In the name of GOD

EE25266 – ASIC/FPGA Chip Design

Dr. Mahdi Shabany

Electrical Engineering Department

Sharif University of Technology

Lab#6 (2nd lab on Zynq) – Spring 2024

Image Processing on Zynq

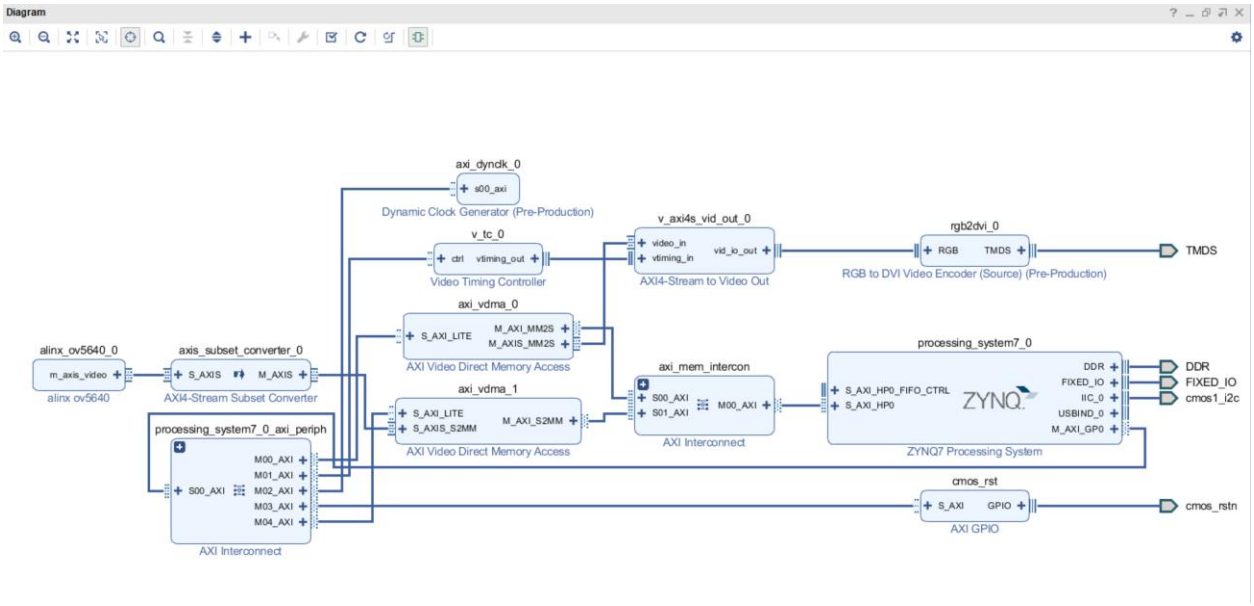
Edge Detection Algorithm Using Camera Interface

Prepared by Mina Khajeh Salehani & Hossein Moghim

Week I:

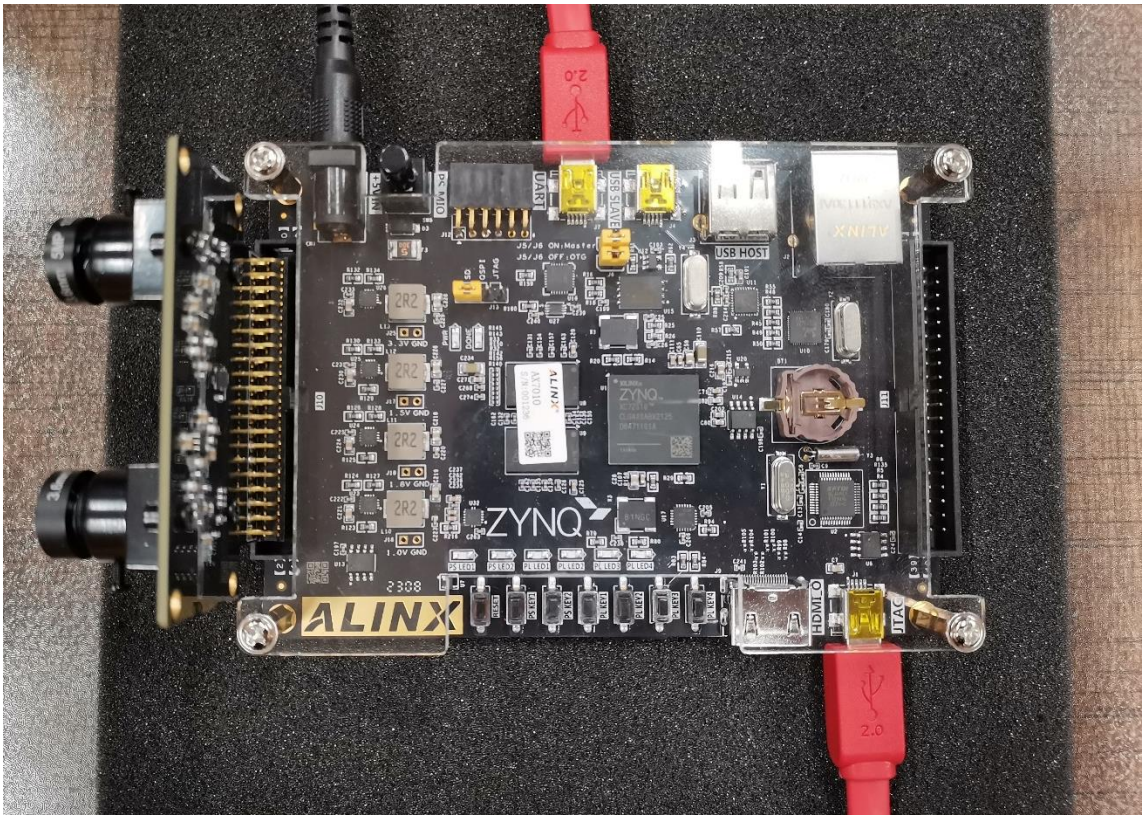
First, open *the template project of video16*.

In this project, image data is captured from the camera board, and the shot of each moment is placed in DDR3 memory using VDMA and then displayed live on the HDMI output. By pressing PS KEY1, the image of that moment (stored in the DDR3 frame buffer) is saved in BMP format on the micro SD. You can see the diagram of this project in the image below:



For further acquaintance with this project, you can watch video number 16. (It is recommended to watch video number 15 beforehand to become familiar with AXI DMA, and videos 18.b and 18.c to master the protocols of image transmission in FPGA and HDMI Interface.)

You can access the datasheet of camera (AN5642) module through this link.



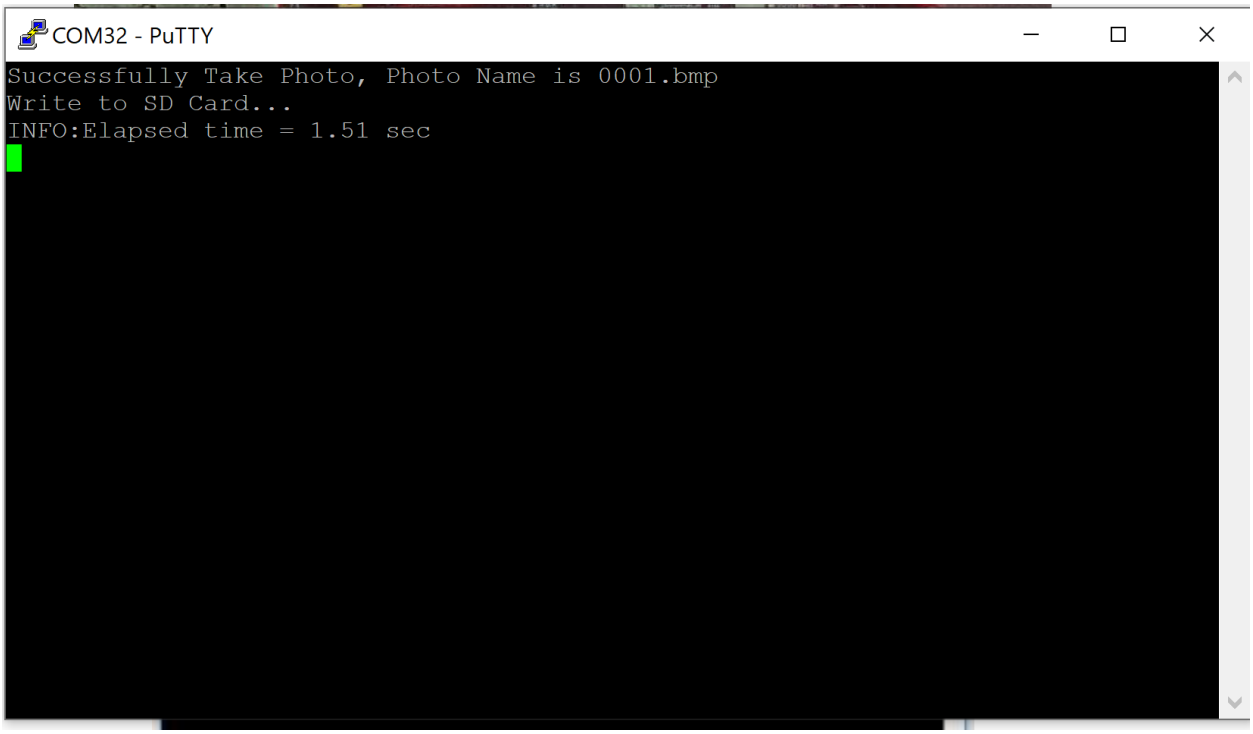
After properly connecting the camera module to the board (as depicted in the above images) and connecting the HDMI output to the monitor, you can proceed to execute the project according to the following steps:

