



מעבדה בארכיטקטורות מחשבים - 045116

חוברת ניסוי למפגשים 1-2

System on Programmable Chip

גרסה 3.0

ספטמבר 2023

הערות לחוברת נא לשלוח

לאינה ריבקין – inna@ee.technion.ac.il

Table of Contents

1	Part A – Introduction	5
1.1	System on a Programmable Chip (SoPC).....	5
1.2	Motivation	7
1.2.1	Internet of Things	7
1.2.2	Experimental Design.....	7
2	Background	8
2.1	The ZYNQ Chip	8
2.1.1	Application Processor Unit	9
2.1.2	Software stack	10
2.1.3	AXI bus.....	11
2.1.4	Programmable logic	12
2.1.5	Hardware Software Co-design	13
2.2	The ZedBoard	15
2.3	Design Environment and Tools	17
2.3.1	Xilinx Vivado	17
2.3.2	Xilinx SDK.....	18
3	Adding Peripherals to the ZedBoard.....	19
3.1	KeyPad specificaions	19
3.2	Specification for the KeyPad Interface Logic.....	19
4	Part A – Preparation report	20
5	Part A – Execution in the Lab	21
6	Part B – Introduction	55
7	Controllers and operating systems	57
7.1	Board Support Package BSP	57
7.2	Event driven programming	58
7.2.1	Polling versus interrupt	58
7.3	Ethernet communication in a basic environment.....	59
7.3.1	GUI for the Host to Zedboard Ethernet Communication	60
7.4	Sound1 (custom IP)	61
7.5	Seven_Segment (custom IP)	63
7.6	Buttons (custom IP).....	64
7.7	Time_Count (custom IP).....	65
7.8	Time of Day concept	66
8	Part B – Preparation report	68
9	Part B – Execution in the Lab	85

מטרת הניסוי - System On Programmable Chip:

- תכנון system מלא על רכיב מתכנת, המכיל בתוכו אלמנטים של חומרה ותוכנה.
- הכרה ויישום flow של חומרה ותוכנה ושילובם.
- טעינה והרצת התכנון על כרטיס חומרה המבוסס על רכיב מתכנת של חברת Xilinx מסוג ZYNQ, המכיל בתוכו מעבדי ARM.

אופן ביצוע הניסוי:

- לפני תחילת הניסוי חובה לקרוא את הנחיות הבטיחות (עמוד 4), את חוברת הדרכה ואת דפי נתונים של הרכיב.
- הניסוי מתחלק לשני חלקים (א' ו – ב') נפרדים, לכל חלק מוקצים 4 שעות במעבדה.
- לחלק א' של הניסוי יש להכין דו"ח הכנה כמפורט בפרק 4 בחוברת, **ללא דו"ח מכין לא יתבצע הניסוי.**
- דו"ח מסכם של חלק א' ייכתב במהלך הניסוי ויתבסס על template אותו תקבלו במהלך הניסוי. יש להגישו למדריך בסיום הפגישה במעבדה.
- לחלק ב' של הניסוי יש להכין דו"ח הכנה כמפורט בפרק 8 בחוברת, **ללא דו"ח מכין לא יתבצע הניסוי.**
- בחלק ב' של הניסוי כל קבוצה תקבל משימה מהמדריך אותה יהיה עליה לממש.
- את הדו"חות המכינים יש להגיש לפני הפגישה במעבדה.
- תוך שבועיים מביצוע חלק ב' של הניסוי יש להגיש דו"ח מסכם.
- **הדו"ח המסכם יכלול תדפיס של כל המשימות שבוצעו במעבדה והסברים מפורטים.**
- הניסוי מתקיים בבניין פישבך חדר 413.

הנחיות בטיחות

עשה ואל תעשה:

1. "אין" לבצע ניסוי במעבדה ללא קבלת ציון עובר בקורס הדרכת בטיחות בחשמל.
2. במעבדה על הסטודנטים לקיים את הוראות הבטיחות כפי שגולמזו בקורס בטיחות.
3. אין לבצע פעילות במעבדה ביחידות.
4. יש לידע את המדריך על מצב מסוכן וליקויים במעבדה או בסביבתה.

בטיחות חשמל

1. המעבדה מצוידת במגני זרם פחת אך זכור שהם אינם מקנים בטחון מלא נגד התחשמלות.
2. הניסוי כולל מחשב ומדגם ניסוי המורכב מכרטיס הדגמה ומסך. כל הכבלים בניסוי מחוברים ואין לשונותם ו/או לנתקם (לא של המחשב ולא של המדגם).
3. במידה ואחד החיבורים מנותק יש לקרוא למדריך.
4. אין לנסות לתקן תקלה כלשהי במערכת הניסוי ו/או המחשב אלא לקרוא למדריך.
5. בשום מקרה אסור לפרק את המחשב או את מערכת הניסוי!

יציאות חירום

1. במעבדה ישנה דלת יציאה נוספת שנעולה בשימוש שותף. לפתיחתה במקרה חירום ניתן להיעזר במפתח החירום הממוקם בקופסא הצמודה לדלת.
2. בקומה ישנן דלתות יציאת חירום – זהה אותן.
3. דלתות החירום שאינן בשימוש שוטף נעולות. לפתיחתן ניתן להיעזר במפתחות החירום הממוקמים בקופסא הצמודה לדלת. שימוש במפתחות יגרום להפעלת צופר.
3. באירוע חירום הדורש פינוי, כגון שריפה, יש להתפנות מיד מהמעבדה דרך פתח החירום הנגיש ביותר.

דיווח בעת אירוע חירום

1. יש לדווח מידית למדריך או לסגל המעבדה. כולל "כמעט תאונה".
2. המדריך ידווח מידית לקצין הביטחון בטלפון: 2222, נייד: 054-544575. במידה ואינו יכול לעשות כך, ידווח אחד הסטודנטים לקצין הביטחון.
3. לפי הוראת קצין הביטחון, או כאשר אין יכולת לדווח לקצין הביטחון, יש לדווח, לפי הצורך: משטרה: 100, מגן דוד אדום: 101, מכבי אש: 102. בנוסף, יחידת סגן המנמ"פ לענייני בטיחות: 2146/7, 3033.
4. בהמשך, יש לדווח לאחראי משק ותחזוקה: 4776, 054-9456346.
5. לסיום, יש לדווח לאחראי אקדמי 4637, למהנדס המעבדה: 4664/4789, לעוזר למנהל: 4678 ולאחראי הנדסי 4794.

Smart Devices for the Internet of Things using SoPC

1. Part A – Introduction

1.1 System on a Programmable Chip (SoPC)

A SoPC is a variation of a System on a Chip (SoC). In earlier days, the hardware of a computer system consisted out a processor chip, a memory management unit, and a variety of auxiliary chips to work with all the peripheral such as display, keyboard etc.. Those components where connected via busses. With the increasing possibilities in IC (Integrated Circuit) manufacturing more and more components could be moved onto a single chip.



Figure 1 From Mother Board to SoC

In other words, a SoC is an electronic component which features a processing element and several I/O control elements all on a single IC. The configuration of a SoC is fixed. A major disadvantage of such a chip is its lack of flexibility together with high development costs. It is well suited for the high volume market where no changes or upgrades are expected. The main chip in a cellphone which contains processor cores but also a great number of peripherals, it may even integrate a GPS receiver or a gyroscope adapter is a good example for fixed configured SoC.

For low volume a fixed configured SoC is rather unattractive. This market requires high flexibility parts. Here comes the SoPC into play. A SoPC is based on FPGA (Field Programmable Gate Array) technology. It also includes a processor core on chip. Sometimes the processor core is synthesized like the rest of the FPGA. In this case we talk about a softcore. Other versions carry an optimized processor on the chip which only needs to be connected. Such processors are called hardcore. For small systems the memory will probably be on chip, for larger systems the chip contains a memory controller to access off-chip memory. There may be other components on the chip but the main feature is the FPGA area which allows the customized design of any peripheral component. The processing element

and its periphery are connected through standard busses. For the processing element to make use of the fixed or custom designed peripherals, the control registers of those peripherals must be all part of the IO-register space of the processor.

Cell Phones, Digital Cameras, Set Top Boxes are typical representatives for use of SOC's. For a SoPC there are four main uses:

- Prototyping for an SOC
- Very low volume SoC applications
- Applications with frequently changing requirements of the interfaces
- Hardware based accelerators.

In other word, where flexibility is the key requirement use a SoPC; where you have an established environment without much changes use a SoC.

1.2 Motivation

1.2.1 Internet of Things

The Internet of Things is the next step. It is a gathering of “things” which contain electronic and connectivity to communicate with other “things” and or operators. So what will really happen when things, homes and cities become smart? The result will probably be a tsunami of what at first looks like very small steps, small changes. If you have a thermostat, alarm system, or lights in your home that you can control with your computer or phone, you entered the internet of things already.

One enabler for the internet of things is the increasing use of SoC to make the “things” smart. At least at the beginning, let’s say in the prototyping stage or so SoPC are widely used. This set of experiments will draw on the increasing use of “smart things” and will bring you in contact with a variety of aspects of the design process for those components.

1.2.2 Experimental Design

In the set of experiments presented here, we will use a Xilinx development kit termed ZedBoard. The Zedboard (see 2.2) is based on a ZYNQ SoPC (see 2.1).

Our aim is it to create some example system including hardware logic design and software applications development. For the hardware design we will use

- System Verilog design language
- Xilinx VIVADO development environment (see 2.3.1 for brief overview)

Using those tools, we will develop some external interface, package the design into an IP (Intellectual Property core - a reusable unit of logic) and integrate the IP into the hardware system. Our new design is connected via a standard interconnect with the processor in the SoPC. The necessary mini driver to access our design from a program on the processor is created automatically during the IP packaging process.

After the hardware is completed it is tested with software running on the processor. For the development of the application software we will use

- C/C++ programming language
- Xilinx SDK software development environment (see 0 for brief overview)

All over we will learn how

- To build an IP
- To integrate several IP into a system
- To control the hardware from software running on the processing element
- To control the system from external over the internet

2 Background

2.1 The ZYNQ Chip

Xilinx, the producer of the ZYNQ calls it an All Programmable SOC to manifest it as one step up from a SoPC. The chip is divided into two distinct areas, the PS (Processing System) and the PL (Programmable Logic). Both are connected through AXI bus system. The PS system features a Dual Core ARM Cortex A9 processor together with Cache and Memory interface and a number of fixed IO interfaces. As such it has a fixed architecture with only the possibility of selecting some of the pins for the fixed IO.

The PL side, however, provides FPGA logic to design custom logic blocks or use commercial available logic blocks to achieve the desired behavior. There are design tools provided to interconnect between the design blocks themselves as well as to the processor bus interface and the IO driver/receiver parts

It is up to the system designer to decide which parts of his system will be implemented in software running on the two processors and which part will be implemented in logic running in the PL part and communicating with the program and the outside world.

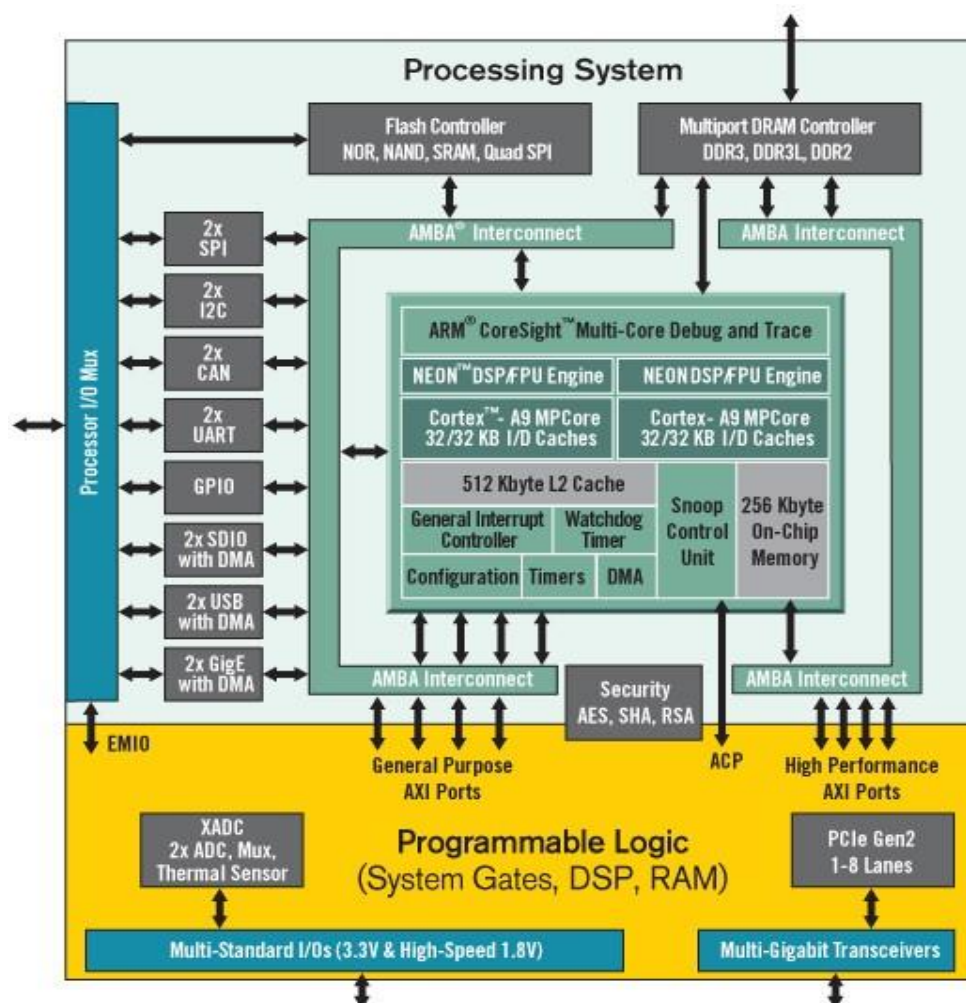


Figure 2 ZYNQ Architecture

2.1.1 Application Processor Unit

The ARM Cortex A-9 is a multiprocessor core which implements the ARMv7 architecture. Two of those cores together with the snoop control unit and 512 KB of L2 cache form the applications processor unit of the ZYNQ chip.

Each of the cores is a powerful state of the art RISC processor which can run up to 1 GHz depending on the speed grade of the chip. Beside the fixed point engine it also features a floating point engine and the NEON Media processing engine. To enable cache coherency for multiprocessor operations, the snoop control unit is included connecting the first level caches of each processor to the second level cache of the processing unit.

The processing unit runs a bare metal operating system or a version of either LINUX or Android. Programming and debug environment for code running on the processing unit under one of the operating systems is provided via XILINX Software Development Kit (SDK).

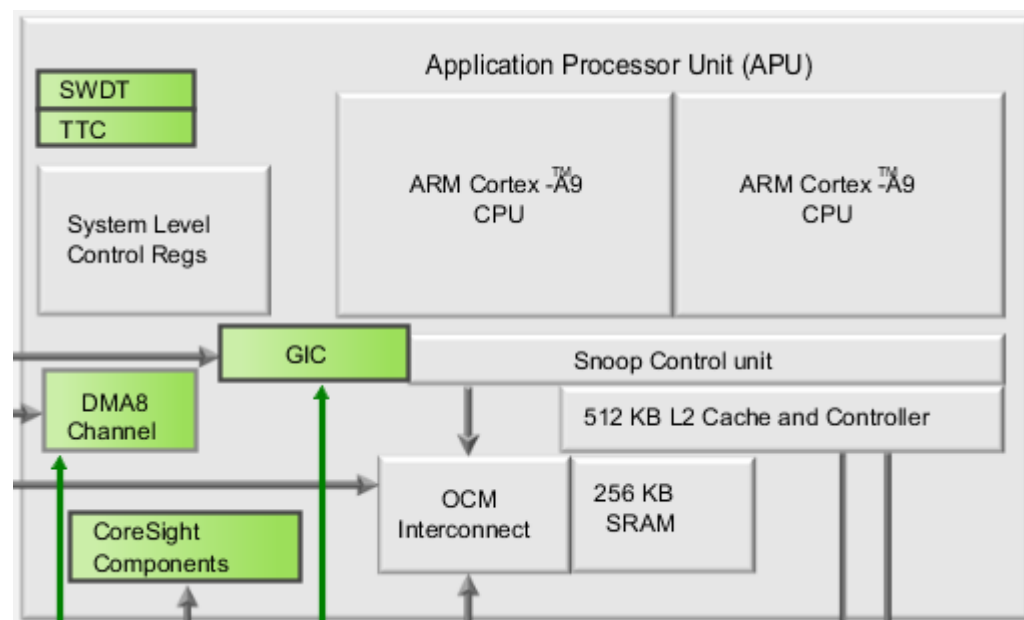


Figure 3 Application Processing Unit

Part of the processing system portion of the ZYNQ chip is a set of interfaces. Those interfaces are not directly wired to IO pins but rather can be multiplexed to a set of 54 pins. These interfaces include Gigabit Ethernet, USB, CAN, UART and I2C as well as SD card and GPIO. The interfaces are hard core blocks. During a processor configuration phase the interfaces for a given task are selected. This enables some flexibility. However, the interfaces need physical connections outside the chip. So once the chip is mounted on a printed circuit board which contains the physical connectors, the flexibility becomes very limited. In the section about the ZedBoard we discuss which interfaces are useable.

2.1.2 Software stack

From the SoC point of view the application processor is a part of the overall hardware, it may be its central part. The functionality of this hardware part generated through the software stack running on it. Generally the stack is divided into three layers.

- The application layer
- The operating system layer
- The interface layer

The interface layer is the lowest layer which is talking to the underlying hardware interfaces. It will make use an address map of all the hardware registers in the system accessed via the AXI bus. If the software part for the product is small we may do most of our development at this layer. In the other case a driver needs to be developed connecting the interface with the OS layer.

For the ZYNQ / ARM A9 we have the choice between

- a bare metal OS which provides only the basic functions like setting up cache, interrupts and exceptions as well as other hardware functions
- a version of embedded Linux
- Android (widely used with touch screens)

The top layer of our stack is the applications program which is designed for our specific need.

2.1.3 AXI bus

The AXI bus forms the interface between the processing system and the programmable logic as well as it is widely used in the programmable logic to connect between logic blocks. AXI4, the version implemented on ZYNQ, is part of the open **ARM Advanced Microcontroller Bus Architecture** (AMBA3) standard. Contrary to the suggestion in the name, AXI4 is a point to point master slave interconnect. Interconnect hardware which perform the bridging between multiple sources may also convert between different width of the data path and different clock speeds. The AXI4 bus comes in 3 different versions

- AXI4 For high performance memory mapped operations. It supports burst mode with up to 256 data-words and a maximal width of 1024 bits per data word.
- AXI4-light For register style memory mapped interfaces. No burst support. Maximal width for the data word is 64 bits.
- AXI4-stream For streaming data. No addresses provided. Length of stream is unlimited as well as the width of the data word transferred in each cycle.

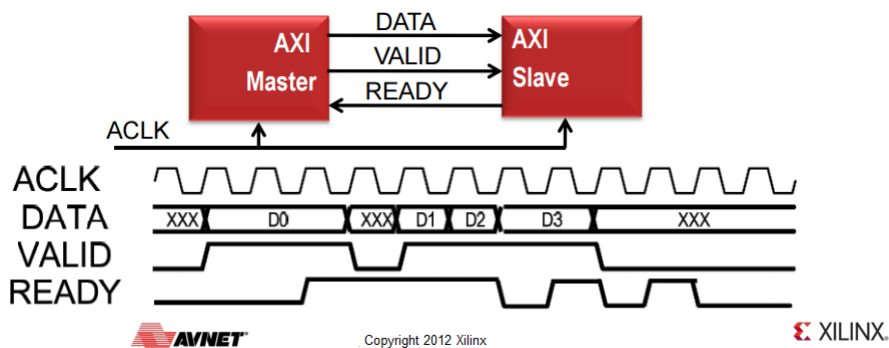


Figure 4 AXI4 Data Signaling

Data are transferred when both Valid and Ready are active. Full AXI4 consist of 5 channels

- Read Address Channel
- Read Data Channel
- Write Address Channel
- Write Data Channel
- Write Response Channel

Each of those channels has its pair of ready and valid signals. For burst mode a **Last** transfer signal is added to the read and write data channels.

On ZYNQ there are 4 AXI4-light interfaces between PS and PL. On two of them the processing system is the master, for the other 2 the PL forms the master. The interfaces are 32 bit wide. For high performance access to the processing system memory from the PL side, ZYNQ provides 4 full AXI4 interfaces of 32 or 64 bit data width. PL is the master for all of these interfaces.

The high performance interfaces connect directly into the memory subsystem. A data transfer via these interfaces will not snoop and thus will not be part of the cache coherency protocol. This means if data is stored by the PL into a certain location in memory and the processor had previously accessed this location and the corresponding data from this access are still in its cache, it will not see the new data until the cache line was purged. For cases where coherency is needed an additional full AXI4 coherency interface with 64 bit data word width between PL and the snoop control unit is provided.

2.1.4 Programmable logic

The second part of the ZYNQ chip which transforms this chip to an all programmable SOC is the PL area. This area is an Artix 7 FPGA. A FPGA is a semiconductor device based on an array of Configurable Logic Blocks (CLB's) connected through programmable interconnects.

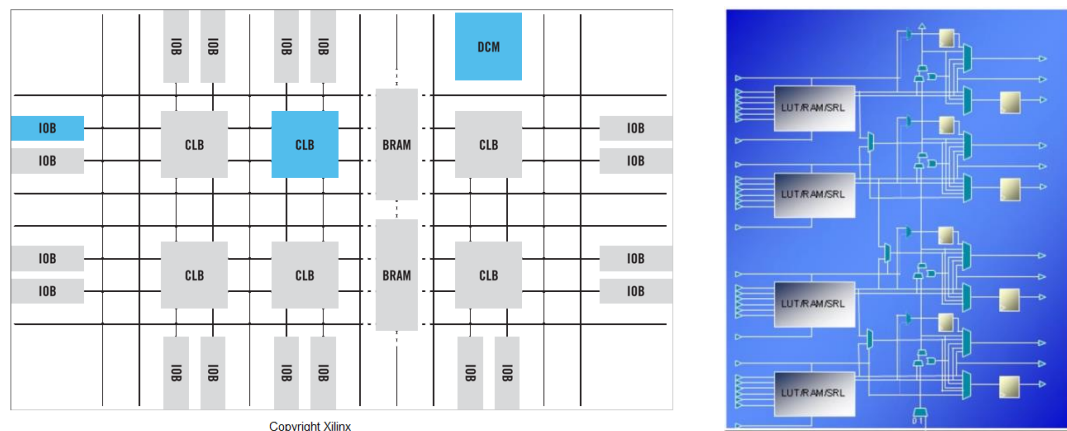


Figure 5 FPGA Basic Structure

Each CLB contains some input select logic by which the required inputs are selected from interconnect and 2 logic slices. Each of the slices contain 4/6 input LUT's and 8 flip flops as well as routing logic. Other elements on the FPGA are selectable IO's, Block Memory (BRAM), Clocking circuitry and DSP's.

The clocking circuitry include special receivers including dedicated locations of those receivers, a number of possible PLL(Phase Lock Loop) structures and dedicated wiring channels to minimize clock skew between Flip Flops within clock regions. A PLL is an electronic circuit consisting of a variable frequency oscillator and a phase detector allowing to generate internal clocking signals which are locked in phase to an external supplied clock.

Using LUT's from the logic slices, every arithmetic operation of any word length can be implemented. However, such an implementation may become large in size and/or slow in its performance. The DSP is a specialized block dedicated to high performance fixed format multiplications and or additions.

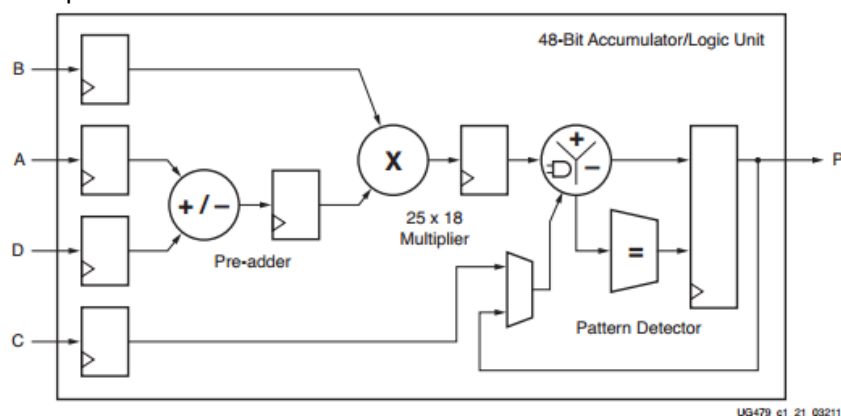


Figure 6 DSP Basic Function

The design is specially optimized for the design of FIR filter with an even number of coefficients but it is not limited to this application. It is rather widely used in all designs.

2.1.4.1 IP's

The PL part normally contains some portion which will perform a logic function. This logic functions is designed in a High level Design Language such as VHDL or Verilog. It is not uncommon that parts of logic functions are used over and over again within the same project or between projects. To enable such reuse this logic parts need to be packaged. The design should be verified independently of its original design surrounding and if possible the design should be parameterized. If all those hold we speak of an Intellectual Property Core (IP).

Customizability of the IP depends on the form the IP is delivered in. If it is a hard core, there is almost no customization possible except for setting some configuration registers, if, however, the IP is synthesizable HDL its customization possibilities are very high.

For FPGA design systems a great number of IP cores already exist. IP cores are available either from Xilinx, one of their partners or may be home grown. Xilinx groups its designs into the following categories.

- Interfaces and Interconnects (all AIX4 interconnect IP's and PCIe)
- DSP and Math (counters as well as complex multipliers and FFT's)
- Embedded (Soft processors, timers and interfaces)
- Communication (Ethernet and packed processing)
- Memory controllers
- Video (Image processing and compression)

As said earlier, also self-designed IP's should be packaged in a way that enables its reusability. The Xilinx design environment provides this packaging option and may automatically add an AXI4-light interface to integrate it easier into the final design. If such an interface is desired, the designer has to obey naming conventions expected by the Xilinx IP Packager.

There are many ways to design your own IP, first would be the use of a high level design language such as VHDL or Verilog. Another method is the use of Simulink System Generator which can output RTL (Register Transfer Language) files to be packaged into IP's. Last but not least, High Level Synthesis can be used to transform C/C++ code into RTL and from there into an IP. For now we will concentrate on the first method.

In the experiment we will modify existing Verilog code for a peripheral interface, add a corrected AXI4-light bus interface to it and package it all up into a new IP core. The IP core will then be integrated and tested in subsequent steps of the experiment.

2.1.5 Hardware Software Co-design

There are many different ways to design a SoC. One would be if possible to design a pure software system, profile it and decide which parts would be best utilized in hardware.

However the more commonly used way is a hardware software co-design. It is based on a top down design approach and is depicted in Figure 7 below.

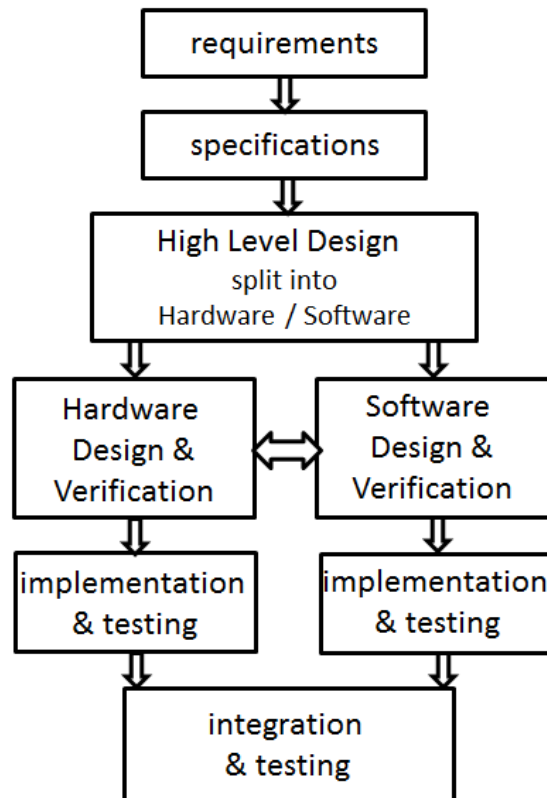


Figure 7 Design Flow

As part of the high level design of the system it is decided which functions are to be realized in hardware and which in software. For the verification process of the hardware there should be a high level model of the software and vice versa. After each part has been tested stand-alone the final integration step brings everything back together.

