**In the name of GOD**

# EE25266 – ASIC/FPGA Chip Design

Dr. Mahdi Shabany

Electrical Engineering Department

Sharif University of Technology

## Lab#5 (1st lab on Zynq) – Spring 2024

## Introduction to Zynq

## Communication between PS and PL through AXI GPIO and BRAM

Prepared by Hossein Moghim & Mojtaba Pour Ali Mohammadi

## Objective:

In this lab, you become familiar with the Zynq SoC and some minimal capabilities and interfaces of the versatile chip. These include LEDs, keys, PS/PL connections, GP port and AXI GPIO, PS and PL data interaction through BRAM, (System) ILA as a debugging tool and micro SD.

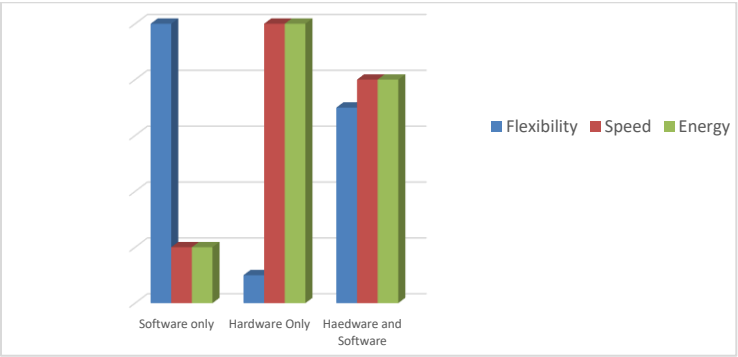
## Introduction:

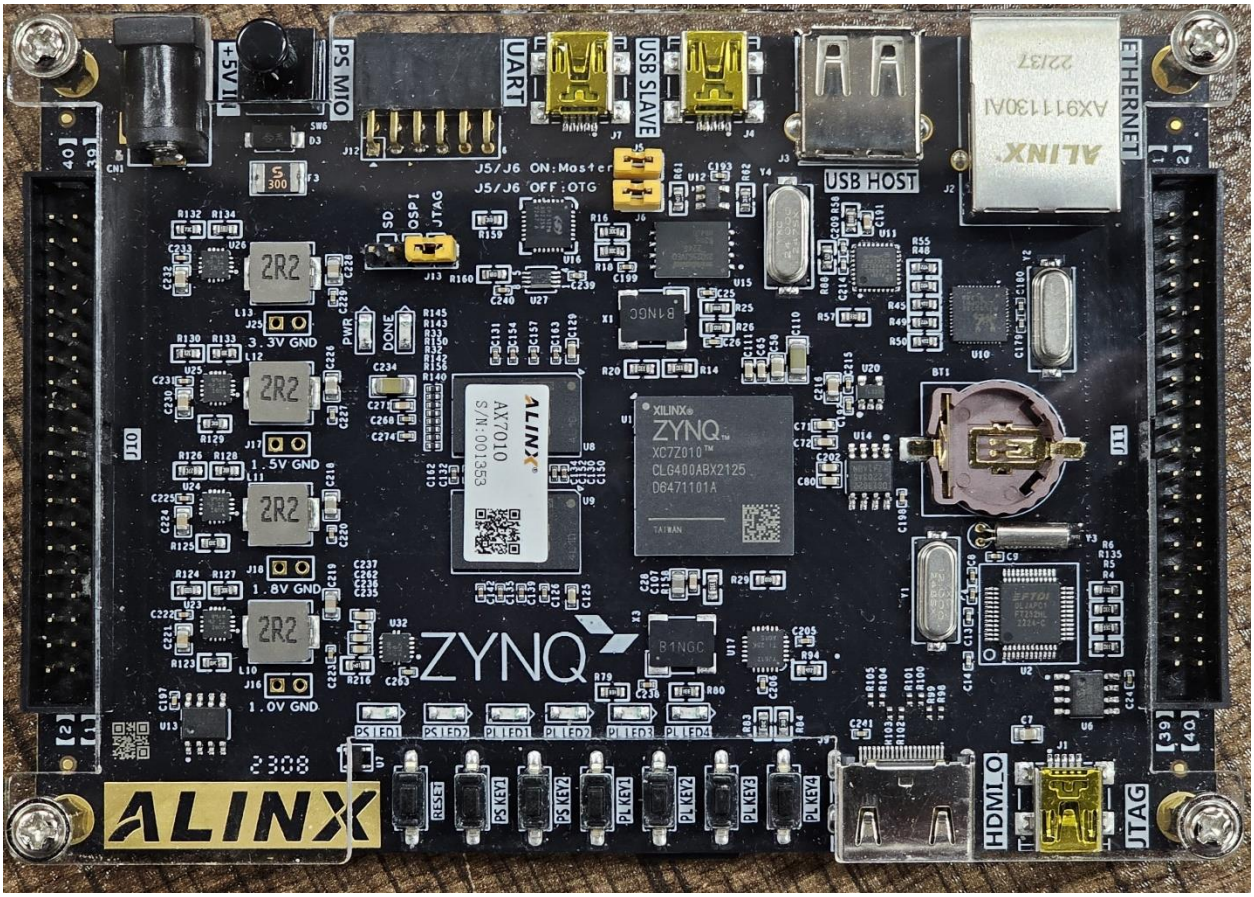
To realize your descripted digital circuit using Hardware Description Language (HDL), we have used FPGAs in the previous labs.

