CE 435

Embedded Systems

Spring 2019

Lab2

Processor-Based

SoC Design and SW Development

Introduction

Lab2 is an introduction to the software part of an FPGA-based System On Chip. It guides you through the process of using Vivado and IP Integrator to create a simple ARM Cortex-A9 based processor design targeting the ZedBoard development board. You will use IP Integrator to create the hardware block diagram. Then, you will use the SDK (Software Development Kit) to create software applications to verify the hardware functionality and to profile your code.

In the first step 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 the second step of lab2, you will use the interrupt facilities of the CPU to count the number of times Zedboard buttons have been pushed by the user. For the third step, you will be introduced to code profiling and benchmarking using the hardware timers that are available in the platform.

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 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.

A brief look at the Zynq Architecture

Zynq (Figure 1) is what Xilinx calls an All Programmable System On Chip (APSoC) architecture. As shown in Figure 1 is 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)

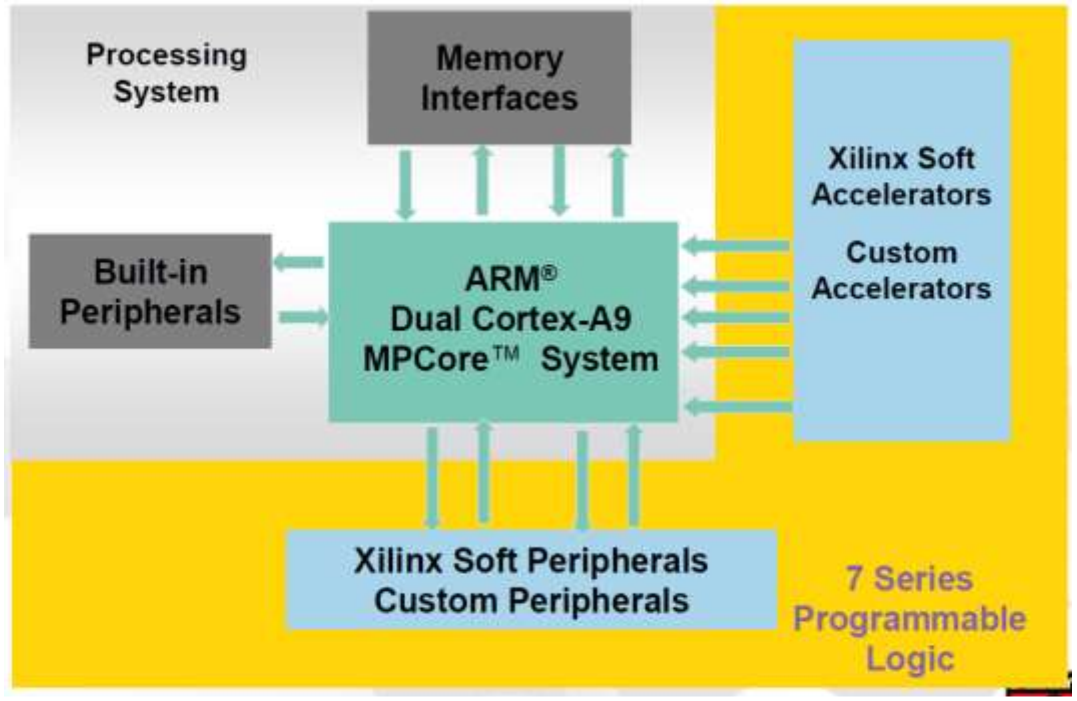


Figure 1. Simplified model of the Zynq architecture.

which contains the FPGA fabric and can be used to generate hardware accelerators, peripherals, etc. In lab1, you only used PL to implement the Gray counter. 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 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

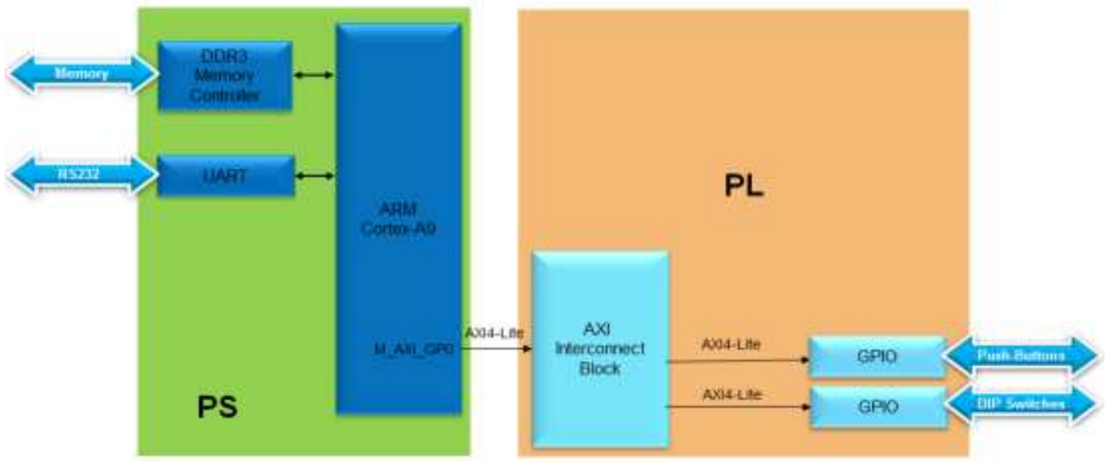


Figure 2. Block diagram of SoC for step 1 of Lab2. The figure emphasizes the partitioning of the Zynq-7000 FPGA in a) the Processing System (PS) which includes the ARM subsystem (hard IP), and b) the Programmable Logic (PL) which includes the FPGA fabric. The PS is only software programmable, i.e. we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

to the screen.

