TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Yangpeng Miao 177358IASM

HIGH SPEED HARDWARE IMAGE FILTER SYSTEM BASED ON SOC

Master's thesis

Supervisor: Aleksander Sudnitsõn
PhD

Tallinn 2019

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Yangpeng Miao 177358IASM

KIIPSÜSTEEMIL PÕHINEV KIIRE RIISTVARALINE PILDIFILTER SÜSTEEM

magistritöö

Juhendaja: Aleksander Sudnitsõn
PhD

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: "Yangpeng Miao"

06.05.2019

Abstract

Median filter is a well-known and widely used nonlinear image filter. Comparing with linear image filters, median filter is more effective for eliminating image salt-and-pepper noise and able to preserve more image details. However, median filter requires a huge amount of calculation. Most of embedded systems hard to accept it. Finding out an effective and practical solution for doing median filtering will let median filter become available in many resource-limited real-time systems.

Using FPGA as hardware accelerator to help CPU to finish median filtering task will significant increase the speed and reduce resource requirement.

In this thesis, I finished the design in Xilinx ZC702 evaluation board, which carries Xilinx XC7Z020 SOC. In the Processing System (PS) side, Linux OS is used, a C program is done for transmitting and receiving image data. A hardware digital image filter is done in Programmable Logic (PL) side. Direct memory access (DMA) channels are established for high speed data exchange between PS side and PL side.

According to the experiment result, this design is able to do the 3x3 median filtering for input image in high speed. Besides, it provides convenient interface for users to interact with the device. Whole system runs very stable and total reach task requirements.

Temporary, this design is only available for processing picture, but the “Digital Image Filter” IP core is divided by models, many improvements can be done conveniently.

This thesis is written in English and is 72 pages long, including 7 chapters, 68 figures and 1 tables. Design source code can be downloaded in https://github.com/miaoyangpeng/AXI_digital_median_filter_IP

List of abbreviations and terms

AXI	Advanced eXtensible Interface
CMA	Contiguous Memory Allocator
CPU	Central processing unit
DDR	Double Data Rate
DMA	Direct memory access
EXT	extended file system
FAT	File Allocation Table
FPGA	field programmable gate array
FSBL	first stage boot loader
GPU	Graphics processing unit
IP	intellectual property
JTAG	Joint Test Action Group
LAN	Local area network
OS	Operating system
PC	Personal Computer
PL	Programmable Logic
PS	Processing System
RAM	Random-access memory
ROM	Read-only memory
SOC	System on a Chip
UART	universal asynchronous receiver-transmitter

Table of contents

List of figures

List of tables

1 Introduction

1.1 Task definition

The task is based on Xilinx ZYNQ SOC, to complete a hardware image filtering system. Using hardware median filter to eliminate image salt-and-pepper noise without CPU involvement. Besides, it needs to provide user interface and Linux commands for sending original image and receiving the result.

1.2 Scope

This design is done in Xilinx ZC702 evaluation board, which carries Xilinx XC7Z020 SOC.

Figure 1-1 shows the basic structure of this system. It is divided into Processing System (PS) side and Programmable Logic (PL) side.

PS side is responsible for the communication between the device with PC via LAN or UART, transmit and receive image data via DMA channel. There are two ARM Cortex-A9 CPUs in PS side, as well as Ethernet and USB interface, and 1GB DDR3 memory. Linux OS will be used in PS side.

In PL side, AXI DMA IP is Xilinx official IP core, for receiving and transmitting data with DMA channel. The main task in PL side is to create a new IP core named “Digital Image Filter”, fulfil the function of hardware image median filter.

Data transfer between PS and PL side is through AXI Interface. AXI GP is responsible for low speed data transfer. In this design, AXI GP transfers information between CPU and AXI DMA IP core, to set up some basic property of DMA channel. AXI HP performs high speed data transfer. AXI DMA IP core can get image data directly from DDR memory by AXI HP and provide them to Digital Image Filter. Similarly, AXI DMA IP core can send back image data directly to DDR memory via AXI HP from Digital Image Filter.

Design in PS side is easier, because many works have been done by others already. Xilinx also provide many useful reference designs. PL side is more difficult, as a total new IP core need to be created, and configuration and device tree writing of AXI DMA IP core have to consider the compatibility with PS side.

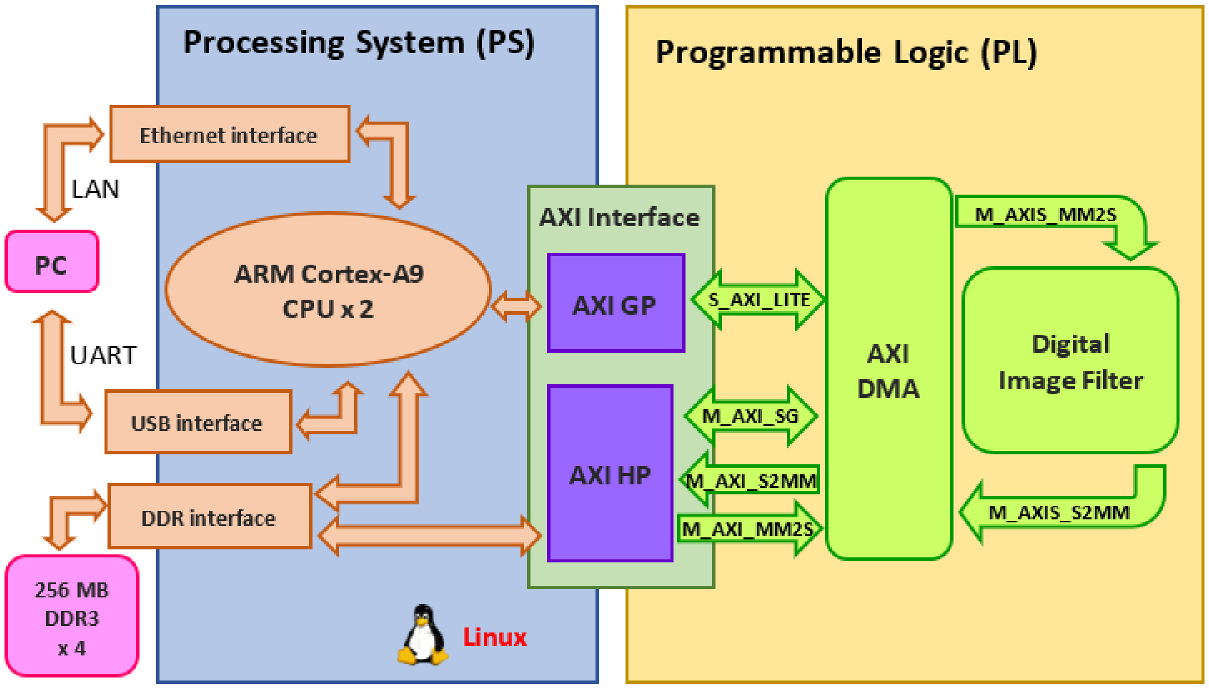


Figure 1-1: overview diagram of final system

1.3 Task Requirements

Excepting for the basic image filtering function, this system is also required to run in high speed, easy-to-use, stable and reliable. So, the following requirements should be achieved:

1) User interface convenience

   To let users can control the system. Users should be allowed to access the device via UART or LAN. Users should be able to copy original image from PC to device's ROM conveniently. One commends line should be enough to send an original image to DMA channel and receive the result.

2) Internal data communication compatibility

   DMA channels configuration must be compatible with PS side and PL side. "Digital image filter" IP input and output interface should be compatible with AXI4 interface protocol, must be able to communicate with "AXI DMA" IP correctly and effectively.

3) Image filtering requirement

   System should implement the same function as a standard median filter with 3x3 window.

4) System speed requirement

   DMA channel data transmission and receiving should be able to run in full speed. DMA channel should not stop during data transmission and receiving without any special situation, as channel stop too frequently will influent speed.

5) System stability and reliability

   System should be stable and reliable enough. Such as, start-up procedure should not have any error, DMA channel driver should be installed correctly, image filtering program can be done for many times correctly without rebooting the system.

1.4 Thesis Outline

Chapter 2 introduces the background of median filter and sorting network, compares different implementation methods in different platform and introduce some resource, theories and tools that will be useful for the task.

Chapter 3 analyses the hardware diagram structure in PL side and shows how to step-by-step finish such diagram in Vivado.

Chapter 4 explains all necessary files for PS side and how to compile them. Those files help PS side to get hardware information and run Linux OS properly. It also introduces useful Linux-C drivers and examples for using DMA channels.

Chapter 5 is the main part of this design. It analyses every part of “Digital Image Filter” IP, lists their functions and problems. And it explains some details about their implementation methods.

If everything from Chapter 3 to 5 is done, the SOC device should be able to run properly. Chapter 6 shows PL side implementation result, provides a method of doing experiment, shows and analyse the experiment result.

Chapter 7 summarize this task and provides possible work for the future.

2 Background

2.1 Median filter

Nowadays, image or video collecting device, such as Camera, has been used more and more widely. But due to the imperfections of image sensors, faulty memory locations in hardware, or errors in the data transmission, images are often corrupted by noise. The impulse noise is the most frequently referred type of noise[1]. For removing distracting and useless information from the image and make it more recognizable, image must pass through a stage of image preprocessing[3]. Comparing with linear filters, nonlinear filter has ability to preserve edges and suppress the noise without loss of details[1]. Median filter, which is a well-known and widely used nonlinear filter, is very effective for eliminating salt-and-pepper noise[2].

However, most of embedded system's calculation ability and hardware resource is very limited, while median filter requires a huge amount of calculation. This conflict makes median filter become a very difficult choice. Finding out an effective and practical solution for doing median filtering will let median filter become available in many resource-limited real-time systems.

Figure 2-1 is an example of how a kind of impulse noise looks like from an intensified digital imager. Observing the brightness value table, two especial large values are no real. They are impulse noise and influencing the quality of the image. As the brightness value of the pixel with impulse noise always especial large or small. Using the median value will be able to avoid them.

Figure 2-2 is the application principle of a 3x3 median filter (or window), which centre an image pixel. This model will be applied to each pixel of the image and replace the pixel value by the median value of its neighbours, like the procedure shown in Figure 2-3.

Size of median filter window is not fixed. Both Figure 2-2 and Figure 2-3 show 3x3 size window. Bigger size windows, such as 5x5, 10x10, are also very popular. The most suitable window size depends on the density and size of noise.

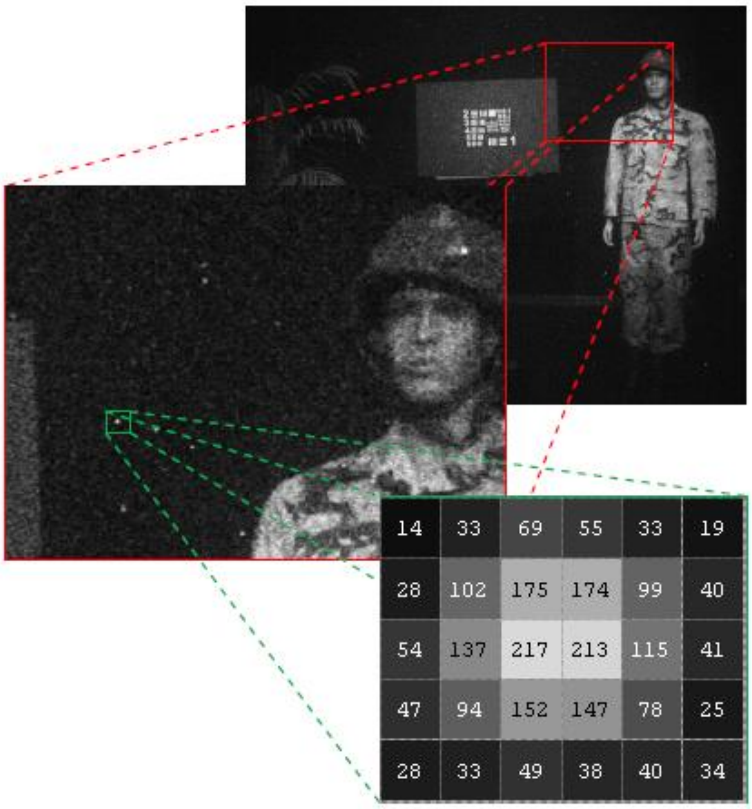


Figure 2-1: Example of scintillation noise from an intensified digital imager [4]

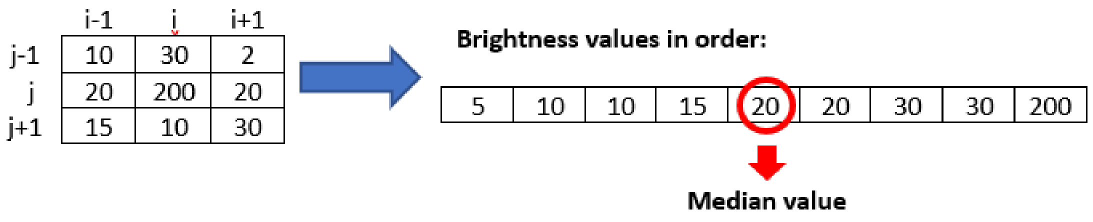


Figure 2-2: Application of the median filter [3]

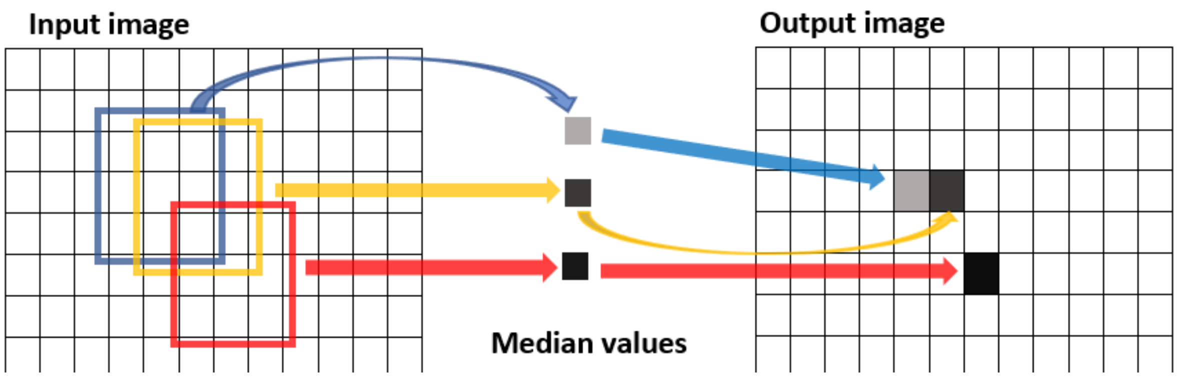


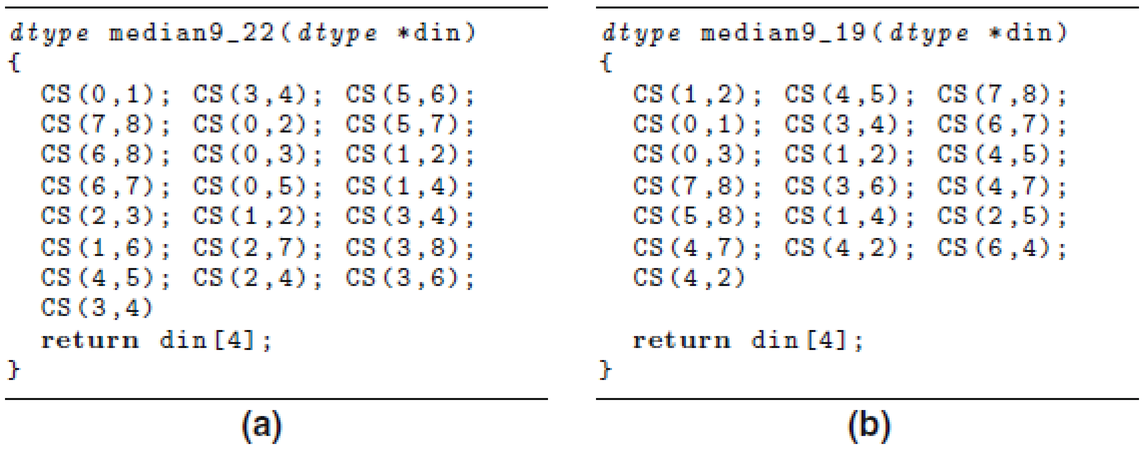
Figure 2-3: illustration about how median value replace the original pixel value.

2.2 median filter implementation methods comparison

Figure 2-4 is two software implementation method for finding out the median value amount 9 input values. Function CS (a, b) assigns the lower input value to din[a] and higher input value to din[b]. [9]

Figure 2-4(a) is a kind of standard sorting network solution (Batcher’s odd-even merge sorting [9]) with 22 operations, while Figure 2-4(b) is an optimized method of Figure 2-4(a). They fulfill the same function as Figure 2-2.

Figure 2-4 implies that using CPU platform and software method for median filtering is a very time-consuming task. The output of Figure 2-4 is only one pixel of an image. Even for a median size image, CS (a, b) operation may need to be repeated more than millions of times, especially for a color image.



```
dtype median9_22(dtype *din)
{
  CS(0,1); CS(3,4); CS(5,6);
  CS(7,8); CS(0,2); CS(5,7);
  CS(6,8); CS(0,3); CS(1,2);
  CS(6,7); CS(0,5); CS(1,4);
  CS(2,3); CS(1,2); CS(3,4);
  CS(1,6); CS(2,7); CS(3,8);
  CS(4,5); CS(2,4); CS(3,6);
  CS(3,4)
  return din[4];
}
```

(a)

```
dtype median9_19(dtype *din)
{
  CS(1,2); CS(4,5); CS(7,8);
  CS(0,1); CS(3,4); CS(6,7);
  CS(0,3); CS(1,2); CS(4,5);
  CS(7,8); CS(3,6); CS(4,7);
  CS(5,8); CS(1,4); CS(2,5);
  CS(4,7); CS(4,2); CS(6,4);
  CS(4,2)

  return din[4];
}
```

(b)

Figure 2-4: two software methods for returning the median value from 9 inputs. [9]

Even nowadays, there are many new improvements of standard median filter have been put forward, like [10] and [11], their time consuming still very high, and no significant improvement in de-noising ability.

Time consumption of GPU can be much lower than CPU, as GPU is able to create a lot of threads for calculation task. Like the experiment result in [12], for a 1024^2-size image, procedure time will be as lower as 0.2ms. But high efficiency requires high resource (calculation resource, memory, etc.) and power consumption. For imbedded system, FPGA-based implementation will be more practical. More details about the different between the performance of FPGA and GPU in this topic can refer to [13].

There are two approach for FPGA to implement a standard median filtering function[16]. The first method is provided by Smith[15], which is a Xilinx's reference implementation. It provides a median filter for nine input pixels. And it is possible to be extended to 25 or more input pixels. If the purpose of design is to minimize FPGA implementation area, [15] is a very suitable solution. But the delay of accepting input value not suitable for pipeline operation.

Another approach is based on sorting networks. Sorting network is a new kind of method and very suitable for FPGA-based median filter, as it can be designed and optimized with the aim of reducing the number of comparators or delay[16]. Without any delay for input data is the main point of this design. Based on this principle, Comparators quantity and implementation area should be minimized as well. Figure 2-8 and Figure 2-9 shows the basic idea of finding out the median value by sorting network, Figure 5-37 is the real implementation method in FPGA. More details, please refer to section 2.4 and section 5.4.

Using pipeline operation, one sorting network in FPGA can work out one median value in every clock cycle, very low time consumption. Besides, FPGA will occupy very few resources, and it is able to release CPU resource, let CPU is free do other work.

2.3 BMP file format

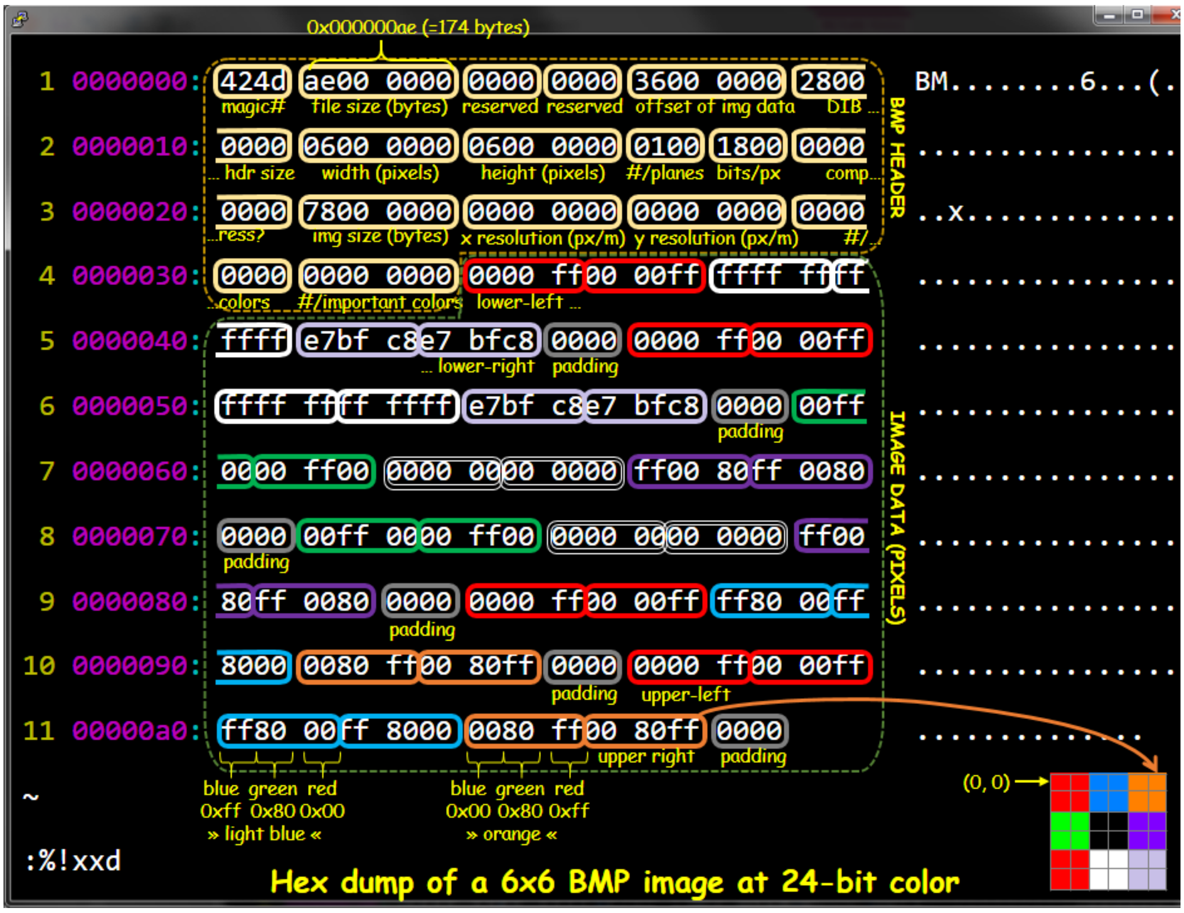
2.3.1 Basic information about BMP file format

BMP format is a kind of image storage format without any compression. So. BMP format is very suitable for this design. This system only accepts BMP format image.

Figure 2-5 is an example of a 24-bit color 6x6 BMP image file.

A BMP format image starts with “BMP header”. The size of header is indicated by byte 0E to 11. The value of those bytes in Figure 2-5 is 00000028h, which means the size of header is 40. BMP header describes all basic information about this image, such as width, height, etc.

After BMP header are values for image pixels. Every 24-bits, or 3 bytes, define one pixel. As every pixel combined by Red, Green and Blue three color, each byte indicates the value of one of those color.



```
0000000: 424d ae00 0000 0000 0000 3600 0000 2800  BM........6...(.
0000010: 0000 0600 0000 0600 0000 0100 1800 0000  ................
0000020: 0000 7800 0000 0000 0000 0000 0000 0000  ..x.............
0000030: 0000 0000 0000 0000 ff00 00ff ffff ffff  ................
0000040: ffff e7bf c8e7 bfc8 0000 0000 ff00 00ff  ................
0000050: ffff ffff ffff e7bf c8e7 bfc8 0000 00ff  ................
0000060: 0000 ff00 0000 0000 0000 ff00 80ff 0080  ................
0000070: 0000 00ff 0000 ff00 0000 0000 0000 ff00  ................
0000080: 80ff 0080 0000 0000 ff00 00ff ff80 00ff  ................
0000090: 8000 0080 ff00 80ff 0000 0000 ff00 00ff  ................
00000a0: ff80 00ff 8000 0080 ff00 80ff 0000       ..............
~
:%!xxd
```

Figure 2-5: example of how a 24-bit color 6x6 BMP image file's data look like [27]

2.3.2 Row size

As for Windows OS, the minimum scanning unit is 4 bytes. For increasing the speed of acquiring data, ROW size of BMP image should be able to be divided by 4 bytes. So, each row ends with 0 to 3 extra bytes, so that each row contains a multiple of four bytes.[28] The total number of bytes necessary to store one row of pixels can be calculated as:

$$\text{ROW size} = 4 * \left\lceil \frac{BitsPerPixel * ImageWidth}{32} \right\rceil \ (bytes)$$

That's why in Figure 2-5, there are two padding bytes in the end of every row.

2.4 Sorting Network

The task of "sorting network" in this design is to find out the median value amount 9 input data. Figure 2-6, Figure 2-7 and Figure 2-8 are implementation ideas for FPGA provided by [3].

Three of them are suitable for pipeline operation without any delay from accepting input data. Figure 2-9 is the diagram of basic node in Figure 2-6, Figure 2-7 and Figure 2-8. The function of basic node is very simple, it is an 8-bit comparator. A, B is two input data, if A > B, then the output higher <= A, lower <= B. If B > A, then higher <= B, lower <= A.