FPGAs are used to create any specific hardware you want, there is almost no built-in fixed hardware on the chip, you need to create the hardware dedicated to performing your particular application. Because the designed hardware is customized for the specific task, you can benefit from its high performance and parallelism. The drawback might be high power consumption.

Microcontrollers on the other hand, are a set of fixed circuits, each designed to be able to carry out some basic widely used tasks. By programming the microcontroller, you are actually configuring the general-purpose chip to execute your operations one by one.

Designers have long known that general-purpose processors are flexible, while single-purpose processors are fast. Most applications include both general-purpose needing tasks and single-purpose tasks. To address the demand of a chip able to execute both applications, in March 2011, Xilinx introduced the Zynq-7000 family, which integrates the software programmability of an ARM®-based processor (PS for Processing System) with the hardware programmability of an FPGA (PL for Programmable Logic).






In this lab, you will learn the PS/PL connections by verifying some modules. These verification methods will turn out to be useful in future. For realizing your described digital circuit on Zynq FPGAs you need to use Xilinx's own software, Vivado. You can watch videos 1 to 5 from the Zynq tutorial series, if you want to know more about Zynq, installing Vivado, getting acquainted with Vivado, simulation on it, synthesis, implementation and programming the Zynq chip using Vivado resting on your basic knowledge and preparedness. Before you begin the lab, please read the following document completely, this should answer many of your questions. If you have any other questions or problems - ask!

## Procedure:

# Part I:

*To begin with, watch video number 6 "Project AND" to get ready for the following lab.*

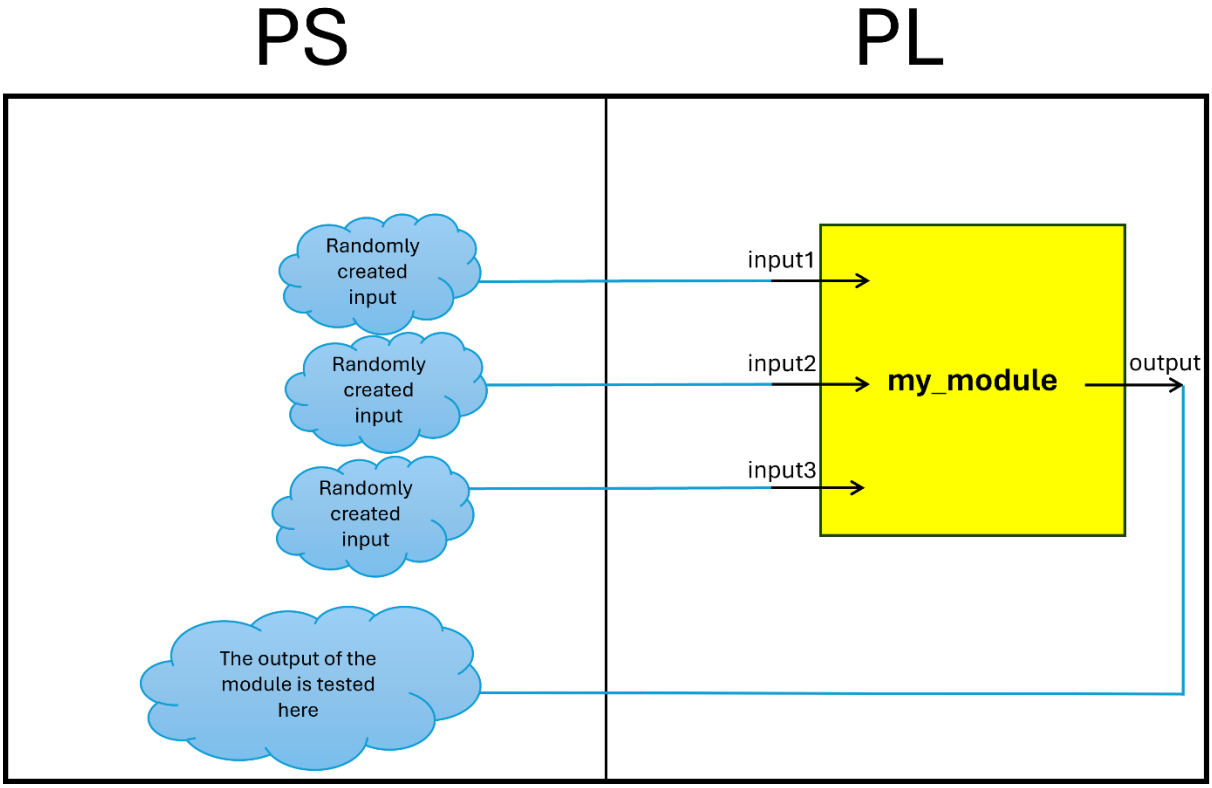
For this part, you need to create a 2-bit signed adder. The inputs would be the 4 PL keys, and the 3-bit output must be shown on 3 LEDs. You need to create the module, the constraint file and implement the circuit on the FPGA. You must use the board's user manual documentation to check which pin of the FPGA is connected to what peripheral of the board. Note that all the keys and LEDs are active-low.