- 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 offer 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*

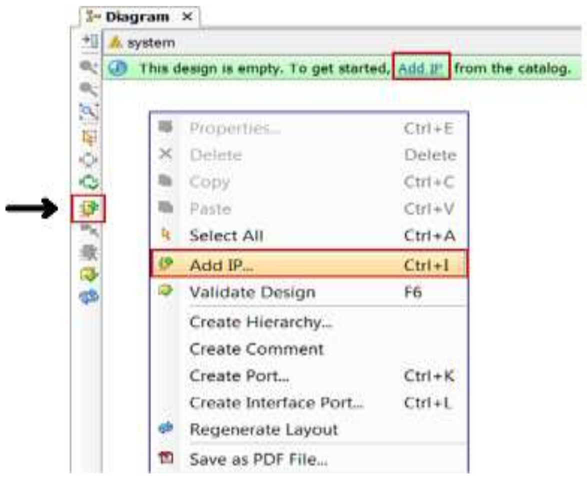
In the *Default Part* form, select *Boards*, and select *Zedboard Zynq Evaluation and Development Kit*.

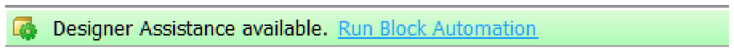
Check the *Project Summary* and click *Finish* to create an empty Vivado project.

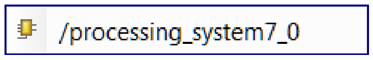
Our next step is to use the IP integrator to create a new block design and generate the ARM Cortex-A9 processor based hardware system.

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. Click on *Add IP* in the message at the top of the *Diagram* panel, or click the *Add IP icon* in the block diagram side bar.

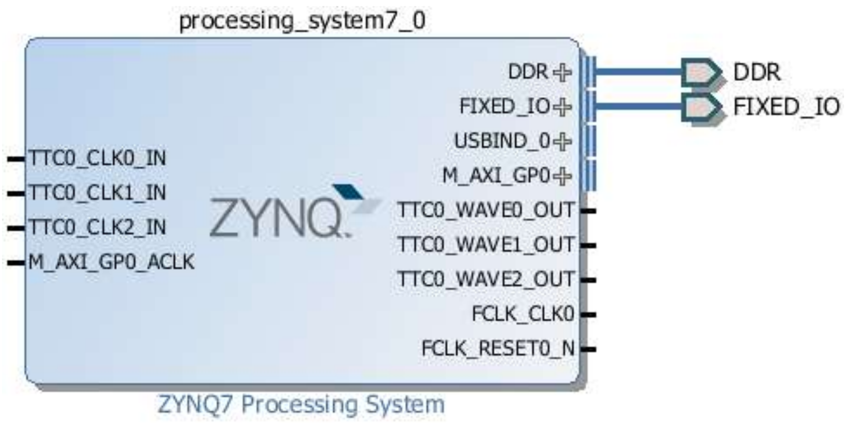
Once the IP Catalog is open, type "zy" into the Search bar, find and double click *on ZYNQ7 Processing System* entry, or click on the entry and hit the Enter key to add it to the design. Notice the message at the top of the Diagram window that *Designer Assistance* available. Click on *Run Block Automation* and select */processing_system7_0* and click *OK* when prompted to run automation. Once Block Automation has been complete, notice that ports


This design is empty. To get started, Add IP from the catalog.





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 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.*

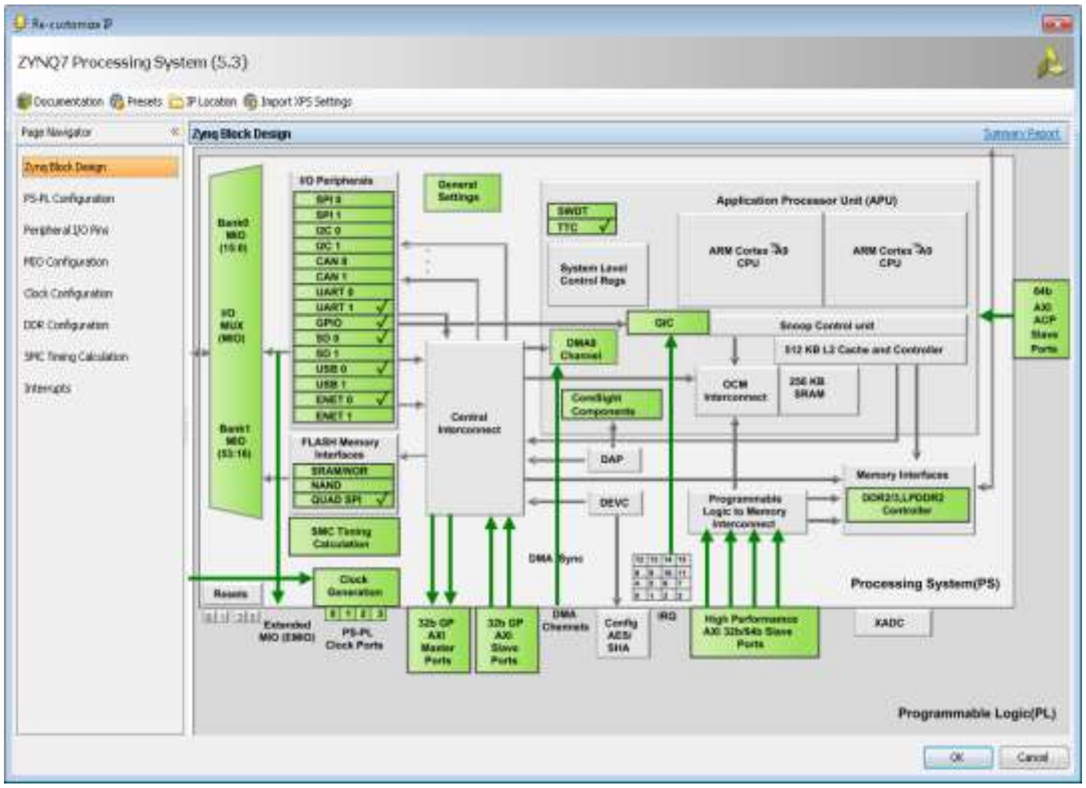
Expand **GPIO** to deselect *GPIO MIO*

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.

