Report for SoC Lab

English version

Name：陈宇阳 Student Number：3220101054 Name：邓高远 Student Number：3220106412

2025

目录

[Lab Content Overview 2](#_Toc203500117)

[Device and Tool Introduction 2](#_Toc203500118)

[Lab Report 2](#_Toc203500119)

[**Phase 2: HDMI controller design** 2](#_Toc203500120)

[**Phase 3: AXI\_Lite Slave interface for HDMI controller design** 8](#_Toc203500121)

[**Phase 4: AXI\_Full Master interface for HDMI controller design** 17](#_Toc203500122)

[**Phase 5: Real Time Camera Input to HDMI Output** 28](#_Toc203500123)

[Issues Record 46](#_Toc203500124)

[Personal summary 49](#_Toc203500125)

[Reference 52](#_Toc203500126)

[Appendix 53](#_Toc203500127)

# Lab Content Overview

The lab has 5 phases in total. Phase 1 is a basic project that aims to familiarize students with the workflow of Vivado block design using the powerful ZYNQ FPGA chip and its integrated Arm CPU. The real challenge begins at phase 2, through which we are obliged to design an HDMI controller without integrating it into the SoC system. After finishing phase 2, we are going to design an AXI\_Lite bus slave interface for the HDMI controller in phase 3, which makes it possible to be controlled by the program running on the CPU. Then, a master type of AXI\_Full interface is designed to meet the requirement of higher performance in phase 4. In the end, we shall try to use the circuit we have designed to reach the goal of showing the content of the camera on the screen in real time. In addition, to focus on the main content of the lab, phase 1 will not be included in this report.

# Device and Tool Introduction

The devices and tools we use during the lab are as follows:

a. Zedboard(carry the FPGA chip of xc7z020clg484)

b. Vivado and Vivado SDK 2018.2

c. Raspberry Pi Camera V2.1(carry the camera of IMX219)

# Lab Report

## **Phase 2: HDMI controller design**

In phase 2, we are going to design an HDMI controller using Verilog HDL. Before designing a 1080P HDMI controller, we need to design a 720P HDMI controller for which requires less in timing, so it is easier to achieve.

The first thing we need to know is how HDMI works. [[1]](https://en.wikipedia.org/wiki/HDMI)**High-Definition Multimedia Interface** (HDMI) is a proprietary digital interface used to transmit high-quality video and audio signals between devices. It is commonly used to connect devices such as televisions, computer monitors, projectors, gaming consoles, and personal computers. HDMI supports uncompressed video and either compressed or uncompressed digital audio, allowing a single cable to carry both signals. Usually, HDMI drives an external display by a kind of differential signal called **Transition-minimized differential signaling** (TMDS). But on Zedboard, we use an HDMI Transmitter chip, ADV7511, to generate the signals that are needed. So, what we should do is give the required signal to the ADV7511. ADV7511 supports multiple modes, but it is designed to support “HDMI 1.4 and DVI 1.0 compatible supporting 1080P60 with 16-bit, YCbCr, 4:2:2 mode color” only on the Zedboard. In conclusion, our mission is to send the right control signals to the ADV7511, and the control signals are listed in the table below:

|  |  |
| --- | --- |
| Signal Name | Description |
| hdmi\_scl | I2C interface. Supports CMOS logic levels from 1.8V to 3.3V |
| hdmi\_sda |
| hdmi\_clk | Video Clock Input. Supports typical CMOS logic levels from 1.8V up to 3.3V |
| hdmi\_hsync | Horizontal Sync Input |
| hdmi\_vsync | Vertical Sync Input |
| hdmi\_de | Data Enable signal input for Digital Video |
| hdmi\_d[15:0] | Video Data Input |

Table 1. HDMI Signals

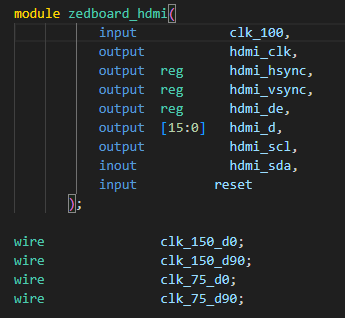
The signal definition in Verilog HDL code is like the picture below:

Figure 1. HDMI Signals

The key control signals here are hdmi\_hsync and hdmi\_vsync. They works like Figure 2.

The elements in Figure 2 are:

**Horizontal Timing (across a single line):**

**HSync**: The horizontal synchronization signal that marks the start of a new line.

**Horizontal Sync Time**: Duration of the HSync pulse.

**Back Porch**: Idle time after the HSync pulse before the visible area begins.

**Horizontal Active Start**: The start of the visible video.

**Addressable Video**: The actual visible image content.

**Left/Right Borders**: Optional padding around the image.

**Front Porch**: Idle time after the visible area before the next HSync.

**Horizontal Blank Start/End**: Marks the beginning and end of the horizontal blanking interval, which includes the front porch, sync, and back porch.

**Vertical Timing (down a frame of lines):**

**VSync**: The vertical synchronization signal that marks the start of a new frame.

**Vertical Sync Time**: Duration of the VSync pulse.

**Back Porch**: Idle lines after the VSync pulse before visible content.

**Vertical Active Start**: The start of the visible lines.

**Addressable Video**: Actual lines of video content.

**Top/Bottom Borders**: Optional blank lines around the visible content.

**Front Porch**: Idle lines after visible content and before VSync.

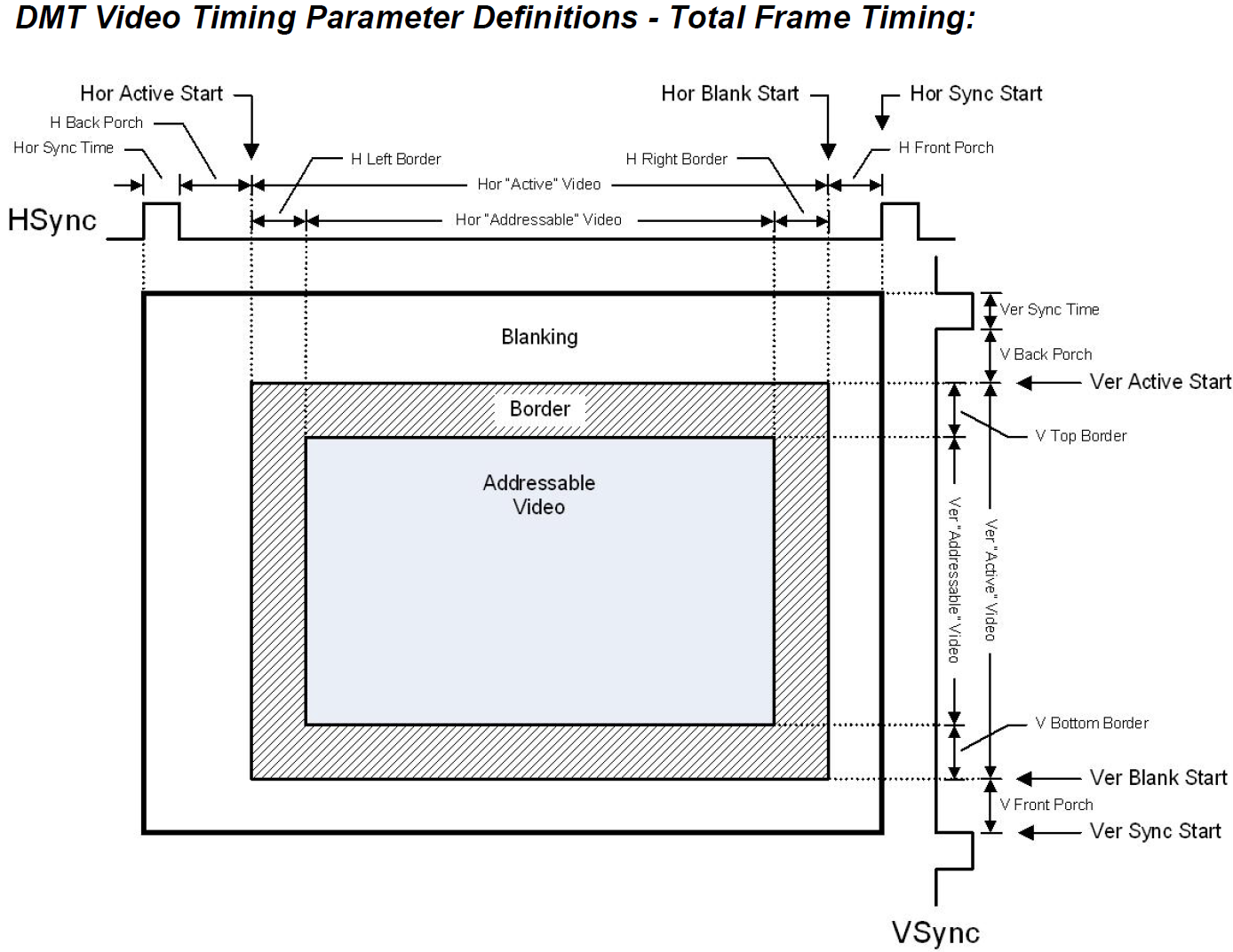
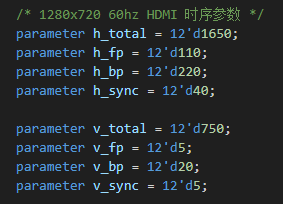
**Vertical Blank Start/End**: Marks the start and end of the vertical blanking period.

Figure 2. HDMI Timing

In 720P HDMI Timing, the parameters in Figure 2 is defined as in Figure 3:

Figure 3. 720P HDMI Parameters

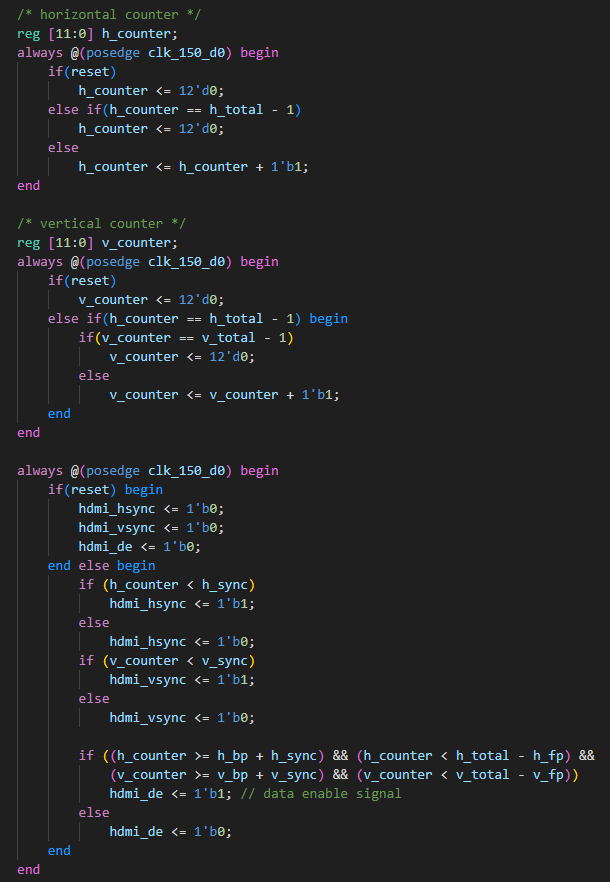
Then we can design the circuit that can drive the ADV7511 at the right timing. The Verilog HDL code is shown in Figure 4.

Figure 4. Generation of hdmi\_vsync, hdmi\_hsync and hdmi\_de

With the code of the I2C initialization for the ADV7511.**(The code is in the file i2c\_sender.v**) and the clock generation module**(clk\_pll.v, 75MHz for 720P and 150MHz for 1080P, and if you cannot find the pll file, just create a new one, here I won’t introduce how to create an IP of pll here**), we can try to drive the display now. There are 9 sub-labs in phase 2.3. Here I only show the code and the result of sub-lab 8 and sub-lab 9.

The pixel information of the pictures that are going to be displayed is stored in the BRAMs using the coe file. So our mission is to find the pixel information we need according to the HDMI timing. In other words, we are going to calculate the address in the BRAM and pass the data that is stored in this location to the ADV7511.

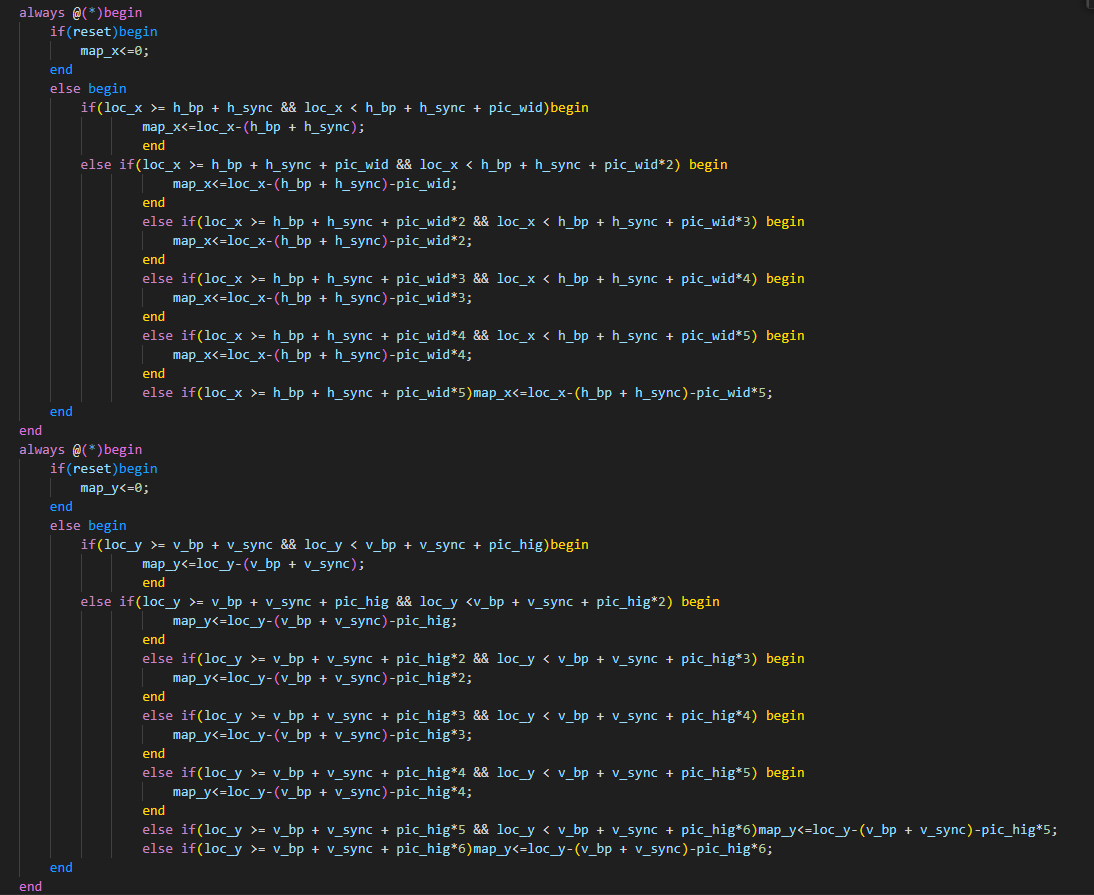
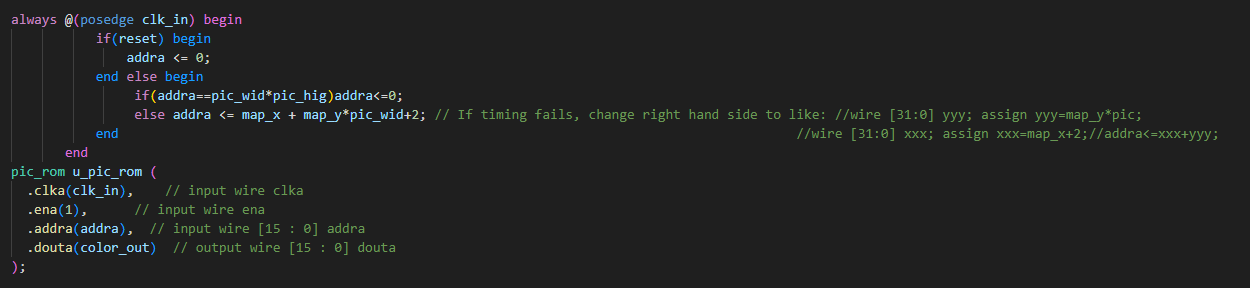
Since the goal is to repeatedly display a 320x175 picture of a little Airbus, the address mapping is as shown in Figure 5.