Figure 2-6 is a kind of classic method and require 41 basic nodes. Figure 2-7 is an optimized solution, 27 basic nodes are required. But if look closer to Figure 2-7, 4 basic nodes are actually not necessary, because this design just require the median value. So, Figure 2-7 is a very good design.

Figure 2-8 is another area saving design, which just require 19 basic nodes.

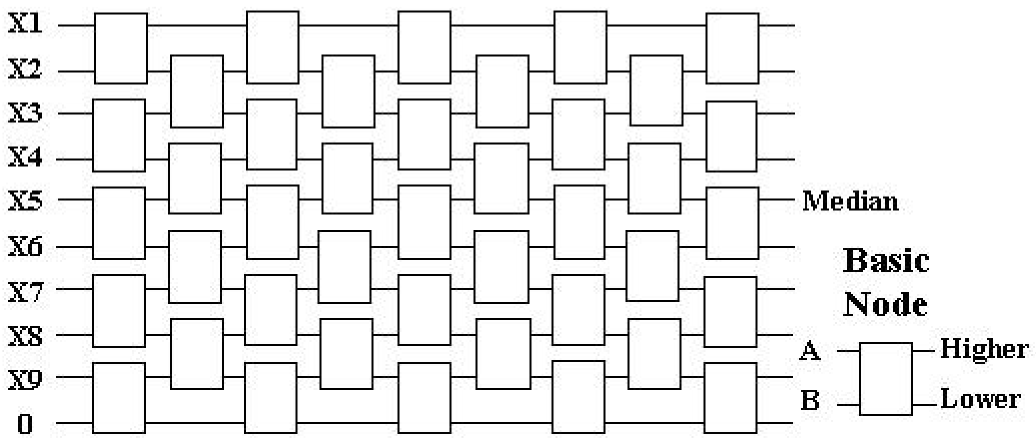


Figure 2-6: Classic sorting network for sorting 9 pixels [3]

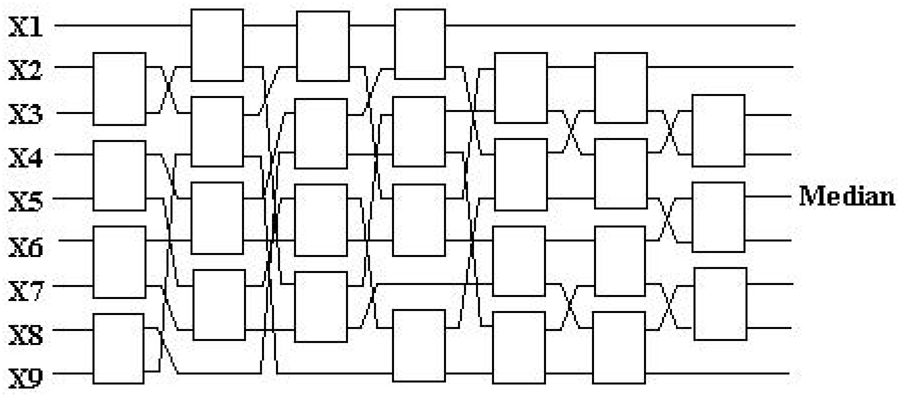


Figure 2-7: Optimized sorting network [3]

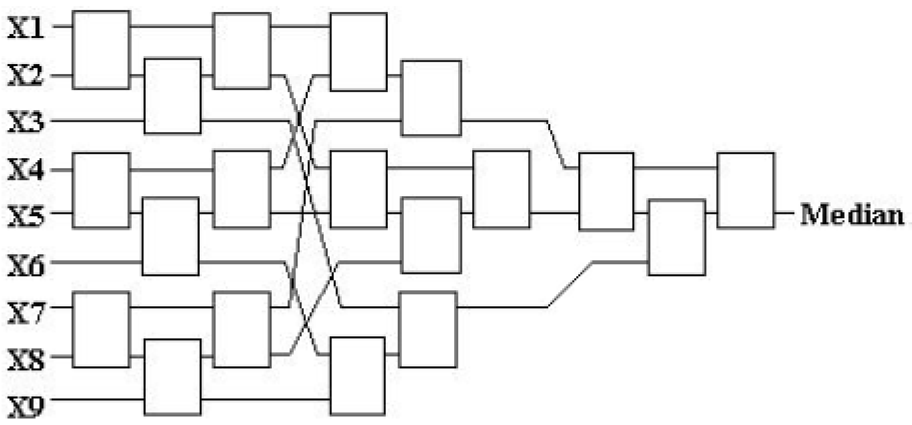


Figure 2-8: Sorting network circuit with 19 basic nodes [3]

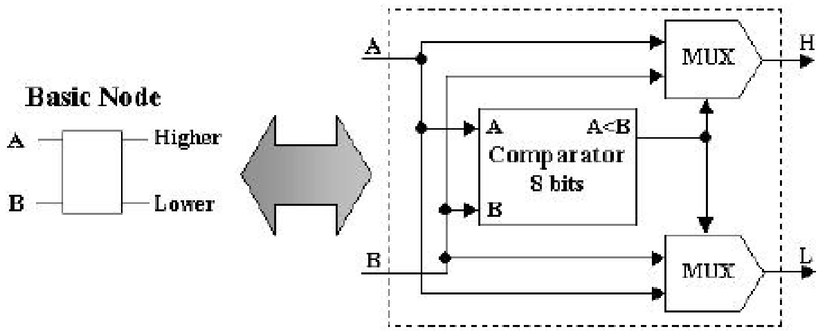


Figure 2-9: Scheme for each basic node [3]

2.5 Useful resource

2.5.1 Xilinx ZYNQ 7000 SOC

The Zynq®-7000 family is based on the Xilinx® SoC architecture. These products integrate a feature-rich dual or single-core ARM® Cortex™-A9 MPCore™ based processing system (PS) and Xilinx programmable logic (PL) in a single device. [8]

2.5.2 Linux Kernel and Linux file system

Linux Kernel and its file system are free and open source. Xilinx provides modified Linux Kernel which suitable for ZYNQ device, as well as its corresponding Ubuntu and Debian file system.

2.5.3 Necessary IP cores provided by Xilinx

Many useful IP cores are provided by Xilinx, such as AXI DMA IP, which provides DMA channels for the communication between PL side and DDR RAM. Those IP cores support at least 100Mhz clock frequency, it makes sure this system can run in high speed.

3 Hardware diagram design

3.1 Overview

For fulfilling the final task step by step, it is a good idea to replace the “Digital Image Filter” model in Figure 1-1 into “AXI-STREAM DATA FIFO” IP code, which is provided by Xilinx. The diagram will become Figure 3-1. In this case, PS side will transmit a stream data toward DMA channel. “AXI DMA” IP will receive those data correctly and provide them to “AXI-STREAM DATA FIFO” IP. FIFO will transfer same data back to “AXI DMA” IP. In the end of procedure, same data will return to PS side.

If PS side can receive data totally the same as the data it sent, it will mean DMA channel can run properly. Figure 3-2 is the brief diagram.

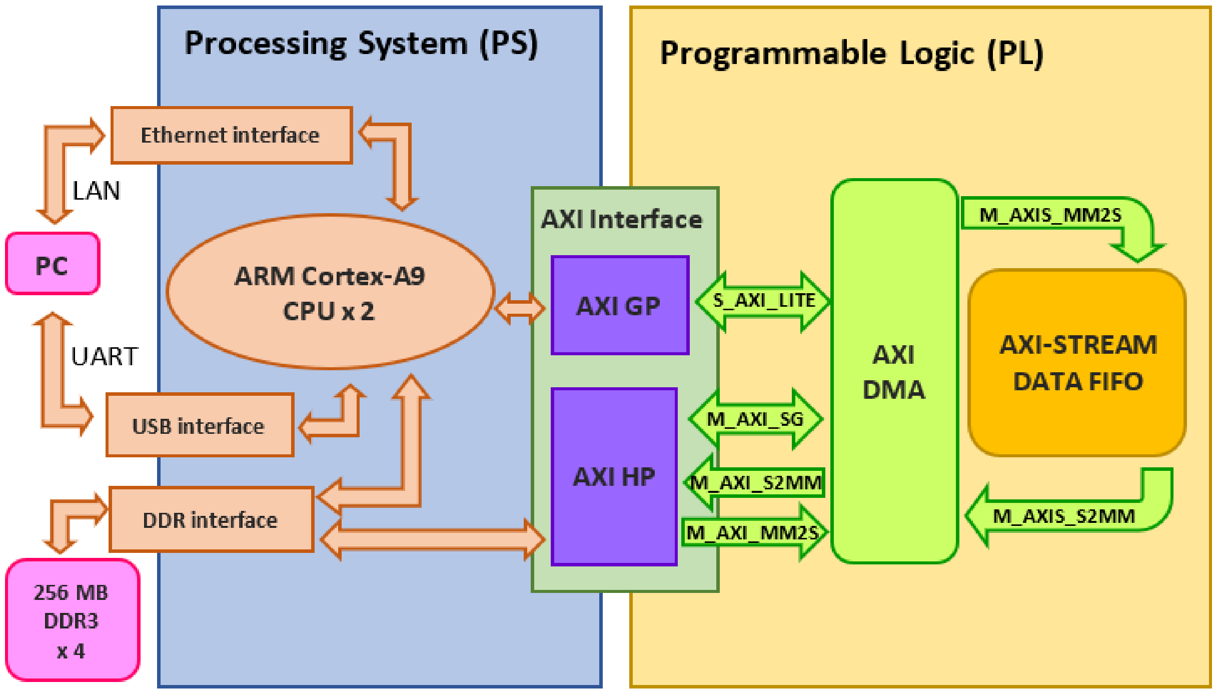


Figure 3-1: diagram of replacing the “Digital Image Filter” IP in Figure 1-1

3.2 Required IP cores

For fulfilling this system, first of all, need to create a diagram in Vivado. And add the first IP name “ZYNQ7 Processing System”. This IP core acts as a logic connection between PS side and PL side while assisting users to integrate custom and embedded IP cores with the processing system using the Vivado® Design Suite [17]. General speaking, “ZYNQ7 Processing System” can be considered as PS side. The setting of this IP can also be considered as the setting for PS side.

Click “Run Block Automation” to generate necessary port for “ZYNQ7 Processing System” automatically.

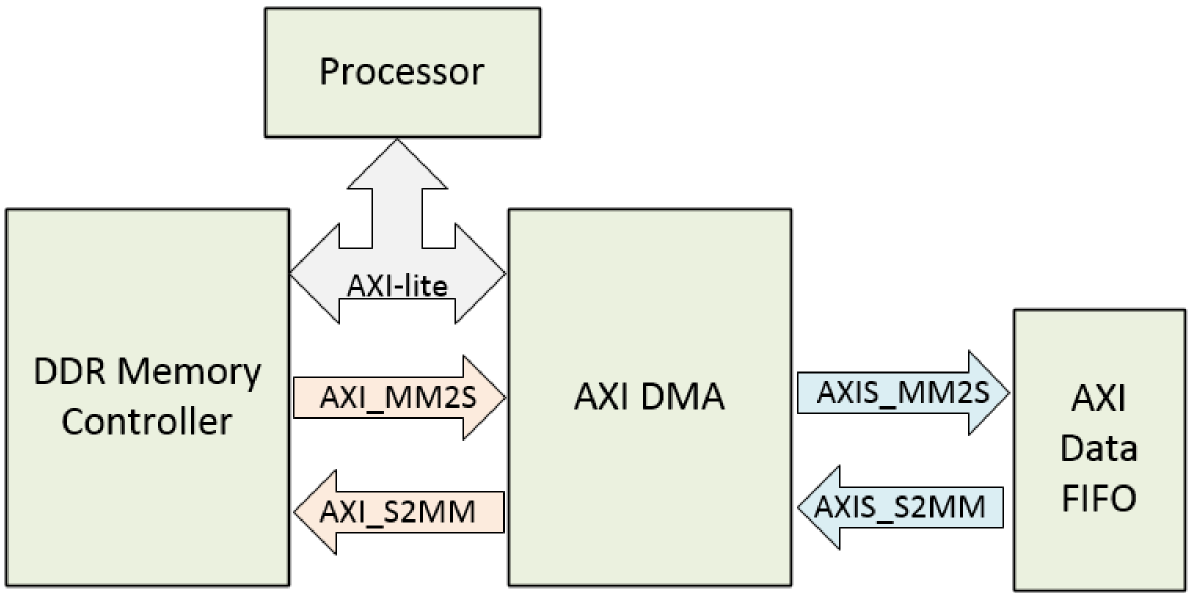


Figure 3-2: brief diagram of Figure 3-1 [14]

The second IP core is “AXI Direct Memory Access”, or “AXI DMA” IP. It provides high-bandwidth direct memory access between memory and AXI4-Stream target peripherals [18].

And the third IP is “AXI4-Stream Data FIFO”. It is a data FIFO with AXI4-stream interface [19].

The last one is “concat”, it is used for combining separated signal lines into one[20]. In this design, it combines two interrupts output lines (one line for transmission and another one for receiving) from “AXI DMA” IP.

3.3 IP cores configuration

There are several settings need to be done for those IPs. For “ZYNQ7 Processing System”, settings are listed as following:

1) In “PS-PL” Configuration, “S AXI HP0 interface” need to be selected.
2) In “Clock Configuration”, “requested frequency” of FCLK_CLK0 is 100MHz, it is the minimum clock frequency for “AXI DMA” IP.
3) Check “Fabric Interrupts” and “IRQ_F2P[15:0]” under “Interrupts”.

In “AXI Direct Memory Access”, settings are as following:

1) Make sure “Enable Scatter Gather Engine” is checked, it will reduce CPU intervention during data transmission. Useful for transmitting bigger size file.
2) Uncheck the “Enable Control/Status Stream”
3) Modify the value of “width of Buffer Length Register” into the maximum value, 26. To avoid mistake occur when transferring very big file.

Default configuration of “AXI4-Stream Data FIFO” and “concat” is OK.

3.4 Signal lines connection

In “ZYNQ7 Processing System” PL clock output port “FCLK_CLK0” should connect with “M_AXI_GP0_ACLK” and “S_AXI_HP0_ACLK”.

Then master port “M_AXIS_MM2S” in “AXI DMA” IP should connect with slave port of “AXI4_Stream Data FIFO” IP, named “S_AXIS”. While master port “M_AXIS” in “AXI4_Stream Data FIFO” connect to “S_AXIS_S2MM” in “AXI DMA” IP.

For “concat” IP, its “In0” connect to “mm2s_introut” in “AXI DMA” IP, “In1” connect to “s2mm_introut” in “AXI DMA” IP. “dout” connect to “IRQ_F2P” in “ZYNQ7 Processing System”.

Click “Run Connection Automation”, let vivado generate other necessary IP cores and do some necessary connection automatically. One more thing need to be careful is, “s_axis_aresetn” in “AXI4_Stream Data FIFO” IP will connect to the output pin “interconnect_aresetn” of “Processor System Reset” IP automatically, that’s not good. Right click “s_axis_aresetn”, select “disconnect pin” and connect “s_axis_aresetn” to “peripheral_aresetn” in “Processor System Reset” manually. Figure 3-3 shows how final connection looks like.

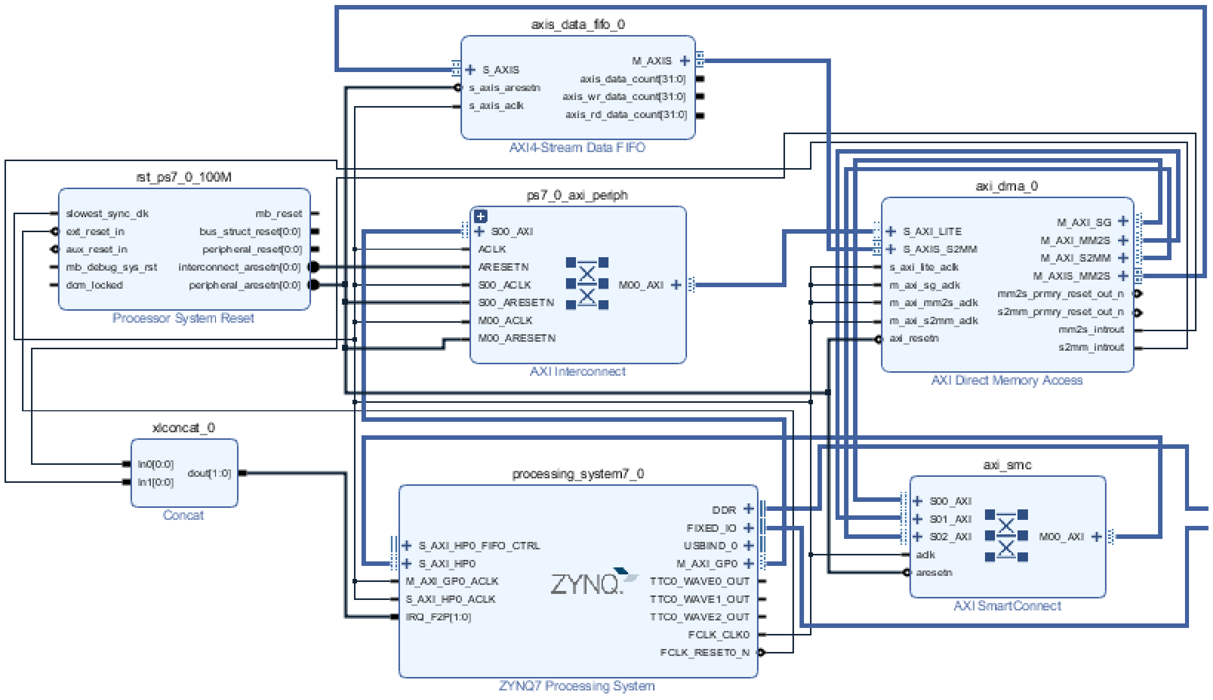


Figure 3-3: hardware diagram in Vivado

3.5 Other steps for finishing the diagram

1) Right click the diagram, and “validate design”.
2) “Generate output products” and “create HDL wrapper”.
3) Generate Bitstream

After all operation in this chapter, Vivado generated Bistream file, which contains the programming information for FPGA (PL side).

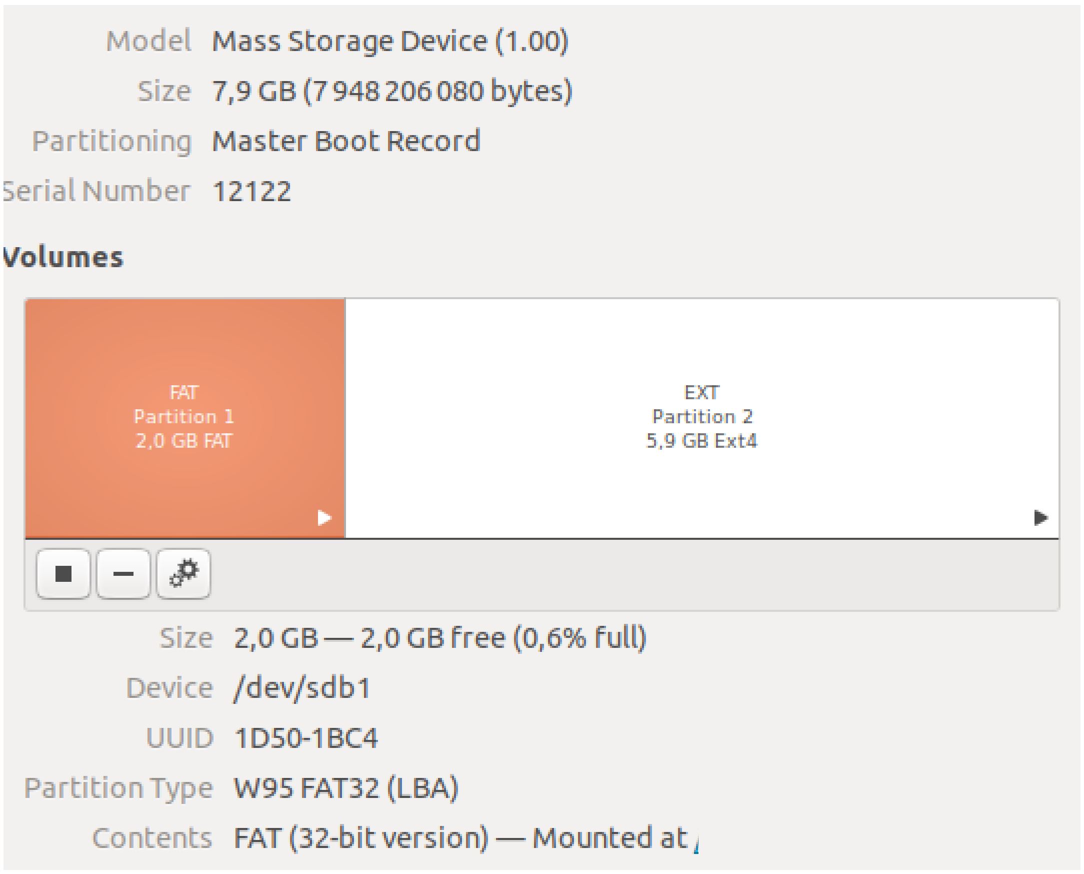
4 Linux OS environments and boot up files

4.1 Overview

The task of this chapter is to make necessary files for the Xilinx ZYNQ 7000 device, let the device be able to run Linux OS stable and use DMA channels to receive and transmit data.

Almost all operations in this chapter are done in PC with Linux, because of some required tools are only available in Linux. I did it in Ubuntu 18.04.1 LTS, which is run in VirtualBox.

First of all, one SD card with at least 4GB space is need. The SD card should be formatted similar with Figure 4-1.



Model	Mass Storage Device (1.00)
Size	7,9 GB (7 948 206 080 bytes)
Partitioning	Master Boot Record
Serial Number	12122

Volumes

Size	2,0 GB — 2,0 GB free (0,6% full)
Device	/dev/sdb1
UUID	1D50-1BC4
Partition Type	W95 FAT32 (LBA)
Contents	FAT (32-bit version) — Mounted at

Figure 4-1: format method for SD card

In Figure 4-1, the first volume is in FAT format, 2GB is much enough. The rest space can be distributed to the second volume, which is in EXT4 format.

FAT is used for storage three necessary files for system boot, they are:

- BOOT.BIN (boot image, contains First Stage Boot Loader (FSBL), bitstream and u-boot.elf)
- devicetree.dtb (device tree)
- uImage (Linux kernel)

EXT4 volume is for Linux file system.

The basic system boots up process is: first of all, bootROM (a ROM inside PS side)'s code will be run, and let the system get FSBL. With FSBL, system will initialize MIO, Clock, DDR, etc. And then, bitstream will be loaded and program for PL. u-boot will be loaded as well, it provides necessary codes to lead the system to get Linux Kernel and start initializing Linux operation system, device tree file will be executed as well.

4.2 BOOT.BIN file

This file contains FSBL, bitstream and u-boot. All of them are necessary for system start up.

4.2.1 u-boot

When the processor is powered on, the memory does not contain an operating system, so special software is needed to bring the OS into memory from the media on which it resides. This software is normally a small piece of code called the bootloader.[21]

u-boot is an open source bootloader that is frequently used in the Linux community, and used by Xilinx for the MicroBlaze™ processor and the Zynq-7000 AP SoC processor for Linux.[21]

Xilinx provides its u-boot source file for free in https://github.com/Xilinx/u-boot-xlnx

Just download the source files and make them, "u-boot" file will be generated.

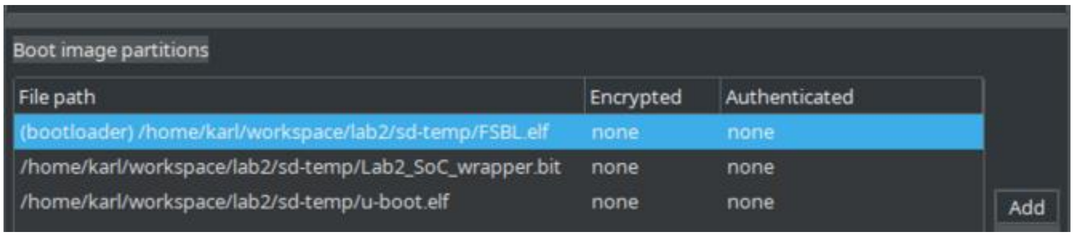
4.2.2 FSBL file generation

After all operations in Chapter 3, hardware's bitstream already generated. Export Hardware in "file" -> "Export", check "include bitstream". And then, launch SDK, SDK will load and create necessary files automatically.

In SDK window, create an application project by "Zynq FSBL" template, FSBL file will be generated by SDK.

4.2.3 Combine FSBL, bitstream and u-boot into BOOT.BIN

"Create Boot Image" tool is provided by SDK under "Xilinx" tab. Setup correct output paths. Then, choose all necessary files under "Boot image partitions". The order of files is: 1. FSBL.elf (Partition type: bootloader). 2. **.bit (bitstream, Partition type: datafile) 3. u-boot.elf (Partition type: datafile). [22] Finally, "Boot image partitions" will look like Figure 4-2. Click "Create Image", BOOT.BIN will be generated.



Boot image partitions

File path	Encrypted	Authenticated
(bootloader) /home/karl/workspace/lab2/sd-temp/FSBL.elf	none	none
/home/karl/workspace/lab2/sd-temp/Lab2_SoC_wrapper.bit	none	none
/home/karl/workspace/lab2/sd-temp/u-boot.elf	none	none

Figure 4-2:correct order and property of "Boot image partitions"[22]

4.3 Device tree

Device tree is a data structure describing the information about non-discoverable hardware components.[23] Those components may include Memory, peripherals, etc.