## Part II:

*Please watch both parts of video number 10 "PS-PL interconnection" before continuing to the procedure. It is also optional to watch video number 8 "PS Intro GPIO" prior to them, if you want to know more about PS and SDK.*

Now you need to verify a simple 32-bit ALU using PS. You should first describe the ALU module to be implemented on the PL, then connect the PL to the ARM processor (PS), ultimately create a software program to test the module with 5,000,000 randomly created test vectors.

Whenever you create a module, you need to test it. First off, you simulate your module and test it via a testbench. The testbench as you know, creates some inputs, and feeds them into your module and gets the output of it to be checked. But when you implement it on a FPGA, you cannot use testbench as it is not synthesizable. Thanks to Zynq, you can test the module by the PS! Create your inputs on the microcontroller, pass them to your module on the PL side, and get the results from the module's output and verify it on the PS. Look at the diagram to understand the verification process better. Follow the steps bellow to easily perform the verification.

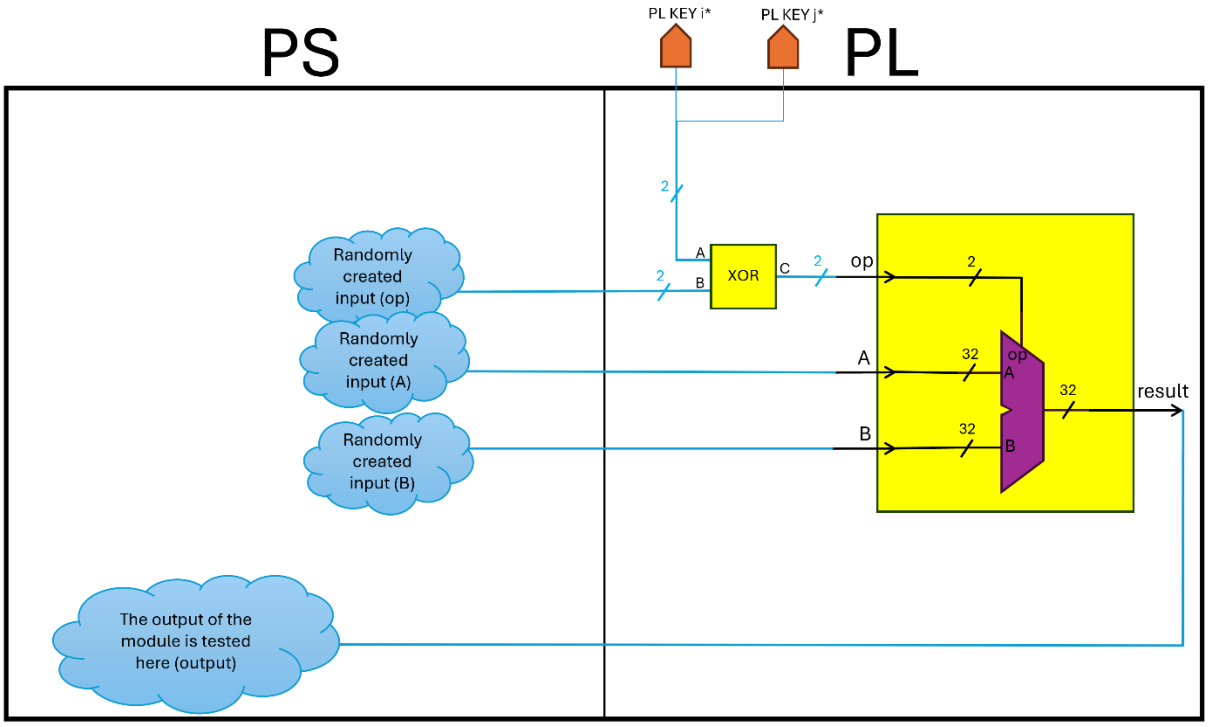


First, describe a 32-bit combinational ALU able to execute ADD, SUB, NAND, and NOR functions on the two 32-bit inputs using Verilog HDL. Any carry or overflow should be ignored. There must be one “op” input to customize the ALU, selecting which of the four operations is going to be performed.

Then, create another module that takes two 2-bit inputs A and B. This module should XOR A and B (bitwise) and have a 2-bit output, C.

After adding both of your modules to Vivado, create a block design to add the necessary IP Cores such as the PS and some AXI GPIOs.

Configure the IP Cores and attach them properly. A diagram for what is expected is shown in the following figure. One of the 2-bit inputs of the XOR module should be connected to 2 of the PLs’ keys, and the other input port of the module must be attached to the op output of the PS, which is the random op created by the processor. The output of the XOR module, inputs the op input port of the ALU. The XOR module is there to test your verification method. As long as the two PL key inputs of the modules are low and serve as zeros for the XOR, the PS evaluating the logic should not report any errors as the XOR doesn’t impact the op signal from the PS and passes them directly to the ALU, but when any of the keys go high, the PS must report an error. This is because the high key creates a 1 input for the XOR, and it negates the corresponding op bit from the PS, inputting a wrong bit to the ALU causing it to result an unwanted answer. You need to use both two channels of AXI GPIO IP cores to lessen the resources used as much as possible.




* Choose two keys randomly out of the four PL KEYs. You do not want everyone to use certain keys such as PL KEY 1 & PL KEY 2 wearing them out. Keep this in mind for future labs as well.