Figure 5. Location Mapping

In this code, ***loc\_x*** and ***loc\_y*** are the current pixel address on the display according to the ***h\_counter*** and ***v\_counter***. Using the equation in the code in Figure 5, we successfully map it to the x and y values for the BRAM. Then what we need to do is sum them up with them multiplied by the coefficient to get the actual address in BRAM, like in Figure 6. To be noticed, because there is a delay of one or two clock cycles(depends on how you set the BRAM IP core) for BRAM to get the address and output the data, add 2 to the addra to get the data in advance.

Figure 6. Address Calculating

Here are the Figures of the results:

Figure 7. Single Little Airbus

Figure 8. Looping Little Airbuses

**Notice: Results of Dynamic Bricks and Dynamic ball are in directory “./pics”.**

## **Phase 3: AXI\_Lite Slave interface for HDMI controller design**

[[2]](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface#AXI4-Lite)AXI4-Lite is a [subset](https://en.wikipedia.org/wiki/Subset) of the AXI4 protocol, providing a [register-like](https://en.wikipedia.org/wiki/Processor_register) structure with reduced features and complexity. Notable differences are:

1. all bursts are composed by 1 beat only
2. all data accesses use the full data bus width, which can be either 32 or 64 bits

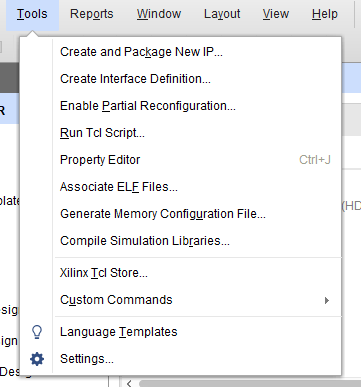
AXI4-Lite removes part of the AXI4 signals but follows the AXI4 specification for the remaining ones. Being a [subset](https://en.wikipedia.org/wiki/Subset) of AXI4, AXI4-Lite transactions are fully compatible with AXI4 devices, permitting the [interoperability](https://en.wikipedia.org/wiki/Interoperability) between AXI4-Lite initiators and AXI4 targets without additional conversion logic.

|  |  |  |
| --- | --- | --- |
| **Signal**[**(1)**](https://docs.amd.com/r/iDBE_pP8gVw3O_auFL6zrQ/LSQAg_LbI9Kgl7s357XzmA?section=XREF_64859_1_This_signal) | **Direction** | **Description** |
| s\_axi\_ctrl\_aclk | Input | Clock signal. All inputs/outputs of this bus interface are rising edge aligned with this clock. |
| s\_axi\_ctrl\_aresetn | Input | Active-Low synchronous reset signal |
| s\_axi\_ctrl\_awalid | Input | Write address valid. This signal indicates that the channel is signaling valid write address. |
| s\_axi\_ctrl\_awready | Output | Write address ready. This signal indicates that the slave is ready to accept an address. |
| s\_axi\_ctrl\_awaddr | Input | Write address. The write address gives the address of the transaction. |
| s\_axi\_ctrl\_wvalid | Input | Write valid. This signal indicates that valid write data are available. |
| s\_axi\_ctrl\_wready | Output | Write ready. This signal indicates that the slave can accept the write data. |
| s\_axi\_ctrl\_wdata | Input | Write data. |
| s\_axi\_ctrl\_bvalid | Output | Write response valid. This signal indicates that the channel is signaling a valid write response. |
| s\_axi\_ctrl\_bready | Input | Write response ready. This signal indicates that the master can accept a write response. |
| s\_axi\_ctrl\_bresp | Output | Write response. This signal indicate the status of the write transaction. |
| s\_axi\_ctrl\_arvalid | Input | Read address valid. This signal indicates that the channel is signaling valid read address. |
| s\_axi\_ctrl\_arready | Output | Read address ready. This signal indicates that the slave is ready to accept an address. |
| s\_axi\_ctrl\_araddr | Input | Read address. The read address gives the address of the transaction. |
| s\_axi\_ctrl\_rvalid | Output | Read valid. This signal indicates that the channel is signaling the required read data. |
| s\_axi\_ctrl\_rready | Input | Read ready. This signal indicates that the master can accept the read data and response information. |
| s\_axi\_ctrl\_rdata | Output | Read data. |
| s\_axi\_ctrl\_rresp | Output | Read response. This signal indicate the status of the read transfer. |
| **Notes:**  1. This signal description is taken from the Arm AMBA Protocol Specification. | | |

Table 2. AXI\_Lite Signals[[3]](https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Lite-Interface-Signals)

After understanding the working mechanism of the AXI protocol, we are now able to design a AXI\_Lite Slave interface for the HDMI controller, and then change what is displayed on the screen by the program running on the CPU.

Firstly, we click **Tools, Create and Package New IP**

Figure 9. Step 1

Then click **Next** on the pop-up window to enter this window. After that, choose

**Create a new AXI4 peripheral**, then click **Next**.

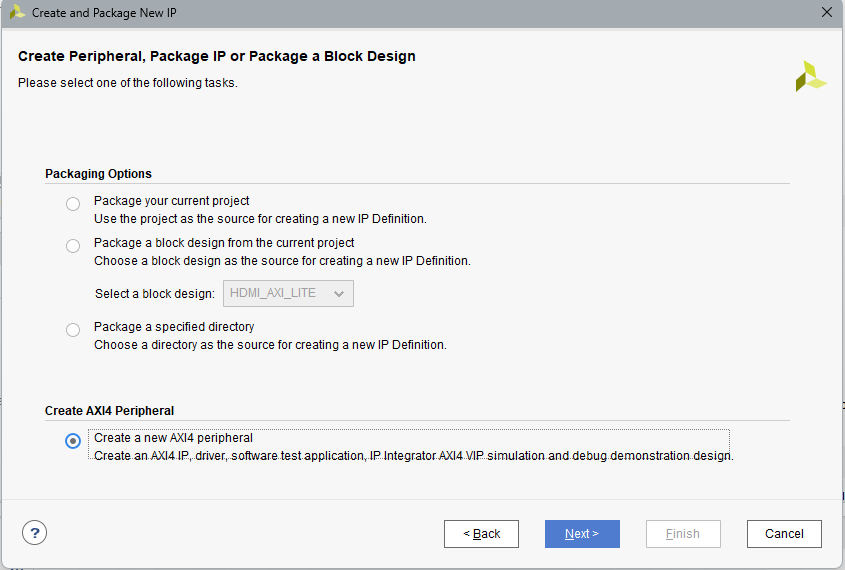


Figure 10. Step 2

After naming the IP you want to create like in Figure 11, we enter the window like Figure 12.

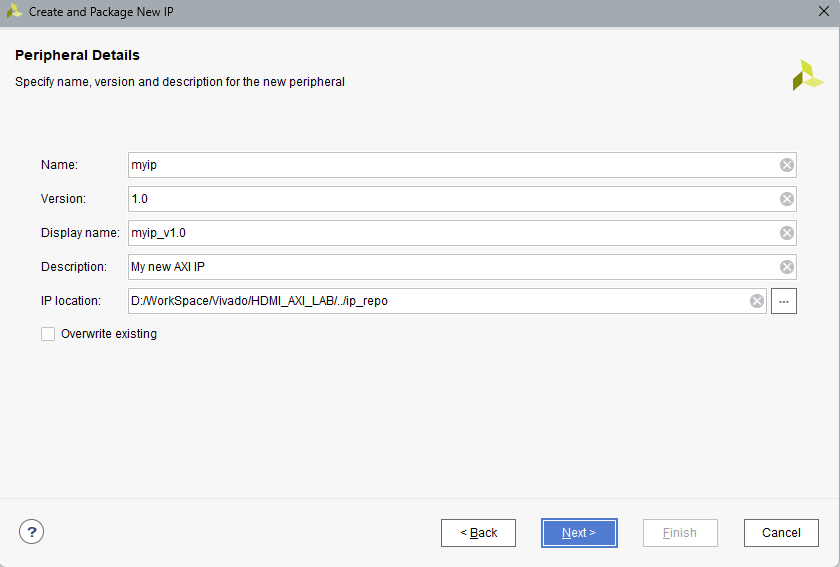


Figure 11. Step 3

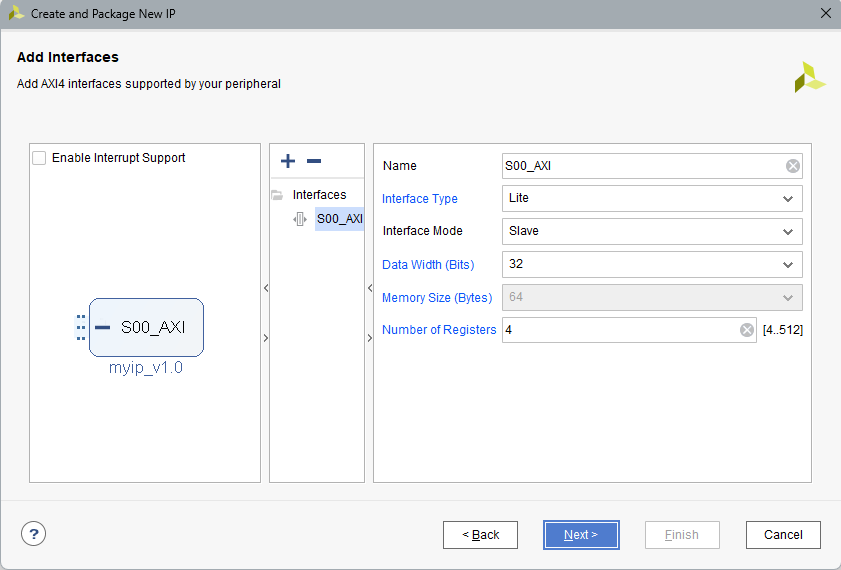


Figure 12. Step 4

In the window like Figure 12, we choose to generate an AXI\_Lite Slave interface. The data width for the AXI Bus is 32 bits, and the number of slave registers is 4 here. After all is set, click **Next** to enter the last step, like Figure 13.

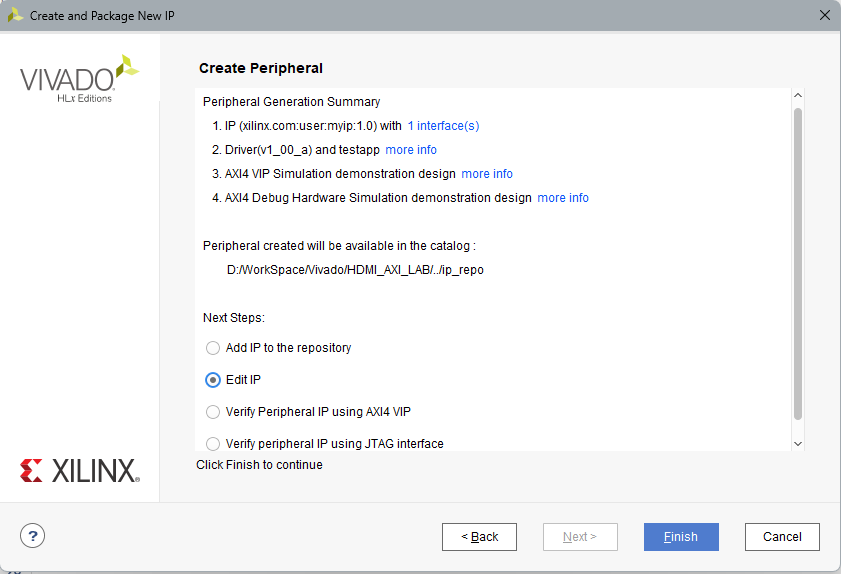


Figure 13. Step 5

After clicking **Finish**, we enter a new project to edit the IP we want to package. Vivado generates two files of the AXI peripheral. We can the add ports to the specified areas like in Figure 14, 15, 16 and 17.

