

ECE 340

Embedded Systems

Spring 2024

Lab2

**Processor-Based
SoC Design and SW Development**

Introduction

Lab2 is an introduction to the software part of an FPGA-based System On Chip (SoC). It guides you through the process of using Vivado to create a simple ARM Cortex-A9 based processor design targeting the Zedboard development board. You will use Vivado to create the hardware block diagram, and, then, you will use the Vitis IDE (Xilinx Embedded Software Development Environment) to create software applications to verify the hardware functionality and to profile your code.

In step 1a of the lab, you will create a basic ARM Cortex-A9 processor-based platform, and execute an application program that runs on the ARM processor. The software reads in the values of five push buttons and the values of the switches in Zedboard and writes their value to the desktop monitor. For step 1b, you will use the interrupt facilities of the CPU to count the number of times Zedboard buttons have been pushed by the user. For step 2, you will be introduced to code profiling and benchmarking using the hardware timers that are available on the platform. Last but not least, in step 3, you will integrate your FPU design from lab1 to the SoC developed in lab2.

Besides developing software applications, this lab offers an introduction to the ARM-based SoC which is very typical of a modern embedded system. You should take the time to study the user manuals and data sheets of the Zedboard board, ARM processing system, and the AMBA bus standard. At the end of lab2, we will have gained a basic idea of all these tools and their functionality in developing more complex hardware and software platforms.

Also, note that all tools that we are using (Vivado + Vitis) run on the x86-64 platform, but generate executables for the ARM architecture. This is called cross-development (e.g. cross-compilation, cross-linking), and is necessary since the ARM embedded ecosystem in the Zedboard does not have enough resources in terms of CPU speed and memory to support native development.

A brief look at the Zynq Architecture

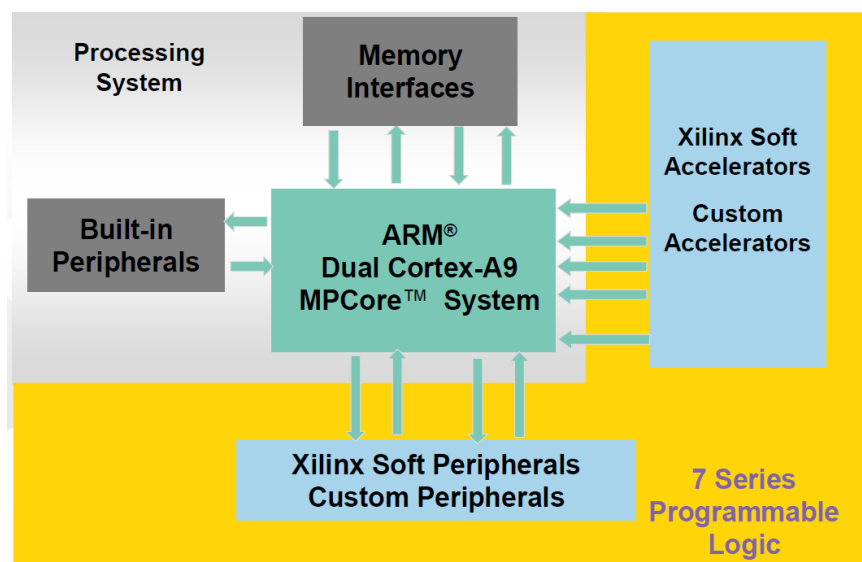


Figure 1. Simplified model of the Zynq architecture.

Zynq as shown in Figure 1 is what Xilinx calls a MultiProcessor System On Chip (MPSoC) architecture. As shown in Figure 1 it consists of two parts: a) the Processing System (PS) formed around a dual-core ARM9 Cortex-A9 processor with vector processing extensions with interfaces to built-in memories and peripherals. b) The Programmable Logic (PL) which contains the FPGA fabric and can be used to generate hardware accelerators, peripherals, etc. In lab1, you used only the PL to implement the FP Adder. PL is ideal for implementing high-speed hardware accelerators using a hardware description language such as Verilog or VHDL (but not only) whereas PS supports software routines, operating systems, and so on. There is a well-defined interface between the two domains, the widely used industry-standard AXI (Advanced eXtensible Interface). We need to emphasize that PS is only programmable in software and you cannot modify its hardware as in PL. Besides the ARM dual-core processors, PS contains internal memory and a multitude of useful peripheral I/O devices to communicate to Zedboard.

Step 1. Basic Hardware/Software Platform

The objective is to design a complete ARM-based SoC using Vivado. Figure 2 represents the design of the first step of lab 2. You will first build the hardware platform and then the software that will run on the ARM processor and make use of the peripherals. Since we will introduce a lot of new concepts in this lab, we will describe the steps to use the Vivado tool in some detail.

a. Hardware Design

The SoC of Figure 2 shows the platform architecture that you will build in the context of step1 of lab2:

- ARM Cortex A9 core (PS). This is a 667 MHz “hard” processor that already exists in the FPGA as the main component of the PS. Zynq is equipped with two ARM processors thus enabling multi-threaded parallelism. Each processor includes vector processing NEON accelerators and FP units. The two cores are supported by standard

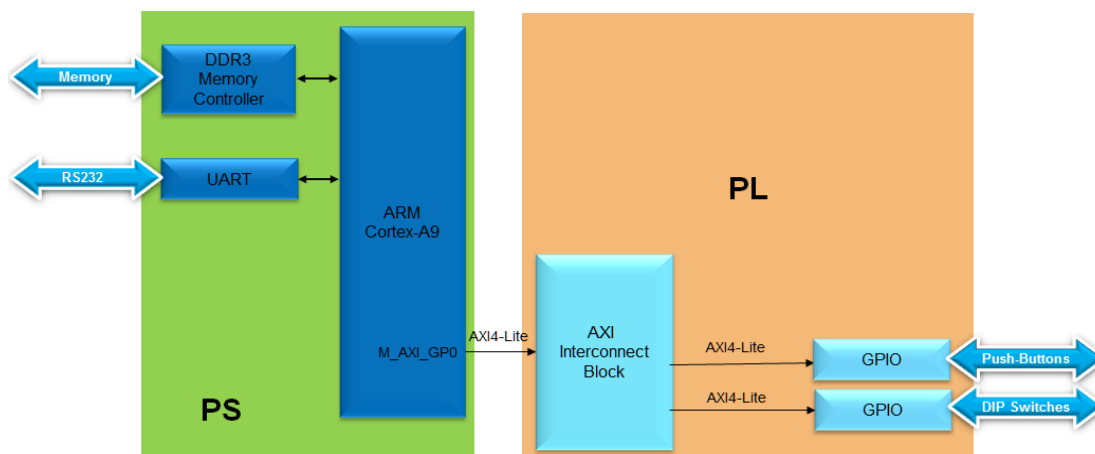
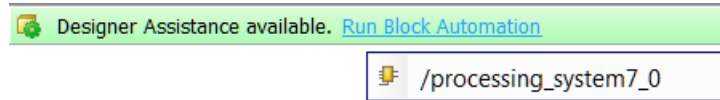


Figure 2. Block diagram of SoC for step 1 of Lab2. The figure emphasizes the partitioning of the Zynq-7000 FPGA in a) the Processing System (PS) which includes the ARM subsystem (hard IP), and b) the Programmable Logic (PL) which includes the FPGA fabric. The PS is only software programmable, i.e. we cannot modify its hardware. Only a small part of the PS is shown in the Figure.



peripherals, also hard IPs, such as the UART to be used in this lab.

- UART for serial communication (PS). This is a connection that enables serial communication protocols such as RS232. It will be used to print out information over to the display.
- DDR3 controller for external DDR3_SDRAM memory (PS). Zedboard includes an external 512 MB DDR3 SDRAM memory. The DDR3 SDRAM memory controller is used to offer shared access of the DDR3 memory to the ARM cores and other components. It has 4 AXI slave ports to provide access to the ARM cores (via their common L2 cache), the PL fabric, and the rest of the peripherals.
- AXI Interconnect block (PL). It is used to multiplex the AXI4-Lite bus out of the ARM core towards the slave GPIO components. All interconnects in Zynq are based on the AMBA bus standard, whose current version is the AXI4.
- Two instances of GPIO peripheral to connect push-buttons and slide switches (PL).

Create the design

Open Vivado and create a new project with name *lab2_simple_arm* under the directory location *lab2_simple_arm* or something similar.

Select *RTL Project* in the *Project Type* form, select *Verilog* as the *Target language* and as the *Simulator language* in the *Add Sources* form.

Click *Next* to skip *Adding Existing IP* and *Add Constraints*

In the *Default Part* form, select *Boards*, and select *Zedboard Zynq Evaluation and Development Kit*.