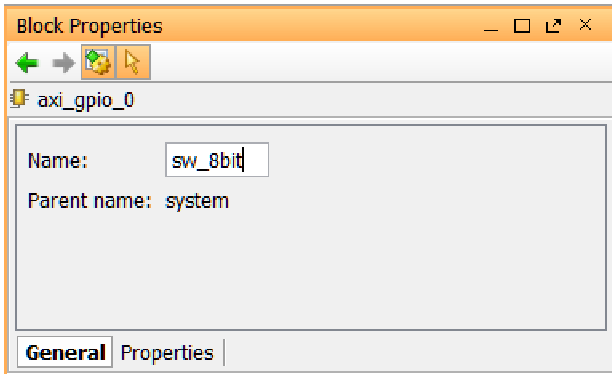
We left the rest of the configuration as is since we want to add two GPIO peripherals in the PL section which will be connected through the GP0 Master AXI interface, using FCLK_CLK0 as the clock source and FCLK_RESET0_N as the reset control signal. At that point, we have concluded the configuration of 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, it just 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. Internal connection also includes assigning memory ranges.
- All previous steps concern the hardware part of the peripheral. You also need to develop



axi_gpio_0

Name: sw_8bit

Parent name: system

the software drivers to enable the ARM CPUs to monitor and control the functionality of the peripheral.

- Validate the design

Click the *Add IP* icon 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 properties tab, change the name to *sw_8bit*. This GPIO peripheral is used to connect the 8 switches of Zedboard to the SoC.

Double click on the AXI GPIO block to open the customization window. As the Zedboard was selected during the project creation, and a board support package is available for the Zedboard, Vivado has knowledge of available resources on the board. Click on *Generate Board Based IO Constraints*, and under *Board Interface*, for *GPIO*, click on *Custom* to view the dropdown menu options, and select *sws_8bit.* This instructs Vivado to connect GPIO nets to the appropriate FPGA pins.

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 *GPIO Supports Interrupts* and *Enable Channel 2* unchecked. Click *OK* to save and close the customization window

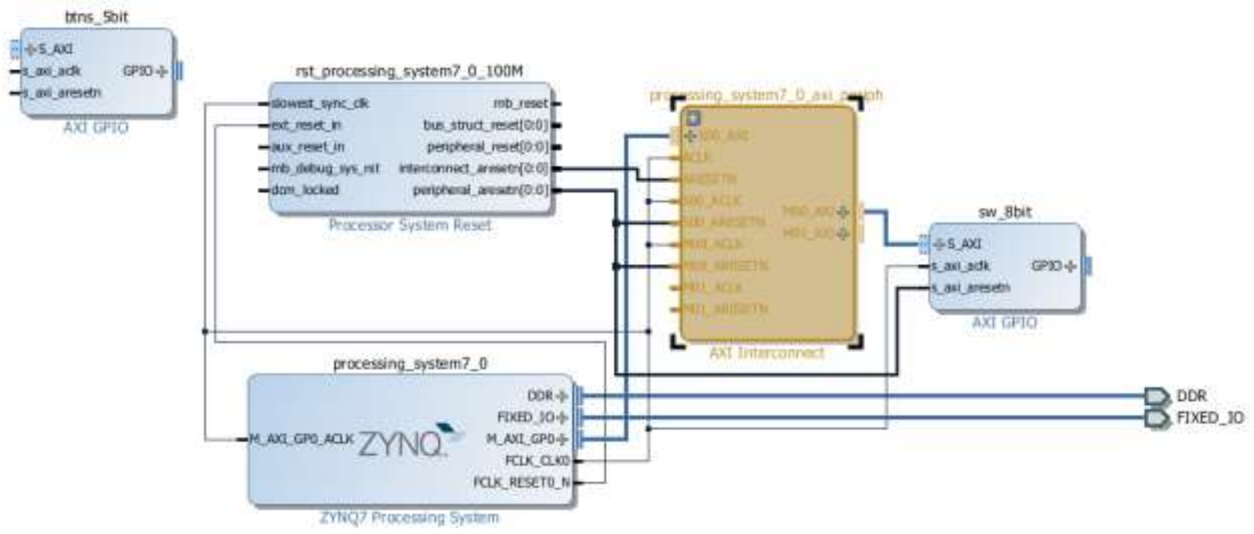
Notice that *Design assistance* is available. Click on *Run Connection Automation*, and select */sw_8bit/S_AXI*. Click *OK* when prompted to automatically connect the master and slave interfaces.

Notice two additional blocks, *Proc 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 in the next page.

We will add another instance of the *GPIO* peripheral (Add IP), and using the board flow, configure it to connect to the *btns_5bit*. Change the name of the block to *btns_5bit* (Click on the block to select it, and change the name in the 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.*