After exporting the implemented hardware, open the SDK and start writing a software program that creates random input vectors and their results to test the module. Note that you need to initialize each AXI GPIO regardless of the number of channels activated for each of them. Additionally, you need to set data direction, read and write for each of the channels independently. The second argument of these three functions is channel selection and can be set as 1 or 2 based on your usage.

After any error, the evaluation should stop for 2 seconds and the expected result versus the gotten result from our module must be shown via UART on your PC during the test, and then go on. At the end of evaluation, the number of tests and the number of errors are to be shown on your PC.

# Part III:

**In this section, you will become familiar with how communication is established between PS and PL through internal Block RAMs (BRAMs) of FPGA.**

*You can use the template projects of video 11 & video 13 for Parts III, IV and V.*

*Before proceeding with this section, it is necessary to watch video number 11 from the Zynq tutorial series for mastering "Bidirectional Communication between PS and FPGA BRAMs", "Types of Accessible Memories", "Debugging with ILA and System ILA" and "AXI Bus".*

*Watch video number 13 to realize "PS and PL data interaction through BRAM".*

*And watch video 18.a for understanding "How to read and write files on the micro SD of the board from the PS".*

Consider question 2 of your second homework. You can view the question in the image below for a reminder:

2- In this question, you will become familiar with the verification of implemented structures using Matlab. It is recommended to study fixed-point arithmetic.
a) Using Matlab software, consider a complete period of a sinusoidal wave with a length of 1024 and convert it to fixed-point format (1,16,14). Now, write a code that initializes two 16-bit arrays of depth 1024 with sinusoidal values generated in Matlab (sin and cosine). To do this, use the system task $readmemh or $readmemb in the initial block. The procedure of this task can be easily found through an internet search. Then, write a module that uses these two arrays and generates a sinusoidal output with frequencies of 1, 2, 4, 8, and so on, multiplied by the frequency of the generated sinusoidal wave. To specify the output frequency, consider a 2-bit input that can only take values of 1, 2, 3, and 4, which correspond to the initial frequency, twice the initial frequency, four times the initial frequency, and so on.
b) To test the module written in part a, write a testbench that writes the module's output to a .txt file based on two desired input states. In Matlab, represent each of the two states in the form of sin + i*cos and plot their spectra using the fft or pwelch command. Consider an arbitrary sampling frequency and validate the frequencies of the generated waves.