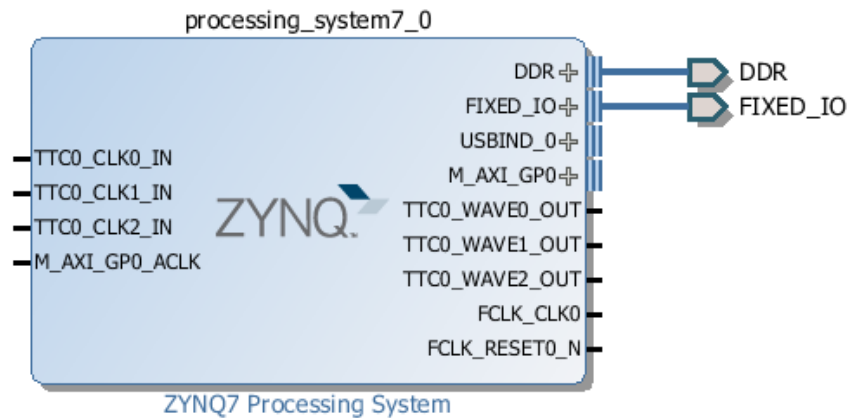
Check the *Project Summary* and click *Finish* to create an empty Vivado project.

Our next step is to use the IP integrator to create a new block design and generate the ARM Cortex-A9 processor-based hardware system.

In the *Flow Navigator* panel, click *Create Block Design* under *IP Integrator* and then give the design name of the block design. IP from the catalog can be added in different ways. In our case, we just click the + icon in the middle of the block diagram panel.

Once the IP Catalog is open, type “zy” into the Search bar, find and double click on *ZYNQ7 Processing System* entry, or click on the entry and hit the Enter key to add it to the design. Notice the message at the top of the Diagram window that *Designer Assistance* available. Click on *Run Block Automation* and select */processing_system7_0* and click *OK* when prompted to run automation. Once Block Automation has been complete, notice that ports have been automatically added for the DDR and Fixed IO, and some additional ports are now visible. A default configuration for the Zynq has been applied which will now be modified.

In the block diagram, double click on the *Zynq* block to open the *Customization* window for the



Zynq processing system. A block diagram of the Zynq-7000 MPSoC ecosystem should now be open, showing various configurable blocks of the Processing System (PS). You should take some time and study this diagram trying to understand its organization. As shown in Figure 2, only UART1 and the DDR3 memory controller are required in this step.

The designer can click on various configurable blocks (highlighted in green) and change the system. Click on one of the peripherals (in green) in the *I/O Peripherals* block, or select the *MIO Configuration* tab on the left to open the configuration form. Expand I/O peripherals if necessary, and deselect all the *I/O peripherals* except *UART 1*. You need to remove *ENET 0*, *USB*, and *SD 0*.

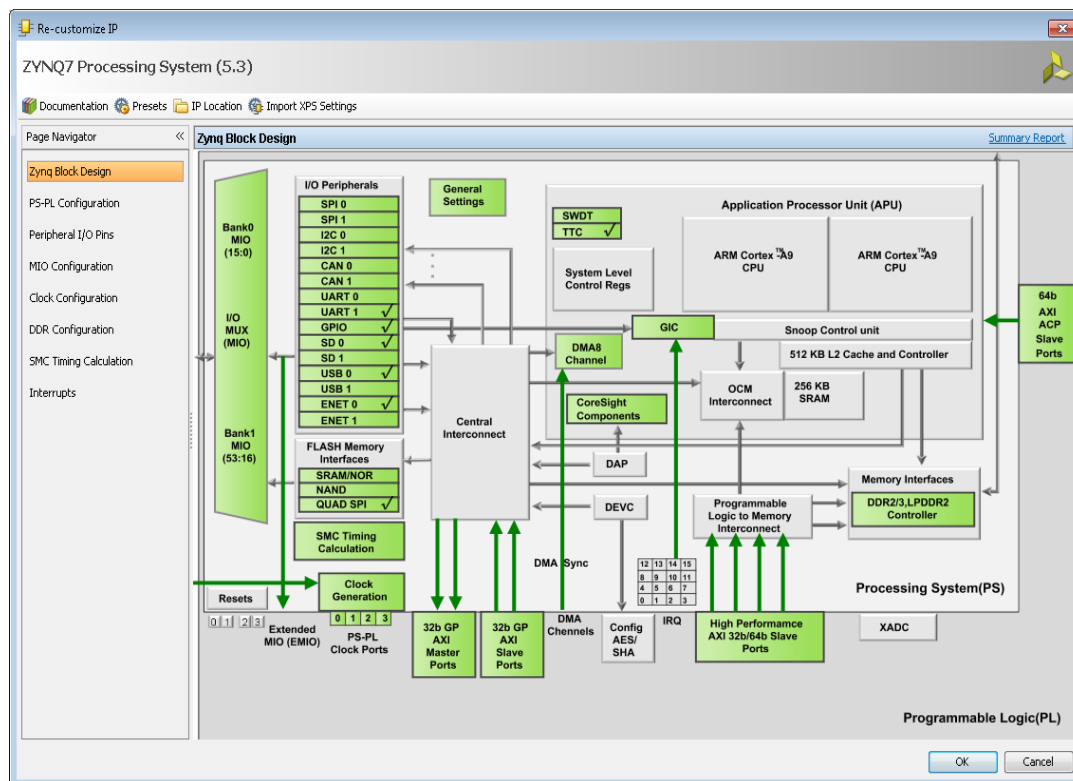
Expand **GPIO** to deselect *GPIO MIO*

Expand **Memory Interfaces** to deselect *Quad SPI Flash*

Expand **Application Processor Unit** to disable *Timer 0*.

Click on the *Clock Configuration*, expand *PL Fabric Clocks*, and observe that *FCLK_CLK0* is enabled with 100 MHz frequency. Click OK.

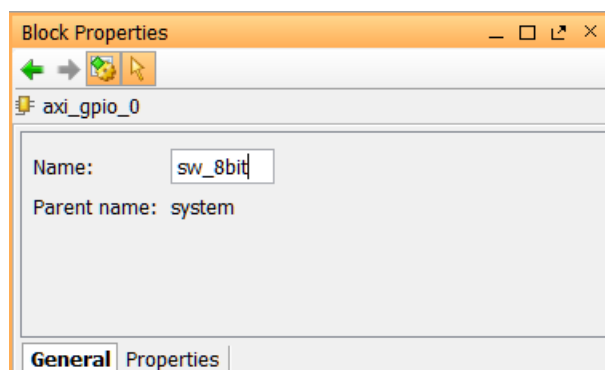
We left the rest of the configuration as is since we want to add two GPIO peripherals in the PL section which will be connected through the GP0 Master AXI interface, using *FCLK_CLK0* as the clock source and *FCLK_RESET0_N* as the reset control signal. At that point, we have concluded the configuration of the PS subsystem. Keep in mind, that deselecting I/O peripherals of the PS subsystem, we only instruct the Vivado integrator that it does not need to include SW drivers for the deselected peripherals. The hardware for these peripherals continues to exist, but it is not going to be used for the specific application.



We now turn our attention to the PL subsystem: we will add two instances of GPIO peripheral as shown in Figure 2. The following process should be followed for the instantiation of any PL peripheral in a design:

- Search for, select and instantiate the peripheral from a list of peripherals (or build the HW of the peripheral yourself).
- Configure the peripheral.
- Connect the peripheral internally to other components and externally to specific pins of the FPGA. The internal connection also includes assigning memory ranges.
- Validate the design.
- All previous steps concern the hardware part of the peripheral. You also need to develop the software drivers to enable the ARM CPUs to monitor and control the functionality of the peripheral.

Click the + icon to add a new IP and search for the *AXI GPIO* peripheral. Double-click the AXI



GPIO to add the core to the design. The core will be added to the design and the block diagram will be updated. Click on the AXI GPIO block to select it, and in the *Board* tab, change the Board Interface to *sws_8bit*. This GPIO peripheral is used to connect the 8 switches of Zedboard to the SoC.