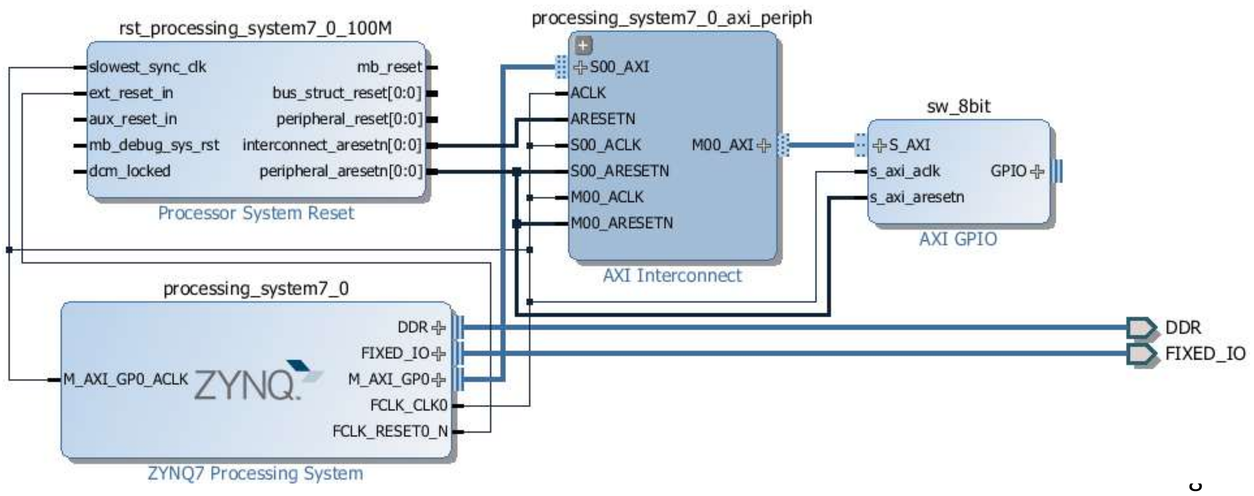
Click on the *s_axi* port of the new *btns_5bit* block, and drag the pointer towards the AXI Interconnect block. The message *Found 1* interface should appear, and a green tick should appear beside the *M01_AXI* port on the AXI Interconnect indicating this is a valid port to connect to. Drag the pointer to this port and release the mouse button to make the connection.



In a similar way, connect the following ports:

btns_5bit **s_axi_aclk** -> *Zynq7 Processing System* **FCLK_CLK0**
btns_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 figure at the end of the page.



The next step is to assign memory ranges to the peripherals: click on the Address Editor, and expand *processing_system7_0 > Data > Unmapped Slaves* if necessary. Notice that *sw_8bit* has been automatically assigned an address, but *btns_5bit* has not. Right click on *btns_5bit* and select Assign Address. Note that both peripherals are assigned in the address range of 0x40000000 to 0x7FFFFFFF (GP0 range).

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.

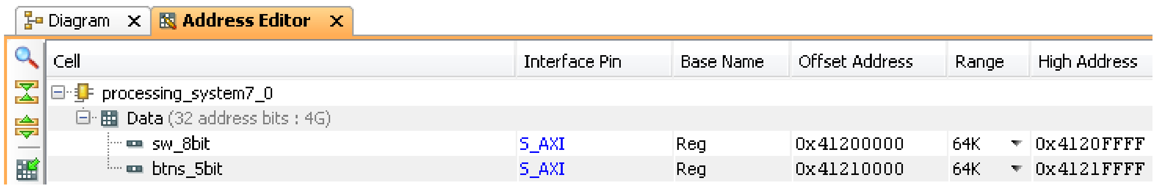


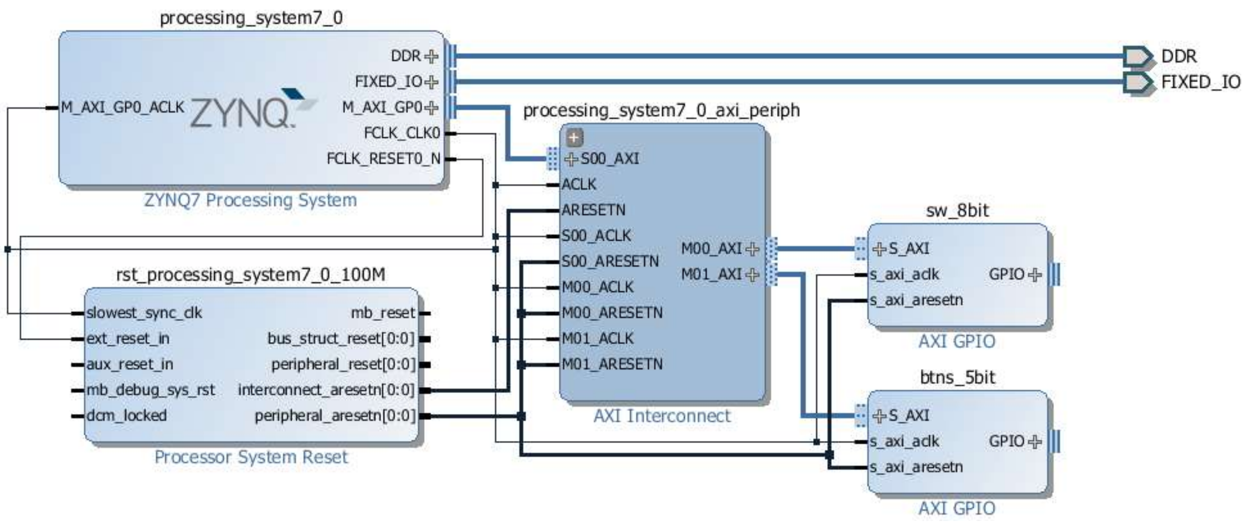
Diagram | Address Editor

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
btns_5bit	S_AXI	Reg	0x41210000	64K	0x4121FFFF

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 *btns_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 *sw_8bit*s 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 btns_5bit block. Add the port for the *btns_5bit* component automatically, by clicking on *Run Connection Automation*, and selecting */btns_5bit/GPIO*. In the *Select Board Interface* drop down menu, select *btns_5bits*, and click OK to create and connect the external port.

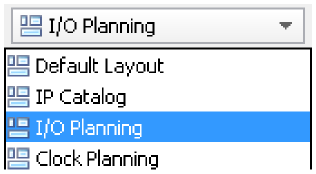
Run Design Validation (*Tools* -> *Validate Design*) and verify there are no interconnection errors. The design should now look similar to the diagram block. You should keep in mind that all components *btns_5bit* and *sw_8bits* are implemented in PL using FPGA fabric.

