***Advanced System on Chip Course***

**LAB 7: AXI-Lite and GPIOs**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc86679751)

[1.1 Learning Outcomes 1](#_Toc86679752)

[2 Details 1](#_Toc86679753)

[2.1 Hardware 1](#_Toc86679754)

[2.2 Memory Map 1](#_Toc86679755)

[3 Procedure 2](#_Toc86679756)

[3.1 Creating custom AXI4 Peripheral 5](#_Toc86679757)

[3.2 Connecting AXI4 Peripheral to ZYNQ Processing System 11](#_Toc86679758)

[3.3 Bitstream generation and Vitis coding 15](#_Toc86679759)

[3.4 Troubleshooting 19](#_Toc86679760)

[3.4.1 Makefile errors in Vitis 19](#_Toc86679761)

# Introduction

## Learning Outcomes

In this module, we will design and implement AXI-Lite peripheral to control General Purpose Input and Output Ports (GPIOs).

At the end of this module, you will be able to:

* Configure and integrate an AXI-Lite peripheral with a Cortex-A9 Processing System.
* Modify a C code to program the Cortex-A9 processor so that it reads the state of switches and control the LEDs.
* Demonstrate a functional simple system that lights up the LEDs based on the status of the switches.

# Details

## Hardware

DDR Controller

GPIO  
Peripheral

LEDs

Switches

Cortex-A9

Interconnect

Programmable Logic

Processing System

Figure 1. Hardware system diagram.

## Memory Map

GPIO PERIPHERAL REGISTERS

|  |  |  |
| --- | --- | --- |
| **Register** | **Base address** | **Size** |
| Data | 0x43C0\_0000 | 4 bytes |
| Direction | 0x43C0\_0004 | 4 bytes |

# Procedure

Create a new workspace such that the path to the workspace **does not contain any spaces**. For Windows this could be a folder in C directory (C:/Workspace).

Open Xilinx Vivado and create a new project in your workspace as shown below:

A picture containing graphical user interface

Description automatically generated

Figure 2. Creating a project

Give it a meaningful name and click next. Select RTL Project and click “Next”. We will not be adding any sources or constraints so click “Next” for the next two pages. In the next page, click the “Boards” tab and search for “Zybo” in the search bar and **click the download button** next to “Zybo Z7-10” if the option is available. Then select it and click “Next” (Don’t click on the hyperlink but click on the empty area next to “Zybo Z7-10”).

Graphical user interface, application

Description automatically generated

Figure 3. Board Selection

Click “Finish”.

In the “Flow Navigator” menu on the left, click “Create Block Design” under “IP Integrator”:

Graphical user interface, text, application, email

Description automatically generated

Figure 4. Creating a new block design

Choose a design name or use the default one and click “Ok”.

Within the empty “Diagram” box on the right-hand side, right-click and select “Add IP”. Enter “Zynq” in the search box and choose “ZYNQ7 Processing System”. Click the “Run Block Automation” option that has now appeared and click “Ok” on the pop-up window.

Graphical user interface, text, application

Description automatically generated

Figure 5. Running Block Automation

Graphical user interface, text, application

Description automatically generated

Figure 6: Run Block Automation default settings

## Creating custom AXI4 Peripheral

Xilinx Vivado provides a simplified way to create an AXI4 peripheral using a wizard. Click “Tools” -> “Create and Package New IP”. Click “Next” and choose the following option:

Graphical user interface, text, application, email

Description automatically generated

Figure 9. Creating AXI4 peripheral

Give the following name to the IP (you may keep the IP location path as provided by default):

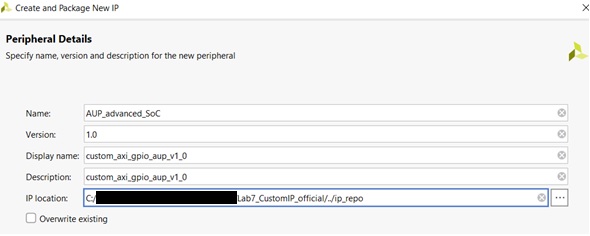


Figure 10. Adding peripheral details

We need a Subordinate interface and four 32-bit registers are enough for our switched and LEDs.

