped to the VGA driver hardware.

1. VGA Colour Representation

- On the **Basys3 board**, the VGA output uses **12 bits** to represent colour signals.
- These 12 bits are divided equally among the three colour channels:
  - **4 bits for Red (R)**
  - **4 bits for Green (G)**
  - **4 bits for Blue (B)**
- With 4 bits per colour, each channel can represent values from **0 to 15 (0000 to 1111 in binary)**.
- Combining these channels provides up to **4096 different colours (16 × 16 × 16)**.

In the hardware design, the colour components are driven by a register called **colour**, which is connected to the VGA output ports in the Verilog design. This register is exposed to software via an **XGpio object**, named VGA_COLOUR.

The bit assignment for the colour signals is shown in **Table 1**:

VGA Colour	R	R	R	R	G	G	G	G	B	B	B	B
`XGpio VGA_COLOUR`	Bit3	Bit2	Bit1	Bit0	Bit3	Bit2	Bit1	Bit0	Bit3	Bit2	Bit1	Bit0

Table 1: VGA colour assignment

This means:

- Writing a value to VGA_COLOUR in software will directly determine the colour that is displayed on the screen.
- For example, setting R=1111, G=0000, B=0000 will display **bright red**.

2. VGA Screen Regions

To make the display more structured, the VGA screen has been divided into **9 regions** (see the figure on the monitor). Each region is labeled **Region 0 to Region 8**.

- The specific region to be updated is selected using another XGpio object, called **VGA_REGION**.
- This object assigns **bits 0–8** to represent the corresponding region numbers.

Region	8	7	6	5	4	3	2	1	0
`XGpio VGA_REGION`	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Table 2: VGA region selection mapping to `XGpio VGA_REGION` object

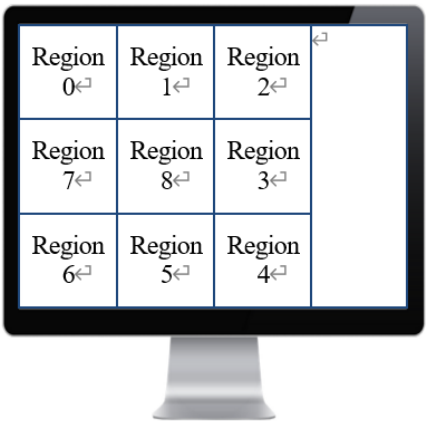

Figure 2. distribution from Region 0 to 9 on screen

This means:

- To change the colour in **Region 3**, you must write a 1 to **Bit3 of VGA_REGION**, while keeping all other bits 0.
- The colour to be displayed in that region is then determined by the value written to **VGA_COLOUR**.
- By combining VGA_REGION and VGA_COLOUR, you can selectively assign different colours to different screen regions.

`XGpio VGA_COLOUR`	**Colour**	**Colour Bit Positions**
Bit 11	Red	3
Bit 10		2
Bit 9		1
Bit 8		0
Bit 7	Green	3
Bit 6		2
Bit 5		1
Bit 4		0
Bit 3	Blue	3
Bit 2		2
Bit 1		1
Bit 0		0

Table 3: VGA 12-bit colour mapping to `XGpio VGA_COLOUR` object

Table 3 shows how the 12-bit colour value used for the VGA output is mapped into the **XGpio object VGA_COLOUR**. The VGA interface on the Basys3 board uses 12 bits in total to represent colour information, divided equally among the three colour channels: **Red, Green, and Blue**. Each channel is assigned **4 bits**, which allows intensity values from **0 (binary 0000) to 15 (binary 1111)**. By combining these three channels, the system can display up to **4096 different colours (16 × 16 × 16)**. The mapping is organised so that:

- **Bits 11–8** represent the **Red channel** (from most significant bit R3 at Bit 11 to least significant bit R0 at Bit 8).
- **Bits 7–4** represent the **Green channel** (from G3 at Bit 7 to G0 at Bit 4).
- **Bits 3–0** represent the **Blue channel** (from B3 at Bit 3 to B0 at Bit 0).

In software, a 12-bit colour value should be generated by packing the three channels together. For example:

- To display **bright red**, set R=1111 (15), G=0000 (0), B=0000 (0), which corresponds to 0xF00.
- To display **bright green**, set R=0, G=15, B=0, giving 0x0F0.
- To display **bright blue**, set R=0, G=0, B=15, giving 0x00F.
- Setting all channels to 1111 (0xFFF) produces **white**, while all zeros (0x000) produces **black**.

When writing colour values in software, the 12-bit packed number is sent to the VGA_COLOUR register of the XGpio object. The hardware then interprets the bits according to this mapping and drives the VGA output with the corresponding Red, Green, and Blue intensity levels.

Activities

1. **Open and Export Hardware Design**

   Follow the procedure described in Lab 1 instructions to unzip and open **Lab 7 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 3a)
2. The Export Hardware Platform wizard opens (Figure 3b). Click **Next** to start the process.
3. In the Output window (Figure 3c), select **Include bitstream** to ensure the complete hardware implementation is included, then click **Next**.
4. In the Files window (Figure 3d), 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 3e) 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.

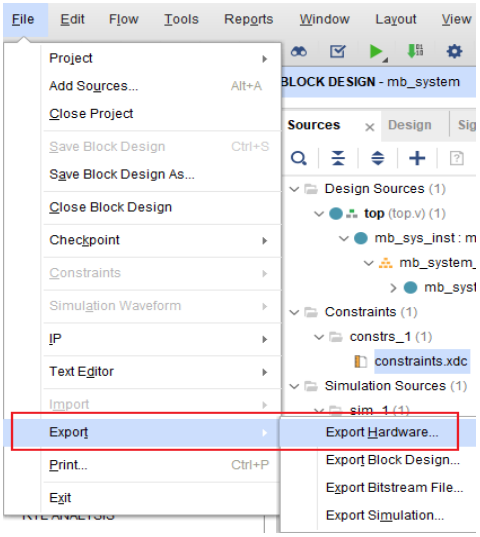

Figure 3(a)