Generate HDL

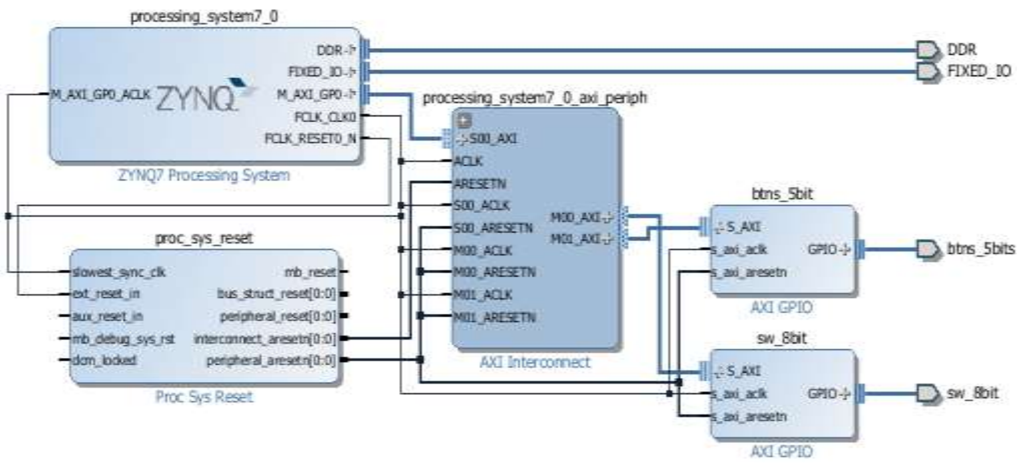
Once the hardware design is complete, we will create the HDL code for all the PL components: In the *sources* view, Right Click on the block diagram file, *system.bd*, and select *Create HDL Wrapper* to create the HDL wrapper file. When prompted, select *Let Vivado manage wrapper and auto-update*, click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

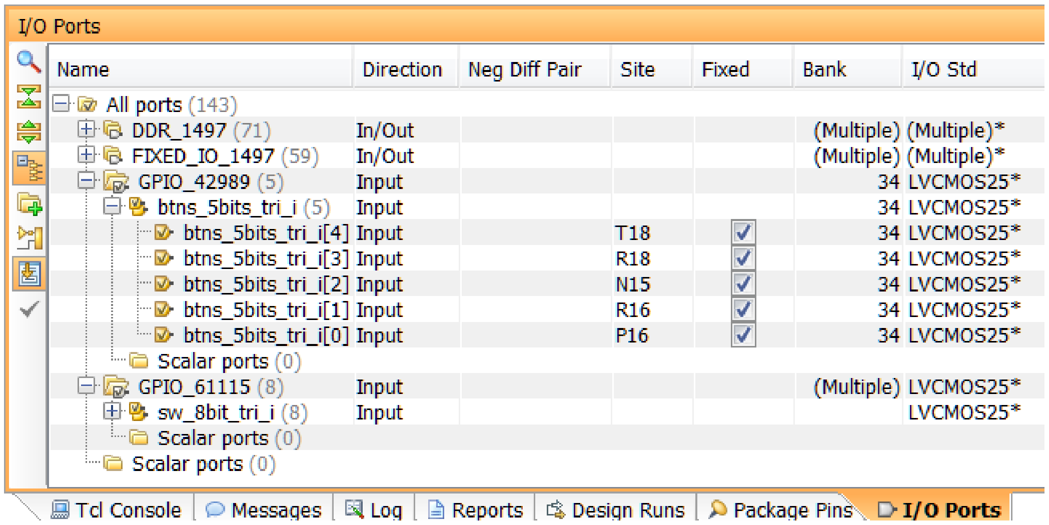
Synthesis, Placement and Routing

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 assigning when the IP was automatically connected to the port. The *sw_8bit_tri_i* have also been automatically assigned pin locations, along with the other fixed ports in the design as shown in the figure.




I/O Ports

Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std
All ports (143)						
DDR_1497 (71)	In/Out				(Multiple)	(Multiple)*
FIXED_IO_1497 (59)	In/Out				(Multiple)	(Multiple)*
GPIO_42989 (5)	Input				34	LVCMOS25*
btns_5bits_tri_i (5)	Input				34	LVCMOS25*
btns_5bits_tri_i[4]	Input		T18	✓	34	LVCMOS25*
btns_5bits_tri_i[3]	Input		R18	✓	34	LVCMOS25*
btns_5bits_tri_i[2]	Input		N15	✓	34	LVCMOS25*
btns_5bits_tri_i[1]	Input		R16	✓	34	LVCMOS25*
btns_5bits_tri_i[0]	Input		P16	✓	34	LVCMOS25*
Scalar ports (0)						
GPIO_61115 (8)	Input				(Multiple)	LVCMOS25*
sw_8bit_tri_i (8)	Input					LVCMOS25*
Scalar ports (0)						
Scalar ports (0)						

The final step is to generate the bistream and finalize hardware generation. Click on *Generate Bitstream*, 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 *system_wrapper.bit* file is produced under the *<project_name>.runs/impl_1* directory.

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 Software Development Kit (SDK) provides an environment for creating software platforms and applications targeted for Xilinx embedded processors. SDK works with hardware designs created with the Vivado toolset. SDK is based on the Eclipse open source standard.