In this design, the main point of device tree is the structure of DMA channels.

The Linux device tree generator for the Xilinx SDK is provided by Xilinx and can be found in: https://github.com/Xilinx/device-tree-xlnx

Using this generator will save much time. Download the device tree generator and use the following way to add it into the repositories in SDK:

"Xilinx" -> "repositories" -> "New" in "Local Repositories" -> choose the position of device tree generator -> OK

Next, use "board support package" in "File"-> "New", and choose "device_tree", and finish. A folder named "device_tree_bsp_0" will be generated. It contains all necessary files for generating device tree. But some modification needs to be made for them.

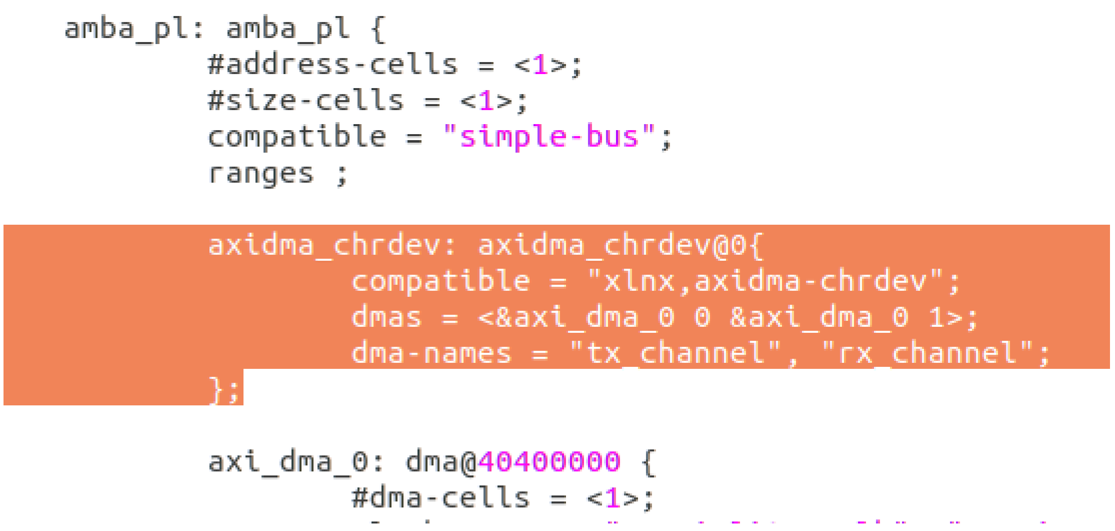
First of all, in “system-top.dts”, property of “bootargs” should be changed as:

bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=1 cma=25M earlycon"; [24]

above code will lead the Linux kernel boot file system from SD card

“CMA” stands for “Contiguous Memory Allocator”, memory space for DMA.[24][25]

And in file “pl.dtsi”, following high-lighted code in Figure 4-3 should be added, to define name, number and driver for AXI DMA channel:



```
amba_pl: amba_pl {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        ranges ;

        axidma_chrdev: axidma_chrdev@0{
                compatible = "xlnx,axidma-chrdev";
                dmas = <&axi_dma_0 0 &axi_dma_0 1>;
                dma-names = "tx_channel", "rx_channel";
        };

        axi_dma_0: dma@40400000 {
                #dma-cells = <1>;
```

Figure 4-3: necessary device tree code for file “pl.dtsi” [24]

And in this file, ID “xlnx,device-id” in both master and slave channel are the same, “0x0”. Should change one of them into “0x1”.

After all of those modification, “dtc” tool can be used in Linux command line to generate the device tree file.

4.4 Linux kernel

Linux kernel is the base of Linux OS. Its source files for Xilinx ZYNQ can be found in https://github.com/Xilinx/linux-xlnx or https://gitlab.pld.ttu.ee/Karl.Janson/xilinx_linux.git

During the kernel configuration, “Contiguous Memory Allocator” and “DMA Contiguous Memory Allocator” should be turned on. More details can refer to [25].

Make the kernel, we will get “uImage” file.

4.5 Linux-C code driver and execution result

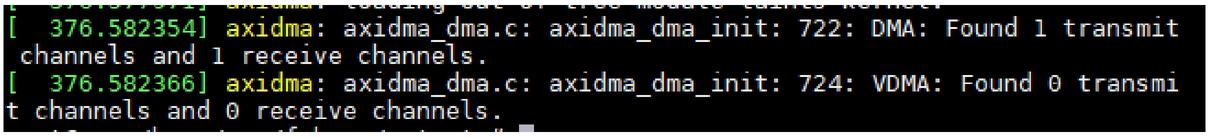
After above operations, three necessary files for the FAT volume are ready. For EXT4 volume, Linux file system can be found in here: https://rcn-ee.com/rootfs/eewiki/minfs/

If SD card prepared, and Linux system can run in the device properly, then “Samba” can be installed to the Linux system. “Samba” will be useful for exchanging file between the device and windows (PC).

Besides, using SSH to build up the connection between the device and PC is also a good idea.

[26] provides useful DMA channel drivers and examples. Download them, compile them, and copy the output files to the device.

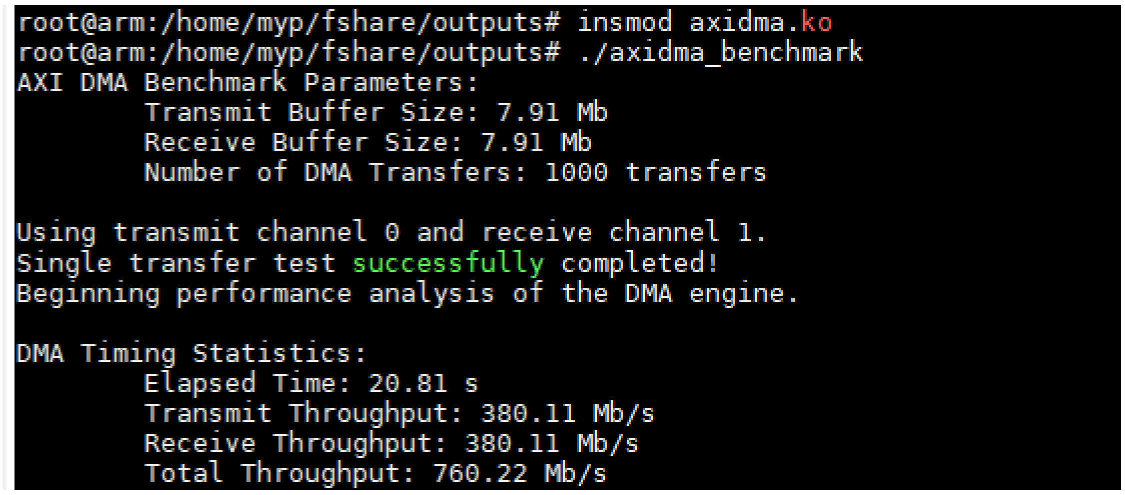
There are two very important output files. The first one is “axidma.ko”, which is the kernel driver of DMA channels. So, after running the command “insmod axidma.ko”, and use “dmsg”, will see the result like Figure 4-4. It means both DMA channels (one for transmission, another for receiving) are found by driver.



```
[  376.582354] axidma: axidma_dma.c: axidma_dma_init: 722: DMA: Found 1 transmit
 channels and 1 receive channels.
[  376.582366] axidma: axidma_dma.c: axidma_dma_init: 724: VDMA: Found 0 transmi
t channels and 0 receive channels.
```

Figure 4-4: after installing DMA kernel driver “axidma.ko”

Another important file is “axidma_benchmark”, it can help you to check the communication and speed of DMA channels. More information can be found in Figure 4-5.

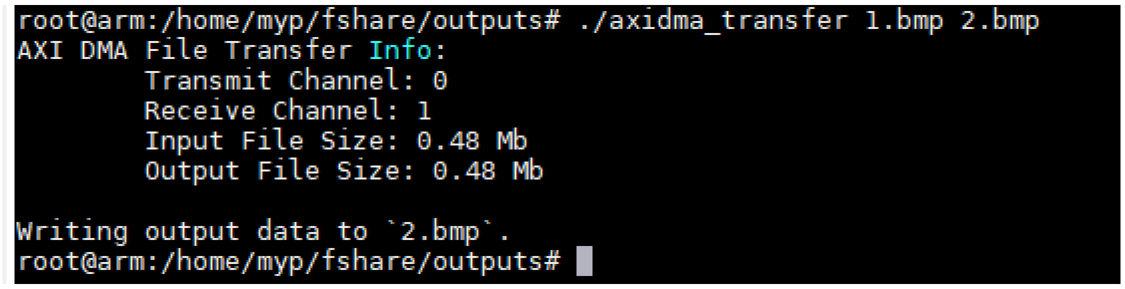


```
root@arm:/home/myp/fshare/outputs# insmod axidma.ko
root@arm:/home/myp/fshare/outputs# ./axidma_benchmark
AXI DMA Benchmark Parameters:
        Transmit Buffer Size: 7.91 Mb
        Receive Buffer Size: 7.91 Mb
        Number of DMA Transfers: 1000 transfers

Using transmit channel 0 and receive channel 1.
Single transfer test successfully completed!
Beginning performance analysis of the DMA engine.

DMA Timing Statistics:
        Elapsed Time: 20.81 s
        Transmit Throughput: 380.11 Mb/s
        Receive Throughput: 380.11 Mb/s
        Total Throughput: 760.22 Mb/s
```

Figure 4-5:running result of “axidma_benchmark”

In Figure 4-5, both transmit and receive packages size are 7.91 Mb, received data are total match the transmitted data. Transfer speed are both 380.11Mb/s. In this case, I can conclude that the DMA channels can work properly.

One more file named “axidma_transfer” is also very useful. Its function is to transfer a file to DMA channel, and receive data from DMA channel as well. It will write the received into a file. Figure 4-6 is an example. More usage information can be found by execute command “./axidma_transfer -help”.



```
root@arm:/home/myp/fshare/outputs# ./axidma_transfer 1.bmp 2.bmp
AXI DMA File Transfer Info:
        Transmit Channel: 0
        Receive Channel: 1
        Input File Size: 0.48 Mb
        Output File Size: 0.48 Mb

Writing output data to `2.bmp`.
root@arm:/home/myp/fshare/outputs# 
```

Figure 4-6: usage example of “axidma_transfer”

In current system, the write back file will be total the same as the transmitted file.

5 Image filter IP implementation

5.1 overview

After above operations, we can make sure that both Linux OS and DMA channels are working properly. Next step, the “AXI-STREAM DATA FIFO” in Figure 3-1 should be changed back the “Digital Image Filter” IP, like Figure 1-1.

Figure 5-1 shows the basic structure about the Digital Image Filter IP. As this IP core receives data from “AXI DMA” IP and needs to send data back to DMA channel through “AXI DMA” IP as well, Digital Image Filter IP must be compatible with AXI4-stream protocol. “AXI receiver” and “AXI sender” models are responsible for the compatibility.

When DMA channel’s data arrived, “AXI receiver” will setup relevant signals and start to receive DMA data. Necessary DMA data will be extracted by “AXI receiver” and deliver to DATA input BUFFER. “AXI sender” is for transmitting the result. More details about those two models can be found in section 5.2.

The task of “DATA input BUFFER” is to storage at least three lines image pixel data. And distribute those data to Sorting Networks. There are four same “Sorting Network” are used and all working in pipe line. As the DMA data width is 32-bit, which means “DATA input BUFFER” will receive 4 bytes data in each clock cycle during data transmission. But each image pixel has only 3 bytes (red, green and blue). To make sure the system can run in highest speed, one extra Sorting Network is used.

Figure 5-2 shows the basic working principle about “DATA input BUFFER”. If the extra Sorting network has been used, then the next operation will be like Figure 5-3. After the operation of the third line of input image, the fourth line data of image will go to the first line storage place in “DATA input BUFFER”, and so on. Figure 5-4 is an example. Please refer to section 5.3 for more details about “DATA input BUFFER”. And section 5.4 for more details about sorting network.

The output of Sorting Networks' data will go to an output FIFO. This FIFO will receive and assemble those 8-bit data, deliver them to "AXI sender" in 32-bit format. More details, please refer to section 5.5.

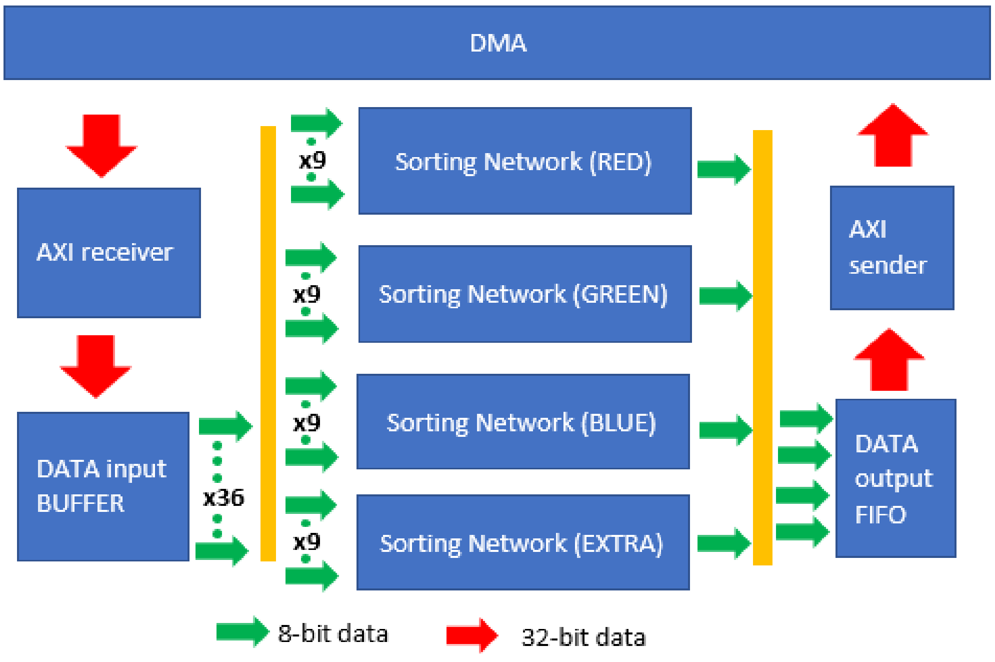


Figure 5-1 basic structure about the image filter IP core

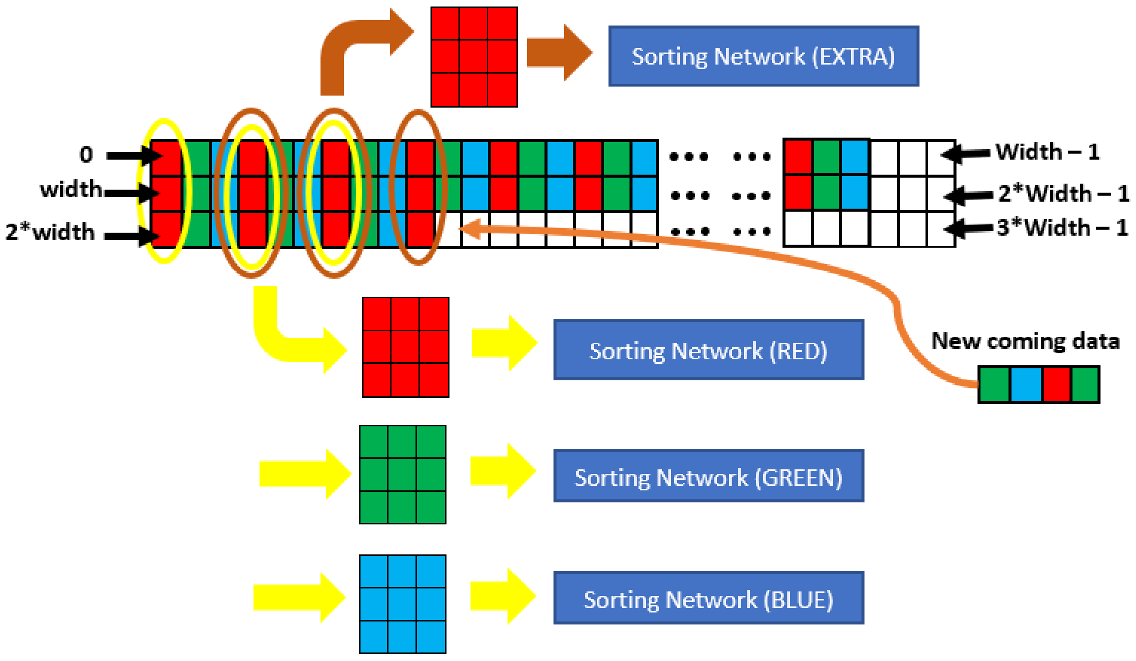


Figure 5-2: working principle of "DATA input BUFFER"

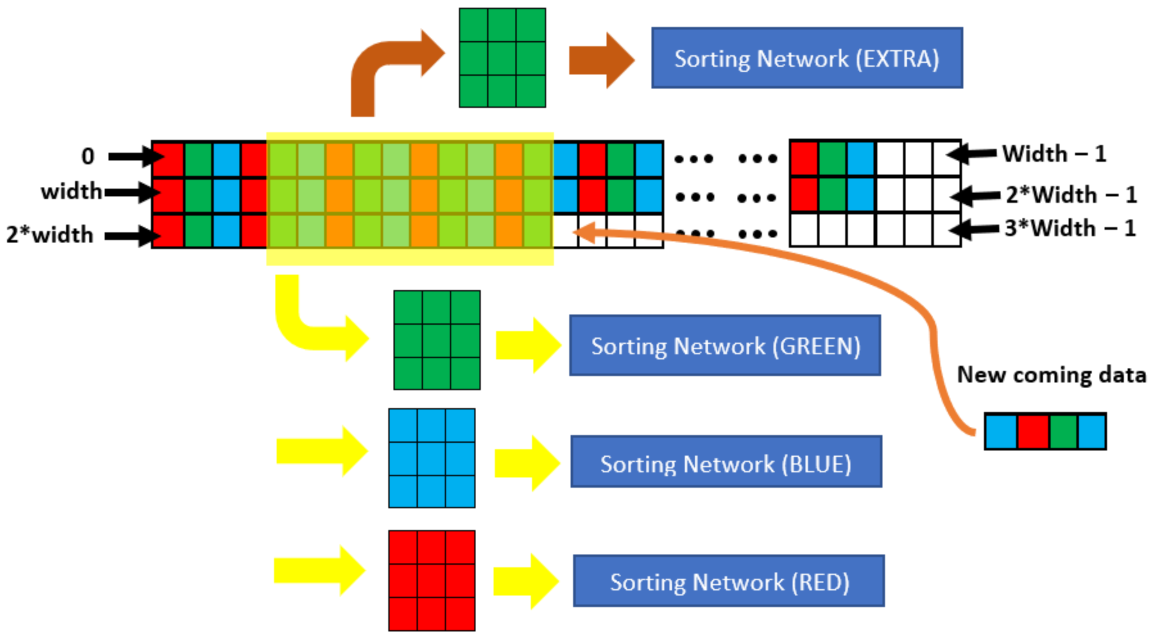


Figure 5-3: "DATA input BUFFER" next operation after Figure 5-2

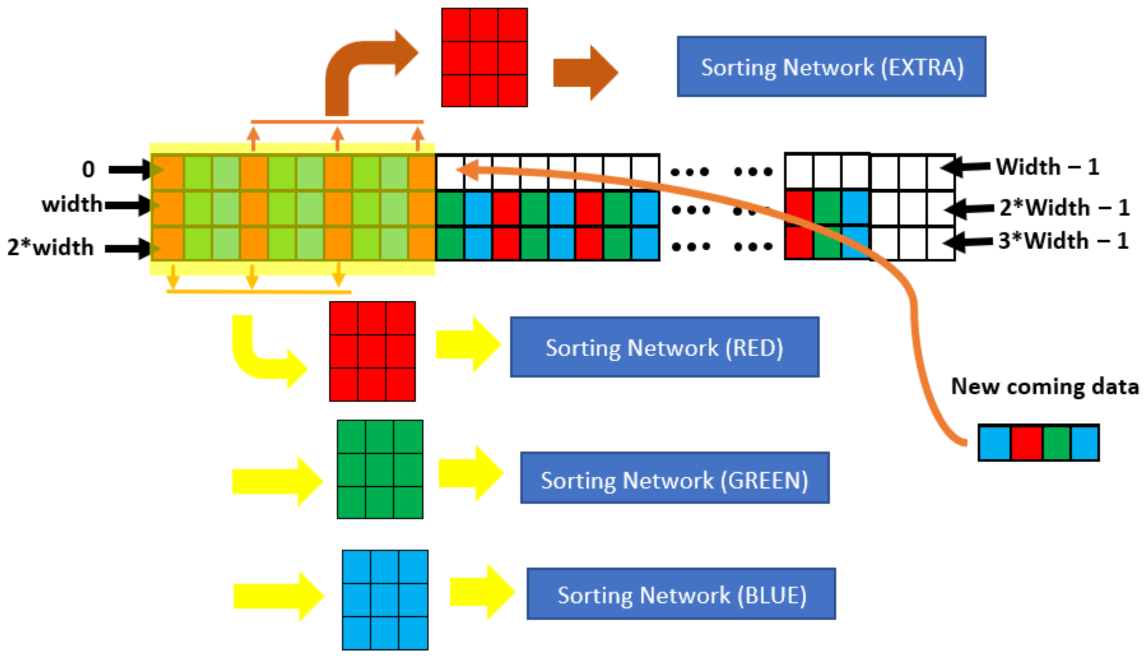


Figure 5-4: "DATA input BUFFER" after filling the third line, the fourth line pixel data of input image will go to BUFFER first line

5.2 AXI4 stream interface

For communicating with "AXD DMA" IP, "AXI receiver" and "AXI sender" models are required to make this IP compatible with AXI-stream protocol.

5.2.1 “AXI receiver” model

“AXI receiver” is acted as an AXI4-stream slave, receiving data from AXI4-stream master (AXI DMA IP master) and deliver useful data to DATA input BUFFER. Besides, it should control the “tready” signal line according to the status of “DATA output FIFO”. Figure 5-5 list the details of all input and output signal lines relevant with “AXI receiver” model.

Left part of interface details in Figure 5-5 corresponding to the “M_AXIS_MM2S” signal in Figure 1-1. An example of a whole data receiving process is shown in Figure 5-7. More details can refer to section 5.2.3.

Signal lines in the right part are explained as following:

1) “data_out” is the payload transfer to “DATA input BUFFER” in Figure 5-1.
2) “keep_out” is similar with “tkeep”, which will be explained in section 5.2.3, but it is associated with “data_out”.
3) “trans_start” indicates that the data stream transmission is started. Once it is set to ‘1’, it will keep its value until “tlast” is high.
4) “trans_en” indicates that “data_out” and “keep_out” signals in this clock cycle can be used.
5) “buf_full” indicates the output FIFO in Figure 5-1 “DATA output FIFO” model is full or not. This signal will be used for deciding the output value of “tready”.

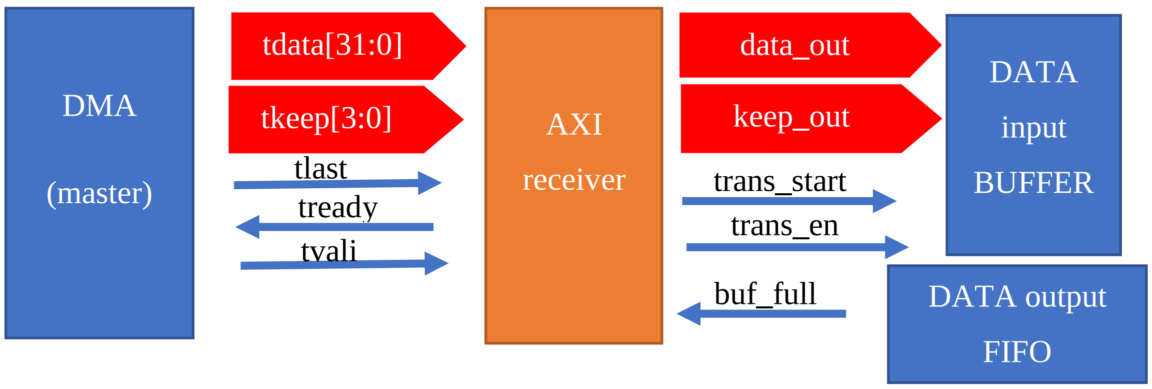


Figure 5-5: “AXI receiver” interface

5.2.2 “AXI sender” model

“AXI sender” is acted as an AXI4-stream master. It is responsible for transmitting data to AXI4-stream slave (AXI DMA IP slave) according to the data provided by “DATA output

FIFO". During the data transmission, it is necessary to consider the status of "AXI DMA" IP master and slave.

Right part of Figure 5-6 is the interface details between "AXI sender" and "AXI DMA IP" slave, corresponding to "M_AXIS_S2MM" in Figure 1-1.

The communication protocol between this model with AXI DMA slave is explained in section 5.2.3.

Signal lines in the left part are explained as following:

1) "data_in" is the payload that ready to transfer back to DMA channel.
2) "data_keep" is similar with "tkeep" signal which is mentioned in section 5.2.3, but it associates with "data_in".
3) "buf_pre_empty" indicates the output FIFO has only one data left.
4) "buf_empty" indicates the output FIFO is empty.
5) "read_en" means read enable. If it set as high, indicating "AXI sender" is reading output FIFO's data, output FIFO should go to the next data in every clock cycle.
6) "get_done_out" is a useful signal line for "DATA Output FIFO" model. When it is set as high, means DMA data receiving has finished. But it doesn't mean DMA data transmission finish.
7) "fifo_rd_en" indicates whether the output FIFO can be read or not.