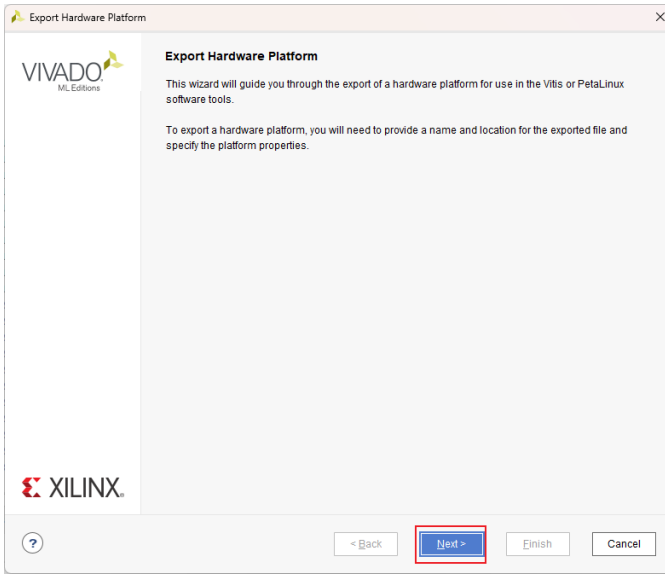

Figure 3(b)

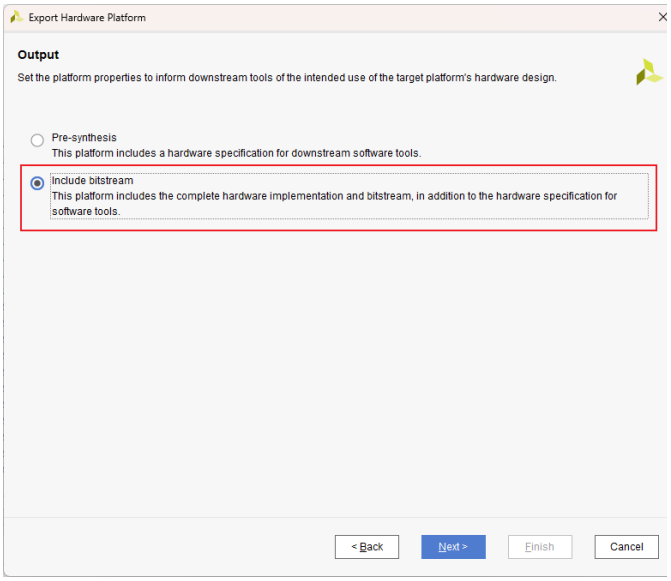

Figure 3(c)

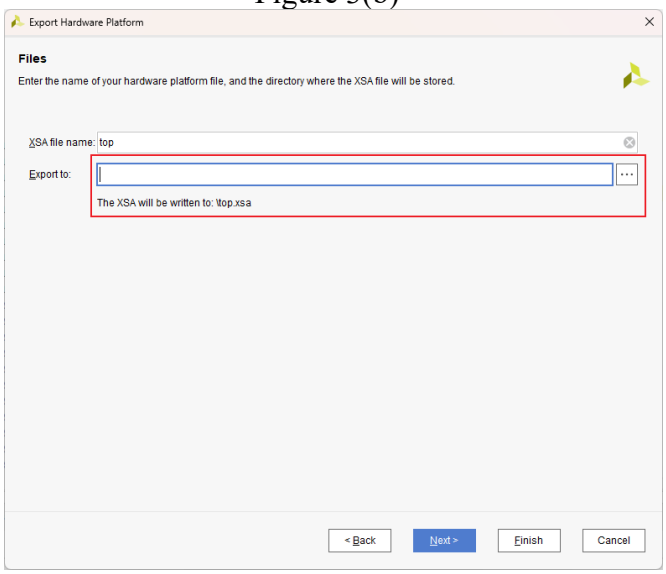

Figure 3(d)

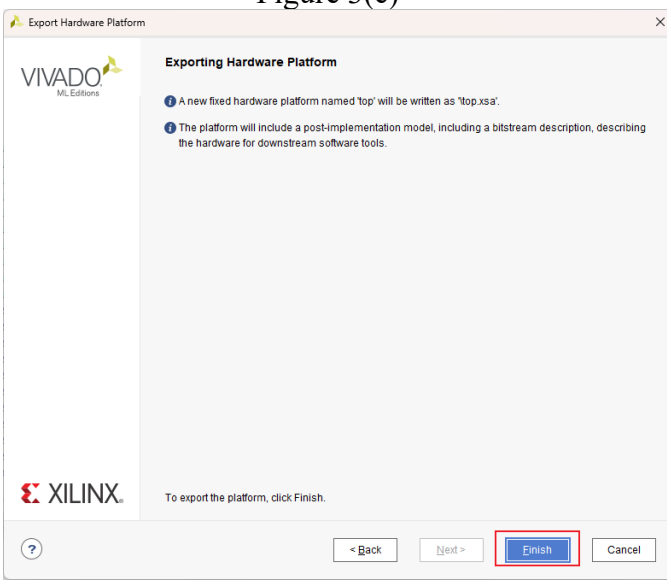

Figure 3(e)

2. Launch Vitis

Create the new folder in the PC, named Lab7, then open Vitis and select the file path of Lab7 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)**

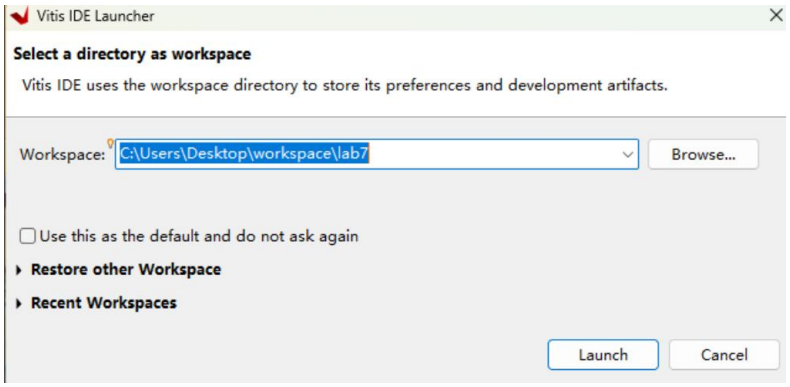

Figure 4

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 5a), 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 5b), 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 5c).

Step 3: Configure Platform On the platform selection page (Figure 5d), 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 5e), 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 5f), 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 5g), 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 Lab7 embedded application.