The typical flow for using SDK to develop a software application for an embedded system design is described next (to better understand the concepts, we will be rather explanatory in this section):

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*

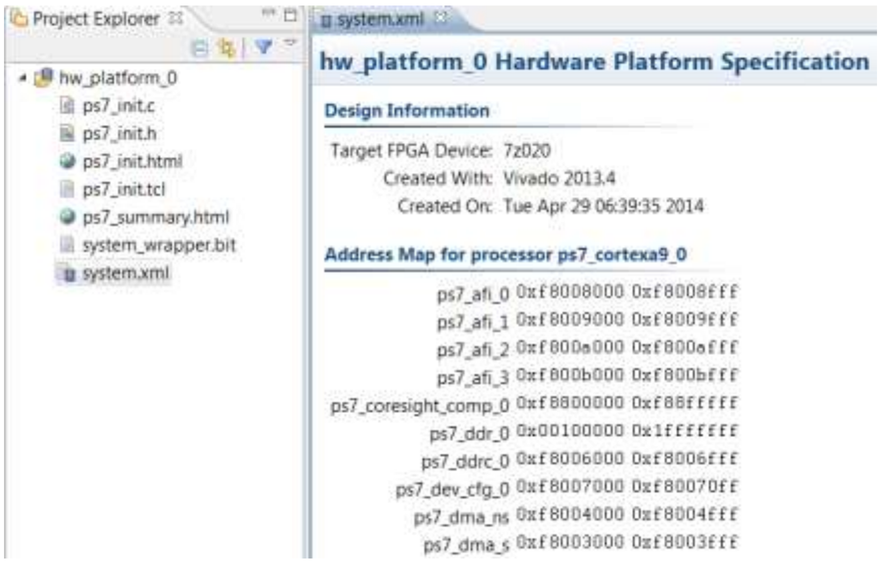
   Note that both the IPI Block Design and the Implemented Design must be open in the Vivado workspace for this step to complete successfully.

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.

Invoke SDK from Vivado using *File → Launch SDK* (or from the Linux prompt using *xsdk*) and open an existing SDK Workspace, or create a new one. When creating a new workspace, use a Hardware Platform that was generated from XPS. A default directory *<project_name>.sdk* is created to store the exported architecture. This directory contains (among others) the description of the FPGA SoC in the form of an *HDF* file (*system_wrapper.hdf*).

Basic information about the hardware configuration of the project can be found in the *system_wrapper.hdf* file, along with the Address maps for the PS systems, and driver information as shown below. The *ps7_init.c* file is also automatically created and is used to define processor, memory and peripheral configuration. It is executed by the processor immediately after booting.

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*.



hw_platform_0 Hardware Platform Specification

Design Information

Target FPGA Device: 7z020
Created With: Vivado 2013.4
Created On: Tue Apr 29 06:39:35 2014

Address Map for processor ps7_cortexa9_0

ps7_afi_0	0xf8008000	0xf8008fff
ps7_afi_1	0xf8009000	0xf8009fff
ps7_afi_2	0xf800a000	0xf800afff
ps7_afi_3	0xf800b000	0xf800bfff
ps7_coresight_comp_0	0xf8800000	0xf88fffff
ps7_ddr_0	0x00100000	0x1fffffff
ps7_ddrc_0	0xf8006000	0xf8006fff
ps7_dev_cfg_0	0xf8007000	0xf80070ff
ps7_dma_ns	0xf8004000	0xf8004fff
ps7_dma_s	0xf8003000	0xf8003fff

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

2. **Create libraries and support software from SDK**. Based on the exported hardware of the previous step, create board support package (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 SDK. Step 2 is needed only when you export new hardware in the previous section.

   From SDK, click *File → New → Xilinx Board Support Package*. And in Project name field, type a name for the board support package if you wish to override the default value, such as *lab2_simple_bsp_0*. Then, select the location for the board support project files. From the *Board Support Package OS* drop-down list, select the type of board support package to create (standalone) and click *Finish*.

   SDK creates a new directory structure (e.g. *lab2_simple_bsp_0),* invokes *libgen* tool to create drivers for the processor and the peripherals of the platform, and also to generate the file *system.mss*. The *system.mss* (Microprocessor Software Specification) is used as an input file to the Library Generator (*Libgen*). The MSS file contains directives for customizing OSs, libraries, and drivers.

3. **Create application project from SDK**. Click *File → New → Application Project*.

In the Project Name field, type a name for the new project (e.g. *lab2_simple*). Make sure that the *ps7_cortexa9_0* (i.e. the first ARM processor) has been selected. Use the existing BSP support package (e.g. *lab2_simple_bsp_0*) and click *Next.*

SDK 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 *lab2_simple* in the *Project Explorer*, and right-click on the *src* folder, and select *Import.*

Expand *General category* and double-click on *File System*. Browse to the directory where you have saved the *lab2.c* source code (see figure below). Select *lab2.c* and click *Finish*.

```
#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++);
    }

}
```

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. Take also a look at the *xparameters.h file* in the *lab2_simple_bsp_0* package which 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.

4. **Develop your software application in SDK**. Optionally, use SDK linker script generation tools to modify your application's memory map. Bypass the linker step in lab2. Then, compile and link your project in SDK by pressing *Project-> Build All*. SDK calls the *gcc* compiler and linker for ARM using a Makefile, and generates the executable (e.g. *lab2_simple.elf)* file which should be downloaded in the DDR3 memory.

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 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 file minirc.dfl (in directory /etc)

Exit minicom

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

Connect up the Zedboard to power supply, download cable, and USB cable using the microUSB-UART connection.
Program the FPGA using *Xilinx Tools → Program FPGA.*
Once the bitstream has been downloaded, the DONE Blue Led is lit to show that the bitstream has been downloaded successfully.

Select *lab2_simple* in *Project Explorer*, right-click and select *Run As → Launch on Hardware* (GDB) to download the application *lab2_simple.elf* to the DDR3 main memory. The processor *ps7_cortexa9_0* processor is reset, then executes *ps7_init*, and finally *lab2_simple.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. Explain your answer.