Click the *IP Configuration* tab. Notice the GPIO Width is set to 8. Notice that the peripheral can be configured for two channels, but, since we want to use only one channel without interrupt, leave the *Enable Dual Channel* and *GPIO Supports Interrupts* unchecked. Click *OK* to save and close the customization window

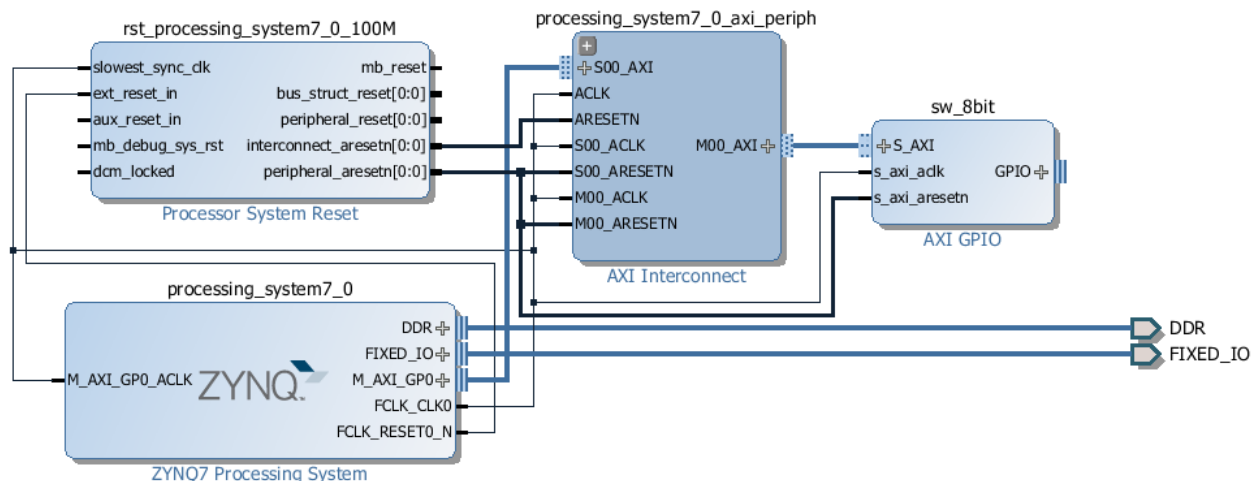
Notice that *Design assistance* is available. Click on *Run Connection Automation*, and select */sw_8bit/S_AXI*. Click *OK* when prompted to automatically connect the master and slave interfaces.

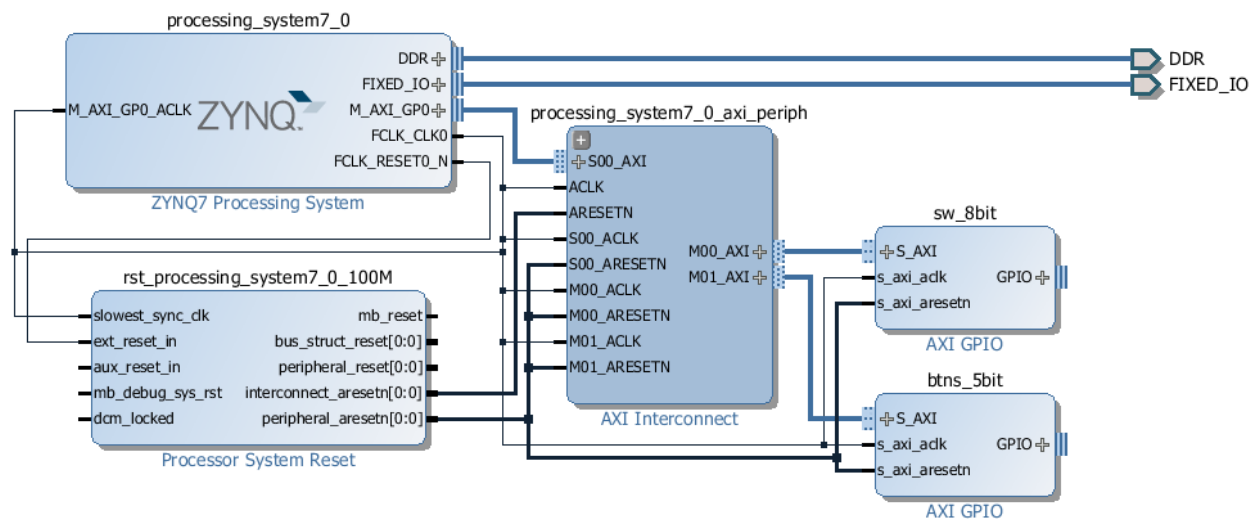
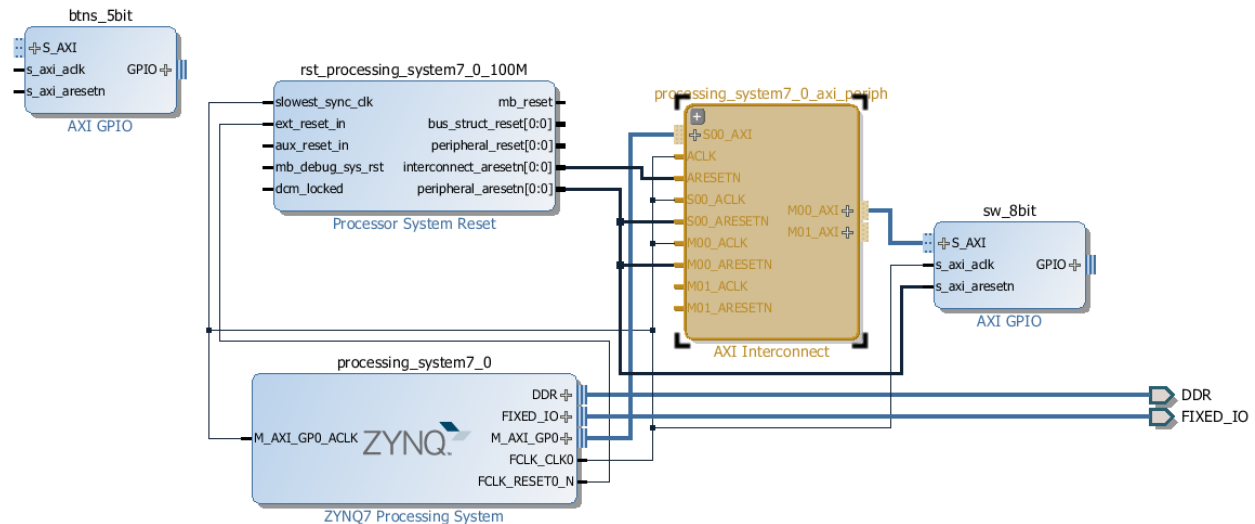
Notice two additional blocks, *Processor Sys Reset*, and *AXI Interconnect* have automatically been added to the design. (The blocks can be dragged to be rearranged, or the design can be redrawn.). The new block diagram is shown on the next page.

We will add another instance of the GPIO peripheral, and using the board flow, configure it to connect to the *btms_5bit*. Change the name of the block to *btms_5bit* (Click on the block to select it, and change the name in the Block Properties view).

At this point, connection automation could be run, or the block could be connected manually. This time the block will be connected manually as follows: double click on the *AXI Interconnect* and change the *Number of Master* Interfaces to 2 and click *OK*.

Click on the *s_axi* port of the new *btms_5bit* block, and drag the pointer towards the AXI Interconnect block. The message *Found 1* interface should appear, and a green tick should appear beside the *M01_AXI* port on the AXI Interconnect indicating this is a valid port to connect to. Drag the pointer to this port and release the mouse button to make the connection.





Similarly, connect the following ports:

btms_5bit **s_axi_aclk** -> Zynq7 Processing System **FCLK_CLK0**

btms_5bit **s_axi_aresetn** -> Proc Sys Reset **peripheral_aresetn**

AXI Interconnect **M01_ACLK** -> Zynq7 Processing System **FCLK_CLK0**

AXI Interconnect **M01_ARESETN** -> Proc Sys Reset **peripheral_aresetn**

The block diagram should look similar to the one in the figure on the page. Note that you can press the *Regenerate Layout* button (cyclic arrow) at the top of the Diagram to better place the components in the palette.

The next step is to assign memory ranges to the peripherals: click on the Address Editor, and expand *processing_system7_0/Data* if necessary. Notice that *sw_8bit* has been automatically assigned an address, but *btms_5bit* has not. Right-click on *btms_5bit* and select *Assign Address*. Note that both peripherals are assigned in the address range of 0x40000000 to 0x7FFFFFFF (GP0 range).

Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 4G)					
sw_8bit	S_AXI	Reg	0x41200000	64K	0x4120FFFF
btms_5bit	S_AXI	Reg	0x41210000	64K	0x4121FFFF

The GPIO peripheral *sw_8bit* has been mapped to the 64 KB range 0x41200000 to 0x4120FFFF (is this physical or virtual memory?) which means that any memory access within that range will access the GPIO peripheral and not the physical memory. In other words, if the application running on the ARM CPU wants to read from the *sw_8bit* GPIO, it just needs to perform a load from that memory range. ARM processor is memory-mapped, meaning that accessing a peripheral is similar to accessing any other memory location.

The final step to conclude the hardware part of our design is to make the two GPIO connections external to the FPGA. The push button and dip switch instances will be connected to the corresponding pins on the Zedboard. This can be done manually or using *Designer Assistance*. The location constraints are automatically applied by the tools as the information for the Zedboard is already known. Normally, one would consult the Zedboard user manual to find this information.