Onboard Verification:

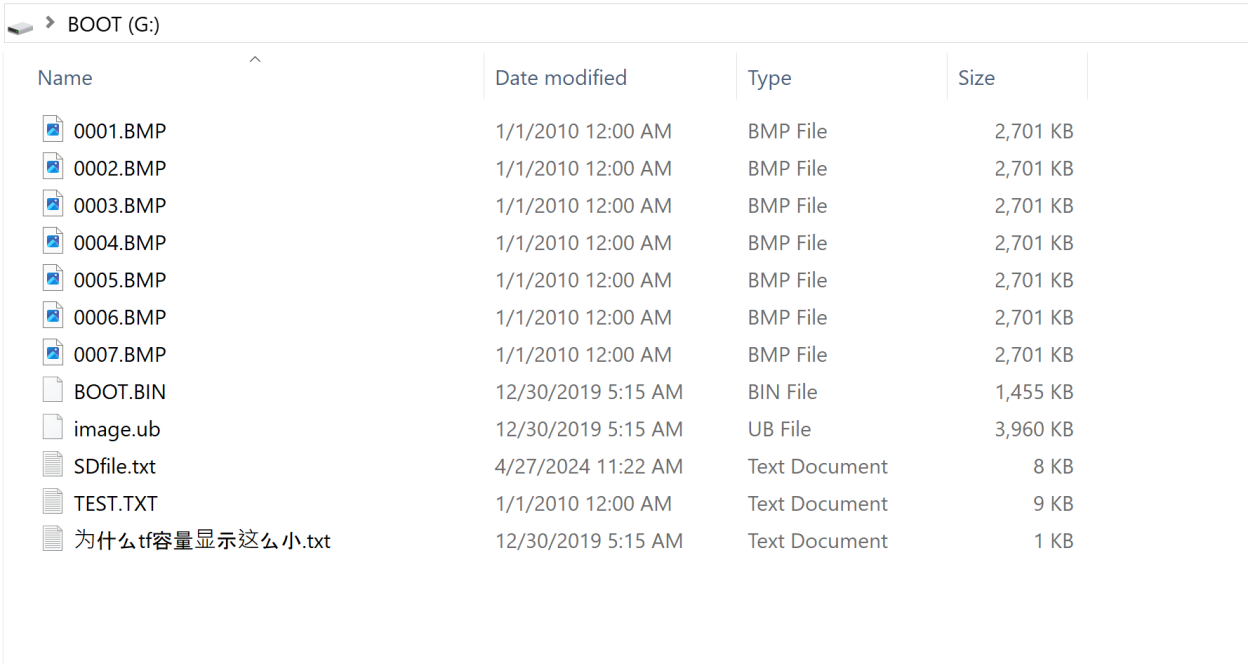
insert the SD card into the SD card holder on the back of the FPGA development board, turn on the power, and turn on the “putty”.

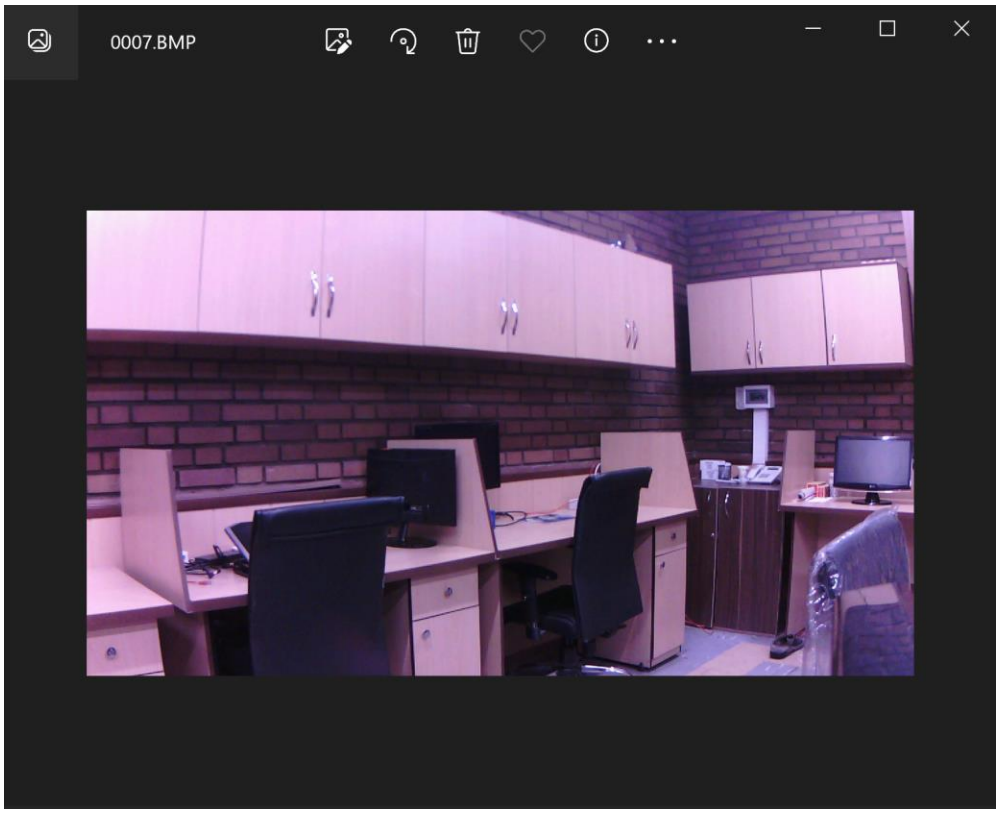
Download the program, after the image is displayed, press the button (PS KEY1), the PS LED1 will light when writing SD, and will be extinguished after writing.

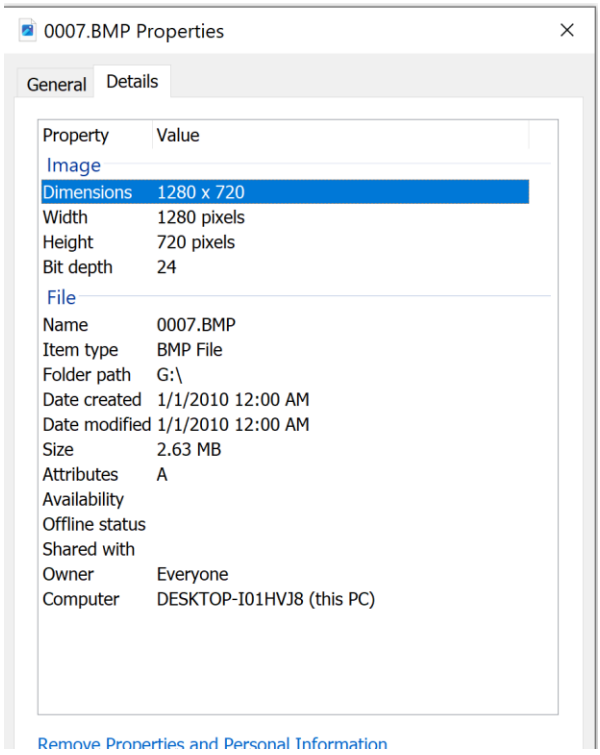
In putty, you can see the print information (After press PS KEY1 and take a photo):



Remove the SD card after power off, and you can see the BMP picture captured in the SD card on the computer:




0007.BMP


0007.BMP Properties
General
Details
Property
Value
Image
Dimensions
1280 x 720
Width
1280 pixels
Height
720 pixels
Bit depth
24
File
Name
0007.BMP
Item type
BMP File
Folder path
G:\
Date created
1/1/2010 12:00 AM
Date modified
1/1/2010 12:00 AM
Size
2.63 MB
Attributes
A
Availability
Offline status
Shared with
Owner
Everyone
Computer
DESKTOP-I01HVJ8 (this PC)
Remove Properties and Personal Information

Due to the processing tasks intended for the next weeks on this image, it is necessary to change its dimensions to 512×512 and convert the image to black and white with a Bit depth of 8. (These modifications are required because for image processing, it is necessary to input the pixel data line by line into the FPGA and its BRAM memories; hence, we encounter a memory shortage issue.)

For this purpose, you can utilize specialized image editing software or use *this website*.

Bonus: Perform the process of converting the image to black and white, and changing its dimensions and bit depth, on the image stored in the micro SD card using either FPGA or PS. (You can utilize *this module* to assist in converting the image to black and white.)

Now, after testing the mentioned project, proceed step by step according to *video number 20*, and create the bootloader (FSBL) and Boot.bin file for this project. Place it in the root directory and the first partition of the primary micro SD drive.