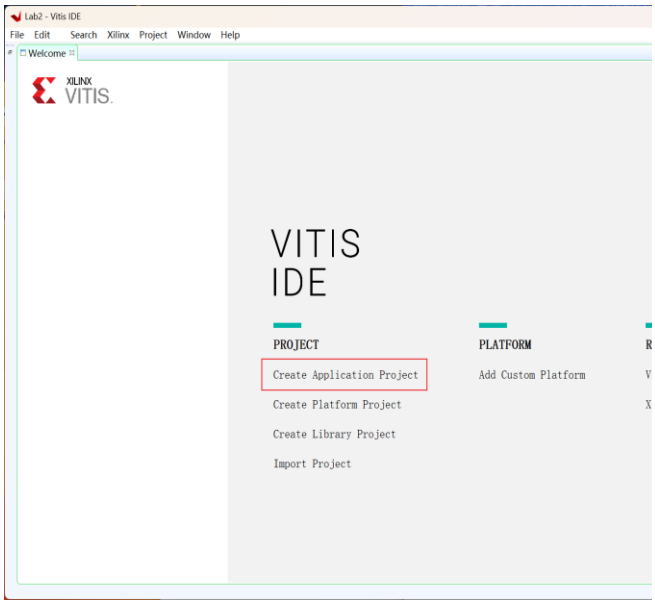

Figure 5(a)

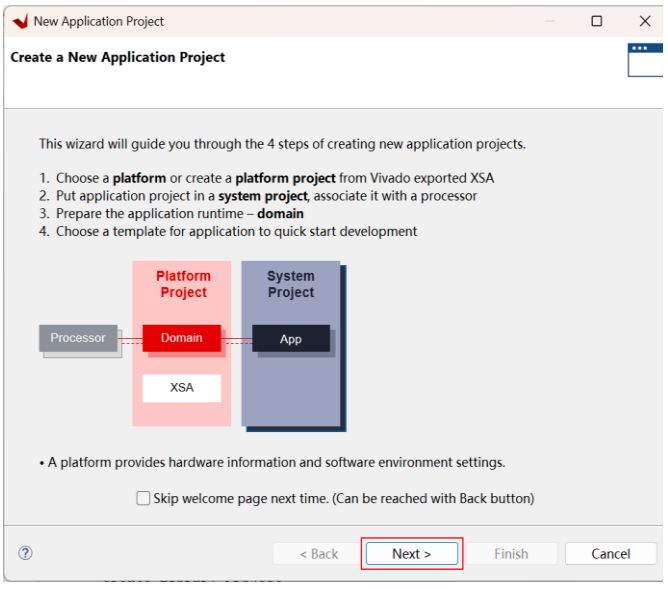

Figure 5(b)

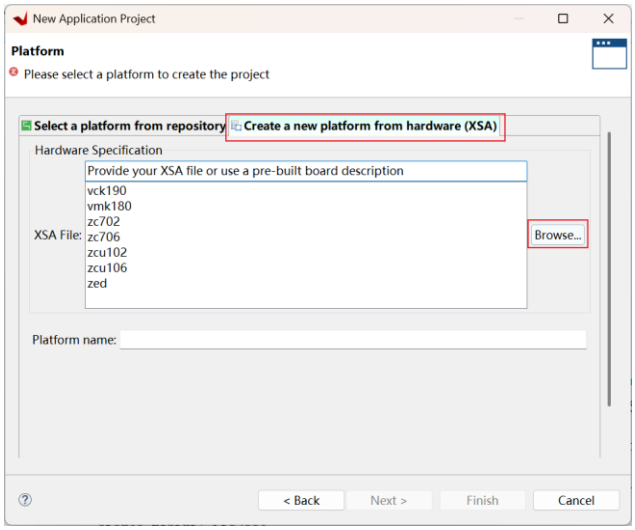

Figure 5(c)

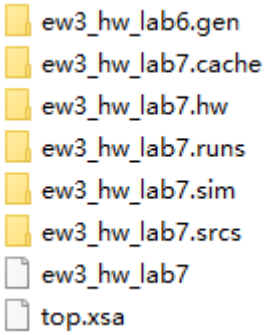

Figure 5(d)

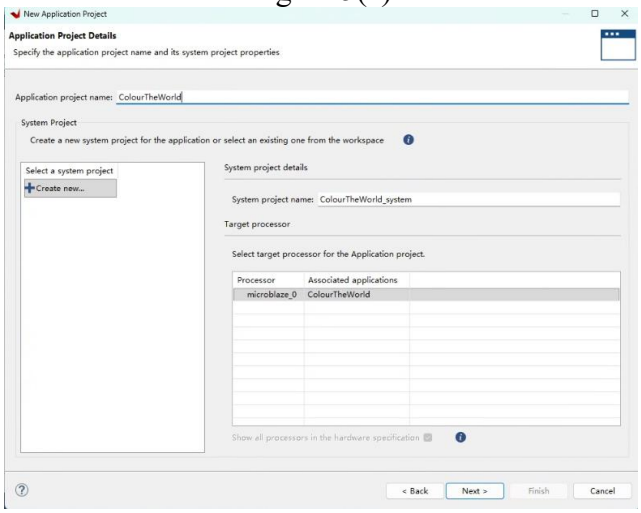

Figure 5(e)

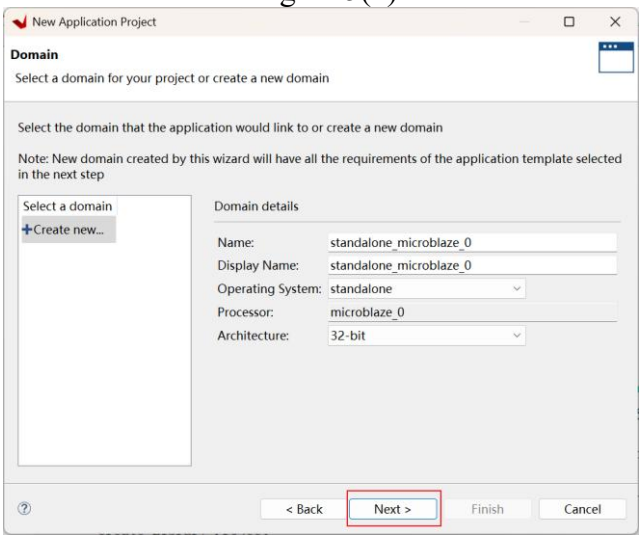

Figure 5(f)