2.2 The ZedBoard

The ZedBoard is an evaluation and development platform which enables hardware and software developers to create or evaluate Zynq™-7000 All Programmable SoC designs. It features the XC7Z020 ZYNQ device and is also targeted at academics and students to start SoC designs and having a community (ZedBoard.org) to support them.

The XC7Z020 ZYNQ device is one of the smaller ZYNQ devices featuring 13,300 logic slices, 220 DSP's and 140 BRAM's. There is an ADC hardcore XADC but no high speed transceivers needed for a PCIe interface.

Figure 8 depicts the ZedBoard. Even we speak of a SoC at the hard of the board there are a number of external components on board to provide the needed flexibility of an evaluation platform.

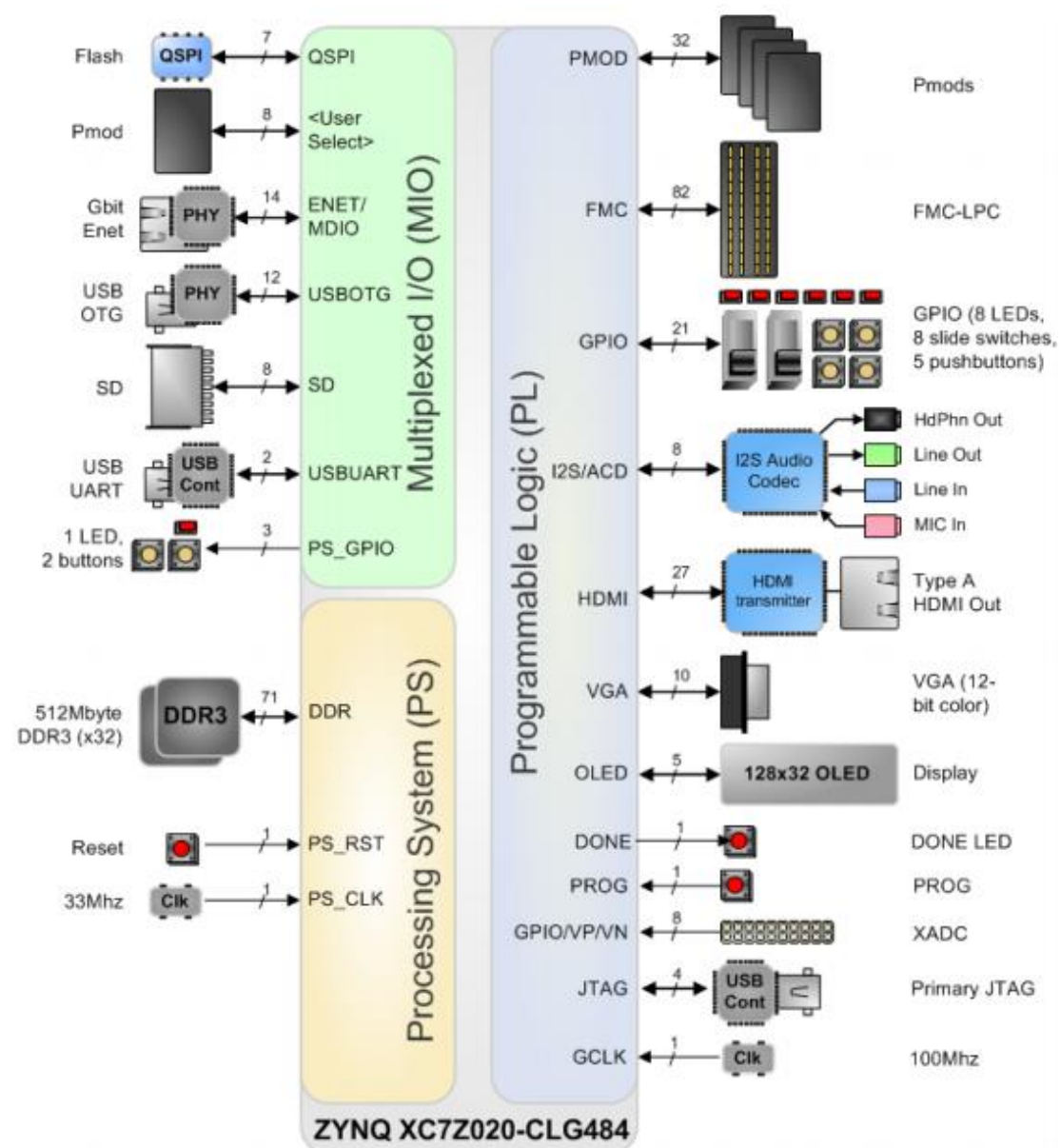


Figure 8 ZedBoard

Part of the external interfaces and components are connected to the PS side and only need the software to access them. Others are connected to the PL side. These interfaces need also IP's (interface logic) to enable their use.

A detailed description can be found in the Hardware User Guide

http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

The figure below shows the physical locations of the components and interfaces. SD card cage and QSPI Flash are not shown as they reside on the backside of the card.

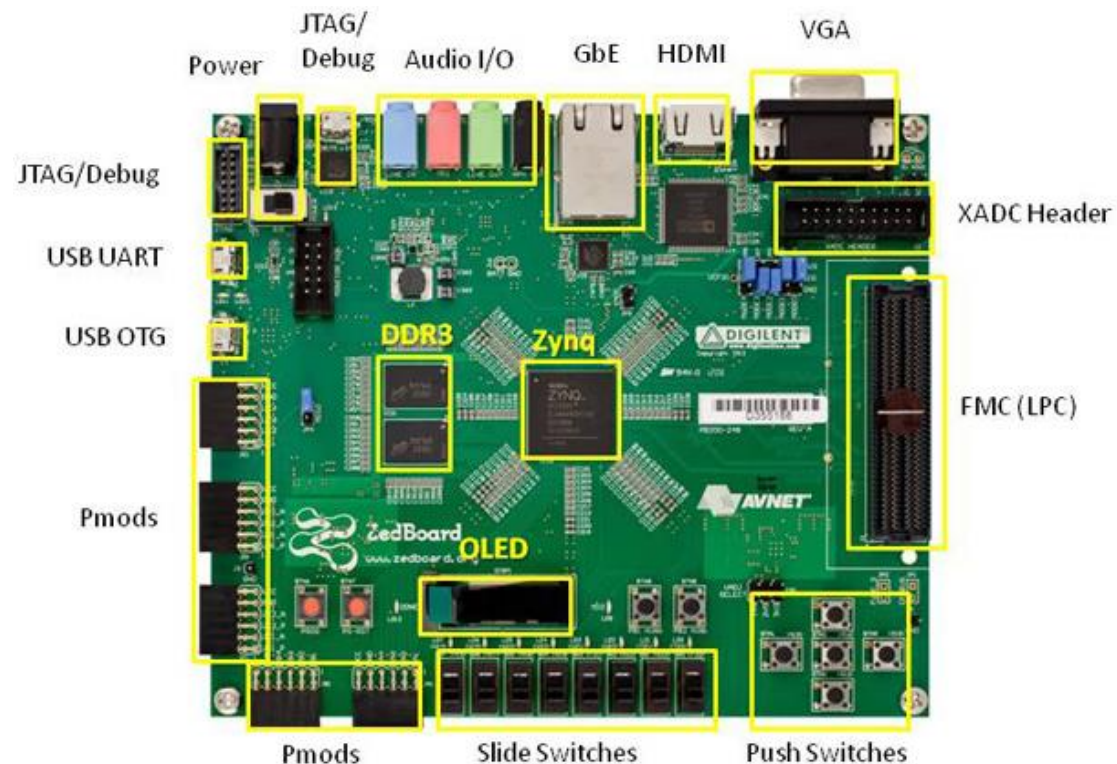


Figure 9 Interface layout on ZedBoard

For a start the ZedBoard needs to be connected to the power and using a USB cable for communicating between the host PC and the JTAG port for programming and debug purposes. A second USB cable may use the USB UART connection to establish communication with the host PC running a terminal application or other programs.

For designs which only involve the PL, they can be loaded through the VIVADO tool (see 2.3.1) hardware manager. If some software has to run on the A9 processors both soft and hardware are loaded through the SDK tool (see 2.3.2).

2.3 Design Environment and Tools

2.3.1 Xilinx Vivado

Xilinx Vivado is a suite of hardware design tools built from the ground up for enhancing IP-centric and system-centric developments. On one hand it is a project based tool suit with a GUI that leads through the complete design flow. On the other hand all underlying design steps are script based so that all tools which are part of the flow can be run just executing a script file. For the experiment we will exclusively use the GUI based flow.



Figure 10 XILINX VIVADO

First a project will be created and loaded with a set of prepared design sources. Those sources will be integrated into a system. Whatever is missing in the design to perform the desired task will be added using a high level design language e.g. VHDL or Verilog. After verifying the designed components they will be converted to IP's and integrated into the previously started system.

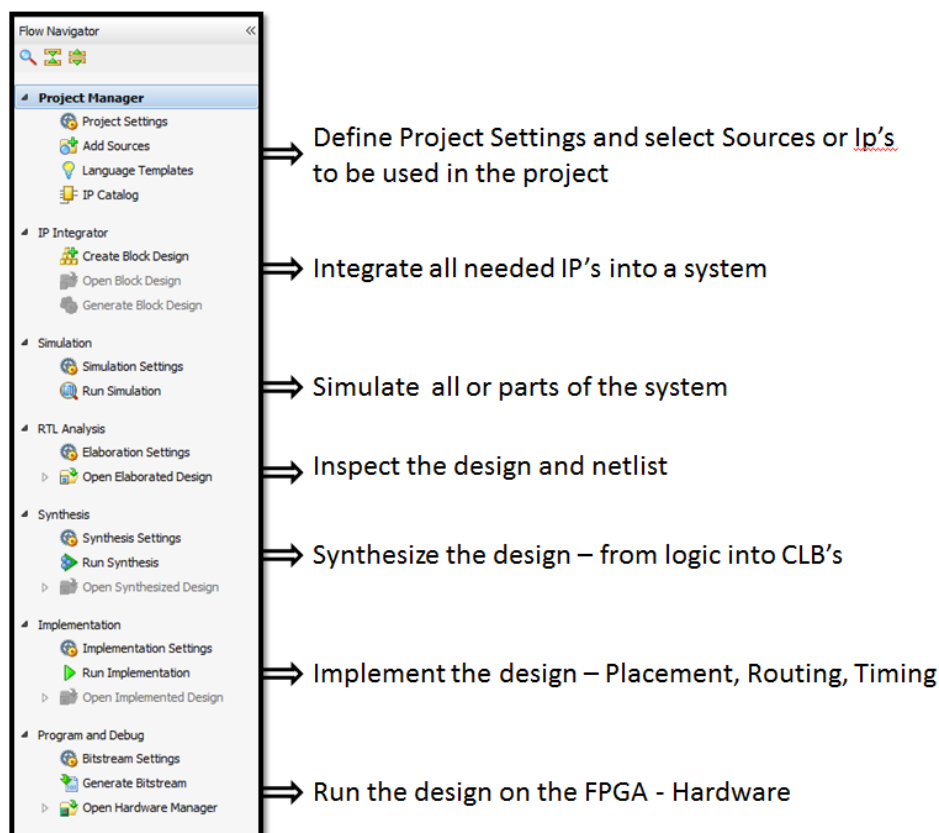


Figure 11 Design Flow

The final output of the design process on Vivado is a bitstream to be loaded into the FPGA. A bitstream is a stream of data that contains the contents of LUTs, memories and location information for logic on a device, that is, the placement of configurable logic blocks (CLBs), input/output blocks (IOBs), 3-state buffer (TBUFs), pins, and routing elements. When downloaded to a device, a bitstream configures the logic of a device and may load initial values into memory elements. A bitstream file has a .bit extension.

2.3.2 Xilinx SDK

The Xilinx® 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 Xilinx Platform Studio (XPS) embedded development tools or XILINX VIVADO. SDK is based on the Eclipse open source standard. SDK features include:

- Feature-rich C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic Makefile generation
- Error navigation
- Well-integrated environment for seamless debugging and profiling of embedded targets
- Source code version control

2.3.2.1 SDK Development Flow Overview

The typical flow for using SDK to develop a software application for an XPS embedded system design is as follows:

1. Invoke SDK and open an existing SDK Workspace, or create a new one.
Note: When creating a new workspace, use a Hardware Platform that was generated from Vivado® IP Integrator or XPS.
2. In SDK, create board support package projects that contain a library of routines that your application can use.
3. Create application projects that will capture your application source files and settings.
4. Develop your software application. SDK provides documentation for the embedded system and software platform (including drivers).
5. Use SDK linker script generation tools to modify your application's memory map.
6. When you are ready to test your application on the hardware target, run/debug the application by setting the required run/debug configuration.
Also, if required, download the hardware bitstream to the FPGA device.
7. Use SDK to download and run the software executable. You can also debug and profile your application.
8. You can design your hardware platform in parallel with software development. You can have SDK update your hardware platform specification by pointing to a different version and have SDK modify your workspace for the new hardware platform.

Further documentation under

https://www.xilinx.com/html_docs/xilinx2018_3/SDK_Doc/index.html

3 Adding Peripherals to the ZedBoard

As shown in the previous chapter, the ZedBoard contains several switches and pushbuttons to be used for input devices but this might not be enough. To increase the available inputs, a KeyPad is to be connected to one of the PMOD connectors Figure 9 of the ZedBoard.

3.1 KeyPad specifications

From <https://digilent.com/reference/pmod/pmodkypd/reference-manual?redirect=1>

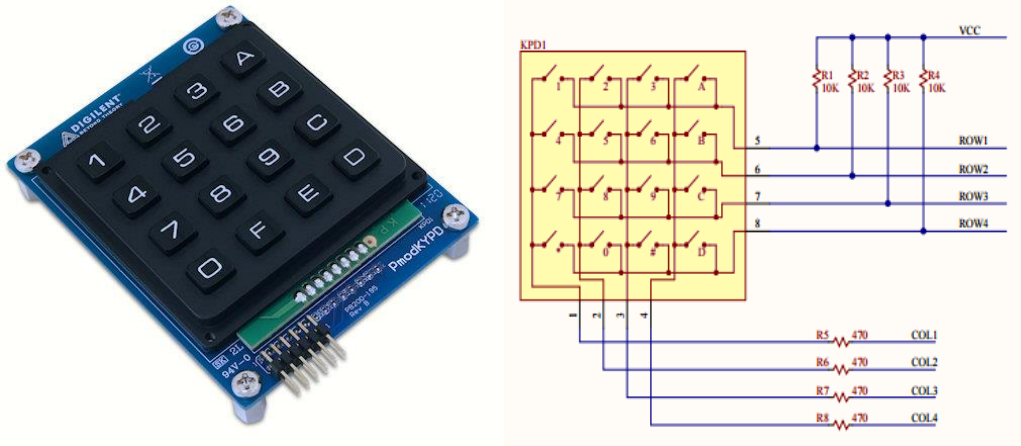


Figure 12 Key Pad Image and Schematics

The PmodKYPD is set up as a matrix in which each row of buttons from left to right are tied to a row pin, and each column from top to bottom is tied to a column pin. This gives the user four row pins and four column pins to address the button push. To read a button's state, the column pin in which the button resides must be pulled low. This enables all of the buttons in that column. When a button in that column is pushed, the corresponding row pin will read logic low. All of the buttons can be read by walking a logic 0 through each column pin (keeping the other pins at logic high) and reading the row pins. This will read the state of each button.

In other words, if button 5 is depressed only when driving column 2 to '0' a '0' will be read on row 2. The combination of row and column number being '0' identifies uniquely the button pressed.

3.2 Specification for the KeyPad Interface Logic

The logic needs 4 physical interface output pins, one per column, and 4 physical input pins, one per row. Using a state machine the output pins are cyclically driven to a logic '0' one at a time. Whenever one pin is '0', all input pins are read. After all 4 columns have been set to '0' once, it can be determined if a button had been pressed and if so which one. A value corresponding to the 'hex' number of the button is loaded into a register.

In another routine which interfaces with the AXI4-light bus, this register is read and presented to software on the ARM A9 processor. Software then will print out the value received from the interface

4 Part A – Preparation report

דו"ח מכין חלק א':

1. הסבר **במילים שלך** את המושגים הבאים:
 - a. SoC
 - b. ZYNQ
 - c. ZedBoard
 - d. UART
 - e. AXI
 - f. IP
2. עיין בדפי נתונים של רכיב ה-ZYNQ (קובץ ds190-Zynq-7000-Overview.pdf). הסבר את המושגים PS ו-PL, מה המרכיבים העיקריים של כל חלק.
3. עיין בסעיף 2.1.3 בחוברת והסבר מה לדעתך תפקידו של AXI BUS במערכת שתמומש בניסוי.
4. באיזה כלי פיתוח נשתמש ע"מ ליצור תכן ברכיב ה-ZYNQ?
5. נתון קוד System Verilog (קובץ keypad.sv) שמממש בקר של פריפריית keypad וקוד C (קובץ part1.c) שרץ על מעבד ה-ARM ומשתמש ברכיב ה-keypad. מטרתו של קוד ה-C לזהות לחיצה על אחד המקשים ולהדפיס את המקש שנלחץ. חשוב שכל לחיצה תזוהה רק פעם אחת, ללא קשר לזמן הלחיצה. שחרור המקש משמעותו שאף מקש לא לחוץ.

קרא את קוד ה-System Verilog ואת קוד ה-C וענה על שאלות הבאות:

 - a. בהינתן שקצב סיגנל ה-clk הוא 100MHz כל כמה זמן נבדקת כל שורה של keypad וכל כמה זמן נבדק כל מקש?
 - b. צייר תרשים זרימה של הלוגיקה שממומשת בקובץ keypad.sv. פרט בכל שלב מה הפעולה המבוצעת ומה הזמנים בין פעולה לפעולה.
 - c. אנו נרצה לממש מנגנון שיאפשר זיהוי בתוכנה של כל לחיצה רק פעם אחת. קוד ה-C בודק את סיגנל ה-released. סיגנל זה מופיע בהגדרת הממשק של הבקר, אך חסר הקוד של הצבת ערך לסיגנל.

שנה את קוד ה-System Verilog של הבקר כך שסיגנל ה-released ייצג את שחרור המקש.

שימו לב – סיגנל ה-released נקרא ע"י התוכנה ולא ניתן לסנכרן בין תוכנה לחומרה ברמת מחזורי שעון, לכן כל משך הזמן שמקש כלשהו לחוץ סיגנל ה-released צריך לקבל ערך כשלהו שמציין "יש מקש לחוץ" וכל משך הזמן שכל המקשים משוחררים סיגנל ה-released צריך לקבל ערך כלשהו שמציין "כל המקשים משוחררים".

רוחב סיגנל ה-released הוא 32bit מהסיבה שהממשק בין חומרה לתוכנה הוא ע"י רגיסטרים ברוחב 32bit. אין חובה להשתמש בכל הביטים.

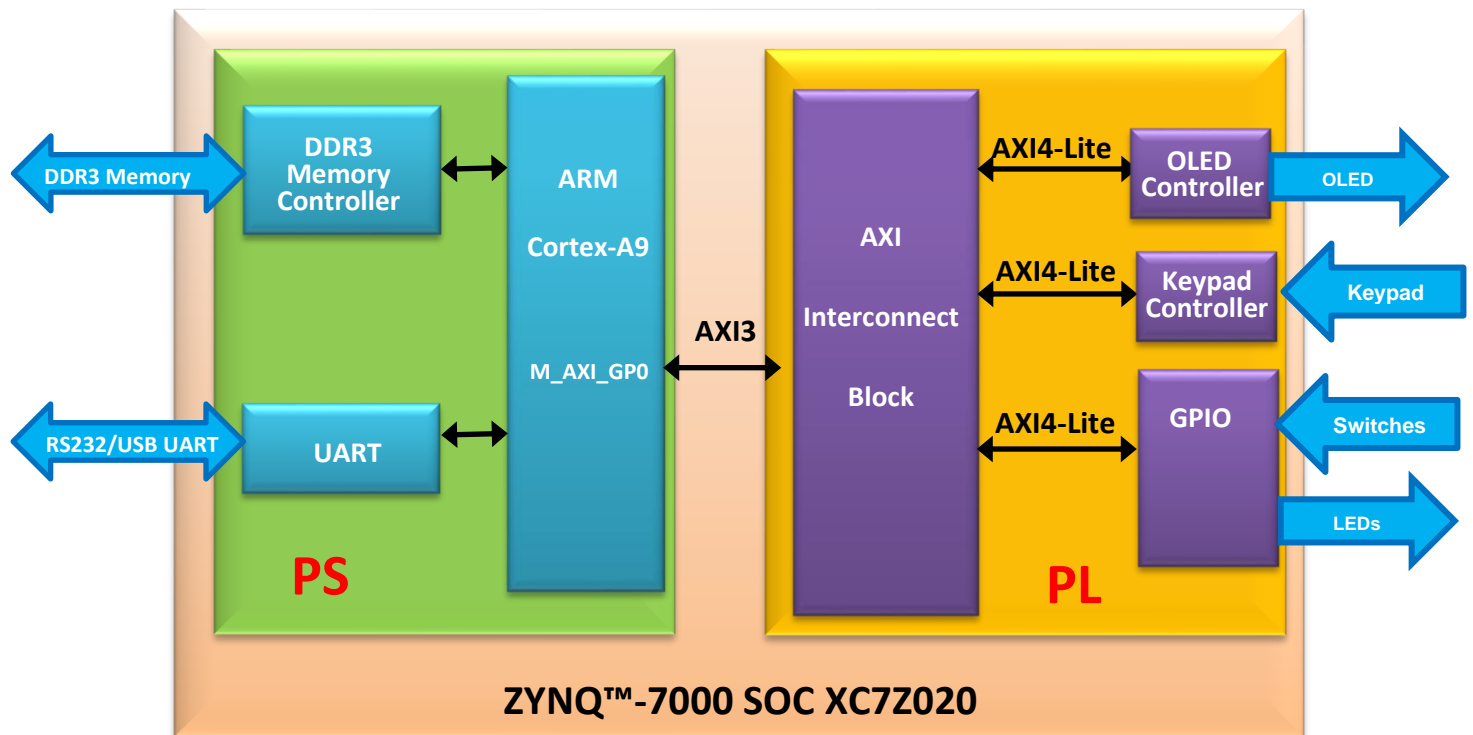
יש לתקן את הקבץ keypad.sv ולהגישו כקובץ נפרד.
6. האם לדעתך אפשר היה לבצע את משימה מסעיף קודם בתוכנה בלבד ללא השינוי של קוד ה-System Verilog הנתון? אם כן הסבר איך. אם לא הסבר למה.

5 Part A – Execution in the Lab

מפגש ראשון – ביצוע הניסוי במעבדה

In the first part of the experiment you will design a complete embedded system consisting of the ARM Cortex-A9 PS and the following peripherals: LEDs, Switches, Keypad and LCD display (OLED).

Block diagram that represents the design:

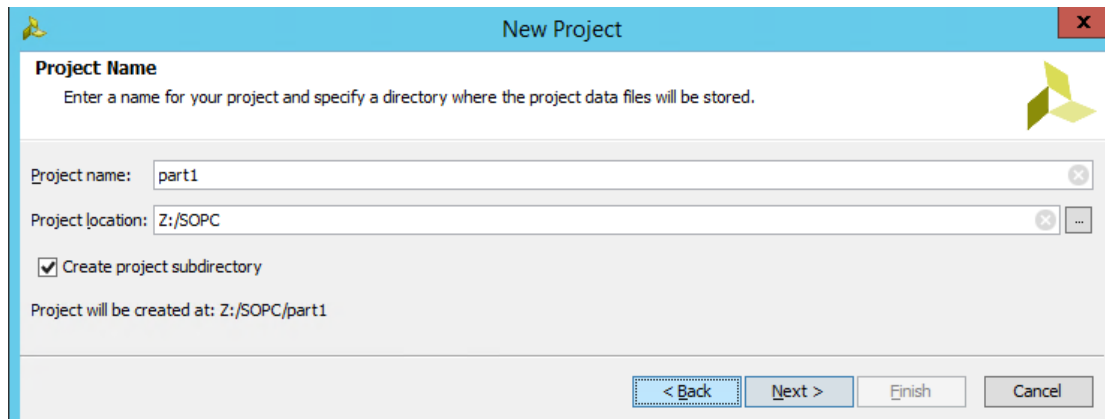


Launch Vivado 2018.3 using  icon.

Click **Create Project** to start the wizard. You will see the Create a New Vivado Project dialog box. Click **Next**

Enter **part1** in the Project Name field. Make sure that the Create Project Subdirectory box is checked.

Click the Browse button of the Project Location field of the New Project form, type **Z:/SOPC/** and click **Select**.



New Project

Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project name:

Project location:

☒ Create project subdirectory

Project will be created at: Z:/SOPC/part1

< Back Next > Finish Cancel

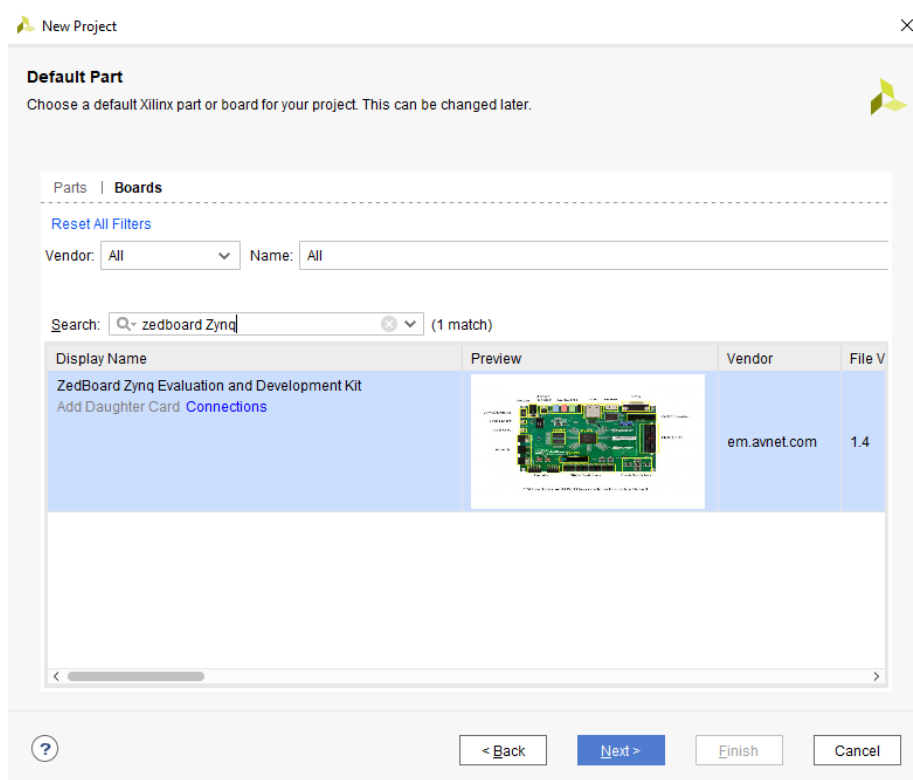
Click **Next**.

Select **RTL Project** in the Project Type form, and click **Next**.