Set the boot jumper on the board to the SD position and power up the board to automatically load the camera template project and image saving into the micro SD. The FPGA will be programmed, and the program will be executed.

Week II:

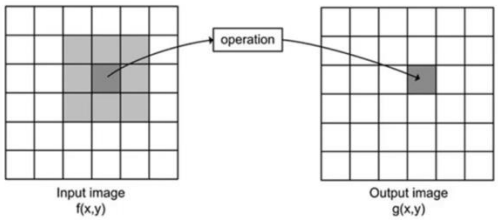
Introduction:

Images in grayscale mode are just two dimensional matrices, which each pixel has 1 byte data. In image processing we might use point processing or neighborhood processing.

In point processing, we calculate the new pixel value (g(x, y)) based on the value of the pixel in the same position (f(x, y)):

$$g(x,y) = T[f(x,y)]$$

In neighborhood processing (as shown in the below figure), we calculate the new pixel based on not only its value but based on its neighbors as well as a kernel.



Basically, you are doing 2-D convolution between the image pixels and the kernel.

$$g(x,y) = \sum_{j=-1}^{1} \sum_{i=-1}^{1} m(i,j) \cdot f(x+i, y+j)$$

You should consider that we can't use pure streaming architecture for neighborhood operation since pixels for processing aren't consecutive. For solving this problem, we need buffer pixels inside the IP (the processing module) before processing it. But it isn't practical to buffer the entire image inside the IP, so we buffer just enough pixels for processing. For example, if you decide to use 3×3 kernel, you just need to buffer 3 lines to start processing.

These memories can be built using BRAMs or distributed RAMs. If you have a 512×512 image, one line buffer will be 512 Bytes size. So it only depends on the width of the image.

For start processing, we need to send 3 lines of pixels from DDR to IP. Then after IP processes it, sends it back to DDR and receive the next data. Now we can improve the system performance if we add a fourth line buffer. While the system is processing 3 line buffers, data can be sent to the 4^{th} buffer so that data transfer happens in parallel to data processing.

For this purpose, we need to configure the DMA controller (axi_dma) to stream first four lines to the user IP. Then IP streams back the processed line to the DMA controller and interrupts the

Zynq PS. After that DMA controller receives the stream data from the IP and sends it to the external DDR memory trough the AXI4 interface.

Sobel Algorithm:

The image processing that we want to talk about is edge detection and one popular method to do edge detection is Sobel operator. Basically, we are using two kernels here in two directions.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \times A \quad , G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \times A$$

The G_x kernel is used for detecting vertical edges and G_y kernel is used for detecting horizontal edges.

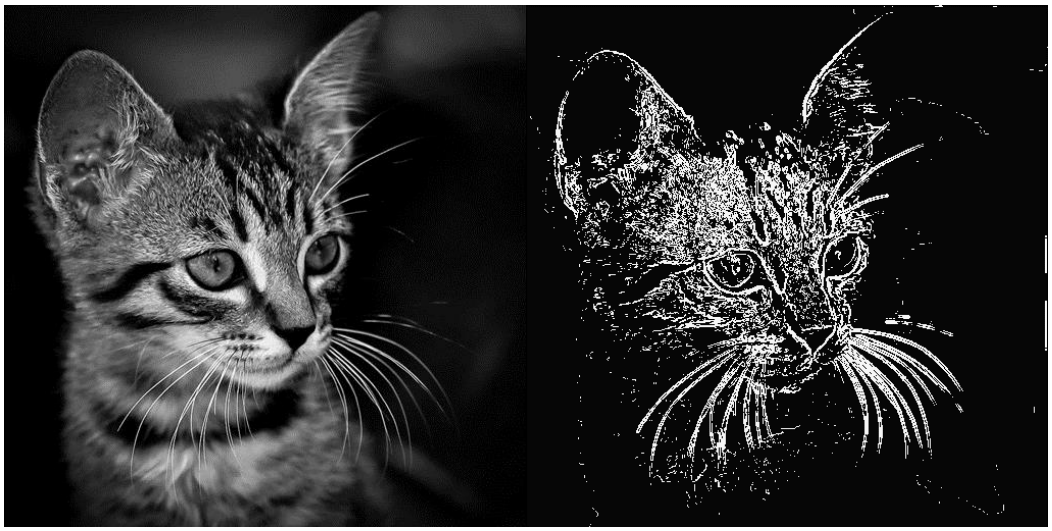
$$G = \sqrt{G_x^2 + G_y^2}$$

You need a CORDIC IP core to calculate the square root, but it uses many resources. So you can set a threshold. When G^2 is higher than that threshold, set the output color to white. Otherwise set the output color to black.

It is easier to process images in grayscale mode, so we suggest that if you have a color image, change it to grayscale mode and then process it.

Note: The only file you need to write is the one that is just doing the convolution.

You can see some examples to understand better what we expect from you in this part.



Original Image *Edge detected Image*

Note: If you are willing to use different kernels or other sizes of the input image, you should change below modules in a way that is compatible with your code.

Line buffer:

This module is called "lineBuffer.v" in starter kit.

Before start coding, we need to make some assumption. First assumption is that we will be writing 1 pixel at a time. We also assume that the size of target image is 512×512 in grayscale mode. Since we use 3×3 kernel, so the output of each line buffer would have 3 Bytes.

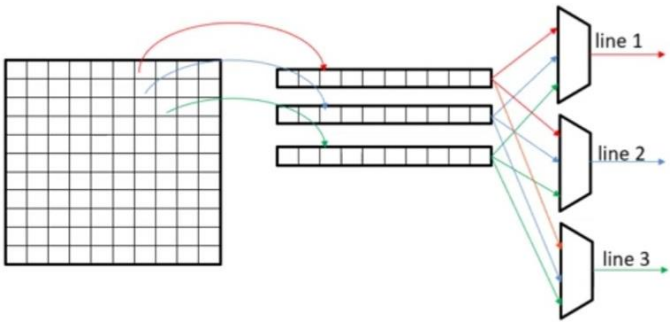
This module works like FIFO memory. We have read and write pointers. In writing mode, we write input data to the cell to which the write pointer refers and then add 1 to the write pointer. In reading mode, we read from where the read pointer refers to and add 1 to the read pointer. In this module, we can read from 3 cells at once and store in output data.

We store one line of image in our line buffer, then by shifting the read pointer one by one, output data will change to the next 3-pixels of that line.

Control Logic:

This module is called "imageControl.v" in starter kit.

Look at the figure below. We are going to instantiate our line buffers inside this module. We get a stream of input data and generate $9 \times 8 = 72$ bits of output data to convolve them with our kernel.



As we mentioned before, we need 4 line buffers. We have a counter that counts how many pixels are received. Whenever that counter hits 512, we should store data in the next line buffer. Essentially, we need a demultiplexer to tell us to which line buffer the incoming data should go. We increase the "current write line buffer" each time the pixel counter becomes 511 and the new data arrives. To specify the line buffer that the input data should be stored in, we just need to make the "input data valid" of that line buffer one and make the other zero.

Based on the value of the “current read line buffer”, we concatenate the output data of 3 specific line buffers and generate an output of this module. When the read counter hits 511 in reading mode, we should increase the “current read line buffer”. The logic of the read counter and choosing which line buffer we should read from are similar to the pixel counter and choosing which line buffer we should write to.

Consider that you should wait in writing mode until you have 3×512 pixels. After that, you can start reading from the line buffer. It means that at least 3 line buffers must be full. So, we need to count all the written pixels, too. When we want to write data, we increase this counter. When we want to read data we decrease this counter. Then we read 512 pixels from line buffers, the interrupt becomes high.

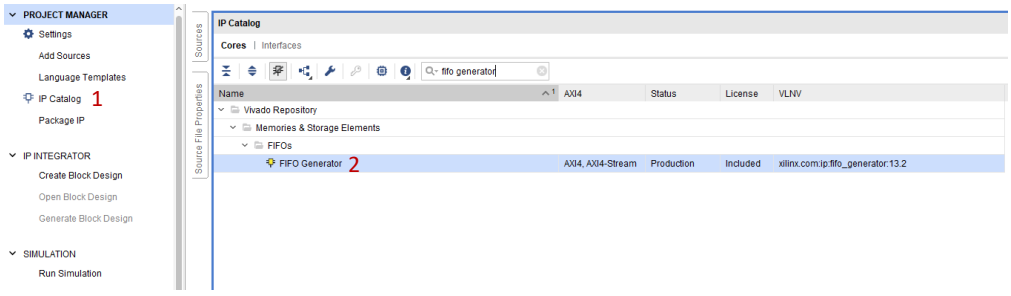
IP Packaging:

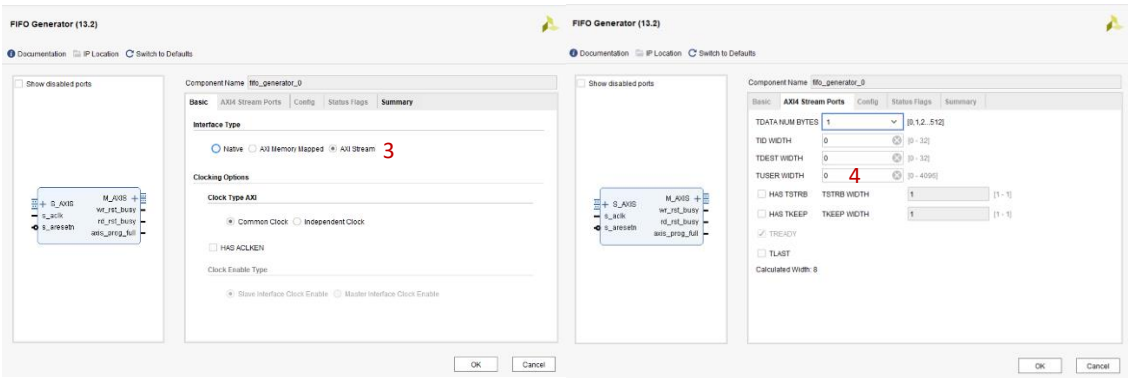
This module is called “imageProcessTop.v” in starter kit. Please watch [video number 7 “IP Core”] before continuing to the procedure to have better understanding on IP packaging.

