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 Embedded Development Kit, 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 peripheral configuration as well as for reading from an IPMI identification EEPROM

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 XPS 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 occasionally you will need to re-download the FPGA bitstream and/or reboot the ZedBoard.

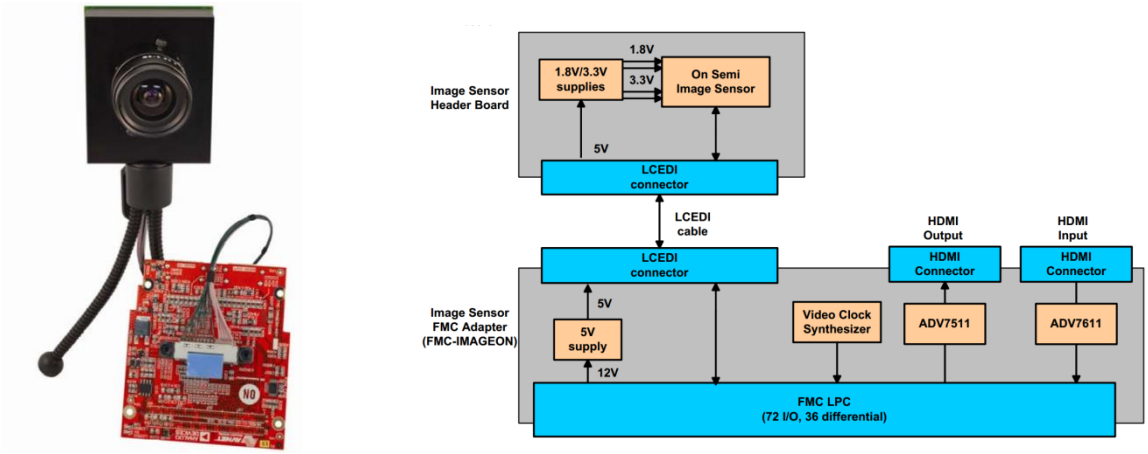

Figure: ON Semiconductor Image Sensor with HDMI Input/Output FMC Bundle – photo and block diagram. They're quite expensive so please don't break it. Pretty pretty please.

Given the complexity of this assignment, we have provided a starter XPS project that you can use as the baseline for implementing the digital camera functionality. Perform a check-out and peruse the directory structure:

```
>> cd cpre488
>> svn checkout https://source.ece.iastate.edu/svn/cpre488/MP-2
```

After sourcing the `setup.sh` file (as always), open up the XPS project and get to work:

```
>> xps system/system.xmp &
```

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 select the "Export to SDK" option and choose to "Export and Launch SDK" – but don't walk away! While the system is being exported, analyze the design using the various System Assembly views and the Graphical Design view, as well as directly through the `system.mhs` and `system.ucf` files. 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 XPS 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).

When the FPGA synthesis is complete and the SDK Export operation is ready, make sure to select your `MP-2/sw` directory as the target SDK project directory. The FMC-IMAGEON comes with several software drivers that we are making use of, and to allow SDK to use them, select the "Xilinx Tools -> Repositories" option, and add a new local repository that points to the `MP-2/repository/ProcessorIPLib` directory. Also, we have provided two software projects you will need to import using the "File -> Import -> General -> Existing Projects Into Workspace" option: `system_bsp` and `camera_app`.

Note that we are intentionally using older versions of some of the XPS IP cores (both hardware and software), so do not attempt to remove / re-add them, as there is no guarantee that the `camera_app` framework will still function correctly.

Download the `system.bit` 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 `xtpg_app.c` 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 XPS, it is time to interact with the Image Sensor on the FMC-IMAGEON board. The appropriate IP core for this purpose is found under "Project Peripheral Repository" and is called *FMC_IMAGEON_VITA_RECEIVER*. The good news is you only have to change 1 option when configuring the core – set the XSVI_DATA_WIDTH to 8. The bad news is that the rest of this configuration is not trivial, and the vita receiver core does not come with any documentation (which is frankly ridiculous). Consequently the following steps are not especially intuitive, so work with the TA if you get stuck:

1. The component is not provided with an address range on the main AXI bus, so manually set C_BASEADDR to 0x52100000 and C_HIGHADDR to 0x5210FFFF.
2. The external IO ports are automatically made external, although they are not constrained. Uncomment out the commented placement, IO standard, and timing-related lines in your `system.ucf`. Say a quiet "thank you" to your instructor who went without sleep for nearly two weeks to sort these out. Note that several of these ports 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.
3. The vita receiver requires 4 different clocks: the AXI bus clock for reading slave registers (which should already be connected), a 200 MHz clock connected to the "clk200" port, a 37.125 MHz clock connected to the "clk" port, and a 148.5 MHz clock connected to the "clk4x" port. Two of these are

generated by the *clock_generator* core, and the other is directly generated via the Processing System.

4. The output enabled port, named "oe", requires a software-accessible wired connection. Use a signal called "fmc_enable", which is connected to the Gpo port of the ZedBoard I2C controller component. This mapping on the I2C side has to be done directly in the `system.mhs` file, so comment the last line of the `IIC_FMC` component and uncomment the previous line which provides that mapping. Walk through the `camera_app` application. In your writeup, describe where in software this output enable is set. Be specific.
5. Ports "reset" and "trigger1" can be connected to ground. Port "fsync" and all the debug ports do not need to be connected.
6. The vita receiver has to be connected to the *vid_in_axi4s* module. Disconnect the sync, blank, and video data ports on the vid_in that are currently connected to the test pattern generator, and connect them to the equivalent signals coming from the vita. The names do not exactly match up, but it should be clear what goes where.
7. Finally, the vid_in and the *axi_vdma* module are expecting a 16-bit video data stream, while our vita is only generating 8-bit pixels. For this reason, for now, pad the incoming data to the vid_in with the binary string "10000000". Syntactically, in your `system.mhs` file, prepend `0b10000000 &` to the current signal being mapped to the vid_data port on your vid_in module. '`&`' is the concatenation operation in this context. Make sure that the data width configuration for the vid_in is still 16-bits, as depending on the order in which you added signals this may have changed.

Confirm that your design passes the design rule check (click the "Run DRCs" button, this is in general a good idea every time you modify your system), and then Export back into 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. Create a software implementation for the Bayer color filter array in function *camera_loop()*. 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 XPS.

The three cores you will need to integrate to your project are the "Color Filter Array Interpoloation" (*v_cfa*), the "RGB to YCrCb Color Space Conversion" (*v_rgb2ycrcb*), and "Chroma Resampler" (*v_cresample*). Note the YCrCb format as opposed to YCbCr. At this point in the semester, you should be sufficiently experienced in XPS to be able to integrate these cores without a detailed walkthrough. Some gotchas to look out for:

- We still want a 16-bit VDMA and HDMI output, but since the *v_cfa* core is applying the Bayer filter for us, we can convert the *vid_in* back to using an 8-bit video data interface.
- These streaming video components all require a separate data clk (labelled "aclk" in the port description). Connect these to the 148.5 MHz clock we are already using. Depending on the order you connect your cores, these "aclk" ports are sometimes hidden. When asking questions on Blackboard, copy/pasting the relevant subsections of your `system.mhs` is fair game (but not the entire file).
- The image pipeline should proceed from vita -> vid_in -> cfa -> rgb2ycrcb -> cresample -> 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.
- You'll need to generate AXI addresses for these cores, similar to what you did for the vita interface in part 4). While 64K should be enough for anyone, make sure your cores don't overlap with other reserved address spaces.

- The default options for *v_cfa* should suffice.
- For the *v_rgb2ycrcb* 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.
- For the *v_cresample* component, we are converting from 4:4:4 (slave side) to 4:2:2 (master side). For the "Resampling Filter Type", select "Drop / Replicate Samples".

After your design passes the rule checks, export back to 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 the chroma resampler, but it should follow a similar pattern. In your writeup, 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 your modified source files (your final `system.mhs` and `system.ucf`, as well as the previously mentioned directories with changes to `camera_app.c` and `fmc_imageon_utils.c`) and your writeup in PDF format containing the highlighted sections of this document. In the Blackboard 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)

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 XPS 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.