CprE 488 – Embedded Systems Design

MP-2: Digital Camera Design

Assigned: Monday of Week 6
Due: Monday of Week 8
Points: 100 + bonus for additional camera features

[Note: at this point in the semester you should be fairly comfortable with using the Xilinx VIVADO development environment, and so these directions will only expand upon the parts that are new. The goal of this Machine Problem is for your group to become more familiar with three different aspects of embedded system design:

1. *IP integration – you will work with several different IP cores that interface on the AXI Stream bus.*
2. *Digital image processing – you will gain exposure to some of the basic computational image processing kernels, and will build a digital camera by combining the individual components.*
3. *HW/SW tradeoffs – you will analyze the performance tradeoffs inherent in an embedded camera system as you first design software components and iteratively replace them with equivalent hardware IP cores.]*

1) Your Mission. You've had a great run as a Chemical Process Engineer at Eastman Kodak. While it's true that cameras using Polaroid and other film-based technology are not as popular as in their heyday, look on the bright side! You have a generous pension plan, great coworkers, and a modest 4 bedroom home in lovely Rochester, NY. With mere months to go until you can start enjoying your retirement, you're surprised by an impromptu meeting request by the CEO. The (one-sided) conversation starts a bit ominously, and proceeds at a rapid pace: "Sit down, we need to talk. They say you're my most capable engineer. Now I keep hearing about these so-called *digital* cameras. I don't know what that is, and frankly, new technology scares me. So I need a prototype on my desk in no more than 14 days – no excuses!"

It's time to get to work. You know a little bit about camera optics (and certainly HW-2 provided a quick refresher), but how to transform that to a useful digital output is well outside your comfort zone. Fortunately, you have a skeleton project that provides the basic framework. Your task is to use system design techniques to implement an image processing pipeline and other functionality commonly found in digital cameras.

2) Getting Started. The ZedBoard wasn't sufficiently complicated by itself, so we've coupled it with the Avnet FMC-IMAGEON card. The FMC-IMAGEON connects via the FPGA Mezzanine Card (FMC) connector on the ZedBoard, and provides the following features:

- Video input via two sources: the ON Semiconductor VITA family of image sensors, and an HDMI input interface
- Video output via an HDMI output interface
- A configurable video clock synthesizer
- I2C interfaces for FMC board configuration as well as for reading from an IPMI identification EEPROM
- SPI interface for Camera Sensor configuration

The FMC connector's pins are directly routed to the Zynq FPGA on the ZedBoard, and so the I2C and other peripheral controllers need to be instantiated in our VIVADO design. Make sure the ZedBoard is turned off while plugging in the FMC-IMAGEON. **Note:** that the HDMI output controller can become incorrectly configured when a new software application is downloaded, so when testing new software designs you will want to first re-download the FPGA bitstream and/or reboot the ZedBoard.

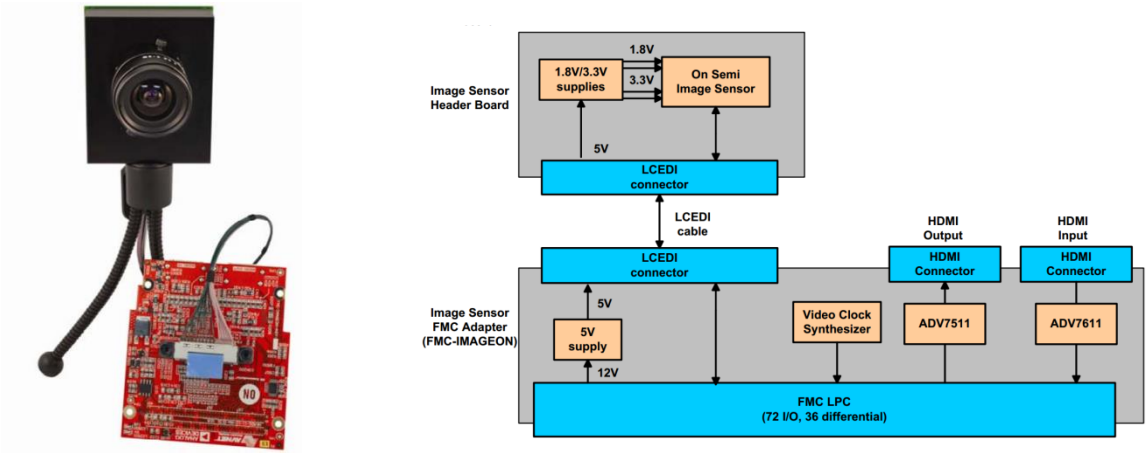

Figure: ON Semiconductor Image Sensor with HDMI Input/Output FMC Bundle – photo and block diagram. They're quite expensive so please don't break it. Pretty pretty please.

Given the complexity of this assignment, we have provided a starter VIVADO project that you can use as the baseline for implementing the digital camera functionality. Download the provided MP-2.zip file, unzip, and peruse the directory structure. You will find another zip file called mp2.xpr.zip. This contains a base project with an initial design for HDMI video output. Unzip this zip file and open the project in Vivado using Open Project, then find the xpr file within the project folder.

3) Design Test and Analysis. The initial design provides a Test Pattern Generator (TPG) that creates a 1080p image, which is streamed into DRAM via our old the friend the Video Direct Memory Access (VDMA) module. A software loop performs some simple processing on the incoming video stream, and the VDMA takes the processed pixels and streams them to the HDMI out on the FMC-IMAGEON card.

The design is already 100% functional, so you just need to 1) Generate a bitstream, 2) Export HW (Include Bitstream), 3) Launch SDK – but don't walk away! While the system is building, analyze the design using the Block Diagram view, and Implementation view, as well as directly through the `zedboard_fmc_imageon_gs.xcd` file. In your writeup provide the following:

- A detailed system diagram that illustrates the interconnection between the various modules in the system, both at the IP core level (i.e. the components in your VIVADO design) as well as the board level (i.e. the various chips that work together to connect the output video to your monitor). The documents found in `MP-2/docs/Camera` will be of assistance in understanding the various components in the FMC-IMAGEON board, and the IP core documentation is found in `MP-2/docs/IP`.
- A detailed description of how the hardware in the starter MP-2 design is intended to operate. Make sure to describe the role of the various I2C interfaces, how the Video Timing Controllers (VTCs) are

being used, and what differentiates this VDMA from the version we used in MP-0. Also, explain the role of the various clocks in the system (be specific).

After FPGA bitfile generation has completed, and you have Exported the Hardware and Launched SDK, we need to set up the software side of the project. Use the follow steps:

1) Create a New Applicate Project. Call it MP2-TPG. Select the "Hello World" Template
2) Delete "hellowold.c"
3) In the MP2 file structure you downloaded copy all the files in the directory `sw/camera_app/src` and paste them into the MP2-TPG/src directory.
4) Add the Repository for the Camera/HDMI software drivers to the project path
    a) Go to Xilinx -> Repositories. Select New under "Local `Repositories`".
    b) Navigate and select in the MP2 file structure: `sw/ip_repo_sw`
    i) Click "Rescan Repositories"
    ii) Click "Apply"
    iii) Click OK
5) Add the drives to your MP2-TPG_bsp:
    a) For MP2-TPG_bsp, right-click and select "Board Support Package Settings"
    b) Check the boxes for: 1) fmc_iic_sw, 2)fmc_imageon_sw, 3)fmc_ipmi_sw, 4) onsemi_vita_sw
    c) "Regenerate BSP Sources"
6) Rebuild the MP2-TPG project:
    a) Right click and select "Clean Project"
    b) Right click and select "Build Project"

Download the bitfile and `camera_app` executable to your board to ensure that the starter design is working correctly. Modify the test pattern that is being generated to demonstrate your understanding of the general `camera_app` structure. Provide at least two modifications: one which configures the TPG core directly (see the provided `TPG datasheet` for several examples of this), and one which uses the software processing loop in `camera_app.c` to modify the incoming video stream. Describe in your writeup what changes you made, and save a copy of any files modified (presumably only `camera_app.c` and `fmc_imageon_utils.c`) during this process into a folder named `part3/`.

4) Grayscale Camera. Navigating back to VIVADO, it is time to interact with the Image Sensor on the FMC-IMAGEON board. The following gives detailed direction for how to connect to the Camera sensor. Also a nice detailed PDF of what your system should look like after these steps is provided in the top level of the MP2 file structure. It is called "Gray-Scale-PassThrough.pdf:

1) Add the VITA SPI Controller IP block
    i) The default name of the component should be "onsemi_vita_spi_0", if not then update it.
    ii) Make the "IO_SPI_OUT_spi" port an External Connection ("Make External").
    iii) Rename the created port to "IO_VITA_SPI", allowing the bus name to match the xdc constraint file
    iv) Connect the "oe" pin to 1.
    v) Connect "s00_axi_aresetn" to the "peripheral_aresetn" of the 76 MHz Processor System Reset block
    vi) Connect the s00_axi_aclk to the 76 MHz clock (FCLK_CLK0)
    vii) Connect the S00_AXI bus to the AXI Interconnect (easiest way is to use the "Run Connection Automation" and selcet teh FCLK_CLK0 option for the interconnect)
    viii) Uncomment the "SPI for Camera configuration" section of the xdc constriants file

2) Add the VITA Camera Receiver IP block

i) The default name of the component should be "onsemi_vita_cam_0", if not then update teh name to this.

ii) Make the IO_CAM_IN port an Exteranl Connection ("Make External").

iii) Rename the created port to IO_VITA_CAM, this will allow the name of this bus to match what is in the xdc constraint file.

iv) Connect the "oe" pin to 1.

v) Connect the "s00_axi_aresetn" to the "peripheral_aresetn" of the 76 MHz Processor System Reset block

vi) Connect the s00_axi_aclk to the 76 MHz clock (FCLK_CLK0)

vii) Connect the S00_AXI bus to the AXI Interconnect (easiest way is to use the "Run Connection Automation" and select the FCLK_CLK0 option for the interconnect)

viii) Connect the "trigger pin" to 0.

ix) Connect the "clk" pin to pin "clk_out1" of the clk_wiz_0 component

x) Connect the "reset" pin to the "peripheral_reset" of the 148 MHz Processor System Reset block

xi) In the ZYNQ7 Processing System block enable FCLK_CLK2 and set to 200 MHz

xii) Connect pin "clk200" to the "FCLK_CLK2" pin of the ZYNQ7 Processing System block

xiii) Configure this IP core as follows: Default setting should be fine and be set to: Video Data Width = 8, VITA Data Channels = 4

xiv) Uncomment the "Camera data, clocking, reset, trigger, and sync" section of the xdc constraints file (Except for the one labeled "phjones: NOT USED").

3) Add the Video In to AXI4-Stream IP block

i) The default name of the component should be "v_vid_in_axi4s_0", if not then update teh name to this.

ii) Configure this IP core as follows: Video Format = Mono/Sensor, Input Component Width = 8, Output Component Width = 8, FIFO Depth = 4096, Clock Mode = Common

iii) Connect the "vid_io_in_ce", "aclken", and "axis_enable" pin to 1.

iv) Connect the "aclk" pin to pin "clk_out1" of the clk_wiz_0 component

v) Connect the "aresetn" pin to the "peripheral_aresetn" of the 148 MHz Processor System Reset block

vi) Connect the "vid_io_in" pin to pin "VID_IO_OUT" the onsemi_vita_cam_0 component.

vii) Disconnect the pins of the v_tpg_0 component, using "Disconnect Pin" for each pin (Note: do not just delete the commonent, as the tool may move more wires than you want).

vi) Delete the v_tpg_0 component

vii) Configure component "axis_subset_converter_0" as follows: Slave Interface Signal Properties: TDATA Width = 1, Master Interface Signal Signal Properties = 2. Extra Settings, TDATA Remap String = 8'b10000000,tdata[7:0]

viii) Connect the "video_out" pin to the pin "S_AXIS" of component axis_subset_converter_0

4) Set FCLK_CLK0 of the Processing System to 75 MHz (It will generate a 76'ish MHz clock)

5) Check that all lines of the XDC file are uncommented, except for the one labeled as "phjones: UNUSED"

Questions:

1) Note that several of these ports in the XDC file are paired together, with one port ending in `_p` and the other ending in `_n`. In your writeup, briefly describe what this pairing of signals signifies, and what this configuration is typically used for.

2) We convert the 8-bit output of the "Video In to AXI4-Stream" IP core to 16-bits to be given to the VDMA by appending the 8-bit value "`10000000`" (see step 3.vii),. Explain why this is an appropriate value to append, and why appending "`00000000`" would not make sense.

Confirm that your Block Design passed the validate check, and then follow the typical steps for building a bitfile (See MP-0 tutorial if you need a reminder on these steps), then Export the Hardware and Launch SDK. While you wait, meditate on what you have learned.

Things on the software side are not nearly as complicated. In `fmc_imageon_utils.c`, uncomment the functionality for the vita receiver initialization code, and make sure you call *fmc_imageon_enable_vita()* instead of *fmc_imageon_enable_tpg()*. Note that since the appropriate libraries do not get included until the core is added to the project, you will also need to uncomment the vita-related code in `camera_app.h` and `camera_app.c` as well. Remove any previous transformation code in *camera_loop()*, and test that your design works as expected. In your writeup, briefly explain why the camera at this stage is not outputting any color.

5) Color Conversion Software. As discussed in HW-2, we can colorize this grayscale camera by applying a Bayer filter.

A) Matlab prototype of Color Conversion Software:

- Use Matlab to create a "Bayer" image that emulates what the camera will provide your system. You can create such a test image by starting with a RGB 24-bit BMP color image (choose an image) and converting it to a "Bayer" image by just using a signal color component from each pixel in the color image based on the "Bayer Pattern" (as a quick estimate) or you could apply the reverse transformation that would be applied by the CFA.
- Now use Matlab to implement an algorithm to convert this "Bayer" image into a 4:2:2 Color image.
- Note: you will find the IP core documentation useful for details as you prototype your conversation algorithm.
- Provide your Matlab Prototype software and your original RGB image, corresponding Bayer image, and final output of your conversion algorithm in a folder named `part5/`

B) Create a software implementation for the Bayer color filter array in function *camera_loop(), based on your Matlab algorithm prototype*. Describe in your writeup what changes you made, and save a copy of any files modified (presumably only `camera_app.c`) during this process into a folder named `part5/`. While you have already derived a pseudocode implementation for this operation, there are several complicating details:

- Although we are streaming 16-bit values to the VDMA (to be processed via software), based on our system configuration above, only every other byte represents the vita camera output.
- The output of your Bayer pattern will be an RGB image, presumably with a 24-bit pixel representation (since you can directly capture the 8-bit R, G, or B component). However, both the VDMA and the HDMI output is configured to use 16-bit pixels. Specifically, the HDMI is expecting 16-bit values in a 4:2:2 YCbCr pattern.
- YCbCr is a family of color spaces used as part of the color image pipeline in video and digital photography systems. The 'Y' component corresponds to the relative luminance, with 'Cb' and 'Cr' corresponding to the blue-difference and red-difference chroma components. Note that YCbCr is

not a color space in the strict sense that RGB is; it is more of an encoding scheme for a color space (such as RGB). The matrix equation for this conversion is given as follows:

$$[Y \quad Cb \quad Cr] = \begin{bmatrix} 0.183 & 0.614 & 0.062 \\ -0.101 & -0.338 & 0.439 \\ 0.439 & -0.399 & -0.040 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}$$

- The YCbCr 4:2:2 pattern is an example of an encoding scheme referred to as chroma subsampling: http://en.wikipedia.org/wiki/Chroma_subsampling#4:2:2. Because the human visual system is less sensitive to the position and motion of color than it is to luminance, bandwidth can be optimized by storing more luminance detail than color detail. Look at the VDMA initialization code in function *fmc_imageon_enable()*, and infer from the Red, Green, and Blue examples how the 16-bit 4:2:2 YCbCr format is encoded. Briefly describe this in your writeup, and use this format as the output of your *camera_loop()* conversion pass.

In your writeup, describe the performance of your software-based color conversion (in terms of frames per second), and how you measured it. Overall this is a non-trivial piece of software, so put in a good faith effort for this part and in your writeup, describe your testing methodology. If you get really stuck, fork your project so that you can continue to work on the remaining system design parts.

6) Image Processing Pipeline. Although there are all sorts of software optimizations that can be applied to the color filter array in the previous section, the overall performance will likely remain insufficient for our digital camera needs. Fortunately for us, we can build a hardware image processing pipeline that should be able to keep up with the input and output throughput requirements. Comment out the relevant *camera_loop()* code, and switch back to VIVADO.

The three cores you will need to integrate into your project are the "Sensor Demosaic" (*v_demosaic*), the "RGB to YCrCb Color Space Conversion" (*v_rgb2ycrcb*), and the "Video Processing Subsystem" (*v_proc_ss*). Note the YCrCb format as opposed to YCbCr. Some important information:

- The image pipeline should be the following:
  vita -> vid_in -> demosaic -> rgb2ycrcb -> proc_ss -> axis_subset_converter -> vdma_S2MM -> vdma_MM2S -> vid_out -> hdmi_out.
  Provide a diagram for this awesome pipeline in your writeup, making sure to label the bit width of the relevant signals.
- It is recommended that you configure each of these new cores properly before attempting to connect them up. If a particular setting isn't mentioned in this guide, use your best judgment based on your existing design.
- Ensure the Sensor Demosaic core Interpolation Method is set to Fringe Tolerant Interpolation with the Horizontal Zipper Artifact Removal unchecked.
- For the RGBtoYCrCb component, select "0 to 255 for Computer Graphics" as the input range, and the "HD_ITU_709__1125_NTSC" option for the standard selection (obviously). Also, enable the AXI register interface.
- We will be using the Video Processing Subsystem as a Chroma Resampler tool. Thus, change the Video Processing Functionality to 422-444 Chroma Resampling only. This greatly simplifies the IP core. Change the Chroma Resampler algorithm to Predefined.

Do not run Connection Automation just yet. Next, connect all *aclk*/*ap_clk* pins from your new cores to the clock signal used for the *vid_in* core (*aclk*). You should do something similar for the *aresentn*/*ap_rst_n* signals.

Next, we should fix those *s_axi* signals on the RGB2YCrCb IP. Connect these three signals to the same signals that are going to the corresponding inputs on the *v_tc*.

Now, feel free to connect the video pipeline portion of these cores (labeled as *m_axis_video, video_in, s_axis, m_axis*, etc.) Your new cores should be fully connected, with the exception of the CTRL bus.

Save your block design and run Tools -> Validate Design. You shouldn't have any fatal errors (critical warnings are fine at this point. We're not done yet!) This part can be tricky, as we need to rely on the Connection Automation system to do its job. Run the Connection Automation, and check the box for all three cores to be connected. You may have noticed that the v_rgb2ycrcb core operates with two separate clock rates, but the *v_demosaic* and *v_proc_ss* cores only have one input for a clock. In order for these two IP cores to operate at the 148MHz video clock rate while operating on the 76MHz AXI bus, the Connection Automation tool needs to add a few AXI Clock Converters to cross the clock domain.

- If everything works correctly, you should be able to Validate your design again (this is important, as it forces Vivado to "notice" the clocking changes). There may still be warnings about other unrelated things.
- If you're interested in checking if this worked correctly, you can view these newly added Clock Converters in the massive *ps7_0_axi_periph* block. Expand this block and examine the *m0x_couplers*, where *x* is the connection number. Expanding some of these will reveal that the two couplers connected to our two IP blocks with the clocking issue should have Clock Converters in them, while the others have nothing in them. How neat is that.

We are getting close to being ready to generate a bitstream, but let's check some bit widths first. Your *axis_subset_converter* should be set to Auto for all signal properties, except for the Master TDATA width - this should be 2 bytes. After fixing these settings, you should be able to Validate Design again and note that the 24 bits of video data coming out of the v_proc_ss are converted down to 16 bits and fed into the VDMA.

Build that Bitstream, Export that Hardware, and Launch that SDK! Most of the image pipeline-related code is already provided for you in the camera_app project, so just uncomment out the headers, configuration data, and pipeline enable calls in camera_app.c, camera_app.h, and fmc_imageon_utils.c. You will have to investigate the initialization code for a few of the new cores you've added, and write in some configuration that is missing. In your write up, describe the performance of your image processing pipeline (in terms of frames per second), and how you measured it.

7) Making the Camera. At this point you've put in a considerable amount of effort, but the current system only implements a video pass-through. Create a new function called *camera_interface()* which adds the following user interface functionality:

1. Pressing the middle button should capture the current frame as a raw image. Store up to 32 images (in memory), and when a new image is captured, it should be displayed on the screen for 2 seconds.
2. Have one of the switches activate playback mode. In this mode, the left and right buttons rotate through the previously captured images.

Provide a copy of any modified code for this section in a folder named `part7/`.

What to submit: a .zip file containing 1) A cleaned archive of your xpr project, 2) Your updated design_1_wrapper.vhd and zedboard_fmc_imageon_gs.xdc files, 3) your Matlab prototype for color conversion (including the image you converted), 4) your modified source files (the previously mentioned directories with changes to `camera_app.c` and `fmc_imageon_utils.c`), and 5) your writeup in PDF format containing the highlighted sections of this document. In the Canvas submission, list each team member with a percentage of their overall effort on MP-2 (with percentages summing to 100%).

What to demo: at least one group member must be available to demo the current state of your implementation. A full demo score requires a system that can capture images using the completed hardware pipeline, but partial credit will be given for effort. Be prepared to briefly discuss your source code.

BONUS credit. MP-2 has two separate bonus point criteria. The first is for extra camera *features*. Consider what additional features a typical point-and-shoot camera has over our simple MP-2 implementation. Some possible examples (and their bonus point worth):

- A video mode, which records and can replay up to 5 seconds of 1080p video. (10 bonus points).
- A digital zoom mode, which uses the up and down buttons to zoom in and out of the current scene. (10 bonus points).
- Various analog and digital adjustments for the gain, exposure, and other common user-configurable digital camera settings. (2 bonus points each: up to 8pts)

The second MP-2 bonus point criterion is additional image processing *pipeline stages*. Similar to the steps followed in part 6), to be eligible for bonus points each new pipeline stage will need a comparison between a software and hardware-based implementation. Of particular interest is edge detection using Sobel-based or Laplacian-based filters for which VIVADO has an appropriate core: (http://en.wikipedia.org/wiki/Sobel_operator, http://en.wikipedia.org/wiki/Discrete_Laplace_operator) (25 bonus points).

Each group is limited to 100 bonus points for the entire semester.