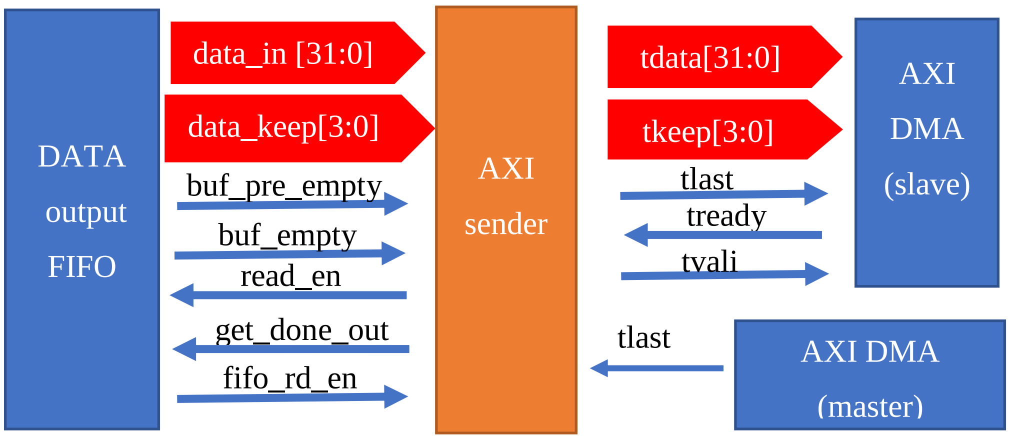


Figure 5-6: "AXI sender" interface

Signal "fifo_rd_en" is different with "buf_empty". "fifo_rd_en" may be set to '0' even when the output FIFO is not empty. The purpose is to let "tkeep" signal's value is always "0xF", let every byte in “tdata” can be used, except for the last clock cycle before transmission finish. So, “fifo_rd_en” will be set as ‘1’ when output FIFO has 4 bytes or more than 4 bytes, or when transmission is in the last clock cycle.

5.2.3 AXI4 stream receiving process

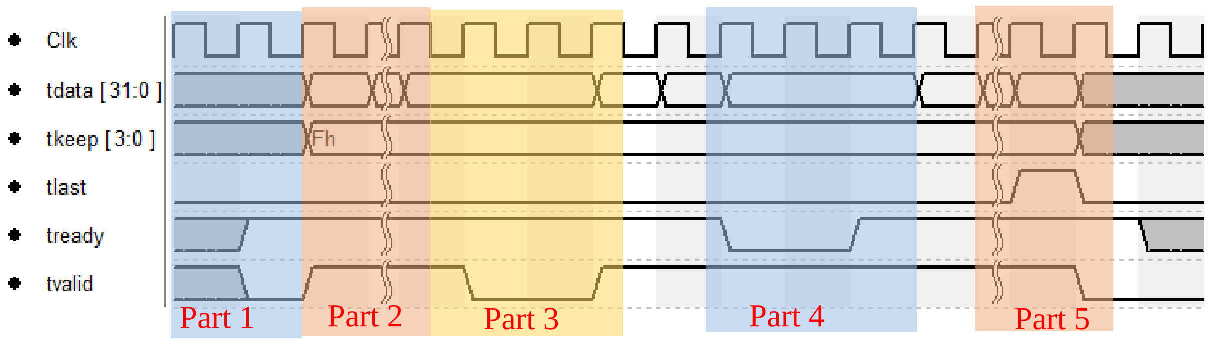


Figure 5-7: An example of a whole AXI4 stream receiving process

Figure 5-7 is an example of a whole AXI4 stream receiving process. Those signal lines are corresponding to the left part of Figure 5-5. This process can be divided into 5 parts.

Part 1 in Figure 5-7 is initialization stage. In this stage, signal “tready” set as high first, indicates that the slave device (AXI receiver) is ready to receive data.

When DMA channel start to transmit, signal “tvalid” will be asserted high, let the process step into Part 2. And signals “tdata” and “tkeep” will start to change immediately. Signal “tvalid” set as high indicates the transfer is valid, or both “tdata” and “tkeep” are valid.

“tdata” is the RGB image pixels data (payload) from DMA channel.

“tkeep” signal width is depend on the width of “tdata”. More specific,

$$width\ of\ \text{“}tkeep\text{”}\ = \frac{width\ of\ \text{"}tdata\text{"}}{8}$$

Width of “tdata” must able to be divided by 8, and each bit of “tkeep” associate with 1 byte in “tdata”. If a bit in “tkeep” is high, means the “tdata” byte which associate with the bit is a valid data byte, should be kept. On the contrary, if the “tkeep” bit is low, its “tdata” byte should be abandoned. In most of time, bits in “tkeep” are all set as ‘1’, which means “tdata” should be totally kept.

There is another possible situation of Part 1 and Part 2 can be found in Figure 5-8, even this situation has very low possible to happen. In this case, “tvalid” is asserted high earlier than “tready”. “tdata” and “tkeep” change together with “tvalid” assertion. But because “tready” still low, transmission has not started yet, “tdata” and “tkeep” keep their value until both “tready” and “tvalid” are high.

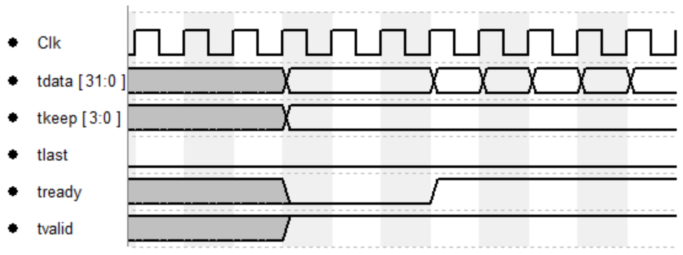


Figure 5-8 initial stage of AXI4 stream when “tvalid” set as high earlier than signal “tready”

Part 3 in Figure 5-7 is the situation that “tvalid” become low during transmission. “tdata” and “tkeep” will not change since the falling edge of “tvalid”. They will recover the data transmission in the rising edge of “tvalid”, if “tready” value is still ‘1’.

And the situation in Part 4 (signal “tready” set as low during transmission) is slightly different with Part 3. “tdata” and “tkeep” stop changing in the next clock cycle after “tready” set low. And recovering transmission time also has one clock cycle gap after “tready” return to high.

The difference between Part 3 and Part 4 is very easy to understand. Because signal “tdata”, “tkeep” and “tvalid” are from master side, so they all be controlled together. Slave side will collect their value in the next clock cycle. But “tready” is from slave, master device needs one clock cycle time to make reaction.

“tlast” is used for indicating the last data of a data packet. Part 5 in Figure 5-7 is an example. In the same clock cycle when “tlast” set high, “tdata” and “tkeep” provide the last DMA data. Value ‘1’ of “tlast” will last for only one clock cycle. After this cycle, “tvalid” will be asserted to low, DMA transmission finish.

One more thing need to pay attention in the last transmission cycle is, the value of “tkeep” may change in this clock cycle with high possibility. As most of time during transmission, every bit of “tkeep” is ‘1’. But DMA channel may doesn’t have so many data need to be transferred in the last clock cycle, some bits in “tkeep” may be ‘0’.

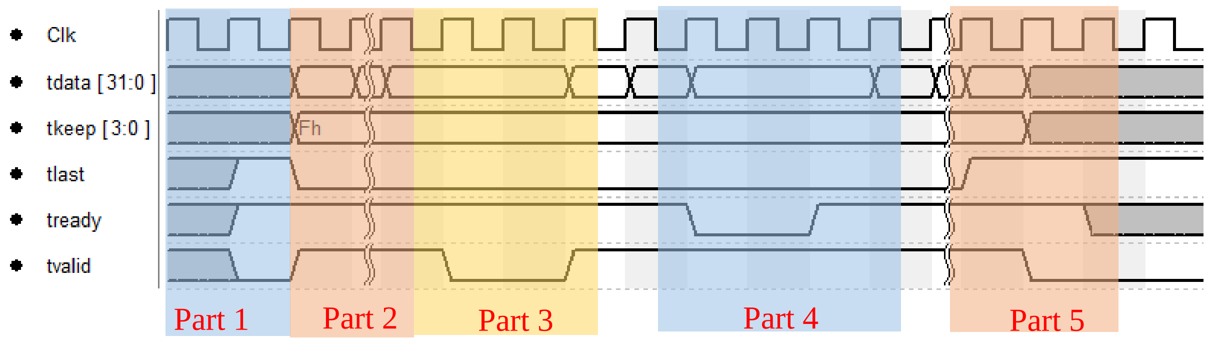


Figure 5-9: An example of a whole AXI4 stream transmission process

Figure 5-9 is an example of a whole AXI4 stream transmission process. It is almost similar with Figure 5-7, except for signal "tlast" in part 5 and part 1. In this process, "tlast" is start from high level. When then transmission begin, it changes to low level. At the last transmission clock cycle, "tlast" not just keep one clock cycle high level, but change to high, and keep the value until the next transmission start.

5.2.4 "AXI receiver" Simulation result

Figure 5-10 is the simulation result when DMA data just arrive. "trans_start" and "trans_en" change to '1' immediately when "tvalid" become high. Figure 5-12 is the end of DMA data receiving. "trans_start" and "trans_en" change when "tlast" set to '1'. During transmission, "tvalid" from master may change to '0' for a short time. In this case, DMA transmission pause, but not finish, so "trans_en" will be assert to '0', but "trans_start" keeps its value. Figure 5-11 illustrate this situation.

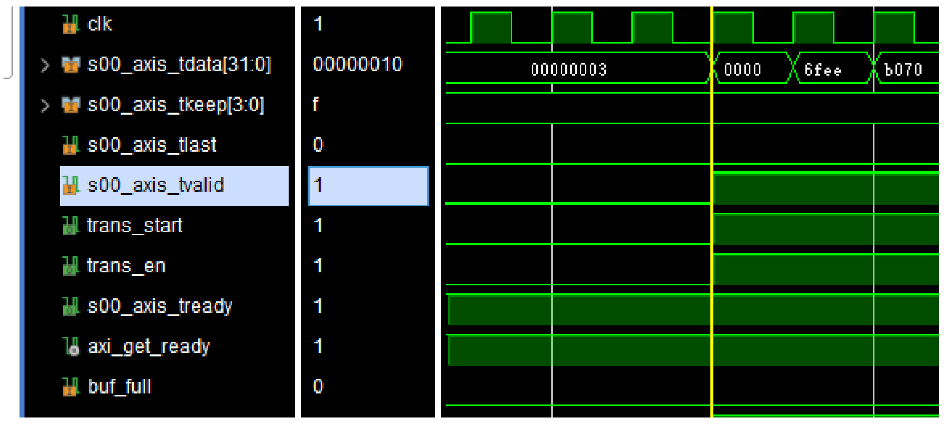


Figure 5-10: Simulation result of "AXI receiver" in startup stage

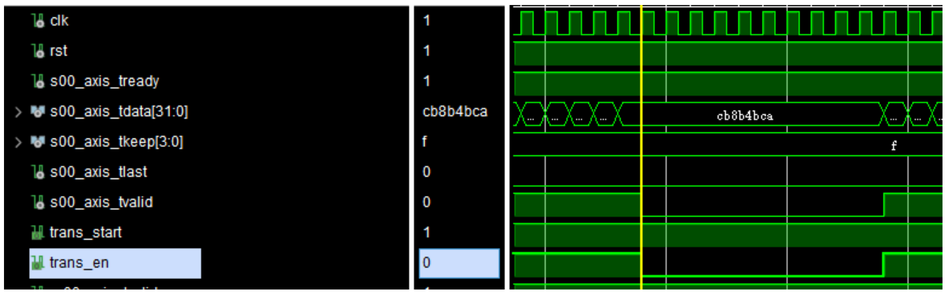


Figure 5-11: Simulation result of "AXI receiver" when "tvalid" fall during transmission

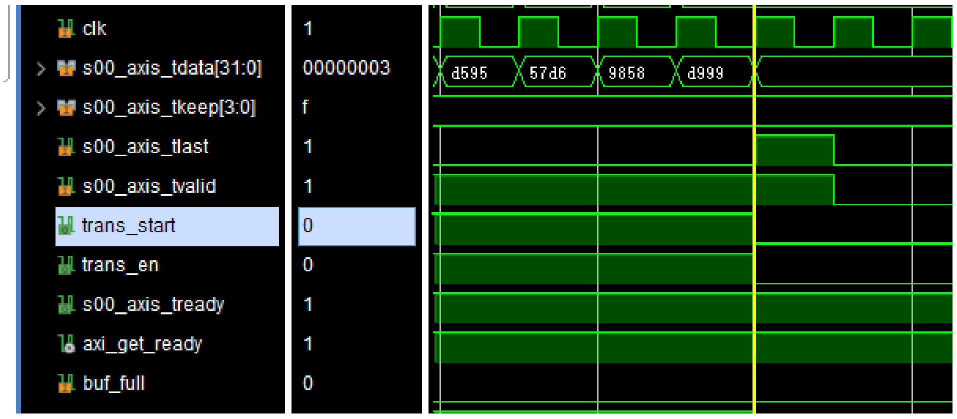


Figure 5-12: Simulation result of "AXI receiver" when receiving is going to finish

5.2.5 "AXI sender" Simulation result

Figure 5-13, Figure 5-14 and Figure 5-15 are the simulation waveform of "AXI sender" model. Figure 5-13 describes how "AXI sender" start to transfer data. First of all, "read_en" set to '1'. As the output FIFO need one clock cycle to react. So, in the next clock cycle, all signals relevant with AXI4 interface are act the same as Figure 5-9 part 2.

The blue part in Figure 5-14 corresponding to part 3 in Figure 5-9, and the red part is Part 5. "tkeep" value change only in the last clock cycle.

Figure 5-9 part 4 is shown in Figure 5-15.

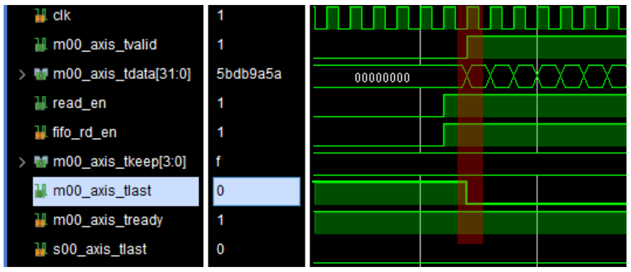


Figure 5-13: Simulation result of "AXI sender" in startup stage

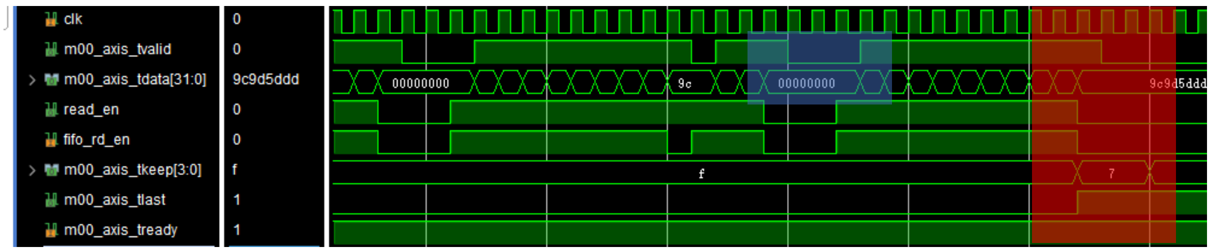


Figure 5-14: Simulation result of "AXI sender" in the end of transmission

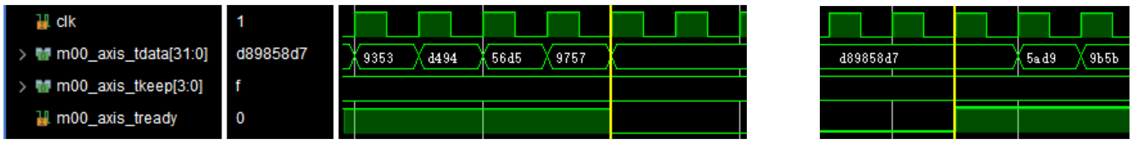


Figure 5-15: Simulation result of "AXI sender" when "tready" set to low in midway

5.3 DATA input BUFFER implementation

5.3.1 Implementation analyzing

Figure 5-2, Figure 5-3 and Figure 5-4 show the function that this model needs to fulfill. As memory resource in PL side is limited, "DATA input BUFFER" only storage three lines image pixels data. The requirements of this BUFFER are:

1) It equips RAM that with enough space to storage at least three lines of image data. In this system, the maximum length of the image is 1920 pixels. Each pixel is combined by Red, Green and Blue, three color. Each color occupies one-byte space. So, the minimum required space is: (1920 pixels) x (3 color) x (3 lines) = 17280 bytes.
2) It has ability to receive 4-byte data every clock cycle. In this system, the DMA channel data width is 4 bytes. When the transmission stream begin, DMA channel will start to send data, 4 bytes in each clock cycle. For reaching the highest data proceeding speed, this model must be able to handle 4-bytes in each clock cycle, and last for long time.
3) It has ability to provide correct R, G, B data to four sorting networks.
4) Every clock cycle, model should be able to provide 36 bytes from RAM.
5) RAM can read and write in the same clock.
6) It has ability to handle "tkeep" signal, only keep useful data in RAM.

For the requirement 1) above, as the size of RAM is not small, so using registers to assemble such size RAM is impossible, even this method can make a very flexible RAM.

It is lucky that PL side has enough “block RAM” resource, which helpful for establishing big size RAM system. But there is one serious problem is, none of those block RAM resources have multi output port. In this case, requirements 4) and 5) above will become a very difficult problem. What the system need is a RAM with 4-byte input and 36-byte output, or four 1-byte input and 9-byte output RAMs. But block RAMs are all with two R/W ports maximum. Even the width of port can up to 4096 bits, read port’s width cannot bigger than write port’s [6][7].

“block RAM” resource is significant for this model. It has to be used. To solve problems described in above paragraph, this model should be separated into four parts. The internal structure of this “DATA input BUFFER” model will look like Figure 5-16.

In Figure 5-16, block RAM array contains 6 block RAMs. And two memory controllers are used, one is for block RAM array’s input, another is for output. Those memory controllers are responsible for writing data in correct position of block RAM array and reading correct data from block RAM array. They will change those block RAMs into a 4-byte input and 36-byte output memory device. More details, please refer to section 5.3.2, section 5.3.3 and section 5.3.4.

5.3.2 Block RAM array

Looking closer to Figure 5-2, Figure 5-3 and Figure 5-4, even though the BUFFER need to provide 36 bytes to Sorting Networks in every clock cycle, only maximum 12 bytes are new. The rest 24 bytes are the same as last clock cycle. This conclusion is very important, as in this case, block RAM array just need 12-bytes output.

Figure 5-17 is the first solution about block RAM array. In Figure 5-17, RAM 1 is for storage the first-ROW data in Figure 5-2. Then, RAM 2 is the second ROW, and RAM 3 is the third ROW. Each clock cycle, this block RAM array can provide four new bytes from each ROW. Their address for writing and data for writing are connected together, writing enable bus (WR_EN) are separated. Because in each clock cycle, only four bytes need to write.

Figure 5-17 solution is able to solve requirement 1), 4) and 5) in section 5.3.1. But it cannot handle the situation that when “tkeep” not equal to “1111” (or number of BYTEs for writing are not 4). It violates the requirement 6) in section 5.3.1. More specific, please refer to Figure 5-18.

So, Figure 5-17 need to be improved to handle writing different quantity of BYTEs. Figure 5-19 is the final solution. It uses column A and column B RAMs to solve above problems. Figure 5-20 describes the solution.

This Block RAM array model is done by six “block memory generator” IPs, which is provides by Xilinx.

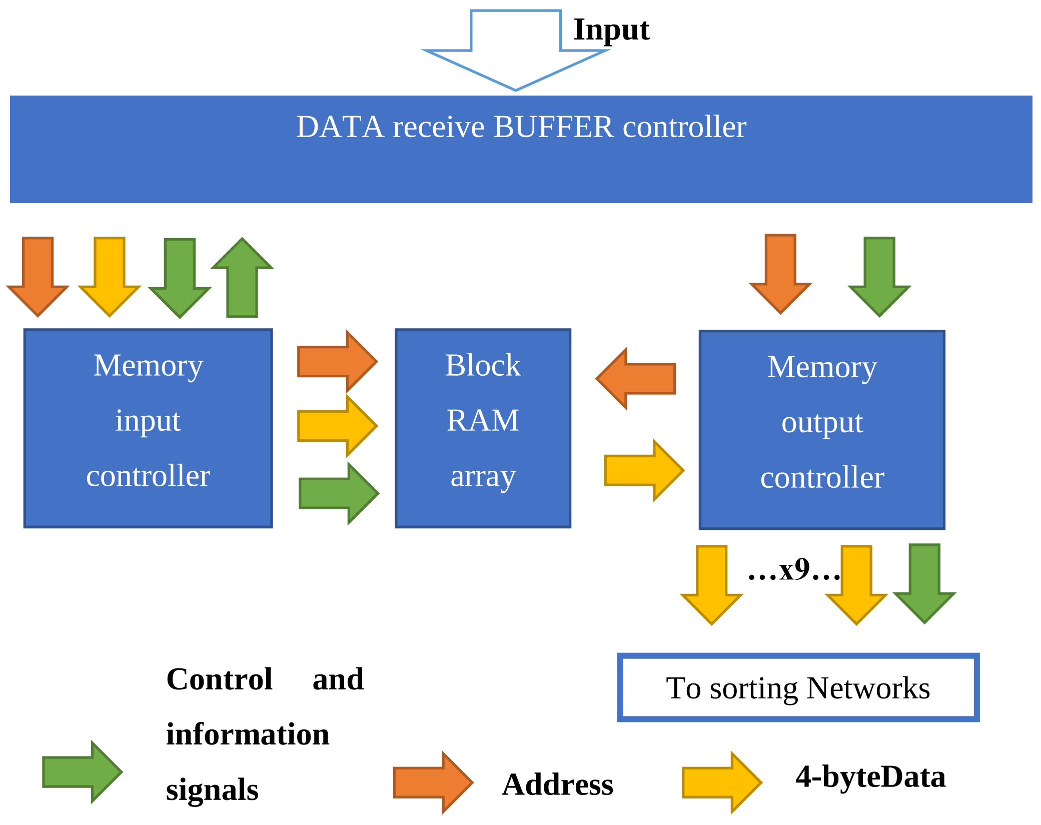


Figure 5-16: internal structure of “DATA input BUFFER”

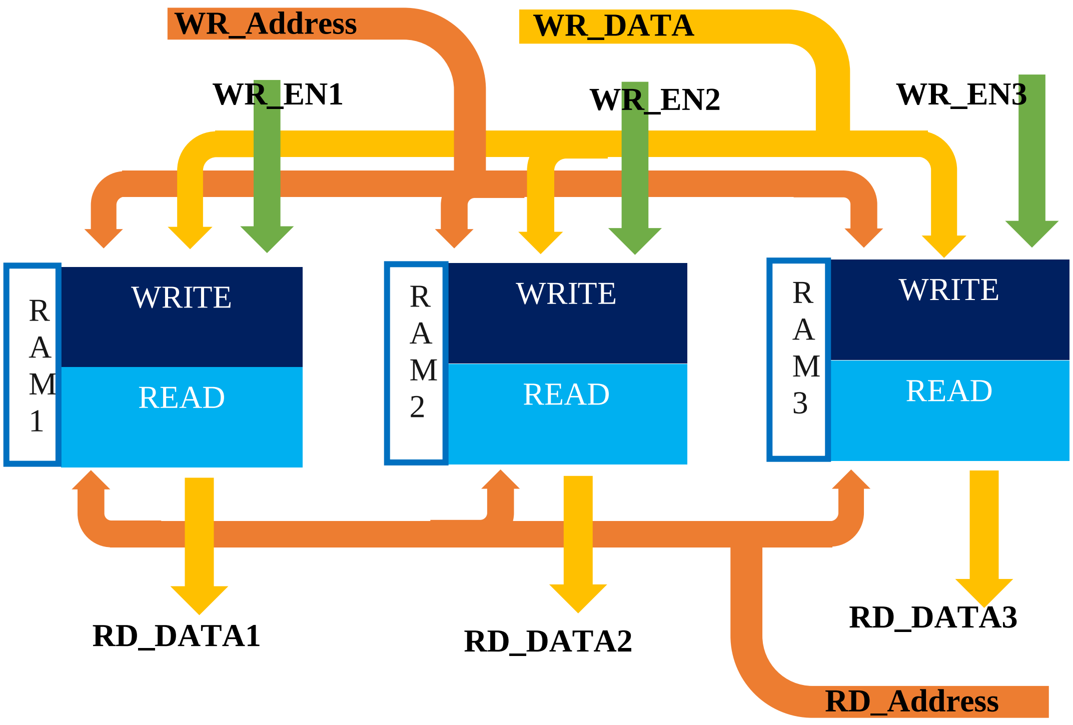
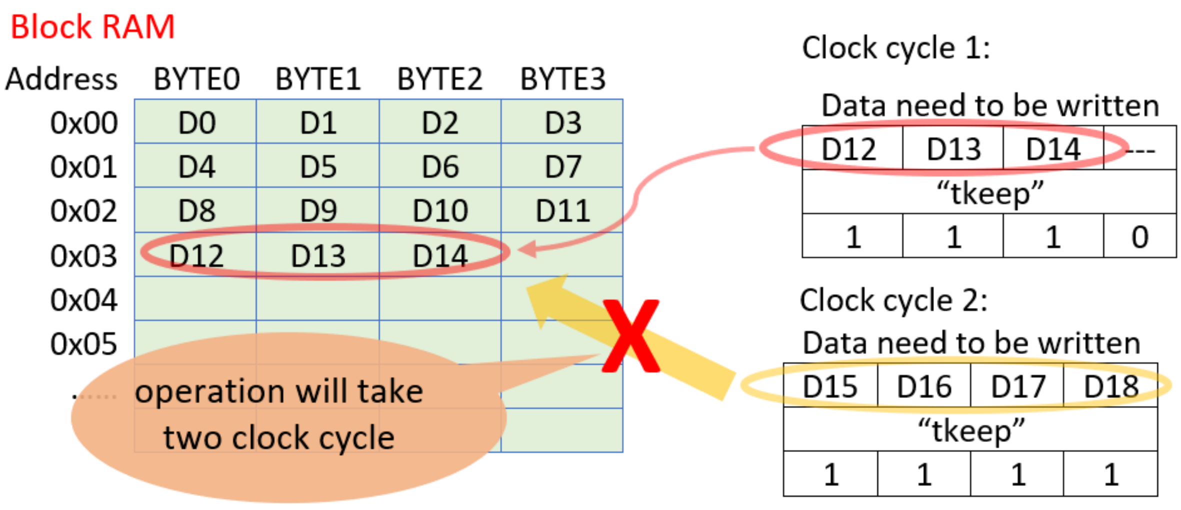