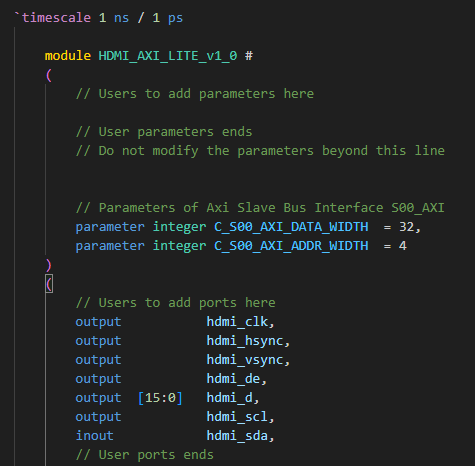


Figure 14. Add User Ports

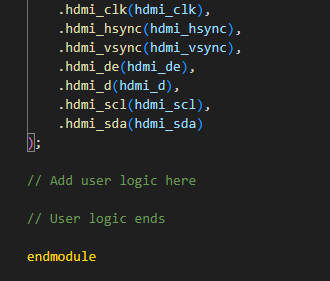


Figure 15. Instantiation of the newly added ports

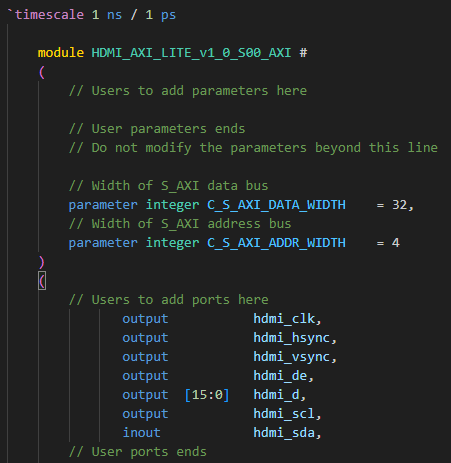


Figure 16. Add User Ports to the Lower Level

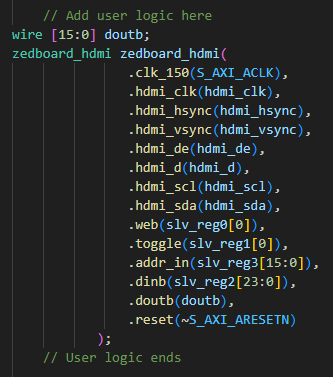


Figure 17. Instantiation of self-defined modules

Besides the global signals like clock signals and reset signals, and the HDMI signals that we’ve introduced before, we add new signals like ***web***, ***toggle***, ***addr\_in***, ***dinb,*** and ***doubt***. The ***web*** is the write enable signal for the BRAM that stores pixel information. BRAM is able to be write when the ***web*** is 1. To be noticed, the slv\_reg0, slv\_reg1, slv\_reg2, and slv\_reg3 are the four slave registers that can be manipulated by the CPU program. Port B (***web,dinb, and addrb or addr\_in in higher level module***) of the BRAM is designed to be manipulated by the CPU(***slv\_reg0[0],slv\_reg2[23:0],slv\_reg3[15:0]***), and ***doubt*** is not used here, so you can just ignore it.

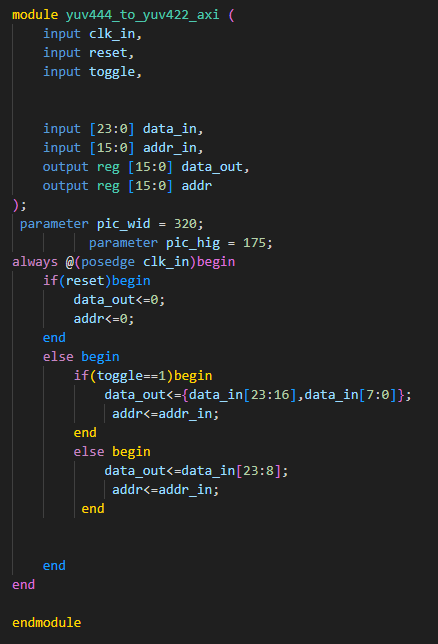
As for the result of Dynamic Bricks and Dynamic ball, they are similar to what is in phase 2, so I will not show them repeatedly here and after. And due to our negligence, we forgot to record the result of “Repeatedly Display many Airbus”. So I will only introduce its code.As required, the format of Airbus.h here is YUV444, but ADV7511 here can only take YUV422. Thus, we need to transfer the format. Considering that transferring format using CPU is not only slow, but also consumes a lot of resources, we decided to design a hardware-accelerating module to transfer the format from YUV444 to YUV422. The Verilog HDL code of this module is like in Figure 18.

Figure 18. YUV444 to YUV422

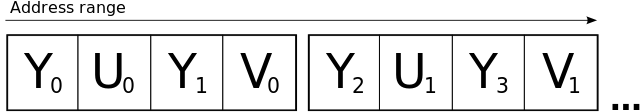
To understand this code, we need to explain these two format fisrt. [[4]](https://en.wikipedia.org/wiki/Y%E2%80%B2UV)Y′UV, also written YUV, is the color model found in the PAL analogue color TV standard. A color is described as a Y′ component (luma) and two chroma components U and V. The prime symbol (') denotes that the luma is calculated from gamma-corrected RGB input and that it is different from true luminance. Today, the term YUV is commonly used in the computer industry to describe color spaces that are encoded using YCbCr. YUV422 is the compressed form of YUV (YUV444) (this is why YUV encoding is used, it is very convenient to compress, the human eyes are sensitive to Y, so Y is not compressed, and the human eyes aare not sensitive to U and V. So, in order to compress the data, U and V will use many methods of sub-sampling), to display a pixel, you need complete Y/U/V information to compress, YUV422 adopts the following method:

Figure 19. YUV422

As shown in Figure 19, there are 8 bytes in total, describing the color information of the 4 pixels.

1. The first point is Y0U0V0;
2. The second point is Y1U0V0;
3. The third point is Y2U1V1;
4. The fourth point is Y3U1V1;

That is, each pixel has its own Y, and two adjacent points share you and V, and half of the U and V information is lost, because the human eye is not sensitive to you and V, many times it has little impact on the display effect.

So, if you want to transfer YUV444 to YUV422, you need a signal ***toggle*** like in the code in Figure 18. When the ***toggle*** is 1, put the top 8 bits and the bottom 8 bits of the input 24-bit YUV444 data together to get the output 16-bit YUV422 data. And when the ***toggle*** is 0, take the top 16 bits of the input 24-bit YUV444 data as the output 16-bit YUV422 data. And the ***toggle*** is manipulated by the program running on the CPU, to make sure the pass of the toggle signal, address signal, and data signal are synchronous.

The program running on the CPU is written in the C language. As shown in Figure 20, we define the base address of our AXI\_Lite Slave IP, along with four slave registers’ offsides.

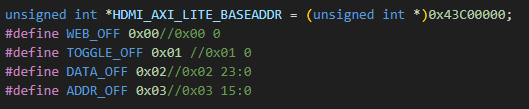


Figure 20. Address Definition

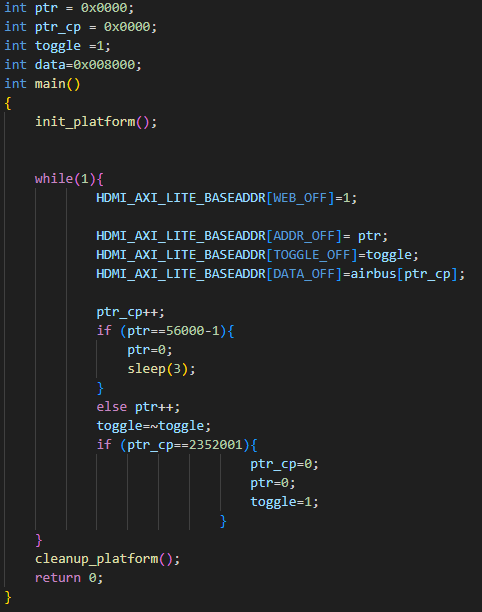
And in Figure 21, we present to you the main function of this program. The program will continuously send data and address from the array ***airbus***, which stores the pixel information of the pictures, to the slave registers, along with the toggle signal. The result is that many different airbus pictures will be displayed in order and repeatedly. Each picture will be displayed for 3 seconds each time.

Figure 21. C source code

## **Phase 4: AXI\_Full Master interface for HDMI controller design**

After achieving the goal of creating an AXI\_Lite Slave interface for the HDMI controller, we are going to create an AXI\_Full Master interface for the HDMI controller. Because we want to display 1920x1080 pictures on the screen, and the AXI\_Lite Slave interface does not have enough capabilities to do it, so we have to use more powerful AXI\_Full Bus. AXI\_Full is the full functional version of Advanced eXtensible Interface, it featured with its handshake mechanism, independent read and write channels, and burst-based protocol.

First thing is about the handshake mechanism. [[5]](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface) AXI defines a basic handshake mechanism, composed by an **xVALID** and **xREADY** signal. The xVALID signal is driven by the source to inform the destination entity that the payload on the channel is valid and can be read from that clock cycle onwards. Similarly, the xREADY signal is driven by the receiving entity to notify that it is prepared to receive data. When both the xVALID and xREADY signals are high in the same clock cycle, the data payload is considered transferred and the source can either provide a new data payload, by keeping high xVALID, or terminate the transmission, by de-asserting xVALID. An individual data transfer, so a clock cycle when both xVALID and xREADY are high, is called "beat". Two main rules are defined for the control of these signals:

(1). A source must not wait for a high xREADY to assert xVALID.

(2). Once xVALID is asserted, a source must maintain the assertion until a handshake occurs.

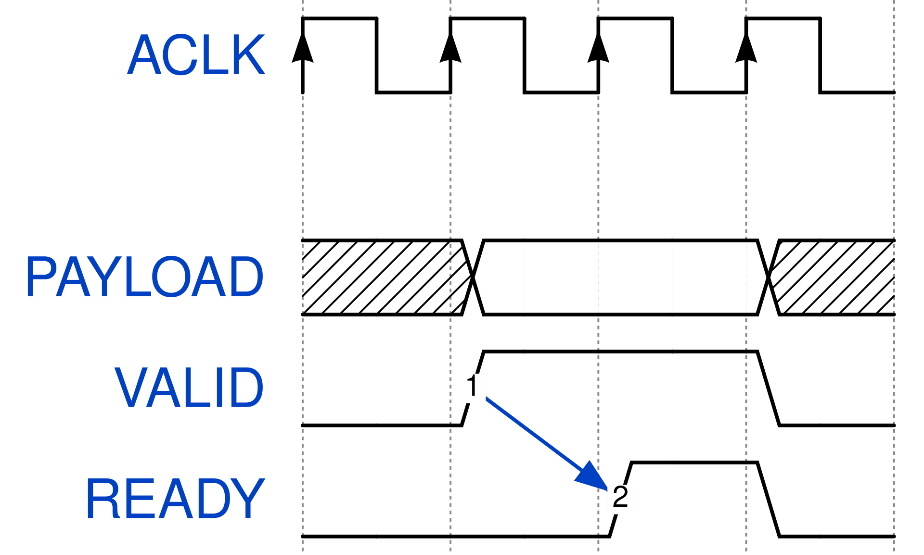
Thanks to this handshake mechanism, both the source and the destination can control the flow of data, throttling the speed if needed.

Figure 22. Handshake Mechanism

What comes next is the independent read and write channels. In the AXI specification, five channels are described:

1. Read Address channel (AR)
2. Read Data channel (R)
3. Write Address channel (AW)
4. Write Data channel (W)
5. Write Response channel (B)

Other than some basic ordering rules, each channel is independent from each other and has its own couple of xVALID/xREADY handshake signals.[[10]](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface#cite_note-10)

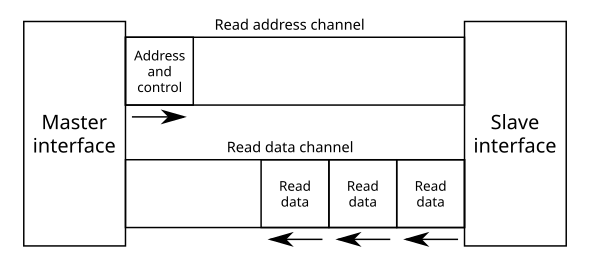
[](https://en.wikipedia.org/wiki/File:AXI_read_channels.svg)

Figure 23. AXI Read Address and Read Data channels

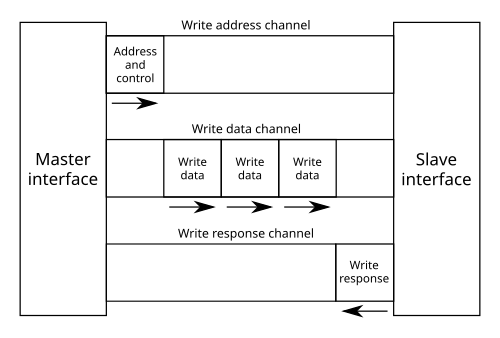
[](https://en.wikipedia.org/wiki/File:AXI_write_channels.svg)

Figure 24. AXI Write Address, Write Data and Write Response channels

The most important thing is the burst-based protocol, which is the key to enhancing the transportation bandwidth. Burst-based protocol, meaning that there may be multiple data transfers (or beats) for a single request. This makes it useful in the cases where it is necessary to transfer large amount of data from or to a specific pattern of addresses. In AXI, bursts can be of three types, selected by the signals ARBURST (for reads) or AWBURST (for writes):

1. FIXED
2. INCR
3. WRAP

In FIXED bursts, each beat within the transfer has the same address. This is useful for repeated access at the same memory location, such as when reading or writing a FIFO.



In INCR bursts, on the other hand, each beat has an address equal to the previous one plus the transfer size. This burst type is commonly used to read or write sequential memory areas.



WRAP bursts are similar to the INCR ones, as each transfer has an address equal to the previous one plus the transfer size. However, with WRAP bursts, if the address of the current beat reaches the "Higher Address boundary", it is reset to the "Wrap boundary":



with

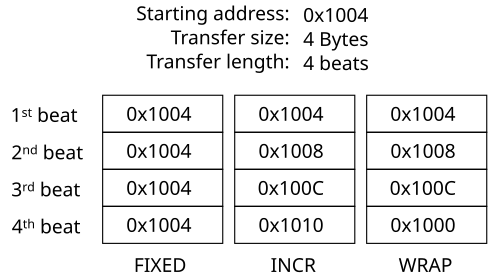


Figure 25. 3 different types of burst transportation

Now we’ve learned the basic knowledge of AXI\_Full, so we can dive into the Verilog HDL code immediately. The steps of creating an AXI\_full master interface are the same as creating an AXI\_Lite Slave interface in phase 3, with the only difference being that we need to choose **Full** and **Master** instead of Lite and Slave.