Select **Verilog** as the Target language and Mixed as the Simulator language in the Add Sources form, and click **Next**.

Click **Next** to skip Add Constraints.

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



Default Part

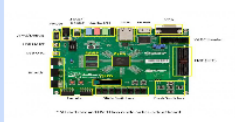
Choose a default Xilinx part or board for your project. This can be changed later.

Parts | **Boards**

[Reset All Filters](#)

Vendor: Name:

Search: (1 match)

Display Name	Preview	Vendor	File V
ZedBoard Zynq Evaluation and Development Kit Add Daughter Card Connections		em.avnet.com	1.4

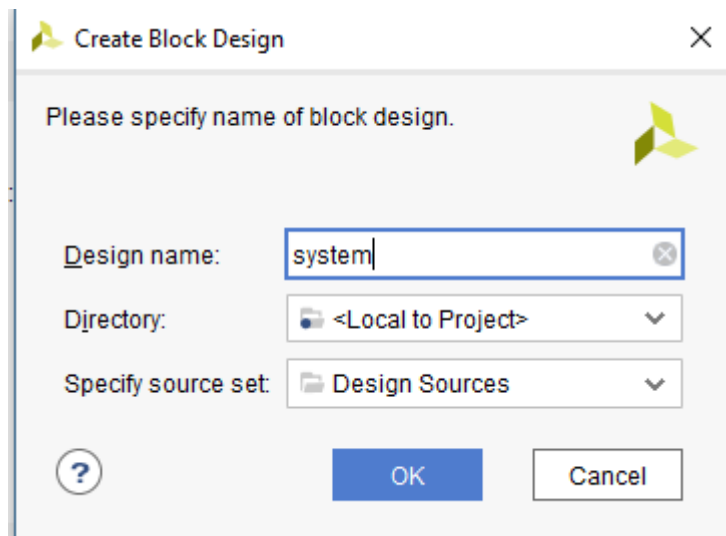
< ? < Back Next > Finish Cancel

Click **Finish** to create an empty Vivado project.

Now we are going to use the IP Integrator to create a new Block Design, and generate the ARM Cortex-A9 processor based hardware system, targeting the ZedBoard.

Click on **Flow** -> **Create Block Design**

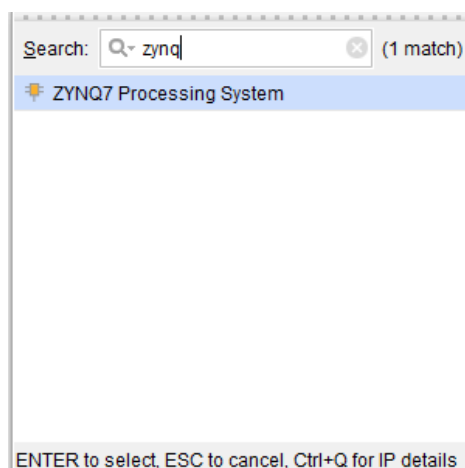
Enter **system** for the design name and click **OK**.



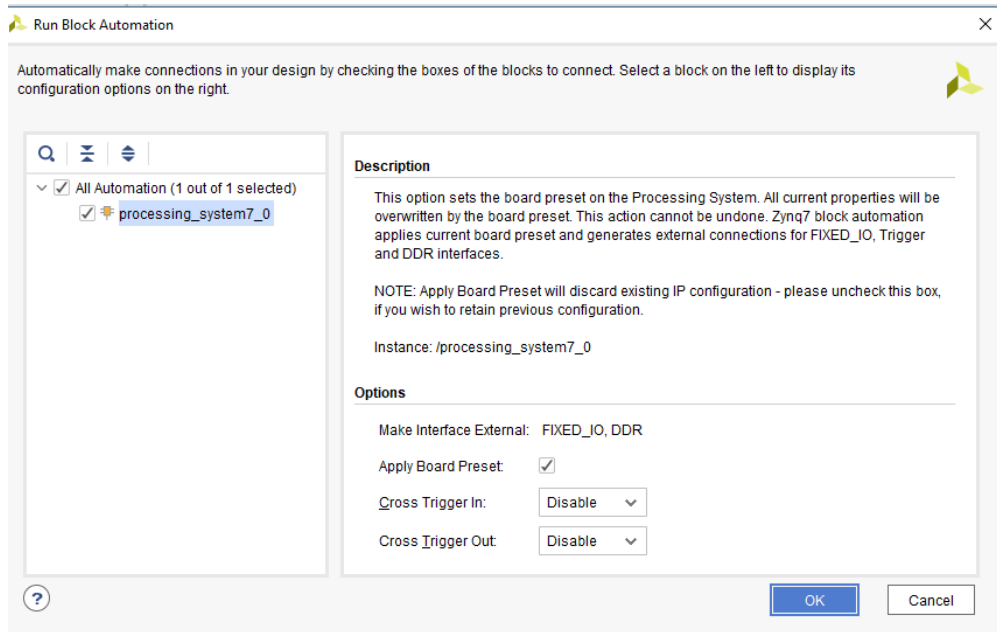
Now we are going to add the PS-Processing System (ARM) to the design using integrated IP Catalog.

IP from the catalog can be added in different ways. Click on Add IP icon **+** or press Ctrl + I or right-click anywhere in the Diagram workspace and select Add IP.

Search zynq and double-click the **ZYNQ7 Processing System** to add it to the design

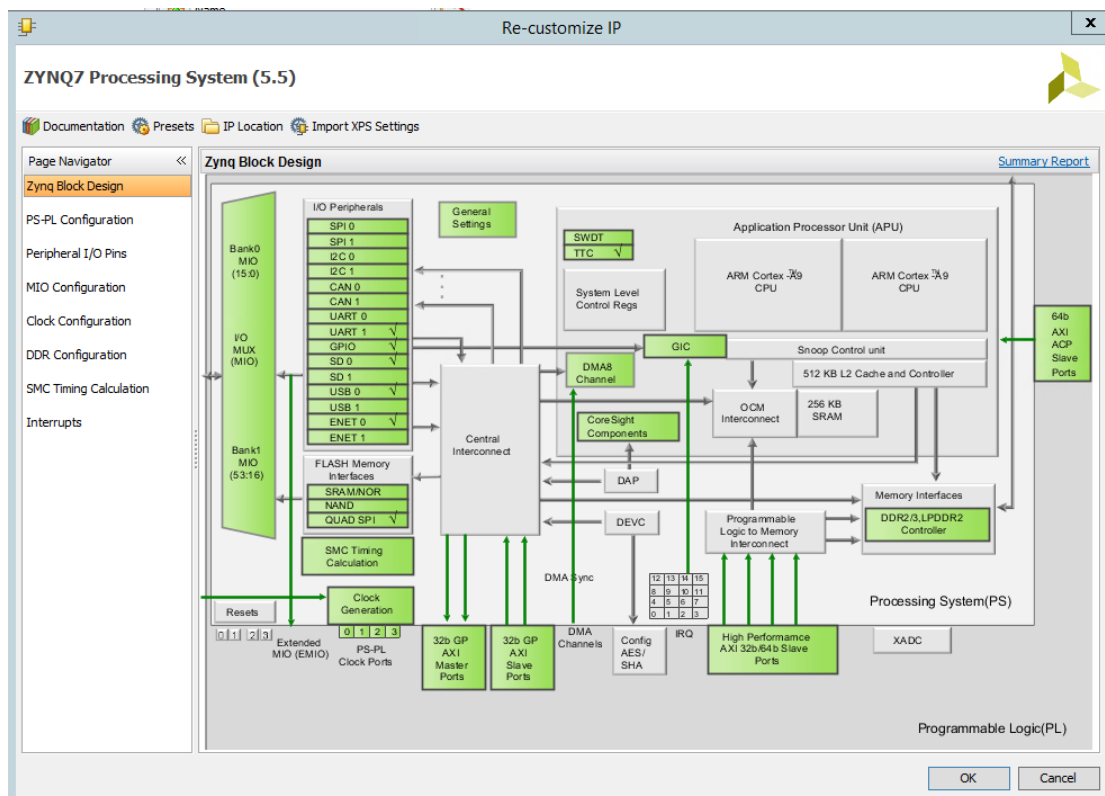


Click on **Run Block Automation** and click **OK** to automatically configure the board presets.



Double click on the Zynq block to open the Customization window for the Zynq processing system.

At this stage, you can click on various configurable blocks (highlighted in green) and change the PS (Processing System) system configuration.



Now we will configure the processing system I/O Peripherals block :

Click on the **MIO Configuration** panel to open its configuration form.

Expand the **I/O Peripherals** and deselect all the peripherals except ENET 0 and UART 1.

Don't change the default values under **GPIO**.

Expand the **Application Processing Unit** and uncheck the **Timer 0**.

Re-customize IP

ZYNQ7 Processing System (5.5)

Documentation Presets IP Location Import XPS Settings

Page Navigator

- Zynq Block Design
- PS-PL Configuration
- Peripheral I/O Pins
- MIO Configuration**
- Clock Configuration
- DDR Configuration
- SMC Timing Calculation
- Interrupts

MIO Configuration [Summary Report](#)

Bank 0 I/O Voltage: LVCMOS 3.3V Bank 1 I/O Voltage: LVCMOS 1.8V

Search:

Peripheral	IO	Signal	IO Type	Speed	Pullup	Direction
I/O Peripherals						
> <input checked="" type="checkbox"/> ENET 0	MIO 16 .. 27					
> <input type="checkbox"/> ENET 1						
<input type="checkbox"/> USB 0						
<input type="checkbox"/> USB 1						
> <input type="checkbox"/> SD 0						
> <input type="checkbox"/> SD 1						
> <input type="checkbox"/> UART 0						
> <input checked="" type="checkbox"/> UART 1	MIO 48 .. 49					
<input type="checkbox"/> I2C 0						
<input type="checkbox"/> I2C 1						
> <input type="checkbox"/> SPI 0						
> <input type="checkbox"/> SPI 1						
> <input type="checkbox"/> CAN 0						
> <input type="checkbox"/> CAN 1						
GPIO						
Application Processor Unit						
<input type="checkbox"/> Timer 0						
<input type="checkbox"/> Timer 1						
<input type="checkbox"/> Watchdog						
> Programmable Logic Test and Debug						

OK Cancel

Click on the Clock Configuration and

Click **OK**.


The configuration form will close and the block diagram will be updated as shown *below*:



Answer question #1 in Part A report

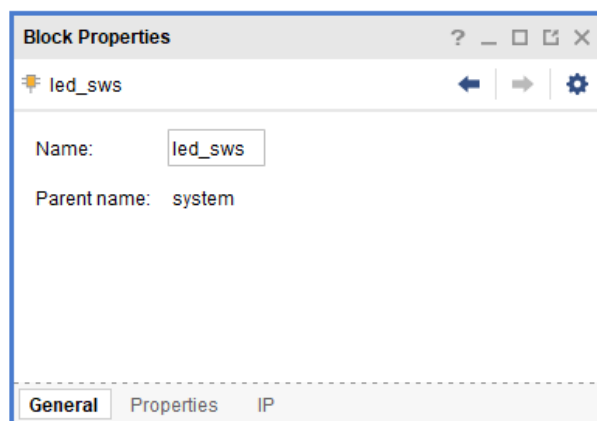
Now we will add additional peripherals (LEDs, SWITCHES, KEYPAD and OLED display) to our design.

We will use GPIO (General Purpose I/O), which is simply a register connected to AXI bus, for connecting LEDs and SWITCHES.

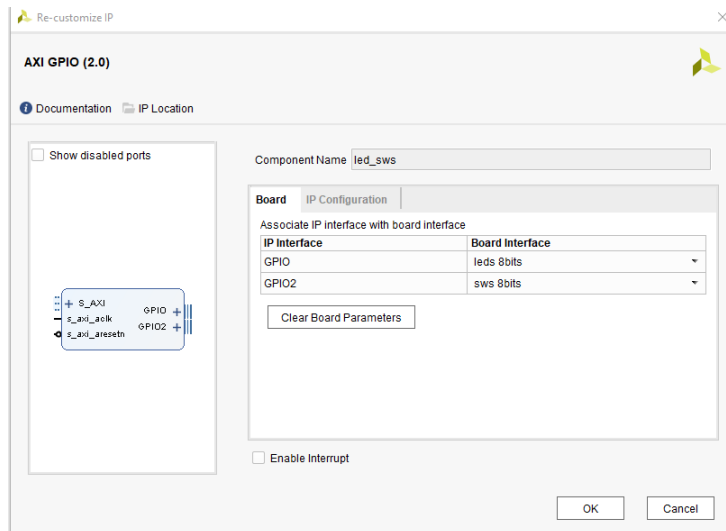
Click the Add IP icon  and search for **AXI GPIO** in the catalog.

Double-click the **AXI GPIO** to add an instance of the core to the design.

Click on the **AXI GPIO** block to select it, and in the *Block properties* tab, change the name to **led_sws**.



Double click on the AXI GPIO block to open the customization window. Under Board Interface, for GPIO, click on **Custom** to view the dropdown menu options, and select **leds 8bits** for **GPIO** and **sws 8Bits** for **GPIO2**.

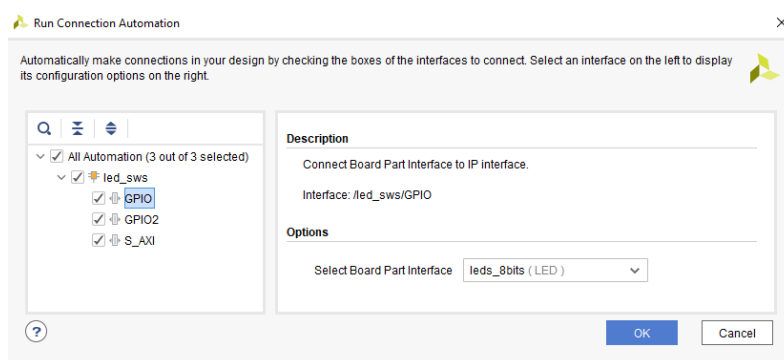


As the Zedboard was selected during the project creation, and a board support package is available for this boards, Vivado has knowledge of available resources on the board.

Click the IP Configuration tab. Notice the GPIO Width is set 8 and is greyed out. If no board support package was available, the width of the IP could be configured here.


Click OK to finish configuring the GPIO and to close the Re-Customize IP window.

Click on Run Connection Automation, and select **led_sws** (which will include GPIO, GPIO2 and S_AXI)

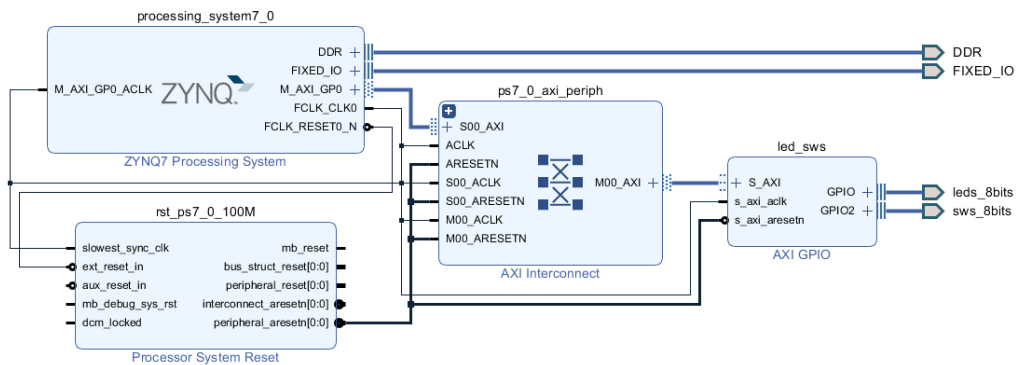


Click OK to automatically connect the S_AXI interface to the Zynq GP0 port (through the AXI interconnect block), and the GPIO port to an external interface.

Notice that after block automation has been run, two additional blocks Processor System Reset, and AXI Interconnect have automatically been added to the design.

Click on Regenerate Layout icon 

The block diagram will be updated as shown below:



Next step we will add two additional cores: **OLED** and **Keypad**. Both of them are not part of the Xilinx IP catalog and need to be added manually.


You will receive a ready-made core for OLED.

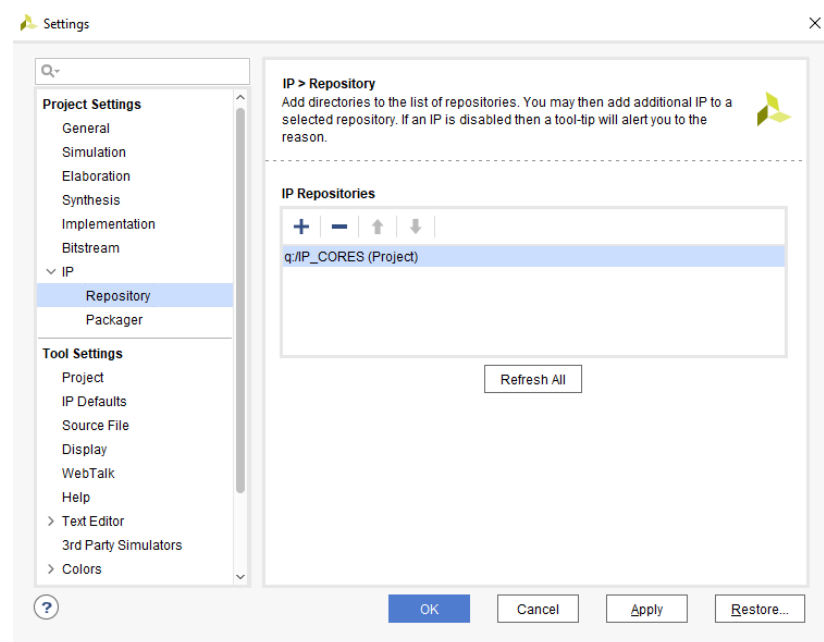
We will create Keypad core with Xilinx Create and package IP tool using the code from the preparation report.

To add the OLED core we need to add the core location to IP repositories search path.

Click on **Tools-> Settings** and expand IP.

Click on **Repository**.


Press the  button to add an OLED core path to the repository. Browse to Q:\IP_CORES click on select and OK.



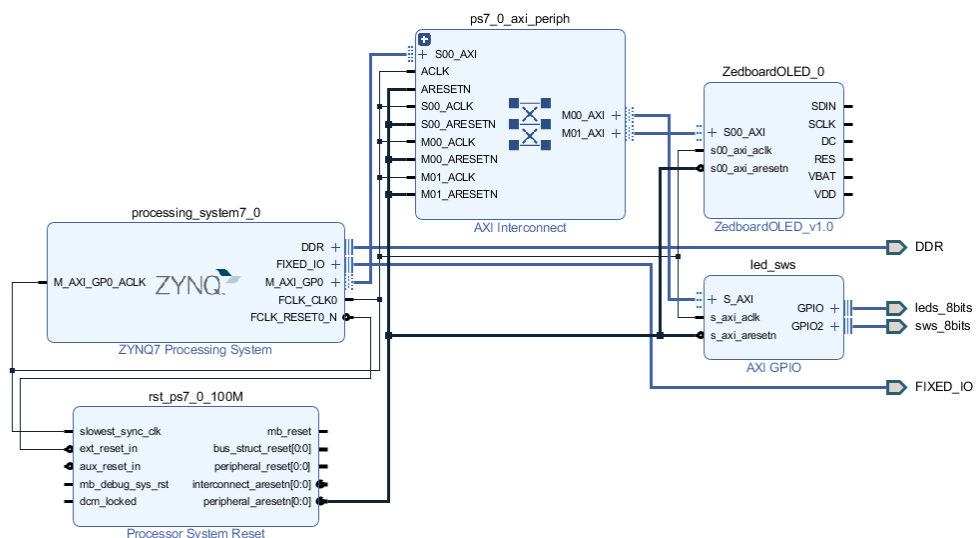
Now instantiate OLED by clicking the Add IP icon, search for **OLED** and double-click the **ZedboardOLED_v1.0** to add an instance of the core to the design.

Run Connection Automation to connect the core to AXI bus.

Notice that the AXI Interconnect block has the second master AXI (M01_AXI) port added and connected to the S00_AXI of the **ZedboardOLED_0**.

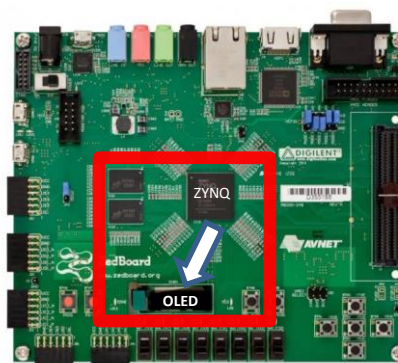
Click on Regenerate Layout icon 

The block diagram will be updated as shown below:



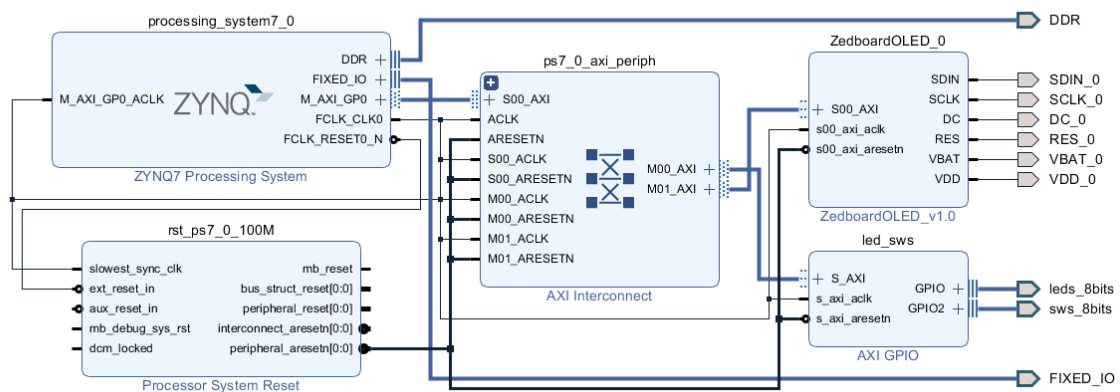
Notice that 6 ports (SDIN SCLK, DC, RES, VBAT and VDD) remained unconnected.

Those pins need to be connected to external OLED display on Zedboard.



Click on the **ZedboardOLED_0** instance, and select Make External to create an external ports.

At this stage the design should look like as shown below.



Now we will create a custom IP block in Vivado with AXI interface and modify its functionality by integrating custom System Verilog code.

Our custom IP will implement a Keypad controller which our processor will be able to access through register reads and writes over an AXI bus.

The block diagram and the functionality of the Keypad Controller AXI4 peripheral we are going to create:

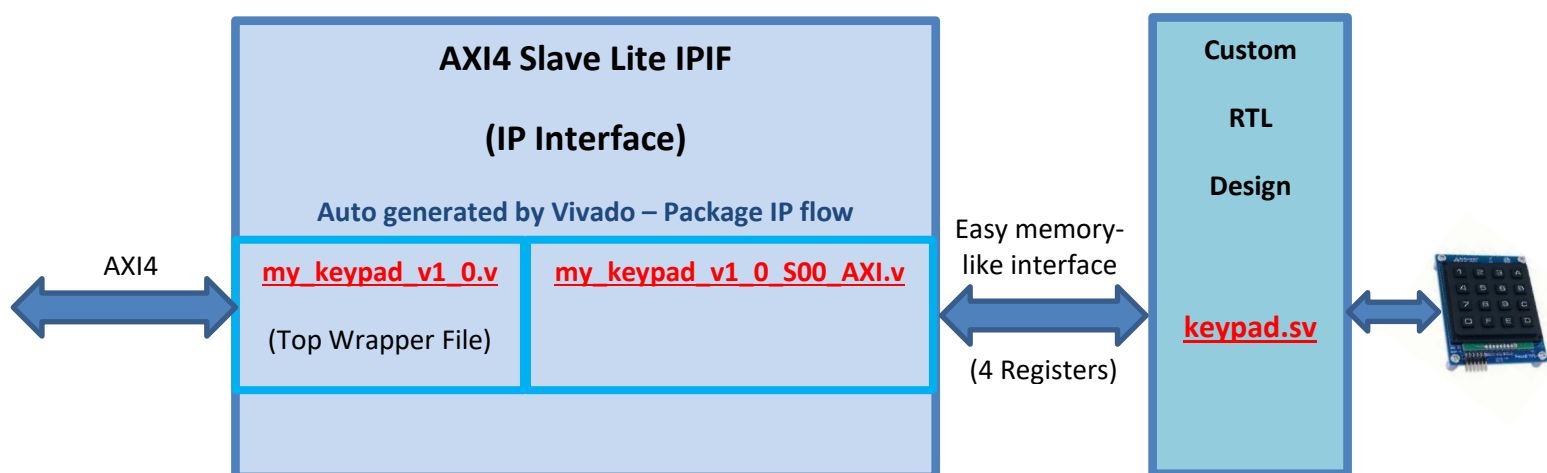
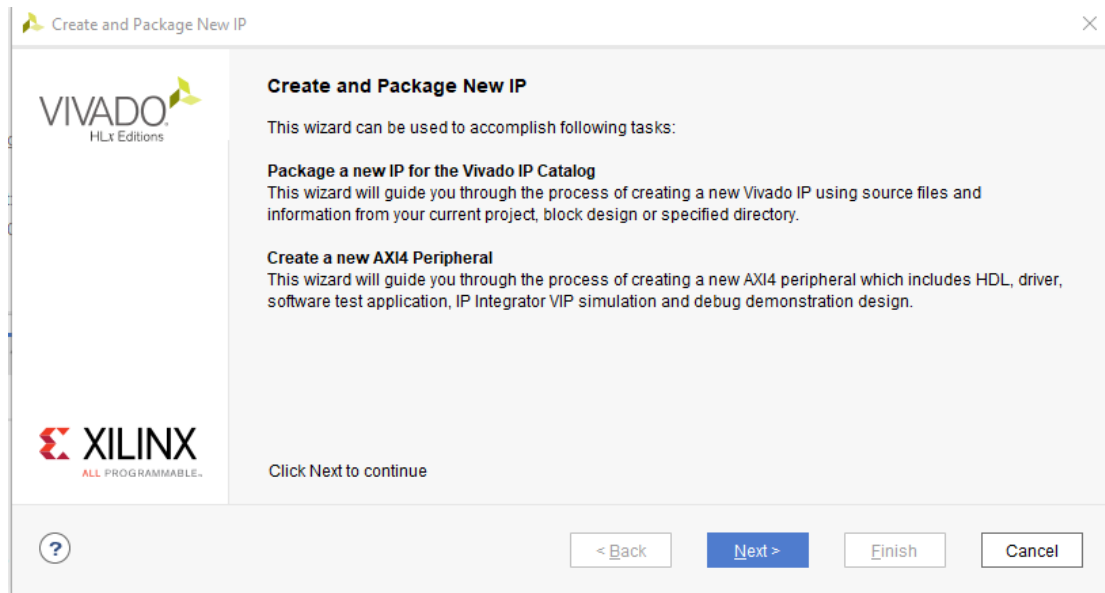