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. Go to *Run→Debug Configurations* to select the C/C++ Application to be used for debugging (by default under the *Debug* directory), and press *Debug*. A new *Debug perspective* opens up that can be used to debug your software running on ARM processor. You can execute the program one instruction at a time, monitor variables, registers, memory locations and so on.

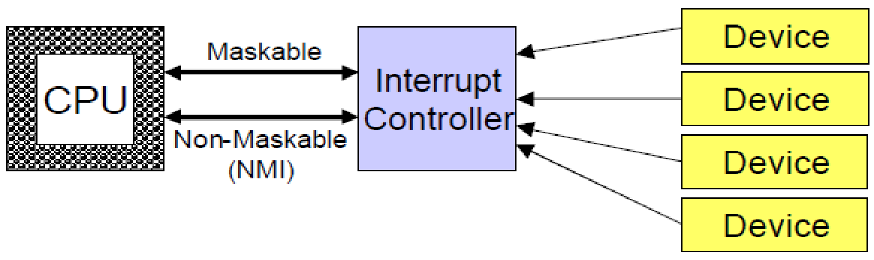
Step 2 – Interrupts

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 if 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. Interrupt is an event external to the currently executing process that causes a change in the normal flow of instruction execution. It is usually generated by hardware devices external to the CPU (USBs, sensors, microphone, keyboards, etc.) and typically indicates that the device needs service. Unlike polling, interrupts are asynchronous to the current process, i.e. it can happen at any time. Interrupts are very important in embedded systems because such systems contain multiple input device 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 which 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 1 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 1. Save the *lab2_simple* project as *lab2_interrupt* in a different directory so that you can reuse the work you did for step 1. You may also delete the *sdk* directory, since we will create a new software project in step 2. Open the new project *lab2_interrupt* and double click the *btns_5bit* GPIO peripheral to open the *Customize IP* window. Select the *IP Configuration* tab and enable the interrupts from this peripheral by clicking the box. This will add an additional port (*ip2intc_irpt*) in the block of the *btns_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 *Project Navigator*. Since we want an interrupt form the Programmable Logic (PL) to the Processing System (PS), expand the menu under *Fabric Interrupts* and tick the shared interrupt port as shown in the Figure below. This enables the interrupts from the PL to the PS. 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 1. 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 you generate the bitstream as we explained in detail in step 1.

Software platform

The interesting part of step 2 is the methodology we use to integrate the interrupt facility to the application program. First, we should export the hardware design (*File→Export*). 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:

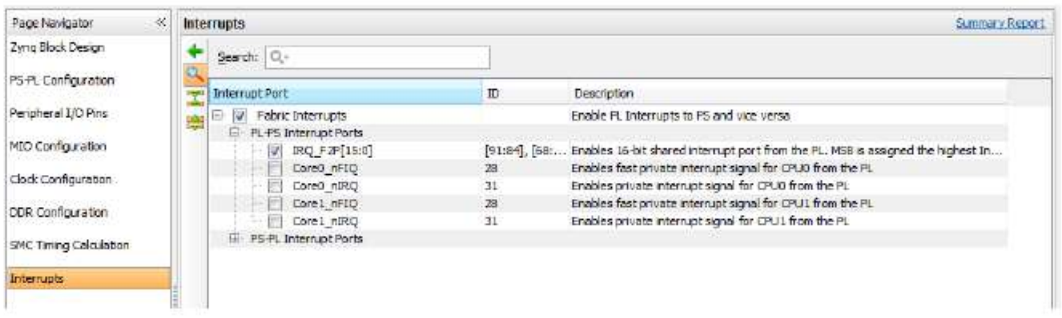
- *xparameters.h* which contains the processor address space and the ID of the peripheral devices.
- *xscugic.h* which contains definitions of the drivers for configuring and utilizing the interrupt controller.
- *xil_exception.h* which contains definitions of exception functions for the ARM Cortex A9 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 are able to 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



Interrupts

Interrupt Port	ID	Description
☑ Fabric Interrupts		Enable PL Interrupts to PS and vice versa
PL-PS Interrupt Ports		
☑ IRQ_F2P[15:0]	[91:84], [68:...	Enables 16-bit shared interrupt port from the PL. MSB is assigned the highest In...
☐ Core0_nFIQ	28	Enables fast private interrupt signal for CPU0 from the PL
☐ Core0_nIRQ	31	Enables private interrupt signal for CPU0 from the PL
☐ Core1_nFIQ	28	Enables fast private interrupt signal for CPU1 from the PL
☐ Core1_nIRQ	31	Enables private interrupt signal for CPU1 from the PL
PS-PL Interrupt Ports		

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 error.
- Connect the GIC interrupt controller 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 2: For the needs of step 2, 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 3 – Application Profiling

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, in order to assess the relative performance of an object, normally by running a number of 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 ½ of the CPU frequency (Figure 3). 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 ¼ to 1/6 of the CPU frequency or can be clocked by an external signal from MIO pin or from the PL. The two triple timer/counters are always clocked at ¼ to 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 in 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 take a look at the *ps7_cortexa9_0/libsrc/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.