In this part, we intend to perform the same task as requested in that question, with the difference that it should be implemented on the Zynq chip, and we want to utilize all of its capabilities.

Therefore, we plan to store 1024 sin and cos data instead of placing them in a text file (.txt) or memory file (.mem) that are suitable for simulation, on the Block RAMs inside the FPGA chip so that they can be usable after hardware implementation.

Naturally, the initial sin(x) and cos(x) data are placed by the PS on the FPGA's BRAMs. You have two options for generating sine and cosine data:

1) You can perform this task in the PS (using the chip's processor by writing C code in SDK) with the help of the math.h library. Then, convert the data as requested in the question to fixed-point format right there and store them in the BRAMs in binary form, just as you learned in video 11.
2) Another method is to generate the data similar to HW2 in MATLAB and save them in the correct fixed-point and binary format in a file. Then, place this file on the micro SD of the board and use the commands learned in video 18.a to read the data from this file and place it onto the BRAMs by the PS.

After placing 1024 samples of sin(x) and cos(x) on the FPGA's BRAMs, these data are always accessible to your hardware module so that after implementation on the FPGA, it can use them to generate sinusoidal and cosinusoidal signals with higher frequencies.

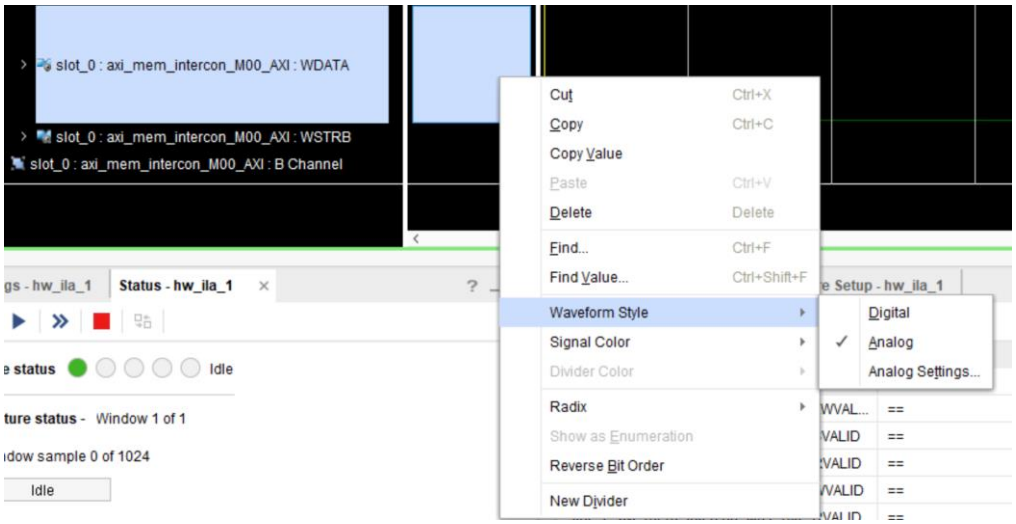
Therefore, the next step is to convert your module into a custom IP according to video 13, so that it can communicate effectively with the Block Memory Generator IP Core.

Note: The PS specifies that the sinusoidal signal should be generated with what frequency multiple of the initial frequency; place this two-bit command in the first location used by the BRAMs.

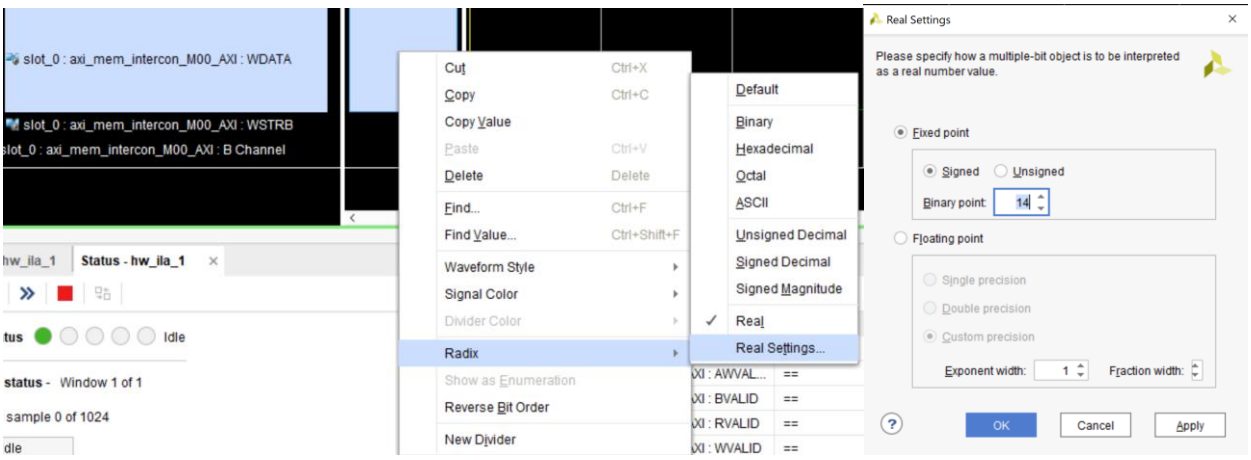
Finally, place the output of your module in certain locations of the BRAMs so that the PS can access them. Retrieve these output data from the BRAMs in the PS and save them as a text file on the micro SD of the board. Verify the correctness of your output, as mentioned in the question of HW2.