In the Diagram view, notice that *Designer Assistance* is available. This will be ignored for now, and a port will be manually created and connected for the *sw_8bit* instance. *Designer Assistance* will be used to connect the *btms_5bit* peripheral.

Right-Click on the GPIO port of the *sw_8bit* instance and select *Make External* to create the external port. This will create the external port named *gpio* and connect it to the peripheral. Select the *gpio* port, right-click and change the name to *swsbtsdd_8bits* in its properties form. The width of the interface will be automatically determined by the upstream block.

Connection automation will be used to create a port for the *btms_5bit* block. Add the port for the *btms_5bit* component automatically, by clicking on *Run Connection Automation*, and selecting */btms_5bit/GPIO*. In the *Select, Board Interface* drop-down menu, select *btms_5bits*, and click OK to create and connect the external port.

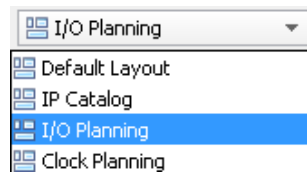
Run Design Validation (*Tools -> Validate Design*) and verify there are no interconnection errors. The design should now look similar to the diagram block. You should keep in mind that all components *btms_5bit* and *sw_8bits* are implemented in PL using the FPGA fabric.

Generate HDL

Once the hardware design is complete, we will create the HDL code for all the PL components: In the *sources* view, Right Click on the block diagram file, *design_1.bd* (or whatever the name is), and select *Create HDL Wrapper* to create the HDL wrapper file. When prompted, select *Let Vivado manage wrapper and auto-update*, click OK. This step will automatically create a Verilog top-level wrapper file *design_1_wrapper.v* that encompasses the design. Open the Verilog file and check it out.

Synthesis, Placement, and Routing

In the Flow Navigator, click *Run Synthesis*. (Click Save when prompted) and when synthesis completes, select *Open Synthesized Design* and click OK. In the shortcut Bar, select *I/O Planning* from the *Layout* dropdown menu:



In the I/O ports tab, expand *BTNs_5bit_tri_i*, and notice pins have already been assigned to this peripheral. The pin information was included in the board support package and automatically assigned when the IP was automatically connected to the port. The *sws_8bits_tri_i* have also been automatically assigned pin locations, along with the other fixed ports in the design as shown in the figure below.

Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref
> FIXED_IO_54576 (59)	INOUT					✓	(Multiple)	(Multiple)*	(Multiple)	(Multiple)
▼ GPIO_1435 (8)	IN					✓	(Multiple)	LVC MOS25*	2.500	
▼ sws_8bits_tri_i (8)	IN					✓	(Multiple)	LVC MOS25*	2.500	
sws_8bits_tri_i[7]	IN	sws_8bits_tri_i_7			M15	✓	34	LVC MOS25*	2.500	
sws_8bits_tri_i[6]	IN	sws_8bits_tri_i_6			H17	✓	35	LVC MOS25*	2.500	
sws_8bits_tri_i[5]	IN	sws_8bits_tri_i_5			H18	✓	35	LVC MOS25*	2.500	
sws_8bits_tri_i[4]	IN	sws_8bits_tri_i_4			H19	✓	35	LVC MOS25*	2.500	
sws_8bits_tri_i[3]	IN	sws_8bits_tri_i_3			F21	✓	35	LVC MOS25*	2.500	
sws_8bits_tri_i[2]	IN	sws_8bits_tri_i_2			H22	✓	35	LVC MOS25*	2.500	

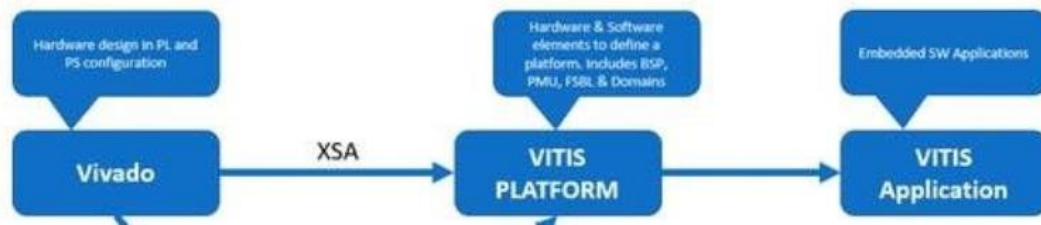
The final step is to generate the bitstream and finalize hardware generation. Click on *Generate Bitstream*, select the *Launch runs on local host* and click *Yes* if prompted to Launch Implementation (Click *Yes* if prompted to save the design.)

Select *Open Implemented Design* option when the bitstream generation process is complete and click *OK*. (Click *Yes* if prompted to close the synthesized design.). At the end of the process, a new *design_1_wrapper.bit* bitstream file is produced under the *<project_name>.runs/impl_1* directory. This is the Xilinx executable that includes all placement and routing information of the design.

b. Software Design

All previous steps developed the hardware components of the targeted platform. The next step is to create the application software and drivers to be executed in one of the ARM processors. The Vitis Unified Software Platform tool provides an environment for creating software platforms and applications targeted for Xilinx embedded processors. Vitis works with hardware designs created with the Vivado toolset.

The typical flow for using Vitis to develop a software application for an embedded system design is shown in the Figure below and is also described next (to better understand the concepts, we will be rather explanatory in this section). Note that, at this point, we have completed the Vivado task at the left and we have generated the XSA file.



1. **Create and Export Hardware Description from Vivado.** First of all, SDK requires that you specify a target hardware platform specification before you can start developing C/C++ application projects. In other words, we need to know what the target system is before we develop software for that system.

In Vivado, select *File*→*Export* →*Export Hardware*

Note that both the Block Design and the implemented Design must be open in the Vivado workspace for this step to complete successfully.

The *Export Hardware for GUI* will be displayed. Click **OK** to export hardware local to Project and make sure that you also include the generated bitstream. The description of the hardware is exported in the Xilinx Support Archive (XSA) proprietary file format that will be then used by the Vitis software platform. The XSA file, in this case, is *design_1_wrapper.xsa*.

2. **Invoke Vitis** from Vivado using *Tools* → *Launch Vitis IDE* and open an existing Workspace directory or create a new one. The workspace directory will contain all the software code and drivers of your project.

Note that you can also invoke Vitis from the Linux prompt (i.e. no need to invoke Vivado first) as follows:

vitis -workspace <Workspace location>

3. **Create New Platform** (e.g. *simple_arm_platform*). A platform includes the hardware configuration (XSA file) and all system software environment settings, i.e. a) Operating System or b) Board Support Package, BSP in case of bare metal development. This step will only include the BSP since we will not involve an OS in Lab2. A platform can contain one or more applications that run on the same platform. The hardware configuration of the project is the *design_1_wrapper.xsa* file (or whatever name you have used in your hardware design). You can also create a new platform project from within the Vitis IDE (File->New->Platform Project).

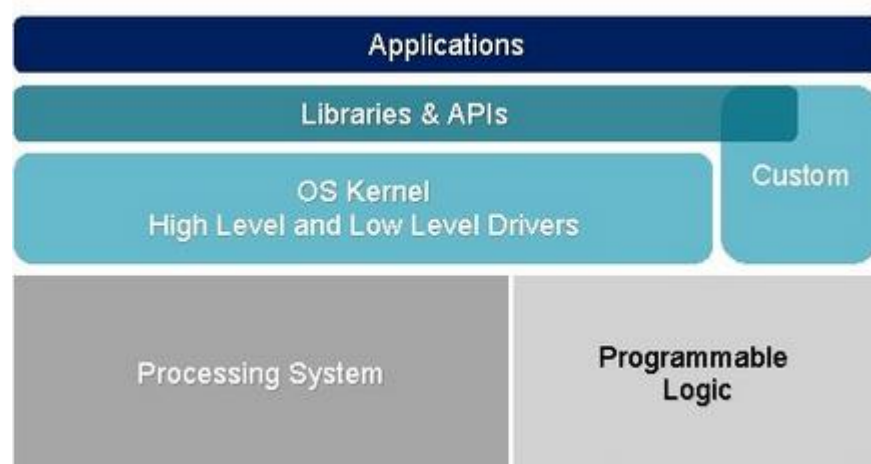
After you specify the name of the platform (e.g. *simple_arm_platform*) you press *Next* to move to the *Platform* window. You specify the XSA hardware configuration file by selecting the “*Create a new platform from hardware (XSA)*” tab and browsing to the directory of the

design_1_wrapper.xsa file. Make sure that the Operating System is *standalone* and the Processor is the *ps7_cortex9_0*. Click *Finish* to enter the Vitis IDE environment.

At the Vitis IDE, **note that the *simple_arm_platform* is outdated (at the Explorer window)**. Right-click on top of the *simple_arm_platform* and press *Build project* to compile the BSP packages. Step 3 is needed only when you create and export new hardware in steps 1 and 2. You do not need to re-create a platform if you do not change the hardware configuration (XSA file) and you keep the same BSP.