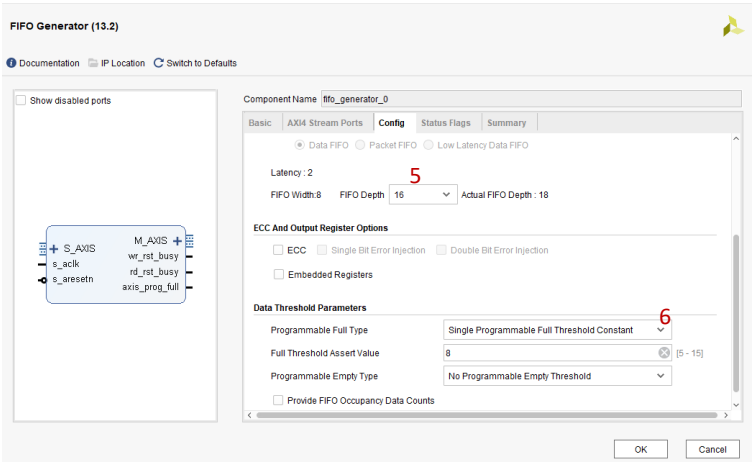
As mentioned, the processing module will be interfacing with the DMA controller. The DMA controller has an AXI stream interface. So, we should write the input and output of processing IP in AXI stream format. It must have a slave and master interface. In addition to the AXI stream interface, we have an interrupt signal, too.

We can’t connect the output data of the convolution module directly to the output of the top module. The data coming from the line buffer doesn’t immediately go to the output, because of the pipelining in the convolution module. The simplest way to solve this issue is using FIFO memory. By adding FIFO to the output, it can manage the mismatch between the input and output.

The FIFO IP core is already in the module. So you don’t need to add this IP again. If you need to know the configuration of this IP core, we follow these steps:





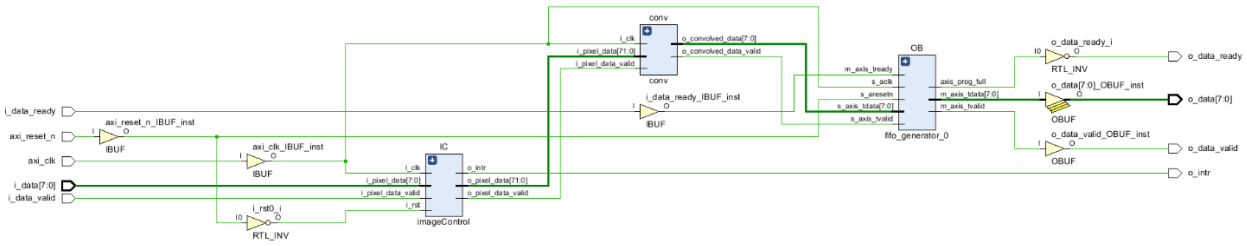


Note: The “conv” module in the code is your Sobel algorithm that you should write.

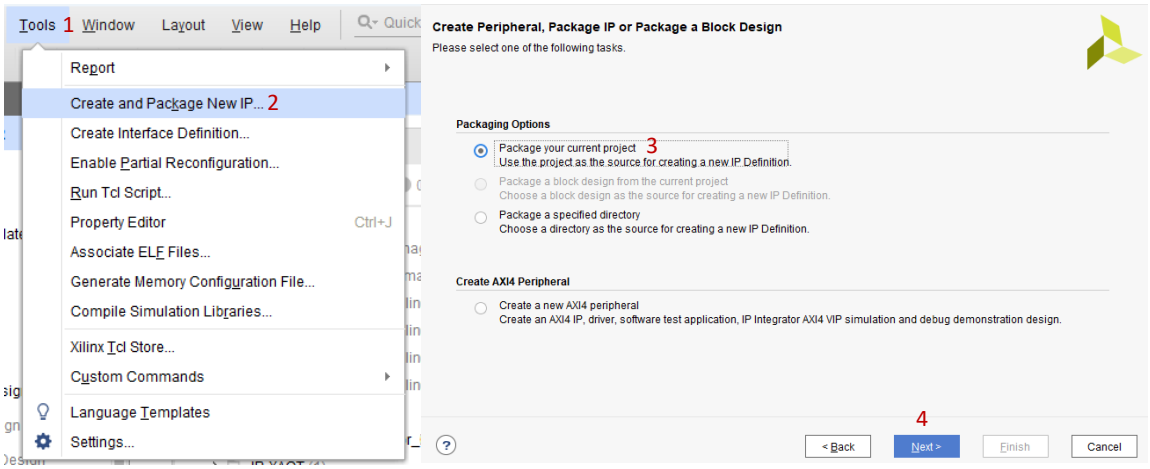
After these steps, you should simulate your code. Write a test bench that takes pixels of the input image in BMP format and generates output pixels. Then by writing output data into a file with BMP format, you can open the processed image and verify your code.

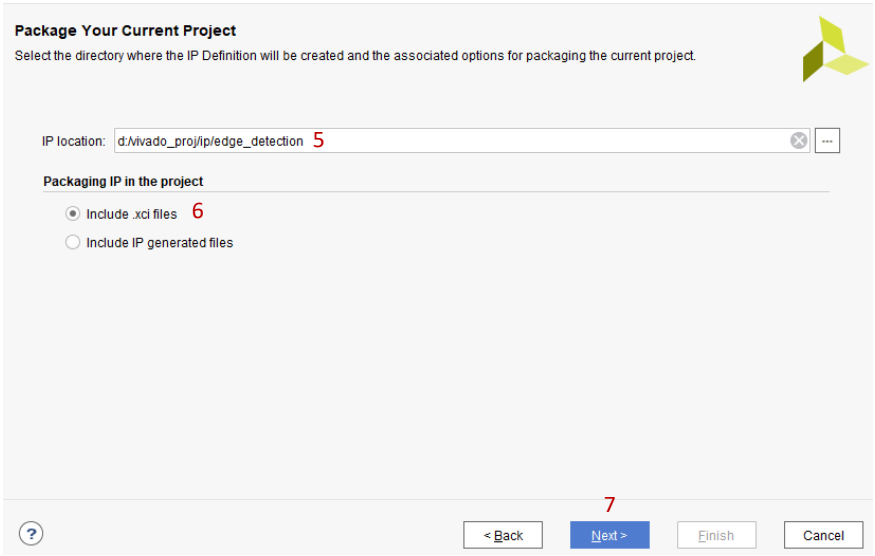
If that helps, the header size of a BMP format image in grayscale mode with an 8-bit pixel size is 1080 Bytes. It means, that if you load your input image as a file, you should consider that the first 1080 Bytes are just the header of that image and start processing after those Bytes.