Figure 5-17: Solution 1 of the Block RAM array



Address	BYTE0	BYTE1	BYTE2	BYTE3
0x00	D0	D1	D2	D3
0x01	D4	D5	D6	D7
0x02	D8	D9	D10	D11
0x03	D12	D13	D14	
0x04				
0x05				
……				

D12	D13	D14	---
"tkeep"			
1	1	1	0

D15	D16	D17	D18
"tkeep"			
1	1	1	1

Figure 5-18: problem of Figure 5-17

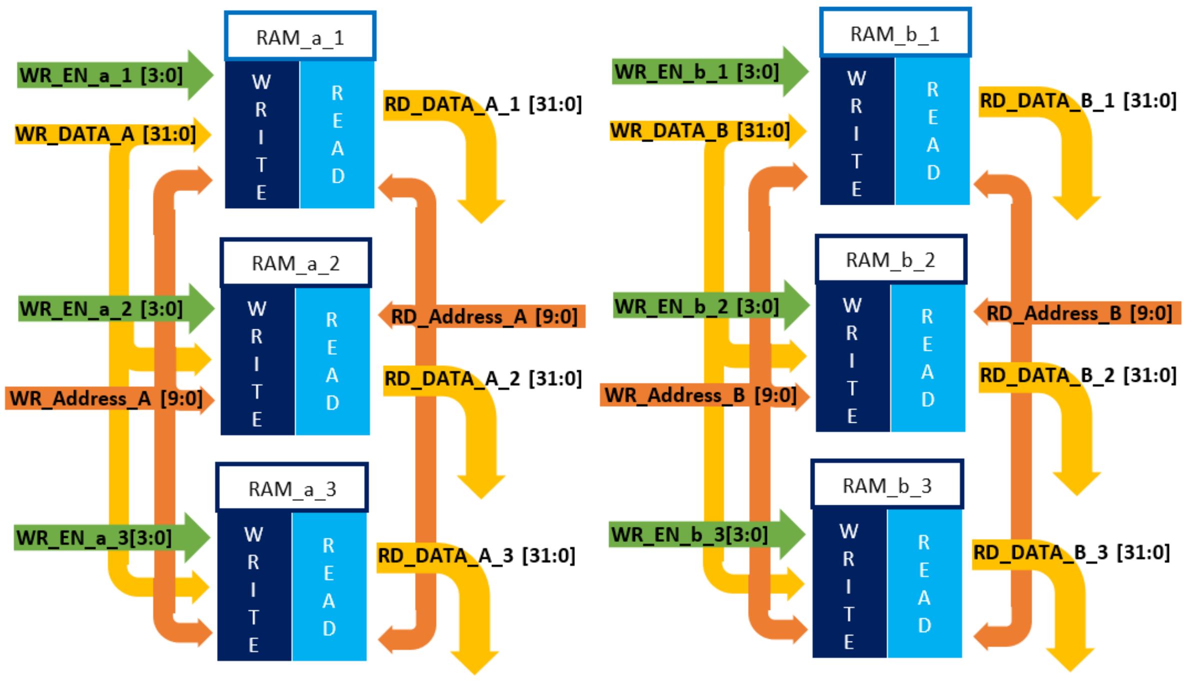
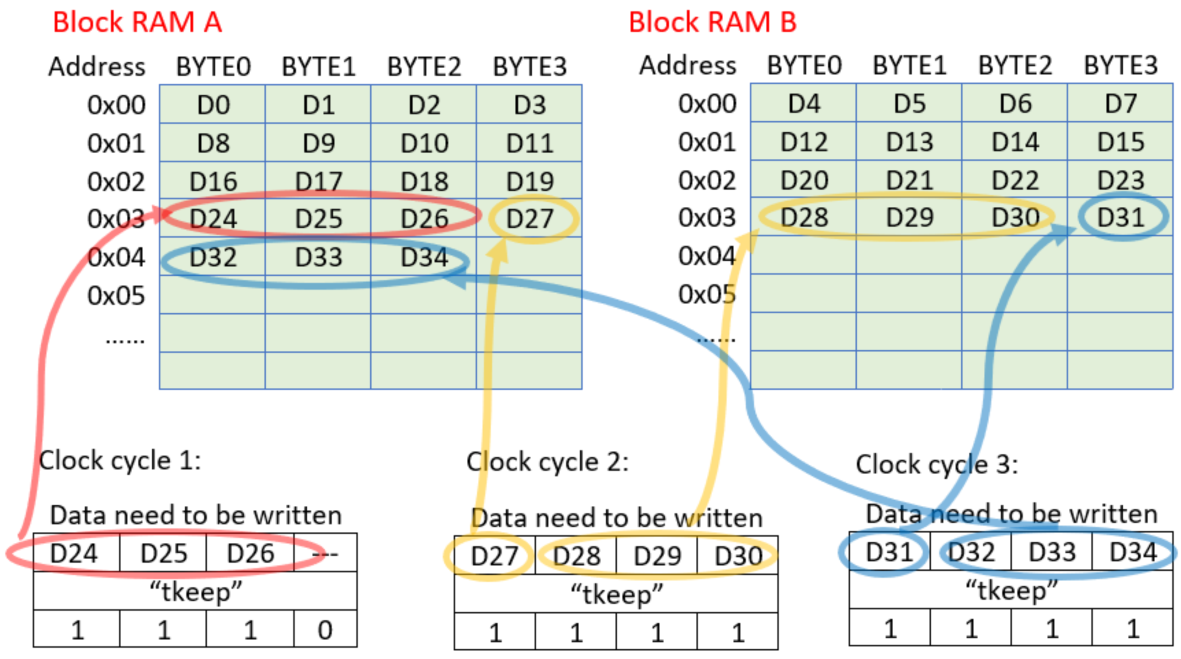


Figure 5-19: final solution of the Block RAM array



Block RAM A

Address	BYTE0	BYTE1	BYTE2	BYTE3
0x00	D0	D1	D2	D3
0x01	D8	D9	D10	D11
0x02	D16	D17	D18	D19
0x03	D24	D25	D26	D27
0x04	D32	D33	D34	
0x05				
......				

Block RAM B

Address	BYTE0	BYTE1	BYTE2	BYTE3
0x00	D4	D5	D6	D7
0x01	D12	D13	D14	D15
0x02	D20	D21	D22	D23
0x03	D28	D29	D30	D31
0x04				
0x05				
......				

Clock cycle 1:

D24	D25	D26	---
"tkeep"			
1	1	1	0

Clock cycle 2:

D27	D28	D29	D30
"tkeep"			
1	1	1	1

Clock cycle 3:

D31	D32	D33	D34
"tkeep"			
1	1	1	1

Figure 5-20: how Figure 5-19 solution solve requirement 6) in section 5.3.1

5.3.3 Memory input controller

The position of "Memory input controller" model in Figure 5-16 shows this model is responsible for writing data to Block RAM array. "DATA receive BUFFER controller" provides BYTEs and address of each BYTE to "Memory input controller", "Memory input controller" need to figure out the position of "Block RAM array" that those BYTEs should be written in. The different between the "address" value provided by "DATA receive BUFFER controller" and their real position in "Block RAM array" is shown in Figure 5-21.

Figure 5-22 list all input and output signal lines of "Memory input controller" model.

Input signal "width" indicates how many pixels in each line of the image. After "Memory input controller" model get this value, it will work out three integer value: "one_width_out", "two_width_out" and "three_width_out". "one_width_out" equal to (width * 3), as each image pixel is combined by three bytes. "two_width_out" equal to (one_width_out * 2), used for working out the ending position of the second line in Figure 5-2. "three_width_out" equal to (one_width_out * 3), used for working out the ending position of the third line in Figure 5-2.

Those three integer values are significant for both "Memory input controller" model and "DATA receive BUFFER controller" model.

If in this clock cycle, there are some data need to be written to "Block RAM array", "wr_en" will be set as '1'. "data_in" is the BYTEs for writing. "keep_in" indicates which BYTE should be written, possible values are "0000", "0001", "0011", "0111", "1111". buf_add_1, buf_add_2, buf_add_3, buf_add_4 are the address value corresponding to each BYTE.

After "Memory input controller" acquire enough data from "DATA receive BUFFER controller", it will start to set its output signals to control "Block RAM array". The following signals will be relevant with "Block RAM array" controlling:

1) din_a and din_b are the data input signals of "Block RAM array". Corresponding to WR_DATA_A and WR_DATA_B in Figure 5-19. Providing the BYTEs need to be written to RAM.
2) wr_add_a and wr_add_b are the data writing addresses of "Block RAM array". Corresponding to WR_Address_A and WR_Address_A in Figure 5-19. Indicating the writing position.
3) wr_en_a_1, wr_en_a_2, wr_en_a_3, wr_en_b_1, etc. are writing enable signal to "Block RAM array", Corresponding to WR_EN_a_1, WR_EN_a_2, etc. signals in Figure 5-19. Indicating which two RAMs are choice to write the data, as well as how many BYTEs should be written in the BYTE position.

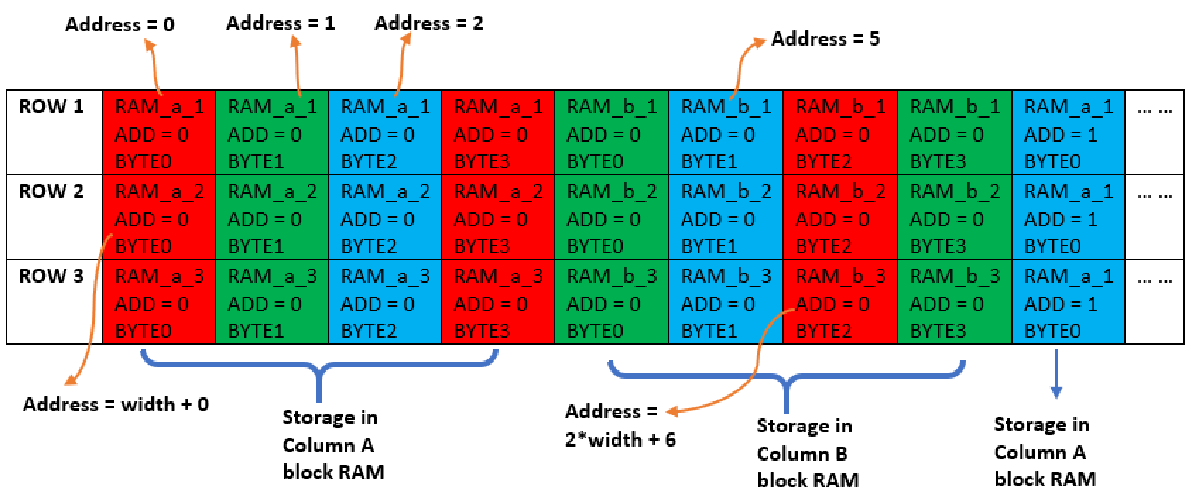


Figure 5-21: relationship between the "Address" provided by "DATA receive BUFFER controller" and their real position in "Block RAM array"

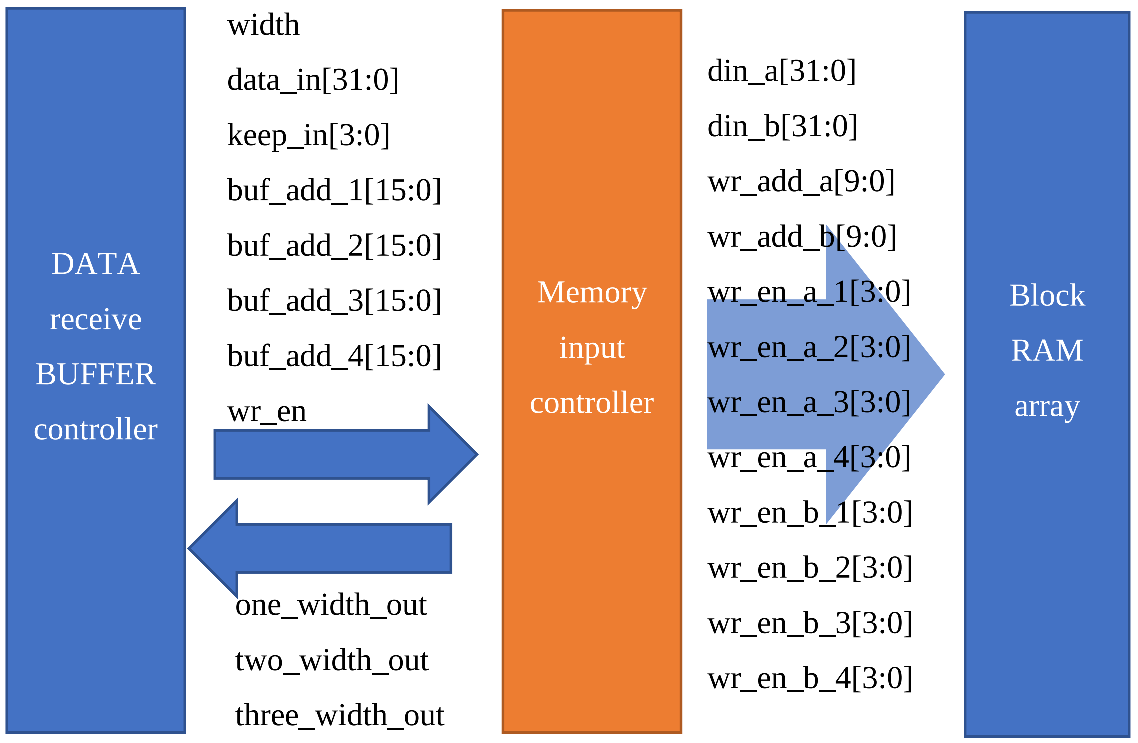


Figure 5-22: interface signals of "Memory input controller"

All operations in this model have to be done in Pipeline, as after transmission begin, data are coming in every clock cycle.

There are total 3 pipeline operations in this model.

The first operation solves which block RAM in column B should be used according to the value of "buf_add_1". And get values of buf_add_obs1, buf_add_obs2, buf_add_obs3 and buf_add_obs4. More details can be found out in Figure 5-23 and Figure 5-24.

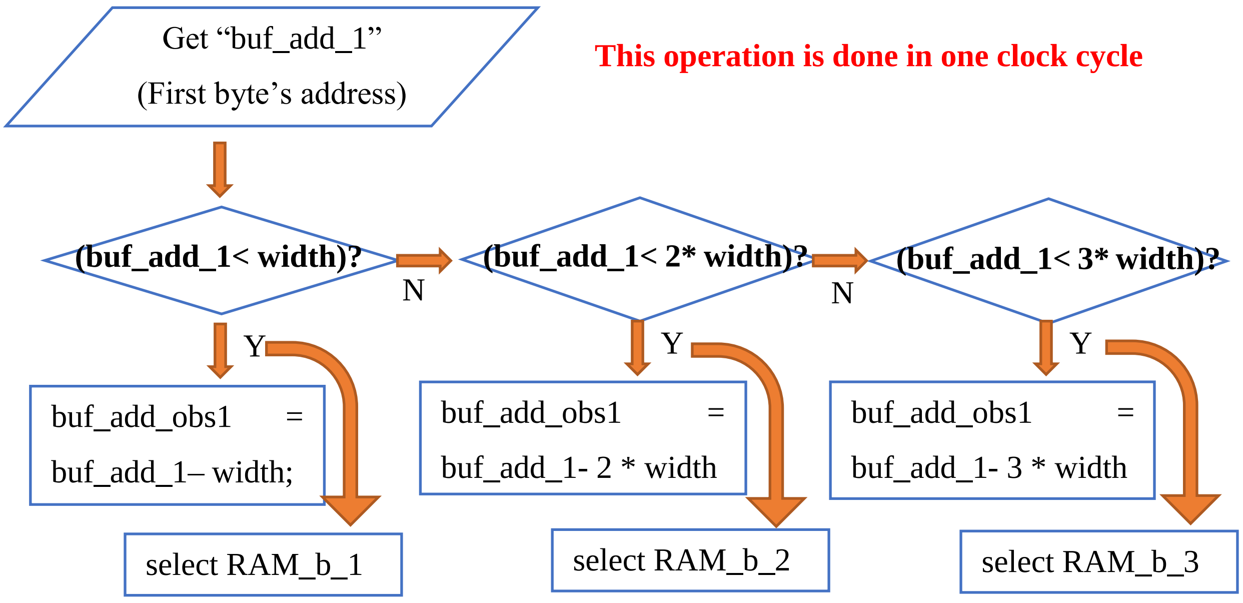


Figure 5-23: first stage pipeline operation, work out the value of "buf_add_obs1", and decide which RAM in B column should be used

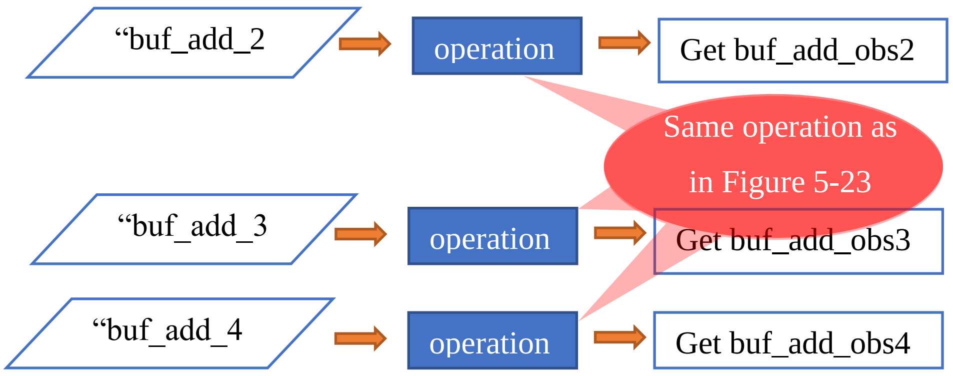


Figure 5-24: first stage pipeline operation, work out the values of "buf_add_obs2", "buf_add_obs3", "buf_add_obs4"

The second pipeline operation works out which RAM in A column should be used. This task cannot be done in the first pipeline operation, because a very special situation, which is shown in Figure 5-25, may happen. In Figure 5-25, the new coming data should be storage in the last column B RAM and the first column A RAM, then the ROW of column A RAM is the next ROW of column B RAM. Except for this situation, ROW index of column A RAM is equal to column B RAM's.

This situation is very rare. Because as mention in section 2.3.2, the bytes number of each line can be divided by 4. And most of time, DMA channel's "tkeep" value is "1111". But because the decision result is not hurrying to use, and in the second pipeline stage, there is another important operation need to be done. So, this operation is valuable to add.

There is one more special situation is, if the last byte of each line is storage in column A RAM, and Figure 5-25 situation happened. This model cannot handle this situation, it is a BUG. But as mentioned above, this situation is very rare, and use software to limit (or change) the size of input image can avoid this situation perfectly. Solving this BUG can be the future work of this system. One idea is, forcing the last 4 bytes in each line has to storage in column B block RAM.

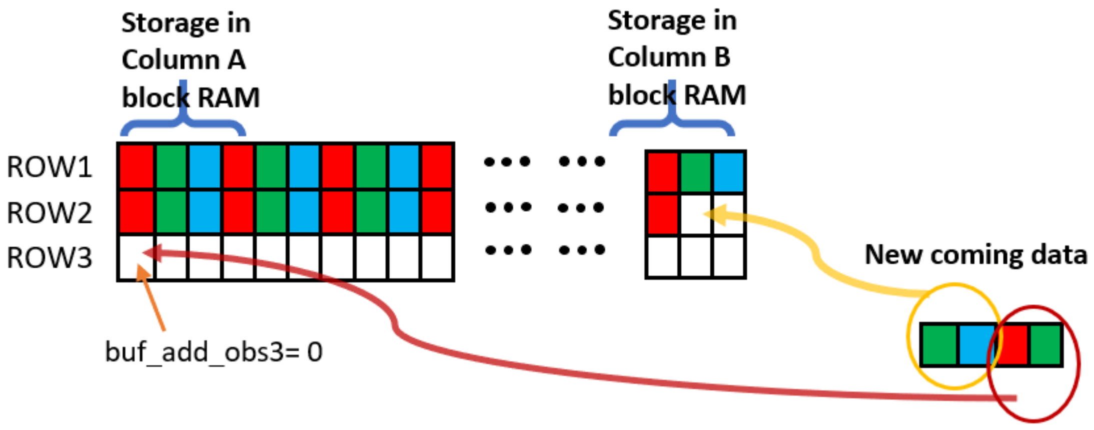


Figure 5-25: a very special situation when deciding which RAM in A column should be used

Another very important operation in the second pipeline operation stage is to decide WR_DATA_A and WR_DATA_B. and prepare enough information to decide the value of WR_EN_a_1, WR_EN_a_2, WR_EN_a_3, WR_EN_b_1, WR_EN_b_2, WR_EN_b_3 (those values will be finally decided in the third stage pipeline operation). Figure 5-26 and Figure 5-27 are two examples about the pipeline operations in the second and third stages.

As each block RAM address corresponding to 4 bytes data, while each "buf_add_obs" signal corresponding to 1 byte. So, the last two bits of "buf_add_obs1" can be used to decide the WR_EN signals and which byte start from.

According to Figure 5-20 and Figure 5-21, BIT2 of "buf_add_obs1" indicates which column of RAM this byte belongs to.