Graphical user interface, application, email

Description automatically generated

Figure 11. Peripheral settings for Subordinate interface

Finally, on the last page, choose “Edit IP” and click “Finish”. In the “Sources” window, you will see two Verilog files in Design Sources:

* A top file, i.e., AUP\_advanced\_SoC\_v1\_0.v
* A file that ends with “S00\_AXI”, i.e., AUP\_advanced\_SoC\_v1\_0\_S00\_AXI.v

They are a basic template for an AXI-Lite peripheral and we can implement our GPIO logic by some modifications. Double-click to open the top-level Verilog file called “AUP\_advanced\_SoC\_v1\_0.v”:

Graphical user interface, text, application, email

Description automatically generated

Figure 12. Editing top-level Verilog file

Add the following lines of code below the comment “// Users to add ports here”:

// Users to add ports here

input wire [3:0] sw,

output wire [3:0] led,

Also add sw and led (underlined in red below) when instantiating the AXI Bus Interface S00\_AXI, so that the code looks like the following:

// Instantiation of Axi Bus Interface S00\_AXI

AUP\_advanced\_SoC\_v1\_0\_S00\_AXI # (

.C\_S\_AXI\_DATA\_WIDTH(C\_S00\_AXI\_DATA\_WIDTH),

.C\_S\_AXI\_ADDR\_WIDTH(C\_S00\_AXI\_ADDR\_WIDTH)

) AUP\_advanced\_SoC\_v1\_0\_S00\_AXI\_inst (

.sw(sw),

.led(led),

.S\_AXI\_ACLK(s00\_axi\_aclk),

…

…

Save the changes in the file (Ctrl+S).

Next, expand and open the other Verilog file (AUP\_advanced\_SoC\_v1\_0\_S00\_AXI.v) shown below:

Graphical user interface, text, application

Description automatically generated

Figure 13: Edit the second Verilog file

Add the following lines of code below the comment “// Users to add ports here”:

// Users to add ports here

input wire [3:0] sw,

output wire [3:0] led,

Scroll down the file and search (Ctrl+F) for “// Address decoding for reading registers”. Then update the code so that it looks like the following (shown in red):

// Address decoding for reading registers

case ( axi\_araddr[ADDR\_LSB+OPT\_MEM\_ADDR\_BITS:ADDR\_LSB] )

2'h0 : reg\_data\_out <= {slv\_reg0[C\_S\_AXI\_DATA\_WIDTH-1:4], sw[3:0]};

Scroll down the file and search (Ctrl+F) for “// // Add user logic here”. Then update the code so that it looks like the following (shown in red):

// Add user logic here

assign led[3:0] = slv\_reg1[3:0];

Save the changes in the file (Ctrl+S).

Next, go to the “Package IP – AUP\_advanced\_SoC” tab, choose the “Customization Parameters” option on the left and click “Merge Changes from Customization Parameters Wizard” to update the IP package with the changes we made in HDL files:

Graphical user interface, text

Description automatically generated

Figure 14. Saving all the changes

Click “Review and Package” on the left and select “Re-Package IP” at the bottom of the page. A prompt will appear – select “Yes” to close the project.

Go to the AXI4 peripheral folder you created earlier <IP location folder>\drivers\<AUP\_advanced\_SoC\_v1\_0>\src and locate a Makefile. In that Makefile, update the contents so that it is the following:

COMPILER**=**

ARCHIVER**=**

CP**=**cp

COMPILER\_FLAGS**=**

EXTRA\_COMPILER\_FLAGS**=**

LIB**=**libxil.a

RELEASEDIR**=**../../../lib

INCLUDEDIR**=**../../../include

INCLUDES**=**-I./. -I${INCLUDEDIR}

INCLUDEFILES**=**\*.h

LIBSOURCES**=** $(wildcard \*.c)

OUTS **=** \*.o

OBJECTS **=** $(addsuffix .o, $(basename $(wildcard \*.c)))

ASSEMBLY\_OBJECTS **=** $(addsuffix .o, $(basename $(wildcard \*.S)))

libs**:**

echo "Compiling axi\_gpio\_asoc..."