Figure 5.1

Answer question #2

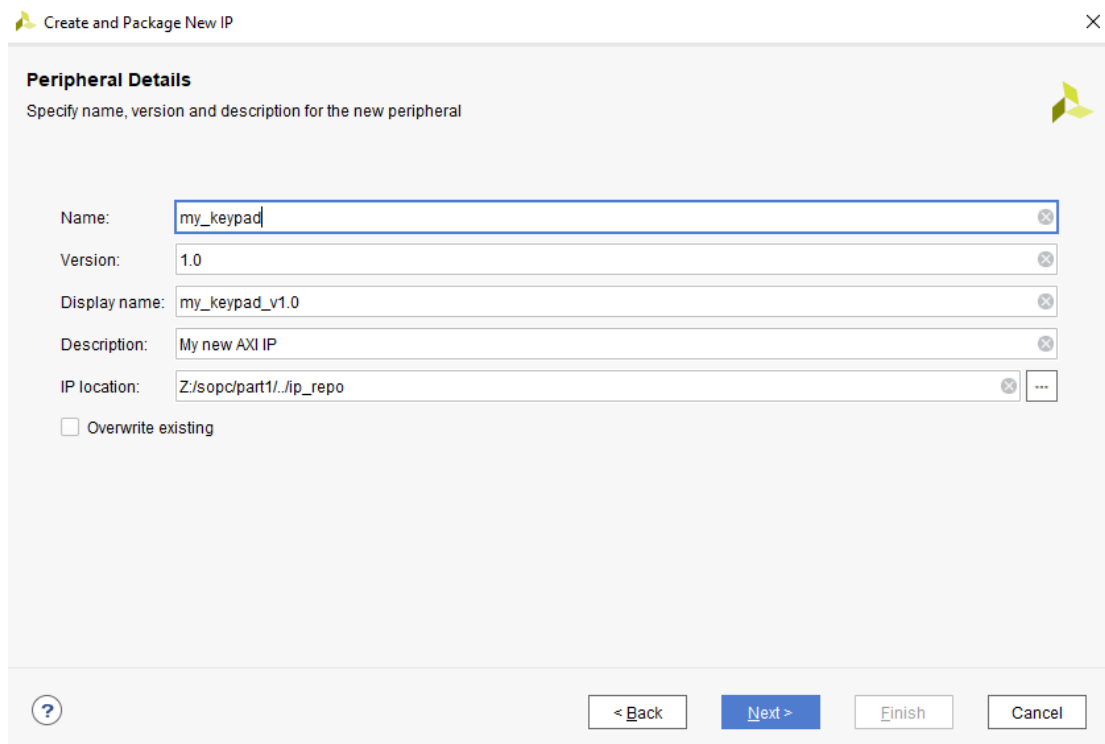
Select **Tools->Create and package New IP**



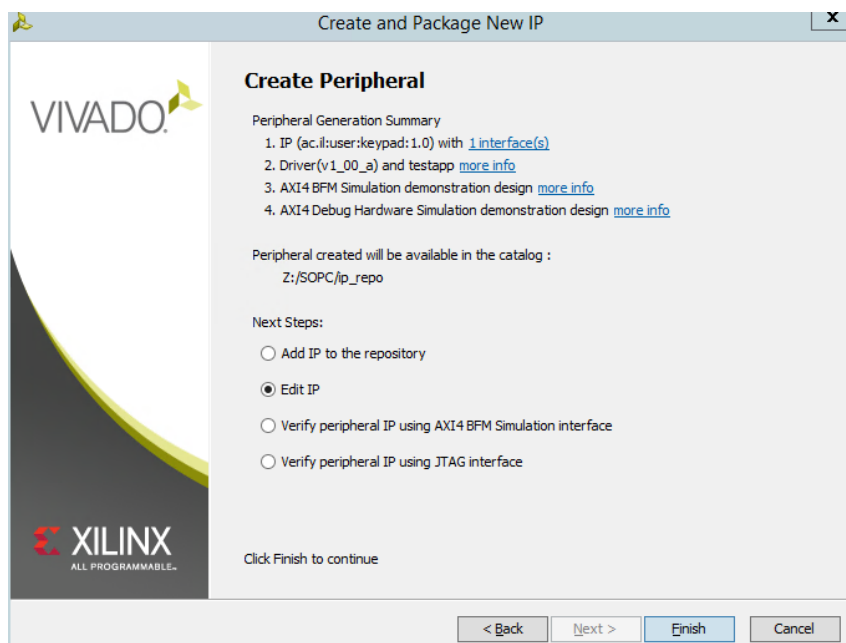
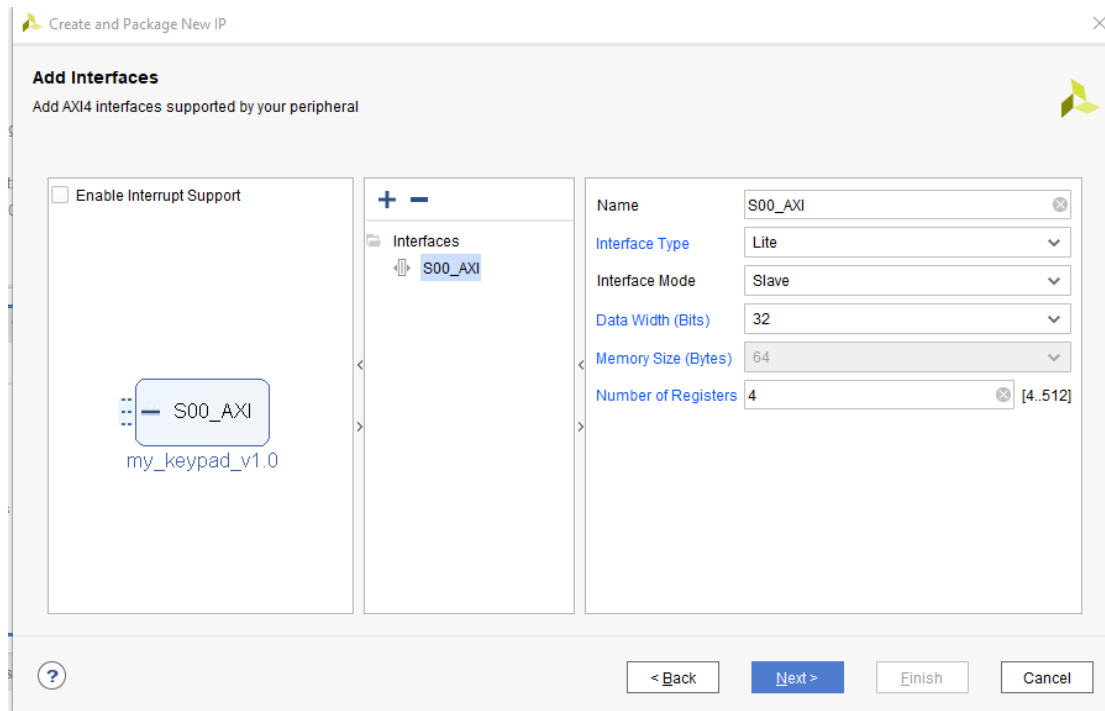
Click **Next**.

On the next page, select **Create a new AXI4 peripheral**. Click **Next**.

Now give the peripheral an appropriate name, description and location. Click **Next**.

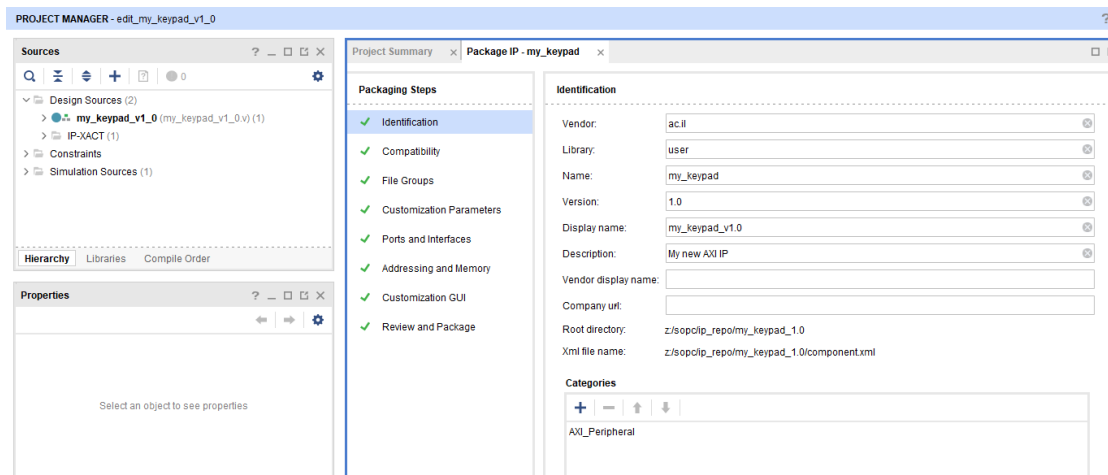


On the next page we can configure the AXI bus interface. For the keypad we'll use AXI lite, and it'll be a slave to the PS, so we'll use default values.



On the last page, select **Edit IP** and click **Finish**.

Another Vivado window will open which will allow you to modify the peripheral that we just created.

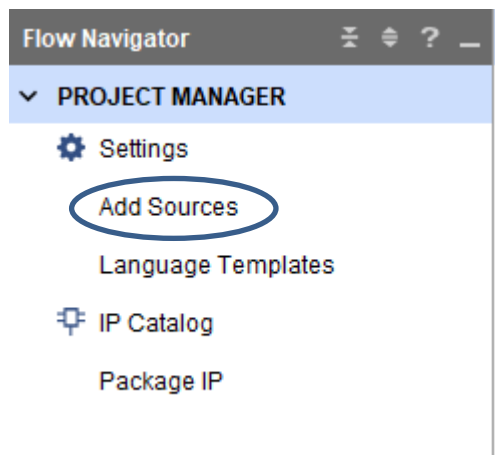


At this point, the peripheral that has been generated by Vivado is an AXI lite slave that contains 4 x 32 bit read/write registers. We want to add our keypad code to the IP and modify it so that two of the registers connects to the keypad outputs.


Use the “keypad.sv” file from the preparation report.

Note that these steps must be done in the Vivado window that contains the peripheral we just created (not the base project).

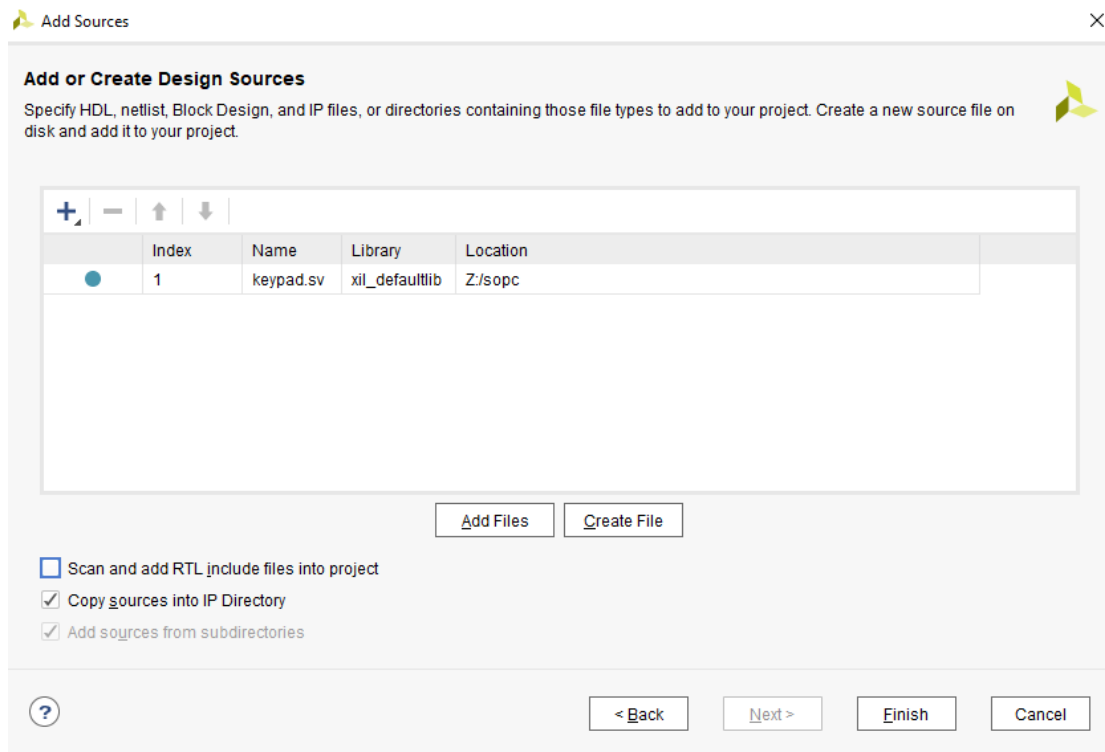
From the Flow Navigator, click **Add Sources**.



Select **Add or Create Design Sources** and click “Next.”

On the next window, click  “Add Files”.

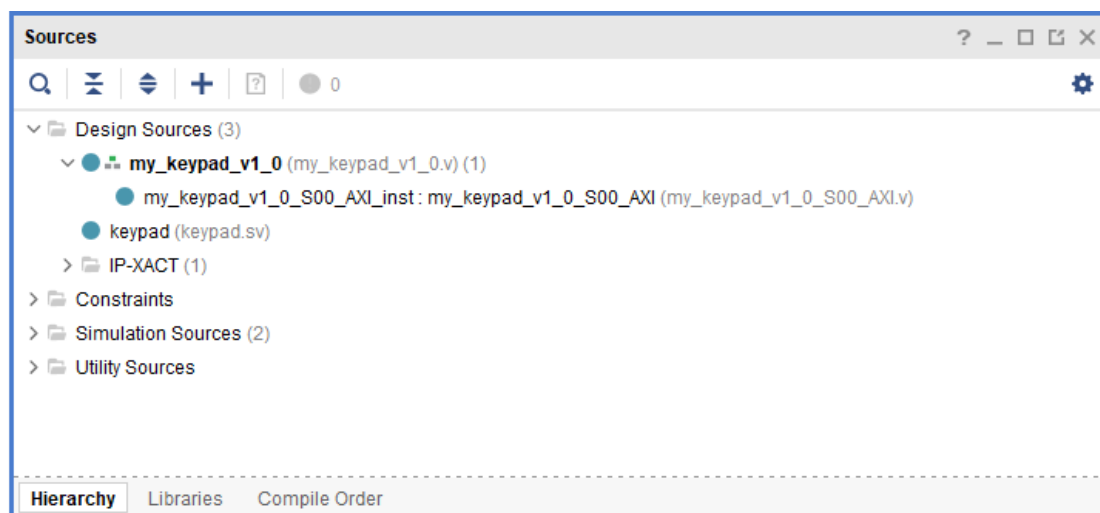
Browse to the “keypad.sv” file, select it and click “OK.”



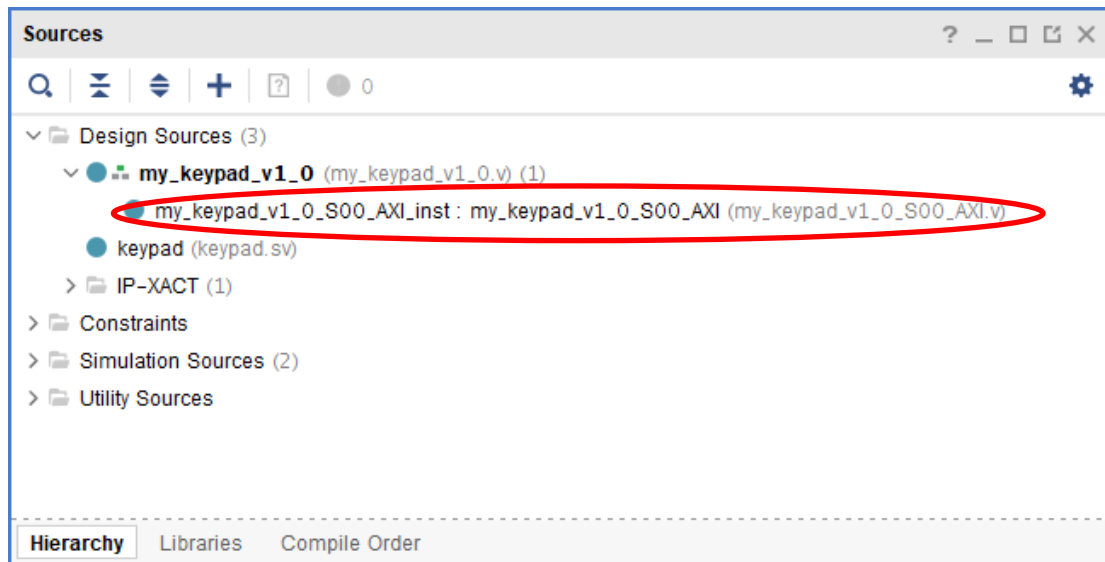
Make sure you tick **Copy sources into IP directory** and then click **Finish**.

The keypad code is now added to the peripheral, however we still have to instantiate it in the AXI lite slave template and connect it to the registers.

At this point, your Project Manager Sources window should look like the following:



Double click on **my_keypad_v1_0_S00_AXI_inst (my_keypad_v1_0_S00_AXI.v file)** to open it.



The source file should be open in Vivado.

Answer question #3a

Find the line that says “-- Users to add ports here” and add the Row and the Col ports that will be connected to external ports in the design.

```

17 | // Users to add ports here
18 | input wire [3:0] Row,
19 | output wire [3:0] Col,
20 | // User ports ends
21 | // Do not modify the ports beyond this line
22 |
23 | // Global Clock Signal
24 | input wire S_AXI_ACLK,
25 | // Global Reset Signal. This Signal is Active LOW
26 | input wire S_AXI_ARESETN,

```

Pay attention to uppercase and lowercase.

Answer question #3b

Find this lines of code “reg_data_out <= slv_reg0;” and “reg_data_out <= slv_reg1;” those lines writes to internal registers.

Add the key and released signal declarations:

```

366 // Implement memory mapped register select and read logic generation
367 // Slave register read enable is asserted when valid address is available
368 // and the slave is ready to accept the read address.
369 assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
370
371 wire [31:0] key;
372 wire [31:0] released;
373
374 always @(*)
375 begin
376     // Address decoding for reading registers
377     case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )

```

Replace the “reg_data_out <= slv_reg0;” and “reg_data_out <= slv_reg1;” with:

```

371 wire [31:0]key;
372 wire [31:0]released;
373 always @(*)
374 begin
375     // Address decoding for reading registers
376     case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
377         2'h0 : reg_data_out <= key;
378         2'h1 : reg_data_out <= released;
379         2'h2 : reg_data_out <= slv_reg2;
380         2'h3 : reg_data_out <= slv_reg3;
381         default : reg_data_out <= 0;
382     endcase
383 end
384

```

Find the line that says “– Add user logic here” and add the following code below it to instantiate the keypad:

```

403
404 // Add user logic here
405
406 keypad keypad0(
407     .clk(S_AXI_ACLK), // 100MHz onboard clock
408     .Row(Row),        // Rows on KYPD
409     .Col(Col),        // Columns on KYPD
410     .key(key),
411     .released(released)
412 );
413
414 // User logic ends
415
416 endmodule
417

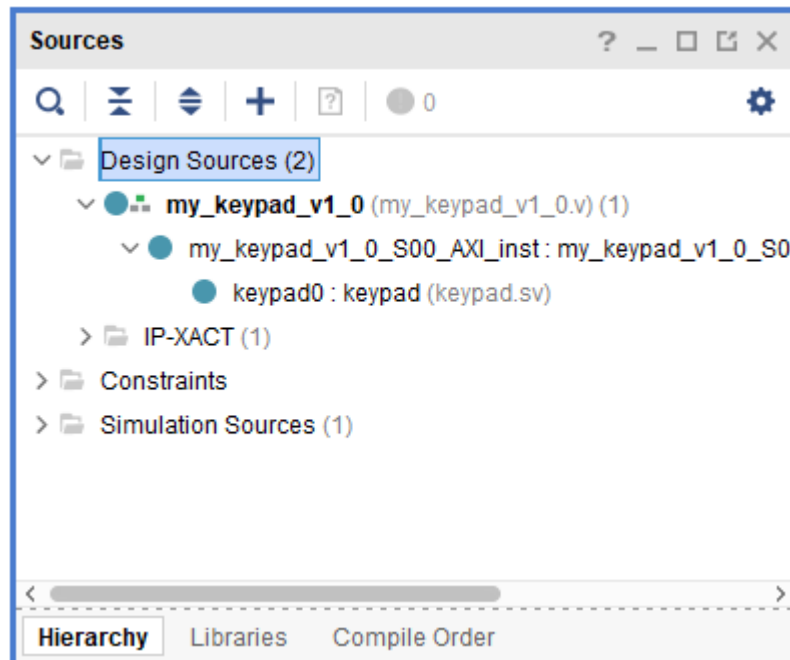
```

Save the file.

Fix all syntax errors.

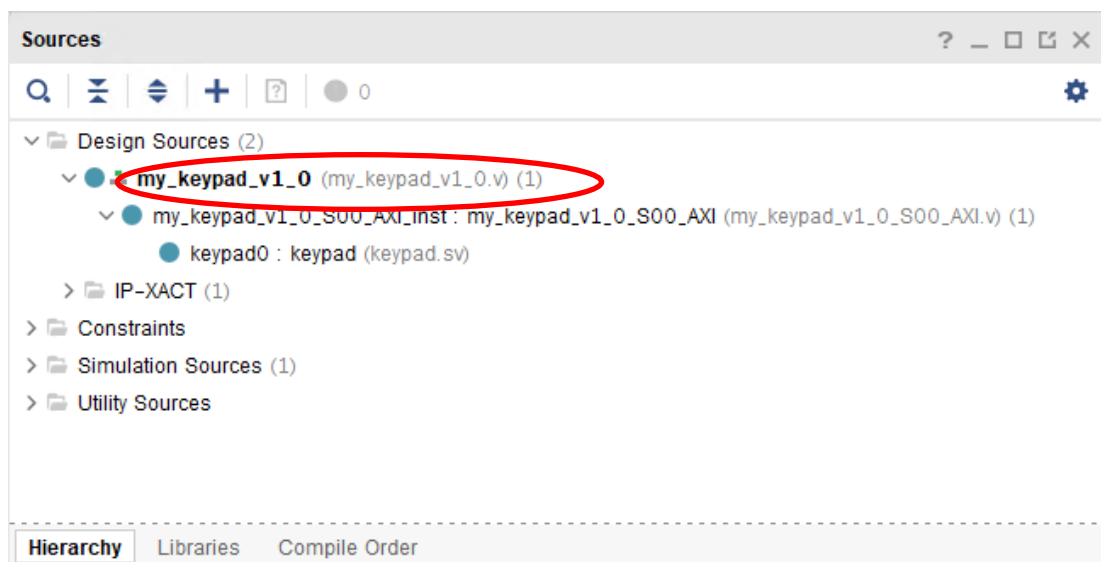
Answer questions #3c-d in Part A report

You should notice now that the “keypad.v” file has been integrated into the hierarchy because we have instantiated it from within the peripheral.



Since we changed the **my_keypad_v1_0_S00_AXI** interface we need to update the **my_keypad_v1_0.v** file.

Double click on the “**my_keypad_v1_0 (my_keypad_v1_0.v)**” file to open it.



Find the line that says “–Users to add ports here” and add the Row and the Col ports that will be connected to external ports in the design.

```

17 // Users to add ports here
18 input wire[3:0] Row,
19 output wire[3:0] Col,
20 // User ports ends
21 // Do not modify the ports beyond this line
22
23
24 // Ports of Axi Slave Bus Interface S00_AXI
25 input wire s00_axi_aclk,

```

Add the Row & Col ports to my_keypad_v1_0_S00_AXI component instantiation:

```

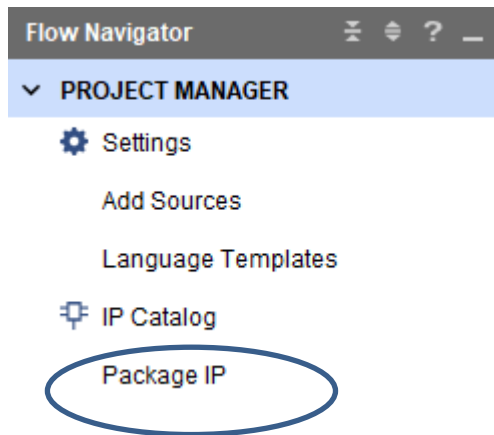
47 // Instantiation of Axi Bus Interface S00_AXI
48 my_keypad_v1_0_S00_AXI # (
49     .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
50     .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
51 ) my_keypad_v1_0_S00_AXI_inst (
52     .Row(Row),
53     .Col(Col),
54     .S_AXI_ACLK(s00_axi_aclk),
55     .S_AXI_ARESETN(s00_axi_aresetn),
56     .S_AXI_AWADDR(s00_axi_awaddr),
57     .S_AXI_AWPROT(s00_axi_awprot),
58     .S_AXI_AWVALID(s00_axi_awvalid),
59     .S_AXI_AWREADY(s00_axi_awready),
60     .S_AXI_WDATA(s00_axi_wdata),
61     .S_AXI_WSTRB(s00_axi_wstrb),
62     .S_AXI_WVALID(s00_axi_wvalid),
63     .S_AXI_WREADY(s00_axi_wready),
64     .S_AXI_BRESP(s00_axi_bresp),
65     .S_AXI_BVALID(s00_axi_bvalid),
66     .S_AXI_BREADY(s00_axi_bready),
67     .S_AXI_ARADDR(s00_axi_araddr),
68     .S_AXI_ARPROT(s00_axi_arprot),
69     .S_AXI_ARVALID(s00_axi_arvalid),
70     .S_AXI_ARREADY(s00_axi_arready),
71     .S_AXI_RDATA(s00_axi_rdata),
72     .S_AXI_RRESP(s00_axi_rresp),
73     .S_AXI_RVALID(s00_axi_rvalid),
74     .S_AXI_RREADY(s00_axi_rready)
75 );

```

Save the file.

Fix all syntax errors.

Open the **Package IP tab**

The image shows a configuration window titled 'Package IP - my_keypad'. On the left, there is a 'Packaging Steps' sidebar with a list of steps: 'Identification' (checked with a green tick), 'Compatibility' (checked with a green tick), 'File Groups' (with a red 'X' icon), 'Customization Parameters' (with a red 'X' icon), 'Ports and Interfaces' (with a red 'X' icon), 'Addressing and Memory' (checked with a green tick), 'Customization GUI' (with a red 'X' icon), and 'Review and Package' (with a red 'X' icon). The main area is titled 'Identification' and contains several fields: 'Vendor:' (ac.il), 'Library:' (user), 'Name:' (my_keypad), 'Version:' (1.0), 'Display name:' (my_keypad_v1.0), 'Description:' (My new AXI IP), 'Vendor display name:' (empty), 'Company url:' (empty), 'Root directory:' (z:/sopc/ip_repo/my_keypad_1.0), and 'Xml file name:' (z:/sopc/ip_repo/my_keypad_1.0/component.xml). Below these fields is a 'Categories' section with a list of categories: '+', '-', '↑', '↓', and 'AXI_Peripheral'.

Click on **File Groups**.

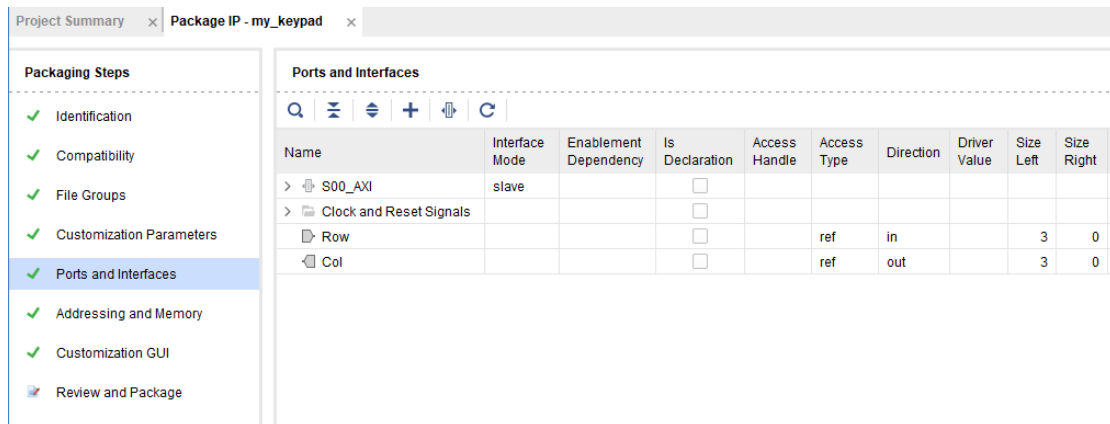
Click the **Merge changes from File Group Wizard** link.