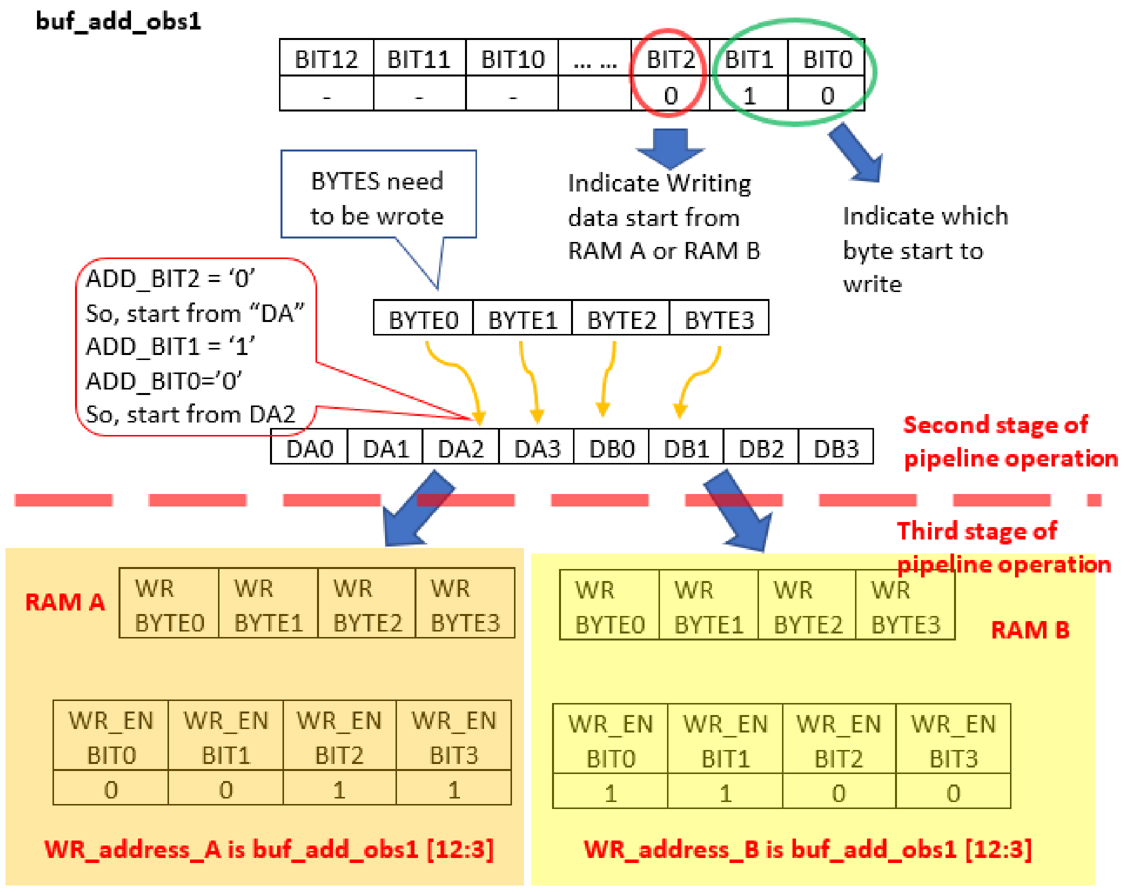

buf_add_obs1
BIT12 BIT11 BIT10 ... ... BIT2 BIT1 BIT0
- - - 0 1 0
BYTES need to be wrote
Indicate Writing data start from RAM A or RAM B
Indicate which byte start to write
ADD_BIT2 = '0' So, start from "DA" ADD_BIT1 = '1' ADD_BIT0='0' So, start from DA2
BYTE0 BYTE1 BYTE2 BYTE3
DA0 DA1 DA2 DA3 DB0 DB1 DB2 DB3
Second stage of pipeline operation
Third stage of pipeline operation
RAM A
WR BYTE0 WR BYTE1 WR BYTE2 WR BYTE3
WR_EN BIT0 WR_EN BIT1 WR_EN BIT2 WR_EN BIT3
0 0 1 1
WR_address_A is buf_add_obs1 [12:3]
RAM B
WR BYTE0 WR BYTE1 WR BYTE2 WR BYTE3
WR_EN BIT0 WR_EN BIT1 WR_EN BIT2 WR_EN BIT3
1 1 0 0
WR_address_B is buf_add_obs1 [12:3]

Figure 5-26: example of how to write bytes to "Block RAM array" (1), pipeline operation in stage 2 and stage 3

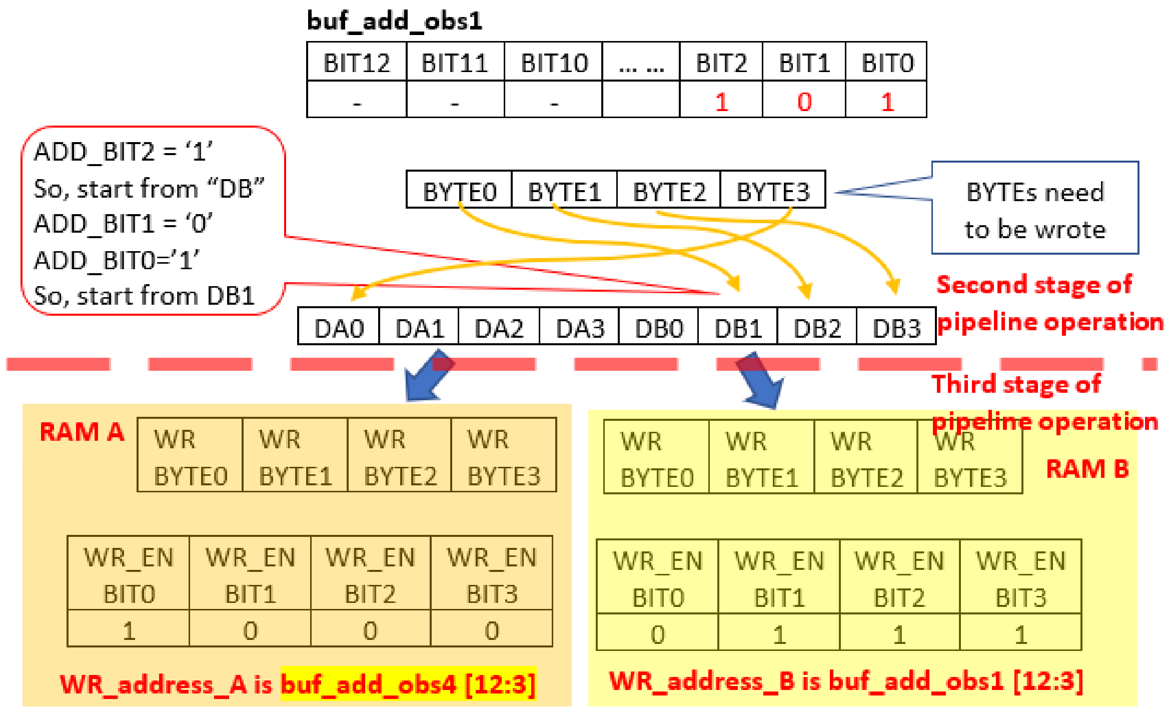

buf_add_obs1
BIT12 BIT11 BIT10 ... ... BIT2 BIT1 BIT0
- - - 1 0 1
ADD_BIT2 = '1' So, start from "DB" ADD_BIT1 = '0' ADD_BIT0='1' So, start from DB1
BYTE0 BYTE1 BYTE2 BYTE3
BYTEs need to be wrote
DA0 DA1 DA2 DA3 DB0 DB1 DB2 DB3
Second stage of pipeline operation
Third stage of pipeline operation
RAM A
WR BYTE0 WR BYTE1 WR BYTE2 WR BYTE3
WR_EN BIT0 WR_EN BIT1 WR_EN BIT2 WR_EN BIT3
1 0 0 0
WR_address_A is buf_add_obs4 [12:3]
RAM B
WR BYTE0 WR BYTE1 WR BYTE2 WR BYTE3
WR_EN BIT0 WR_EN BIT1 WR_EN BIT2 WR_EN BIT3
0 1 1 1
WR_address_B is buf_add_obs1 [12:3]

Figure 5-27: example of how to write bytes to "Block RAM array" (2), pipeline operation in stage 2 and stage 3

5.3.4 Memory output controller

The task of "Memory output controller" model is to read data from "Block RAM array" and distribute those data to their corresponding sorting networks.

All interface signals of this model are list in Figure 5-28. It gets control signals from "DATA receive BUFFER controller", takes necessary data from "Block RAM array", and provides data and "filter_en" signals to sorting networks.

All signals from "DATA receive BUFFER controller" have 4 clock cycles delay before being operated. Because "Memory input controller" has three pipeline operation. The RAM also need 1 clock cycle to storage data. This delay is for avoiding read the data before writing.

Figure 5-29 can explains what's the meaning of signal "filter_position". Its value is range from 0 to "width - 1". It indicates what data need to be sent to sorting network. In Figure 5-29, data inside yellow square are sending to sorting network in current clock cycle.

"rgb_first" in Figure 5-28 indicates which colour is the first colour for filtering. For example, in the situation of Figure 5-29, green is the first colour for filtering, so rgb_first = "01". In Figure 5-2, red colour is the first colour for filtering, rgb_first = "00". This signal is very important as the existence of extra sorting network.

"rd_en" is read enable, tell "Memory output controller" if it can read data or not.

r_filter_en, g_filter_en, b_filter_en and e_filter_en are telling that if the result from associated sorting network can be used or not. As this model has 6 pipeline operations, so all values of "r_filter_en, g_filter_en, b_filter_en and e_filter_en" will be delivered to r_filter_en_out, g_filter_en_out, b_filter_en_out and e_filter_en_out after six clock cycle. Those signals are required, because all sorting networks keep working, never stop until system power off. They won't care the DMA transmission is started or not.

"rd_add_a" and "rd_add_b" are reading addresses to "Block RAM array", after one clock cycle, RAM will provide corresponding data, they are rd_a_1, rd_a_2, rd_a_3, rd_b_1, rd_b_2 and rd_b_3.

data_r_out1…9, data_g_out1…9, data_b_out1…9 and data_e_out1…9 are data for sorting networks.

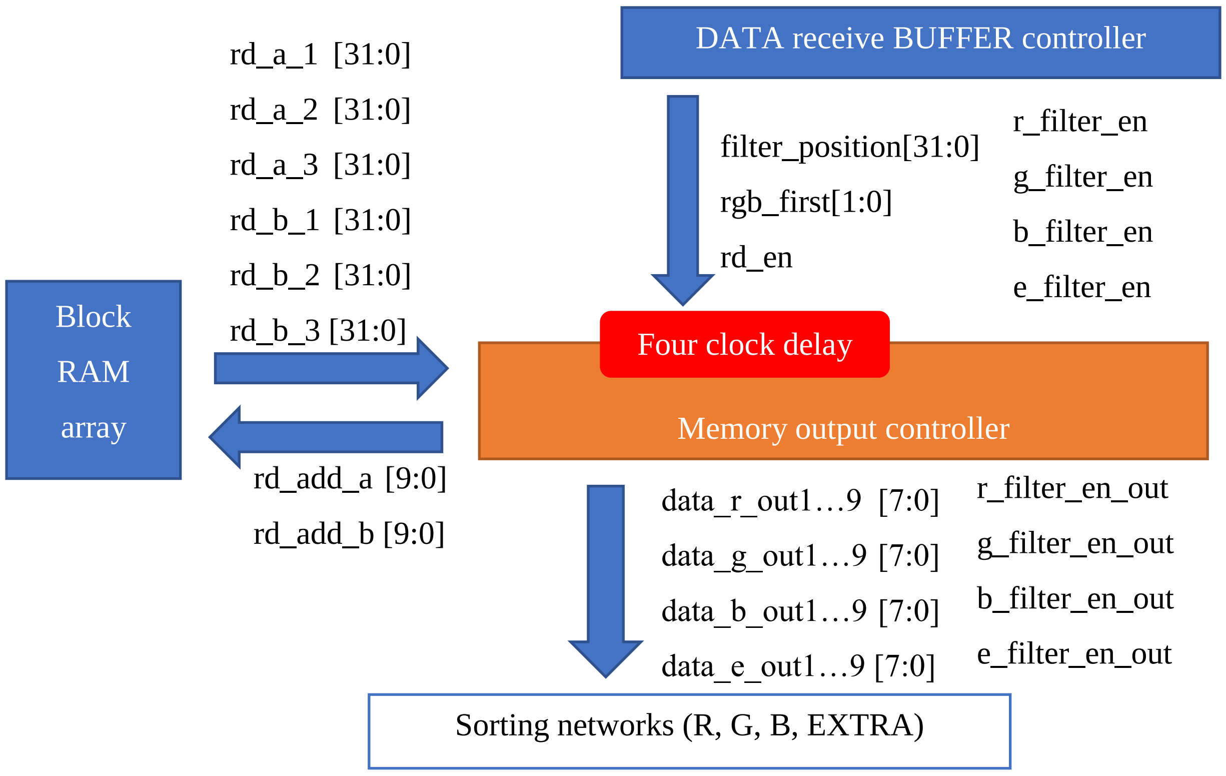


Figure 5-28: interface signals of "Memory output controller"

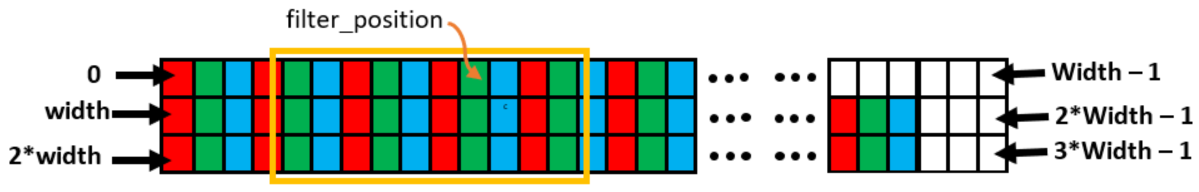
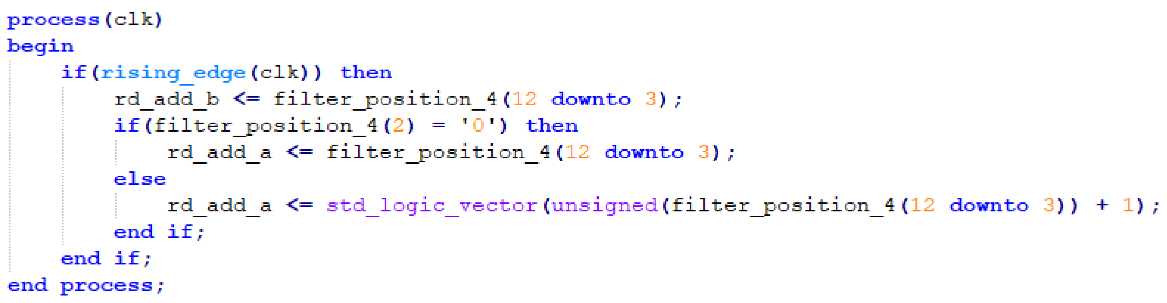


Figure 5-29: meaning of signal "filter_posisition". Data inside yellow square are sending to sorting network in current clock cycle

There are six pipeline operations in this model to fulfill its function.

The first pipeline operation is to work out values of "rd_add_a" and "rd_add_b". "filter_position" is very similar with signal "buf_add_obs1" in Figure 5-26 and Figure 5-27. Similar idea of how to use "buf_add_obs1" to work out "WR_Address_A" and "WR_Address_B" can be applied to "filter_position", "rd_add_a" and "rd_add_b". More specific details can refer to Figure 5-30. "filter_position_4" is "filter_position" after four clock delay. Reason about four clock delay has been mentioned above.

In the second pipeline stage, nothing needs to do. This stage is used to wait for "Block RAM array" provides memory data.



```
process(clk)
begin
    if(rising_edge(clk)) then
        rd_add_b <= filter_position_4(12 downto 3);
        if(filter_position_4(2) = '0') then
            rd_add_a <= filter_position_4(12 downto 3);
        else
            rd_add_a <= std_logic_vector(unsigned(filter_position_4(12 downto 3)) + 1);
        end if;
    end if;
end process;
```

Figure 5-30: determine value of "rd_add_a" and "rd_add_b"

From the third pipeline stage to the fifth pipeline stage, model need to extract necessary data from the output of "Block RAM array". The "Block RAM array" provide 24 bytes, but only 12 of them are required, it explained at section 5.3.2. Their relationship information can be found in Figure 5-31.

Figure 5-32 and Figure 5-33 are two examples about how the 3rd to the 5th pipeline stage work. First step is using bit2 of filter_position to confirm which column RAM's data should place in the front. Then, combine data from column A and column B RAM, and use filter_position bit1 and bit0 to find out the required data is starting from which position. Step three, or stage 5, is to separate all required bytes, later operation will need them.

Actually, those three steps are possible to be finished in one clock cycle. But as they work in pipeline, separate those steps won't influent the speed, and easier to debug and improve.

Figure 5-32 and Figure 5-33 are only shows the operation for ROW 1. Operations for ROW 2 and ROW 3's are totally the same.

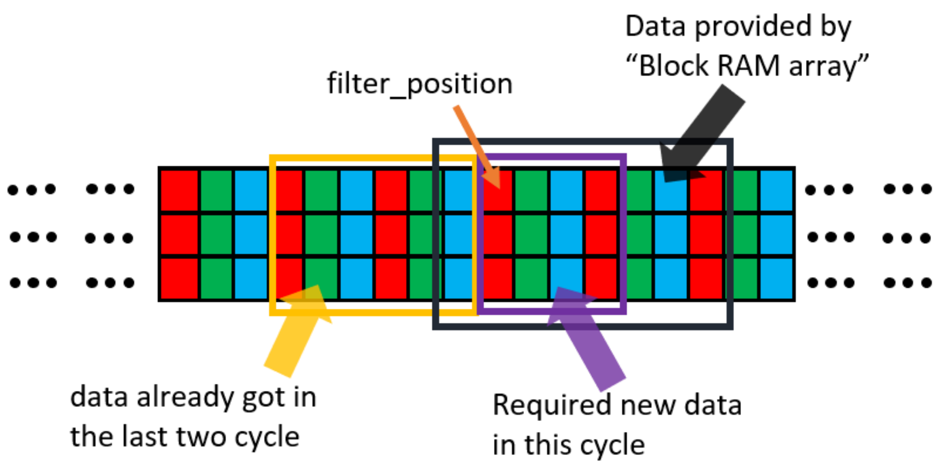


Figure 5-31: relationship between data provided by "Block RAM array" and required new data

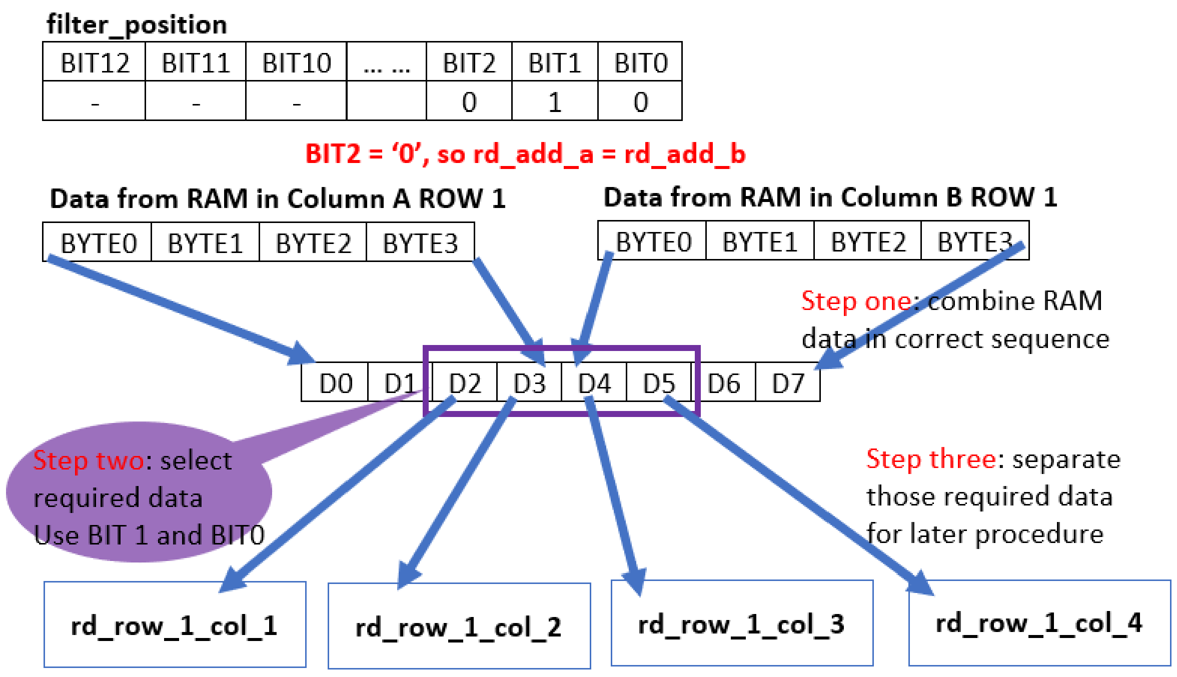

filter_position
BIT12 BIT11 BIT10 … … BIT2 BIT1 BIT0
- - - 0 1 0
BIT2 = '0', so rd_add_a = rd_add_b
Data from RAM in Column A ROW 1
BYTE0 BYTE1 BYTE2 BYTE3
Data from RAM in Column B ROW 1
BYTE0 BYTE1 BYTE2 BYTE3
Step one: combine RAM data in correct sequence
D0 D1 D2 D3 D4 D5 D6 D7
Step two: select required data Use BIT 1 and BIT0
Step three: separate those required data for later procedure
rd_row_1_col_1
rd_row_1_col_2
rd_row_1_col_3
rd_row_1_col_4

Figure 5-32: example about how to extract required data from ROW1 in correct sequence (1)

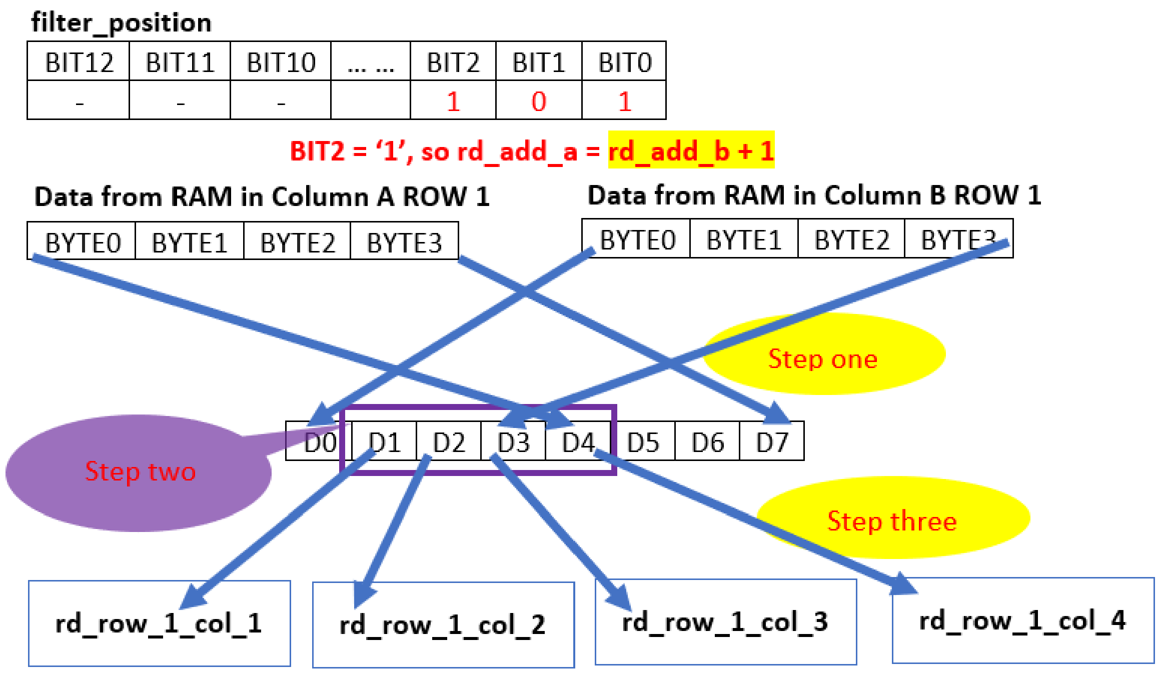


Figure 5-33: example about how to extract required data from ROW1 in correct sequence (2)

The 6th stage of pipeline operation is very significant. Output values will be decided in this stage. Figure 5-34 explain this operation clearly.

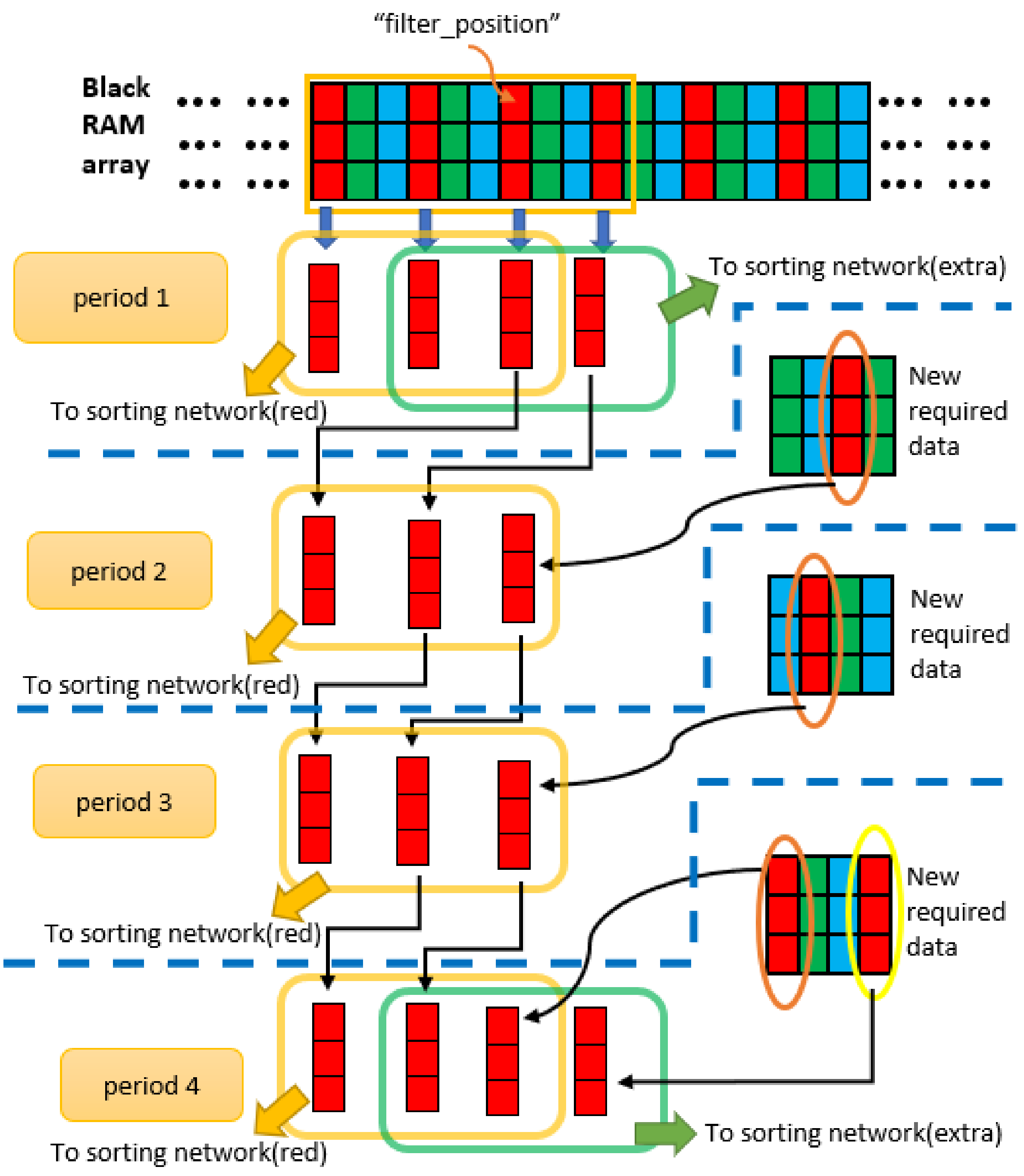


Figure 5-34: example of how to get output data to sorting network

Each period in Figure 5-34 lasts for one clock cycle. It just shows the operation for red color data. operations for Blue and green color data are the same.

Even all sorting networks will get their input data in every clock cycle, doesn't mean all of those input data's result are useable. It depends on signals "r_filter_en, g_filter_en, b_filter_en and e_filter_en".

5.3.5 DATA receive BUFFER controller

“DATA receive BUFFER controller” is the main controller of “DATA input BUFFER” in Figure 5-1. The responsibilities of this model are to give orders to memory input/output controllers according to the signals from “AXI receiver” and current status; to tell memory controllers what they need to write and what they need read. Figure 5-35 is information about “DATA receive BUFFER controller” interface.