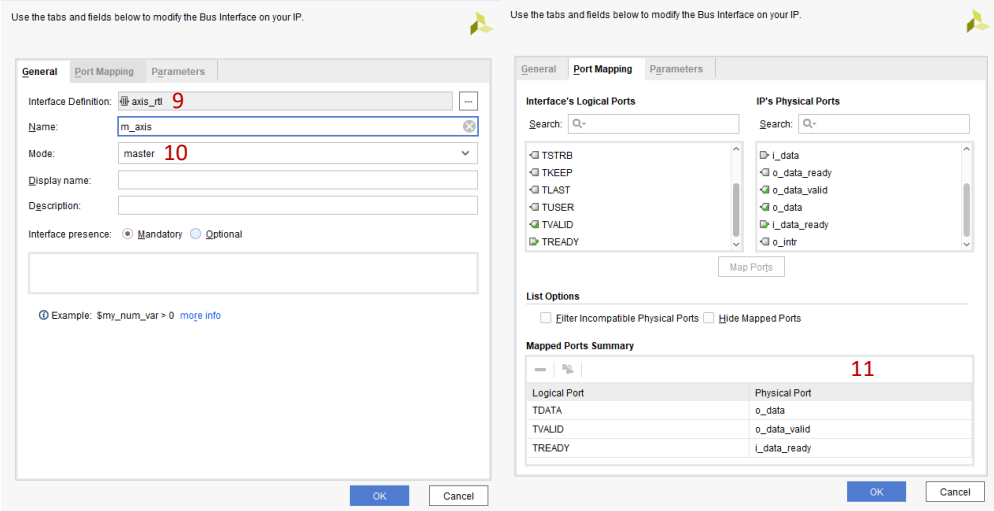
Your elaborated design would be like this:



At the end, package the whole code into the IP for further block designing. At first you should be sure that your code works properly and then package it into an IP. For this part, you need to follow these steps:

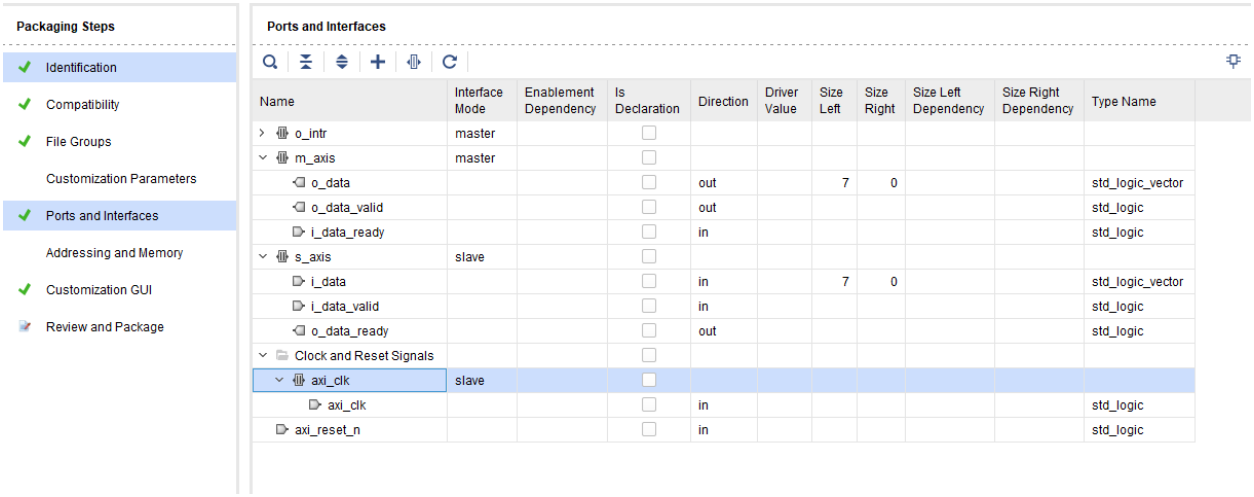




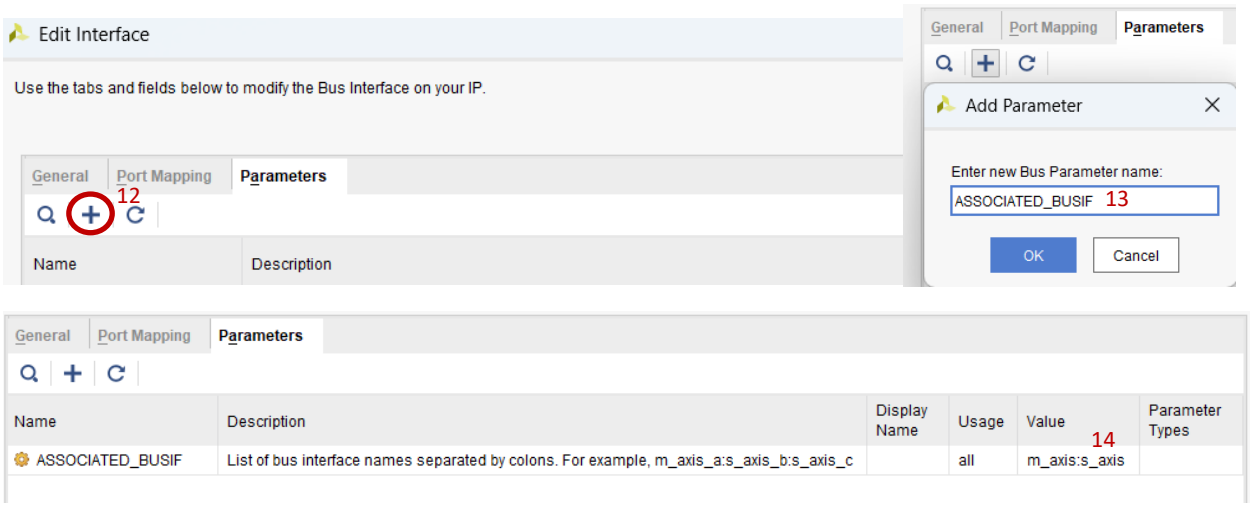


In 11^{th} step, you should map the ports as described in "Mapped Ports Summary". By choosing the logic ports, you can observe which physical ports are compatible with that. Then choose related physical port. After that press on "Map Ports". When it finishes press the "Ok".

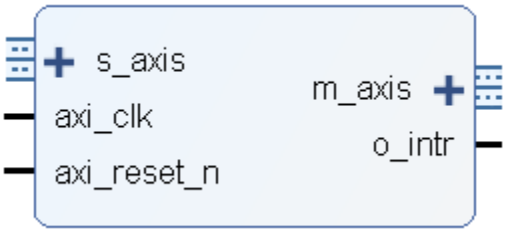
Do the same for grouping slave interface. In 10^{th} step, instead of choosing master, you should choose slave. At the end, ports and interfaces tab must be like this:



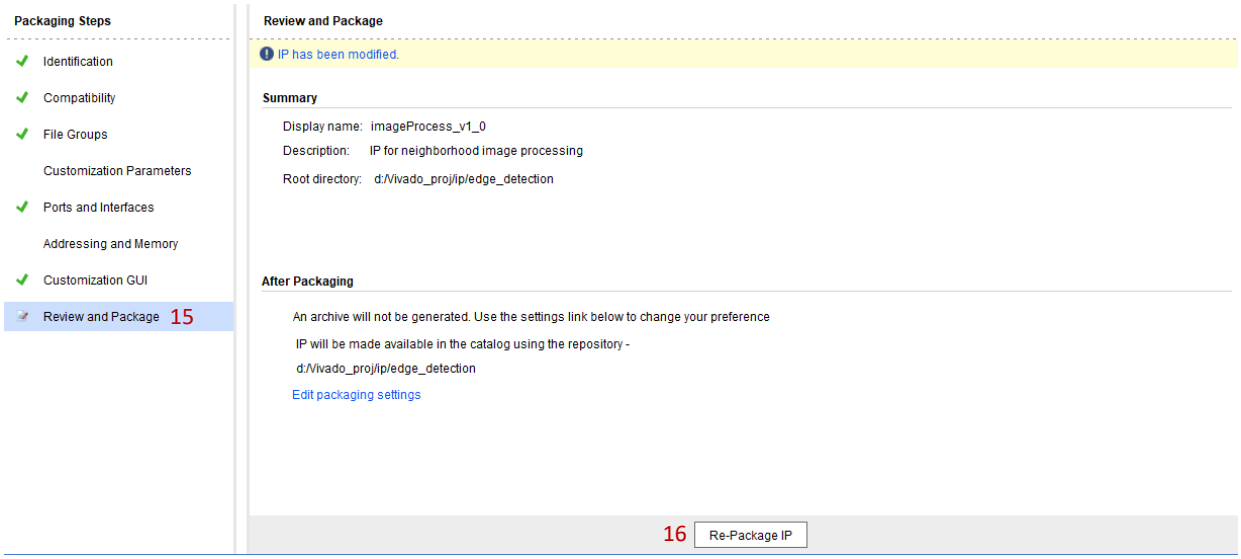
Then you need to specify which interfaces are controlled by this particular clock. Double click on axi_clk, then go to parameters tab.



After writing the value in parameters tab, press the "OK" button. Then you can see your IP preview in "Customization GUI" tab, it should be like this:



And finally:



Now it is ready to use in further block design.