The "File Groups" should now have a green tick.

Click on **Ports and Interfaces** in the Package IP tab of the Project Manager.

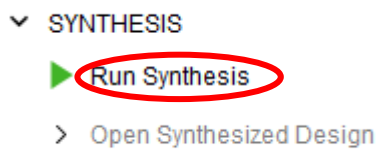
Click the **Merge changes from Ports and Interfaces Wizard** link.

If needed fix all syntax errors.



Notice that Col and Row pins have been added to the core interface.

To check syntax errors run Synthesis.

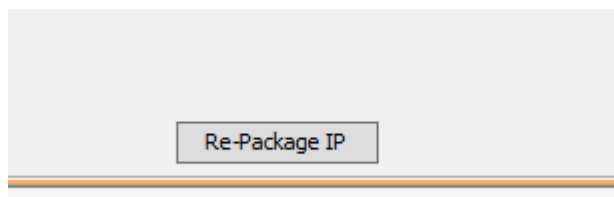


Fix all errors if needed.

When Synthesis finishes successfully click **Cancel**.

Now click “**Review and Package IP**”.

Now click “**Re-package IP**”.



The peripheral will be packaged.

Choose “**Close the project**”.

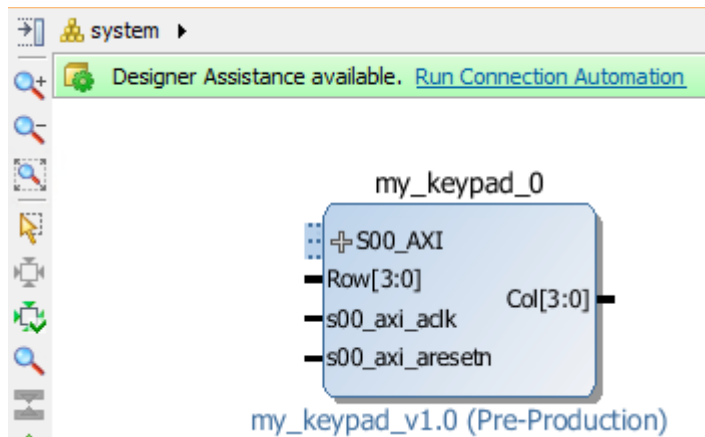
We should now be able to find our IP in the IP catalog.

The rest will be done from the original Vivado window.

Now we will add the keypad IP to the design.

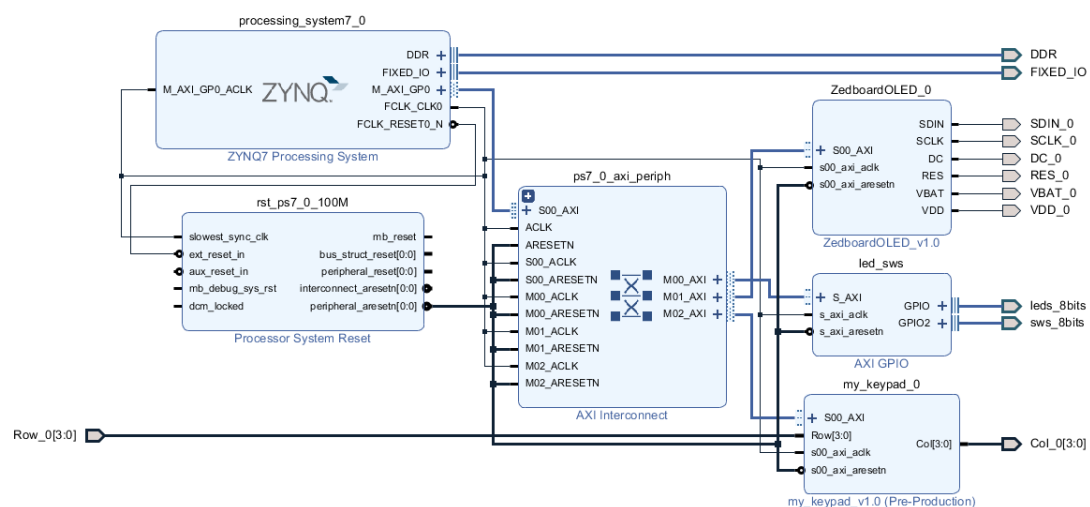
Click the “Add IP” icon, find the “my_keypad” IP and double click on it.

The block should appear in the block diagram and you should see the message “**Designer Assistance available. Run Connection Automation**”. Click the connection automation link.



Right-click on the **Row** and **Col** pins of the **my_keypad_0** instance, and select Make External to create external ports.

The new block diagram should look like this:



Now we will verify that the addresses assigned to all peripheral instances and validate the design for no errors.

Select the **Address Editor** tab and check that the addresses are assigned to the switches, leds, OLED and keypad.

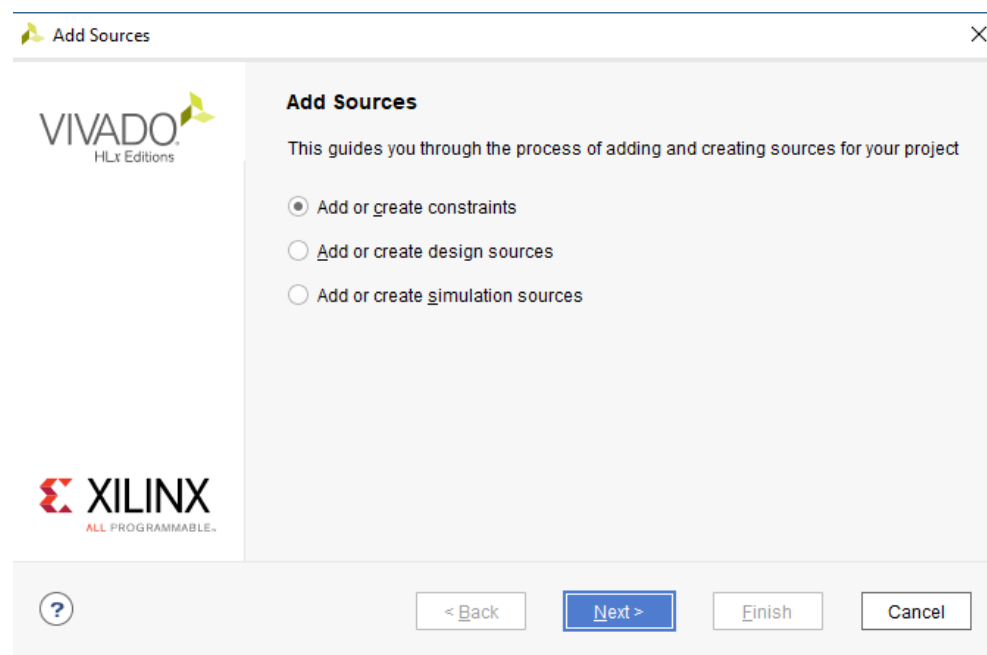
Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
led_sws	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
ZedboardOLED_0	S00_AXI	S00_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF
my_keypad_0	S00_AXI	S00_AXI_reg	0x43C1_0000	64K	0x43C1_FFFF

Select **Tools > Validate Design** to run the design rule checker and to make sure that there are no design errors.

Select **File > Save Block Design** to save the design.

Now we need to define the pin location constraints for OLED and Keypad external pins.

Click the **Add Sources** button in the Flow Navigator and select **Add or create constraints**, and click **Next**.

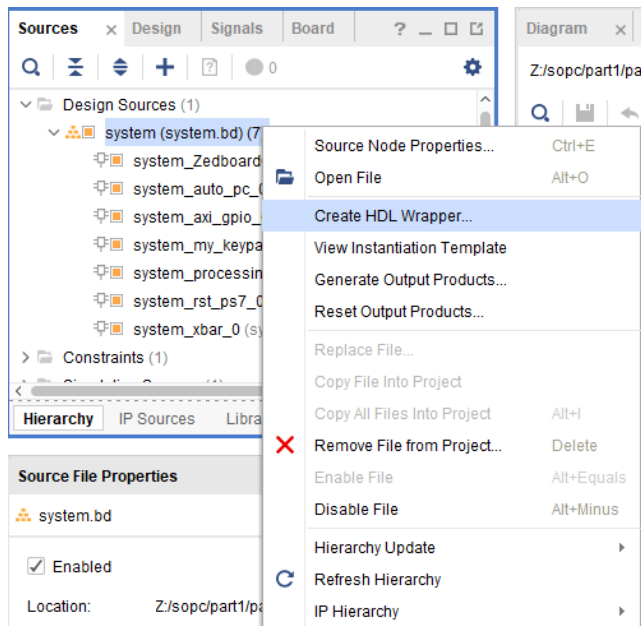


Add the provided Design Constraints file - Q:\part1\part1.xdc

Select **Copy constraints files into project** and click **Finish**

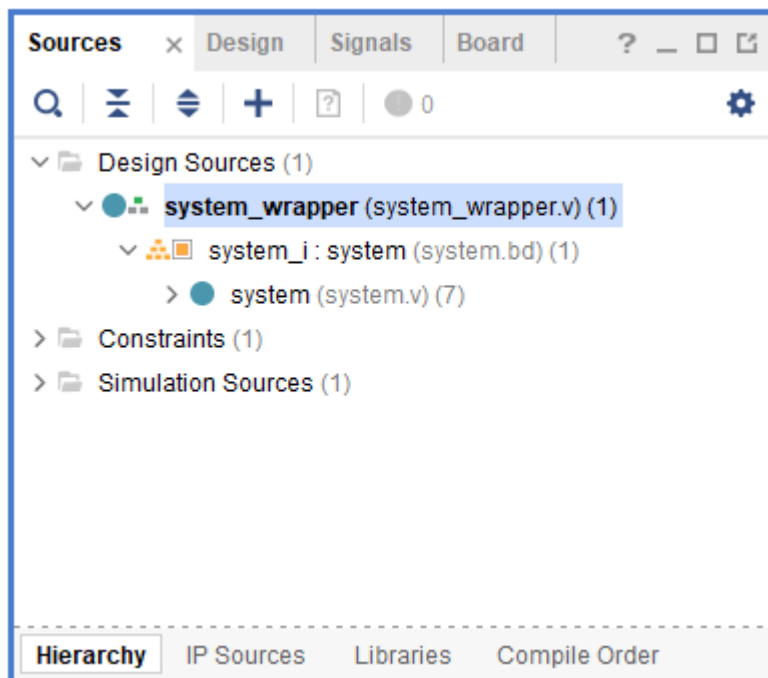
Now create the top-level HDL of the embedded system.

In Vivado, select the Sources tab, expand the Design Sources, right-click the **system.bd** and select **Create HDL Wrapper**:



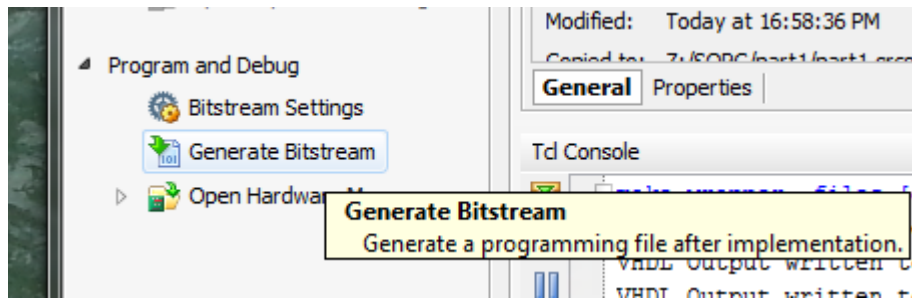
Click OK when prompted to allow Vivado to automatically manage this file.

The wrapper file, system_wrapper.v, is generated and added to the hierarchy. The wrapper file will be displayed in the Auxiliary pane.



At this point we finished creating our hardware design and need to generate the Bitstream to load the FPGA.

Click on the **Generate Bitstream** in the Flow Navigator pane to synthesize and implement the design, and generate the bitstream. (Click Save and Yes if prompted.)



The compilation takes about 5-7 min.

During the compilation change the part1.c to detect the release signal and print the pressed key only once for each key press.

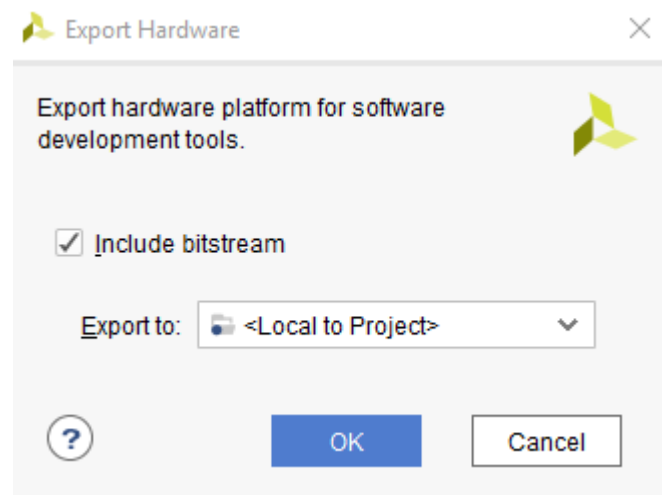
Answer question #4 in Part A report

When the bitstream generation is complete, click **Cancel**.

Software design

Export the hardware configuration by clicking **File > Export > Export Hardware...**

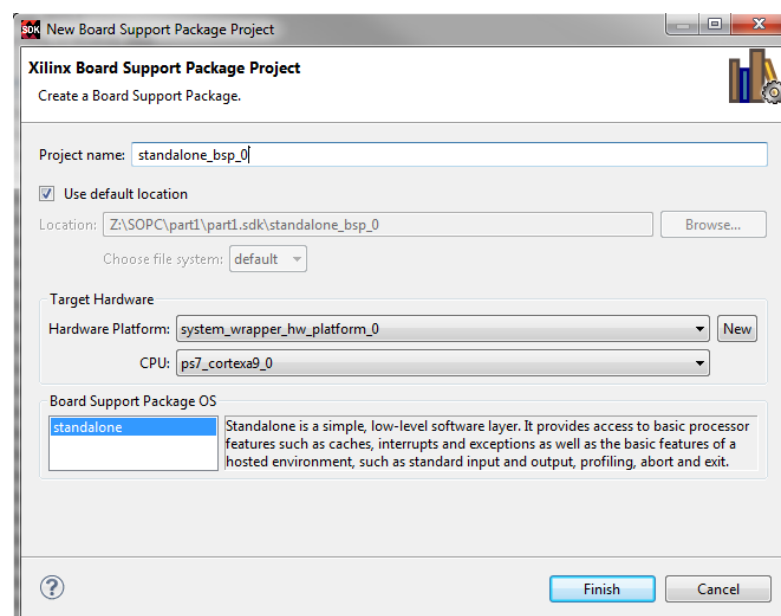
Tick the box to **include the bitstream** and click **OK**.



Launch SDK by clicking **File > Launch SDK** and click **OK**

SDK (Software Development Kit) should open and automatically create a hardware platform project based on the configuration exported from Vivado. A board support package and software application will be created and associated with this hardware platform.

Select **File > New > Board Support Package**



Click **Finish** with the default settings selected (using the Standalone operating system). This will open the Software Platform Settings form showing the OS and libraries selections.

Click **OK** to accept the default settings as we want to create a standalone_bsp_0 software platform project without any additional libraries.

The library generator will run in the background and create the xparameters.h file in the part1.sdk\standalone_bsp_0\ps7_cortexa9_0\include directory.

Answer question #5 in Part A report

Now we are going to create an application project from template and replace the C file with part1.c from preparation report.

Select File > New > Application Project.

In the Project Name field, enter **part1** as the project name.

Select the Use existing option in the **Board Support Package** field.

The screenshot shows the 'New Project' dialog box in the SDK. The dialog is titled 'Application Project' and contains the following fields and options:

- Project name:** part1
- ☒ **Use default location**
- Location:** Z:\sopc\part1\part1.sdk\part1 (with a 'Browse...' button)
- Choose file system:** default
- OS Platform:** standalone
- Target Hardware:**
 - Hardware Platform:** system_wrapper_hw_platform_0 (with a 'New...' button)
 - Processor:** ps7_cortexa9_0
- Target Software:**
 - Language:** C (selected), C++
 - Compiler:** 32-bit
 - Board Support Package:**
 - ☐ **Create New** part1_bsp
 - ☒ **Use existing** standalone_bsp_0

At the bottom of the dialog, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

Click **Next**.

Select the **Hello World template** and click Finish.

The part1 project will be created in the Project Explorer window of SDK.

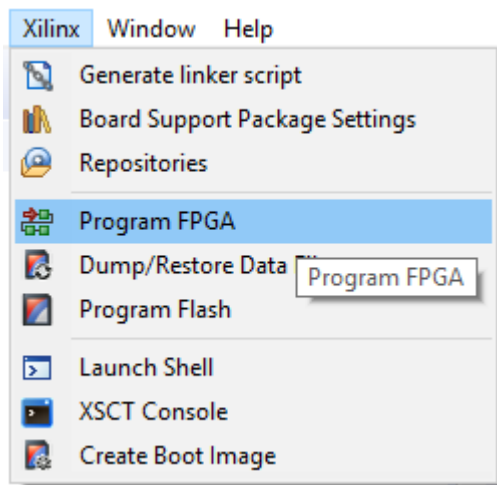
Open the HelloWorld.c and replace its content with content of part1.c file.

Run Project -> Build All

Now we are going to load the FPGA with the bitstream we created and run the part1.c code.

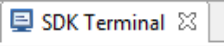
Power up the board.


Program the FPGA by selecting **Xilinx Tools -> Program FPGA**

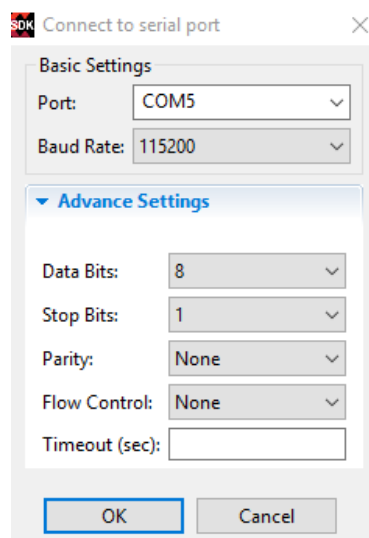


It opens a new window **Program FPGA**, click **Program**.

Establish serial communications using the SDK's Terminal tab

Select the  tab.

Click on  and select appropriate COM port (**depending on your computer**), and configure the terminal with the parameters as shown below.



Select the part1 project in the Project Explorer, right-click and select

Run As > Launch on Hardware to download the application.

You should see the following output on the Terminal console:

```
-- Start of the Program --  
-- Press Keypad to see the output on OLED --  
-- Change slide switches to see corresponding output on LEDs --
```

Change the slide switches and see the corresponding LED turning ON and OFF and press the keypad.

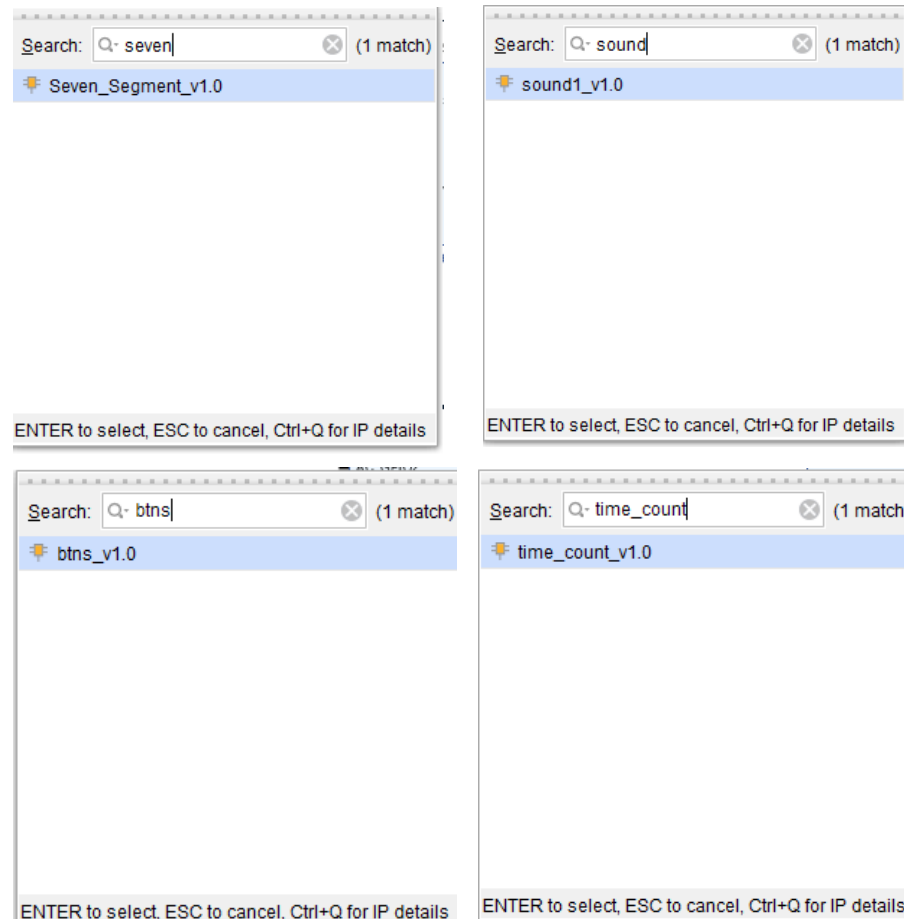
Check the function of detection and printing the key only once for each key press.

Now we will build a system that will use us in Part B of the experiment.

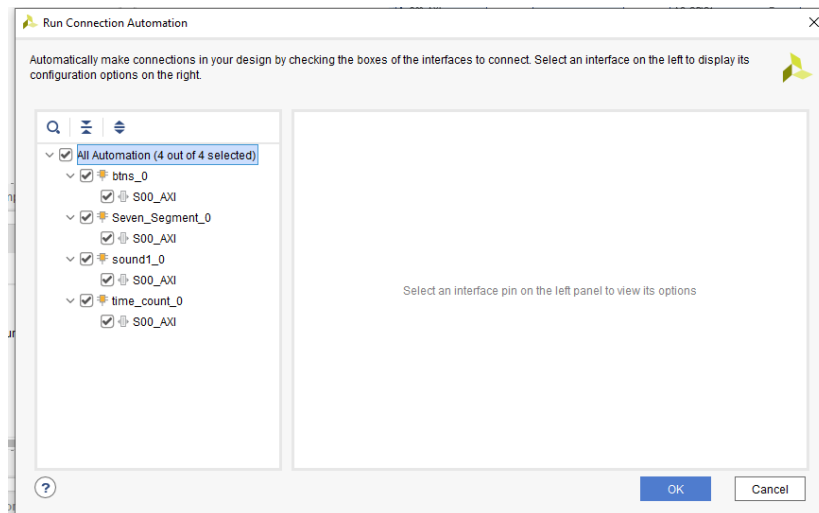
This system will support Ethernet communication and will contain additional peripherals such as 7-SEG, Buttons, Sound and Time Count.

A detailed explanation of the additional peripherals, you will found in Part B preparation report. The goal of this part is just to build and to compile the system.


Click the “Add IP” icon 4 times and add the following peripherals:



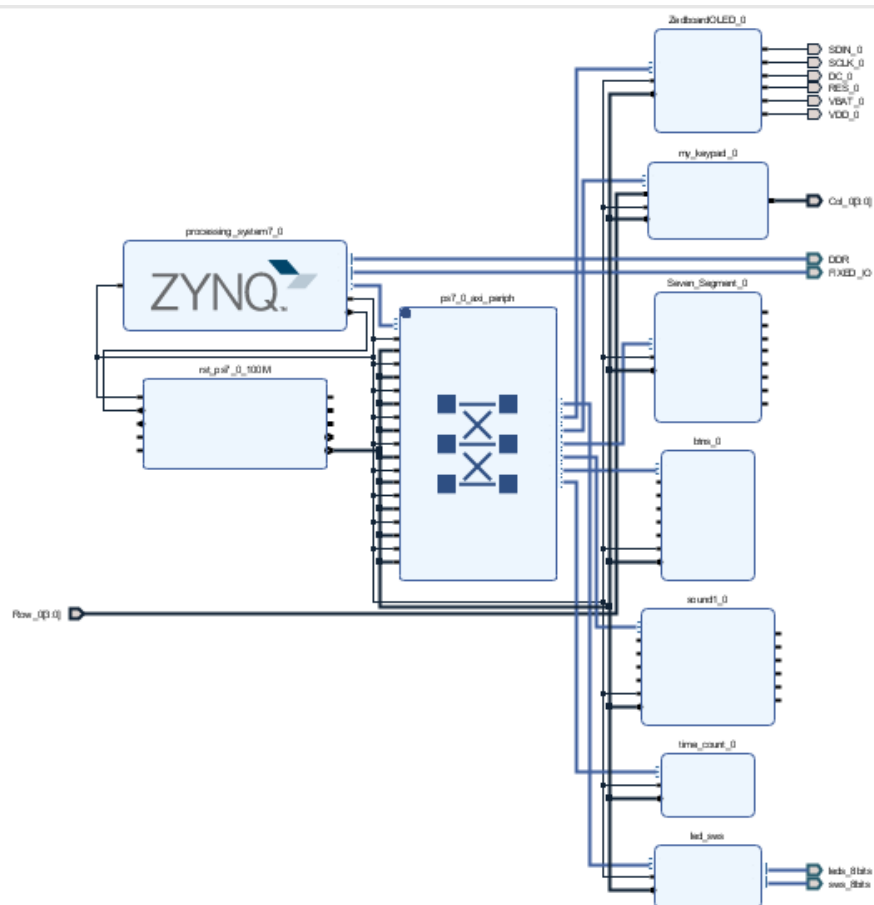
Click on Run Connection Automation, and select all peripherals



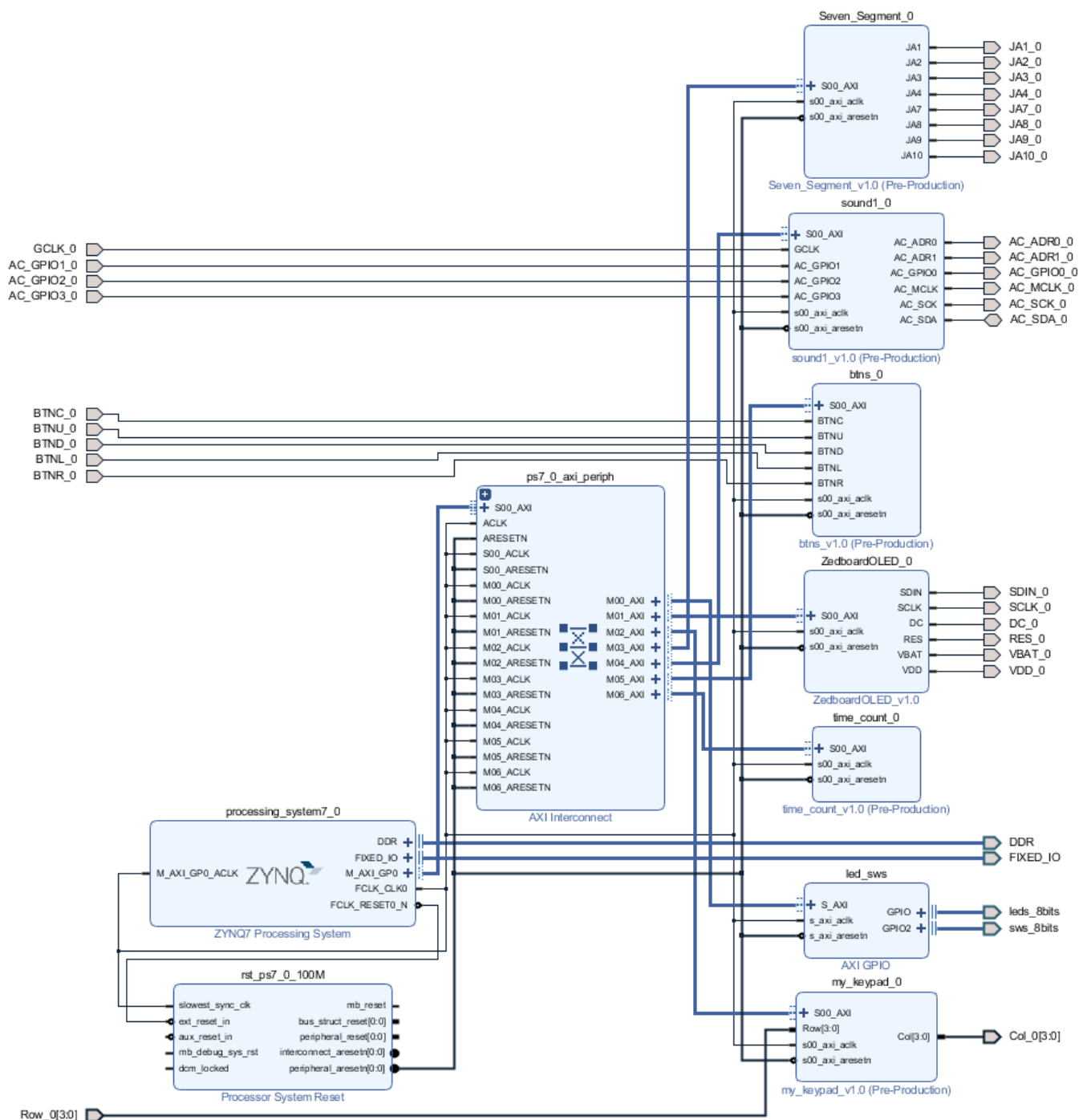
Click OK.