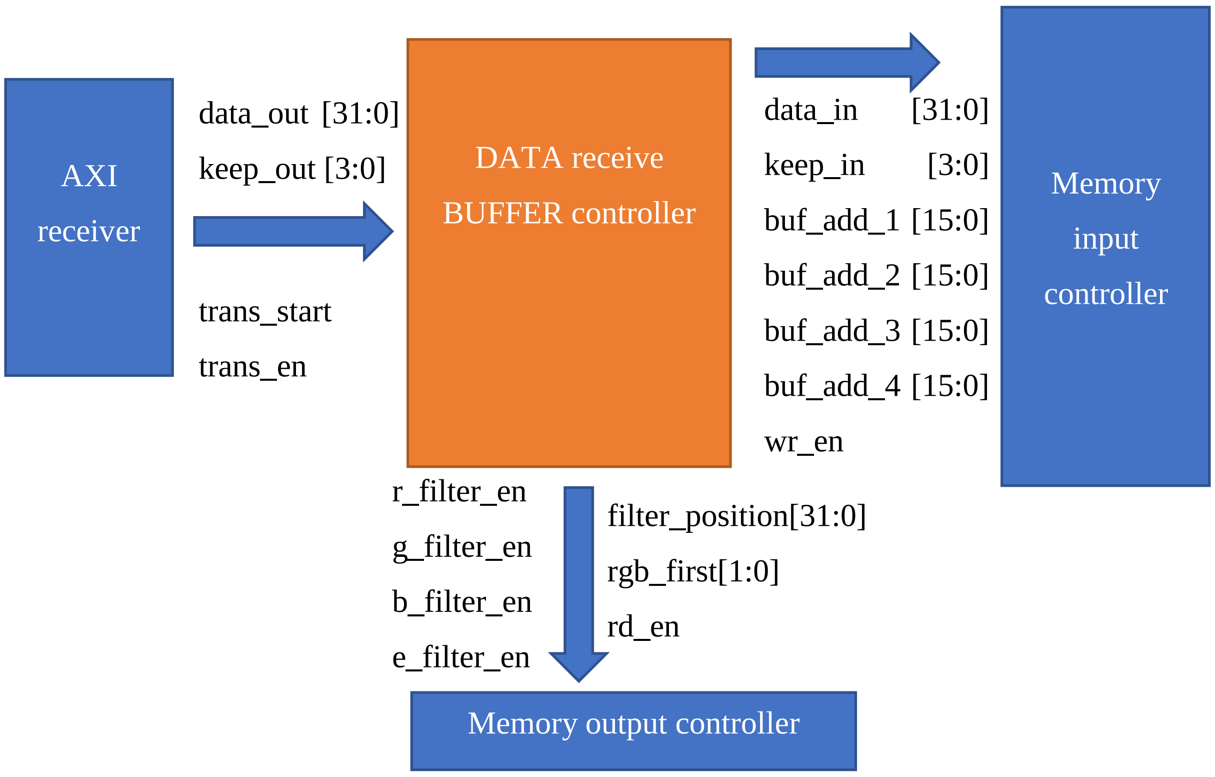


Figure 5-35: “DATA receive BUFFER controller” interface

The most significant task is to confirm the value of buf_add_1,2,3,4 and filter_position. As they control the writing and reading position of RAMs. Figure 5-36 show those signals relationship. The range of “filter_position” is from 0 to (width – 1). Its value asserts according to the value of “buf_add_1”. It always smaller or equal to “buf_add_1”, (“buf_add_1” – width) and (“buf_add_1” – 2*width), to avoid “Memory output controller” read some meaningless data. r_filter_en, g_filter_en, b_filter_en and e_filter_en value also will be asserted according to next “buf_add_1” value. Or according to how many columns can be read. If there are at least 4 useable columns, e_filter_en will be set as ‘1’, extra Sorting network will be used.

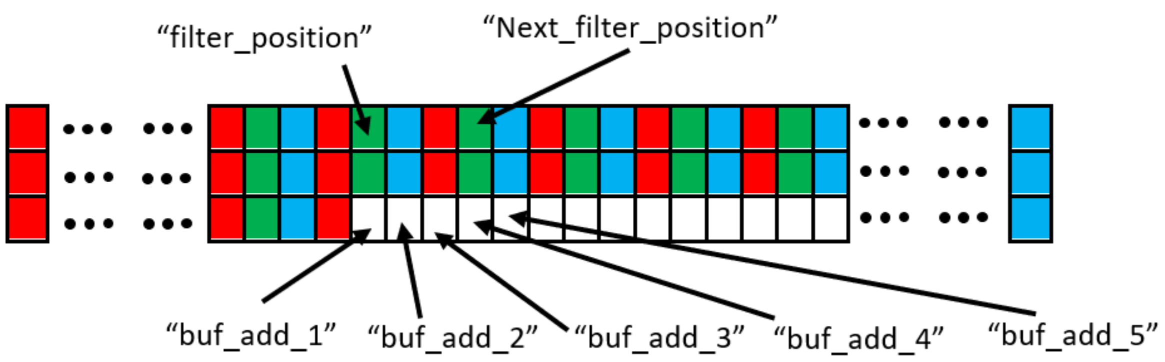


Figure 5-36: relationship of important signals in "DATA receive BUFFER controller"

Once "buf_add_1" value has been confirmed, in the same clock, buf_add_2,3,4,5 will be worked out in the same clock cycle. This is for everything can be done in one clock cycle. This model has no pipeline operation, everything is done in one clock cycle.

When model get data, "buf_add_1" value will get the data according to how many '1' in "tkeep". For example, if "tkeep" = "0111", three '1', then "buf_add_1" <= "buf_add_4". In the same clock period, buf_add_2,3,4,5 will work out their new value according to current value of "buf_add_4".

Signal "Next_filter_position" is the most difficult to decide. For saving time, "filter_position" will get the value of "Next_filter_position" in every clock cycle. In the same clock cycle, "Next_filter_position" should work out its new value according to the relationship between the value of next "buf_add_1" and current "filter_position". For example:

1) If (next "buf_add_1" > "filter_position" + 3), then "Next_filter_position" <= "Next_filter_position" + 4, extra filter will be used.
2) If (next "buf_add_1" = "filter_position" + 3), then "Next_filter_position" <= "Next_filter_position" + 3, extra filter will not be used.
3) If (next "buf_add_1" < "filter_position" + 3), then "Next_filter_position" <= "Next_filter_position" (don't change), no filter will be used.

5.4 Sorting network model implementation

In Section 2.4, three kinds of sorting network and their basic node's diagram have been mentioned. All of them can be applied to this model. But the most suitable one should be choice.

Figure 2-6 requires too many comparators, can be abandoned. Figure 2-8 seems much better than Figure 2-7, as only 19 comparators are required. But Figure 2-8 just a very simple schematic, without clock synchronize. In real implementation in FPGA, 16 “8-bit register” should be added. Figure 5-37 is real implementation structure of Figure 2-8 in FPGA, every part is synchronizing with clock. Registers can make sure correct data will be send to correct comparators. Besides, as all sorting network will keep working since system power on, “en” signal is necessary to indicate the output data is usable or not. Figure 2-7 also has similar requirement, but only 4 “8-bit register” will be enough. For comparing those two methods, I implemented both methods and got resources usage information in Figure 5-38.

According to the result of Figure 5-38, Figure 2-8 requires less look up table (LUT) resource, but requires LUTRAM and more FF. So, performance of Figure 2-7 and Figure 2-8 are similar. Considering Figure 2-8 require near 40 LUTs less than Figure 2-7, other resource requirements are very close, solution of Figure 2-8 has been chosen.

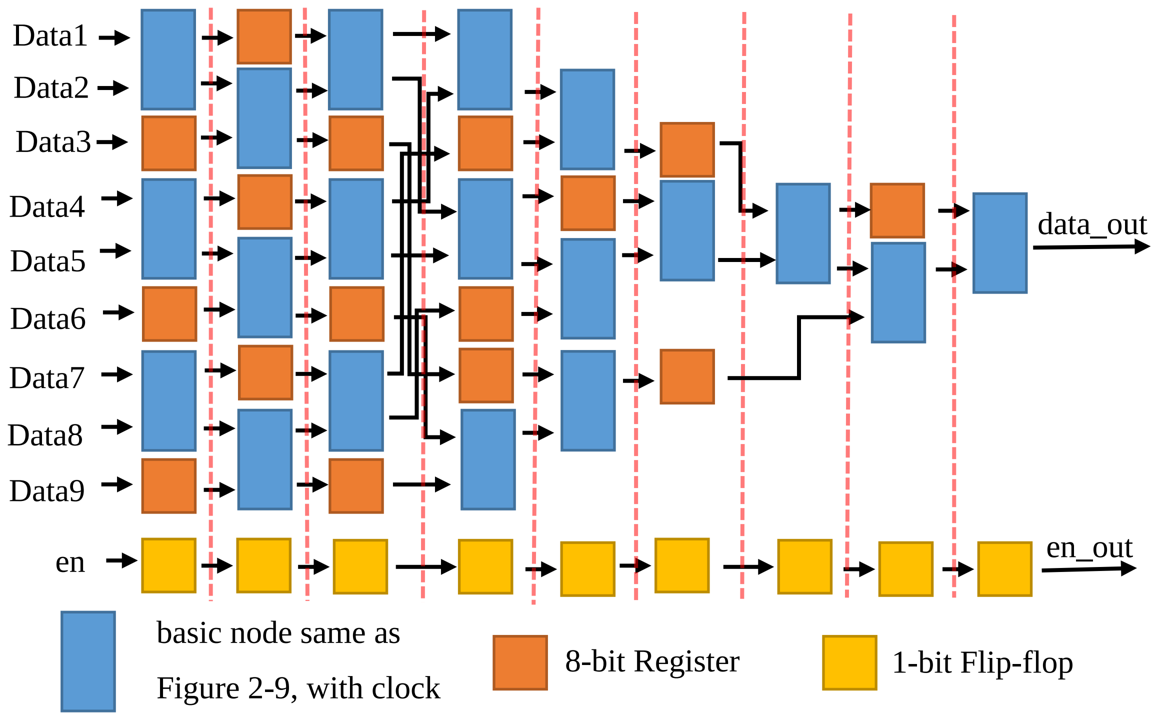
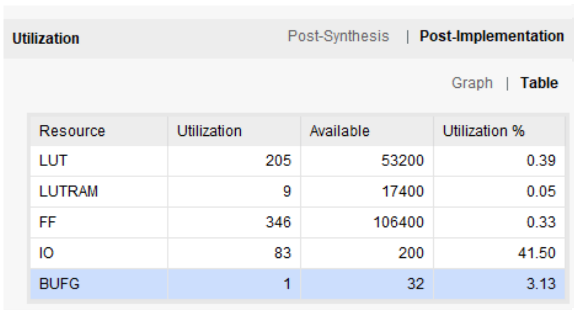


Figure 5-37: real implementation structure of Figure 2-8 in FPGA

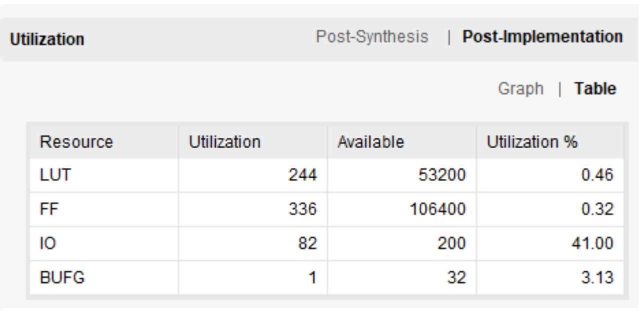


Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	205	53200	0.39
LUTRAM	9	17400	0.05
FF	346	106400	0.33
IO	83	200	41.50
BUFG	1	32	3.13

(A)



Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	244	53200	0.46
FF	336	106400	0.32
IO	82	200	41.00
BUFG	1	32	3.13

(B)

Figure 5-38: required resource for sorting networks (A) using Figure 2-8 diagram (B) using Figure 2-7 diagram

5.5 Data output FIFO

Data output FIFO is also an important part of this IP. Because DMA receiving channel communication is not always reliable. Nothing can make sure that DMA channels is able to always keep receiving data without any interrupt during data transmission. The main purpose of this FIFO is to save some output data, in case DMA channel slave is stopped.

Figure 5-39 is basic structure of Data output FIFO and its interface. This model is divided by two parts, FIFO arrays and FIFO controller. FIFO array contains four FIFO models, and FIFO controller get data from sorting network models, provide correct data and orders to FIFO array, receive data from FIFO array and interact with other models.

In Figure 5-39, when Data output FIFO is full, FIFO controller’s output signal “full” will be set to ‘1’ and tell “AXI receiver” to stop receiving data from DMA channels. Otherwise, FIFO controller will keep receiving data from four sorting networks. “(R, G, B, E) keep” signals tell FIFO controller that, in current clock cycle, the corresponding sorting network output data is usable or not. Usable data will be saved to FIFO array.

As the quantity of usable data from sorting network is not fixed, especially data from the extra sorting network (many procedures may no need to use this network). But in every clock cycle, usable data have to be saved in one clock cycle. So, the idea is to make 4 data FIFO. Input data width of each FIFO is 8 bits. In this case, model will be able to read and write 0 to 4 bytes of data in one clock. Figure 5-40 is the diagram of “FIFO array”. It is created by Xilinx fifo generator IP core. Their all input and output ports are independently.

All signals between FIFO controller and AXI sender are all explained in section 5.2.2.

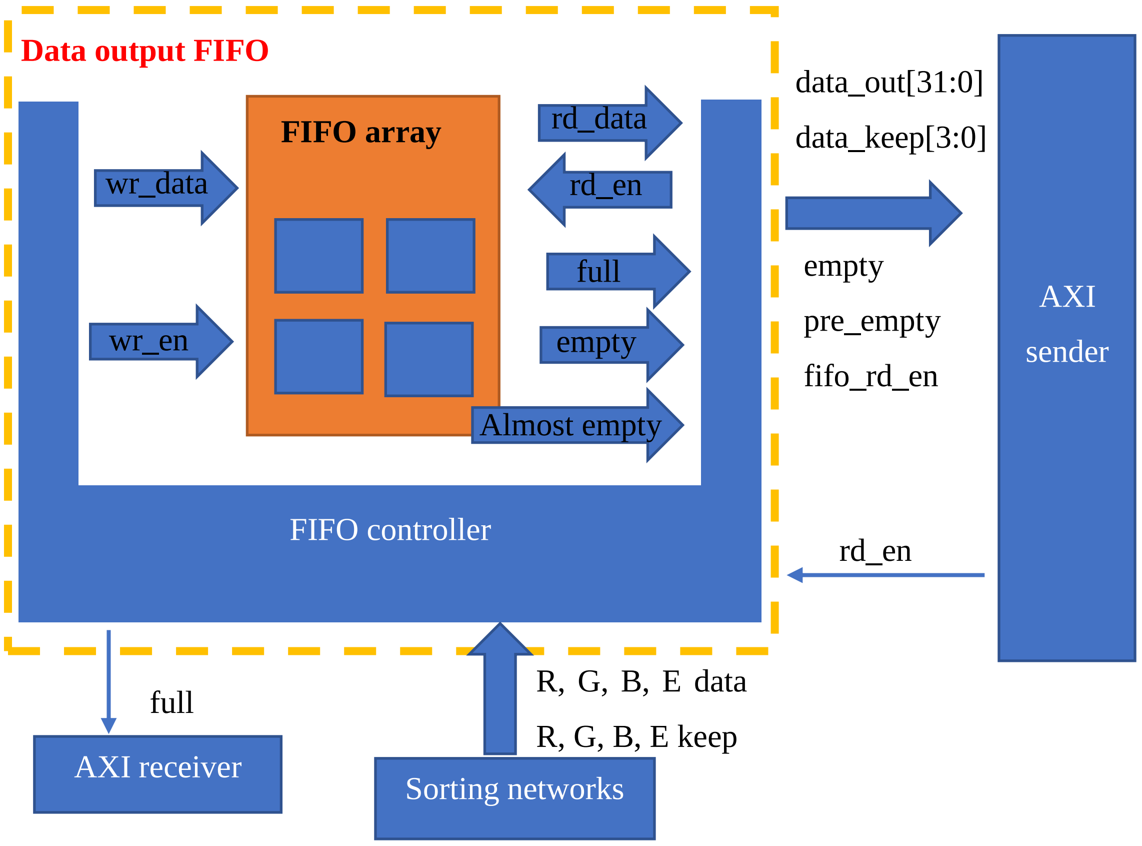


Figure 5-39: basic structure and interface of DATA output FIFO

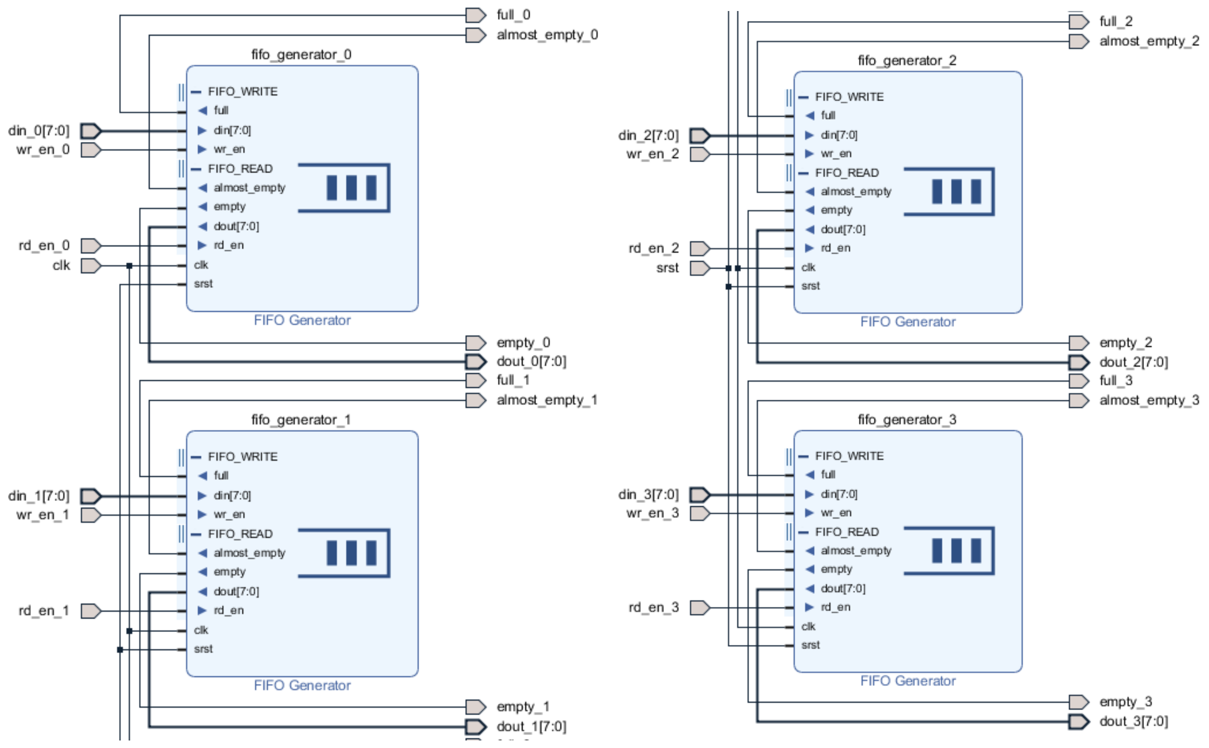


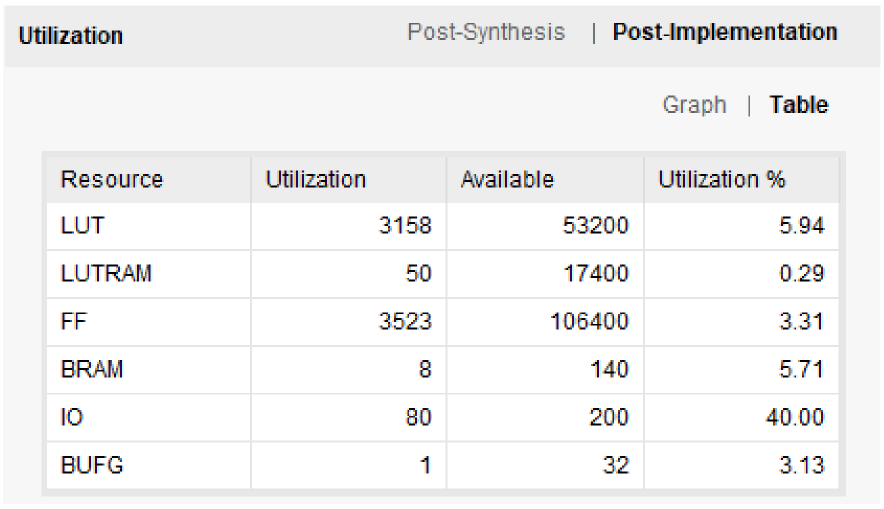
Figure 5-40: diagram of “FIFO array”

6 Experiment result and analysing

6.1 Implementation result and resource usage

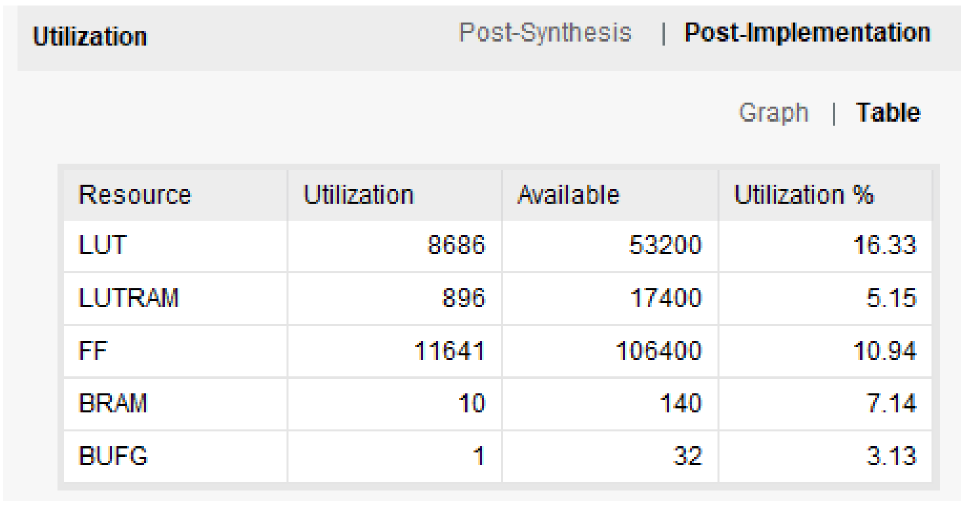
Figure 6-1 shows Digital Image Filter IP resource usage in PL side. Every type of resource usage is less than 10%, and overall less than 5%. Very light weight.

And the resource usage situation of whole system is shown in Figure 6-2. About 10% of PL side resource is used.



Resource	Utilization	Available	Utilization %
LUT	3158	53200	5.94
LUTRAM	50	17400	0.29
FF	3523	106400	3.31
BRAM	8	140	5.71
IO	80	200	40.00
BUFG	1	32	3.13

Figure 6-1: Digital Image Filter IP resource usage in PL side



Resource	Utilization	Available	Utilization %
LUT	8686	53200	16.33
LUTRAM	896	17400	5.15
FF	11641	106400	10.94
BRAM	10	140	7.14
BUFG	1	32	3.13

Figure 6-2: entire system resource usage in PL side

6.2 Image transmission and receiving in PS side

After finishing the PL side design, PS side needs a Linux-c program to transfer necessary data to DMA channel, receive data from DMA channel and form those received data into a new image.

