CprE 488 – Embedded Systems Design

MP-2: Digital Camera

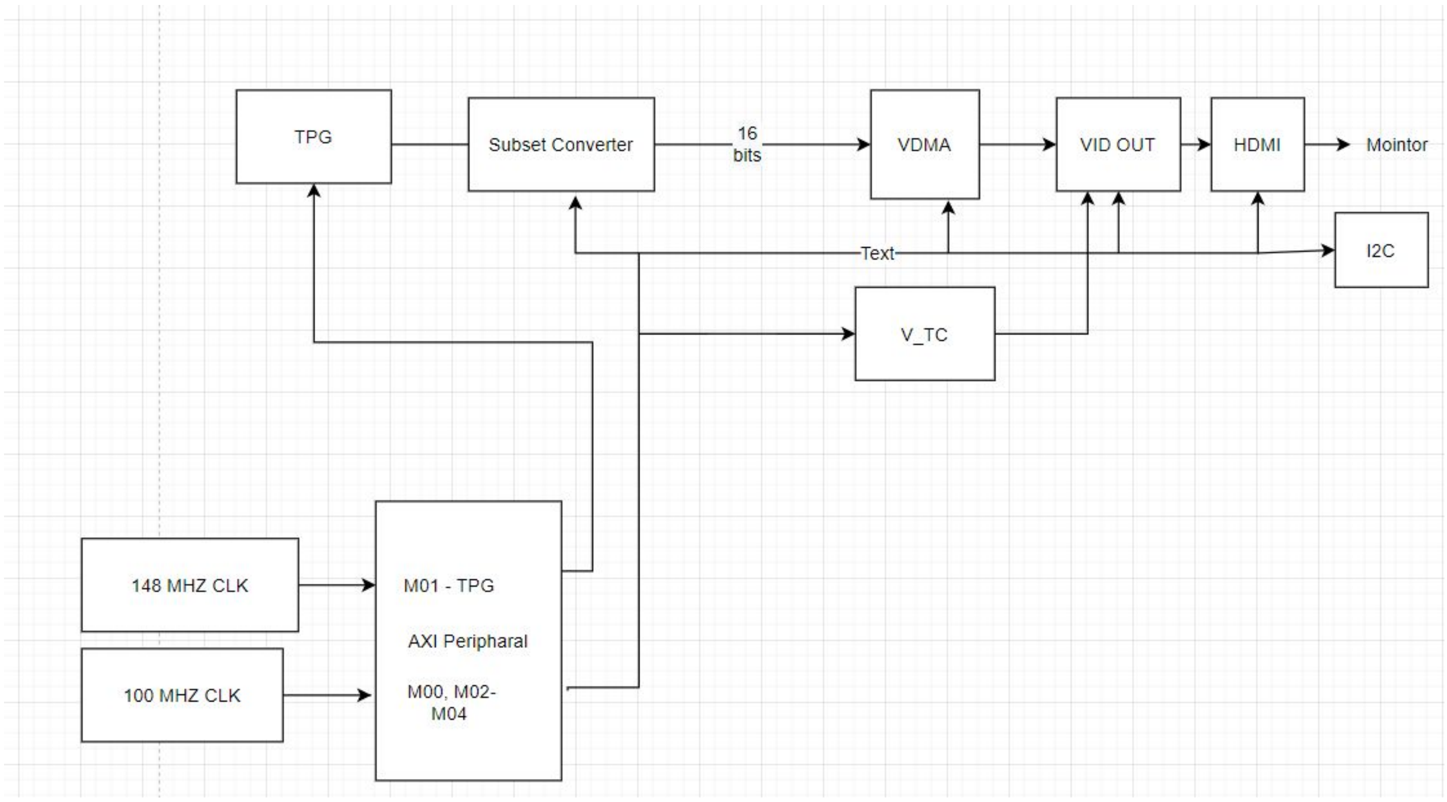
Group: 2C

Amith Kopparapu Venkata Boja

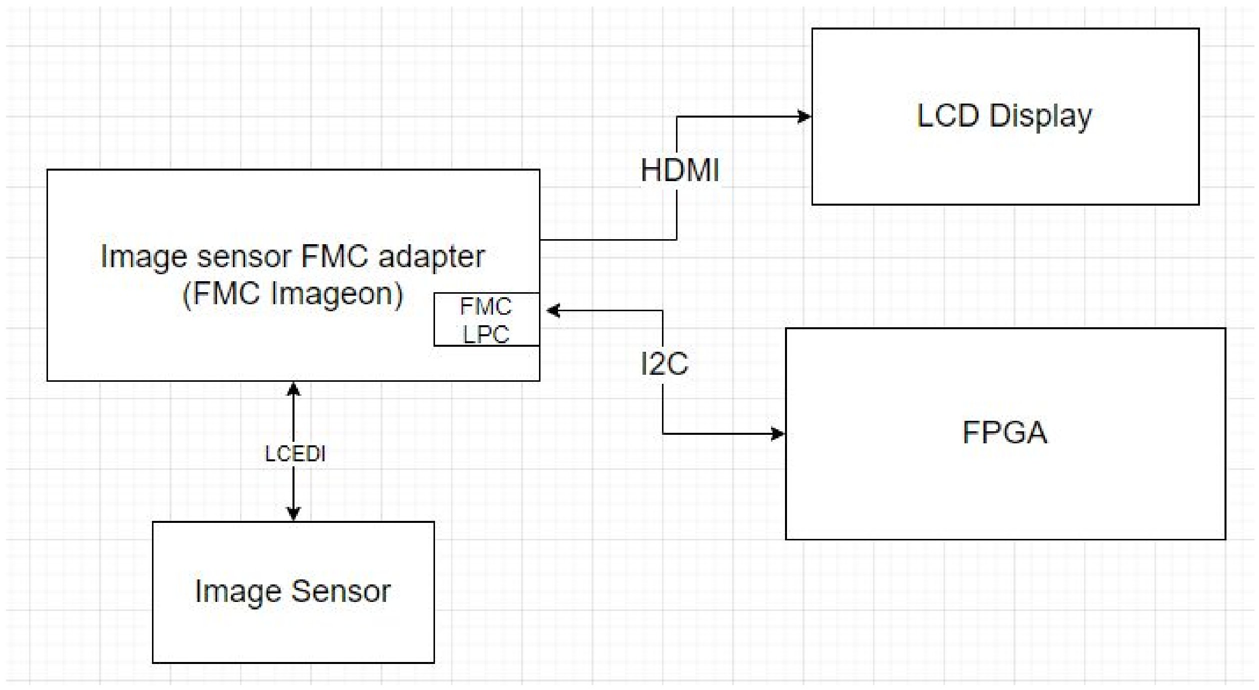
Vignesh Krishnan

Archit Joshi

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).



IP core level connection diagram



Board level connection diagram

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).

The FMC-IMAGEON interfaces the FPGA with display and with the camera sensor.
The camera is connected to FMC via the SPI interface. The camera provides a grayscale image taken by the Bayer filter. The FPGA reconstructs the RGB picture and converts it to YCbCr format before sending it through the HDMI to the display.
In this MP, VDMA is conducting two roles simultaneously. It gets input in the form of YCbCr (16-bit) from the camera processing done in hardware. VDMA stores these values in the memory by sending the data to the Zynq Processing System. VDMA also sends the video signals to Video out to be displayed through HDMI. In MP-0, VDMA only read the data already stored in the memory but here it reads as well as writes. Vtc provides timing references to video out like Tsync same as MP-0 except that here Vtc, VDMA and Video out use 148 MHz instead of 100 MHz.
FMC-IMAGEON uses I2C to perform these operations:

1. I2C interface 1: Selection of EEPROM
2. I2C interface 2: Peripheral configuration like HDMI i/p, o/p, video clock synthesizer

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/.

