ECE437: Sensors and Instrumentation

Lab 8: Image Sensor – Acquiring an Image

Introduction

This is the second part of the image sensor lab. In the first part of this lab, you developed the Verilog and Python code necessary to read and write data to the registers in the image sensor. In this lab, you will focus on acquiring an image from the sensor and displaying it on the screen. We are using the grayscale version of the CMOSIS CMV300 image sensor. The image sensor has 648 by 488 pixels and can operate in two modes of operation: high read out mode achieving 480 frames per second and low read out mode of 120 frames per second. The low read out mode outputs in parallel all bits per pixel together with several control signals. We will focus on this mode of operation to acquire and display images. The datasheet for this sensor is located on the course website.

To be able to acquire data from the image sensor, we will need to understand the high-speed data transfer interface provided by Opal Kelly. We will be using *Block-Throttled Pipe Out* module to transfer data to the PC at rates up to ~300MBps. Since the image sensor produces large amounts of data, we will first focus on understanding the Block-Throttled interface and then we will develop the finite state machine to acquire an image.

Relevant Documents for this Lab

Required reading material for this lab:

1. The following Verilog code is necessary for the Block Throttled Pipe Out tutorial: *BTPipeExample.v* and *ClockGenerator.v*. These files are located on the course website.
2. The following Python code is necessary for the tutorial described in this lab: *BTPipeOut_Example.py*. This file is located on the course website.
3. Online documentation for *Block Throttled Pipe Out* module is located here.
4. Online documentation for ReadFromBlockPipeOut Python function is located here.

Additional reference material for this lab:

1. CMOSIS CMV300 image sensor data sheet is located on the course website. Read over the sensor's datasheet and get familiar with the serial interface protocol.

The Goals of This Lab Are:

1. The objective of this lab is to acquire video signal from the CMOS imaging sensor and display this information on the screen. You will get acquainted with high speed data transfer schemes using Front Panel APIs.

Prelab Questions (30 pts):

1. Look over the online documentation for front panel. What is maximum data transfer rates when using Block Throttled Pipe Out? (5 pts)
2. Describe the output signals from the image sensor necessary to acquire an image in parallel read out mode? (5 pts)
3. Draw a block diagram of the FSM that will enable you to read a single frame of data from the image sensor. Summarize how would you implement the FSM in Verilog. (20 pts)

High Speed USB 3.0 Data Transfers

In this section, you will find in depth tutorial on how to send large amounts of data from the FPGA to the PC using Block Throttled Pipes. You will need to carefully go through all the steps in this tutorial and understand the functionality of the code. At the end of this tutorial, you will have to modify the example code and show the results to the TA. Pay attention to all the details in this tutorial.

Why Do We Need High Speed Data Transfers?

Let's look at the data transfer rates necessary to acquire an image from the FPGA and display it on the PC. The CMOSIS image sensor is comprised of 648 by 488 pixels and supports readout rate of 120 frames per second when operating in parallel read out mode. If the data from each pixel is 8 bits or 1 Byte, then the data generated by the image sensor is:

(648 * 488 pixels/frame) * 1Byte/pixel * 120 frames/second = 37.9 MBytes/second

This is a relatively large amount of data that needs to be transmitted to the PC and displayed on the screen. Luckily USB 3.0 supports data rates over 300 Mbytes/second and we can reliably transfer this amount of data. However, there is a problem in the clock frequencies in this data transfer application. USB 3.0 protocol runs on a clocking frequency of ~1 GHz as demended by the protocol. The image sensor runs at 40 MHz clocking frequency. How can the imager communicate with the USB firmware when they both run on two different clock frequencies?

In these scenarios we used FIFOs – first in first out module. FIFO is a block of memory that has one input port, one output port and separate clock signals for both ports. FIFOs are commonly used when we have to cross clock domains. The underlaying assumption is that the output clock frequency is as fast as the input frequency. In other words, as data is written in the FIFO, the data will be read out at speeds equal to or greater than the write speeds. Otherwise, the FIFO will overflow, and we will lose data.

The FIFOs also has read and write enable signals. These signals are used to control when to read and write data to the FIFO. Remember, the read and write clock signals should never be gated i.e. use combination logic with the clock signal. Vivado does a very good job routing signals and making sure that delays are minimized on clock signals. However, once you put a logic such as AND gate on the clock signal, these delays cannot be guaranteed anymore. This is the reason we never put any logic on the clock signal and use enable signals to determine when to read and write data.

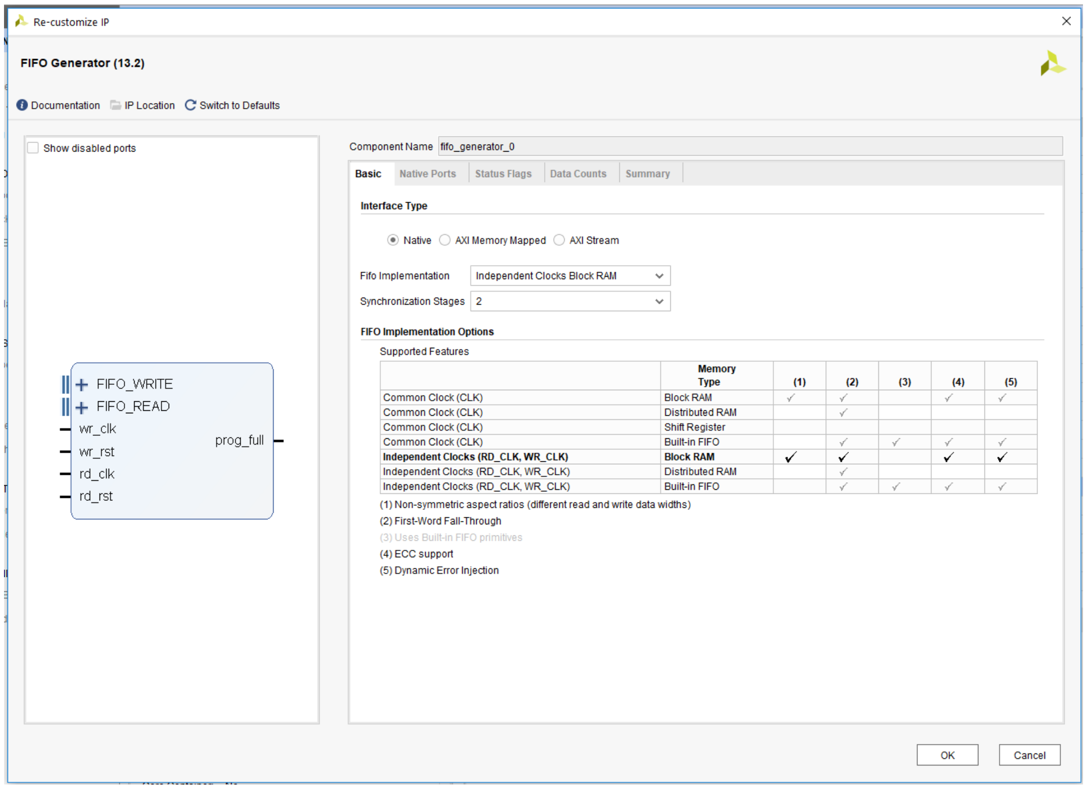
Lastly, FIFOs come with several flag signals. Empty and full signals signal represent if the FIFO has no data or has overflowed, respectively. It also comes with programable full signal. This signal gets asserted when the number of data points, also referred to as words, stored in the FIFO exceeds a set threshold. This feature will be very useful when we setup the control signals with the *Block Throttled Pipe Out* module, which will be explained shortly

Block Throttled Pipe Out Example

Download *BTPipeOut_Example.py* and *BTPipeExample.v* files from the course website. The next step is to create a new project and add *BTPipeExample.v* to the project as a new source file. Add *xem7310_v1.xdc* to the project as a constrain file. If you try to synthetize the project, it will fail because the FIFO module has not been declared. We need to add a FIFO module from the IP Catalog into the project before we can generate a bit file.

We will use the IP catalog to generate a FIFO module via the following steps:

1. Click on *IP Catalog* in the **Flow Navigator** window.
2. Select *Memories and Storage Elements -> FIFOs -> FIFO Generator*.
3. The **FIFO Generator** window will pop up. In the **Basic** sub-window select *Independent Clock Block RAM* in the **FIFO Implementation**. This selection sub-window should look like this:



Component Name fifo_generator_0

Interface Type

[✓ Native] AXI Memory Mapped AXI Stream

Fifo Implementation: Independent Clocks Block RAM

Synchronization Stages: 2

FIFO Implementation Options

Supported Features

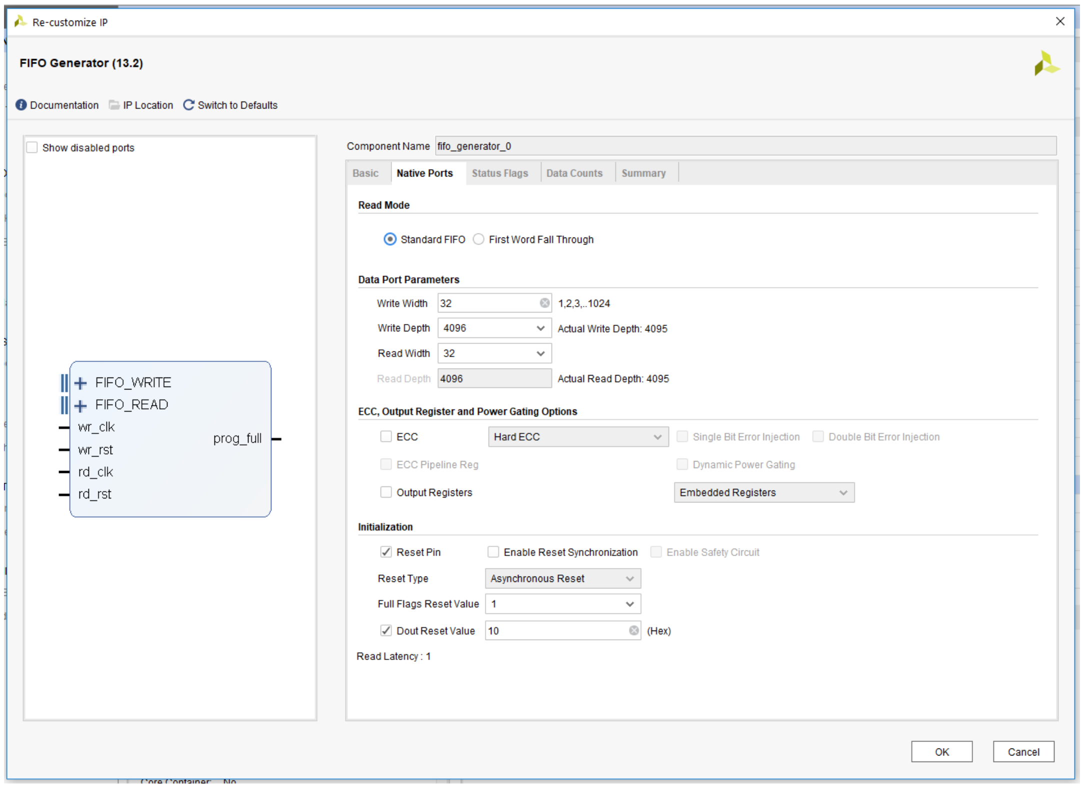
	Memory Type	(1)	(2)	(3)	(4)	(5)
Common Clock (CLK)	Block RAM	✓	✓		✓	✓
Common Clock (CLK)	Distributed RAM		✓			
Common Clock (CLK)	Shift Register					
Common Clock (CLK)	Built-in FIFO		✓	✓	✓	✓
Independent Clocks (RD_CLK, WR_CLK)	**Block RAM**	✓	✓		✓	✓
Independent Clocks (RD_CLK, WR_CLK)	Distributed RAM		✓			
Independent Clocks (RD_CLK, WR_CLK)	Built-in FIFO		✓	✓	✓	✓

(1) Non-symmetric aspect ratios (different read and write data widths)
(2) First-Word Fall-Through
(3) Uses Built-in FIFO primitives
(4) ECC support
(5) Dynamic Error Injection

4. Click on **Native Ports** sub-window next.
5. In this window, we will select the bit depth for both the input and output ports. First, lets look at the output port. The output port of the FIFO will be connected to the USB interface, i.e. to *Block Throttled Pipe Out* module. You can check the requirements

for the input and output wires to *Block Throttled Pipe Out* module here. This module expects a 32-bit data word as an input that will be transferred to the PC. This is the data bit depth that USB 3.0 interface expects when it transfers data to and from the PC. Therefore, we will select **read Width** for the FIFO to be 32 bits long.

6. The data bit depth for the input to the FIFO will be set also to 32-bits. In our example, we will input the values of a 32-bit counter to the FIFO. The Write Depth will be selected to be 4096 word. With these two selections, we have a FIFO that has 4096 words and each word is 32 bits wide. Note, the input and output port bit width does not have to match and you might want to change it depending on your application.
7. Next, we need to select reset pins. We need to be able to reset the FIFO depending on the state machine. For example, at the beginning of our state machine, we might want to reset the FIFO to make sure that there are no erroneous values in the FIFO from the previous data transfer. By turning on the reset signal, we can flush the FIFO. Once you make these selections, this sub-window should look like this:



FIFO Generator (13.2)

Component Name fifo_generator_0

Read Mode

[✓ Standard FIFO] First Word Fall Through

Data Port Parameters

Write Width 32 1,2,3,..1024

Write Depth 4096 Actual Write Depth: 4095

Read Width 32

Read Depth 4096 Actual Read Depth: 4095

ECC, Output Register and Power Gating Options

☐ ECC Hard ECC ☐ Single Bit Error Injection ☐ Double Bit Error Injection

☐ ECC Pipeline Reg ☐ Dynamic Power Gating

☐ Output Registers Embedded Registers

Initialization

☑ Reset Pin ☐ Enable Reset Synchronization ☐ Enable Safety Circuit

Reset Type Asynchronous Reset

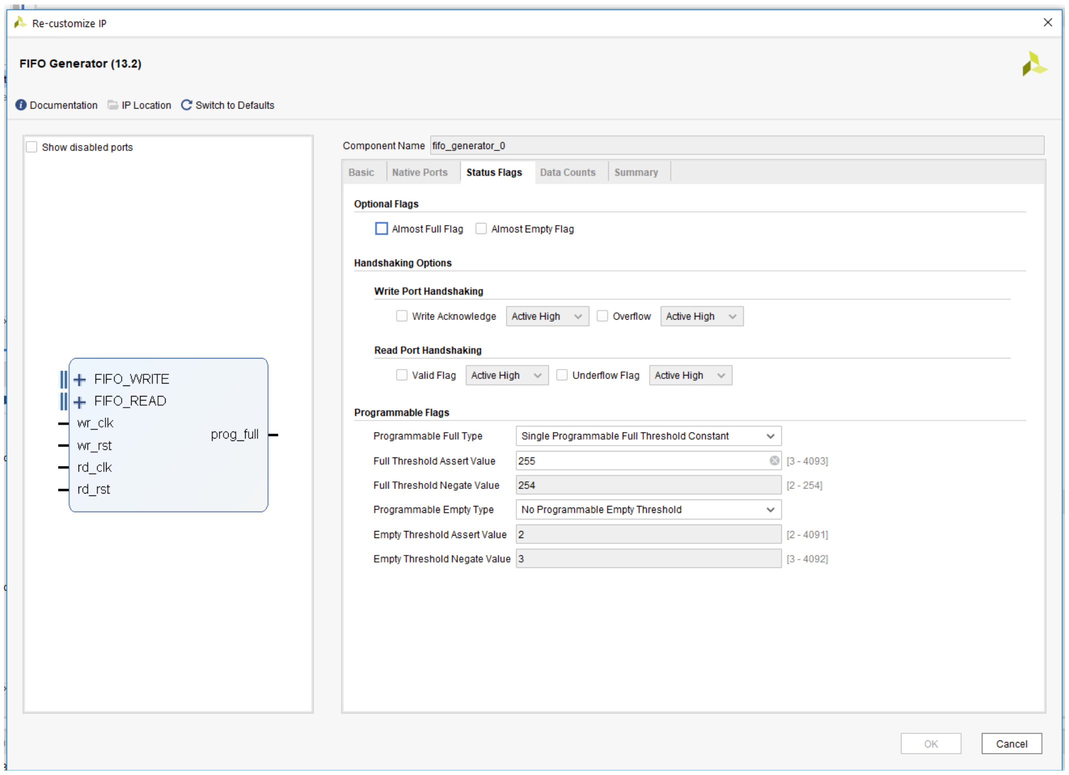
Full Flags Reset Value 1

☑ Dout Reset Value 10 (Hex)

Read Latency : 1

8. Next, we will modify the different flags in the **Status Flags** sub-window. The FIFO comes by default with full and empty signals and we don't need to do anything about these signals. However, we need to declare a programmable full signal and set the threshold to 255. The reason for this is the following. The programmable full signal will be asserted when there are at least 255 words stored in the FIFO. Since each word is 32 bits long or 4 bytes, it means that there are at least 1024 Bytes stored in the FIFO. On the Python side, we will use an instruction that will receive the data from the Block Throttled Pipe. This instruction, which we will discuss shortly, will expect to have at least 1024 Bytes of data in the FIFO before it will start fetching data from the USB.

If there are less than 1024 Bytes of data in the FIFO, then at some point the FIFO will be empty and erroneous data will be sent to the PC.
After these selections, this sub-window should look like this:



Component Name: fifo_generator_0

Optional Flags
- Almost Full Flag ☐
- Almost Empty Flag ☐

Handshaking Options

Write Port Handshaking
- Write Acknowledge ☐ Active High
- Overflow ☐ Active High

Read Port Handshaking
- Valid Flag ☐ Active High
- Underflow Flag ☐ Active High

Programmable Flags

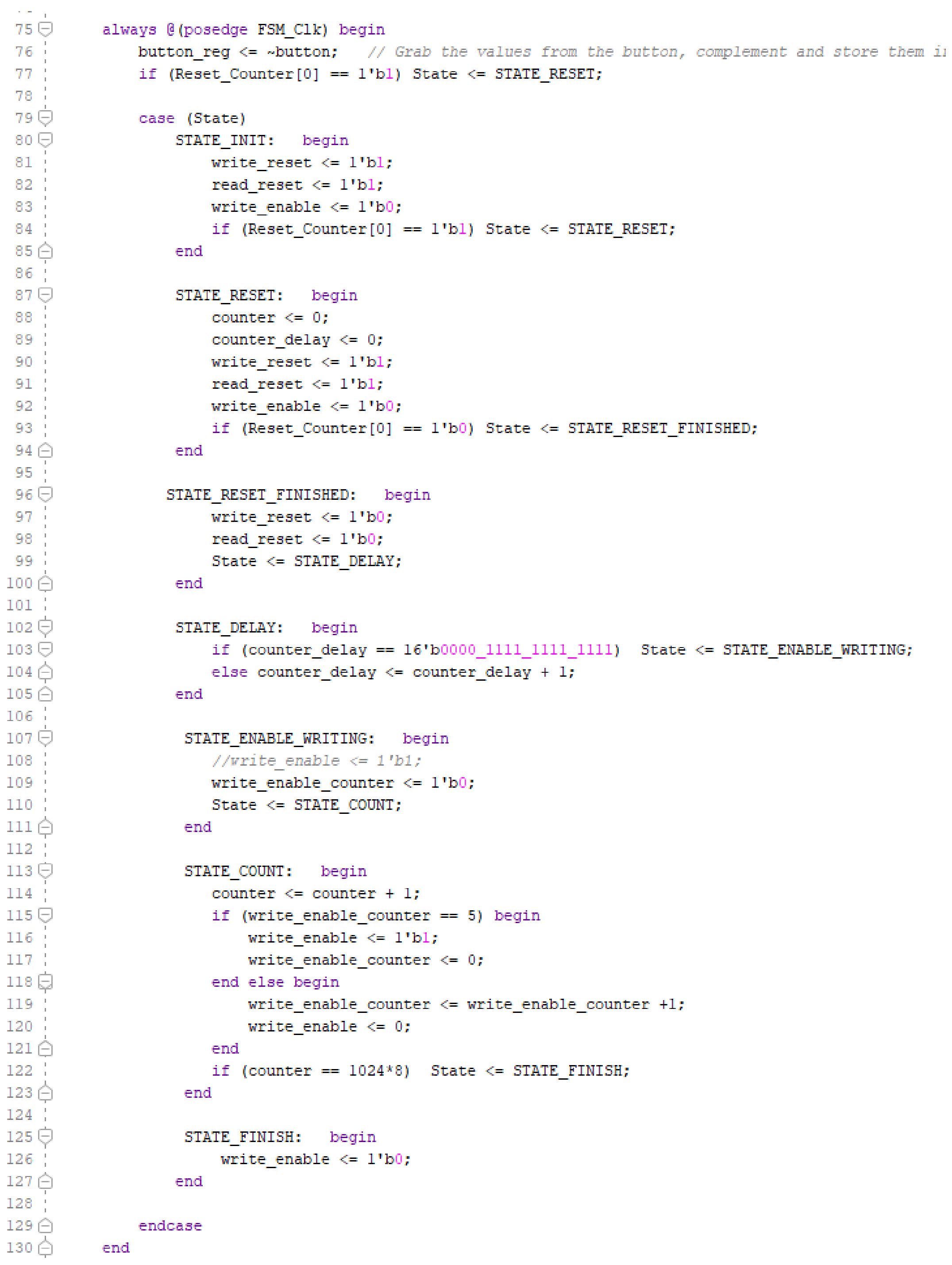
Programmable Full Type	Single Programmable Full Threshold Constant	
Full Threshold Assert Value	255	[3 - 4093]
Full Threshold Negate Value	254	[2 - 254]
Programmable Empty Type	No Programmable Empty Threshold	
Empty Threshold Assert Value	2	[2 - 4091]
Empty Threshold Negate Value	3	[3 - 4092]

9. Click on the *OK* button. A new window titled **Generate Output Products** will appear. Click on the *Generate* button. Vivado will start generating the FIFO module and after few seconds you will receive a message **Out-of-context module run was launched for generating output products.** Clock on the *OK* button. You have successfully generated a FIFO in your project.
10. The next step is to add all the OpalKelly Verilog files necessary for USB communication. Click on *Add Source* and add all the Verilog files located in C:\Program Files\Opal Kelly\FrontPanelUSB\FrontPanelHDL\XEM7310-A75 directory.
11. Now you can synthetize, implement and generate a bit file for your project.

Lets take a look at the example code. Double click on *BTPipeExample.v* file. The Verilog code up to line 43 should be easy to understand as you have seen it in the previous labs. If it is not clear, go over the tutorials in labs 2 and 3.

Lines 44 through 55 define local parameters which will be used to define the 7 different states in the FSM. Lines 52 through 61 define various wires and registers that will be used in the code. For debugging purposes, we have connected various LEDs to different wires in the code between lines 63 and 69.

Next is the FSM code. The FSM code is shown below:



```
always @(posedge FSM_Clk) begin
    button_reg <= ~button;   // Grab the values from the button, complement and store them i
    if (Reset_Counter[0] == 1'b1) State <= STATE_RESET;

    case (State)
        STATE_INIT:   begin
            write_reset <= 1'b1;
            read_reset <= 1'b1;
            write_enable <= 1'b0;
            if (Reset_Counter[0] == 1'b1) State <= STATE_RESET;
        end

        STATE_RESET:   begin
            counter <= 0;
            counter_delay <= 0;
            write_reset <= 1'b1;
            read_reset <= 1'b1;
            write_enable <= 1'b0;
            if (Reset_Counter[0] == 1'b0) State <= STATE_RESET_FINISHED;
        end

       STATE_RESET_FINISHED:   begin
            write_reset <= 1'b0;
            read_reset <= 1'b0;
            State <= STATE_DELAY;
        end

        STATE_DELAY:   begin
            if (counter_delay == 16'b0000_1111_1111_1111)  State <= STATE_ENABLE_WRITING;
            else counter_delay <= counter_delay + 1;
        end

         STATE_ENABLE_WRITING:   begin
            //write_enable <= 1'b1;
            write_enable_counter <= 1'b0;
            State <= STATE_COUNT;
         end

         STATE_COUNT:   begin
            counter <= counter + 1;
            if (write_enable_counter == 5) begin
                write_enable <= 1'b1;
                write_enable_counter <= 0;
            end else begin
                write_enable_counter <= write_enable_counter +1;
                write_enable <= 0;
            end
            if (counter == 1024*8)  State <= STATE_FINISH;
         end

         STATE_FINISH:   begin
             write_enable <= 1'b0;
        end

    endcase
end
```

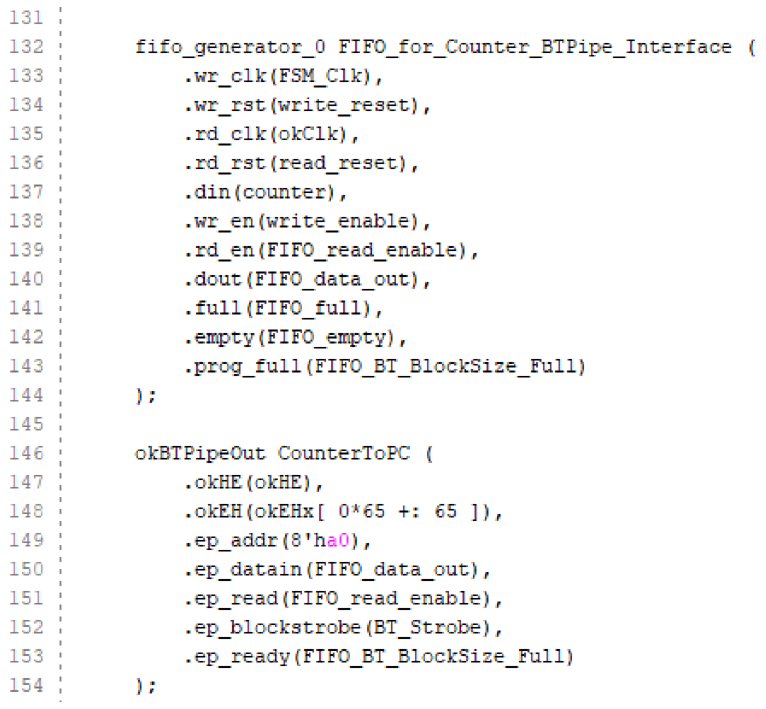
The FSM starts in state STATE_INIT. In this state, we like to reset the FIFO (both the read and write pointer) and disable writing to the FIFO. We will remain in this state until a reset signal is sent from the PC. This signal is sent via WireIn at memory address 0x00. We only examine bit 0 of the received data to determine if we want to go to the reset state. Once we send a reset command from the PC, the FSM will enter the STATE_RESET state. In this state, two registers will be set to 0 value. Once the PC send 0 on memory address 0x00, the FSM will move to the STATE_RESET_FINISH state. Using this sequence of states, we carefully control the reset phase of the FIFO from the PC. In this STATE_RESET_FINISH we release the reset signals and now the FIFO is ready for reading and writing. However, the write_enable signal is still in state "0" and the FIFO will not write any new value until this signal is asserted to "1".

Before we assert the write signal, the FSM will go through a short delay state called STATE_DELAY followed by STATE_ENABLE_WRITING. The reason for this comes from the timing constrains from the PC side. On the PC side, i.e. in your Python code, you will send an UpdateWireIns command followed by ReadFromBlockPipeOut (lines 38 and 41 in the Python code). Although these two instructions get executed one after another, the ReadFromBlockPipeout has to do several handshaking steps before data can be read. If this process takes too long, then the FIFO can overflow if the data was immediately stored after the UpdateWireIns was executed. To avoid this scenario, we incorporated a short delay in the FSM. Once the FIFO has at least 1024 bytes, it will send the programmable full signal high and Python can start reading data from the USB interface – more on this concept shortly.

The next state of the FSM is the STATE_COUNT. In this state a 32-bit counter is increment on every clock cycle and its output is connected to the FIFO input port. However, we control the write enable signal, which is asserted every 6 clock cycles. This means that the FIFO will store first the number 6, followed by 12, 18, 24 and so on. This will give you a sense how you can control what data gets written in the FIFO. **Remember, you do not want to gate your clock signal to the FIFO. Use the read enable signal for this purpose.**

Once we have written 8*1024 words in the FIFO, we will go to the STATE_FINISH state, where we disable the write enable signal.

Lastly, lets take a look at the FIFO and okBTPipeOut modules. Here is the code you should see in the example:



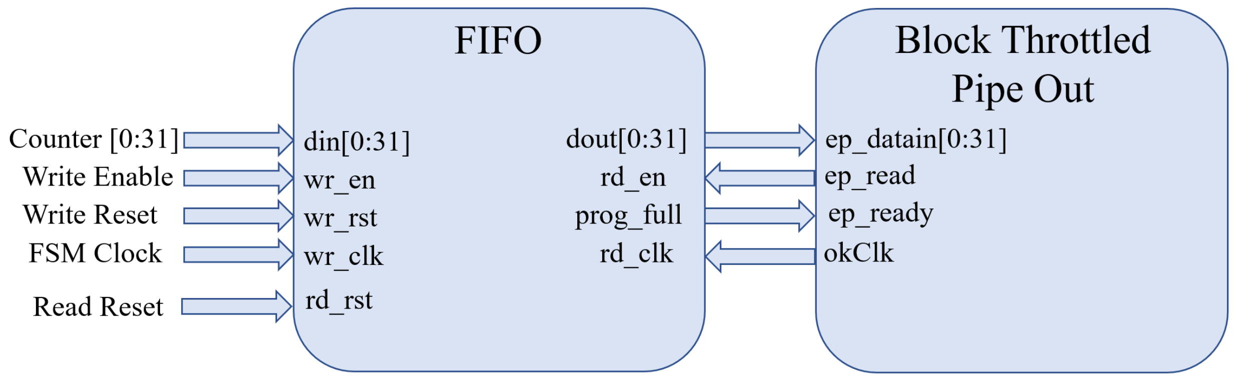
```
fifo_generator_0 FIFO_for_Counter_BTPipe_Interface (
    .wr_clk(FSM_Clk),
    .wr_rst(write_reset),
    .rd_clk(okClk),
    .rd_rst(read_reset),
    .din(counter),
    .wr_en(write_enable),
    .rd_en(FIFO_read_enable),
    .dout(FIFO_data_out),
    .full(FIFO_full),
    .empty(FIFO_empty),
    .prog_full(FIFO_BT_BlockSize_Full)
);

okBTPipeOut CounterToPC (
    .okHE(okHE),
    .okEH(okEHx[ 0*65 +: 65 ]),
    .ep_addr(8'ha0),
    .ep_datain(FIFO_data_out),
    .ep_read(FIFO_read_enable),
    .ep_blockstrobe(BT_Strobe),
    .ep_ready(FIFO_BT_BlockSize_Full)
);
```

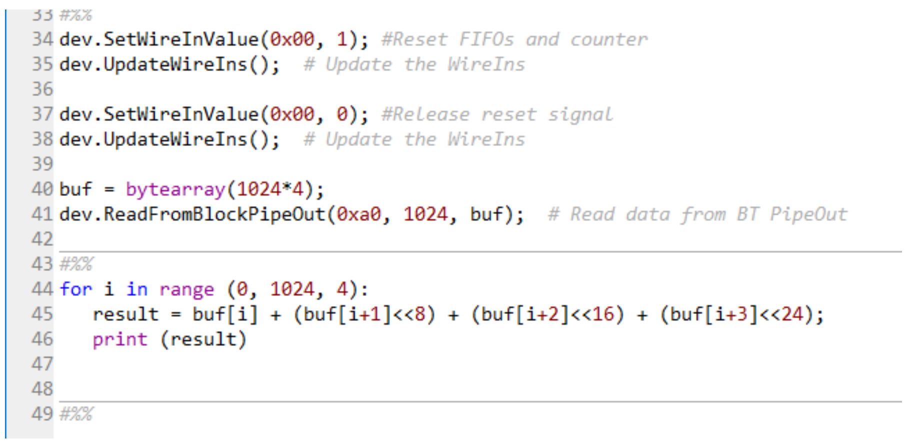
The FIFO module has 11 input and output signals. You will notice that the write clock is connected to the FSM clock signal (*FSM_Clk*) and the read clock is connected to the USB clock (*okClk*). Since the frequency of the USB clock is much higher than the FSM clock signal, we will not overflow the FIFO. The data input to the FIFO comes directly from the counter. The output data from the FIFO goes directly to the *okBTPipeOut* module – this is the data that will be shipped to the PC.

The write enable signal is controlled from our FSM. The read enable signal is controlled by *okBTPipeOut* module – when the USB is ready to read and transfer data, it will assert this signal and fetch another 32-bit word from the FIFO. Another important signal is program full (FIFO_BY_BlockSize_Full). This is an output signal from the FIFO and gets asserted when the FIFO has at least 255 words. *okBTPipeOut* module relays on this signal to start with the data acquisition. If you look at line 41 in the python code (dev.ReadFromBlockPipeOut(0xa0, 1024, buf)), we are declaring that the block size transfer is 1024 Bytes. This means that the FIFO has at least 1024 Bytes ready to transfer to the PC.

For clarity reason, here is block diagram of how the signal between the FIFO and Block Throttled Pipe Out are connected:



Let's take a look at the Python side of this exercise. The code between lines 1 and 32 should be the same as before. Make sure that the path directory to the bit file is set correctly on line number 12. The rest of the code should look like this:



```
33 #%%
34 dev.SetWireInValue(0x00, 1); #Reset FIFOs and counter
35 dev.UpdateWireIns();  # Update the WireIns
36
37 dev.SetWireInValue(0x00, 0); #Release reset signal
38 dev.UpdateWireIns();  # Update the WireIns
39
40 buf = bytearray(1024*4);
41 dev.ReadFromBlockPipeOut(0xa0, 1024, buf);  # Read data from BT PipeOut
42
43 #%%
44 for i in range (0, 1024, 4):
45     result = buf[i] + (buf[i+1]<<8) + (buf[i+2]<<16) + (buf[i+3]<<24);
46     print (result)
47
48
49 #%%
```

In line number 34, we set the wire in on memory address 0x00 to 1 and this value is sent to the PC when UpdateWireIns is executed. You can execute manually these two commands once the bit file is loaded and you can observe the results on the LEDs. Remember that the LEDs are connected to different wires and flags in the FSM which makes is easier to debug. When we send a "1" on memory address 0x00, the FSM is in reset mode. In this state, the read and write pointer to the FIFO are reset. Next, we send a value "0" on the same memory address, which moves the FSM out of the reset state and the FIFO will start filling up after the delay phase is executed in the FSM.

The next two instructions on lines 40 and 41 define the BT Pipe Out. The variable buf will store the results from the USB data transfer. This variable is "1024*4" bytes long. This means that the ReadFromBlockPipeOut will read total of 4096 Bytes. The first parameter in the ReadFromBlockPipeOut defines the memory address where the data will be transferred. The second parameter specifies the transfer block size. The block size defines the packet size of data transfer in bytes. The total amount of data transfer is 4096 because the buf variable is 4096 bytes long. However, this data is transferred in chunks of 1024 and it means that the FPGA has 1024 bytes of data ready for transfer. This is the reason why we set the FIFO programmable full to 255 words, where each word is 4 Bytes long. When programmable full signal is high it means that the FIFO has 4096 Bytes ready for transfer.

The last part of the code in lines 44 to 46 prepare the results for printing. Since the buf variable is 8 bits long and the FIFO output data is 32 bit long, we need to concatenate 4 bytes from the buf variable to form a result. This is achieved in line number 46 of the code by appropriately shifting four consecutive bytes and adding them together.

Checkpoint 1 (100 points)

You will design the necessary Verilog and Python code to acquire a single image and display it on the screen. Look at the datasheet for information on how to implement your finite state machine. To get full credit, you will need to properly display a single image on the computer screen.

Here are some tips on how to avoid common problems:

- Read carefully over the ReadFromBlockPipeOut function requirements. Note that the length of data transfer has to be multiple of 16. Also, the length of the data transfer has to be integer multiple of the block size. The total number of data points per image is 648 x 488 x 1 Byte/pixel = 316,224 Bytes. If you use the same block size as in the tutorial, i.e. 1024, then the total data transfer is NOT an integer multiple of 1024. You will need to change either the imager resolution of the block size.
- Data valid and frame valid signals are very useful control signals for the FIFO write enable.