$(COMPILER) $(COMPILER\_FLAGS) $(EXTRA\_COMPILER\_FLAGS) $(INCLUDES) $(LIBSOURCES)

$(ARCHIVER) -r ${RELEASEDIR}/${LIB} ${OBJECTS} ${ASSEMBLY\_OBJECTS}

make clean

include**:**

${CP} $(INCLUDEFILES) $(INCLUDEDIR)

clean**:**

rm -rf ${OBJECTS} ${ASSEMBLY\_OBJECTS}

Then, click on “Settings” (under “Project Manager”) in the “Flow Navigator” menu on the left. Expand the “IP” section in the new window that appears and choose the “Repository” option.

Graphical user interface, application

Description automatically generated

Figure 15. Adding IP Repository

Click the “+” option under “IP Repositories” on the right and choose the AXI4 peripheral folder you created earlier (if it hasn’t already appeared) and click “Ok”. Right-click in the empty space of the “Diagram” box again and choose “Add IP”. Type “custom” in the search box and choose “custom\_axi\_gpio\_asoc\_v1\_0” from the options. Click “Run Connection Automation” and then click “OK” to connect the AXI-Lite Subordinate interface on GPIO peripheral to the AXI Manager interface on Arm processor.

Graphical user interface, text, application, email

Description automatically generated

Figure 16: Connect AXI-Lite Subordinate interface (Custom IP) to AXI Manager interface

## Connecting AXI4 Peripheral to ZYNQ Processing System

Right-click on the empty space again in the diagram and choose “Create Ports” (Ctrl+K). Create 2 ports with the following settings and names:

Graphical user interface, application

Description automatically generated

Figure 17. Creating output port

Graphical user interface, application

Description automatically generated

Figure 18. Creating input port

Wire the “sw[3:0]” input to the “sw[3:0]” port of the “axi\_gpio\_asoc\_0” block and the same for the “led[3:0]” output to the equivalent port of the block as shown in the diagram below. This connects them to external ports of the ZYNQ chip:

Diagram, schematic

Description automatically generated

Figure 19. Vivado block diagram

Two IP blocks will be generated automatically. The “Processor System Reset” IP is used to generate reset signals for different peripherals. The “AXI Interconnect” IP here is used to interconnect AXI4-Lite Subordinate and AXI Manager.

Select the “Address Editor” tab next to “Diagram” and change the “64K” to “4K”. Save all your progress.

Graphical user interface, text, application

Description automatically generated

Figure 20. Changing peripheral address settings

Create a new file called “pins.tcl” using a text editor of your choice within your workspace. Add the following two lines of code within it. These tell Vivado to ignore the unspecified I/O Pin warnings.

set\_property SEVERITY {Warning} [get\_drc\_checks NSTD-1]

set\_property SEVERITY {Warning} [get\_drc\_checks UCIO-1]

Right-click the “Generate Bitstream” option in the “Flow Navigator” on the left and select “Bitstream settings”. Click the three dots next to “tcl.pre”:

Graphical user interface, application, table

Description automatically generated

Figure 21. Changing bitstream settings

Select the “New Script” option, click the three dots next to the empty box, choose the “pins.tcl” file you created earlier and click “Ok” on all windows.

Right-click the “Constraints” under the sources tab and select “Add sources”:

Graphical user interface, text, application

Description automatically generated

Figure 22. Adding sources

Select “Add or create constraints” and click “Next”. Select “Create File”, give any name to the file for example pin\_constraints, and click “Finish”:

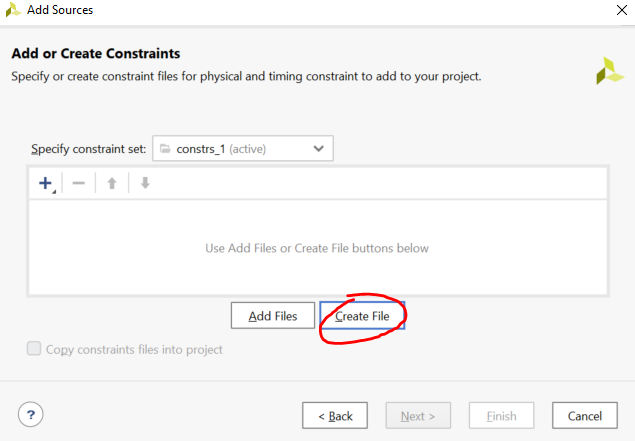


Figure 23. Creating a constraints file

Expand the “Constraints” folder within the “Sources” tab and double-click the file you just created to open it. Add the following constraints from “https://github.com/Digilent/Zybo-Z7-10-Pmod-VGA/blob/master/src/constraints/Zybo-Z7-Master.xdc”, and save the file:

Graphical user interface, text, application, email

Description automatically generated

Figure 24. Editing constraints file

Ensure that your XDC files have the following constraints uncommented:

##Switches

set\_property -dict { PACKAGE\_PIN G15 IOSTANDARD LVCMOS33 } [get\_ports { sw[0] }]; #IO\_L19N\_T3\_VREF\_35 Sch=sw[0]

set\_property -dict { PACKAGE\_PIN P15 IOSTANDARD LVCMOS33 } [get\_ports { sw[1] }]; #IO\_L24P\_T3\_34 Sch=sw[1]

set\_property -dict { PACKAGE\_PIN W13 IOSTANDARD LVCMOS33 } [get\_ports { sw[2] }]; #IO\_L4N\_T0\_34 Sch=sw[2]

set\_property -dict { PACKAGE\_PIN T16 IOSTANDARD LVCMOS33 } [get\_ports { sw[3] }]; #IO\_L9P\_T1\_DQS\_34 Sch=sw[3]

##LEDs

set\_property -dict { PACKAGE\_PIN M14 IOSTANDARD LVCMOS33 } [get\_ports { led[0] }]; #IO\_L23P\_T3\_35 Sch=led[0]

set\_property -dict { PACKAGE\_PIN M15 IOSTANDARD LVCMOS33 } [get\_ports { led[1] }]; #IO\_L23N\_T3\_35 Sch=led[1]

set\_property -dict { PACKAGE\_PIN G14 IOSTANDARD LVCMOS33 } [get\_ports { led[2] }]; #IO\_0\_35 Sch=led[2]

set\_property -dict { PACKAGE\_PIN D18 IOSTANDARD LVCMOS33 } [get\_ports { led[3] }]; #IO\_L3N\_T0\_DQS\_AD1N\_35 Sch=led[3]

## Bitstream generation and Vitis coding

Under the “Sources” tab on the left, expand the “Design Sources” folder, right-click the design1.bd file, choose the “Create HDL Wrapper” and select all default options.

Graphical user interface, text, application, email

Description automatically generated

Figure 25. Creating HDL Wrapper

Save any other changes and click “Generate Bitstream” on the left and click “Ok” for all pop-ups. This process takes some time. Once the process is done, select “Open Hardware Manager”:

Graphical user interface, text, application, email

Description automatically generated

Figure 26. Generating bitstream and uploading to board

Connect the board and make sure the jumper (JP3) above the red LED on the Zybo board is in JTAG mode.

Then, in Vivado, click “Auto Connect” in the Hardware Manager as shown below:

A picture containing diagram

Description automatically generated

Figure 27. Connecting the board

Right-click on the board, select “Program Device” as shown below and click “Program” on the pop-up window.

Graphical user interface, text, application

Description automatically generated

Figure 28. Programming the board

Once the board is programmed, the green LED labeled “LD12” should light up on the board. Click “File” on the main menu bar and select “Export” -> “Export Hardware” and click “Next” on the pop-up window. Choose the following option on the next page:

Graphical user interface, text, application, email

Description automatically generated

Figure 29. Exporting hardware and bitstream file

Choose the “export to” location as the project folder and save the file. Then click “Finish”.

Next, click “Tools” on the main menu bar and select “Launch Vitis IDE”. Choose the same project folder as your workspace. Click “File” -> “New” -> “Application Project”.