The bare-metal (i.e. no Operating System) Board Support Package (BSP) is a collection of libraries and drivers that form the lowest layer of an application. The runtime environment is a simple, semi-hosted, and single-threaded environment that provides basic features, including boot code, cache functions, exception handling, basic file I/O, C library support for memory allocation and other calls, processor hardware access macros, timer functions, and other functions required to support bare-metal applications. Using the hardware platform data and bare-metal BSP, you can develop, debug, and deploy bare-metal applications using the Vitis IDE. Note that the *Workspace* directory now contains the source code of all these libraries of the BSP.

The following Figure shows the software stack in the Xilinx FPGA. The BSP is the Libraries and APIs and is sitting between the User applications and the OS kernel (if the latter exists).



You may also want to wander around the directory structure of the *simple_arm_platform* platform which is automatically generated when we build the BSP and see the XSA and BSP files contained therein. For example:

- *ps7_cortex9_0/standalone_domain/bsp*. It contains the SW drivers for the ARM Cortex A9 CPU, for the hard IP of the PS (e.g. the interrupt controllers), and for all the peripherals we instantiated in the Vivado design (gpios, uart). The file

xparameters.h contains the definition of useful system parameters for the *Xilinx* device driver environment.

- *hw/ps7_init.c*. It is used to define processor, memory, and peripheral configuration. It is executed by the processor immediately after booting. Under the *hw/* directory, you can also find the FPGA bitstream and the XSA hardware description.

Zynq-7000 has support for 32-bit address map (4GB) as shown in Figure 3. The first GB is mapped to the main memory DDR3 and the On Chip Memory (OCM), the next 2 GB to peripherals/accelerators implemented in the PL fabric, and the final GB is used to access PS peripherals. As a Zynq-7000 designer, you should always map your hardware accelerators (such as the two GPIOs) to memory ranges *0x40000000* to *0xDFFFFFFF*.

4. **Create New Application** (e.g. *simple_arm_app*). From the Vitis IDE, click *File* → *New* → *Application Project*. Click *Next* in the first window. In the second window, make sure that you select the correct platform to host the application (*simple_arm_platform*) and press *Next*. In the third window, you have to specify the name of the application (e.g. *simple_arm_app*) and to press *Next*. In the fourth window you select the domain of the application which, in our case is the default selection, and you press *Next* again.

The Vitis IDE provides useful sample applications listed in Select Project Template that you can use to create your project. To create a new project, select the *Empty Application*. You can then add C files to the project. Click *Finish*. Expand *simple_arm_app* in the *Project Explorer*, and right-click on the *src* folder, and select *Import Sources*. Browse to the directory where you have saved the *lab2.c* source code (see figure below). Select *lab2.c* and click *Finish*. The *lab2.c* file is now part of the application.

Start Address	Size	Description
0x0000_0000	1024 MB	DDR DRAM and OCM
0x4000_0000	1024 MB	PL AXI slave port 0
0x8000_0000	1024 MB	PL AXI slave port 1
0xE000_0000	256 MB	IOP devices
0xF000_0000	128 MB	Reserved
0xF800_0000	32 MB	Programmable registers access via AMBA APB
0xFA00_0000	32 MB	Reserved
0xFC00_0000	64 MB — 256 KB	Quad-SPI linear address base (except top 256 KB which is OCM), 64 MB reserved, only 32 MB is currently supported
0xFFFC_0000	256 KB	OCM when mapped to high address space

Figure 3. Zynq-7000 memory map

The library calls are used to access the two GPIO peripherals. They are generated when we produce the BSP package and can be used by the application code if we include the appropriate header files. The *xparameters.h* file package contains definitions of constants such as the *XPAR_SW_8_BIT_DEVICE_ID*. You should consult such header files when you develop your application code.

Take a look at the *src/* directory. Besides the application source code (*lab2.c*), it also contains the *ldscript.ld* file. This script is written in the linker command language. The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. It also specifies the size of the application Stack and the Heap. Note that the default size of these two areas is rather small, and you may want to change it for programs that may allocate a larger chunk of memory in the Stack and the Heap. We will bypass the modification of the linker in this step.

Right-click on the *simple_arm_app* in the Explorer Window and *Build project*. Vitis IDE compiles the application source files using the arm *gcc* compiler (the exact name is *arm-none-eabi-gcc*) and (hopefully) produces the *simple_arm_app.elf* file under *Binaries/*.

```

#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"

//=====

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

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

    XGpio_Initialize(&dip, XPAR_SW_8BIT_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BTNS_5BIT_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    while (1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);
        xil_printf("Push Buttons Status %x\r\n", psb_check);
        dip_check = XGpio_DiscreteRead(&dip, 1);
        xil_printf("DIP Switch Status %x\r\n", dip_check);

        for (i=0; i<9999999; i++);
    }
}

```

5. Test in Hardware.

Now let us turn our attention to the board and the *minicom* utility. This is the Linux equivalent of *Hyperterminal* and will be used as a serial interface between the board and the screen terminal.

Type % **sudo minicom -s** to set up the serial port as a root. Make the following settings:

- Port: /dev/ttyACM0
- Baud rate of 115200 bits/sec
- 8 data bits
- No parity bits
- 1 stop bit

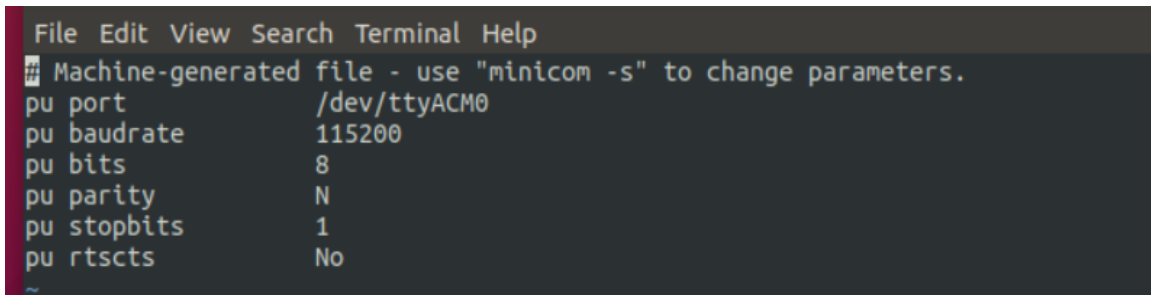
- No software flow control
- No hardware flow control

Save the

settings to the text file `/etc/minicom/minirc.dfl`

Exit minicom with Ctrl+A+Q.

The `minirc.dfl` file should look similar to the one below.



```
File Edit View Search Terminal Help
# Machine-generated file - use "minicom -s" to change parameters.
pu port /dev/ttyACM0
pu baudrate 115200
pu bits 8
pu parity N
pu stopbits 1
pu rtcts No
```

Connect up the Zedboard to the power supply, download cable, and USB cable using the microUSB-UART connection.

Type again **minicom** to start the serial port. You may have to be in supervisor mode to do that (**sudo minicom**)

Program the FPGA using *Xilinx* → *Program Device*.

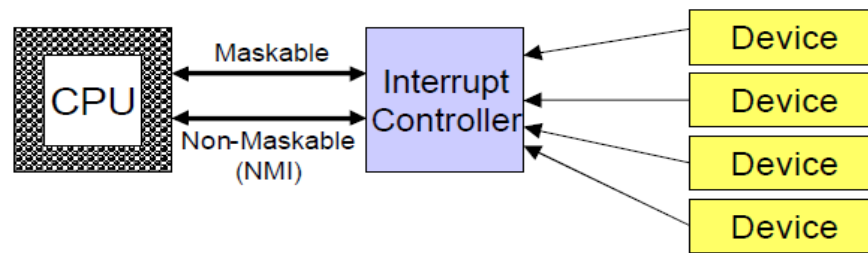
Once the bitstream has been downloaded, the DONE Blue Led is lit to show that the bitstream has been downloaded successfully.

Select `simple_arm_app` in *Project Explorer*, right-click and select *Run As* → *Launch on Hardware* (GDB) to download the application `simple_arm_app.elf` to the DDR3 main memory. The processor `ps7_cortexa9_0` processor is reset, it executes `ps7_init`, and finally `simple_arm_app.elf`.

You should see messages in minicom while C code executes. Explain what happens when the code is compiled with `-O2` optimizations turned on and what happens when the code is compiled with no Optimizations.

6. Software debug using GDB

Vivado provides the usual software debug environment of GDB, the GNU debugger, called Software Debugger by Xilinx. This allows you to execute instructions step-by-step, to set breakpoints, and so on using JTAG-based debugging of one or more ARM processors.