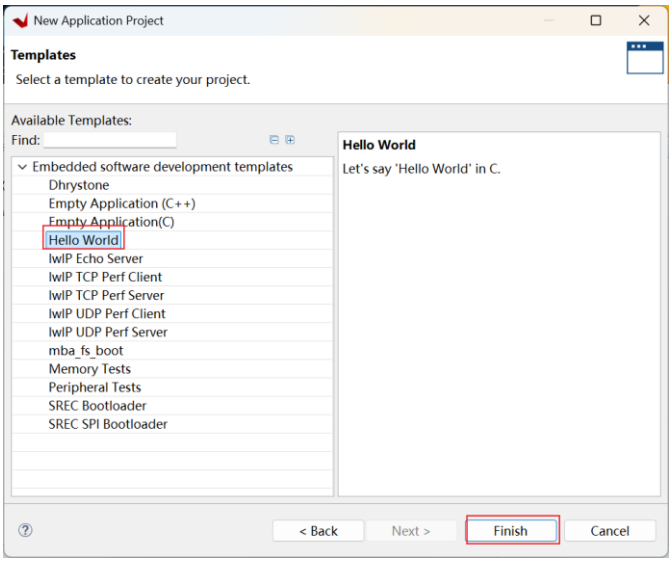

Figure 5(g)

4. Create and Rename the Source File

- As shown in the Figure 6(a), in **Project Explorer**, expand the **Colour_the_world → src** folder.
- Right-click on helloworld.c, select **Rename…**, and rename the file to Colour_the_world.c.

- As shown in Figure 6(b), Copy the gpio_init.c and gpio_init.h files into the **src** folder. you can find them in **Lab7 - codes** on learn, The explanation and comments as shown in Figure 6(c) and Figure 6(d).

The files gpio_init.c and gpio_init.h together provide the foundation for setting up the General-Purpose Input/Output (GPIO) devices that are required in the VGA experiments on the Basys3 FPGA board. These files ensure that the MicroBlaze processor can communicate with both the slide switches, which provide user input, and the VGA driver, which controls the colours and regions displayed on the monitor.

The header file gpio_init.h serves two main purposes. First, it declares the function initGpio(), which will later be called from the main program to initialise all the GPIO devices. Second, it declares three external variables: SLIDE_SWITCHES, VGA_REGION, and VGA_COLOUR. Each of these is an instance of the XGpio data type provided by Xilinx. Declaring them as extern allows the variables to be defined once in the source file and then accessed from other files such as main.c or ColourtheWorld.c. This design follows the principle of separating declarations from definitions, which makes the program structure clearer and easier to maintain.

The source file gpio_init.c contains the actual definitions of the three XGpio instances and the implementation of the initGpio() function. Each instance corresponds to a specific hardware connection on the Basys3 board: SLIDE_SWITCHES represents the bank of 16 physical slide switches that serve as inputs, VGA_COLOUR represents the 12-bit RGB output used to drive colour signals to the VGA monitor, and VGA_REGION represents the 9-bit one-hot output that selects which region of the VGA screen is active. The initGpio() function calls XGpio_Initialize() for each of these devices, using the correct device ID values that were assigned in the Vivado hardware design. These IDs (7 for slide switches, 8 for VGA colour, and 10 for VGA region) map the logical software objects to the physical FPGA peripherals.

The function returns XST_SUCCESS if all devices are initialised correctly, or XST_FAILURE if any of them fail to initialise. This return value is particularly useful in the main program, since it allows the system to check whether the hardware setup has been completed successfully before attempting to use the devices. If initialisation fails, the program can exit gracefully instead of producing unexpected behaviour.

Once the GPIOs have been initialised by initGpio(), other parts of the program can use the Xilinx GPIO driver functions to interact with the hardware. For example, XGpio_DiscreteRead(&SLIDE_SWITCHES, 1) retrieves the state of the slide switches, while XGpio_DiscreteWrite(&VGA_COLOUR, 1, value) and XGpio_DiscreteWrite(&VGA_REGION, 1, value) send colour and region data to the VGA hardware. In this way, the gpio_init module acts as the bridge between the hardware resources described in Vivado and the C code that runs on the MicroBlaze processor.

- Remove the following two lines of code:

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

  print("Successfully ran Hello World application");

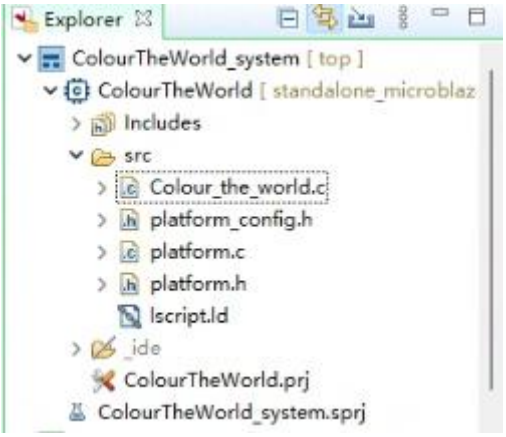

Figure 6(a).

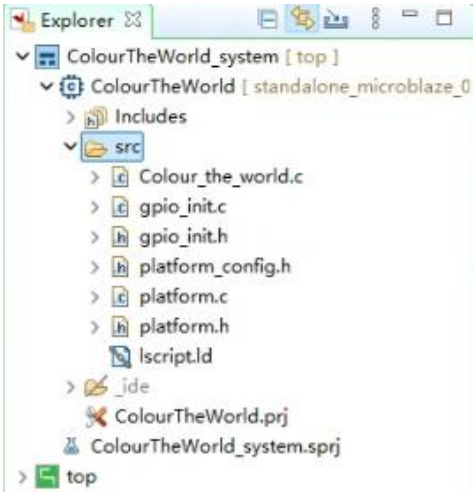

Figure 6(b).

```
#ifndef __GPIO_INIT_H_      // Header guard: prevents the file from being included multiple times
#define __GPIO_INIT_H_

#include "xgpio.h"          // Include the Xilinx GPIO driver definitions (XGpio structure, functions)

// Function prototype for GPIO initialization
// This function sets up all required GPIO peripherals used in the lab
XStatus initGpio(void);

// External declarations of GPIO device instances
// These variables will be defined in gpio_init.c and used in other source files
extern XGpio SLIDE_SWITCHES; // Represents the GPIO device connected to the slide switches
extern XGpio VGA_REGION;     // Represents the GPIO device controlling VGA region selection
extern XGpio VGA_COLOUR;     // Represents the GPIO device controlling VGA colour values

#endif  // __GPIO_INIT_H_
```