Click on Regenerate Layout icon 

The block diagram will be updated as shown below:

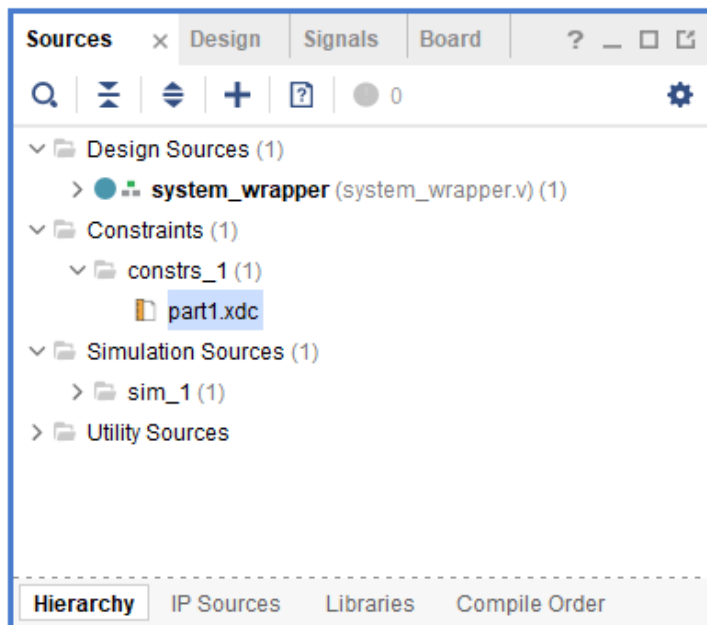


Select the Seven_Segment, Btms and Sound peripherals and select Make External to create external ports:



The last thing is to replace the pin assignments file with the part2.xdc file.

Expand **constrains** in the Sources window:

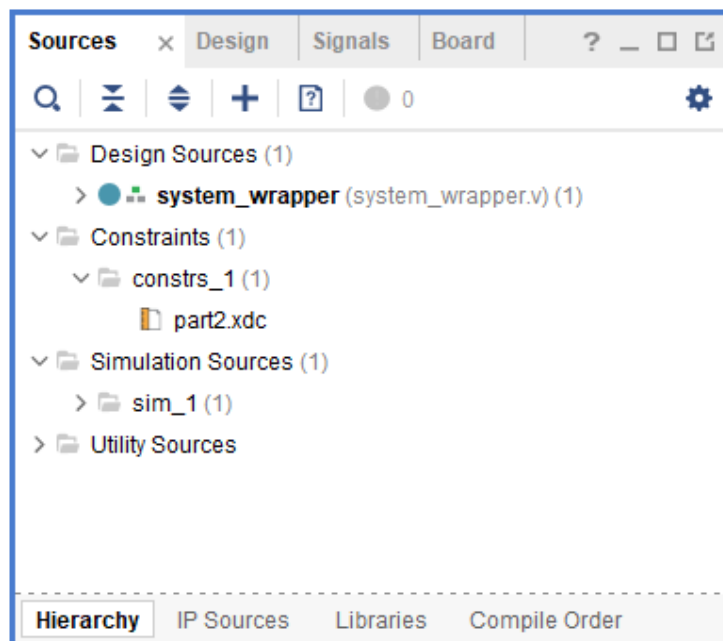


Remove the part1.xdc from the project.

Click the **Add Sources** button in the Flow Navigator and select **Add or create constraints**, and click **Next**.

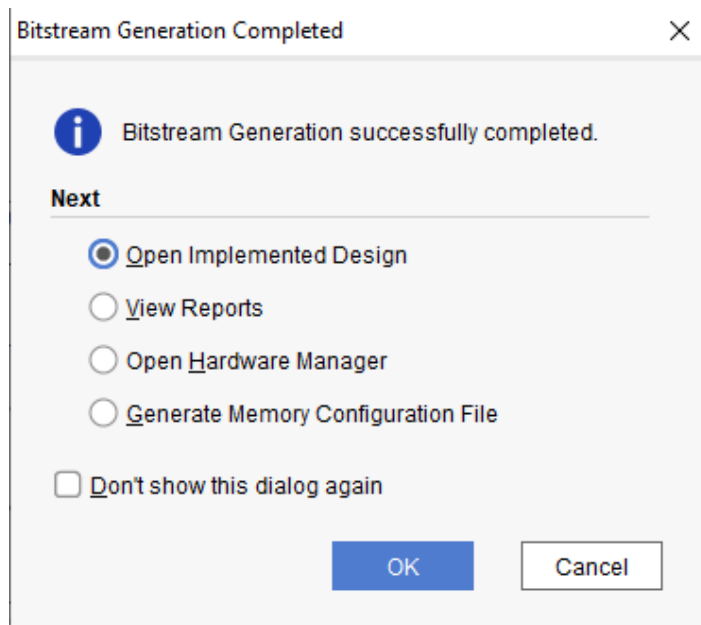
Add the provided Design Constraints file - Q:\part2\part2.xdc

Select **Copy constrains files into project** and click **Finish**.



Click on **Generate Bitstream** to generate the system Bitstream.

After Bitstream generation completed successfully click **Cancel**.



Part B

Smart Home and the Internet of Things

6 Part B – Introduction

In the first part of the Experiment we talked about SoPC's as the enabler for smart devices which are used to build up the Internet of Things. Now a big set of such smart devices are already or will be soon part of our homes. They are used to build the smart home.

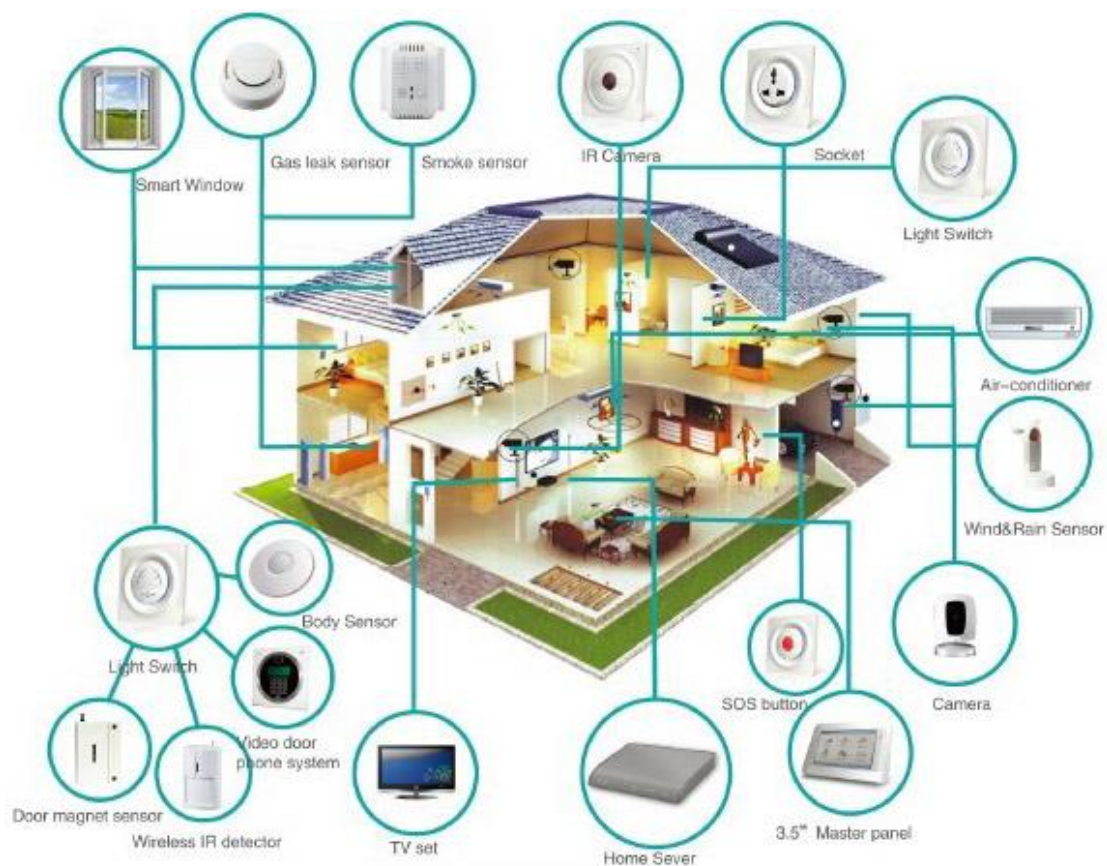


Figure 13 Smart Home Example from <http://smarthomeenergy.co.uk/what-smart-home>

A definition from the same website: "Smart homes use 'home automation' technologies to provide home owners with 'intelligent' feedback and information by monitoring many aspects of a home." If this information can be received by the owner through the internet while he is at some other place, we have included the smart home into the internet. Furthermore, if different smart components 'talk' to other smart component of the house or outside the house via the internet we have established an internet of things with this home.

The smart home may be divided into several categories given in the following with several examples:

- Security
 - Intrusion protection; like door and window open sensors and associated control panel
 - Smoke, CO, gas leak, water leak sensors and alarm control
 - Panic button
 - RFID based automatic door opener
 - Security cameras with image analysis
- Home and energy control
 - Control of AC based on outdoor and indoor temperature, and knowing if or when occupants will be at home
 - Smart Blinds, open and closing based on direction of sun and desired indoor temperature
 - Control of hot water availability
 - Control of lighting (switch off if no one is in the room who needs it)
 - Switch on lights when owner approaches property so he/she will never enter a dark house.
 - Smart shower/bathtub, adjust water temperature according to taste of user
- Smart appliances
 - Smart Refrigerator and Pantry; monitoring food availability / consumption; generation of shopping lists and meal suggestions
 - Smart Microwave; Knows what is going to be cooked, adjust power and time accordingly
 - Smart Washer/Dryer; identifies wash temperature and cycle; warns if clothes are entered which should not be washed together
- Home entertainment
 - Selection what to play on what device in which room
 - Speech recognition driven search of content
 - May suggest content based on time of day and room occupants
 - Streams from owners library or through internet service
- Other smarts
 - In- and outdoor irrigation system using wastewater / recycled water and knowledge about sunlight and humidity around the plants.
 - Smart closet; suggests what to wear based on outdoor weather and owners mood / activity plans

7 Controllers and operating systems

There is very often no operating system running on the processing element of an embedded system. First there might not be enough resources available and secondly there is no necessity if the control program handles only a few devices and can run single threaded. The most ridged form of running such a program is called Bare Metal, here all code including the routines to handle the IO for external devices are combined into a single executable which is pushed into the memory of the processing element and executed from there.

If some more freedom is needed the manufacturer of a controller board will provide a board support package (BSP). This is an implementation of specific support code (software) for a given board (device motherboard). It is commonly built with a bootloader that contains the minimal device support to load device drivers for all the devices on the board. The ZedBoard in conjunction with SDK will build a standalone BSP for a specific configuration.

7.1 Board Support Package BSP

The standalone BSP created by SDK from the Hardware information send over to it from Vivado is based on the Cortex A9 BSP as available from ARM, it supports gcc compilers.

The following lists the Cortex A9 Processor API.

- Cortex A9 Processor Boot Code
- Cortex A9 Processor Cache Functions
- Cortex A9 Processor Exception Handling
- Cortex A9 Processor File Support
- Cortex A9 gcc Errno Function
- Cortex A9 gcc Memory Management
- Cortex A9 gcc Process Functions
- Cortex A9 Processor-Specific Include Files
- Cortex A9 Time Functions

Cortex A9 Processor Boot Code contains a minimal set of code for transferring control from the processor's reset location to the start of the application. It performs the following tasks.

- Invalidate L1 caches, TLBs, Branch Predictor Array, etc.
- Invalidate L2 caches and initialize L2 Cache Controller
- Enable caches and MMU
- Load MMU translation table base address into the TTB registers
- Enable NEON coprocessor

The boot code also starts the Cycle Counter and initializes the Static Memory Controller.

The specific functions are in 'include files', most of them start with Xil_.....

Added to this will be the driver code for each of the devices connected to the AXI busses. They may also provide the ability for interrupts; however the interrupt handler for each device which may cause an interrupt needs to be written by the user.

7.2 Event driven programming

An event-driven program means that the program sits and waits for something to do. This is in opposite to most command-line programs which run with arguments. In other words, a simple editor is an event driven program as it waits for the user to press a key on the keyboard or mouse and then acts on it. E.g. to place the letter just pressed onto the next position in the text to be edited.

On the next higher layer, the operating system has a queue of threads which all wait for events and as soon as the event happens it schedules the specific task which was waiting for the event to execute that portion of the program needed to deal with the event and then give the control back to the operating system.

Not only key-strokes are events programs have to react upon but also packages arriving via the internet or handling of disk read/write. So in a computer system a great number of different threads may be waiting for an even greater number of different events to occur. How the programs are notified depends on the hardware available and the operating system. Basically two methods exist, reading a specific IO address frequently (polling) or waiting for the hardware to send a signal to the processor which forces a specific new instruction address (interrupt).

7.2.1 Polling versus interrupt

If there is a big operating system with a lot of processes all waiting for some events to react upon, than it would be very wasteful to wake up each of the threads, have it check the IO address it is using to see if the event has occurred and if not put it back to sleep. In such an environment using interrupts and interrupt handler which communicate between hardware and program thread is the method of choice.

On the other hand, if you are in a single thread / program environment with a striped down operating system, and there is only a limited number of events which may happen, it might be much less complex to use the polling method. Basically in an endless while structure the IO(s) are read and examined. If the event has happened, the program flow is branched out to do whatever is needed and then returned to continue the while loop polling.

7.3 Ethernet communication in a basic environment

To use the Ethernet to communicate between our controller and the host computer not only requires the hardware connection being enabled in the Zynq processor, it also requires a good deal of software support. In order to send information over the internet require software protocols like the TCP/IP stack.

TCP/IP model

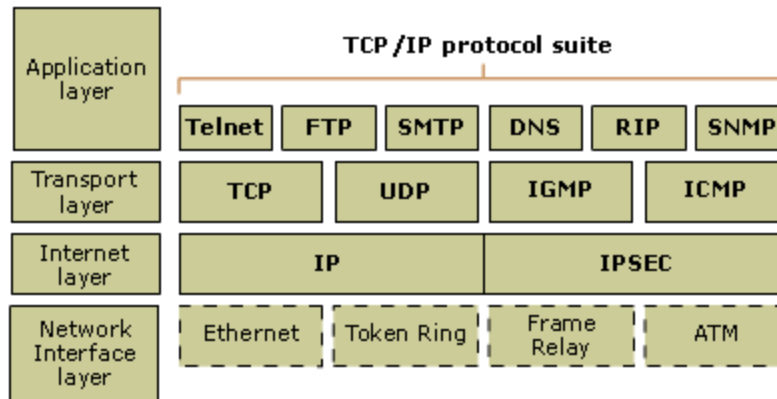


Figure 14 tcp/ip model [from https:// technet.microsoft.com](https://technet.microsoft.com)

In an environment with a full operating system this does not cause any problem; it is just part of the system. In the world of embedded controllers, however, such a stack needs to be provided. A widely used open source TCP/IP stack designed for embedded systems is lwIP (lightweight IP). The focus of the lwIP TCP/IP implementation is to reduce resource usage while still having full-scale TCP functionality. This makes lwIP suitable to be used in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code in ROM.

The lwIP is a standalone stack; there are no operating systems dependencies, although it can be used along with operating systems. It is provided as part of the SDK environment. The 141_v1_1 is an SDK library that is built on the open source lwIP library version 1.4.1. Its Gigabit Ethernet controller and MAC (GigE) core is applicable only for ARM Cortex-A9 system (Zynq®-7000 processor devices).

The lwIP provides support for the following protocols:

- Internet Protocol (IP)
- Internet Control Message Protocol (ICMP)
- User Datagram Protocol (UDP)
- TCP (Transmission Control Protocol (TCP)
- Address Resolution Protocol (ARP)
- Dynamic Host Configuration Protocol (DHCP)
- Internet Group Message Protocol (IGMP) Additional Resources

To read more:

- lwIP wiki: http://lwip.wikia.com/wiki/LwIP_Wiki
- Xilinx® lwIP designs and application examples:
http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf
- lwIP examples using RAW and Socket APIs: <http://savannah.nongnu.org/projects/lwip/>

For an easy use we provide a Host GUI and a set of routines on the ARM processor.

7.3.1 GUI for the Host to Zedboard Ethernet Communication

After host and Zedboard are connected, the GUI provides for the ability to write from host to the board or to initiate reads from the board. Figure 15 depicts the write panel of the GUI in connect state after a time-stamp write has been performed.

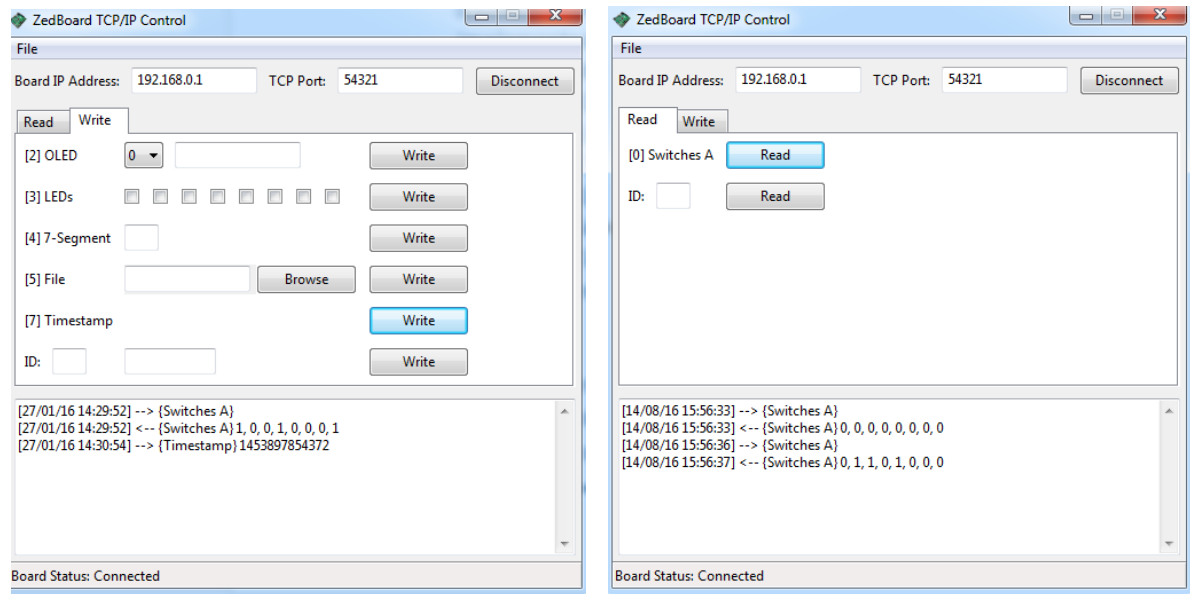


Figure 15 GUI connected; write panel and read panel used in the experiment

For communication the following format for the records to be transferred was defined. The first 4 bytes data[0-3] will contain the length of the entire record send in bytes (not including the 4 length bytes). Byte 4 - data[4] will contain an ID number to identify predefined records or user defined transfers, Rest of the bytes data[5-(length+3)] will contain the payload.

The predefined ID number (data [4]) are:

ID	Sender	Receiver	Action	Data Size[bytes]	Remarks
0	PC	Zedboard	Initiate read from switches	1	
1	currently not implemented				
2	PC	Zedboard	Write to a row on OLED	Max 17	First Byte – line number. Other bytes are chars to be display on the screen.
3	PC	Zedboard	Write to the 8 LEDs	1	
4	PC	Zedboard	Write 2 digits to a Seven Segment Display	1	
5	PC	Zedboard	Write the content of a file to the board	Depends on File size	
6	currently not implemented				
7	PC	Zedboard	Write the timestamp as 64 bit value representing the number of milliseconds past since 1/1/1970	8	

8	Zedboard	PC	Display the message as ASCII text in the GUI message field.	Depends on message size	
9-255	User defined				

Figure 16 predefined ID numbers

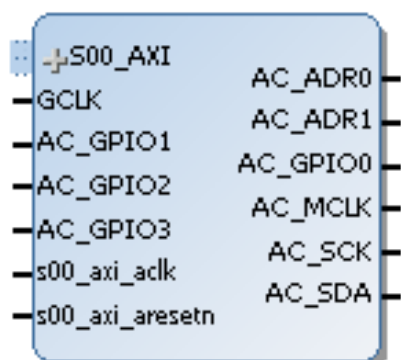
For other actions than pure display of the received values on the host side, the GUI needs to be changed accordingly.

You can open the GUI in the Lab. using the Q:\part2\ZedBoardControl.exe.

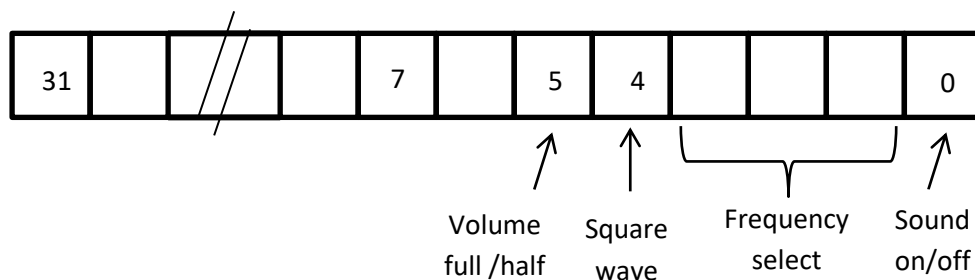
7.4 Sound1 (custom IP)

The ZedBoard carries an audio codec the ADAU1761 from Analog Devices on board. This chip has an I2C interface for setup and another interface for transferring the audio data from/to the codec. The IP enables a specific use of the codec with a simplified interface to the processor.

Beside the control and data interface to the codec, the IP contains a digital sinus generator realized by a Numeric Controlled Oscillator. The oscillator produces selectable sinus signals with approximately one of the following 5 frequencies: 400 Hz, 800 Hz, 1600 Hz, 3200 Hz and 6400 Hz. The interface with the processor through the AXI4-light bus is used for selecting which of those frequencies will be heard.



The AXI light interface to the processor has 4 32 bit registers of which currently only Reg 0 is used. The register has the following layout.



Bit 31 down to 1	frequency
------------------	-----------

000	400 Hz
001	800 Hz
010	1600 Hz
011	3200 Hz
100	6400 Hz

If bit 0 is set to '1' then a signal is sent to the codec. If bit 5 is set to '1' then the signal has half of the possible amplitude. Bits 3 to 1 are coded as following.

Bit 4 select instead of the sinus signals a fixed frequency square wave as input.

Sample invocation: short sound beep at 6.4 KHz :

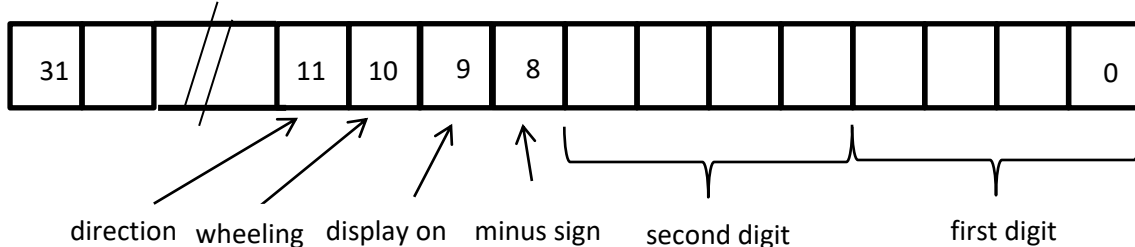
```
// SOUND
ULONG *sound_baseaddr_so = (ULONG *)XPAR_SOUND1_0_S00_AXI_BASEADDR;
int s,I;
s=0x9;
*(sound_baseaddr_so+0)=s;
for (i=0; i<30000; i++); // delay loop
s=0x0;
*(sound_baseaddr_so+0)=s;
```

7.5 Seven_Segment (custom IP)

The ZedBoard carries 5 connectors for Peripheral Module 'Pmod' as defined by Digilent. Digilent and other companies offer a variety of components which can be connected to those connectors. The PmodSSD from Digilent offers a single two-digit seven segment display device. The seven segment IP supports this device. Each of the segments is driven directly through an IO pin while the 8th pin selects which digit is illuminated. The IP switches so frequently between both digits that both seem to be always active.



As the IP drives directly into the segments, its input comes from an AXI light interface to the processor. The IP has 4 32 bit registers of which currently only Reg 0 is used. The register has the following layout. There is a hex to seven segment conversion performed



If bit 8 is set to '1' then a minus sign is displayed on the second digit and not the value set in bits 4 to 7.

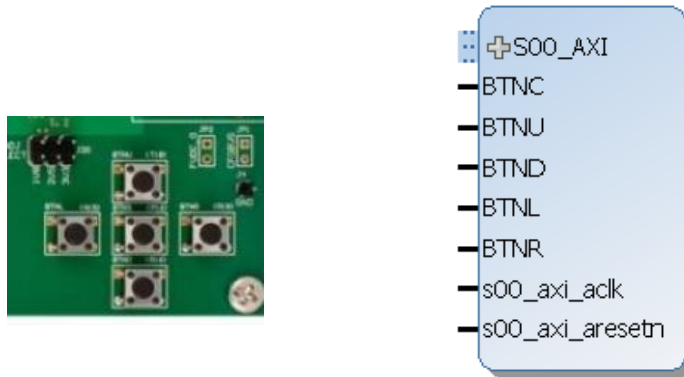
If bit 9 is set to '0' nothing will be displayed. Only if this bit is set to '1' the other bits are examined and displayed.

Sample invocation: display 'E5' on Seven Segment:

```
// seven segment
ULONG *seven_baseaddr = (ULONG *)XPAR_SEVEN_SEGMENT_0_S00_AXI_BASEADDR;
*(seven_baseaddr + 0) = 0x0200+0xE5 //0x02E5;
```

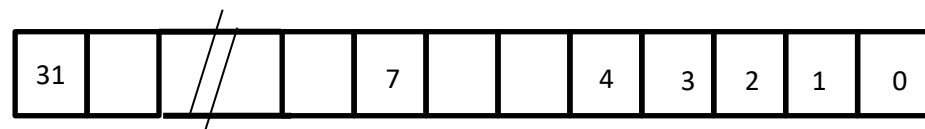
7.6 Buttons (custom IP)

The ZedBoard carries a set of 5 push buttons called: up, down, left, right and center. This IP provide access to those buttons via the AIX-bus.