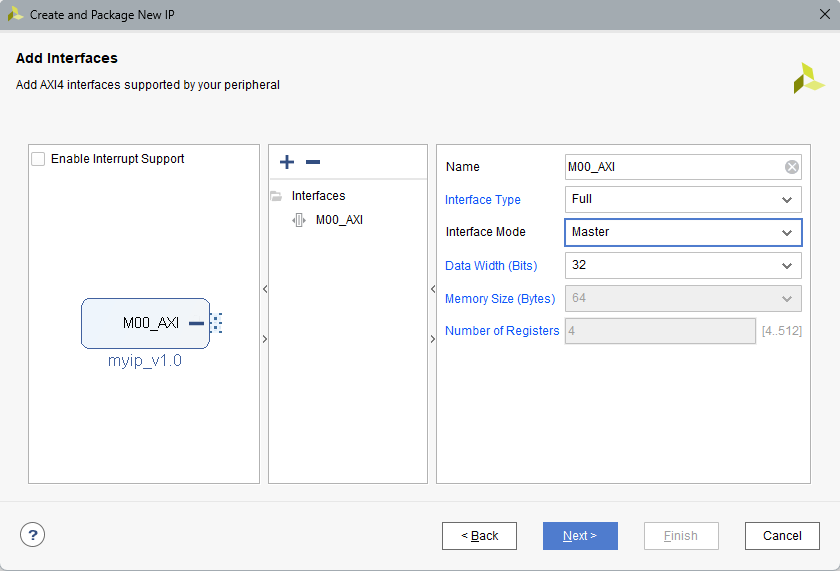


Figure 26. Creating AXI\_Full Master Interface

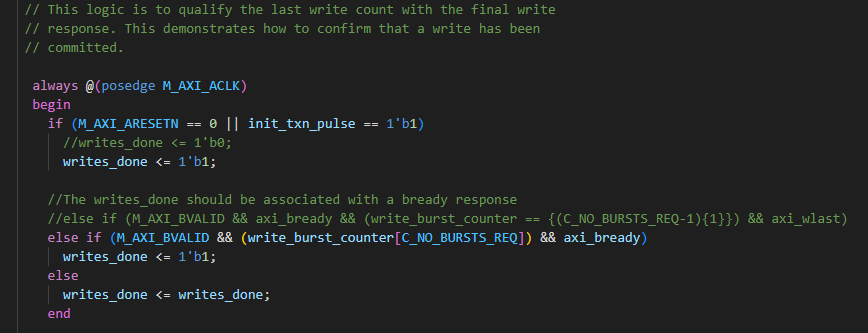
Then Vivado will generate Verilog HDL codes of AXI\_Full Master interface. Since in this phase we only need to use the read channel of the AXI bus, we need to make some changes to the generated codes. Firstly, as shown in Figure 27, we make the signal ***writes\_done*** always be 1 to skip the INIT\_WRITE of the finite state machine(FSM).

Figure 27. Writes\_done is set to 1

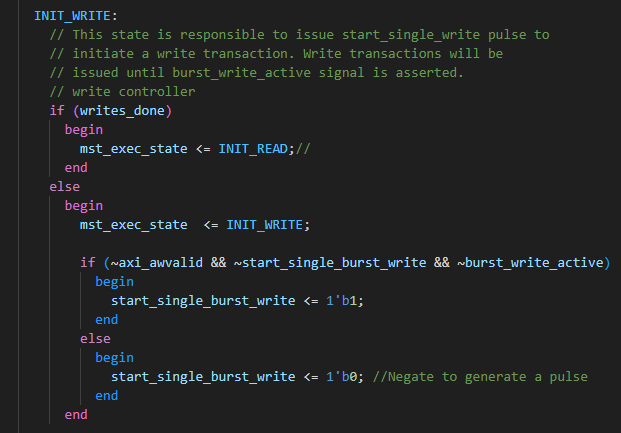


Figure 28. Skip INIT\_WRITE and enter INIT\_READ

After this setting, the AXI Bus can now enter burst transportation repeatedly when the CPU is ready. But when should and how can the burst transportation end? Now we are going to break down the mechanism of the burst transportation. As shown in the next figures, we find some parameters that are vital.



Figure 29. Read Base Address

The first thing is the ***C\_M\_TARGET\_SLAVE\_BASE\_ADDR***. Here, we set it to 0x10000000, which represents a specific part in the DDR. To be noticed, the default storage place of everything generated by compiling the C program is in the DDR, from its base address of 0x00000000. So I set the base address of the picture information as 0x10000000 can avoid conflict between these two parts.

Then comes the ***burst\_size\_bytes***. It helps axi\_araddr to tell the slave target where to read when each time the burst transportation begins. Here, I used the default value 16 for ***C\_M\_AXI\_BURST\_LEN***, but it failed to meet our requirement. To be specific, it took the time to show two lines of pixels on the screen to read one line of pixels from the DDR. Then I changed it to the value of 64, and the issue was solved.



Figure 30. burst\_size\_bytes

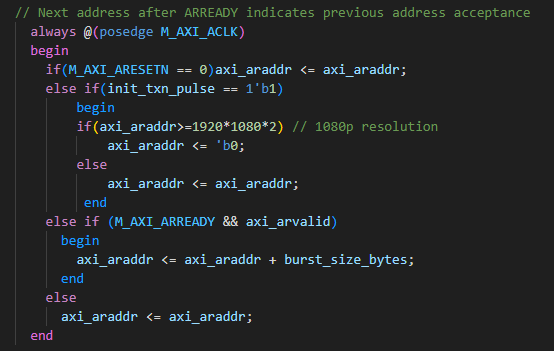


Figure 31. axi\_araddr

Figure 31 shows the logic of ***axi\_araddr***. This is the signal that represents the offside from the base address. Because we use a cacheline as a buffer(we will talk about this cache line later), we need to read one line per time until we read to the last line last pixel, then we go to the start point. So, we need to keep the address of the last read to get the correct address for the next time until axi\_addr = 1920x1080x2. Actually, it is better written in 960x4x1080, because the data width we read is 32 bits, which is 4 bytes, and the address in the DDR increases by 4 each time. Since 32-bit data includes 2 pixels’ information, it needs to read 960 times to get one line of pixels from the DDR.

Now we can see that 64 bytes of data will be read during one burst, so we need 15 bursts to read one line to fill the cacheline. As shown in Figures 32and 33, we change some conditions to make it stop after continuously bursting 15 times. The signal ***read\_burst\_counter*** increases by 1 at the end of each burst until the 15th burst ends. With this condition, ***reads\_done*** will be 1, which represents the end of the transportation and waiting for the next sign of transportation starts, when ***read\_burst\_counter*** reaches 15.

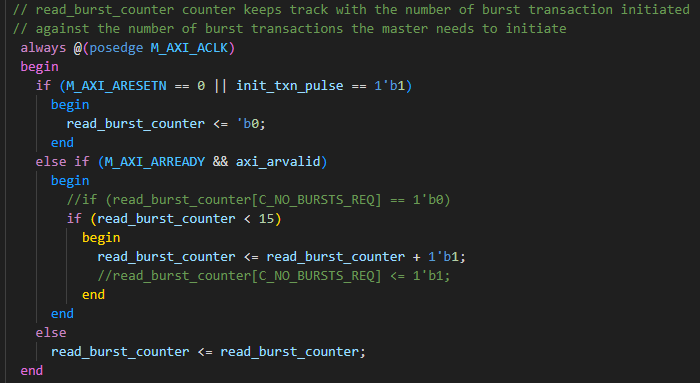
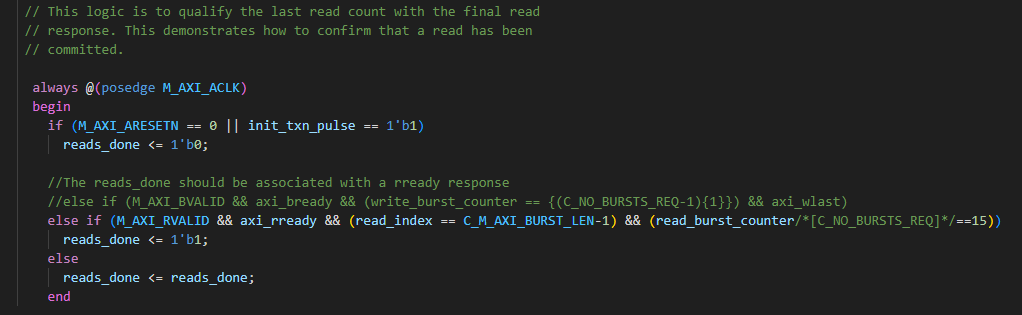
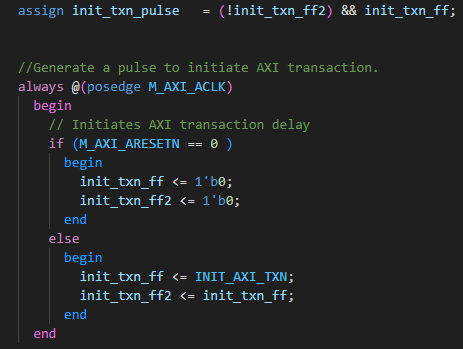
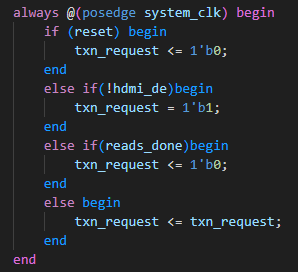


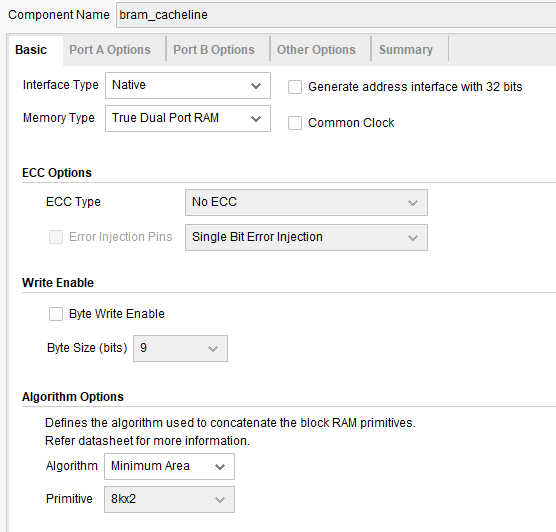
Figure 32. read\_burst\_counter

Figure 33. reads\_done

The next problem is when to start reading. The signal ***init\_txn\_pulse*** represents the start of transportation, and it is generated by **INIT\_AXI\_TXN** as shown in Figure 34, which means that all I need to do is change INIT\_AXI\_TXN from 0 to 1 to start transportation. So I create a signal called ***txn\_request*** to manipulate INIT\_AXI\_TXN, as shown in Figure 35. As soon as one line of pixels is displayed on the screen, txn\_request is set to 1 to start a transportation, and as soon as one line of pixels is stored in the cache line, txn\_request is set to 0 to be ready for the next time of transportation.

Figure 34. init\_txn\_pulse

Figure 35. txn\_request

The next thing we care about is the cacheline. The cache line is a dual-port BRAM. One side of this BRAM accepts data from the DDR, and the other side of the ports drives the HDMI controller. The size of this cacheline is 32-bit data width \* 960 depth. The settings of the cacheline are shown in the next few figures.

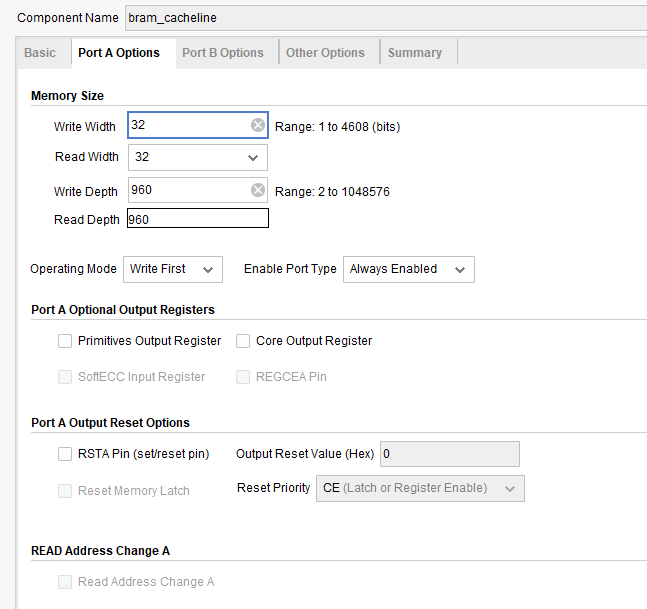
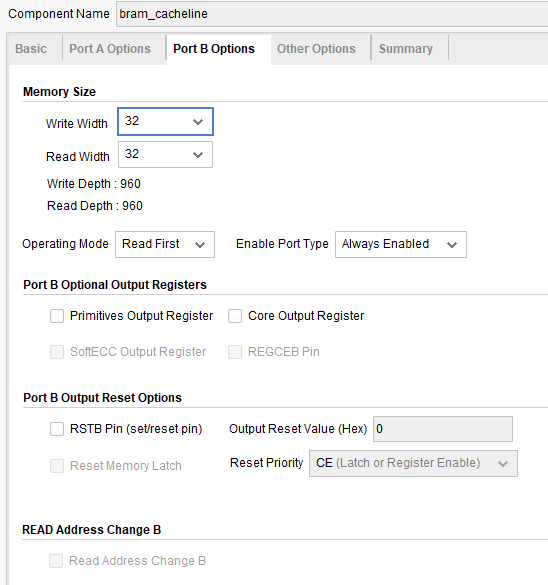
Figure 36. cacheline settings 1

Figure 37. cacheline settings 2

Figure 38. cacheline settings 3

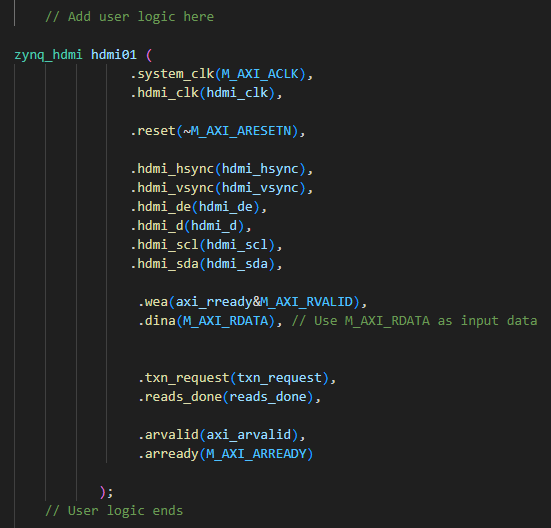
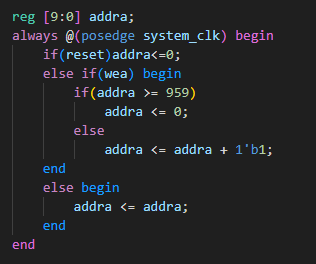
And how can we make sure the address and the data are synchronous, or in other words, how can we make sure the data is stored in the right place?

Figure 39. wea

As shown in Figure 39, we use two signals, ***axi\_rready*** and ***M\_AXI\_RVALID,*** to make sure that only when the data is prepared, then can it be written into the cacheline. Also, only when wea is 1 can addra move to the next position, as shown in Figure 40.

Figure 40. addra

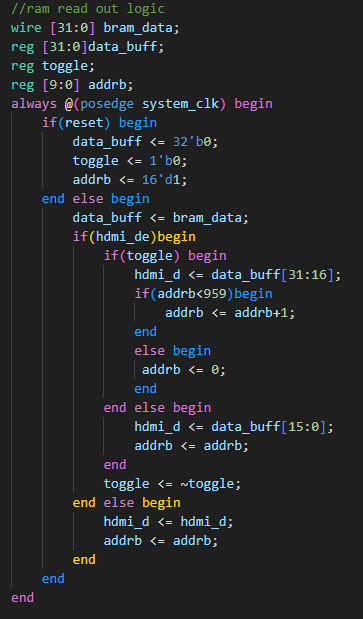
Now we have successfully designed the write port of the cacheline, so we can move on to design the read port of the cache line. Since the data width is 32 bits, which means we will read out 2 pixels at a time, we need a register to to hold the data for two clock cycles, during which we can split it into two 16-bit data and output them in oder. The code is shown in Figure 41.

Figure 41. Cacheline Data Output

Figure 42. C Source Code for 1080P Airbus

After the circuit is designed, we can now focus on the C program. The pixel information is stored in the array ***airbus[]***, and the compiled result is stored in the DDR, so we can use the function ***Xil\_MemCpy()*** to copy the entire frame of the picture to the specific place for the HDMI controller to read. Notice, we need to use ***Xil\_DCacheFlush()*** to clean the CPU cache when memory copy ends. Otherwise, the bottom half of the picture on the display will appear with tear-like horizontal stripes.

The result is “./Pics/ 2\_1080\_Airbus\_Loop”, along with the other two required results “./Pics/ Dynamic ball” and “./Pics/ Dynamic Bricks 2”.

## **Phase 5: Real Time Camera Input to HDMI Output**

In this section, we will use the AXI\_Full Master interface to design a module that can be used to drive the camera module, receive data flow from the camera module, decode the data flow and transfer data format, and write data to the DDR. Then, with the help of the HDMI controller with AXI\_Full Master interface that we have designed in phase 4, we can display what the camera captures on the screen in real time.

So the first thing is to drive the camera. We need two GIPO pins to be manipulated by software, which makes us able to communicate with the camera under IIC protocol. The camera we use in this lab is IMX219.

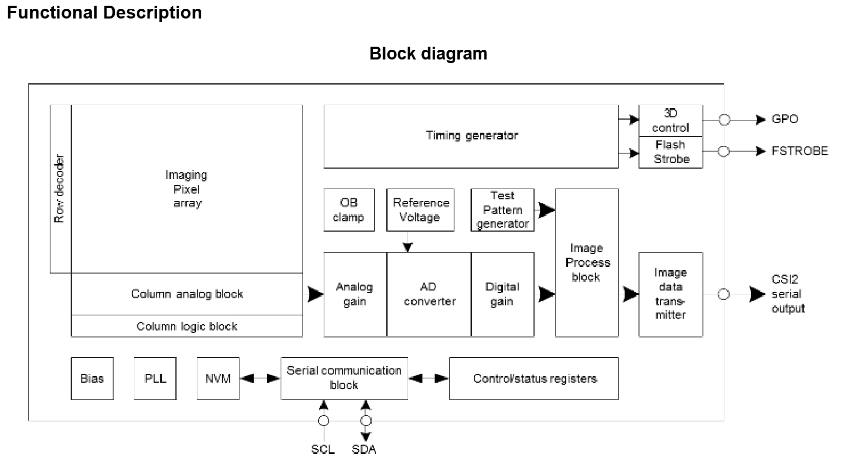


Figure 43. IMX219 Block Diagram

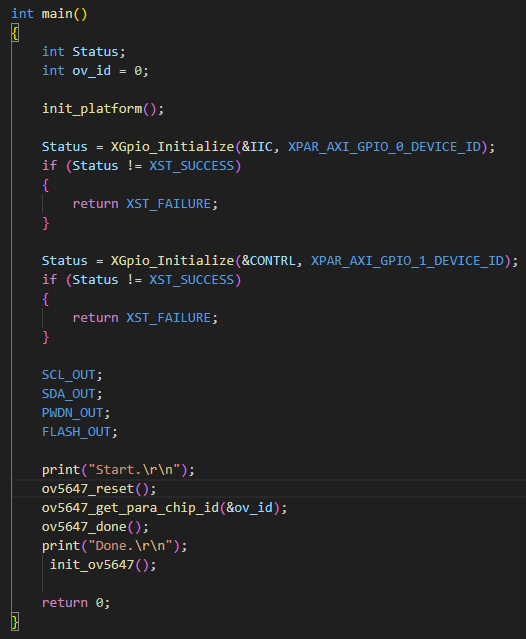
The introduction to the IMX219 and Raspberry Pi Camera V2.1 is stored in “./docs”. We will directly dive into the C source code. We drive the Camera using the **SCCB** protocol.

Figure 44. C Source Code for SCCB

So I will introduce the SCCB protocol here. [[6]](docs/SCCBSpec_AN.pdf)OmniVision Technologies, Inc. has defined and deployed the Serial Camera Control Bus (SCCB),a 3-wire serial bus, for control of most of the functions in OmniVision's family of CAMERACHIPT sensors. In reduced pin package parts, the SCCB operates in a modified 2-wire serial mode. OmniVision CAMERACHIP sensors will only operate as slave devices and the companion back-end interface must assert as the master. One SCCB master device can be connected to the SCCB to control at least one SCCB slave device. An optional suspend-control signal provides the capability for the SCCB master device to power down the SCCB system.

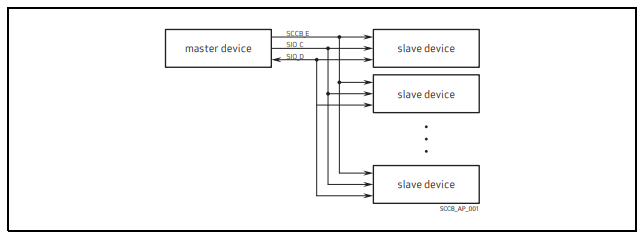


Figure 45. SCCB Functional Block Diagram

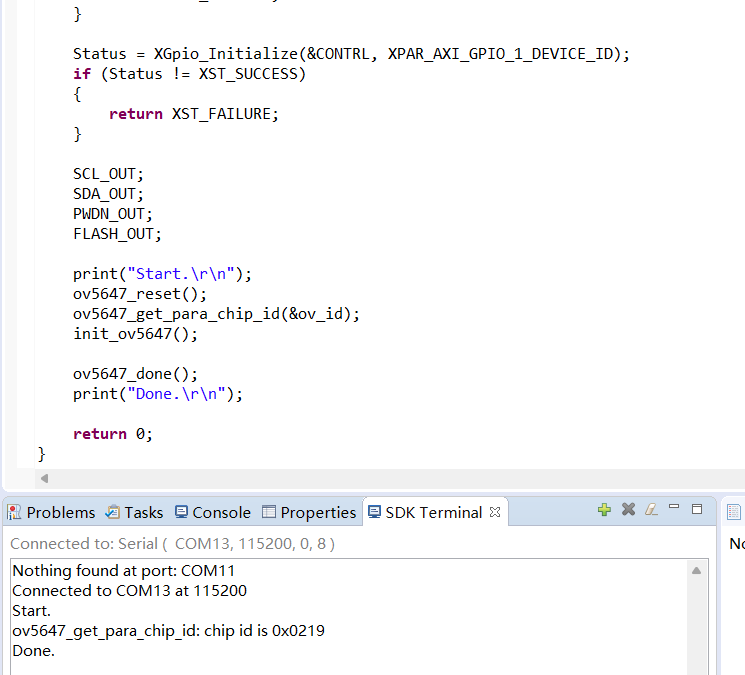
The 3 essential pins are SDA, SCL, and PWDN. We package the process of manipulating these pins to drive the camera into function and we directly use these functions in the main function, as shown in Figure 44. More detailed information about the packaged functions, mostly the register-level operations, can be seen in the C Source file. And more information about the SCCB protocol can be seen in the document in the directory “./docs”. Anyway, if you drive the camera successfully, you can see it returns its camera ID, as shown in Figure 46. (Notice: OV5647 is not the camera we use.)

Figure 46. Camera ID

The next step is to receive the data flow and decode the data flow. This refers to another protocol called the **MIPI** protocol, and an interface called **DSI**. [[7]](https://en.wikipedia.org/wiki/Display_Serial_Interface) The **Display Serial Interface** (DSI) is a specification by the **Mobile Industry Processor Interface** (MIPI) Alliance aimed at reducing the cost of display controllers in a mobile device. It is commonly targeted at LCD and similar display technologies. It defines a serial bus and a communication protocol between the host, the source of the image data, and the device, which is the destination. The interface is closed source, which means that the specification of the interface is not open to the public. The maintenance of the interface is the responsibility of the MIPI Alliance. Only legal entities (e.g., companies) can be members. These members or the persons commissioned and approved by them have access to the specification in order to use it in their possible applications.

At the physical layer, DSI specifies a high-speed (e.g., 4.5 Gbit/s/lane for D-PHY 2.0[3]) differential signaling point-to-point serial bus. This bus includes one high speed clock lane and one or more data lanes. Each lane is carried on two wires (due to differential signaling). All lanes travel from the DSI host to the DSI device, except for the first data lane (lane 0), which is capable of a bus turnaround (BTA) operation that allows it to reverse transmission direction. When more than one lane is used, they are used in parallel to transmit data, with each sequential byte in the stream traveling on the next lane. That is, if 4 lanes are being used, 4 bytes are transmitted simultaneously, one on each lane. The link operates in either low power (LP) mode or high speed (HS) mode. In low power mode, the high-speed clock is disabled, and signal clocking information is embedded in the data. In this mode, the data rate is insufficient to drive a display, but is usable for sending configuration information and commands. High speed mode enables the high-speed clock (at frequencies from tens of megahertz to over one gigahertz) that acts as the bit clock for the data lanes. Clock speeds vary by the requirements of the display. High speed mode is still designed to reduce power usage due to its low voltage signaling and parallel transfer ability.

The communication protocol describes two sets of instructions. The Display Command Set (DCS) is a set of common commands for controlling the display device, and their format is specified by the DSI standard. It defines registers that can be addressed and what their operation is. It includes basic commands such as sleep, enable, and invert display. The Manufacturer Command Set (MCS) is a device-specific command space whose definition is up to the device manufacturer. It often includes commands required to program non-volatile memory, set specific device registers (such as gamma correction), or perform other actions not described in the DSI standard. The packet format of both sets is specified by the DSI standard. There are Short and Long Packets. Short Packet is 4 bytes long; Long Packet can be of any length up to 216 bytes. Packets are composed of a DataID, word count, error correction code (ECC), payload, and checksum (CRC). Commands that require reading data back from the device trigger a BTA event, which allows the device to reply with the requested data. A device cannot initiate a transfer; it can only reply to host requests.

Image data on the bus is interleaved with signals for horizontal and vertical blanking intervals (porches). The data is drawn to displayed in real time and not stored by the device. This allows the manufacture of simpler display devices without frame buffer memory. However, it also means that the device must be continuously refreshed (at a rate such as 30 or 60 frames per second) or it will lose the image. Image data is only sent in HS mode. When in HS mode, commands are transmitted during the vertical blanking interval.

More detailed information about the MIPI and DSI can be seen in the documents ***MIPI Alliance Specification*** ***for D-PHY*** and ***MIPI Alliance Specification for CSI-2*** under the directory “./docs”.

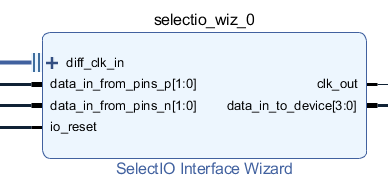
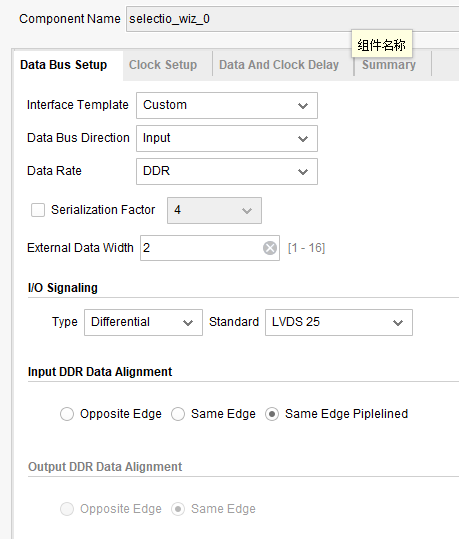
Anyway, we use a finite state machine (FSM) to capture and restore the valid data from the data flow. Firstly, we use an IP called SelectIO Interface Wizard to transfer differential signals into serial signals. Then we enter the design of mipi interface. The 2 key parts are the shift-register buffers and the FSM.