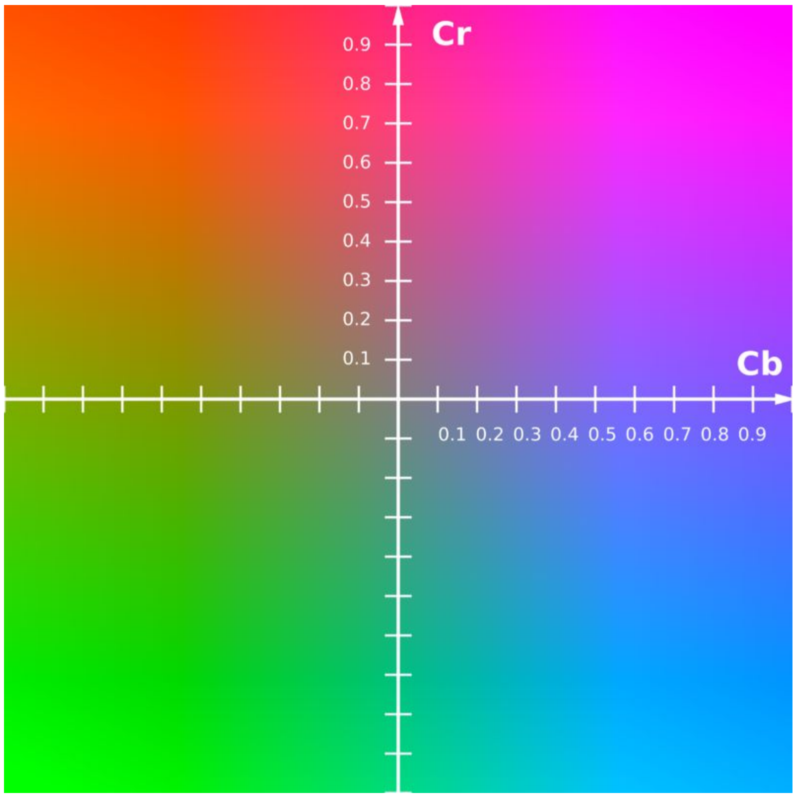
For the camera loop, we made modifications to the data being sent to pMM2S_Mem. We hardcoded arbitrary colors that we want in YCbCr by sending consecutive pixels to pMM2S_Mem to account for Cb and Cr values being sent in alternating pixels. For the TPG core we changed the background pattern components to output different solid colors. This change was made in the fmc_imageon_enable_tpg function of fmc_imageon_utils.c, specifically modifying the BACKGROUND_PATTERN_ID register.

In your writeup, briefly describe what this pairing of signals signifies, and what this configuration is typically used for.

The VITA Camera has 4 data channels and video data width of 8. Vita receiver serial interface works on LVDS (Low Voltage Differential Signals). The pairing of _p and _n signifies the positive and negative signals of LVDS. There are 4 bits of positive and negative data bits.

Explain why this is an appropriate value to append, and why appending "00000000" would not make sense.

The lower 8 bits are appended to convert grayscale images from the camera to YCbCr format while keeping the pixels gray. YCbCr is a 16-bit frame in which Y is b[15 to 8] and Cb (or Cr in alternate frames) is at b[7 to 0]. When we append 8 bits to grayscale and make it 16-bit long, the grayscale values represent Y. When both Cb and Cr are 1000 0000, it gives black to white shades (midpoint of the photo below) depending on Y, which preserves the grayscale image while converting it to YCbCr format.
If we append 0000 0000, it will make Cb, Cr represent green color shades. (bottom left corner of the photo below). Similarly, if we append 1111 1111, that will give shades of pink.



In your writeup, briefly explain why the camera at this stage is not outputting any color.

The camera has an array of sensors, in which every sensor captures the intensity of a fixed color. The sensors have a specific pattern like RGGB in a 2*2 square to create a Bayer filter. The camera only sends the intensity values (in grayscale) at each pixel in the array captured by the respective sensors.

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/.

We changed the direct assignment of the S2MM pointer to the MM2S pointer. We got the grayscale values coming from S2MM, converted it into an RGB image using our matlab RGGB bayer filter algorithm. This gave us RGB values for each pixel. We then used these RGB values and converted them with the conversion matrix to the YCbCr 4:2:2 values. Finally, we wrote to two 16 bit memory spaces(MM2S) at a time, because in 4:2:2 format, the first 16 bit value contains Cb and Y and the second contains Cr and Y, with Cb and Cr changing on alternate pixels.

Briefly describe this in your writeup, and use this format as the output of your camera_loop() conversion pass.

For the output of our YCbCr 4:2:2 we already had Y, Cb, and Cr arrays from our RGB to YCbCr conversion. We took these arrays and used bit shifting and bitwise operations to get them into the correct output format. The first 16 bit value sent to pMM2S_Mem contained Y and Cb and the second Y and Cr. This sequence of CbY then CrY is because of the RGGB bayer filter format. For, BGGR this would be flipped, which is what we did at first.

The Cr and Cb values change on alternating pixels and the Y values are the same for each pair of two pixels.

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.