Figure 6(c)

```c
#include "gpio_init.h"   // Include the matching header to ensure consistency

// Define global GPIO instances declared as extern in gpio_init.h
XGpio VGA_COLOUR;       // GPIO instance for VGA colour output (12-bit RGB signals)
XGpio VGA_REGION;       // GPIO instance for VGA region selection (9-bit one-hot mask)
XGpio SLIDE_SWITCHES;   // GPIO instance for slide switch input (16 switches on Basys3 board)

// Function: initGpio
// Purpose: Initialize all the GPIO peripherals used in the design
// Returns: XST_SUCCESS if all initializations succeed, otherwise XST_FAILURE
XStatus initGpio(void)
{
    XStatus status;

    // Initialize the SLIDE_SWITCHES device
    // The second parameter (7) is the device ID assigned in the Vivado hardware design
    status = XGpio_Initialize(&SLIDE_SWITCHES, 7);
    if (status != XST_SUCCESS)
    {
        return XST_FAILURE; // If initialization fails, return immediately
    }

    // Initialize the VGA_COLOUR device (device ID = 8 in the hardware design)
    status = XGpio_Initialize(&VGA_COLOUR, 8);
    if (status != XST_SUCCESS)
    {
        return XST_FAILURE;
    }

    // Initialize the VGA_REGION device (device ID = 10 in the hardware design)
    status = XGpio_Initialize(&VGA_REGION, 10);
    if (status != XST_SUCCESS)
    {
        return XST_FAILURE;
    }

    // If all devices initialized successfully, return success
    return XST_SUCCESS;
}
```

Figure 6(d)

- Add the code to Colour_the_world.c shown in Figure 7.

```c
#include <stdio.h>
#include "platform.h"
#include "xil_types.h"      // Added for integer type definitions (u8, u16, u32 etc.)
#include "gpio_init.h"      // Header for custom GPIO initialization functions

// External function declaration for reading GPIO discrete values
u32 XGpio_DiscreteRead(XGpio *InstancePtr, unsigned Channel);

// Global variable 'i' used for a simple counter/toggle logic later in the loop.
// Note: It's declared globally here but used locally in 'main' for a simple check.
u8 i = 0;
```

```c
int main()
{
    // Initialize the Xilinx platform (required for all applications)
    init_platform();
    int status;

    // Initialize the GPIOs (General Purpose Input/Outputs)
    // This sets up the hardware connections for slide switches, LEDs, VGA, etc.
    status = initGpio();
    if (status != XST_SUCCESS) {
        // Print error message if GPIO initialization fails
        print("GPIOs initialization failed!\n\r");
        cleanup_platform();
        return 0;
    }
    else {
        // Print success message
        print("GPIOs initialization success!\n\r");
    }

    // Declare and define required variables
    // slideSwitchIn: Stores the 16-bit value read from the physical slide switches.
    u16 slideSwitchIn = 0x000;
    // region: Stores the 9-bit value to control which VGA region is active.
    u16 region = 0;

    // Main program loop - runs indefinitely
  while (1)
  {
      // Read the current state of the slide switches (Channel 1)
      slideSwitchIn = XGpio_DiscreteRead(&SLIDE_SWITCHES, 1);

      XGpio_DiscreteWrite(&VGA_COLOUR, 1, slideSwitchIn);

      u16 region_selector = slideSwitchIn >> 12;

      // Select VGA region based on the 4 MSBs of the slide switches
      switch (region_selector) {
          case 0:
              region = 0b000000000; // Select none of the 9 regions
              break;
          case 1:
              region = 0b000000001; // Select region 0 (LSB set)
              break;
          case 2:
              region = 0b000000010; // Select region 1
              break;
          case 3:
              region = 0b000000100; // Select region 2
              break;
          case 4:
              region = 0b000001000; // Select region 3
              break;
          case 5:
              region = 0b000010000; // Select region 4
              break;
          case 6:
              region = 0b000100000; // Select region 5
              break;
          case 7:
              region = 0b001000000; // Select region 6
              break;
          case 8:
              region = 0b010000000; // Select region 7
              break;
          case 9:
              region = 0b100000000; // Select region 8 (MSB set)
              break;
          default:
              // If region_selector is 10 or higher, the region variable remains unchanged from its previous value
              break;
      }

      // *** Override/Modification Block ***
      // This conditional block completely overrides the region selection logic above.
      // Since 'i' is never incremented in this code, 'i%2' will always evaluate to 0,
      // and 'region' will always be set to 0b00000000.
      if (i%2)
          region = 0b000000; // This code path is unreachable as 'i' is 0.
      else
          region = 0b00000000; // 'i' is 0, so this sets 'region' to 0 (all regions off).

      // Write the final region selection value to the VGA_REGION register
      XGpio_DiscreteWrite(&VGA_REGION, 1, region);
  }

  // Platform cleanup (never reached due to infinite while loop)
  cleanup_platform();
  return 0;
}
```

Figure 7.

Note: This program demonstrates how to connect the slide switches on the Basys3 FPGA board to the VGA display using the MicroBlaze soft processor. The design relies on two GPIO peripherals: VGA_COLOUR, which stores a 12-bit RGB colour value, and VGA_REGION, which stores a 9-bit one-hot value to select one of the nine regions on the screen. By reading the state of the 16 physical slide switches, the software updates these two registers in real time, allowing the user to paint different regions of the VGA display in different colours simply by toggling switches.

When the program starts, it first initializes the platform and the GPIO devices. If GPIO initialization fails, an error message is printed and the program exits. Once the platform is ready, two variables are declared: slideSwitchIn, which will hold the raw 16-bit value read from the slide switches, and region, which will hold the one-hot binary mask used for region selection. The program then enters an infinite loop so that it constantly monitors the switches and updates the VGA output without interruption.

The lower twelve switches (SW0–SW11) are used to control colour. These switches are read into slideSwitchIn and then written directly to the VGA_COLOUR GPIO register. Although slideSwitchIn contains all 16 switch values, the VGA driver only uses the lowest 12 bits, corresponding to Red, Green, and Blue channels. Each channel is four bits wide, allowing intensity levels from 0 to 15. This arrangement produces a total of 4096 possible colours. For example, setting SW8–SW11 to 1111 and the rest to zero produces bright red; setting SW4–SW7 to 1111 produces bright green; and SW0–SW3 equal to 1111 produces bright blue. Setting all twelve switches high gives white, and setting them all low gives black.