Figure 47. SelectIO Interface Wizard



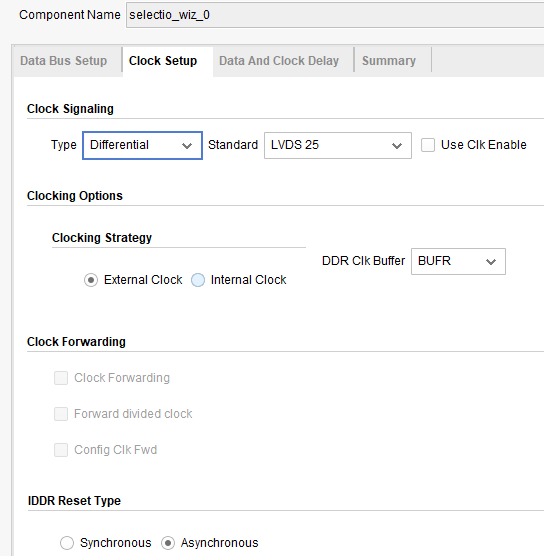
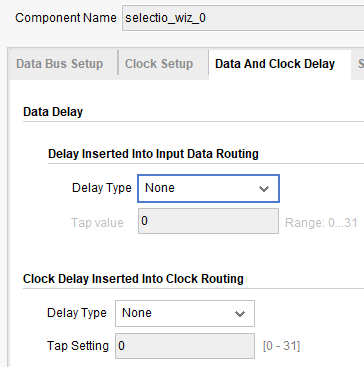
Figure 48. SelectIO Interface Wizard Setting 1

Figure 49. SelectIO Interface Wizard Setting 2

Figure 50. SelectIO Interface Wizard Setting 3

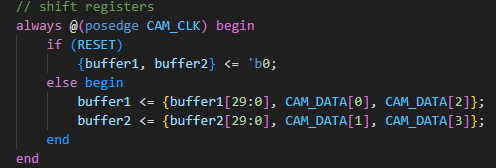
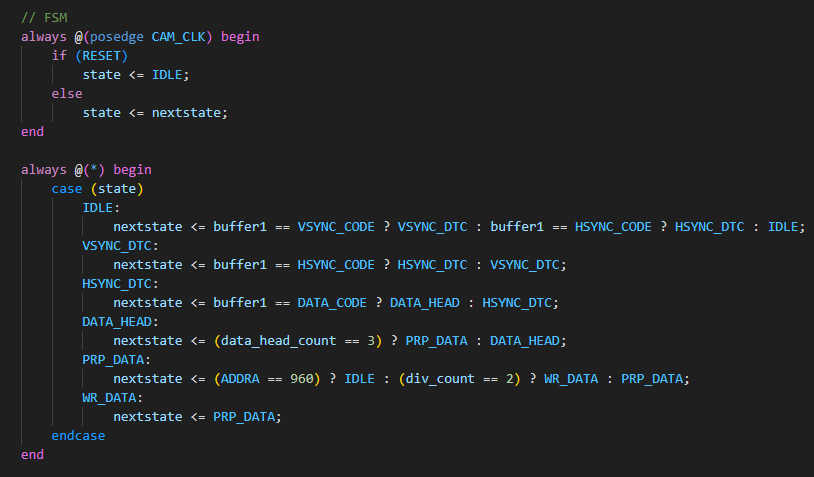
The module uses two 32-bit shift registers, buffer1 and buffer2, to buffer incoming MIPI data for processing. These registers collect and store the 4-bit input data to form 32-bit words that can be compared against MIPI protocol codes (e.g., VSYNC\_CODE, HSYNC\_CODE, DATA\_CODE) or used to extract pixel data. The input data (CAM\_DATA\_i) is inverted (~CAM\_DATA\_i) to form ***CAM\_DATA***, because the PCB board is wrongly designed. On every positive edge of ***CAM\_CLK***, If ***RESET*** is active, both buffer1 and buffer2 are cleared to zero. Otherwise, buffer1 shifts in bits 0 and 2 of CAM\_DATA (2 bits per clock cycle) by concatenating its current contents (***buffer1[29:0]***) with ***CAM\_DATA[0]*** and ***CAM\_DATA[2]***. buffer2 shifts in bits 1 and 3 of CAM\_DATA by concatenating its current contents (***buffer2[29:0]***) with ***CAM\_DATA[1]*** and ***CAM\_DATA[3]***. This results in buffer1 and buffer2 accumulating data in parallel, with each buffer handling two bits of the 4-bit input per cycle, effectively serializing the data into 32-bit chunks.

Figure 51. Shift-register buffers

The FSM manages the parsing of MIPI data by transitioning between states based on the contents of buffer1 and internal counters. It consists of two always blocks: one for updating the current state and one for determining the next state.

1. **State Definitions**:
   1. **IDLE**: Waits for synchronization codes.
   2. **VSYNC\_DTC**: Detects vertical sync.
   3. **HSYNC\_DTC**: Detects horizontal sync.
   4. **DATA\_HEAD**: Processes the data header.
   5. **PRP\_DATA**: Prepares pixel data.
   6. **WR\_DATA**: Writes pixel data to the output.
2. **Next-State Logic**:
   1. **IDLE**:
      1. Transitions to VSYNC\_DTC if buffer1 matches VSYNC\_CODE (32'h00001D00).
      2. Transitions to HSYNC\_DTC if buffer1 matches HSYNC\_CODE (32'h00001D40).
      3. Otherwise, remains in IDLE.
   2. **VSYNC\_DTC**:
      1. Moves to HSYNC\_DTC if buffer1 matches HSYNC\_CODE; otherwise, stays in VSYNC\_DTC.
   3. **HSYNC\_DTC**:
      1. Advances to DATA\_HEAD if buffer1 matches DATA\_CODE (32'h00001D54); otherwise, stays in HSYNC\_DTC.
   4. **DATA\_HEAD**:
      1. Transitions to PRP\_DATA when data\_head\_count reaches 3; otherwise, remains in DATA\_HEAD.
   5. **PRP\_DATA**:
      1. Moves to WR\_DATA when div\_count reaches 2.
      2. Returns to IDLE if ADDRA reaches 960 (end of line).
      3. Otherwise, stays in PRP\_DATA.
   6. **WR\_DATA**:
      1. Returns to PRP\_DATA after writing data.

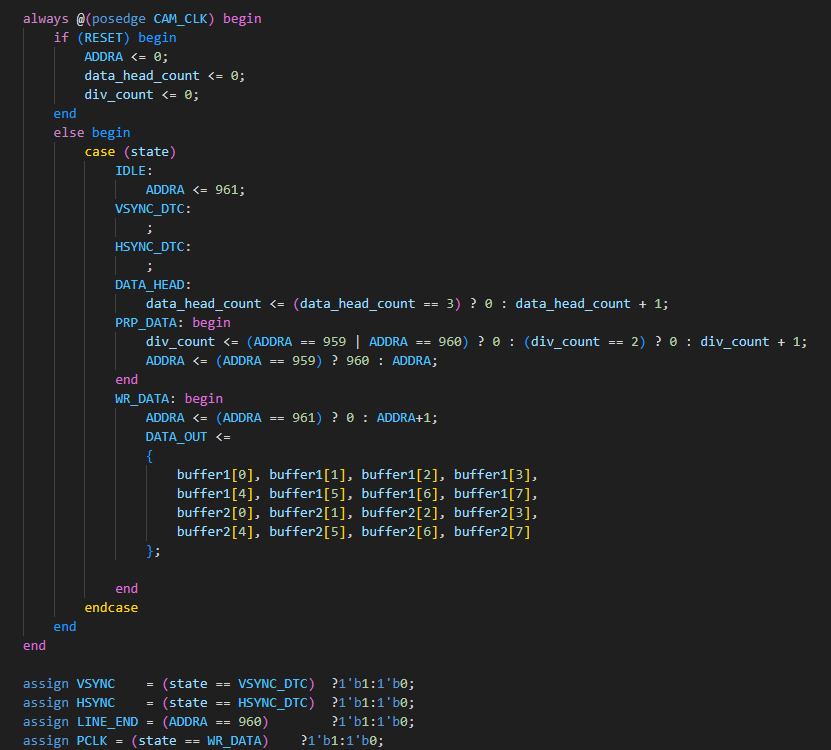
Figure 52. State Change Logic

Figure 53. Operations of Each State

The FSM uses buffer1 to detect synchronization codes and relies on counters (***data\_head\_count*** and ***div\_count***) to manage data processing. The buffers and FSM work together to ensure accurate parsing of MIPI data by aligning state transitions with the buffered data content.

However if you use **ILA** to capture the waveform of the ***next\_state***, you may find that the FSM stucks at stae HSYNC or YSYNC. This maybe is because the lack od constraint to the external clock CAM\_CLK. So you can try to add a clock constraint in the xdc file, like:

**create\_clock -period 7.000 -name cam\_clk -waveform {0.000 3.500} [get\_pins path\_to\_your\_pin/cam\_clk]**

**set\_clock\_groups -name async\_group -asyncronous -group [get\_clocks cam\_clk] -group [get\_clocks clk\_fpga\_0]**

Unfortunately, we left the lab many days early for personal reasons, so we didn't have time to verify if it works.

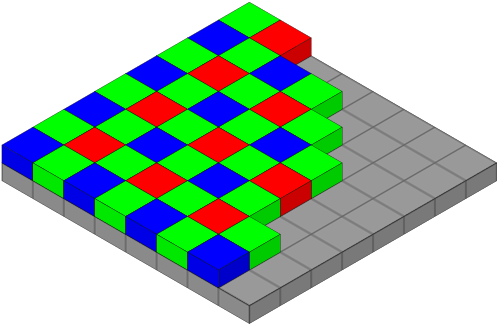
What comes next is format transferring. The data parsed from the dataflow is raw data, or **Bayer** format, to be specific. [[8]](https://en.wikipedia.org/wiki/Bayer_filter) A Bayer filter mosaic is a color filter array (CFA) for arranging RGB color filters on a square grid of photosensors. Its particular arrangement of color filters is used in most single-chip digital image sensors used in digital cameras, and camcorders to create a color image. The filter pattern is half green, one quarter red and one quarter blue, hence is also called BGGR, RGBG, GRBG, or RGGB.

Figure 54. Bayer Filter

Thus, the process of transferring data from Bayer to RGB is called **demosaicing**. [[9]](https://en.wikipedia.org/wiki/Bayer_filter#Demosaicing) Demosaicing can be performed in different ways. Simple methods interpolate the color value of the pixels of the same color in the neighborhood. For example, once the chip has been exposed to an image, each pixel can be read. A pixel with a green filter provides an exact measurement of the green component. The red and blue components for this pixel are obtained from the neighbors. For a green pixel, two red neighbors can be interpolated to yield the red value, also two blue pixels can be interpolated to yield the blue value.

This simple approach works well in areas with constant color or smooth gradients, but it can cause artifacts such as color bleeding in areas where there are abrupt changes in color or brightness especially noticeable along sharp edges in the image. Because of this, other demosaicing methods attempt to identify high-contrast edges and only interpolate along these edges, but not across them.

Other algorithms are based on the assumption that the color of an area in the image is relatively constant even under changing light conditions, so that the color channels are highly correlated with each other. Therefore, the green channel is interpolated at first then the red and afterwards the blue channel, so that the color ratio red-green respective blue-green are constant. There are other methods that make different assumptions about the image content and starting from this attempt to calculate the missing color values.

There are many ways to do the demosaicing. The code provided by the lab uses the simplest way, as shown in Figure 55. For each pixel, its missing color component is taken directly from the corresponding color component of the nearest point.

Figure 55. Simple Demosaic

After demosaicing, we get the RGB888 formatted data. The next thing is to transfer it to YUV444 format and then YUV422 format. The equation we use for RGB888 to YUV444 is :



And YUV444 to YUV422 has already been introduced before.

The Verilog HDL code that implements these transformations uses pure combinational logic. The advantage of this design is easy to implement and has no clock cycle latencies. But pure combinational logic implementation may cause timing violations with high operation frequency, especially with many multi-clock-cycle operations like multiply. So, if the timing violation happens, you should add pipeline registers to shorten the logic path, but remember to adjust the timing correspondence, like Figure 57.

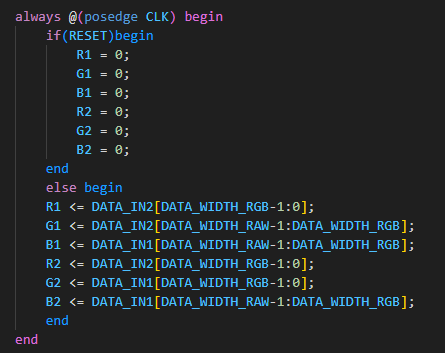


Figure 56. Bayer to RGB888



Figure 57. RGB888 to YUV444 (1)

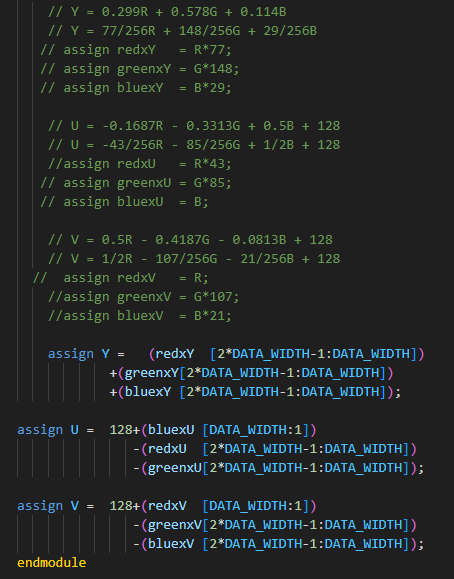


Figure 58. RGB888 to YUV444 (2)

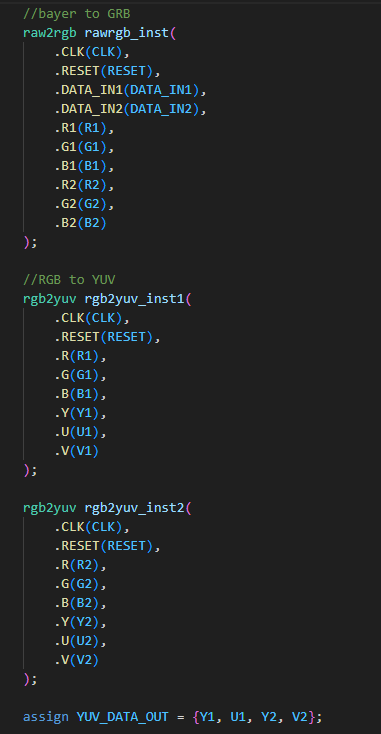


Figure 59. YUV444 to YUV422

Besides the code provided by the lab, I searched on the web and organized some files of better demosaicing and RGB888 to YUV444. You can find them at “./codes/Phase5/ Better\_Bayer\_to\_YUV444”. To be noticed, these code are independent and not been instantiated, so if you want to use them, you need to instantiate them by yourself. Also, in this step, you can do more jobs like Gamma Correction and etc.

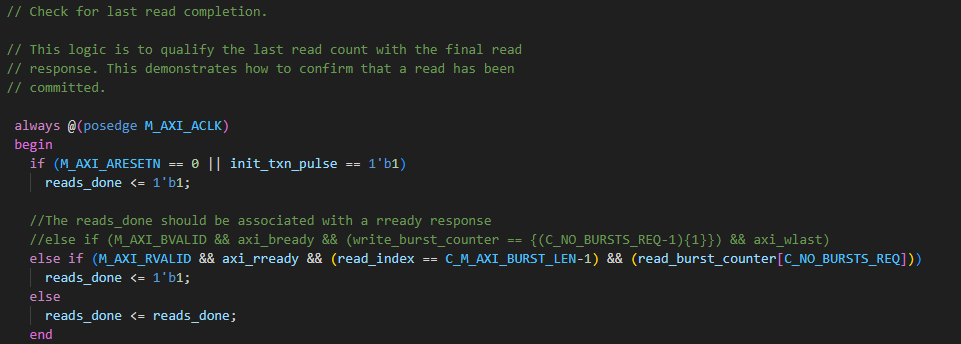
The last thing is to write data to the DDR using AXI\_Full Master interface. What we will do is similar to what we have done in phase 4, only this time we are dealing with the write channel. So firstly, set signal ***reads\_done*** always be 1 to shut down the read channel.

Figure 60. Shut Down the Read Channel

Then set the slave target address, burst length, awaddr and etc. These are all shown in the next few figures.