Week III:

Please watch below videos before continuing to the procedure:

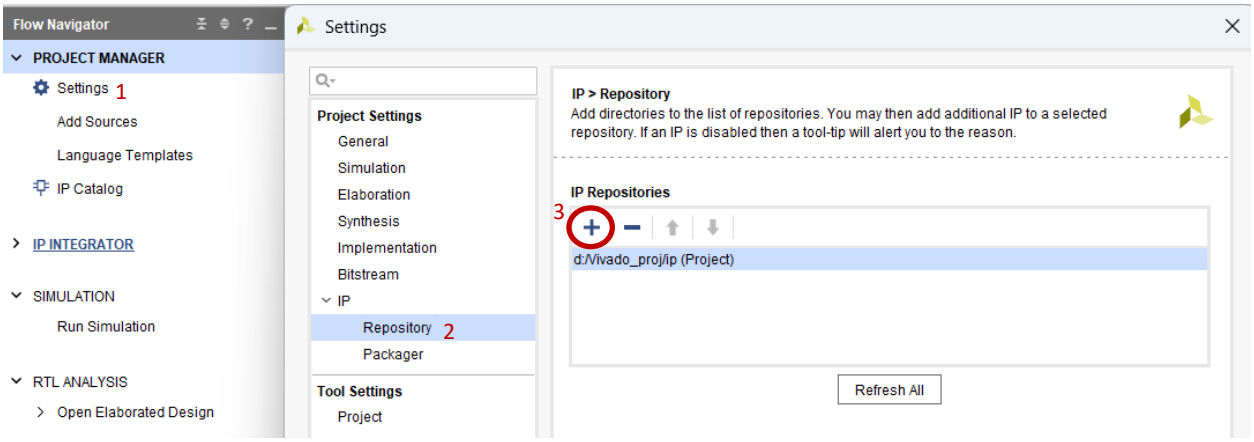
- *3 parts of Video number 15 "AXI DMA"*
- *First part of Video number 18 "micro SD card"*
- *Second and third part of Video number 18 "Image Processing in PL-HDMI out"*
- *Both parts of Video number 12 "DDR3-HP ports-AXI Master MM (PL)"*

In Week I, you learned to take a picture from camera interface and save it on an SD card. So now you have a 512×512 grayscale image. This week, we want to read this image from the SD card (*refer to the bmp_read file in the starter kit*). Then, send this data with AXI DMA from PS to your processing IP. After generating each output pixel, show it on the monitor by HDMI (*refer to the sd_hdmi_out.zip file).*

This week, you will learn how to send these data pixels from PS to PL, process them with your IP, and send them from PL to PS.

System Integration:

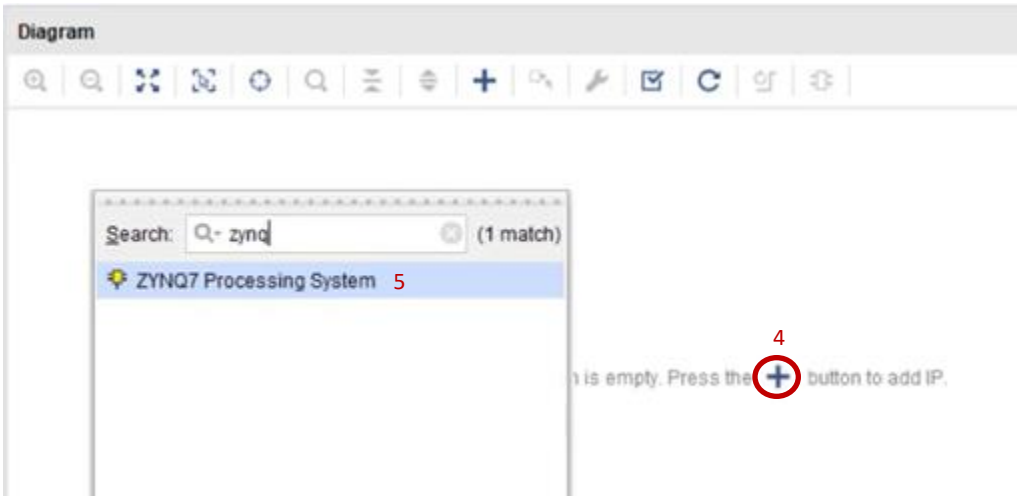
In a new project, select the "xc7z010clg400-1" from Parts. Then, you need to configure the IP repository from settings.



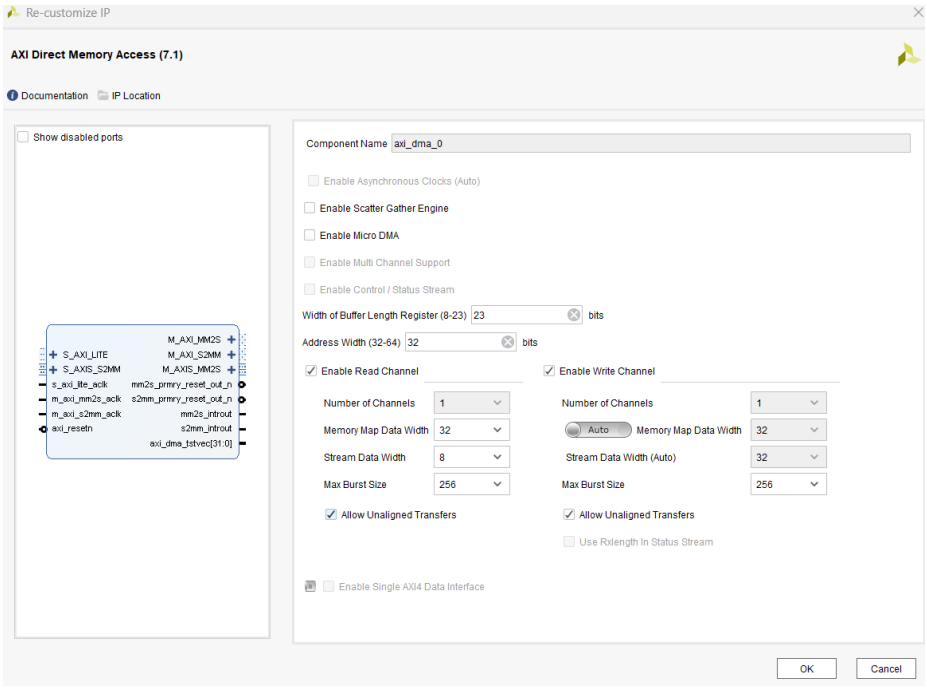
Browse the folder that contains your created IP. Then press the "OK" button.

Note: It's better to save all your created IPs in a specific folder.

Under IP Integrator in the Flow Navigator window, choose Create Block Design. The Diagram window will be opened.

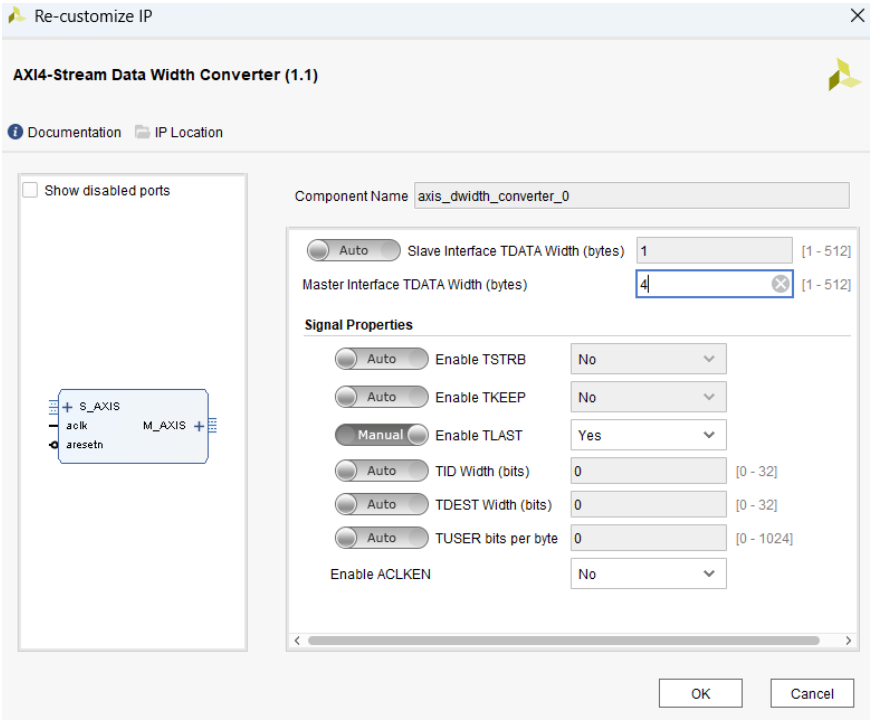


Then click on “Run Block Automation”. After that, press the plus button again and add your created IP. You also need to add AXI direct memory access (AXI DMA). Customize AXI DMA IP like the below figure:



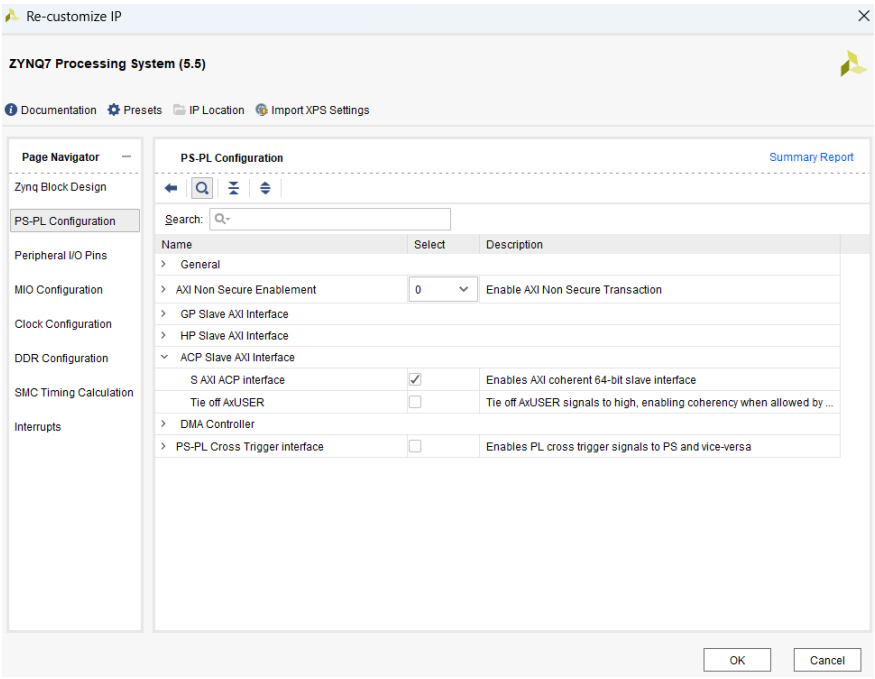
Since our IP has an 8-bit interface, you should change the Stream Data Width to 8. But in the Write channel, you can't change this parameter. Then click on “Run Connection Automation”. So that the AXI Lite of the DMA controller gets connected to the GP port of the PS.