The upper four switches (SW12–SW15) determine which region of the VGA display will be updated. To use them, the program shifts slideSwitchIn right by 12 bits to obtain a 4-bit value called region_selector. This integer ranges from 0 to 15, depending on the combination of the top four switches. A switch-case structure then translates this integer into a one-hot binary mask stored in region. If region_selector is 1, the mask becomes 000000001, which activates Region 0. If it is 2, the mask becomes 000000010, which selects Region 1, and so forth up to value 9, which selects Region 8 with mask 100000000. If the selector is zero, no region is selected. For values above 9, the default case does nothing, leaving the previous region unchanged.

After computing the colour and region values, the program writes them to the corresponding GPIO registers: VGA_COLOUR receives the colour information, while VGA_REGION receives the one-hot mask that selects the active region. Together, these determine which part of the screen will be painted and in what colour. Since the program continuously loops, any change to the physical slide switches is immediately reflected on the VGA monitor.

5. Build and Programm

Once you have completed writing the Colour_the_world.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 8a). 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 8b). This step configures the FPGA with the hardware bitstream containing the MicroBlaze processor and other peripherals. Then, the project should be run, as shown in Figure 8c.

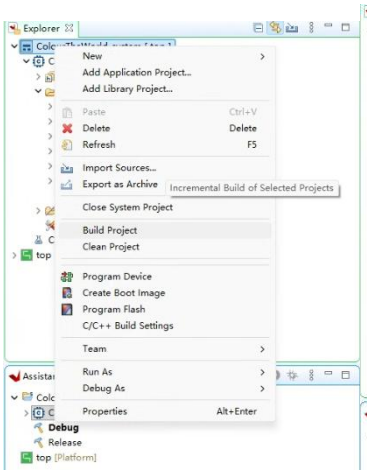

Figure 8(a)

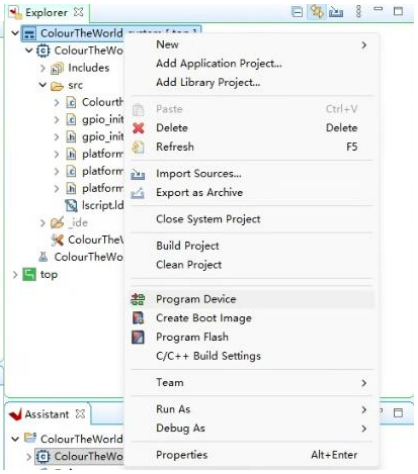

Figure 8(b)

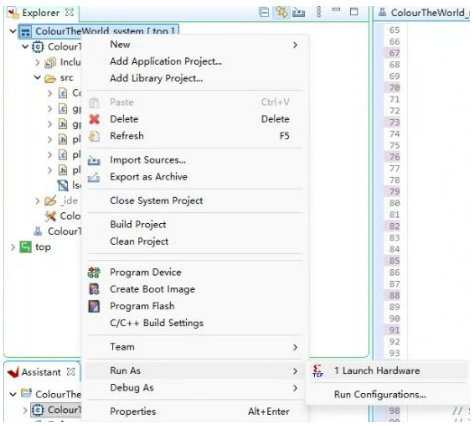

Figure 8(c)

II. Advanced Colour the World

Objectives

- To understand how to use hardware timer interrupts with MicroBlaze.
- To automate the movement of a coloured block on a VGA display.
- To learn how GPIO registers (VGA_COLOUR and VGA_REGION) are controlled by software.
- To demonstrate the use of interrupt service routines (ISR) and global flags for real-time control.

Activity Summary

1. Create a new application named ColourTheWorldAdv
2. Write the application code
3. Import the provided source files (adv_colour.c, adv_colour.h, gpio_init.h, xinterruptES3.c).
4. Build the project and program the Basys3 FPGA.
5. Observe how a red block automatically moves across the screen in a spiral/G-like path, controlled by interrupts.

Specifications

In previous VGA experiments, the display regions and colours were selected manually using slide switches. In this advanced lab, the concept has been expanded by introducing interrupt-driven automatic movement. Instead of using switches to control regions, a hardware timer periodically triggers an interrupt service routine (ISR). The ISR sets a flag that signals the main program loop to advance the coloured block to the next region in a predefined path.

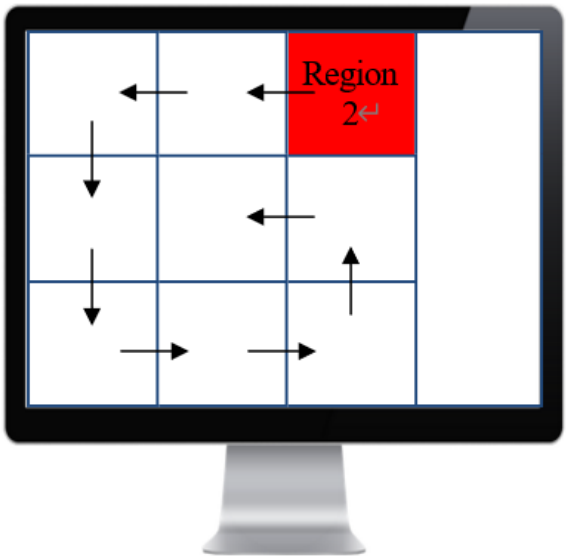

Figure 9.

In this experiment, the program defines a region sequence (a REGION_PATH array) that makes the block follow a spiral or "G" shape across the grid, as shown in Figure 9. The block is coloured red (fixed value 0xF00), and its movement is timed by the hardware timer interrupt. Students will see how software, interrupts, and hardware interact to produce an automated visual effect.

Activities

1. Create a New Application Project

- In the Vitis Welcome Page, click Create Application Project.

- Choose Create from hardware (XSA) and select the exported top.xsa.
- Name the project Colourtheworld_Adv.
- The target processor should be microblaze_0.
- Select the default domain standalone_microblaze_0.
- Choose the Hello World template.
- Click Finish.

2. Create and Rename the Source File

- In Project Explorer, open Colourtheworld_Adv → src.
- Delete or rename helloworld.c as main.c
- Please copy the following five files into the src folder, gpio_init.h , gpio_init.c, xinterruptES3.c, adv_colour.h, adv_colour.c, you can find them in **Lab7 - codes** on learn:

The adv_colour.c and adv_colour.h files together form a small but important module that links the hardware timer interrupt to the main program loop. The main purpose of this module is to provide a safe and reliable way for the interrupt service routine (ISR) to signal the main loop whenever it is time to advance the animation on the VGA screen. Instead of letting the ISR carry out complex display updates, it simply sets a flag, and the main loop checks this flag regularly to decide when to take the next step. This design ensures that the interrupt handler remains short and efficient, while the main program controls the actual logic of updating the display.

In the header file adv_colour.h, three elements are declared. First, there is the global variable move, which is defined as a volatile u8. The keyword volatile is crucial, because it prevents the compiler from making optimisations that might assume the value never changes unexpectedly. Since move is updated by the ISR asynchronously, the main loop must always be able to see the most recent value. This variable is declared as extern in the header, which means that its actual storage is provided elsewhere, specifically in adv_colour.c. Second, the function setUpInterruptSystem() is declared. This function is responsible for configuring the interrupt controller, connecting the hardware timer to the ISR, and enabling CPU exceptions so that interrupts can be serviced correctly. Finally, the prototype for the hwTimerISR() function is provided. This defines the shape of the interrupt handler that will be triggered by the timer, ensuring that other files know how to reference it.

The source file adv_colour.c contains the definition of the move variable. This is the single instance of storage that all other files share. The ISR sets this variable to indicate that a movement step should occur, while the main loop clears it after acting upon it. For example, inside the main loop of the VGA program, we often see a structure such as if (move) { move = 0; … update region … }. This ensures that the block or pattern displayed on the VGA screen advances only once per interrupt cycle. Meanwhile, the interrupt handler increments a small counter, and once a threshold is reached, it sets move = 1 to notify the main loop. This pattern of communication between the ISR and the main loop

is simple, reliable, and avoids the risks of performing long computations inside an interrupt context.

The separation between the header and source files, and between ISR and main loop responsibilities, reflects good design practice. By placing the flag and function prototypes in a dedicated module, the program structure becomes clear and easy to extend. The main program does not need to know about the low-level details of interrupt configuration; it only needs to call setUpInterruptSystem() once at startup and then monitor the move flag. Similarly, the ISR remains very short and efficient, performing only the minimum required action to maintain timing. This modular design also makes the system reusable: the same mechanism can be applied not only to VGA animations but also to other time-based activities such as LED chasers or simple games.

In summary, the adv_colour module demonstrates how interrupts and shared variables can be used to achieve timed, stepwise control in an embedded system. The key idea is that the interrupt does not do the work itself, but instead raises a signal, and the main loop decides when and how to act on that signal. This pattern helps keep the system responsive, reliable, and easy to understand, which is especially important for students learning the fundamentals of real-time embedded programming.

adv_colour.c

```
#include "adv_colour.h"  // brings in the extern declaration and type aliases

/*
 * Define the storage for the global 'move' flag.
 *
 * Why 'volatile'?
 *   The flag is written by an ISR (asynchronous to main) and read in the main loop.
 *   'volatile' guarantees each read observes the most recent write and the compiler
 *   does not cache the value in a register across loop iterations.
 *
 * Typical lifecycle:
 *   - ISR:   move = 1;          // signal "step the animation"
 *   - main:  if (move) { move = 0; ...update state... }
 */
volatile u8 move = 0;
```

adv_colour.h

```c
#ifndef ADV_COLOUR_H_
#define ADV_COLOUR_H_

#include "xil_types.h"   // Xilinx fixed-width types (e.g., u8, u16, u32)

/*
 * 'move' is a global flag that the timer ISR sets when it is time
 * to advance the animation (e.g., move to the next VGA region).
 *
 * - 'extern' means the storage is defined in a .c file (adv_colour.c), not here.
 * - 'volatile' prevents the compiler from optimizing reads/writes away,
 *   because 'move' can change asynchronously inside an interrupt.
 *
 * Typical usage:
 *   In the main loop:
 *      if (move) { move = 0; ...advance state... }
 *   In the ISR:
 *      move = 1;   // signal the main loop to step
 */
extern volatile u8 move;

/*
 * setUpInterruptSystem
 * --------------------
 * Initialize and start the interrupt controller, connect hwTimerISR as the
 * handler for the hardware timer interrupt, and enable CPU exceptions.
 *
 * Return value:
 *   XST_SUCCESS (0) on success, XST_FAILURE (non-zero) on failure.
 *
 * The function is typically implemented in a separate source file that
 * configures XIntc/XScuGic and CPU exception handling (e.g., xinterrupt*.c).
 */
int setUpInterruptSystem(void);

/*
 * hwTimerISR
 * ----------
 * Hardware Timer Interrupt Service Routine.
 * This routine is invoked whenever the timer fires. It should execute quickly:
 *   - update a small counter or flag (e.g., 'move')
 *   - avoid long computations / blocking operations
 *
 * 'CallbackRef' can be used to pass a device/context pointer if needed.
 */
void hwTimerISR(void *CallbackRef);

#endif /* ADV_COLOUR_H_ */
```

- Ensure that adv_colour.h and gpio_init.h are included correctly in main.c.
- Remove the following two lines of code:

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

  print("Successfully ran Hello World application");
- Add the codes in main.c shown in Figure 10.

```c
#include <stdio.h>           // Standard I/O header (used for printing messages to UART/console)
#include "platform.h"        // Xilinx platform initialization and cleanup functions
#include "xil_types.h"       // Xilinx type definitions (e.g., u16 = unsigned 16-bit, u8 = unsigned 8-bit)
#include "gpio_init.h"       // GPIO initialization header (declares initGpio, VGA_COLOUR, VGA_REGION, etc.)
#include "adv_colour.h"      // Advanced colour header (declares interrupt-related variables such as move)
#include "limits.h"          // Provides type limits (optional in this program)

// Global variables
u16 i = 0;                   // Index for REGION_PATH array (tracks which region to move to next)
u16 region = 0;              // Holds the one-hot encoded value of the currently active region
u16 colour = 0xF00;          // Fixed colour value in hexadecimal (0xF00 = full Red, Green=0, Blue=0)
volatile u8 j = 0;           // Counter used inside the timer interrupt to regulate movement speed
                             // Declared volatile because it is modified in an ISR (avoids compiler optimization)

/*
   REGION_PATH: Defines the order of regions to follow on the VGA 3x3 grid.
   The VGA display is divided into 9 regions, indexed as follows:

        0: Top-left      1: Top-center    2: Top-right
        3: Middle-left   4: Center        5: Middle-right
        6: Bottom-left   7: Bottom-center 8: Bottom-right

   The array values represent region indices in the order they should be activated.
   In this version, the chosen sequence is designed to draw a "G" shape on the 3x3 grid.

   Path: Top-right → Top-center → Top-left → Bottom-center → Bottom-left → Middle-right → Middle-left → Center → Bottom-right
*/
static const u8 REGION_PATH[9] = {2, 1, 0, 7, 6, 5, 4, 3, 8};

int main(void)
{
    // Initialize the platform (UART, caches, interrupt controller, etc.)
    init_platform();

    // Step 1: Initialize GPIO devices (slide switches, VGA_COLOUR, VGA_REGION).
    int status = initGpio();
    if (status != XST_SUCCESS) {
        print("GPIOs initialization failed!\n\r");
        cleanup_platform();   // Release resources
        return 0;
    }

    // Step 2: Set up the interrupt system (connects the timer interrupt to hwTimerISR).
    status = setUpInterruptSystem();
    if (status != XST_SUCCESS) {
        print("Interrupt system setup failed!\n\r");
        cleanup_platform();
        return 0;
    }
    while (1)
    {
        // Check if the "move" flag was set by the timer interrupt.
        if (move) {
            move = FALSE;     // Reset the flag so movement happens only once per interrupt cycle

            // Get the next region index from REGION_PATH and convert it to one-hot format.
            // Example: If REGION_PATH[i] = 2 → one-hot = 0b000000100 (activates region 2).
            region = (u16)(1u << REGION_PATH[i]);

            // Increment the index to move to the next step in the path.
            i++;
            // Wrap around when reaching the end of REGION_PATH (loop back to start).
            if (i >= (sizeof(REGION_PATH) / sizeof(REGION_PATH[0]))) i = 0;
        }

        // Write the fixed colour value to VGA_COLOUR (always red in this example).
        XGpio_DiscreteWrite(&VGA_COLOUR, 1, colour);

        // Write the current region mask to VGA_REGION to activate the selected region.
        XGpio_DiscreteWrite(&VGA_REGION, 1, region);
    }

    // This line is normally unreachable (infinite loop above).
    cleanup_platform();
    return 0;
}


/*
   hwTimerISR: Hardware Timer Interrupt Service Routine.
   This function is automatically called each time the timer generates an interrupt.

   Purpose:
   - Count timer "ticks" using the variable j.
   - Only after j reaches a threshold (here, 5) do we set the move flag to TRUE.
   - This effectively slows down the region movement (otherwise it would move too fast).
*/
void hwTimerISR(void *CallbackRef)
{
    if (j >= 5) {   // Threshold for movement speed control
        j = 0;      // Reset counter
        move = TRUE; // Signal main loop to move block to next region
    } else {
        j++;        // Increment counter until threshold is reached
    }
}
```

Figure 10.

Note: This program is an extension of the basic VGA colour control lab. Instead of manually selecting colours and regions using slide switches, this version demonstrates how to use hardware timer interrupts to automatically move a coloured block across different regions of the VGA display. By combining interrupts with GPIO control, students can observe how embedded systems achieve real-time, autonomous behaviour.

At the beginning of the program, the platform is initialised using init_platform(), which sets up the UART, caches, and other board-level features. Then, the function initGpio() is called to initialise the GPIO devices that are connected to the VGA driver. If this step fails, an error message is printed and the program terminates. After the GPIOs are ready, the program calls setUpInterruptSystem() to configure the hardware timer and link its interrupt signal to the hwTimerISR interrupt service routine (ISR). If the interrupt system fails to start, the program again prints an error and exits.

The core of the program runs inside an infinite loop. Instead of responding to user input, the loop constantly checks a global flag called move. This flag is set to TRUE by the interrupt service routine whenever a certain number of timer ticks have occurred. Each time the flag becomes TRUE, the main loop advances the coloured block to the next screen region. The movement path is not random: it is defined by a fixed array called REGION_PATH, which contains a sequence of region indices. The VGA screen is divided into nine regions, arranged in a 3×3 grid (from 0 in the top-left to 8 in the bottom-right). The path array lists these indices in an order that makes the block trace a recognisable shape, in this case a spiral or a letter “G”.

To display the block in the correct place, the program converts each region index into a one-hot binary mask. For example, if the region index is 2, the one-hot mask is 000000100, which activates Region 2. This mask is stored in the variable region and written to the VGA_REGION register, which tells the hardware which part of the screen should be active. Meanwhile, the colour register VGA_COLOUR is written with a fixed value of 0xF00, which corresponds to red (R=15, G=0, B=0). As a result, the moving block always appears red. Students may easily change this value to experiment with different colours.

The timer interrupt routine hwTimerISR controls how quickly the block moves across the screen. Every time the timer triggers an interrupt, the ISR increments a counter variable j. Only when this counter reaches a threshold (here, 5) does the ISR reset the counter and set the move flag to TRUE. This prevents the block from moving too fast and makes its movement visible to the human eye. By adjusting the threshold value in the ISR, students can speed up or slow down the animation.

Overall, this program illustrates three key concepts. First, it shows how interrupts can be used to perform timed actions in an embedded system, freeing the processor from having to constantly check the time. Second, it demonstrates how global flags and volatile variables allow safe communication between an interrupt routine and the main program loop. Finally, it highlights the power of combining hardware registers (for colour and region selection) with software-defined paths (the REGION_PATH array) to create simple but effective animations on the VGA screen.

By modifying the path array, the colour value, or the timer threshold, students can create their own variations of the program. This hands-on exercise helps them understand the interaction between hardware drivers, interrupts, and software logic in real-time embedded applications.

References:

You can go to the lab7 reference in the learn section, or directly copy the link to your browser:

[1] S. Bandyopadhyay, C. A. Murthy, and S. K. Pal, "VGA-classifier: design and applications," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 30, no. 6, pp. 890–895, 2000.

[2] https://www.youtube.com/watch?v=nHphgdE8fT0

[3]https://www.youtube.com/watch?v=AUcl1sjBi5c&list=PLrkc7rkl1Xyiem3OreD12DzmpQk4Zf5Df