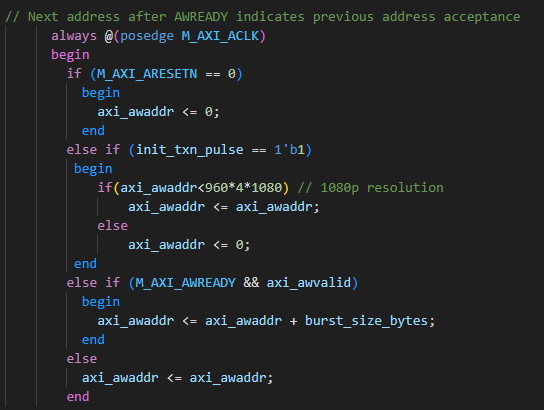
Figure 61. Set the Slave Tareget Address

Figure 62. Set axi\_awaddr

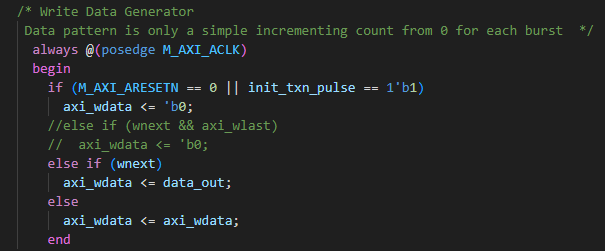


Figure 63. Set Signal data\_out as Output Data

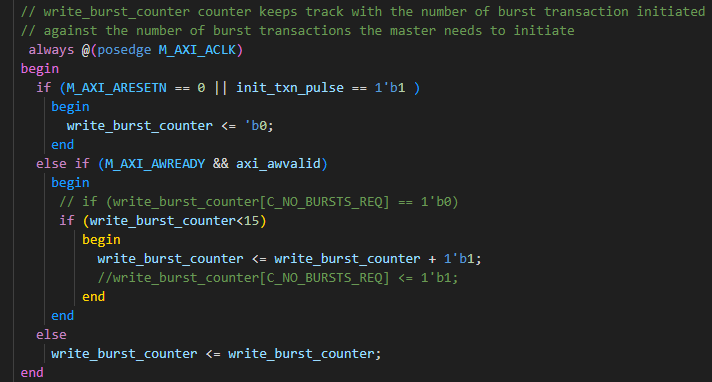


Figure 64. Set write\_burst\_counter

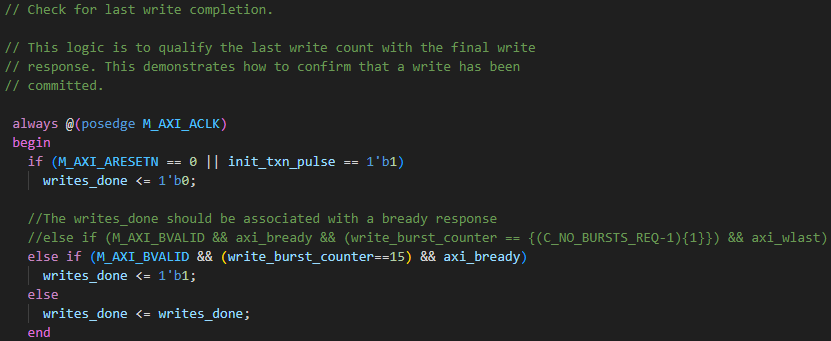


Figure 65. Set writes\_done

After these baic settings, we need to make sure how to pass the right data to the right place. As shown in Figure 66, we have two solutions: First one is using two counters, ***line\_count*** and ***pix\_count***, to store the address information about the current pixel. And the second one is use ***wnext*** to follow the write pace. We finally choose the latter because its more simple and robust.

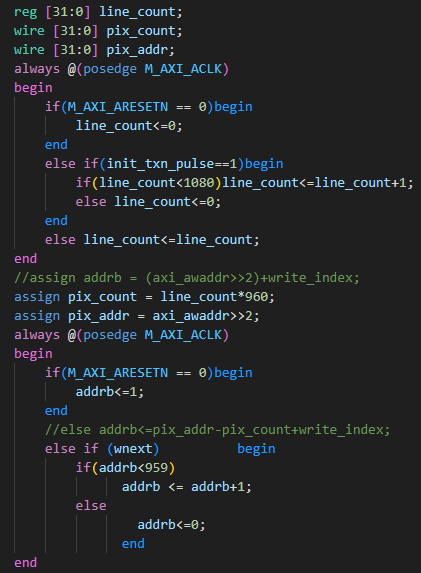
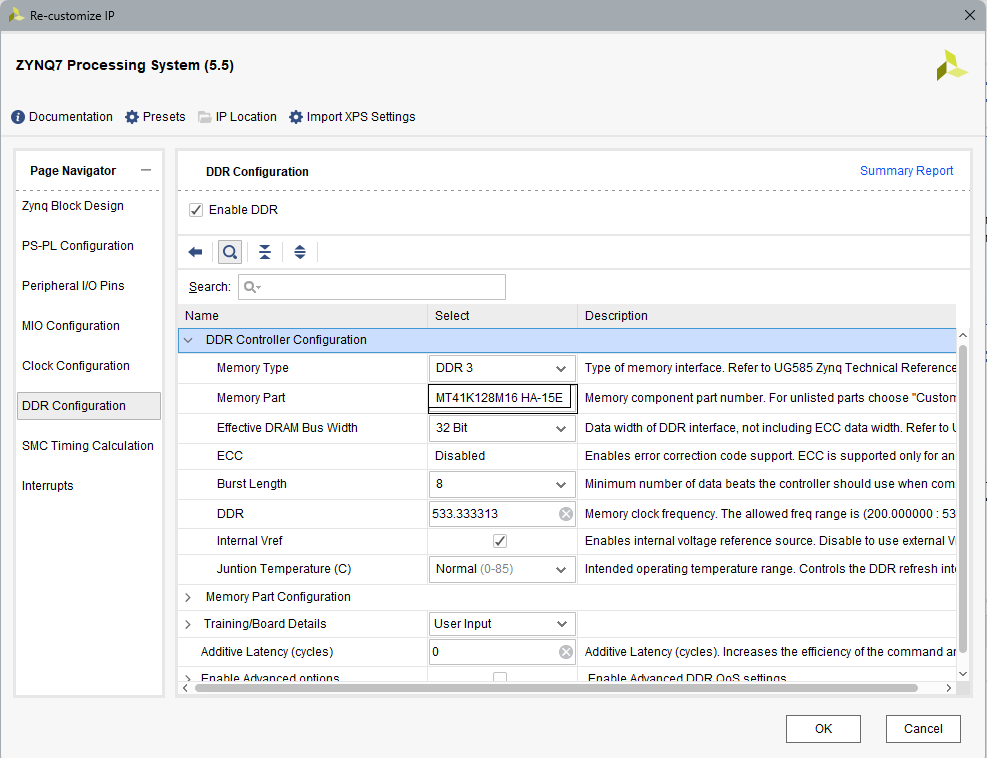
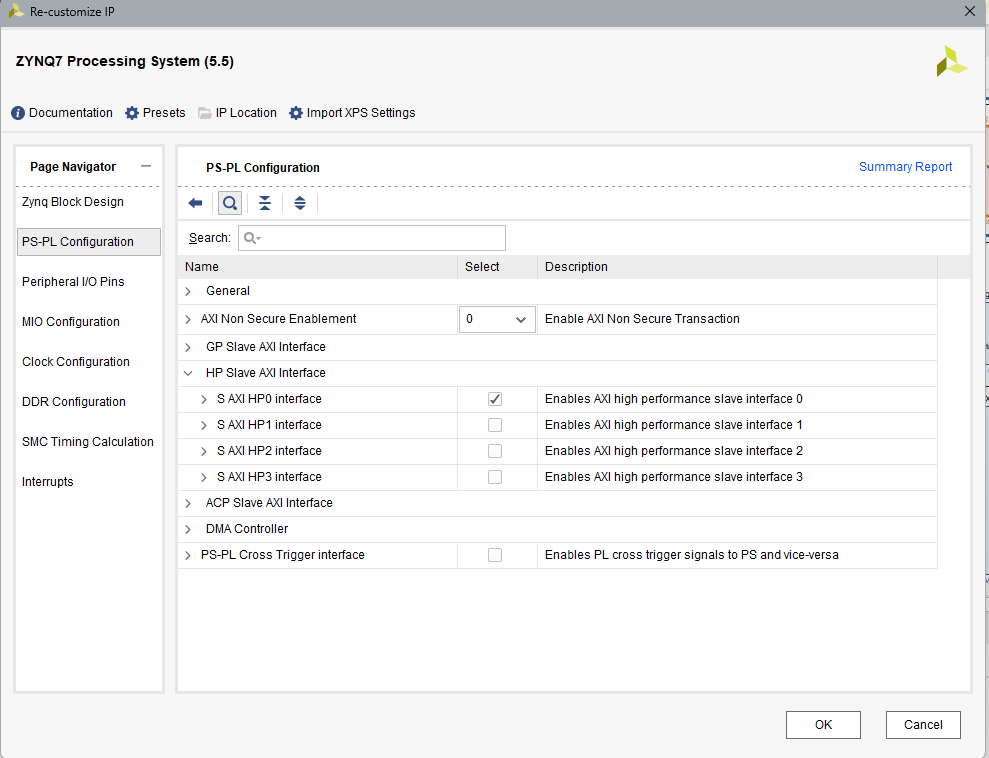


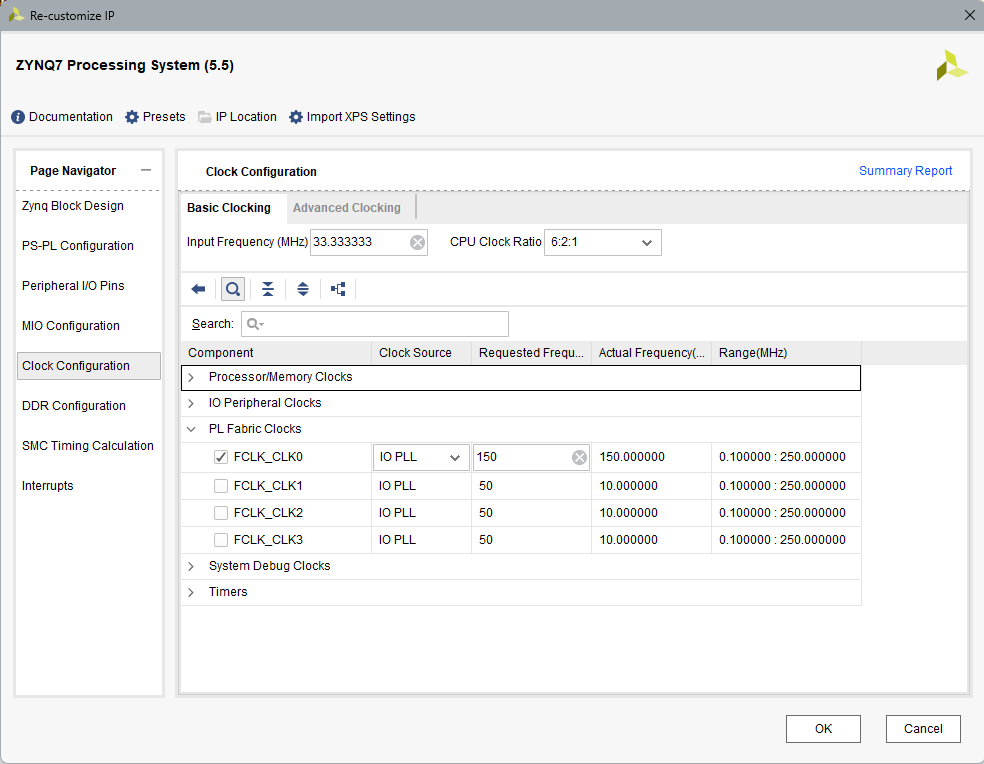
Figure 66. Logic of addrb

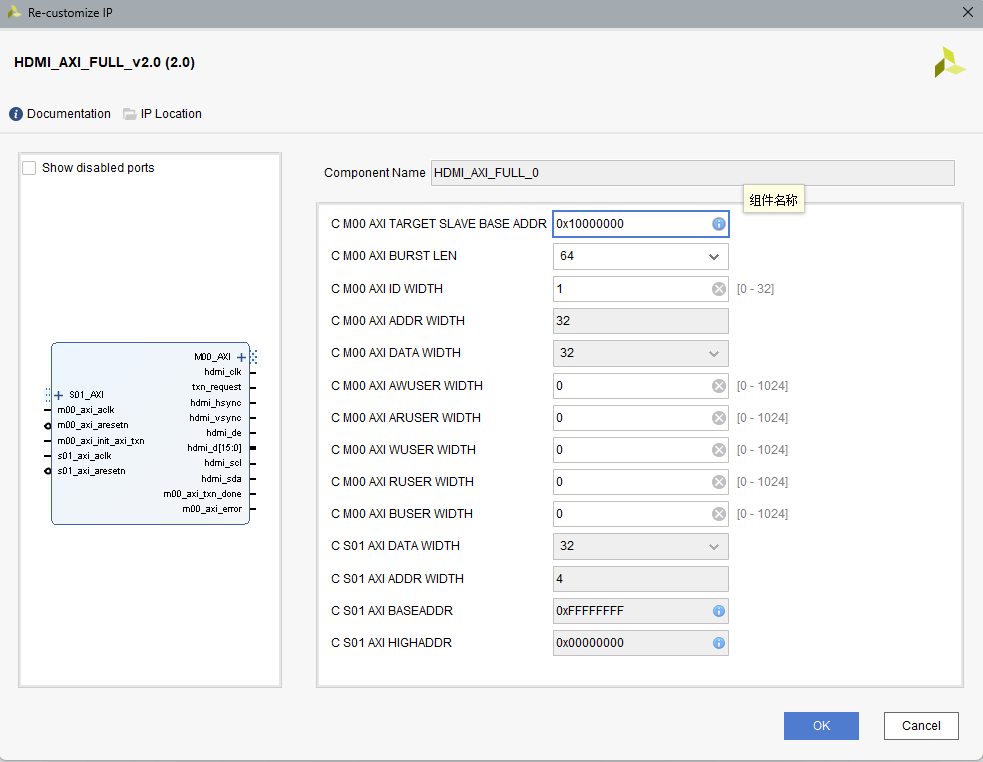
We will not talk about input end of the BRAM because the pixels of this dynamic color bar experiment is calculated, so whether it is synchronous BRAM and asynchronous BRAM, they will always looks the same, you just need to write the pixel from the BRAM to the DDR continuously. The result can be found in directory “./Pics/Master\_Colorbar”.