Select *simple_arm_app* in *Project Explorer*, right-click and select *Run As* → *Launch on Hardware* (GDB) to download Go to *Debug As* → *Launch on Hardware (Single Application Debug)*. A new *Console (XSCT Console)* opens up that can be used to debug your software running on the ARM processor. You can execute the program one instruction at a time, monitor variables, registers, memory locations, and so on.

Step 1b. Interrupts

a. Overview

The previous step of lab2 reads the values of the switches and buttons of the Zedboard and prints them out to the screen using the UART peripheral. The program *lab2.c* iterates and actively samples the two GPIO peripherals to read in their values. This technique, called *polling*, is not very efficient because it requires that the CPU spends all its cycles waiting for external events to take place.

While polling is like continuously picking up your phone to check if you have a new message, *interrupts* correspond to a situation where you are only informed of a new message when that message arrives, thus avoiding continuous checks. An interrupt is an event external to the currently executing process that causes a change in the normal flow of instruction execution. It is generated by hardware devices external to the CPU (USBs, sensors, microphone, keyboards, etc.) or by internal devices such as timers, and typically indicates that the device needs service. Unlike polling, interrupts are asynchronous to the current process, i.e. they can happen at any time. Interrupts are very important in embedded systems because such systems contain multiple input devices which can produce data at any time.

The organization of an interrupt-based system is shown in the Figure below. An *interrupt controller* is used to assemble all interrupt sources from external and internal devices and present two interrupt signals to the CPU: one maskable interrupt request signal (IRQ), and one non-maskable interrupt signal (NMI). Non-maskable interrupts cannot be disabled by the user and will always interrupt the control flow of the CPU (e.g. external resets are NMIs). The interrupt controller will prioritize the incoming interrupts and will present to the CPU the interrupt with the highest priority and the corresponding interrupt ID.

Once the CPU receives an interrupt from the interrupt controller, it will save the state of the executing process and will use the interrupt ID to determine which *interrupt handler* (aka

interrupt service routine, ISR) to invoke to service the interrupt. Interrupt handlers are typically fast and light programs that respond to the needs of the interrupt, sometimes with real-time constraints. Once the interrupt is serviced, the executing process is reloaded and starts executing again.

The Zynq7000 PS subsystem is equipped with the Generic Interrupt Controller (GIC) which manages interrupts that are coming from different PL and PS sources to the CPU. It is a centralized resource that is capable of enabling, disabling, masking, and prioritizing interrupt sources sending them to the CPU(s) in a programmed manner.

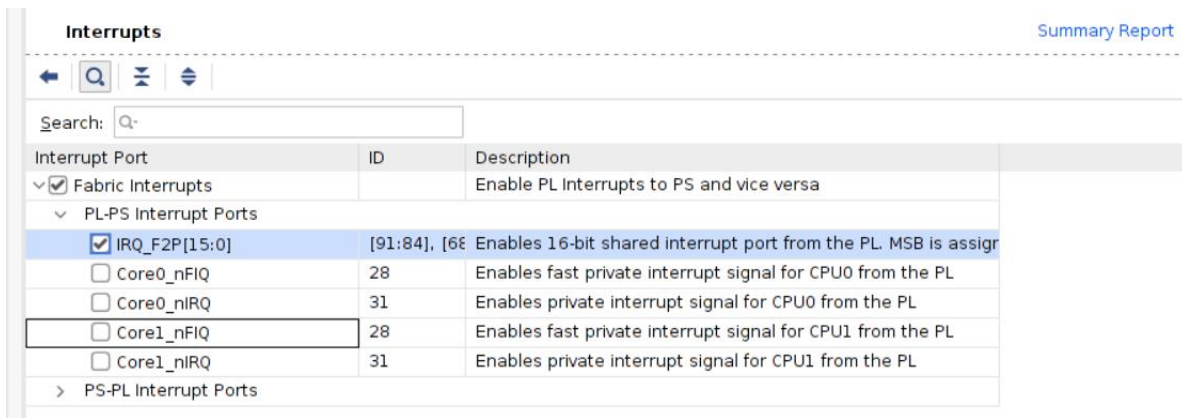
The GIC receives the following interrupt types: a) *CPU Private Peripheral Interrupts (PPIs)* are interrupts from CPUs watchdog timers, private timers, and global timers. b) *Shared Peripheral Interrupts (SPIs)* are a group of 60 interrupts that are generated by PL or PS peripherals. Peripherals like the GPIOs of step 1a which are implemented in the FPGA fabric can send SPI interrupts to the GIC and from there to the CPUs. c) *Software Generated Interrupts (SGIs)* are created by user programs running in the CPU and can interrupt any of the two CPUs.

b. Design of an interrupt-driven system in Zynq 7000

Hardware platform

The hardware design of this system is very similar to that of step 1a. Exit the Vitis IDE environment if you have not done that yet and open the *simple_arm* hardware project using Vivado. Go to *File* → *Project* → *Save As* and save the *simple_arm* project as *interrupt_arm* (or any other name you want) in a different directory so that you can reuse the work you did for step 1a. Open the new project *interrupt_arm* and double click the *btms_5bit* GPIO peripheral to open the *Customize IP* window and enable the interrupts from this peripheral by ticking the box. This will add an additional port (*ip2intc_irpt*) in the block of the *btms_5bit* GPIO.

Now, we need to enable the Zynq PS to accept the interrupt. Double-click the *processing_system7_0* box and select *Interrupts* from the *Page Navigator*. Since we want an interrupt from the Programmable Logic (PL) to the Processing System (PS), expand the menu under *Fabric Interrupts* and tick the shared interrupt port *IRQ_F2P[15:0]* as shown in the Figure. Note that we use one interrupt *IRQ_F2P[0]*, which is assigned to IRQ#61 (LS bit) out of 16 bit shared interrupts from the PL. Click OK. The new port that appeared as input of the *processing_system7_0* box needs to be connected to the interrupt output *ip2intc_irpt* of the GPIO peripheral.



Once you do that, you save and validate your design as we showed in step 1a. The next step is to generate the HDL files for the design: In the *sources* view, Right Click on the block diagram file, *system.bd*, and select *Create HDL Wrapper* to create the HDL wrapper file. When prompted, select *Let Vivado manage wrapper and auto-update*, click OK. Then, you synthesize and generate the bitstream as we explained in detail in step 1a.

Software platform

The interesting part of step 1b is the methodology we use to integrate the interrupt facility into the application program. First, we should export the hardware design (*File* → *Export*).

The *Export Hardware for GUI* will be displayed. Click **OK** to export hardware local to Project and make sure that you also include the generated bitstream. The description of the hardware is exported in the Xilinx Support Archive (XSA) proprietary file format that will be then used by the Vitis software platform. The XSA file, in this case, is *design_interrupt_wrapper.xsa.arm*

Launch Vitis as *Tools* → *Launch Vitis IDE* and make sure to change your workspace to point to the new directory *interrupt_arm* (alternatively, type **vitis -workspace interrupt_arm**). Since we have modified the hardware (the new interrupt signal), we need to create a new Vitis platform (e.g. *interrupt_add_platform*). Make sure to Build the platform (i.e. compile the BSP) before you create the new application. Finally, we need to create a new application (e.g. *interrupt_arm_app*) as we have shown in step 1a. Use the *lab2_interrupt.c* file (which is provided to you)

Then, download the *lab2_interrupt.c* file from the website and build a new BSP package and a new Application Project. Interrupts are supported in a bare-metal system using functions that are automatically generated by the BSP package (drivers). Those drivers are provided in the following header files:

- *xparameters.h* which contains the processor address space and the ID of the peripheral devices.
- *wrapperxscugic.h* which contains definitions of the drivers for configuring and utilizing the interrupt controller.

- *xil_exception.h* which contains definitions of exception functions for the ARM Cortex A9 processor.

In the following description, we will refer to the *lab2_interrupt.c* source code and you should also take the time to study drivers for the GPIO and the interrupt service routines.

The main body of the program in *lab2_interrupt.c* is the empty *while (1);* loop. This is, of course, not typical since in a larger application we would write a regular program to solve a problem and develop an interrupt handler to accommodate an asynchronous interrupt.

An instantiation of the interrupt controller is defined with *XScuGic INTCInst*. The interrupt controller (GIC) handles all incoming interrupt requests passed to PS and performs the actions defined for each interrupt source. The following actions should take place before we can service an interrupt from the GPIO peripheral:

- initialize and configure the GPIO peripheral (function *XGpio_Initialize*).
- Initialize and configure the GIC interrupt controller (functions *XScuGic_LookupConfig* and *XScuGic_CfgInitialize*). The user needs to first call the *XScuGic_LookupConfig* function which returns the Configuration structure pointer which is then passed as a parameter to the *XScuGic_CfgInitialize* function.
- Enable specific interrupts from the GPIO according to the input mask (function *XGpio_InterruptEnable*) and enable global interrupts (function *XGpio_InterruptGlobalEnable*). If the GPIO peripheral has not been built with interrupt capabilities, these functions will return an error.
- Connect the GIC interrupt handler to the hardware interrupt handling logic in the ARM processor (function *Xil_ExceptionRegisterHandler*).
- Connect the GPIO interrupt handler that will be called when an interrupt for the device occurs using function *Xil_ExceptionRegisterHandler*. The interrupt handler (function *BTN_Intr_Handler*) is called every time an interrupt is generated by the GPIO device and performs the specific interrupt processing for the GPIO device.

