Using the HDMI Video Platform

The HDMI Video Platform is based on the Zynq Base TRD 2013.4, and uses some of the PL resources to implement HDMI input and output functionality. The HDMI Video Platform provides the following capabilities:

- One HDMI input channel through the HDMI input connector on the Avnet ImageON FMC card.
- One HDMI output channel through the HDMI connector on the zc702 board.
- The video input chain writes frames into a circular buffer with space for three frames, and the video output chain reads from a circular buffer with space for three frames.
- Software applications and hardware accelerators can access the input and output video through these frame buffers.
- The only supported video format for the input and output frames is 1080p, 60 fps (i.e., 1080 lines per frame and 1920 pixels per line at 60 frames per second) in RGB format.
- Each pixel is represented by a 4-byte value.
- Each line starts at a 4Kbyte address boundary.
- A software *sync* mechanism is provided to allow the software application or accelerator to advance to the next input/output frame when it is available. If the software application or the hardware accelerator completes before the next input frame is available, the software sync mechanism waits until the frame is available.
- If the software application or the hardware accelerator operates at a rate slower than the input video frame rate, some input frames are dropped, and some output frames are re-displayed while waiting for the application to produce the output frame and move on to the next input frame

Structure of an Application Using the Video Platform

Several sample applications using the video platform are provided in the `samples/zc702_hdmi_apps` directory. All the samples have the same directory structure:

- `src`: The top level code for the application is here

- `main()` calls a function called `thread_sw_sync()`, which manages the frame buffers and the software synchronization mechanism and calls a function called `colorflip_processing()` or `sw_sobel_processing()` with pointers to one or two input frames and one output frame. The software synchronization loop looks as follows:

```
while(1){
            cvc_index++; cvc_index %= MAX_BUFFER;
                  …
                  setCVC_TPGBuffer(cvc_index,tpg_index); // softw sync
                  …
                  sobel_in_index++; sobel_in_index  %= MAX_BUFFER;
                  …
sw_sobel_processing( // accelerator or sw call
virt_sob_buff[sobel_prev_index],// prev frame
virt_sob_buff[sobel_in_index], // curr frame
virt_vmem[sobel_out_index]); // out frame
      }
```

- `colorflip_processing()` or `sw_sobel_processing()` calls a function called `img_process()` with the buffer pointers and a few scalar parameters.

- `hw`: The function to be accelerated is here.
  - `img_process()` is the function to be accelerated.
  - In the `colorflip` and `simple_sobel` examples, this is code that can be compiled as software using the **`makefiles`** in the `DebugSw` directory, or compiled as a HW accelerator using the **`makefiles`** in the `Release` directory.
  - In the `manr_sobel`, and `motion_demo` examples, this code contains Vivado® HLS library elements for synthesis, and is not likely to work in the pure software mode.
- `DebugSw`: Contains **`makefiles`** and a sub-structure for compiling the code in `src` and `hw` into a single software-only executable. This directory structure and the **`makefiles`** contained in it are similar to the **`makefiles`** that are automatically generated by the SDSoC GUI or Xilinx SDK, and use `arm-xilinx-linux-gnueabi-gcc` as the compiler.

Note: Some examples use Vivado HLS library elements for improved synthesis results and the `DebugSw` directory is not provided for these examples

- `Release`: Contains **`makefiles`** and a sub-structure for compiling the code in `src` and `hw` into a hardware accelerated application using the SDSoC design environment. The **`makefiles`** were created by copying over the **`makefiles`** from `Debug_sw` and replacing `arm-xilinx-linux-gnueabi-gcc` with `sdscc`, and a few other changes.

Structure of a Sample Video Accelerator

In this section, we examine the structure of a sample video accelerator that flips the green and red bytes. Each pixel consists of the following bytes <x, R, G, B> going from the most significant byte to the least significant byte, where x is ignored, and R, G, B correspond to Red, Green, and Blue pixel values.

```
void img_process (int rgb_in[PIXELS_PER_FRAME],
                  int rgb_out[PIXELS_PER_FRAME], …) {
#pragma AP INTERFACE ap_fifo port=rgb_in
#pragma AP INTERFACE ap_fifo port=rgb_out
```

The `img_process()` function has two parameters representing the input and output frames as arrays of integers. Each integer represents the 4-byte quantity consisting of the R, G, B values as described above. The pragmas define the arrays to be `ap_fifo` interfaces meaning that the frames are streamed-in or streamed-out one pixel (4 bytes) at a time. Note that the programmable logic portion of the Zynq-7000 device or most other FPGAs does not have sufficient RAM to store an entire frame of 1920x1080 pixels. Therefore, video accelerators typically stream the data to/from external memory and store just a few lines at a time within the PL if necessary, as shown in the more complex examples.

Structure of Data-motion Network Generated for Applications with Multiple Accelerators

Two of the video examples contain multiple accelerators. The `manr_sobel` example contains the following code fragment, illustrating calls to multiple accelerators:

```
unsigned short yc_data_prev[NUMROWS*NUMCOLS],
         yc_data_in[NUMROWS*NUMCOLS],
         yc_manr_out[NUMROWS*NUMCOLS],
         yc_sobel_out[NUMROWS*NUMCOLS];
void img_process(unsigned int *rgb_data_prev,
                 unsigned int *rgb_data_in,
                 unsigned int *rgb_data_out, …)
{
    rgb_pad2ycbcr(rgb_data_prev, yc_data_prev);
    rgb_pad2ycbcr(rgb_data_in, yc_data_in);
    manr((char)param1, yc_data_prev, yc_data_in, yc_manr_out);
    sobel_filter(yc_manr_out, yc_sobel_out);
    ycbcr2rgb_pad(yc_sobel_out, rgb_data_out);
}
```

Each function call here is mapped to an accelerator. SDSoC analyzes the above code and determines that the outputs of some accelerators are connected directly to the inputs of other accelerators, and hence, it generates a data-motion network that implements direct streaming connections from one accelerator to another. For example, the `yc_data_prev` output of `rgb_pad2ycbcr()` is connected directly to the input of `manr()`.