Deglitching and edge detection circuitry within the IP will detect when a button is pressed. The IP has 4 32 bit registers of which currently only Reg 0 is used. Pressing a button will result in asserting a bit in this register. The bit will stay set until it is read by the program running on the Arm-processor. Reading the register will reset the bit automatically.

The register has the following layout.



Bit 0 : left button
Bit 1 : right button
Bit 2 : upper button
Bit 3 : lower button
Bit 4 : center button

Bits 31 .. 5 are always set to '0'.

Sample invocation: print button value:

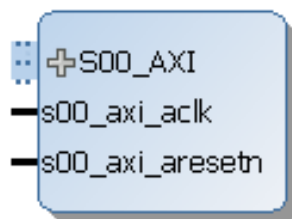
```
// btns
ULONG* btns_baseaddr_p = (ULONG*) XPAR_BTNS_0_S00_AXI_BASEADDR;

int btn = *(btns_baseaddr_p+0);

if ((btn > 0) )
{
    xil_printf("btns %x\n\r", btn);
}
```


7.7 Time_Count (custom IP)

The time count IP should enable a local time of day facility. The IP is connected via the AXI interface to the processing element. It is written to and read by the processor. The IP contains a number of counters. The first counter is driven by the processor clock and counts as many cycles as needed to produce a single cycle pulse every millisecond. The second counter is a 64 bit wide counter which increments by one whenever the millisecond pulse occurs. This counter will be set from the processor with a value representing the current time of day in milliseconds counted from a given start date.



The IP has 4 32 bit registers accessible through AXI light. Only Reg 0 and 1 is used.

Reg 0 contains the upper bits of the 8 byte current time value, while Reg 1 contains the lower bits of the 8 byte current time value. Only when written to Reg 1 the values from reg 0 and reg 1 will be loaded into the counter. Therefore the sequence of writing to these registers is important.

Also for reading the counter values requires the correct sequence. Reading reg 0 will return the upper 32 bits of the current time value. At the same time the lower bits are loaded into an auxiliary latch. The value of the aux latch will be returned (read) when performing a read from reg 1. There is no possibility to read the lower bits directly.

Sample invocation: time_sec contains the time_value in seconds:

```
u32 time_1 ;
u32 time_2 ;
long long unsigned int time_value;
long long unsigned int time_sec;
ULONG *time_count_baseaddr = (ULONG *)XPAR_TIME_COUNT_0_S00_AXI_BASEADDR;
    time_2 = *(time_count_baseaddr + 0);
    time_1 = *(time_count_baseaddr + 1);
    time_value = time_2;
    time_value <= 32;
    time_value = time_value + time_1;
    time_sec = time_value/1000;
//Print the counter value
    printf("The counter value in sec. is %llu:\n", time_sec);
```

7.8 Time of Day concept

Coordinated Universal Time (UTC) is the basis for civil time today. This 24-hour time standard is kept using highly precise atomic clocks combined with the Earth's rotation.

<http://www.timeanddate.com/time/aboututc.html>.

Based on this standard the world is divided into 24 time zones. Normally a computer displays the time and date in accordance with the time-zone it is operating in. When exchanging timestamped message between each other, however, not the time at the message origin is transferred but rather a universal value representing the number of milliseconds passed since a given data (here 1/1/1970) in the GMT zone.

There exist functions in the operating system and also in programming languages like C by which this number can be converted to the data and time at the message origin and or the message destination.

Computer hardware makes use of this in that way that they implement a 64 bit counter which is incremented every millisecond. In most of the systems this counter is also powered via a backup battery in order not to lose time when the system is switched of. Loaded once with the actual time of day count value, the current counter value will represent the current time.

```
// Recieve Timestamp and write it to the counters
else if ((received_data[4] == 7)) {

    unsigned long long int timestamp = 0;
    unsigned long int timeh = 0;
    unsigned long int timel = 0;
    long unsigned int time_value;
    int i, temp;

    // receive 8 byte time of day value.
    unsigned char* my_data = (unsigned char*)malloc(8);
    memcpy(my_data, received_data+5, 8);

    for(i=7;i>=4;i--)
    {
        timeh<<=8;
        temp = my_data[i];
        timeh = timeh+temp;
    }
    for(i=3;i>=0;i--)
    {
        timel<<=8;
        temp = my_data[i];
        timel = timel+temp;
    }
    ULONG *time_count_baseaddr = (ULONG *)XPAR_TIME_COUNT_0_S00_AXI_BASEADDR;

    *(time_count_baseaddr+0) = timeh;
    *(time_count_baseaddr+1) = timel;

    printf("timeh =%lu:\n timel = %lu:\n", timeh,timel);
    free(my_data);}
```

```

// Read from counters and convert to string representing Local Time
// in Www Mmm dd hh:mm:ss yyyy format where Www is the weekday, Mmm the
//month in letters, dd the day of the month, hh:mm:ss the time, and yyyy
//the year.

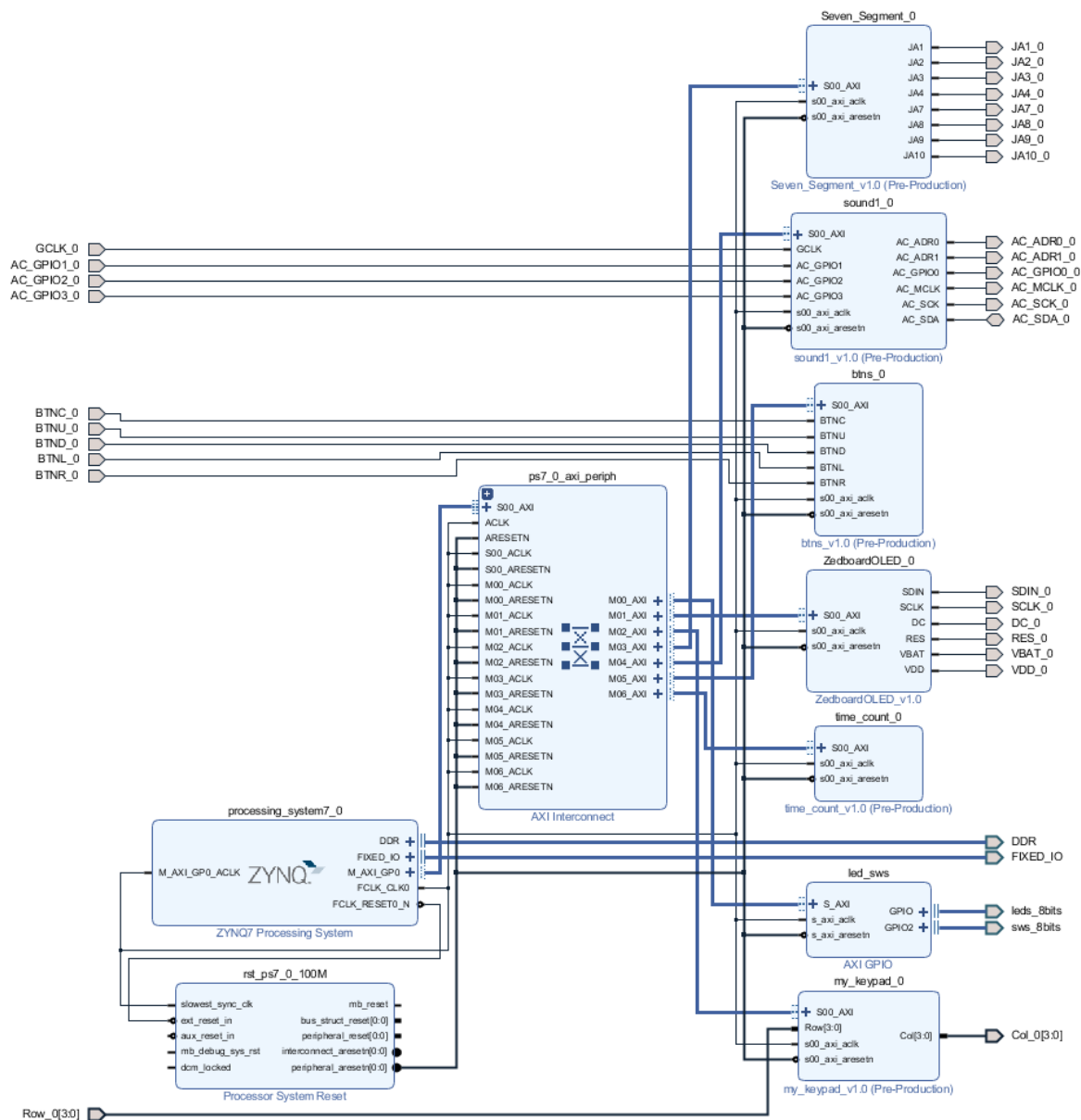
u32 time_1 ;
u32 time_2 ;
long long unsigned int time_value;
long long unsigned int time_sec;
ULONG *time_count_baseaddr = (ULONG *)XPAR_TIME_COUNT_0_S00_AXI_BASEADDR;
    time_2 = *(time_count_baseaddr + 0);
    time_1 = *(time_count_baseaddr + 1);
    time_value = time_2;
    time_value <= 32;
    time_value = time_value + time_1;
    time_value = time_value/1000;
    time_value = time_value + 7200; // for Israel time GMT+2 ;
    xil_printf("The current time is  = %s  \n\r" ,ctime(&time_value) );

```

8 Part B – Preparation report

דו"ת מכין חלק ב'

נתון קוד בשפת C (קובץ part2_template.c) המתאים למערכת חומרה שבנינו בסוף חלק א' של הניסוי:



בלוק ה-Processing System תומך בתקשורת מסוג Ethernet.

הקוד מבוסס על **IwIP Echo Server template** ומכיל את הפונקציות הנדרשות לתמיכה ב- TCP/IP stack כמו פונקציות המטפלות בהקמת הקשר ובשליחת וקבלת הודעות.

נשים לב כי הקוד משתמש ב-callback, קטעי קוד אשר מועברים כארגומנטים עבור פונקציות אחרות ומשפיעות על התנהגות הפונקציה. תוך כדי המעבר על הקוד המצורף נסה למצוא

את כל הקשרים והתלויות בין הפונקציות השונות בקוד, מי קורה למי ומה התוצאה של הקשרים הנ"ל.

קוד הדוגמא מממש את הפונקציות 0,2,3,5 ו-7 (ראה סעיף 7.3.1).

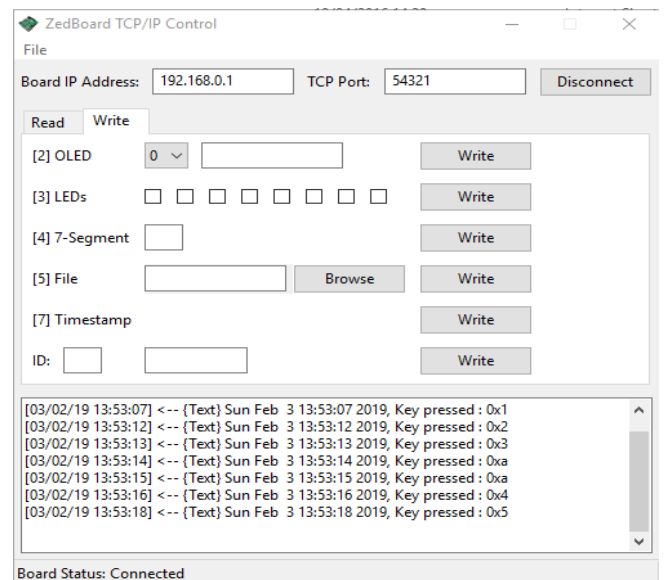
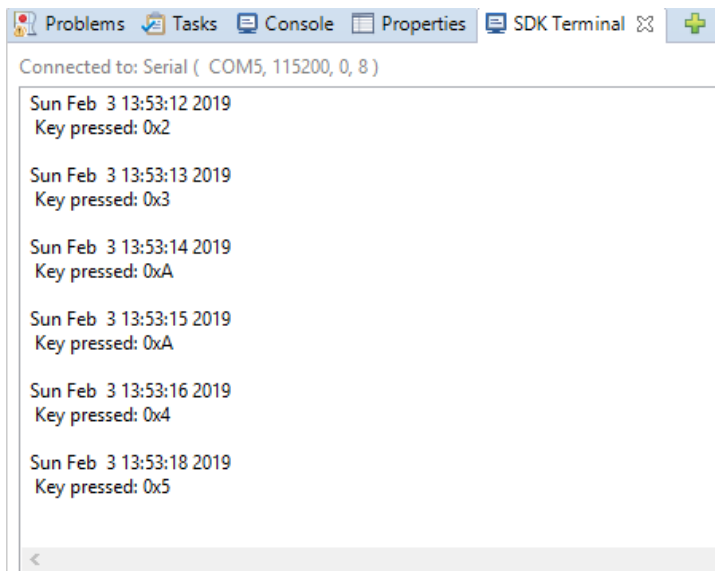
בנוסף הקוד מדגים את הפעולות הבאות:

1. שליחת נתונים למחשב ה-host והדפסתם על מסך ה-GUI עם הקשת מקש ב-keypad.

עבור כל הקשה נשלחת הודעה עם ציון התאריך, הזמן והמקש שהוקשו.

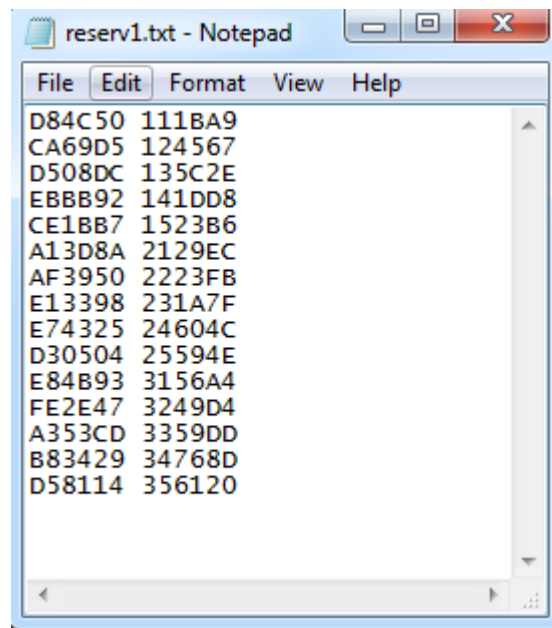
כמו כן מתבצעת הדפסת הנתונים על מסך ה-terminal.

להלן צילומי מסך של הדפסות המתקבלות על מסכי ה-GUI וה-terminal:

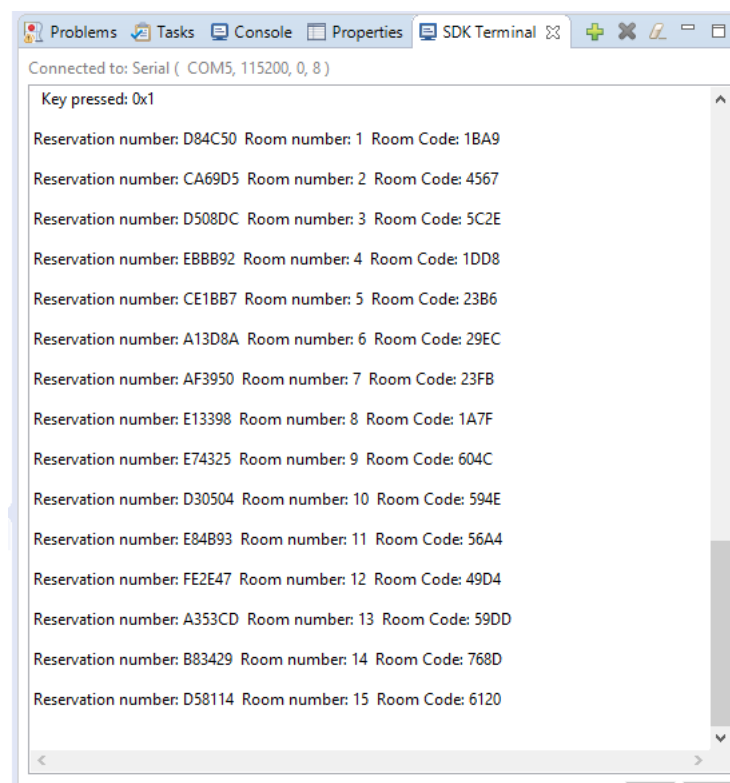


2. קבלת קובץ ממחשב ה-host ושמירת תכולתו במערך.

דוגמא לקובץ שנשלח:



3. עם הקשת המקש '1' ב- keypad מתבצעת הדפסת הקובץ שנשלח על מסך ה-terminal .
להלן צילום מסך של ה-terminal עם הקובץ המודפס:



ראה למטה את קוד הדוגמא ה-C עם הסברים בגוף הקוד.

Part B Template

```
#include <stdio.h>
#include <string.h>
#include "xparameters.h"
#include "netif/xadapter.h"
#include "xil_types.h"
#include "xil_assert.h"
#include "platform.h"
#include "platform_config.h"
#include "xgpio.h"
#include "xgpiops.h"
#ifdef __arm__
#include "xil_printf.h"
#endif
#include "lwip/err.h"
#include "lwip/tcp.h"
struct tcp_pcb* active_client_pcb = NULL;

/*****DEFINE*****/
//Define IP packet Message ID:

#define SWITCH_A 0
#define WR_OLED 2
#define WR_LED 3
#define WR_7SEG 4
#define RD_FILE 5
#define WR_TIMESTAMP 7
#define SENT_TO_HOST 8

//Read file:
#define FILE_LINE_NUM 16 //determine the number of line to
be read from the received file
#define LINE_CHAR_NUM 15 //determine the number of chars at
each file line
#define CHAR_NUM_COL1 6 //determine the number of char at
first line word
#define NUM_OF_SPACE 1 //determine the number of space
between the line words
#define END_LINE_CHARS 2 //determine the number of end line
chars, \n\r
#define DATA_OFFSET 5 //determine the offset of the data
from the packet head
#define ASCII_BIGLETTER_VAL 64 //Big letters offset in ascii table
#define ZERO_ASCII_VAL 48 //
#define A_ASCII_TO_10 55 //The "distance" between A to 10 in
ascii table
/*****END-DEFINE*****/

/* GPIO Globals */
/*Global variables declaration*/

XGpio led_sws_gpio_inst;
```

We use the "lwIP Echo Server" template to create all the needed C files to open and maintain a TCP-IP connection.

Include files are from "lwIP Echo Server" template.

Define constants for GUI ⇔ Zedboard Communication

```

XGpioPs psGpioInstancePtr;

XGpioPs_Config* GpioConfigPtr;
int xStatus;
int iPinNumberEMIO = 54;
u32 uPinDirectionEMIO = 0x0;
u32 uPinDirection = 0x1;
unsigned int code_entered;

/*Variables declarations for time counters and file reading*/
long long unsigned int time_value,time_sec, time_sec2;
u32 time_1 ;
u32 time_2 ;
int reservation[16];
int assign_code[16];
char key_buffer[64];

/* Add Your Global variables */

/*Transfare_data- this function allocate the data structure
and initiate the data transfer to the Zedboard */

int transfer_data(void* data, unsigned int len)
{
    err_t err;
    struct pbuf* transfer_pbuf = pbuf_alloc(PBUF_TRANSPORT, len,
    PBUF_RAM);
    pbuf_take(transfer_pbuf, data, len);
    err = tcp_write(active_client_pcb, transfer_pbuf->payload,
    transfer_pbuf->len, 1);
    tcp_output(active_client_pcb);
    pbuf_free(transfer_pbuf);
    return err;
}

void print_app_header()
{
    xil_printf("\n\r\n\r\n\r-----lwIP TCP echo server ----- \n\r");
    xil_printf("TCP packets sent to port 6001 will be echoed
back\n\r");
}

/*recv_callback- handle the receiving packets, getting the pcb
structure
that contain the ID of the event (the type of the data), packet
length and data
Predefined ID numbers (data[4]):
0. Initiate read from switches A
1. Initiate read from switches B
2. Write to a line on OLED
3. Write to the 8 LEDs
4. Write 2 digits to a seven segment display
5. Write the content of a file to the board
6. When receiving a message, display key-pad pressed and single
digit hex value.
7. Write the timestamp as 64 bit value representing the number of
milliseconds past since 1/1/1970.
8. When receiving a message, display the bytes as ascii text in the
message field.

```

Global variables declaration

Variables declarations for time counters and file reading

Transfer data function that builds and sends the packed in a format that the GUI can understand.


```

    */
err_t recv_callback(void *arg, struct tcp_pcb *tpcb,
                    struct pbuf *p, err_t err)
{
    unsigned char* received_data;
    u16_t received_data_len;

    /* do not read the packet if we are not in ESTABLISHED state */
    if (!p) {
        tcp_close(tpcb);
        tcp_recv(tpcb, NULL);
        return ERR_OK;
    }

    /* indicate that the packet has been received */
    tcp_recved(tpcb, p->len);

    /* Allocate buffer for received data and copy it. */
    received_data = (unsigned char*)malloc(p->len);
    memcpy(received_data, p->payload, p->len);
    received_data_len = p->len;

```

This function is called upon receiving a new packet. First it gets from the packet the eventid and the length of the packet

The predefined ID numbers (data [4]) are:

- | | |
|--|-----------------|
| 0. Initiate read from switches - | implemented |
| 1. | not implemented |
| 2. Write to a line on OLED - | implemented |
| 3. Write to the 8 LEDs - | implemented |
| 4. Write 2 digits to a seven segment display - | not implemented |
| 5. Write the content of a file to the board - | implemented |
| 6. | not implemented |
| 7. Write the timestamp as 64 bit value representing the number of milliseconds past since 1/1/1970 - | implemented |
| 8. When receiving a message, display the bytes as ascii text in the message field - | implemented |

If you want to add a response to received data, you should add it to this function, after the "User Logic start" remark.

received_data[0..3] – packet length

received_data[4] - ID number

received_data[5.. (length+3)] – data (length bits are not included in total packet lengths)

```

/* *****
 * User Logic start
 * Handle received_data here.
 * Example logic: echo server - send all received packets back:
 * transfer_data(received_data, received_data_len);
 * *****
 */

ULONG message_len = *((ULONG*)received_data
if ((message_len == 1) && received_data[4] == SWITCH_A)
{
    // Switches A read request.
    // Read switches.
    unsigned char switches_a = (unsigned char)
    XGpio_DiscreteRead(&led_sws_gpio_inst, 2);

    // Build response message and send it.
    u16_t response_message_len = 4 + 1 + 1;
    unsigned char* response_message = (unsigned
    char*)malloc(response_message_len);
    *((ULONG*)response_message) = 2;
    response_message[4] = 0;
    response_message[5] = switches_a;
    transfer_data(response_message, response_message_len);

    free(response_message);
}
else if (received_data[4] == WR_OLED) {
    // OLED write request.
    // Build the buffer to write.
    unsigned int line_number = received_data[5];

    u16_t string_len = message_len - 2; // Subtracting size
    of messageID and line number.
    if (string_len <= 16) {
        unsigned char* string_buffer = (unsigned
        char*)malloc(string_len + 1); // With a place for
        '\0'.
        memcpy(string_buffer, received_data+6, string_len);
        // 6-> 4 byte + type + line
        string_buffer[string_len] = '\0';

        // Write to OLED.
        clear();
        print_message(string_buffer, line_number);

        free(string_buffer);
    }
}
else if ((message_len == 2) && received_data[4] == WR_LED) {
    // LEDs write request.
    // Write to LEDs.
    XGpio_DiscreteWrite(&led_sws_gpio_inst, 1,
    received_data[5]);
}

```

Example of handling received data.

Example of sending a simple "write data" request

Example of reading a file.

The example is based on the following file format:

"Reservation number – 6 Hex numbers" "Room & Code 6 Hex numbers"

Total of 15 entries in a file.

It reads the file into 2 arrays.

For different data receiving format this part should be changed.

```
else if (received_data[4] == RD_FILE) {
    //Save file to memory array.
    int temp,i,k;
    unsigned int data=0;
    // receive file with reservation data.
    // Build the buffer to write.
    unsigned char* my_data = (unsigned char*)malloc(LINE_CHAR_NUM);
    // 2* 6 char + 1 space + carriage-return + line-feed = 15
    byte per record
    for(k = 1; k < FILE_LINE_NUM ; k++ )
    {
        memcpy(my_data,
            received_data+DATA_OFFSET+((k-
            1)*LINE_CHAR_NUM), LINE_CHAR_NUM);
        for(i=0;i<CHAR_NUM_COL1;i++)
        {
            data<<=4;
            if (my_data[i] < ASCII_BIGLETTER_VAL)
                // ascii numbers are between 48 and 57 decimal value
                temp = my_data[i] - ZERO_ASCII_VAL;

            else
                // ascii big letters are above 64 decimal value
                temp = my_data[i] - A_ASCII_TO_10;

            data = data+temp;
        }
        reservation[k] = data;
        data = 0;
        for(i = CHAR_NUM_COL1 + NUM_OF_SPACE ;
            i < LINE_CHAR_NUM - END_LINE_CHARS ;
            i++)
        {
            data<<=4;
            if (my_data[i] < ASCII_BIGLETTER_VAL)
                temp = my_data[i] - ZERO_ASCII_VAL;

            else
                temp = my_data[i] - A_ASCII_TO_10;

            data = data+temp;
        }
        assign_code[k] = data;
        data = 0;
    }
}
```

```

    }

    free(my_data);

}

else if ((received_data[4] == WR_TIMESTAMP))
{
    // Receive Timestamp and write it to the counters
    unsigned long long int timestamp = 0;
    unsigned long int timeh = 0;
    unsigned long int timel = 0;
    long unsigned int time_value;
    int i, temp;
    // receive 8 byte time of day value.
    unsigned char* my_data = (unsigned char*)malloc(8);
    memcpy(my_data, received_data+5, 8);

    for(i=7;i>=4;i--)
    {
        timeh<<=8;
        temp = my_data[i];
        timeh = timeh+temp;
    }
    for(i=3;i>=0;i--)
    {
        timel<<=8;
        temp = my_data[i];
        timel = timel+temp;
    }
    ULONG *time_count_baseaddr = (ULONG
    *)XPAR_TIME_COUNT_0_S00_AXI_BASEADDR;

    *(time_count_baseaddr+0) = timeh;
    *(time_count_baseaddr+1) = timel;

    free(my_data);

}

/* *****
 * User Logic end
 * *****
 */

/* free the received pbuf */
pbuf_free(p);

/* Free the allocated buffer. */
free(received_data);

return ERR_OK;
}

// accept_callback- received new pcd struct and bind it to the
recv_callback
err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    static int connection = 1;

```

```

    /* set the receive callback for this connection */
    tcp_rcv(newpcb, rcv_callback);

    /* just use an integer number indicating the connection id as
the
    callback argument */
    tcp_arg(newpcb, (void*)connection);

    /* increment for subsequent accepted connections */
    connection++;

    /* Save the new client pcb. */
    active_client_pcb = newpcb;

    return ERR_OK;
}

/*start_application- */
int start_application()
{
    struct tcp_pcb *pcb;
    err_t err;
    unsigned port = 54321;

    /* create new TCP PCB structure */
    pcb = tcp_new();
    if (!pcb) {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return -1;
    }

    /* bind to specified @port */
    err = tcp_bind(pcb, IP_ADDR_ANY, port);
    if (err != ERR_OK) {
        xil_printf("Unable to bind to port %d: err = %d\n\r",
port, err);
        return -2;
    }

    /* we do not need any arguments to callback functions */
    tcp_arg(pcb, NULL);

    /* listen for connections */
    pcb = tcp_listen(pcb);
    if (!pcb) {
        xil_printf("Out of memory while tcp_listen\n\r");
        return -3;
    }

    /* specify callback to use for incoming connections */
    tcp_accept(pcb, accept_callback);

    xil_printf("TCP echo server started @ port %d\n\r", port);

    return 0;
}

/* missing declaration in lwIP */
void lwip_init();

```

This function opens a new connection on the port number 54321.

```

static struct netif server_netif;
struct netif *echo_netif;

void print_ip(char *msg, ip_addr_t *ip)
{
    print(msg);
    xil_printf("%d.%d.%d.%d\n\r", ip4_addr1(ip), ip4_addr2(ip),
                ip4_addr3(ip), ip4_addr4(ip));
}

void print_ip_settings(ip_addr_t *ip, ip_addr_t *mask, ip_addr_t
*gw)
{
    print_ip("Board IP: ", ip);
    print_ip("Netmask : ", mask);
    print_ip("Gateway : ", gw);
}

#ifdef __arm__
#if XPAR_GIGE_PCS_PMA_SGMII_CORE_PRESENT == 1 ||
XPAR_GIGE_PCS_PMA_1000BASEX_CORE_PRESENT == 1
int ProgramSi5324(void);
int ProgramSfpPhy(void);
#endif
#endif

#define KEYPAD_ALL_RELEASED 0x0000000F
ULONG released_state = KEYPAD_ALL_RELEASED;
ULONG released_state_prev =
KEYPAD_ALL_RELEASED;

```

*You need to adjust the
"KEYPAD_ALL_RELEASED" value according
to your keypad implementation.*

```
void handle_events() {
```

This function handles all periodic events in the system.