Exercise for step 1b: For the needs of step 1b, you should study the source code *lab2_interrupt.c* and modify the *BTN_Intr_Handler* to implement the following functionality: every time the user pushes any of the four peripherals buttons of the Zedboard, a counter will increase and its value will be printed on the screen. Pushing the button in the center will reset the counter. Note that every push of a button will generate a GPIO interrupt and should invoke the interrupt handler.

Step 2. Application Profiling and Optimization

a. Overview of Profiling and Benchmarking

Profiling. Code profiling is a very important step in evaluating the quality of the solution of a problem. As engineers, you will often ask yourselves “Where does my code spend most of its time?”, and the answer to that question is the first step towards detecting performance bottlenecks and optimizing your solution. Finding where the bottlenecks are, is what profiling is all about.

The simplest method to measure performance is to use a *timer*, exactly as you do when you want to measure the duration of an everyday event. The theory behind timing an event in the SoC is quite simple. Use a hardware timer in the design, find a piece of code that you suspect takes a lot of time to execute, start the timer, execute the code, stop the timer, and read the value of the timer. The calls to start the timer, stop the timer and read the value back are driver calls that are used to control and monitor the functionality of the timer. The following sequence is a typical (and simplified) example:

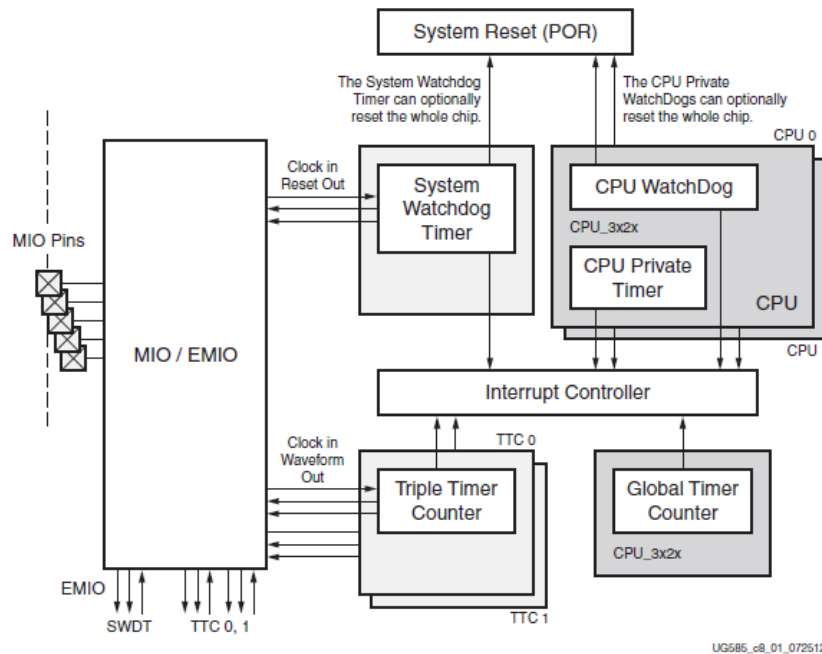
```
start_timer();
<code under surveillance>
stop_timer();
res = read_timer();
xil_printf("The code spent %lld\n" cycles executing the code\n", res);
```

Benchmarking. In computing, a *benchmark* is the act of running a computer program, a set of programs (benchmark suite), or other operations, to assess the relative performance of an object, normally by running several standard tests and trials against it. Benchmarking is usually associated with assessing performance characteristics of computer hardware, for example, the floating point operation performance of a CPU, but there are circumstances when the technique is also applicable to software. Software benchmarks are, for example, run against compilers or database management systems.

Benchmarking in embedded systems has traditionally lagged behind desktop benchmarking. There are simply too many embedded systems to consider and each one is typically optimized to run a small set of applications which makes general-purpose benchmarking meaningless. There is an effort to formally introduce benchmarking in embedded systems as we will examine shortly.

b. Profiling and Benchmarking in Zynq 7000 platform

Each ARM Cortex A9 processor has its own private 32-bit timer and 32-bit watchdog timer. Both processors share a 64-bit global timer. These timers are clocked at $\frac{1}{2}$ of the CPU frequency. On



the system level, there is a 24-bit watchdog timer and two 16-bit triple timer/counters. The system watchdog counter is clocked at $\frac{1}{4}$ to $\frac{1}{6}$ of the CPU frequency or can be clocked by an external signal from the MIO pin or the PL. The two triple timer/counters are always clocked at $\frac{1}{4}$ to $\frac{1}{6}$ of the CPU frequency and are used to count the widths of the signal pulses from the MIO pin or from the PL.

We will use one of the private timers available in the ARM processor to count clock cycles and measure time intervals. Here is an example program that uses the ARM CPU private timer to measure the time it takes to run an application. It is used in the program to read the timer counter register before the program starts and when it has finished.

The BSP package stores information about the timer drivers, as it does for the interrupt drivers of step 2. You should look at the *scutimer_v2_0/src/* directory which contains low-level drivers and example code (*xscutimer_selftest.c*) to be used when writing software to access the private timer of the ARM Cortex-A9 processor.

For this lab, you will develop **your own application** and profile it using the infrastructure we discussed. Make sure that your code does some substantial processing and does not have much I/O (we cannot easily make file I/O at this point since we lack an OS). **Discuss with the instructor** on potential ideas for code profiling. An obvious choice is matrix multiplication, but I am very much open (actually, I prefer) to discuss even more substantial applications.

We provide the *timer_util.ch* files to get you started with code profiling as follows:

```

start_time(); // initializes the private counter of cortex9_0 and triggers it.
<code to be profiled>
stop_time(); // stops the counter
total_time=get_time();
secs_passed = time_in_secs(total_time));

```

Build a new Application Project based on these source files. You can Define compiler flags using the *C/C++ Build Settings* (right-click on the application, first) and then *Symbols*. This may be useful for conditional compilation. You can select optimization under *C/C++ Build Settings*→*Optimizations*. Compile and run your program viewing the output in minicom.

You will also apply optimizations to reduce execution time. The most successful optimizations are the ones concerning loop execution since this is where the program spends most of its time.

Note that the two Cortex-A9 processors have separate 32 KB L1 instruction and data caches, and both caches are 4-way set-associative. The L2 cache is designed as an 8-way set-associative 512 KB cache for dual Cortex-A9 cores.

Step 3. Adding Custom IP to the SoC

This final step of lab2 guides you through the process of creating and adding a custom peripheral to the embedded system. You will add the FP Adder hardware IP of lab1 as an AXI4 peripheral and add ARM software code to control/monitor its functionality on the Zedboard. The design of step 3 has the same functionality as the design of step 4 of lab1, with the exception that you will be using the ARM processor to control and monitor the FP Adder peripheral.

Create the new Hardware IP ¹

In this lab, you will use the *Create and Package New IP* wizard to create a user peripheral from an HDL module, add an instance of the imported peripheral to your baseline design in Vivado, and develop ARM software to control and monitor the functionality of the peripheral. The peripheral is the FP Adder of lab1.

This step comprises several steps involving the creation of an AXI4 custom IP and its addition to the system. Although the change to the hardware is simple, this lab illustrates the integration of a user peripheral through the *Create and Package New IP* wizard.

¹ IP stands for Intellectual Property

Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.



Packaging Options

☐ Package your current project
 Use the project as the source for creating a new IP Definition.

☐ Package a block design from the current project
 Choose a block design as the source for creating a new IP Definition.
 Select a block design:

☐ Package a specified directory
 Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

☒ Create a new AXI4 peripheral
 Create an AXI4 IP, driver, software test application, IP Integrator AXI4 VIP simulation and debug demonstration design.

Open the *interrupt_arm* hardware project (of step 2). Go to *File* → *Project* → *Save As* and save the project as *fpadd_arm* (or a similar name) in a different directory to get a clean new version of our baseline design of step 2.