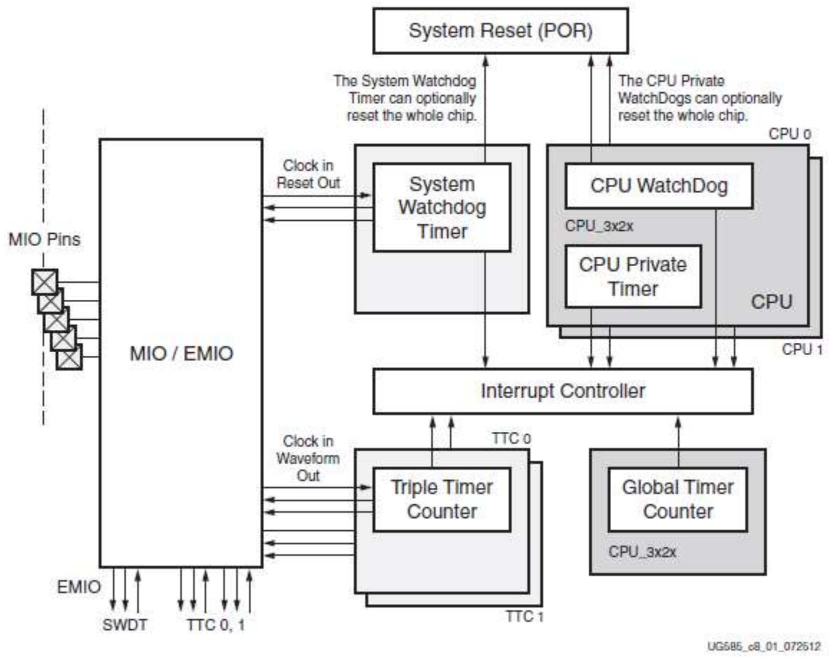
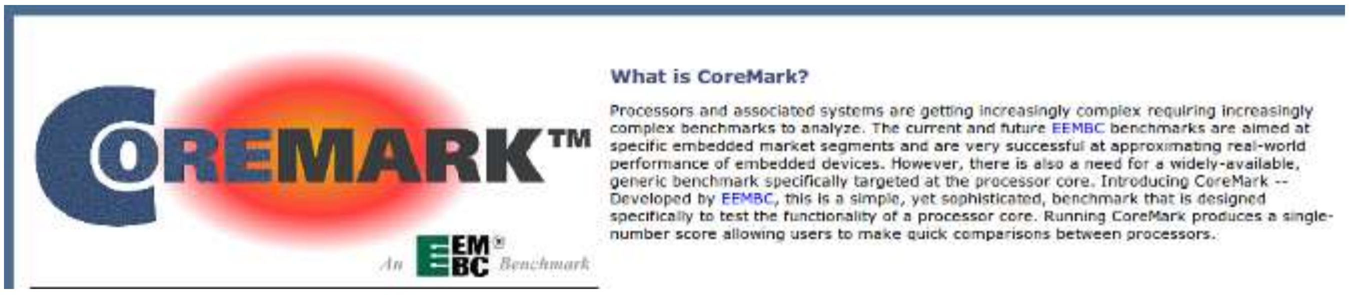


Figure 3. Timers in the Zynq 7000 PS

Exercises for step 3:

a) The Embedded Microprocessor Benchmark Consortium (EEMBC, pronounced embassy) develops benchmarks for embedded systems. Although it doesn't reflect how you would use a processor in a real application, sometimes it's important to isolate the CPU core from the other elements of the processor and focus on one key element. For example, you might want to have the ability to ignore memory and I/O effects and focus primarily on the pipeline operation. *CoreMark* is capable of testing a processor's basic pipeline structure, as well as the ability to test basic read/write operations, integer operations, and control operations.

   CoreMark is a benchmark that aims to measure the performance of central processing units (CPU) used in embedded systems. It was developed in 2009 at *EEMBC* and is intended to become an industry standard, replacing the antiquated *Dhrystone* benchmark. The code is written in C code and contains implementations of the following algorithms: list processing (find and sort), Matrix (mathematics), manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC.



What is CoreMark?

Processors and associated systems are getting increasingly complex requiring increasingly complex benchmarks to analyze. The current and future EEMBC benchmarks are aimed at specific embedded market segments and are very successful at approximating real-world performance of embedded devices. However, there is also a need for a widely-available, generic benchmark specifically targeted at the processor core. Introducing CoreMark -- Developed by EEMBC, this is a simple, yet sophisticated, benchmark that is designed specifically to test the functionality of a processor core. Running CoreMark produces a single-number score allowing users to make quick comparisons between processors.

```
2K performance run parameters for coremark.
CoreMark Size : 666
Total ticks  : 85454207
Total time (secs): 10.249380
Iterations/Sec : 1951.337516
Iterations  : 20000
Compiler version : GCC4.8.1
Compiler flags : -O3
Memory location : STACK
seedcrc   : 0xe9f5
[0]crclist  : 0xe714
[0]crcmatrix  : 0x1fd7
[0]crcstate  : 0x8e3a
[0]crcfinal  : 0x382f
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 1951.337516 / GCC4.8.1 -O3 / STACK
```

For this lab, you will download and test the *CoreMark* benchmark suite running on one of the ARM Cortex-A9 cores. Download the *CoreMark* tarball from the website. The files *timer_util.c* and *timer_util.h* of BSP are low level drivers that define the functions *start_time* and *stop_time* used to time a running application.

Build a new Application Project (named *CoreMark*) based on these source files. Before we run CoreMark, we need to set some compiler flags. Right-click the *CoreMark* project and select *C/C++ Build Settings*, and then *Symbols*. Define the following symbols (equivalent to –D flag in gcc): ITERATIONS = 20000, PERFORMANCE_RUN=1, COMPILER_FLAGS=\"-O3\". Be careful that there should be not shite space between the equal sign "=" and the rest of the characters. Finally, select –O3 optimizations under *C/C++ Build Settings→Optimizations.*

Compile and run the program viewing the output in minicom. When I ran the code in my platform I got the output at the beginning of the next page. A CoreMark score of 1951 iterations/sec means that the 667 MHz ARM CPU will give a CoreMark value of 1951/667 = 2,925 CoreMark/MHz. What is your score?

b) Write your own code and profile it using the infrastructure we discussed. Make sure that your code does some substantial processing and does not have much I/O. If you need to initialize any data structures or to read in input values, you should inline that in your code. Discuss with the Instructor on potential ideas for code profiling.