As you learned in video 11, one of the ways to debug your Verilog code and circuit is to use ILA (Integrated Logic Analyzer). For this purpose, use ILA in the bidirectional path from the PS to BRAM (between AXI Interconnect and AXI BRAM Controller) to ensure the correctness of your code by observing the initial signals written to BRAM and the final signals read from it by the PS. (Similar to video 11, assign the ILA trigger to the Valid signal of the AXI bus so that no data is missed; you can also adjust the ILA memory size and trigger position to display all data and prevent them from exceeding the limited ILA memory).

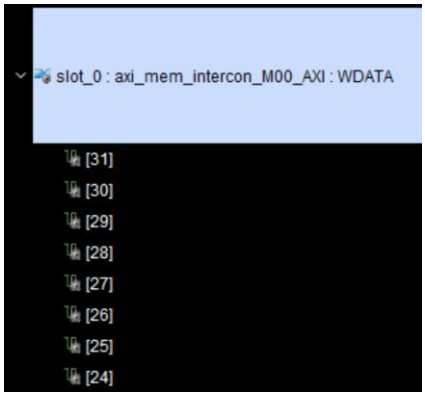
Note: As evident, the signals you intend to observe on the Waveform window of ILA are analog. Therefore, by right-clicking on the signal containing sinusoidal and cosinusoidal waves in the ILA window (WDATA and RDATA), similar to the image below, analogize its style and modify its settings in the Analog Setting section to ensure that the analog sinusoidal and cosinusoidal signals are displayed in the best possible way.



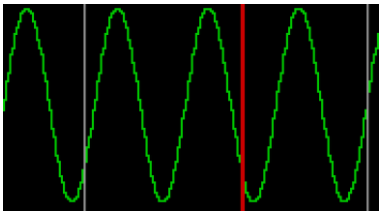
Also, in the Radix --> Real Settings section, modify the signal display settings and its fixed-point representation as you have defined.




Note that AXI bus signals such as WDATA and RDATA are 32 bits by default; consider this when generating sinusoidal and cosinusoidal samples and their signedness:




The final result of dispaying an analog signal (sinusoidal) on the ILA window:



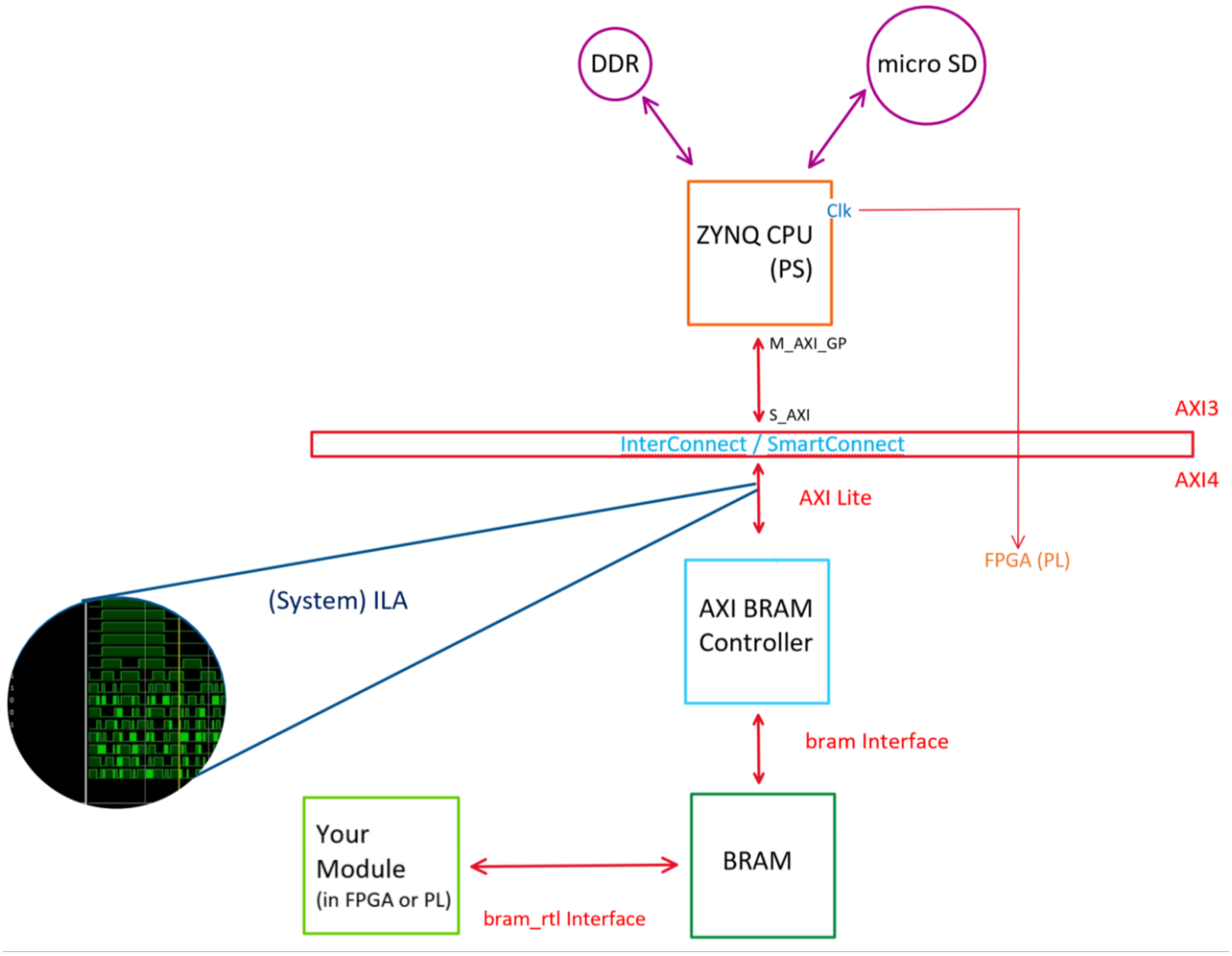
Note: The PS needs to know when the placement of the module's output on the BRAMs is complete and when to start retrieving the final data. For this purpose, you can consider a specific location in the BRAMs (for example, the last location after writing the outputs) and set the value of that location to 1 after the process is complete. (This location is always checked by the PS)

Note: As explained in video 11, the view of the PS and PL to the address of a specific BRAM location is different. For example, the first BRAM location created in the FPGA, which is available for us to use, has an address of 0 from the PL's perspective and can have an address like 0x40000000 from the PS's perspective. You can see and modify these settings in the Address Editor tab in Vivado if necessary. Consider this difference in addressing convention when establishing the address agreement between PL and PS for using specific BRAM locations.

Note: As mentioned before, the addresses from BRAM that both PL and PS use are configured by you and are explicitly defined, and both sides are aware of them by default.

So far, the FPGA's BRAMs have been used both as a memory source for storing data and as a communication path between the PS and PL so that they can inform each other when necessary (such as the completion of a process in the PL). All data interactions between the PS and PL has taken place through BRAM.

The overall flow of your project up to this point is as follows:



# Part IV (Bonus):

*You can see "Communication between PS and PL using AXI GPIO" in video 10 and "enabling interrupt from PL to PS by interrupt option of AXI GPIO" in video 13.*

In this part, we intend to expand the communication between the PS and PL beyond the limitations of BRAM and establish a direct connection (without an intermediary memory like BRAM) between the PS and PL through AXI GPIO.