Put your code in this function according to the events in your system.

*To read from keypad and switches and to write to the OLED or the LEDs - you can use the
'C' code from the first part of the experiment*

*You can see an example that driven by pressing one of the keypad keys. It prints the
current time and the pressed key on the terminal and sends it to be printed on the host GUI
using Ethernet communication.*

*When key "1" is pressed it also writes on the terminal the received file in the following
format: reservation num. 6 char, room number 2 char, and room code 4 char.*

For example for row D84C50 111BA9 will be printed:

Reservation number: D84C50 Room number: 11 Room Code: 1BA9

```

/* *****
 * User Logic start
 * Handle periodic events here.
 * *****
 */

// KEYPAD
ULONG* keypad_baseaddr_p =
(ULONG*)XPAR_MY_KEYPAD_0_S00_AXI_BASEADDR;
ULONG *sound_baseaddr_so = (ULONG
*)XPAR_SOUND1_0_S00_AXI_BASEADDR;

// Update keys and released_states.
released_state_prev = released_state;
released_state = *(keypad_baseaddr_p+1);
int key = *(keypad_baseaddr_p+0);

// Check if keypad has just been released.
if ((released_state == KEYPAD_ALL_RELEASED) &&
(released_state_prev != KEYPAD_ALL_RELEASED))
{

```

```

    int s,i,k;
    s=0x9;
    *(sound_baseaddr_so+0)=s;
    for (i=0; i<30000; i++); // delay loop
    s=0x0;
    *(sound_baseaddr_so+0)=s;

```

**Play a short
beep**

```

    u32 time_1 ;
    u32 time_2 ;
    long long unsigned int time_value;
    long long unsigned int time_sec,new_time;
    ULONG *time_count_baseaddr = (ULONG
*)XPAR_TIME_COUNT_0_S00_AXI_BASEADDR;
    time_2 = *(time_count_baseaddr + 0);
    time_1 = *(time_count_baseaddr + 1);
    time_value = time_2;
    time_value <=& 32;
    time_value = time_value + time_1;
    time_value = time_value/1000;
    time_value = time_value + 7200;
    // for summertime 10800;

```

**Read the
current
time**

```

xil_printf(" %s Key pressed: 0x%01x \n\r"
,ctime(&time_value),key );

```

```

sprintf(key_buffer, "%s, Key pressed : 0x%01x
\n",ctime(&time_value),key);
ul6_t response_message_len = 4 + 1 + 64;
unsigned char* response_message = (unsigned
char*)malloc(response_message_len);
*((ULONG*)response_message) = 65;
response_message[4] = SENT_TO_HOST;
memcpy(response_message+5, key_buffer, 64);
transfer_data(response_message,
response_message_len);
free(response_message);

```

**Build the packet
that contains the
current time and
the pressed key
and call
transfer_data()
function**

```

        if (key == 0x1)
        {
            for (k=1; k<16; k++)
            {
                xil_printf("Reservation number: %x\n\r", reservation[k], assign_code[k]>>16, assign_code[k]-((assign_code[k]>>16)<<16));
            }
        }
    }
}

/* *****
 * User Logic end
 * *****
 */

int main()
{
    ip_addr_t  ipaddr, netmask, gw;

    // AXI GPIO leds & switches Initialization
    XGpio_Initialize(&led_sws_gpio_inst, XPAR_LED_SWS_DEVICE_ID);

    // PS GPIO Initialization
    GpioConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
    if(GpioConfigPtr == NULL) {
        return XST_FAILURE;
    }
    xStatus = XGpioPs_CfgInitialize(&psGpioInstancePtr,
        GpioConfigPtr, GpioConfigPtr->BaseAddr);
    if(xStatus != XST_SUCCESS) {
        print("PS GPIO INIT FAILED\n\r");
    }

    // EMIO PIN Setting to Input port
    XGpioPs_SetDirectionPin(&psGpioInstancePtr, iPinNumberEMIO,
        uPinDirectionEMIO);
    XGpioPs_SetOutputEnablePin(&psGpioInstancePtr, iPinNumberEMIO, 0);

    /* the mac address of the board. this should be unique per board */
    unsigned char mac_ethernet_address[] =
        { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

    echo_netif = &server_netif;
#ifdef __arm__
    if XPAR_GIGE_PCS_PMA_SGMII_CORE_PRESENT == 1 ||
        XPAR_GIGE_PCS_PMA_1000BASEX_CORE_PRESENT == 1
        ProgramSi5324();
        ProgramSfpPhy();
#endif
#endif

    init_platform();
}

```

IP's utilization


```

/* initialize IP addresses to be used */
IP4_ADDR(&ipaddr, 192, 168, 0, 1);
IP4_ADDR(&netmask, 255, 255, 255, 0);
IP4_ADDR(&gw, 192, 168, 0, 1);

print_app_header();

lwip_init();

/* Add network interface to the netif_list, and set it as default */
if (!xmac_add(echo_netif, &ipaddr, &netmask,
              &gw, mac_ethernet_address,
              PLATFORM_ETHADDR)) {
    xil_printf("Error adding N/W interface\n\r");
    return -1;
}
netif_set_default(echo_netif);

/* now enable interrupts */
platform_enable_interrupts();

/* specify that the network if is up */
netif_set_up(echo_netif);

print_ip_settings(&ipaddr, &netmask, &gw);

/* start the application (web server, rxtest, txtest, etc..) */
start_application();

while (1) {
/* receive and process packets-
    The receive interrupt handlers move the packet data from
the MAC and store them in a queue.
    The xemacif_input function takes those received packets
from the queue, and passes them to lwIP
NOTICE- The program is notified of the received data through
callbacks.*/
    xemacif_input(echo_netif);

    /* Handle other events. */
    handle_events();
}

/* never reached */
cleanup_platform();

return 0;
}

/* *****
 * User Logic end
 * *****
 */

```

**Open a TCP connection using
the defined IP address,
NetMask and default Gateway**

**Constantly receive and process
arriving packages.**

**Constantly handle events in the
system**

דו"ח מכין חלק ב' - שאלות הכנה:

מנגנון התקשורת בין ה-Host לכרטיס מבוססת על תקשורת TCP/IP.
באמצעות תקשורת זו אנחנו מקבלים stream של מידע מן המחשב אל הכרטיס ולהפך.

הסבר לגבי פורמט ההודעות הנתון ניתן לראות בסעיף 7.3.1.
מבנה ההודעה שהוגדר מכיל מספר שדות וביניהם גם את סוג ההודעה שנשלחת.
רק חלק קטן מהסוגים מוגדרים מראש ב-GUI וכל השאר פנויים לשימוש חופשי.
קוד ה-C (Part B Template) מכיל 2 פונקציות בהן נתרכז:

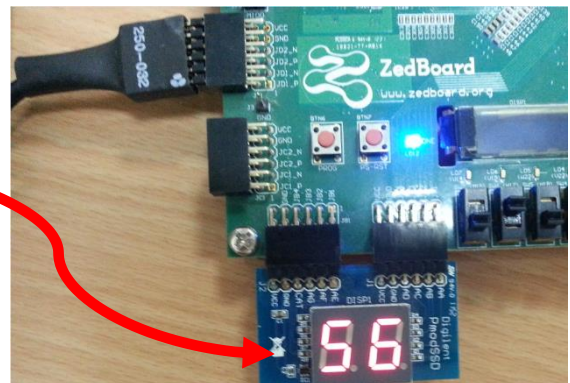
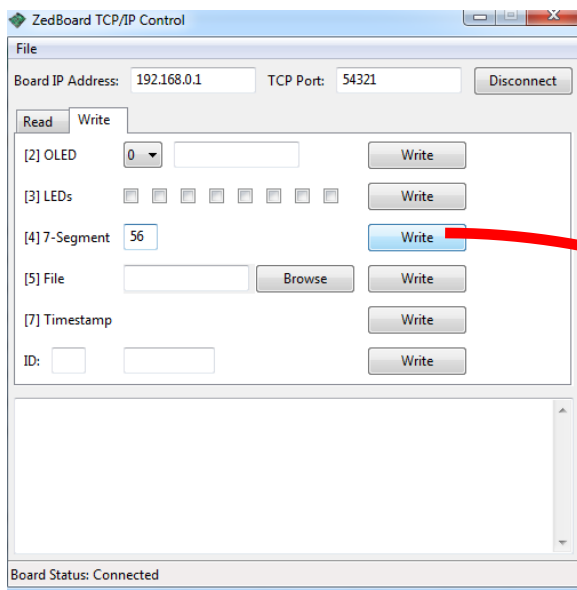
- a. `recv_callback` – פונקציה זו מטפלת בחבילות מידע (packets) המגיעות מהרשת (במקרה שלנו זה חבילות שה-GUI שולח לכרטיס ה-Zedboard דרך קו ה-Ethernet בפרוטוקול TCP/IP).
- b. `handle_events` – פונקציה זו מטפלת באירועים המגיעים מה-PL (FPGA) ל-PS (מעבד ה-ARM) דרך ה-AXI BUS.

כל אחת מהפונקציות הנ"ל יכולה לקרוא/לכתוב לפריפריות המחוברות ל-AXI BUS וכן ליזום שליחת חבילות דרך הרשת אל ה-GUI.

- 1 עבור חבילה (packet) השולחת הודעת כתיבה ל-seven segment:
 - 1.1 מה מספר הבתים הכולל של ההודעה?
 - 1.2 מה אורך ההודעה שיופיע ב-4 הבתים הראשונים?

עליך להוסיף את הפונקציונליות הבאה לקוד ה-C הנתון:

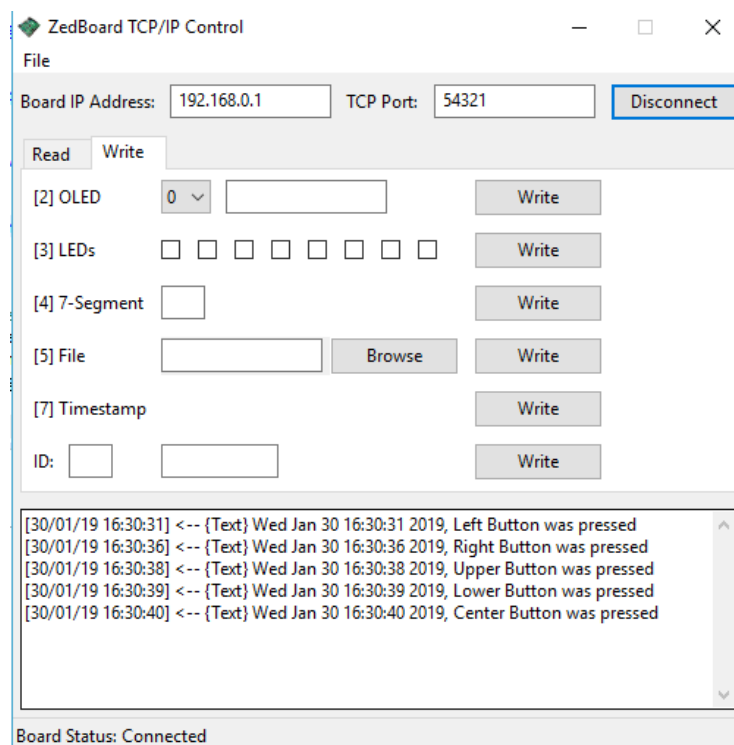
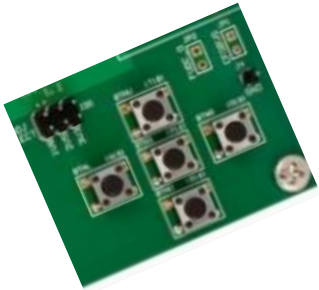
- 2 הוספו תמיכה בפונקציה מס' 4 מסעיף 7.3.1 (Write 2 digits to a Seven Segment Display).
נדרש להדפיס את המספר שנשלח מה-GUI על תצוגת ה-7 Segment שעל הכרטיס.



שימו לב שה-GUI שולח את מספר שיש להדפיס על מסך ה-7 Segment בבית אחד
בשדה [5].received_data.

3 הוסיפו קוד שעבור כל לחיצה על אחד ה-Buttons שולח הדפסה ל-GUI במחשב ה-host
עם מספר הלחצן שנלחץ והזמן בו נלחץ המקש.

להלן דוגמא למסך ה-GUI שאמור להתקבל:



4 - GUI מאפשר לשלוח מידע לכרטיס ה-FPGA עם סוג ההודעה שהשתמש בוחר, כמובן שאסור להשתמש בסוגי הודעה שמוגדרים מראש (ראה Figure 16 - predefined ID numbers).

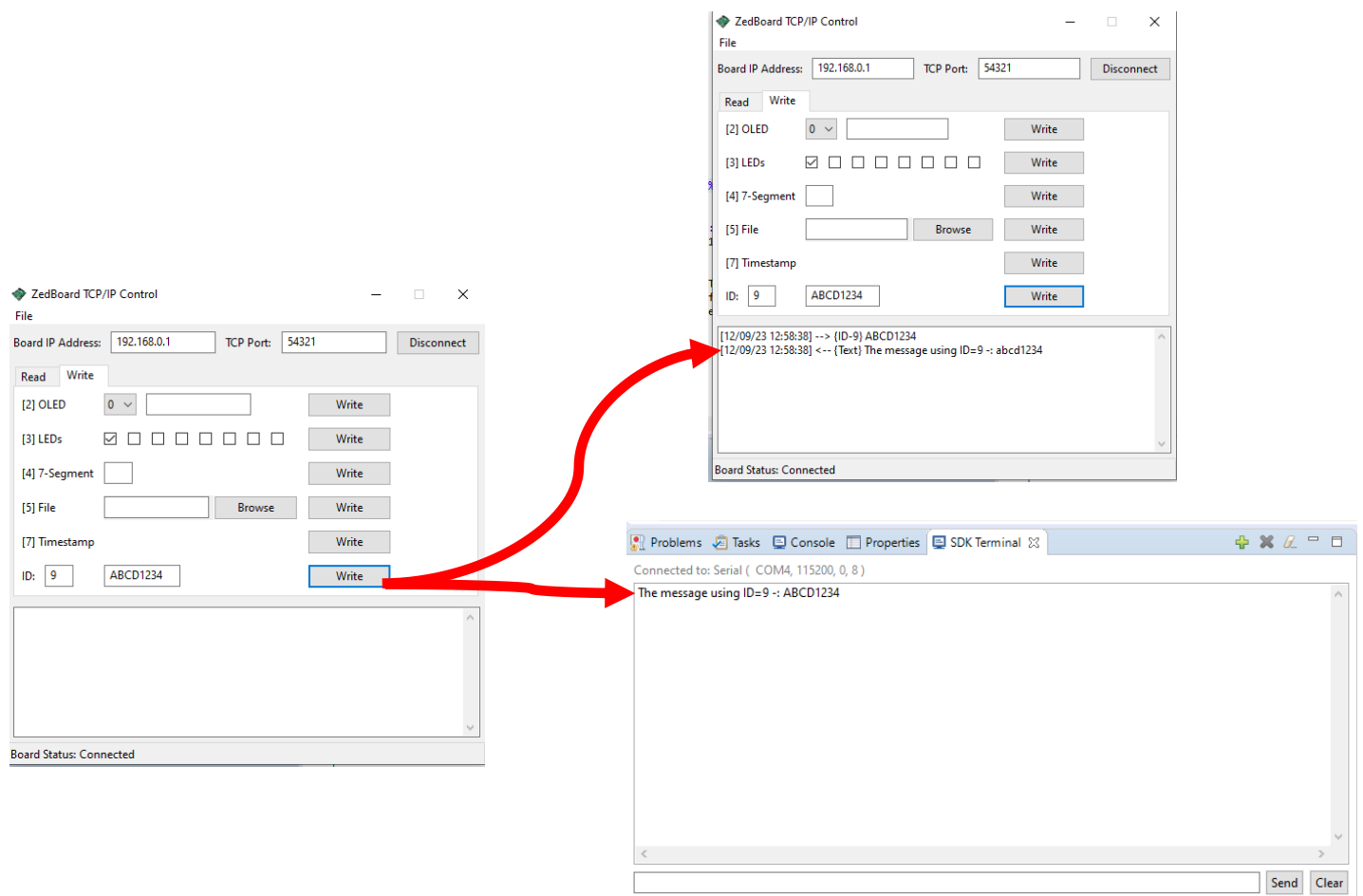
גודל המידע שנשלח הוא 32 ביטים – 8 ספרות הקסדצימליות.
עבור חבילה (packet) השולחת הודעה מסוג זה:

4.1 מה מספר הבתים הכולל של ההודעה?

4.2 מה אורך ההודעה שיופיע ב-4 הבתים הראשונים?

הוסיפו קוד שמדפיס גם למסך ה-GUI וגם למסך ה-terminal את ההודעה שנשלחת ע"י שימוש ב-ID=9.

להלן דוגמא למסך ה-GUI ומסך ה-terminal שאמורים להתקבל אחרי שליחת ההודעה:



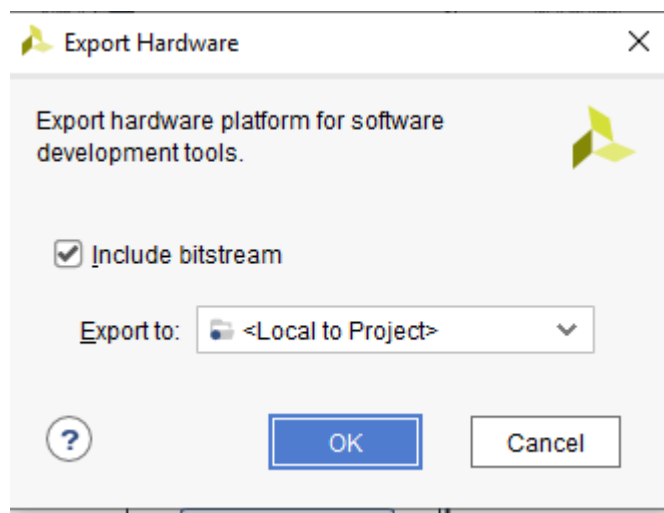
9 Part B – Execution in the Lab

מפגש שני – ביצוע הניסוי במעבדה

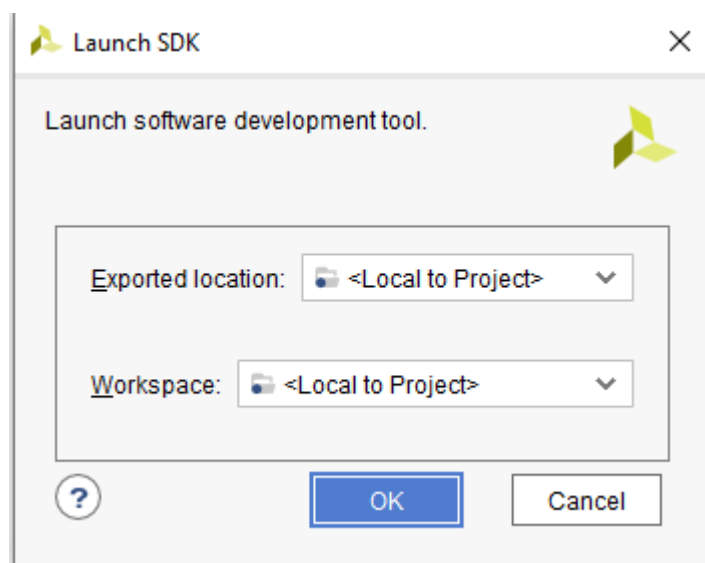
1. פתחו את הפרויקט שיצרתם בסוף חלק א' של הניסוי.

2. מחקו או שנו שם של תיקיית Z:\sopc\part1\part1.sdk

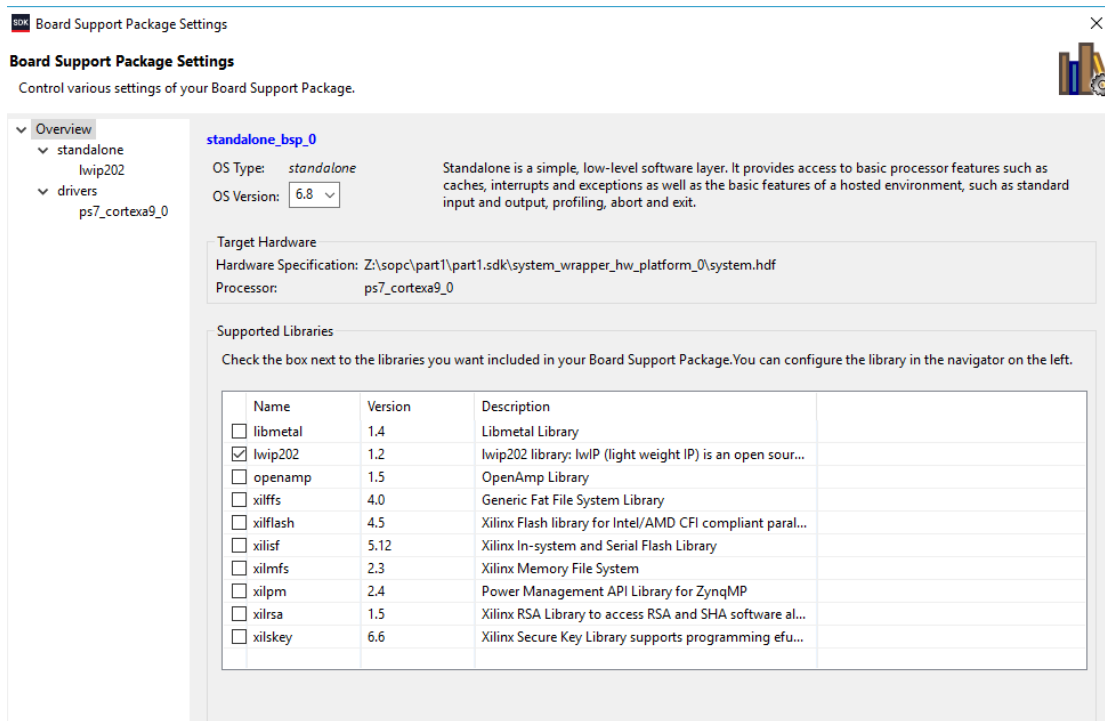
3. בצעו export hardware



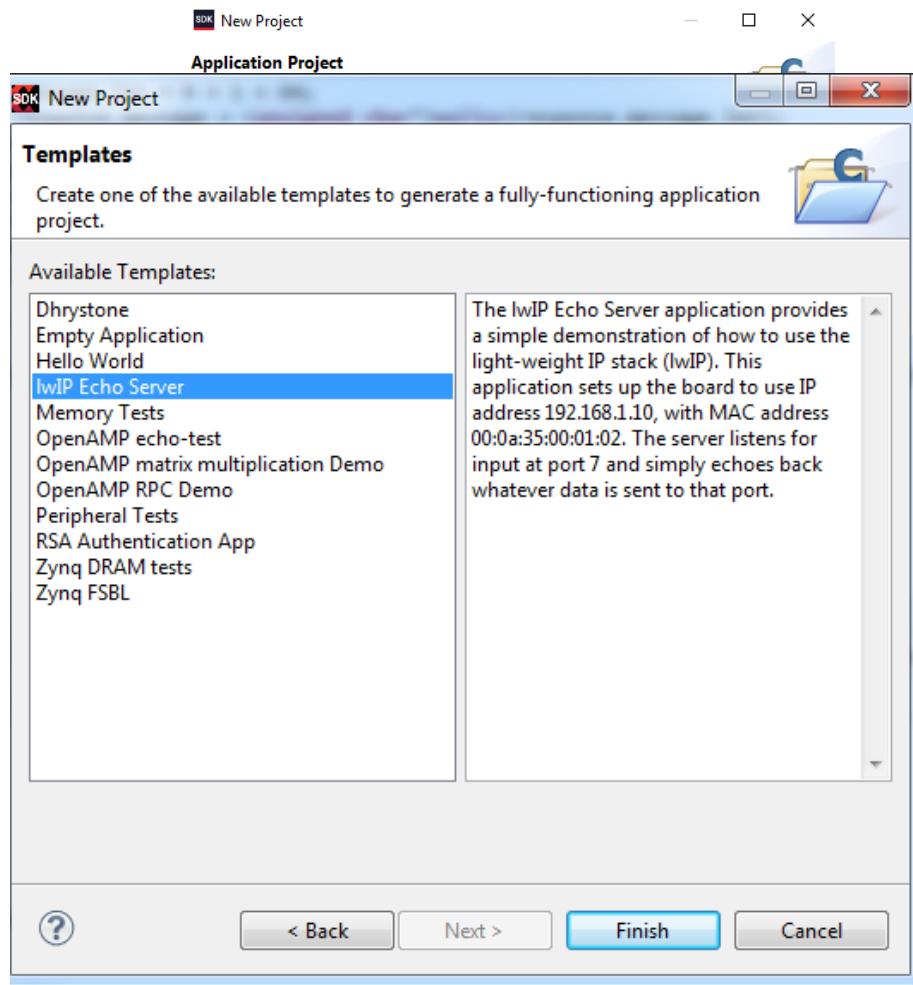
4. הפעילו את ה-SDK.



5. צרו board support package חדש עם תמיכה ב-lwIP TCP/IP stack library :



6. צרו אפליקציה חדשה המבוססת על lwIP Echo Server Template וזאת ע"מ שתכיל את כל קבצי ה-include הדרושים.



- 5 מחקו מה- project את הקובץ echo.c והחליפו את התוכן של הקובץ main.c בקוד שלכם מדו"ח מכין.
- 6 בדקו האם ערך המשתנה `KEYPAD_ALL_RELEASED` מתאים למימוש שלכם של רכיב ה-keypad.
- 7 קמפלו והריצו הכרטיס.
- 8 בדקו את תקינות התכן והראו למדריך.
- 9 בחלק ב' של הניסוי תקבלו משימה מהמדריך. המשימה תתבסס על החומרה שבניתם ועל קובץ C מדו"ח מכין.
- 10 בסיום המעבדה יש להראות למדריך את תוצאות ההרצה של המשימה על הכרטיס. יש להגיש למדריך דו"ח מסכם שיכיל את:
 1. שמות הסטודנטים ותיאור המשימה.
 2. הסבר על אופן המימוש.
 3. קובץ נפרד עם קוד ה-C שנכתב עם הדגשת הקוד שהוסף והסברים בגוף הקוד.