# Issues Record

(1) Remember to change **Memory Part** to **MT41K128M16HA-15E**

(2) Remember to choose HP0 for the CPU Cache

(3) Need to change the FCLK if you don’t want to use the PLL. For example, change FCLK from the default 100MHz to 150MHz as the operation clock for the 1080P HDMI controller and the AXI Bus.

(4) You need to change the parameters for the AXI interface in the block design manually.

(5) If your image is red, that is because the output time series is not aligned with the data. Try adding 1 to the output address to solve this problem.

(6) If the bottom half of your image is torn, try clearing the cache in C code with ***Xil\_DCacheFlush()***.

(7) If you use the FCLK on the CPU side as the clock of the HDMI controller, you need to download the software code to start the CPU clock to drive the HDMI controller.

(8) **The camera we use this time is IMX219, not OV5647!!! So use the register definition file of IMX219.**

(9) 720P Image Quality Issue Display Output Unstable

Solution: Adjust the resolution and frame rate, optimize the HDMI transmission control logic, ensure stable display output.

Method: In the HDMI transmission control logic, add register buffering to address timing issues and improve stability.

(10) 1080P Display Timing Issue

Solution: Vivado Implementation Timing Violation, optimize timing.

Method: Use the ILA to capture waveforms, adjust the address offset unit (e.g., addr <<= 2), and optimize the multiplication operation.

(11) SRAM Data Writing Issue

Solution: Optimize LED display output SRAM data writing.

Method: Use the offset=1 logic to solve the issue, ensuring stable SRAM data writing.

# Personal summary

陈宇阳: Through my participation in this lab, I have gained a substantial amount of knowledge and hands-on experience with the AXI Bus, particularly its two key variants: AXI-Lite and AXI-Full. Initially, these concepts seemed daunting due to their complexity, but the lab provided a structured environment to explore their intricacies. AXI-Lite, with its simplified protocol, taught me the fundamentals of lightweight data transfers, ideal for control and configuration tasks. In contrast, AXI-Full introduced me to the power of high-performance, burst-based transactions, which are critical for data-intensive applications. By working through practical exercises, I developed a clear understanding of their respective use cases, signal structures, and timing requirements.

One of the most enlightening aspects of the lab was diving into the roles of the slave and master interfaces within the AXI framework. I learned how the master initiates transactions and drives communication, while the slave responds and manages data flow. Implementing these interfaces in simulation and hardware environments helped me appreciate the nuances of their handshaking mechanisms and the importance of adhering to protocol specifications to ensure reliable communication.

Beyond AXI, the lab broadened my perspective on other prevalent communication protocols, such as HDMI, MIPI, and SCCB. Exploring HDMI gave me insight into high-speed multimedia data transmission, particularly its role in modern display technologies. MIPI, with its widespread use in mobile and embedded systems, introduced me to the challenges of balancing bandwidth and power efficiency. SCCB, though less familiar initially, proved fascinating as a lightweight protocol for camera module configuration. Comparing these protocols to AXI deepened my understanding of their design trade-offs and application-specific optimizations.

A particularly exciting part of the lab was learning to manipulate Double Data Rate (DDR) memory. This was a new frontier for me, as I had little prior exposure to memory interfacing. Through project-based learning, I explored DDR's architecture, including its high-speed data transfer capabilities and timing constraints. Configuring and optimizing DDR interactions in our projects was both challenging and rewarding, as it required careful attention to signal integrity and latency management. These hands-on tasks not only solidified my technical skills but also sparked a keen interest in memory systems design.

Perhaps the most valuable outcome of this lab was the development of my teamwork and collaboration skills. Working in a group to design, implement, and troubleshoot complex systems fostered a sense of shared responsibility and mutual support. We faced challenges, such as debugging protocol mismatches or optimizing performance, which required clear communication, division of tasks, and collective problem-solving. These experiences taught me the importance of leveraging diverse perspectives and maintaining patience under pressure, skills that I believe will serve me well in future collaborative endeavors.

In conclusion, this lab has been a transformative learning experience. It not only equipped me with technical expertise in AXI Bus, communication protocols, and DDR memory but also cultivated essential soft skills like teamwork and critical thinking. I am excited to apply these insights to future projects and continue exploring the fascinating world of embedded systems design.

邓高远: Through this experiment, I systematically enhanced my software-hardware co-development capabilities on the SoC platform, gaining valuable experience in areas such as C-language driver development, interface protocol debugging, IMX camera documentation review and analysis, and error localization based on waveforms and display outputs.

Across multiple experimental modules, I was primarily responsible for writing and validating control logic in C, including tasks such as AXI-Lite control register access, SRAM data writing, and dynamic image control. For instance, in the dynamic ball display experiment, constrained by the hardware SRAM space, I designed a software-hardware collaborative strategy: the software periodically adjusted the ball’s position coordinates, while the hardware module read these coordinates in real-time to generate the image output. By setting a refresh cycle of approximately 16ms, I ensured a balance between dynamic effects and display stability.

During the experiment, I undertook extensive software-hardware co-debugging tasks. When encapsulating the HDMI IP as an AXI-Lite peripheral, the display initially showed no signal output. Troubleshooting revealed that the PS (Processing System) part needed to be executed first to enable the AXI clock. This issue deepened my understanding of the dependencies between the hard core and soft core in the ZYNQ architecture, allowing me to gradually master its startup process and timing constraints through debugging. In another experiment, an incorrect AXI register offset unit caused the written values to fail to reach the image buffer correctly. By comparing the code line by line and capturing waveforms with the Integrated Logic Analyzer (ILA), I identified the root cause.

I also handled camera-related tasks, reviewing the IMX219 chip documentation and understanding the details of the SCCB protocol and its differences from standard I²C. I implemented the SCCB analog protocol’s signal control logic based on GPIO interfaces and identified an error in the sample code where the wakeup process was missing, successfully verifying the correctness of the camera register configuration process.

In the HDMI image output and DDR interaction experiment, I further strengthened my understanding of timing and resource constraints. To troubleshoot issues like abnormal image positioning and screen artifacts, I adopted a “reverse-engineering” approach, starting from the display output to trace back to the code logic. By combining simulation waveforms and real-time signal captures from the ILA, I precisely pinpointed issues such as address pipelining and multiplier delays. In the 1080P image processing experiment, I addressed multiple timing violation risks by using bit-shifting instead of multipliers, adding register buffers, and optimizing lookup tables, ultimately improving the stability of image processing.

Overall, this experiment significantly enhanced my engineering practice capabilities. It not only deepened my mastery of low-level knowledge such as the AXI bus protocol, embedded display control, and camera communication but also provided comprehensive training in waveform analysis, problem localization, and software-hardware co-debugging. I gained a deeper understanding that “system design” is not merely about module integration but, more critically, about holistic control of timing, resources, and processes. In the future, I aim to apply these experiences to more complex embedded system development, continuously improving my system design capabilities.

# Reference

[1] Wikipedia, <https://en.wikipedia.org/wiki/HDMI>

[2] Wikipedia, <https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface#AXI4-Lite>

[3]Xilinx, <https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Lite-Interface-Signals>

[4] Wikipedia, <https://en.wikipedia.org/wiki/Y%E2%80%B2UV>

[5] Wikipedia, <https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface>

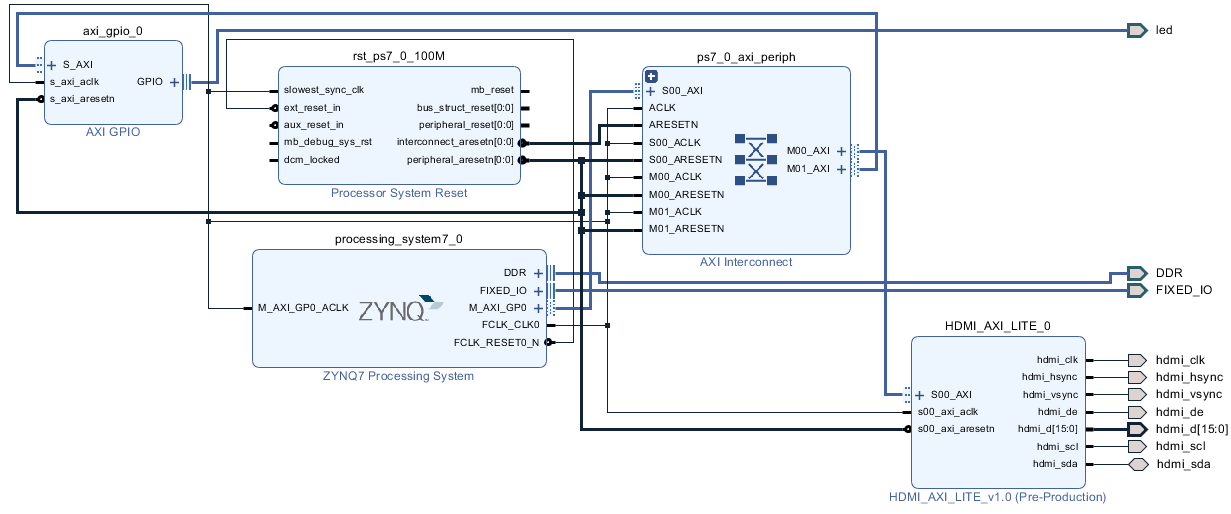
[6] [SCCBSpec\_AN.pdf](docs/SCCBSpec_AN.pdf)

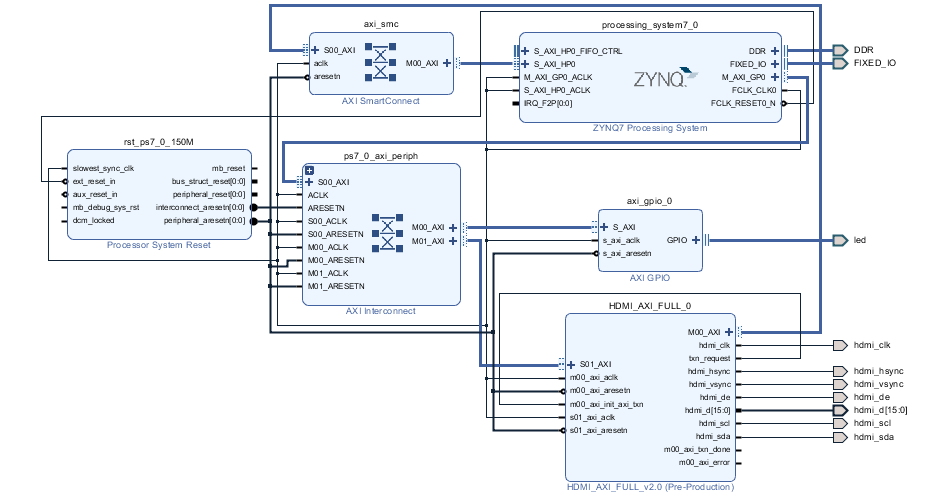
[7] Wikipedia, <https://en.wikipedia.org/wiki/Display_Serial_Interface>

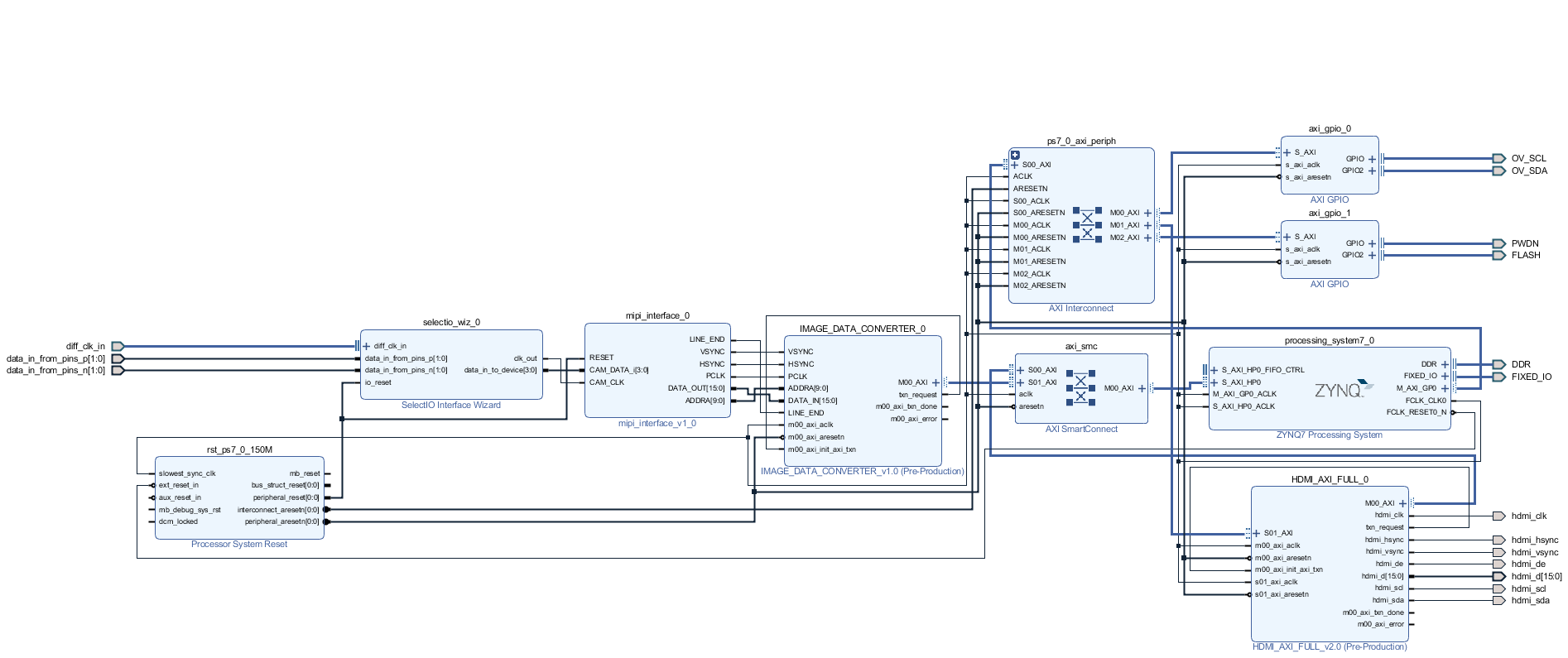
[8] Wikipedia, <https://en.wikipedia.org/wiki/Bayer_filter>

[9] Wikipedia, <https://en.wikipedia.org/wiki/Bayer_filter#Demosaicing>

# Appendix

(1) The **Block Design** for Phase 3

(2) The **Block Design** for Phase 4

(3) The **Block Design** for Camera project in Phase 5

(4) The **Block Design** for color bar project in phase 5