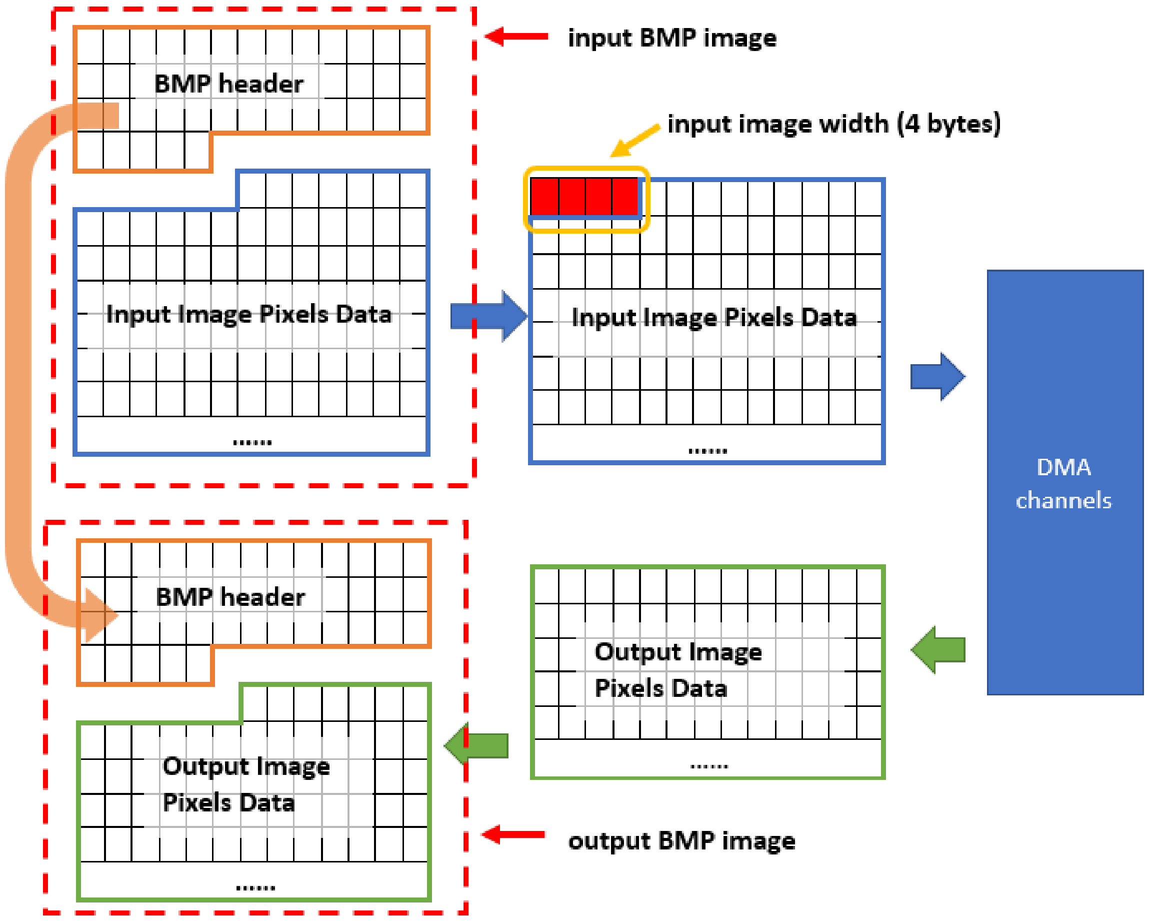


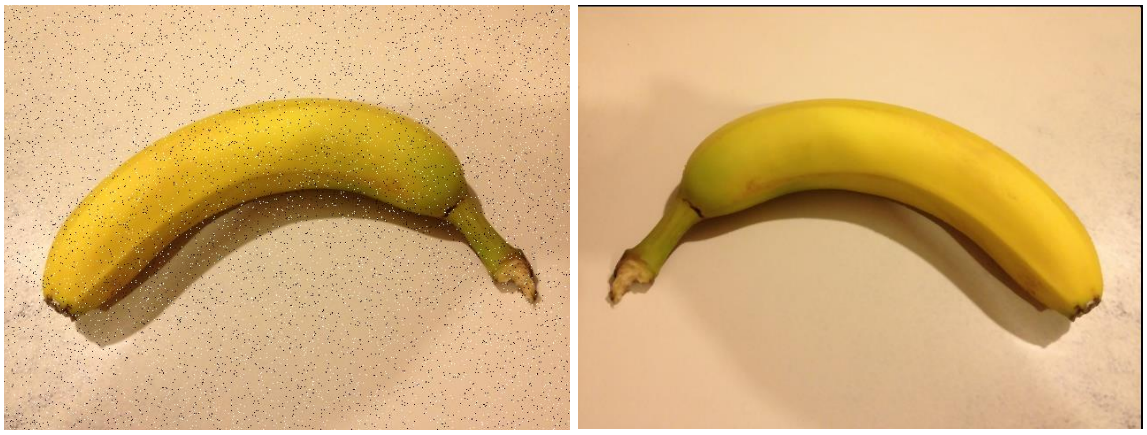
Figure 6-3: Linux-C program in PS side

According to the information in 2.3.1, BMP header should not be transmitted to PL side. Digital image filter only requires the information about image width, and Pixels data. In PS side, some Linux-C programming should be done. The function of the program is shown in Figure 6-3.

As mentioned in section 4.5, there are some useful DMA communication examples already done by others. So Figure 6-3 function can be fulfilled by modifying those example codes. Example file "axidma_transfer.c" is a good choose. Some introduction about this c file can be found at the end of section 4.5, the introduction about "axidma_transfer".

6.3 Image median filter result

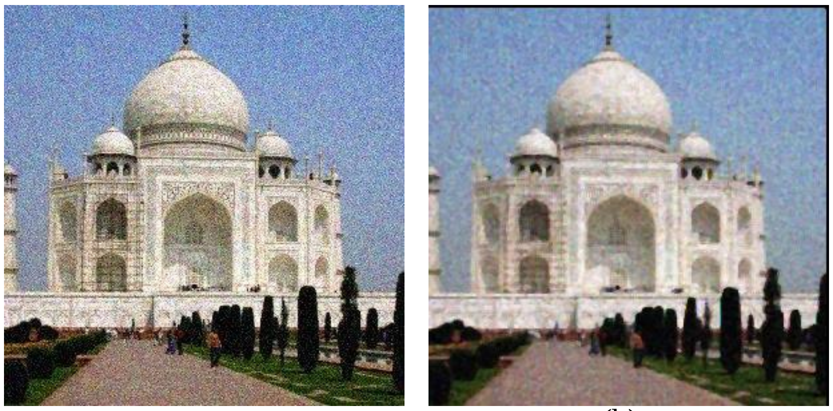
Figure 6-4 (a) is the original image with salt-and-pepper noise. After the operation of this system, get Figure 6-4(b), the result is very good and almost all noise is filtered.



(a) (b)

Figure 6-4: (a) original image with low density salt-and-pepper noise [29] (b) image filtering result

Figure 6-5(a) is an image with many different kinds of noise, not only “salt and pepper noise”. In this case, median filter cannot handle them. So, from the result in Figure 6-5(b), median filter fail to filter all types of noise.



(a) (b)

Figure 6-5: (a) original image with many types of noise [30] (b) image filtering result

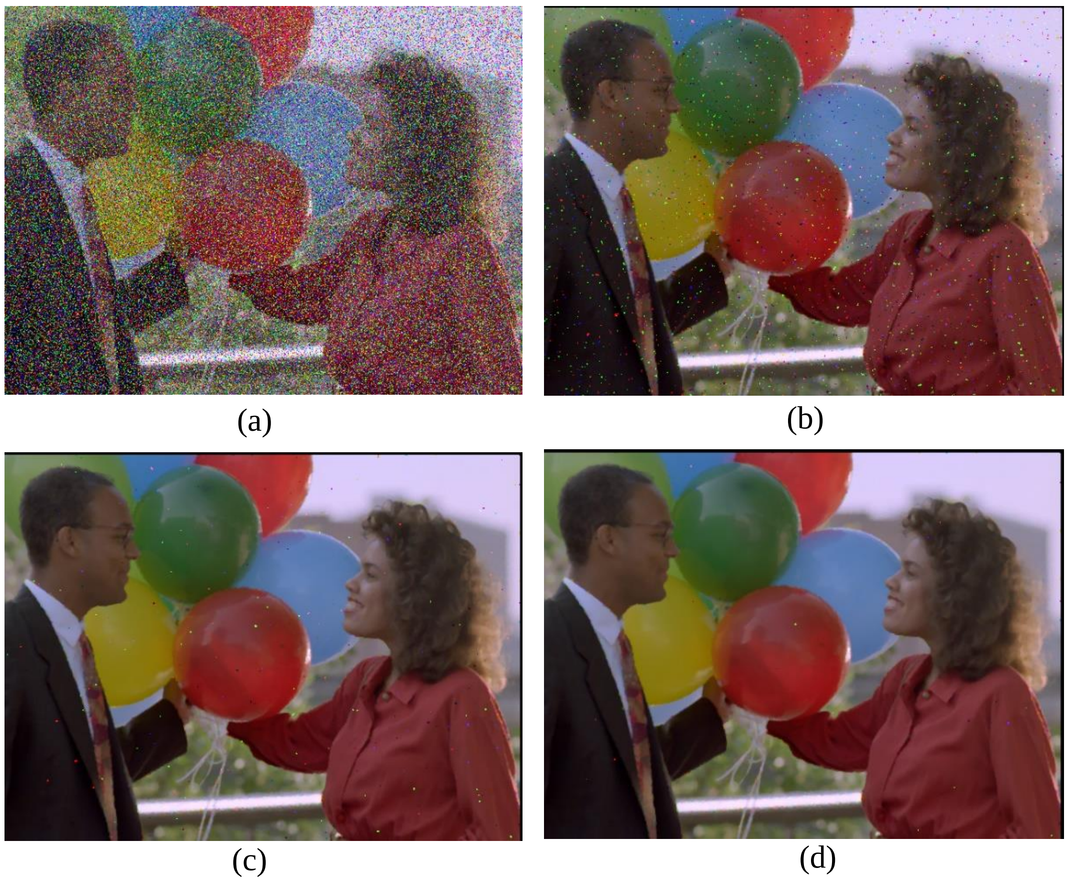


Figure 6-6: (a) original image with high density salt-and-pepper noise [31] (b) result after the first time median filtering (c) result after the second time median filtering (d) result after the third time median filtering

If the density of salt-and-pepper noise is too high, 3x3 median filter may be not able to filter all noise in one time, but such process can be done for more than once, until the noise decreases to acceptable low level. Figure 6-6 is an example of this procedure.

6.4 Time consumption analysing

For estimate the time consumption of filtering image in this system, first of all, Integrated Logic Analyzer (ILA) IP should be added in the diagram. ILA IP core is a logic analyser that can be used to monitor the internal signals of a design [32].

As it is impossible and not practically to use a time counter to measure the exact time consumption of filtering an image. So, the idea is using ILA to monitor the input and output signal of “AXI DMA IP” core. If “tvalid”, “tready” and every bit of “tkeep” are keeping in value ‘1’ during the whole procedure of image filtering and data transferring, we will be able to conclude that DMA channels run in full speed during this whole procedure. Then, theoretical calculation result will be reliable.

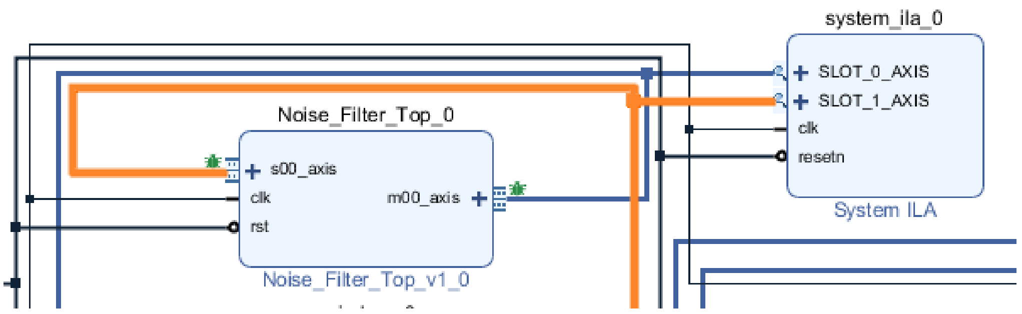
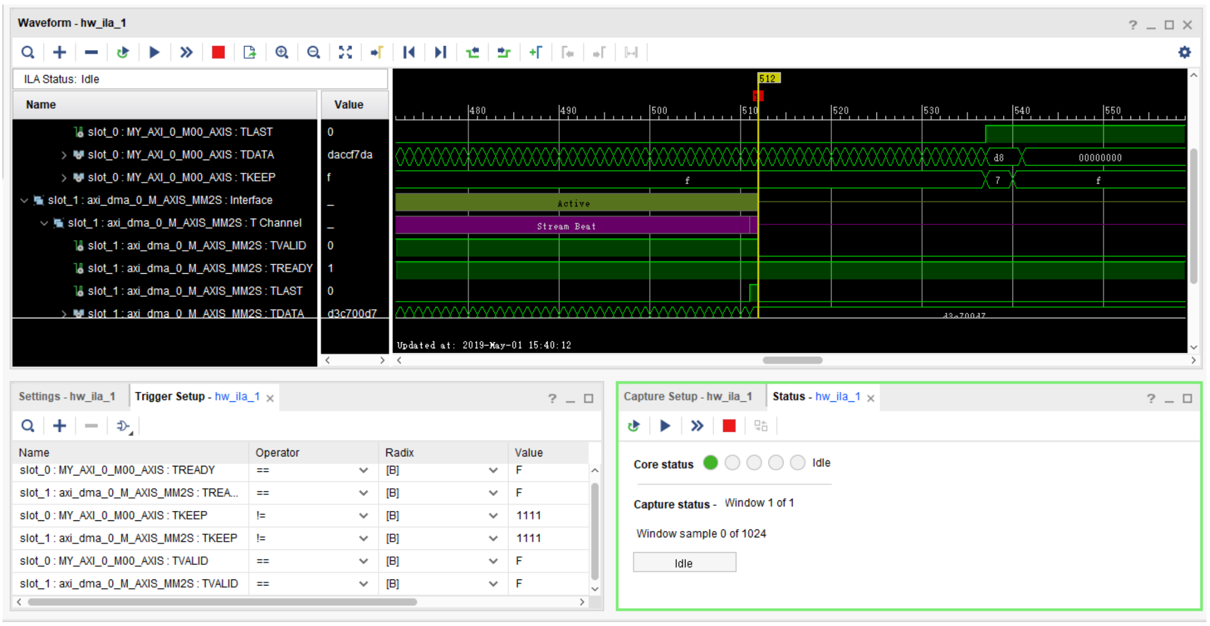


Figure 6-7: using system ILA IP to monitor input and output of “Digital image filter”


Waveform - hw_ila_1

ILA Status: Idle

Name	Value
slot_0 : MY_AXI_0_M00_AXIS : TLAST	0
slot_0 : MY_AXI_0_M00_AXIS : TDATA	daccf7da
slot_0 : MY_AXI_0_M00_AXIS : TKEEP	f
slot_1 : axi_dma_0_M_AXIS_MM2S : Interface	_
slot_1 : axi_dma_0_M_AXIS_MM2S : T Channel	_
slot_1 : axi_dma_0_M_AXIS_MM2S : TVALID	0
slot_1 : axi_dma_0_M_AXIS_MM2S : TREADY	1
slot_1 : axi_dma_0_M_AXIS_MM2S : TLAST	0
slot_1 : axi_dma_0_M_AXIS_MM2S : TDATA	d3c700d7

Updated at: 2019-May-01 15:40:12

Settings - hw_ila_1 | Trigger Setup - hw_ila_1

Name	Operator	Radix	Value
slot_0 : MY_AXI_0_M00_AXIS : TREADY	==	[B]	F
slot_1 : axi_dma_0_M_AXIS_MM2S : TREA...	==	[B]	F
slot_0 : MY_AXI_0_M00_AXIS : TKEEP	!=	[B]	1111
slot_1 : axi_dma_0_M_AXIS_MM2S : TKEEP	!=	[B]	1111
slot_0 : MY_AXI_0_M00_AXIS : TVALID	==	[B]	F
slot_1 : axi_dma_0_M_AXIS_MM2S : TVALID	==	[B]	F

Capture Setup - hw_ila_1 | Status - hw_ila_1

Core status Idle

Capture status - Window 1 of 1

Window sample 0 of 1024

Idle

Figure 6-8: triggering result during hardware debugging

In the lower-left part of Figure 6-8, I setup six triggering situation. If one of them happen, all signal lines of AXI master and slave will be captured. In the “value” column, “F” means signal’s value from ‘1’ fall to ‘0’. According to the result in the upper side of Figure 6-8, ILA is triggered at the end of image “AXI DMA” IP Master transmission. After that, the slave signals still running in full speed. As the result, we can conclude that, DMA channels is running in full speed in the whole procedure. This experiment has been done for many different sizes input images, all of those experiment got same result.

Figure 6-9 shows how to calculate the clock cycles consumption for one image filtering operation. Signals “M_AXIS_MM2S” and “M_AXIS_S2MM” can be found in Figure 1-1. At the beginning of the operation, DMA channel will transmit image pixels data to “digital image filter”. When “digital image filter” acquire enough pixel’s data (two-lines- plus three- or four-pixels data), it will start to do its pipeline operation. The pipeline operation will take 26 clock cycles to get the first output data. After that, every lock will work out a data (4 bytes). After “M_AXIS_MM2S” transmission finish, there are 26 clock cycles leave for “M_AXIS_S2MM”, because “Digital Image Filter” requires this time to work out the last pipeline data.

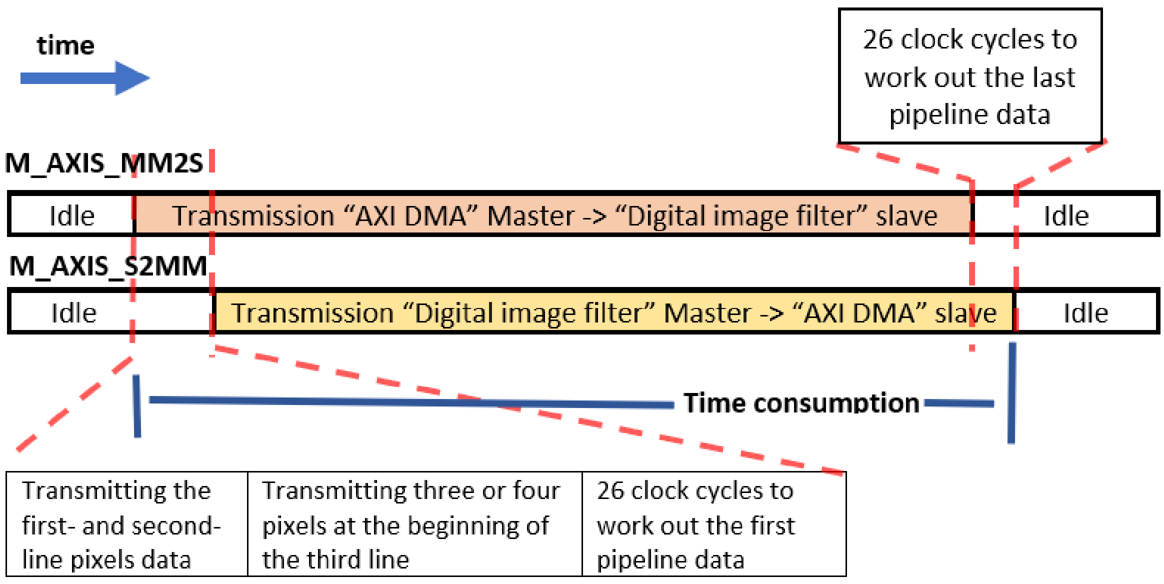

time
26 clock cycles to work out the last pipeline data
M_AXIS_MM2S
Idle
Transmission “AXI DMA” Master -> “Digital image filter” slave
Idle
M_AXIS_S2MM
Idle
Transmission “Digital image filter” Master -> “AXI DMA” slave
Idle
Time consumption
Transmitting the first- and second-line pixels data
Transmitting three or four pixels at the beginning of the third line
26 clock cycles to work out the first pipeline data

Figure 6-9: Clock cycles consumption calculation diagram

Look closer to Figure 6-9, clock cycles consumption only depends on input image size. The time consumption can be calculated by the frequency of clock and Clock cycles consumption. As every clock will handle 4 bytes data, the algorithm is:

$$\text{time} = \frac{\frac{\text{width} * \text{height} * 3}{4} + 26\ \textit{clock cycles}}{\text{clock frequency}}$$

Clock frequency is 100Mhz. Some calculation result can be found in Table 6-1.

Table 6-1: calculation result for some different size of images

width	height	Clock cycle consumption	Time consumption (ms)
300	200	45026	0.45026
640	480	230426	2.30426
800	600	360026	3.60026
1024	768	589850	5.8985
1280	1024	983066	9.83066
1920	1080	1555226	15.55226

7 conclusion and Future work

In this work, I established a 3x3 median filtering system for image filtering. Linux OS runs properly in PS side. Because Xilinx ZC702 evaluation board equips all necessary hardware Peripheral (e.g. Ethernet interface, USB JTAG, UART, etc.), many necessary debugging procedures have been done. Using SSH and Samba, PC (windows) can communicate and interact with the device conveniently.

According to the results in Chapter 6, the system can do 3x3 median image filtering work correctly. DMA channels' communication runs smoothly and the interface of "Digital Image Filter" is totally compatible with AXI4-stream protocol. System's time consumption and resource usage are low enough. So, I can conclude that the task is done. And the source code of this design can be downloaded in [34].

In my opinions, this system has great potential to upgrade. There are a lot of advance and useful improvements can be done. The following future work are possible to do:

1) For some bigger size images with higher density salt-and-pepper noise, using bigger filtering window can filter more noise in one time. But big filtering window may don't have good effective for the image corrupted by lower density noise. Adaptive Median Filter, which is mentioned in [16] is a good choose to replace the 3x3 median filter. Because system works in pipeline, size of filtering window won't influent speed.
2) There are two useful methods for increasing the speed or decreasing time consumption of each filtering operation. First is to increase the PL side clock frequency. But the maximum PL side clock frequency of this SOC device is 250Mhz. And higher clock frequency may influent stability. More effective and practical method is to increase DMA channel data width. The data width in this design is 32bit. The maximum data width of DMA channel is 1024 bits[18]. Using wider data width will requires more PL side resource, but still affordable.
3) Applying this image filtering system for video, or even livestream is possible. Xilinx provides "AXI VDMA" IP core. Comparing with "AXI DMA", "AXI VDMA" IP provide useful functions for video data transfer[33]. Using FPGA to do filtering task for video can highly decrease CPU's burden.

Reference

[1] Z Vasicek, L Sekanina "Novel hardware implementation of adaptive median filters" IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2008 11th

[2] G Perrot, S Domas, R Couturier "Fine-tuned High-speed Implementation of a GPU-based Median Filter" Journal of Signal Processing Systems, 2014

[3] Miguel A. Vega-Rodríguez, Juan M. Sánchez-Pérez, Juan A. Gómez-Pulido "An FPGA-based implementation for median filter meeting the real-time requirements of automated visual inspection systems" 2002

[4] J Scott, M Pusateri, MU Mushtaq "Comparison of 2D median filter hardware implementations for real-time stereo video" IEEE Applied Imagery Pattern Recognition Workshop, 2008 37th

[5] Xilinx, "PG035 - AXI4-Stream Interconnect v1.1 Product Guide (v1.1)", Oct 04, 2017

[6] Xilinx, "Block Memory Generator v8.3 LogiCORE IP Product Guide (PG058)", April 5, 2017

[7] Xilinx, "7 Series FPGAs Memory Resources, UG473 (v1.13)", February 5, 2019

[8] Xilinx, "Zynq-7000 SoC Technical Reference Manual", July 1, 2018

[9] Z Vasicek, V Mrazek "Trading between quality and non-functional properties of median filter in embedded systems" Genetic Programming and Evolvable Machines, 2017

[10] KS Srinivasan, D Ebenezer "A new fast and efficient decision-based algorithm for removal of high-density impulse noises" IEEE signal processing letters, 2007

[11] F Kong, W Ma "A fast adaptive median filtering algorithm" 2010 2nd International Conference on Industrial and Information Systems, 2010

[12] G Perrot, S Domas, R Couturier "Fine-tuned High-speed Implementation of a GPU-based Median Filter" Journal of Signal Processing Systems, 2014

[13] Karl Pauwels, Matteo Tomasi, Javier Dı´az, Eduardo Ros, Marc M. Van Hulle "A comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features" IEEE Transactions on Computers (Volume: 61, Issue: 7, July 2012)

[14] Jeff Johnson, "Using the AXI DMA in Vivado" Retrieved April 4th 2019, from http://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html

[15] JL Smith "Implementing median filters in xc4000e fpgas" Xilinx Xcell, 1996

[16] Z Vasicek, L Sekanina "An area-efficient alternative to adaptive median filtering in fpgas" International Conference on Field Programmable Logic and Applications, 2007

[17] Xilinx “Processing System 7 v5.5 LogiCORE IP Product Guide (PG082)” May 10, 2017

[18] Xilinx “AXI DMA v7.1 LogiCORE IP Product Guide (PG021)” April 4, 2018

[19] Xilinx “AXI4-Stream FIFO v4.1 LogiCORE IP Product Guide (PG080)” April 6, 2016

[20] Xilinx “LogiCORE IP Concat v2.1 Product Brief (PB041)” April 6, 2016

[21] Xilinx “Zynq-7000 All Programmable SoC Software Developers Guide (UG821)” March 18, 2014

[22] Retrieved April 29th, 2019, from https://ati.ttu.ee/wiki/soc/index.php/Lab_2

[23] “Device Tree Reference” Retrieved May 5th, 2019, from https://elinux.org/Device_Tree_Reference

[24] CSDN, Retrieved April 29th, 2019, from https://blog.csdn.net/long_fly/article/details/80482248

[25] Xilinx “Linux DMA In Device Drivers” Retrieved April 29th, 2019, from https://www.xilinx.com/video/soc/linux-dma-in-device-drivers.html

[26] Retrieved April 29th, 2019, from https://github.com/bperez77/xilinx_axidma

[27] Retrieved April 30th, 2019, from https://engineering.purdue.edu/ece264/17au/hw/HW15

[28] David Charlap “The BMP File Format, Part 2” April 01, 1995, Retrieved May 5th, 2019, from http://www.drdobbs.com/the-bmp-file-format-part-2/184409533?pgno=5

[29] “COMPUTER VISION” Retrieved April 30th, 2019, from http://www.csfieldguide.org.nz/releases/1.9.9/ComputerVision.html

[30] Harsh Prateek Singh, Ayush Nigam, Amit Kumar Gautam, Aakanksha Bhardwaj, Neha Singh “Noise Reduction in Images using Enhanced Average Filter” International Journal of Computer Applications, 2014

[31] “IMAGE_DENOISE Remove Noise from an Image” Retrieved April 30th, 2019, from https://people.sc.fsu.edu/~jburkardt/f_src/image_denoise/image_denoise.html

[32] Xilinx “Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide (PG172)” October 5, 2016

[33] Xilinx “AXI Video Direct Memory Access v6.2 LogiCORE IP Product Guide (PG020)” November 30, 2016

[34] Thesis design source code: https://github.com/miaoyangpeng/AXI_digital_median_filter_IP