Open Vivado with the new project (*fpadd_arm*) and invoke *Tools* → *Create and Package IP*. The wizard will take you through a pre-defined set of steps to configure the hardware and software drivers of your new peripheral and to generate template HDL files that you can use to describe the functionality of your peripheral. Click *Next* to continue, select *Create new AXI4 peripheral*, and click *Next* again. In the *Peripheral Details* panel, change the following settings (indicative):

- Name: *fpadd_ip_v1.0*
- Description: *FP Adder AXI IP*

The *fpadd* peripheral IP you will create is an AXI4-Lite compliant slave IP. Click *Next* to go to the *Add Interfaces* panel. Note that the name of the interface is *S00_AXI*, the Interface Type is *Lite* (use AXI4-Lite protocol), the peripheral is an AXI4 *Slave*, the width of the Data Bus of the AXI4 Lite is 32 bits, and there are 4 registers in the peripherals. The registers are memory-mapped and can be accessed by the ARM CPU to control and monitor the functionality of the peripheral. In the *Create Peripheral* page, select *Edit IP* and then click *Finish*.

Click on *IP Catalog* tab in *Flow Navigator* and search for “fpadd”. The *fpadd_ip_v1.0* peripheral will appear in the IP repository to show that the peripheral which you just added is now part of the list of the available IPs. Moreover, Vivado has generated a new *ip_repo* directory that contains all your own IP, an *fpadd_ip_1.0/* subdirectory, and some other subdirectories which include

HDL files, and C source code, and header files for the drivers. These are template files to help the designer integrate the functionality of the peripheral into the rest of the system. For example, *FPADD_IP_Reg_SelfTest(void *baseaddr)*, is a generic Read/Write test to the new peripheral.

Most of the work for designing a new peripheral involves filling up the details for these template files to define both the functionality (HDL code), as well as the software drivers that the peripheral should offer to the CPU programmer. This last point is very important: as designers of a new hardware component, you should provide not only the fully verified hardware but also a set of driver functions to control and monitor the functionality of the component from a CPU software program. These drivers abstract out the complexity of the hardware functionality and the interface and should cover all different capabilities of a hardware component.

Even though the new *fpadd* IP has appeared in the IP catalog, we still need to develop its functionality. Right-click on the *fpadd_ip_v1.0* peripheral and select *Edit in IP Packager*. This will create a **new** Vivado project, named *fpadd_ip_v1.0_project*, where the functionality of the peripheral can be modified in the HDL code and then packaged for future use. In the *Design Sources*, you can see two Verilog source files (all template HDL files are automatically generated in Verilog).

The following two files are generated by Vivado as template files:

- *fpadd_ip_v1_0* is a wrapper file that instantiates the following file.
- *fpadd_ip_v1_0_S00_AXI.v* file. This template file contains the AXI4-Lite interface functionality which handles the interaction between the peripheral and the AXI4-Lite bus. The user can include code to define the functionality of the peripheral. It is a very good idea to study the Verilog code to understand the functionality of the AXI4-Lite bus and its interface to a standard peripheral.

At this point, you must add to the *fpadd_ip_v1.0_project* all the FP Adder Verilog files that you have developed in lab1. You use *Add Files* (like what you have done when adding source files for Vivado projects in lab1). Your source files together with the two template files (which now sit at the top of the FP Adder hierarchy) are now part of the *fpadd_ip_v1.0_project*.

Next, you will modify the Verilog code of the two template files to connect them to your FP Adder design. The two template files include detailed comments on where this extra glue logic needs to be placed (so no further instruction from me, sorry!). Note that this is probably the most time-consuming part of this IP generation step.

Next, click the *Edit Package IP* in the Flow Navigator. Under Packaging Steps, select Ports and Interfaces. A window like the one below should appear. Click the *Merge Changes from Ports and Interfaces* Wizard link. Finally, under Packaging Steps, select *Review and Package*. At the bottom of the Review and Package page, click *Re-Package IP*. The view that opens states that packaging is complete and asks if you would like to close the project. Click *Yes*.

Merge changes from Ports and Interfaces Wizard 1 warning										
Name	Interface Mode	Enablement Dependency	Direction	Driver Value	Size Left	Size Right	Size Left Dependency	Size Right Dependency	Type Name	
> Clock and Reset Signals										
noisy_level			in						std_logic	
leds			out		7	0			std_logic_vector	
an0			out						std_logic	
a0			out						std_logic	
b0			out						std_logic	
c0			out						std_logic	
d0			out						std_logic	
e0			out						std_logic	
f0			out						std_logic	
g0			out						std_logic	
an1			out						std_logic	
a1			out						std_logic	
b1			out						std_logic	
c1			out						std_logic	
d1			out						std_logic	
e1			out						std_logic	
f1			out						std_logic	
g1			out						std_logic	

It is a good idea to *Run Synthesis* and *Save* if prompted to detect any synthesis errors for the IP. Make sure that there are no synthesis errors by checking the Messages Tab. Finally, close the *fpadd_ip_v1.0_project* in Vivado.

Include the new hardware IP to the Baseline Design

We will now add *fpadd_ip* to the *fpadd_arm* hardware design and connect it to the AXI4-Lite interconnect. Then, we will make internal and external port connections and establish the LED and 7Seg ports as external FPGA pins. This is identical to what we have done in step 1a of lab2.

Make sure to select the **Address Editor** tab and assign an address to the *fpadd_ip*. I have changed the address to the range `[0x4122_0000, 0x4122_FFFF]` but feel free to pick up your own address range. Make sure to validate your new design using *Tools* → *Validate Design* and to update the wrapper file to include the new IP (*Create HDL Wrapper*).

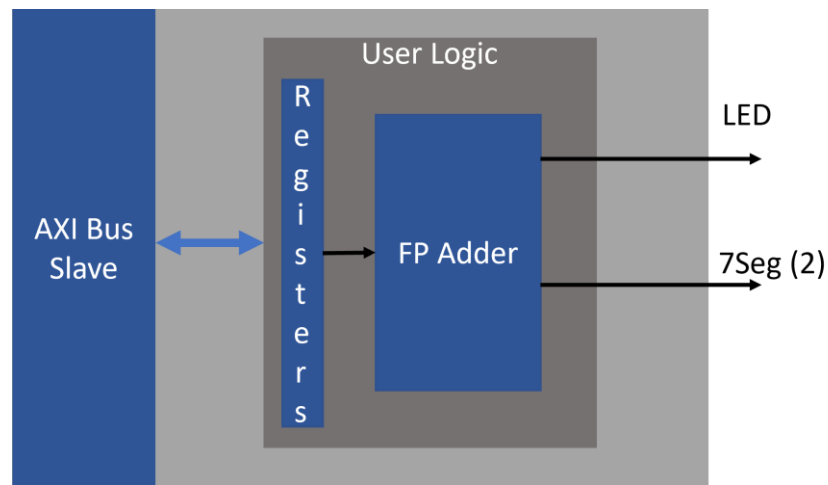
Finally, you need to add the design constraints to specify the FPGA I/O pins which will be assigned to the I/O module ports (LEDs, and PMOD for the two 7seg displays). In the *Flow Navigator* view, navigate to *RTL Analysis* and select *Open Elaborated Design*. Click OK. After the elaborated design opens, click the *I/O Ports window* and expand All ports. You now have to manually place all I/O constraints similarly to what you have done in lab1 (shown in Figure below).

The screenshot shows the Xilinx IDE interface. The top menu bar includes 'Window', 'Layout', 'View', 'Help', and a search bar with 'I/O Ports' entered. The main workspace is divided into several panes: 'Sources' (showing 'design_1_wrapper' and 'Nets (167)'), 'Properties' (empty), 'Project Summary' (showing 'Schematic'), and 'I/O Ports' (selected). The 'I/O Ports' pane displays a table of I/O components.

Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	B:
> FIXED_IO_54576 (59)	INOUT					✓	(I
> sws_8bits_54576 (8)	IN					✓	(I
▼ leds_0 (8)	OUT					✓	
leds_0[7]	OUT				U14	✓	
leds_0[6]	OUT				U19	✓	
leds_0[5]	OUT				W22	✓	
leds_0[4]	OUT				V22	✓	
leds_0[3]	OUT				U21	✓	
leds_0[2]	OUT				U22	✓	
leds_0[1]	OUT				T21	✓	

Click on the *Generate Bitstream* in the Flow Navigator to run the synthesis, implementation, and bitstream generation processes. (Click *Save* if prompted.). Click *Yes* to run the synthesis process again as the design has changed. When the build completes, click *OK* if prompted to open the implemented design.

The next figure shows how the FP Adder IP is connected to the rest of the system. The four registers (we actually need only one register, but the extra overhead of using four is negligible) are used as interfaces to transfer control/data from/to the ARM CPU to/from the FP Adder peripheral. Note that the LEDs and the two 7Seg need to be connected to the external ports of our design.



Application code using the Custom IP

The final step is to create the software platform and the application code that a) monitors the buttons/switches of the Zedboard, to b) control the functionality of the FP Adder. The FP Adder is triggered when ARM writes the value 1 to the first register. Your output should appear in the LEDs and the 7Seg displays similarly to lab1.