Graphical user interface, text, application

Description automatically generated

Figure 30. Creating a new application project

Select the “Create a new platform from hardware (XSA)” tab and click browse to select the XSA file you saved earlier:

Graphical user interface, text, application, email

Description automatically generated

Figure 31. Adding the XSA file

Click next and give a name (e.g. led\_system) to the application project. Click “Next” until you reach the following page and choose “Empty Application(C)” and click “Finish”:

Graphical user interface

Description automatically generated

Figure 32. Creating an empty C Application

Then right-click the “src” folder within the application project you created and add a new file called “main.c”.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 33. Adding a main.c file

Add the following code to the file and save all your changes:

volatile unsigned int value;

int main(){

while(1){

value = \*(unsigned int\*)0x43c00000;

\*(unsigned int\*)0x43c00004 = value;

}

}

From the address tab in Vivado, we know the switches are located at the address “0x43c00000”. All switches together are read as 4 bits with sw[3] being the MSB (most significant bit) and sw[0] as the LSB (least significant bit). If a switch is turned on then it’s corresponding bit will be 1, else it will be 0. We store the positions of the switches into a variable called “value” and write that into the address of the LEDs “0x43c00004” so that the LEDs mirror the On/Off state of the switches.

Right-click the application project in the explorer tab, select “Build Project” and ensure that the build is successful. Then right click again and select “Run As” and then “1 Launch Hardware” to upload everything to the board.

Graphical user interface, application

Description automatically generated

Figure 34. Running the program on the board.

Once this is done, you should be able to toggle the LEDs using the switches.

## Troubleshooting

### Makefile errors in Vitis

Double check if the 3 Makefiles in the project **wrapper** files which you can access from the explorer tab in Vitis:

* If you did not create a custom AXI4 peripheral, then the paths are as follows:

1. hw/drivers/axi\_gpio\_asoc\_v1\_0/src/Makefile
2. ps7\_cortexa9\_0/standalone\_ ps7\_cortexa9\_0/bsp/ ps7\_cortexa9\_0/libsrc/ axi\_gpio\_asoc\_v1\_0/src/Makefile
3. zynq\_fsbl/zynq\_fsbl\_bsp/ps7\_cortexa9\_0/libsrc/axi\_gpio\_asoc\_v1\_0/src/Makefile

* If you created a custom AXI4 peripheral, then the paths are (assuming your IP name is *AUP\_advanced\_SoC\_v1\_0)*:

1. hw/drivers/*<AUP\_advanced\_SoC\_v1\_0>*/src/Makefile
2. ps7\_cortexa9\_0/standalone\_ ps7\_cortexa9\_0/bsp/ ps7\_cortexa9\_0/libsrc/ *< AUP\_advanced\_SoC\_v1\_0>*/src/Makefile
3. zynq\_fsbl/zynq\_fsbl\_bsp/ps7\_cortexa9\_0/libsrc/*<AUP\_advanced\_SoC\_v1\_0>*/src/Makefile

Make the following changes (underlined in red below) so that the Makefiles are as shown below:

**Note: If you are copying and pasting the whole code snippet below into your Makefiles, please ensure that the tab indentations are correct as this may give a Makefile error.**

COMPILER**=**

ARCHIVER**=**

CP**=**cp

COMPILER\_FLAGS**=**

EXTRA\_COMPILER\_FLAGS**=**

LIB**=**libxil.a

RELEASEDIR**=**../../../lib

INCLUDEDIR**=**../../../include

INCLUDES**=**-I./. -I${INCLUDEDIR}

INCLUDEFILES**=**\*.h

LIBSOURCES**=** $(wildcard \*.c)

OUTS **=** \*.o

OBJECTS **=** $(addsuffix .o, $(basename $(wildcard \*.c)))

ASSEMBLY\_OBJECTS **=** $(addsuffix .o, $(basename $(wildcard \*.S)))

libs**:**

echo "Compiling axi\_gpio\_asoc..."

$(COMPILER) $(COMPILER\_FLAGS) $(EXTRA\_COMPILER\_FLAGS) $(INCLUDES) $(LIBSOURCES)

$(ARCHIVER) -r ${RELEASEDIR}/${LIB} ${OBJECTS} ${ASSEMBLY\_OBJECTS}

make clean

include**:**

${CP} $(INCLUDEFILES) $(INCLUDEDIR)

clean**:**

rm -rf ${OBJECTS} ${ASSEMBLY\_OBJECTS}

Save all the modified changes in the Makefile.