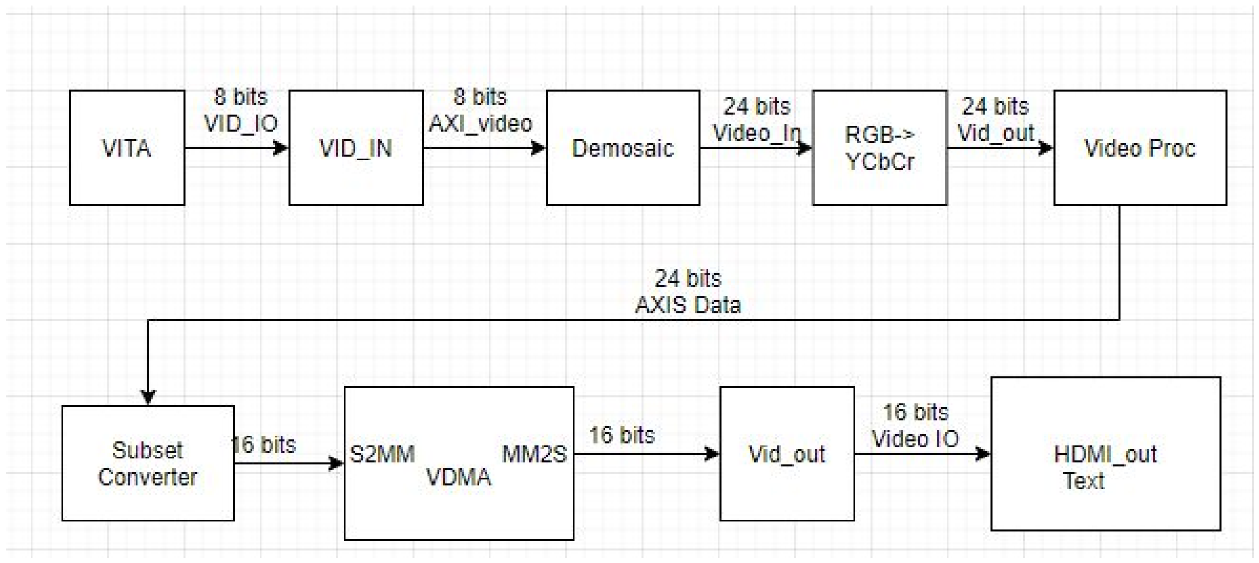
Our software based implementation was 0.54788 fps. We used the XTime to record the start time in the beginning of the frame, and the end time at the end of a frame. This gave us the number of cycles counts in 10 frames. Following is the formula we use to get fps,

```
10.0 * COUNTS_PER_SECOND / (end_t-start_t)
```

This gave us our fps which is very slow. It makes sense because of all the processing the needs to be done for every frame.

While writing the software, we first started by displaying colors to the output to get a feel for how YCbCr is actually being outputted to HDMI and how we will have to format our data to match that. In testing, we implemented our matlab prototype and realized we were getting flipped Cb and Cr values because our image had a blue hue. We realized that we created our bayer filter in matlab and software for a BGGR sensor, while the actual sensor on the camera is RGGB, so then we had to change our code bit to match that. After that, we had no other issues.

Provide a diagram for this awesome pipeline in your writeup, making sure to label the bit width of the relevant signals.



In your writeup, describe the performance of your image processing pipeline (in terms of frames per second), and how you measured it.

Our image processing pipeline ran at 59.867 fps. We calculated this using the control and status registers in VDMA. In the control register, there is a field to set a frame counter that acts as a down counter to 0. We set this field to its max value, 255, so we can measure the time it takes for 255 frames to be processed. We also enabled the FrameCntEn bit which will set the halted bit in status register to 1 when the down counter reaches 0, this way we can check when 255 frames have been processed. Then, we used XTime to wait for the halted bit to be set to high, which indicates that we have captured the time it took for 255 frames to process. We used this time and formula below to calculate our FPS.

FPS = (255.0 / ((end_t - start_t) / COUNTS_PER_SECOND))

The image processing pipeline has a much faster framerate, showing that hardware performs much better than software. This code is implemented in a print_frame_rate() function that is within camera_loop.

We ran into a lot of issues with this implementation because we tried using the FrameCntIRQ trigger first to trigger an interrupt out when the counter reaches 0. This never worked for us for some reason, so we decided to use the FrameCntEn and halted bit. In our calculation we had some problems in our types which caused our frame rate to be rounded to a whole number, but we noticed that quickly and type casted as needed.

Part 7:

For our capturing and playback, we implemented our code in the camera_interface function. For capturing, we stored our 1080*1920 pixels in a pointer array of length 32. For playing this back, we simply iterated that specific element in the recordings array back into pMM2S_Mem by means of a for loop. This method was not the most efficient, as it decreased our great hardware frame rate, but it was still much better than the software implementation's frame rate.