Now, you should connect MM2S and S2MM. You need to connect the slave axis stream of your IP to the master axis memory map to stream. Since both of them have 8-bit data width, there wouldn’t be any problem. But you have 32-bit data width in the slave axis stream to the memory map. It also lets you connect it to the master axis stream of your IP core, but it causes problems. The issue is that only the lower 8 bits will be used as valid data. The upper bits will always remain zero. When your IP sends data to AXI DMA, its interface is 32-bit. So it will send 4 pixels at a time to memory. One of those pixels would be valid, but the other don’t. Because of this mismatch, you need to concatenate four consecutive pixels coming out of this master interface, convert it into 32-bit, and then connect it to the slave interface of AXI DMA. For this purpose, you can add “AXI4-Stream Data Width Converter”. Customize this IP based on the below figure:



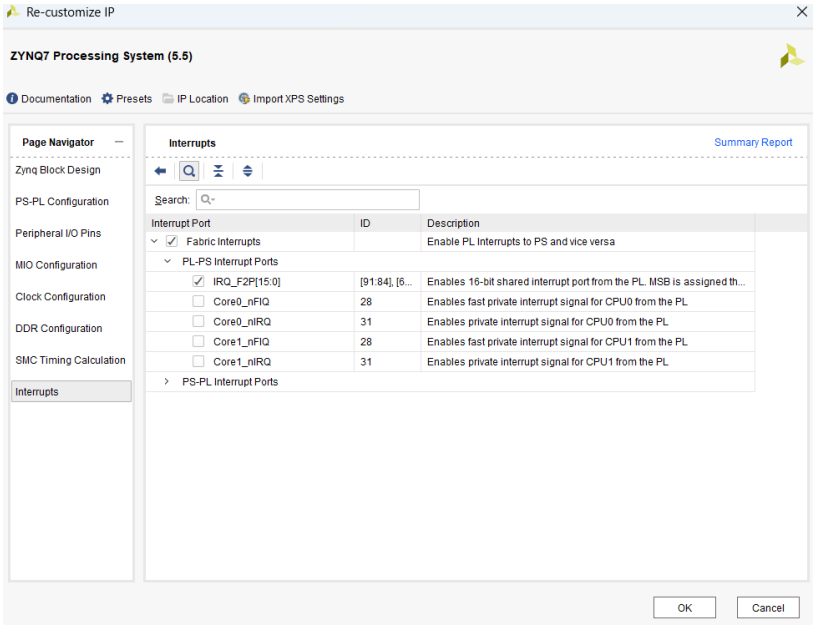
Then click on “Run Connection Automation”.

Now you want one of the ACP ports of the PS and connected master axi memory map to stream (M_AXI_MM2S) and master axi stream to memory map (M_AXI_S2MM) to ACP ports. For this, you need to customize the PS setting. First, because we are going to use DDR, use config.tcl file to configure DDR in PS. After that add ACP port to PS:

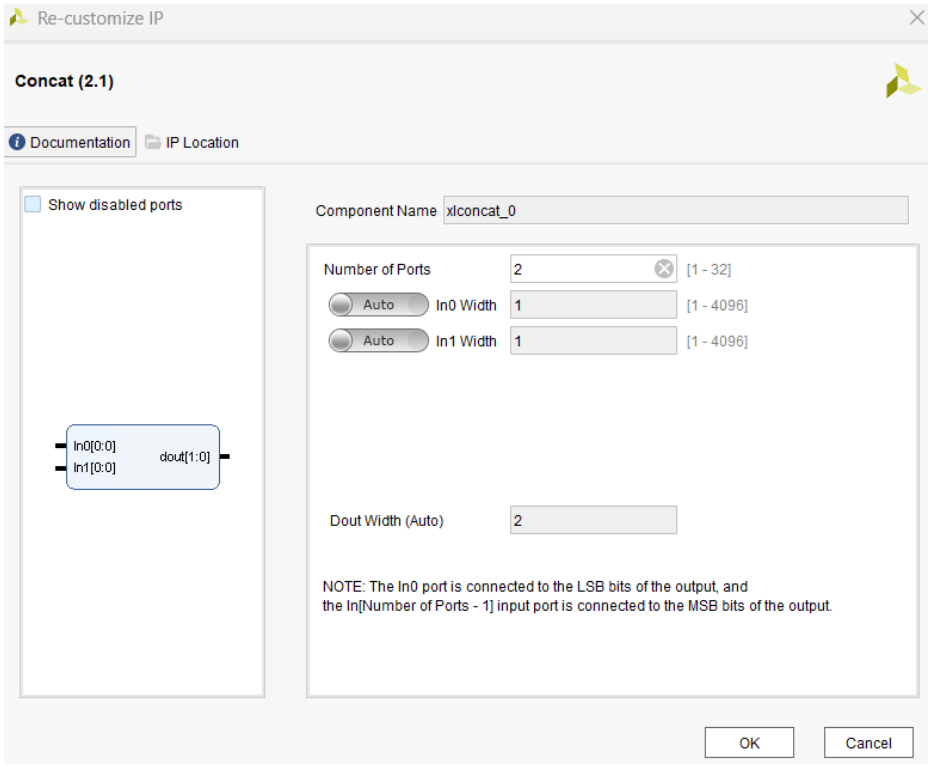


Then click on “Run Connection Automation”.

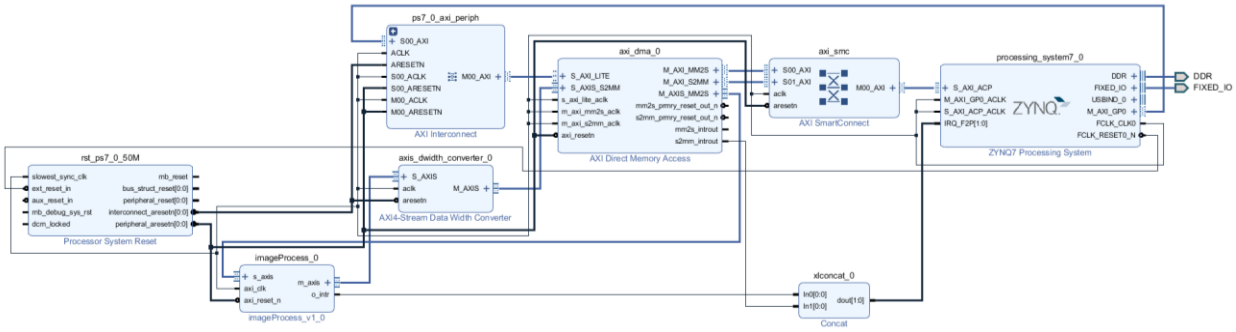
In AXI DMA IP, we have two interrupts. One of them is mm2s_introut, which tells AXI DMA has finished data transfer to the IP. The other one is s2mm_introut, which tells AXI DMA has finished transmission from your IP to the memory. When the second interrupt gets asserted, you can be sure that data processing is over. So you only need to work with two interrupts. One of them is s2mm_introut, and the other is the interrupt register of your created IP. First, you should enable the interrupt in the PS:



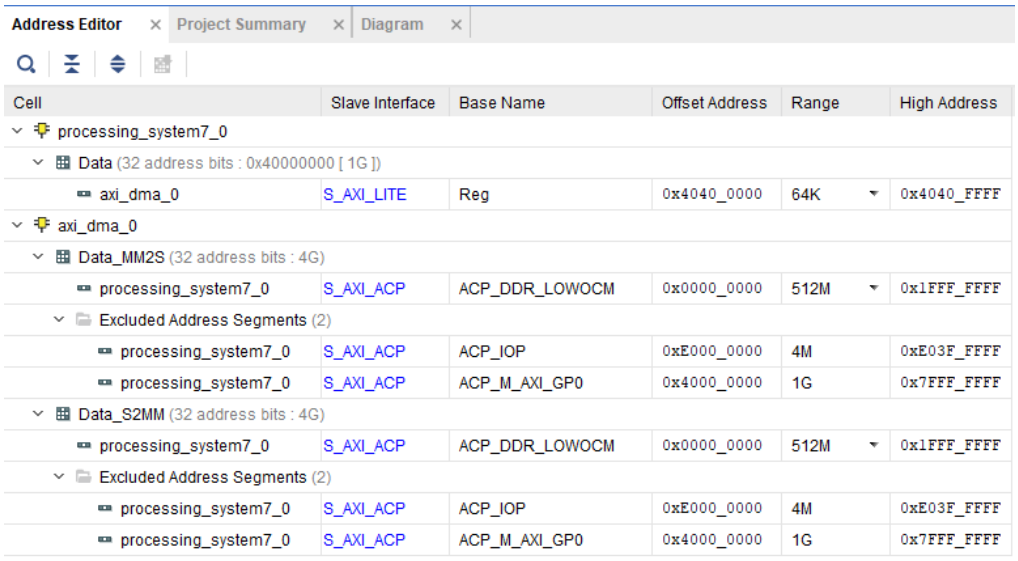
Then, you should concatenate those two interrupts. So add Concat IP:



After that, connect those two interrupts into the inputs of Concat IP and connect output of this IP to IRQ_F2P of PS. At the end, the diagram would be like this:

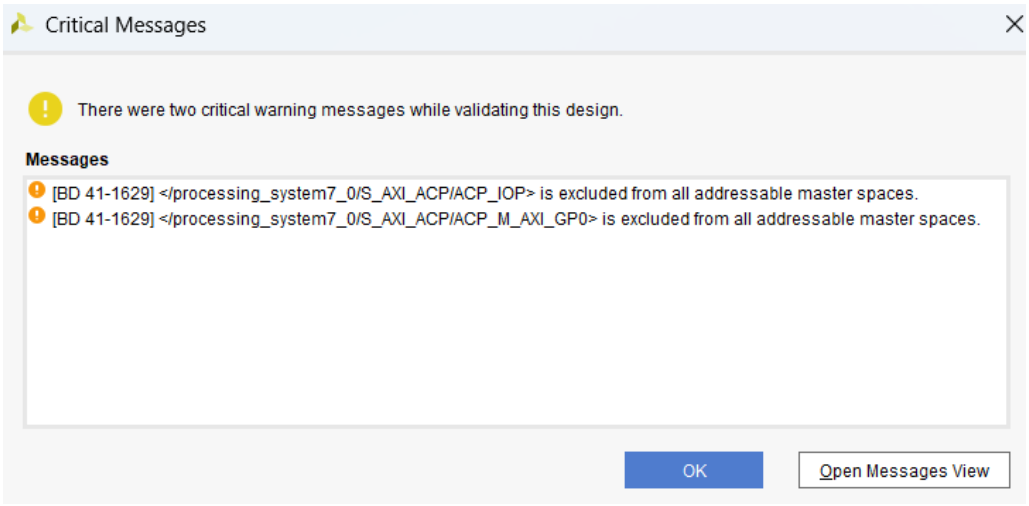


You should also check address editor tab. If it looks like below and doesn't have missing address, it's all fine.

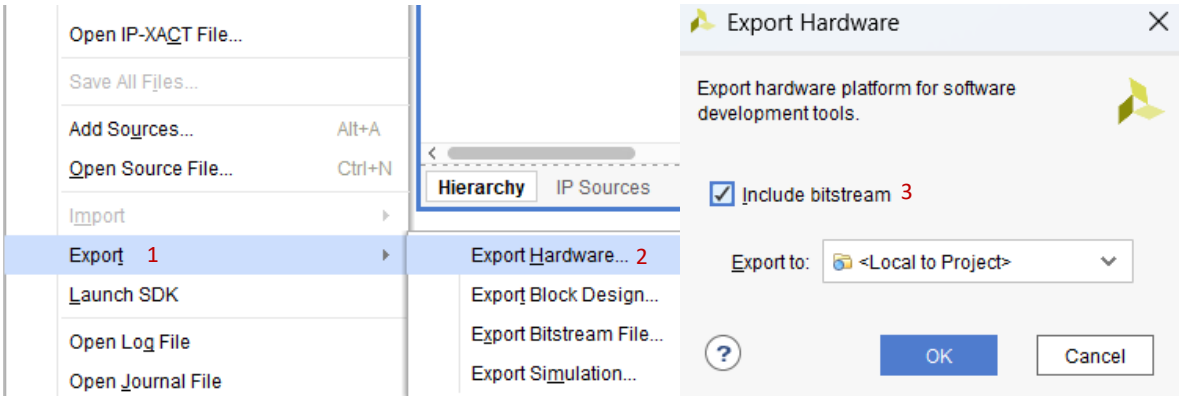


Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Excluded Address Segments (2)					
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Excluded Address Segments (2)					
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF

Now you can click on validate button of diagram tab and you may get below warnings:



Under design sources, right click on the block design that appears with your selected name, and click on Create HDL Wrapper. Then you can generate bitstream. After that export your bitstream to SDK.



Software Development:

The related file to this part is called "imageIpTest.c" in starter kit.

Note: In this part, you should change the "imageIpTest.c" code based on what you learned in the week I and videos.

You will use interrupts and by default, interrupts aren't enabled in the DMA controller. So we configure them. Remember that we only use s2mm_introut. By using the XAxiDma_IntrEnable function in xaxidma.h file, we can enable interrupt of DMA. The "Mask" input in this function tells which interrupts are enabled and which are disabled. We enable "Mask" for input/output interrupt.

```
*/
#define XAXIDMA_IRQ_IOC_MASK        0x00001000 /**< Completion intr */
#define XAXIDMA_IRQ_DELAY_MASK      0x00002000 /**< Delay interrupt */
#define XAXIDMA_IRQ_ERROR_MASK      0x00004000 /**< Error interrupt */
#define XAXIDMA_IRQ_ALL_MASK        0x00007000 /**< All interrupts */
/*@}*/
```

We have two directions and we choose the device to DMA direction in this function.

Then, we configure the interrupt controller. We have two interrupts, and we need to set priority for both of them. In the xparameters.h file, you can see the definitions for both of the interrupts. We set the priority for our created IP to zero. The "3" as the last input of the SetPriorityTriggerType, sets that the interrupt would be edge triggered. Then, we connect this interrupt to a specific function. In the end, we are calling the function which enables interrupts. we do the same for s2mm_introut, but set its priority to 1.

At first, we get the whole image from the device and save it to memory. Then, we send four lines from the image $(4 \times image_{width})$ until an interrupt is asserted. After that, we can send the next line of pixels.

The first interrupt function that we need is imageProcISR. This function will be automatically called when our IP gets interrupted. This interrupt tells us that our IP has finished processing one

line buffer. So we can send a new line buffer. When you get to interrupt, you should disable it, do what you want, and then enable it again. Imagine the case, that you want to send the new line buffer, and the DMA controller may be still sending some previous image data. In this case, you may stop previous data transmission and restart it, which causes data loss. To solve this problem, we use the “checkIdle” function.

Table: MM2S_DMASR Register Details

Bits	Field Name	Default Value	Access Type	Description
0	Halted	1	RO	DMA Channel Halted. Indicates the run/stop state of the DMA channel. • 0 = DMA channel running. • 1 = DMA channel halted. For Scatter / Gather Mode this bit gets set when DMACR.RS = 0 and DMA and Scatter Gather (SG) operations have halted. For Direct Register mode (C_INCLUDE_SG = 0) this bit gets set when DMACR.RS = 0 and DMA operations have halted. There can be a lag of time between when DMACR.RS = 0 and when DMASR.Halted = 1. Note: When halted (RS= 0 and Halted = 1), writing to TAILDESC_PTR pointer registers has no effect on DMA operations when in Scatter Gather Mode. For Direct Register Mode, writing to the LENGTH register has no effect on DMA operations.
1	Idle	0	RO	DMA Channel Idle. Indicates the state of AXI DMA operations. For Scatter / Gather Mode when IDLE indicates the SG Engine has reached the tail pointer for the associated channel and all queued descriptors have been processed. Writing to the tail pointer register automatically restarts DMA operations. The IDLE bit is associated with the BDs. The DMA may be in IDLE state, there may be active data on the AXI interface. For Direct Register Mode when IDLE indicates the current transfer has completed. • 0 = Not Idle. For Scatter / Gather Mode, SG has not reached tail descriptor pointer and/or DMA operations in progress. For Direct Register Mode, transfer is not complete. • 1 = Idle. For Scatter / Gather Mode, SG has reached tail descriptor pointer and DMA operation paused. for Direct Register Mode, DMA transfer has completed and controller is paused. Note: This bit is 0 when channel is halted (DMASR.Halted=1). This bit is also 0 prior to initial transfer when AXI DMA configured for Direct Register Mode.

When the output of this function becomes one, we can send the new line buffer, and the last sending is finished. We choose address $0X4$ because this is a memory-to-device direction.

After these steps, we write the “dmaReceiveISR” function. This function is related to the s2mm_introut interrupt. In this function, we know that all the data from our IP is received in DMA. We set a signal named “done” to 1. In this case, if we want to use interrupt again, we must clear the status register. Otherwise, it remains one. For this purpose, we use the interrupt acknowledge function (“XAxiDma_IntrAckIrq”). When we run the entire software next time, we can use this interrupt again.