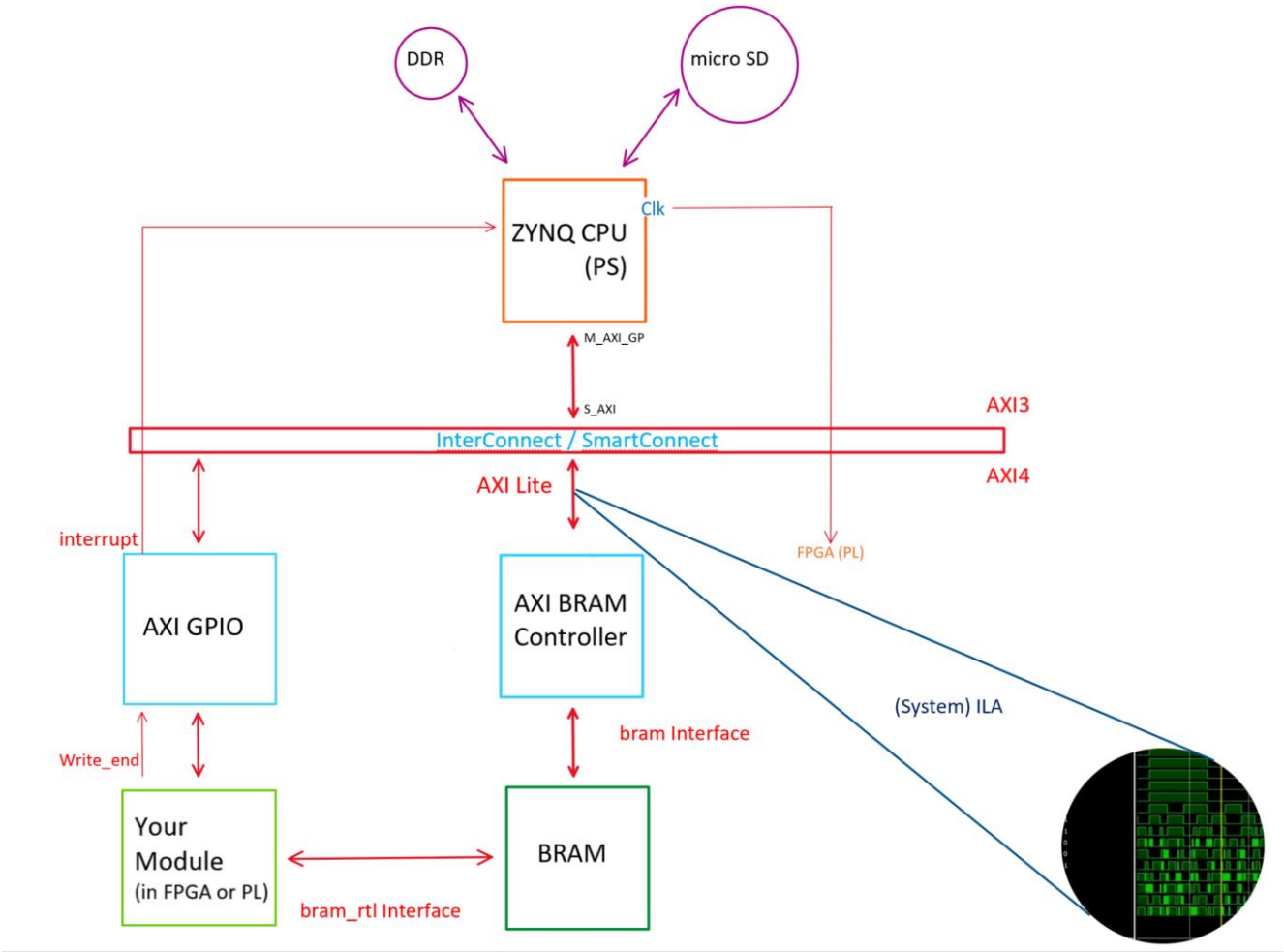
AXI GPIO, like AXI BRAM Controller, exchanges data with the PS via the Zynq's GP port.

- Share the starting write addresses (from PS to PL initially for initial data and from PL to PS for final data) between the PS and PL through this method. By doing this, you won't need to synchronize the two sides manually, and you can start writing data from any address.

- The two-bit input that determines the output signal frequency should be sent from the PS to the PL (your module) through this method.

- After the completion of writing the output data on the BRAM by the PL (your module), this event should be communicated to the PS through a signal named write_end and in the form of an interrupt (via AXI GPIO) so that the PS can then read the final data. (As done in video 13)

The overall flow of your project will be as follows after doing the bonus section:



# Part V (Bonus ×2):

*You can watch video 7 to learn "how to create a custom IP and config it from PS", and especially you can see "Initial configuration of a PL BRAM Controller from PS" in video 13.*

Now you will notify the PS of the completion of the output signal generation process from PL through AXI GPIO and as an interrupt, and your only use of AXI GPIO will be for this purpose.

You can obtain the PS's write start address in BRAM and the start signal that allows PL to start reading data from BRAM directly (without AXI GPIO intermediary) from your own custom IP's AXI bus and its 4 registers, which Vivado provides to you during the custom IP creation. Take these directly from the PS and its GP port and use them in your module. (You can obtain the PS's write start address in BRAM in SDK and get it from the user via UART and the corresponding terminal.)

The overall flow of your project will be as follows after doing this bonus section: