Document for a General FPGA Platform

I. Introduction
This platform is used to speed up the verification process of image or video related IPs.

In general case, to verify an image processing design, designers need to:
1. prepare a source image in external memories
2. launch the design
3. fetch the sink image from external memories

It seems like a simple task, however, by step 1, designers may need to prepare:
   a. an off-chip camera like D5M
   b. an I2C or other similar communication IPs to configure the camera
      (It is decided by the communication interface adopted by the camera you choose)
   c. a software or hardware driver for the camera
      (If it is a software driver, a CPU is also needed)
   d. an interface to dump the source images

By step 2, designers may need to prepare:
   a. an interface to fetch the source images and dump the sink images.

By step 3, designers may need to prepare:
   a. a display channel like VGA
   b. a software or hardware for the channel
      (If it is a software driver, a CPU is also needed)
   c. an interface to fetch the sink images

The above tasks may be a little difficult for designers who mainly concentrate on IP-level designs because a bunch of background knowledge in system design (AXI Bus, GM/S interface, on-chip CPU, bare machine programming …) and peripherals integration (I2C, UART, D5M, VGA …) are needed. In addition to this, IPs like image stitching, depth extracting, image dehazing need some specific source images which can hardly captured by general cameras.

In view of these, I established a general FPGA platform for verification use. Aiming at this target, this platform has the following features:
   a. a simple method to prepare source images
   b. a simple interface to access external memories
   c. a simple method to fetch sink images
   d. NOT suitable for demo use because the source images are not updated in real time

The above content gives a brief introduction and motivation to this platform while the rest of this document is composed of three questions:
How to prepare a source images?
How to access the source images in external memories?
How to fetch the sink images?
II. How to prepare a source images?
A. Backgrounds
   By using control panel, one could easily access any resources on the DE2_115 board.
   However, the address mapping relationship of control panel is different from our SDRAM controller. In order to put source images in the wanted position in a wanted way, I wrote a script named "jpg2dat.m". It is located in "/fpga/ALTERA_DE2/test_GFP_display/scripts/". The input is "img_i.jpg". (Of course, you could replace it with any supported image format and name) The output is "img_o.dat", which can be recognized by control panel.

B. Using Examples
   To prepare source images,
   1. Prepare a source image and put it in "/fpga/ALTERA_DE2/test_GFP_display/scripts/", for example,
2. If needed, change the parameter in "jpg2dat.m" according to image size, for example,
   ```matlab
   HON = 640;
   VER = 480;
   ``
   If needed, change the input and output target, for example,
   ```matlab
   INPUT = 'img_i.jpg';
   OUTPUT = 'img_o.dat';
   ```

3. Run the scripts in Matlab
4. Dump it into SDRAM by control panel.
   Address box should be 0
   Check the file length box

5. Then it is dumped into SDRAM, with a base address of 0.
   From designers’ point of view, it is arranged as follows:

   ![Diagram of mapping relationship between pictures and SDRAM]

   In this figure, one black square stands for one pixel, which contains three color components red, green and blue. All these color components are 8 bits in data width. In another word, one pixel can be represented with 24 bits, however, to align the access address, it is stored with 32 bits, namely, 1 word.

   Now we assume the total amount of pixels is N, then from designers’ point of view, these pixels are arranged in the memory space from word address 0 to word address N. For example, if one need to read the blue component of the 94\textsuperscript{th} pixels in the source pixels, the access address would be exactly 94 and the third byte of the data fetched would be the blue component.

   Here, word address means each address is counted in words instead of in bytes. To translate word address 94 into byte address, just times it to 4. However, one needs to pay attention to the endian problem.
III. How to access the source images in external memories?

A. Architecture, I/O and Timing

After the above operations, source images are already dumped into external memories. However, one still need a bus system, an sdram controller and an interface to access these images. In our platform, these components are packaged into one block. Thus, for designers, the only cared thing is the timing of the read and write interface provided by this platform. Nevertheless, I will still provide you with the inside architecture.

![Architecture of GFP_process](image)

**Table III-1. Main part of GFP_process**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus System</td>
<td>AXI bus</td>
</tr>
<tr>
<td>GM_RD</td>
<td>general AXI master for read</td>
</tr>
<tr>
<td>GM_WR</td>
<td>general AXI master for write</td>
</tr>
<tr>
<td>WRAPPER</td>
<td>convert simple rd/wr to general AXI format rd/wr</td>
</tr>
<tr>
<td>SDRAM CNTL</td>
<td>sdram controller with AHB interface</td>
</tr>
</tbody>
</table>

**Table III-2. I/O of GFP_process**

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>I/O</th>
<th>Viewer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>board_clk</td>
<td>1</td>
<td>I</td>
<td>whole system</td>
<td>board clk</td>
</tr>
<tr>
<td>board_rstn</td>
<td>1</td>
<td>I</td>
<td>whole system</td>
<td>board reset, low valid</td>
</tr>
<tr>
<td>sys_clk</td>
<td>1</td>
<td>O</td>
<td>whole system</td>
<td>system clk</td>
</tr>
<tr>
<td>sys_rst</td>
<td>1</td>
<td>O</td>
<td>whole system</td>
<td>system reset, high valid</td>
</tr>
<tr>
<td>rd_ena_i</td>
<td>1</td>
<td>I</td>
<td>read master</td>
<td>read enable</td>
</tr>
<tr>
<td>rd_adr_i</td>
<td>32</td>
<td>I</td>
<td>read master</td>
<td>read address</td>
</tr>
<tr>
<td>rd_ack_o</td>
<td>1</td>
<td>O</td>
<td>read master</td>
<td>read acknowledge</td>
</tr>
<tr>
<td>rd_dat_o</td>
<td>64</td>
<td>O</td>
<td>read master</td>
<td>read data</td>
</tr>
<tr>
<td>wr_ena_i</td>
<td>1</td>
<td>I</td>
<td>write master</td>
<td>write enable</td>
</tr>
<tr>
<td>wr_adr_i</td>
<td>32</td>
<td>I</td>
<td>write master</td>
<td>write address</td>
</tr>
<tr>
<td>wr_dat_i</td>
<td>64</td>
<td>I</td>
<td>write master</td>
<td>write data</td>
</tr>
<tr>
<td>wr_byt_i</td>
<td>8</td>
<td>I</td>
<td>write master</td>
<td>write byte enable</td>
</tr>
<tr>
<td>wr_ack_o</td>
<td>1</td>
<td>O</td>
<td>write master</td>
<td>write acknowledge</td>
</tr>
<tr>
<td>sdram_clk</td>
<td>1</td>
<td>O</td>
<td>sdram</td>
<td>sdram clk</td>
</tr>
<tr>
<td>sdram_addr</td>
<td>32</td>
<td>O</td>
<td>sdram</td>
<td>sdram address</td>
</tr>
<tr>
<td>sdram_dq</td>
<td>4</td>
<td>IO</td>
<td>sdram</td>
<td>sdram data</td>
</tr>
</tbody>
</table>
The related design files are located in "/rtl" and "/lib"
Top module is
"/rtl/GFP_process.v"
Sub modules include:
design files in "/rtl/GFP_axi"
design files in "/rtl/GFP_clkgen"
design files in "/lib/ALTERA_DE2/pll"
design files in "/lib/ALTERA_DE2/tri"

B. Using Example
The architecture of this example is shown in the following figure.

![Architecture of test_GFP_process](image)

Fig. III-4. Architecture of test_GFP_process

It fulfill a task of reading a 640×480 source images located in word address 0, inversing it, and then storing it to word
address 640×480. I establish a simulation environment in
"/sim/rtl_GFP_process"
and a corresponding FPGA project in
"/fpga/ALTERA_DE2/test_GFP_process"

To run the simulation example,
1. upload whole project onto the server
2. change directory to "/sim/rtl_GFP_process"
3. run "make ncsim" in terminal
4. open the waveform located in "/sim/rtl_GFP_process/simul_data" with Verdi, you will see,

![Waveform Image]

Source images is initialized by the following statements in "/sim/rtl_GFP_process/simul_data/tb_GFP.v", namely, the testbench.

```v
25 define PROG_FILE "./scripts/img_o.dat"
26 parameter MEMSIZE = 640*480-1

146 reg [31 : 0] dat_32bit;
147 integer mem_cnt;
148 integer ram0_fp;
149 integer ram0_tp;
150 initial begin
151 ram0_fp = $open("PROG_FILE","rb");
152 for(mem_cnt=0; mem_cnt<(MEMSIZE); mem_cnt=mem_cnt+1) begin
153 ram0_tp = $readmemb(dat_32bit, ram0_fp);
154 if(def_BIG_ENDIAN_MIF)
155 u_m481c4m6a2.bank0[mem_cnt] = [dat_32bit[23:16], dat_32bit[31:24]];
156 u_m481c4m6a2.bank0[mem_cnt] = [dat_32bit[07:00], dat_32bit[00:15]];
157 else
158 u_m481c4m6a2.bank0[mem_cnt] = [dat_32bit[15:08], dat_32bit[07:00]];
159 u_m481c4m6a2.bank0[mem_cnt] = [dat_32bit[31:24], dat_32bit[23:16]];
160 end
161 end
162 // valid when use HIGHER ahh_address_bits for sdram_bank
```

In other words, you need to prepare the PROG_FILE, "./scripts/img_o.dat" and change MEMSIZE according to size of the source image.

To prepare the PROG_FILE, you just follow a similar way to the one in Section II.
1. Prepare a source image and put it in "/sim/rtl_GFP_process/scripts"
2. If needed, change the parameter in "jpg2dat.m" according to image size, for example,
   HON = 640;
   VER = 480;
   If needed, change the input and output target, for example,
   INPUT = 'img_i.jpg';
   OUTPUT = 'img_o.dat';
3. Run the scripts in Matlab

To run the FPGA example,
1. Program "/fpga/ALTERA_DE2/test_GFP_process/outputfiles/test_GFP.sof" to DE2_115
2. Press KEY2, you will see LEDG0 is on, which indicated the inverse task is done.

In the next Section, I will show how to fetch the sink image.
IV. How to fetch the sink images?
There are two different ways to fetch the sink images. One is using control panel, while the other one is using VGA.

A. Using Example
To fetch sink images by VGA,
1. Program "/fpga/ALTERA_DE2/test_GFP_display/outputfiles/test_GFP.sof" to DE2_115
2. Turn on SW0, you will see,

In fact, if you turn on SW1, another module will be launched to write over this image,

In case you are interested in the inner logic, I will provide the architecture of this example here,

Fig. IV-1. Architecture of test_GFP_display
If you changed the parameter RGB_ADR from "640*480*4" to "0", and regenerate the sof file again, it will display the source images, since you put it there. This parameter is located in "/rtl/GFP_vga/vga_axi_if/vga_axi_if.v". Attention should be paid that this parameter is expressed in byte address.

Although, we can directly see it on displayer by using VGA, the image size is limited to the bandwidth of SDRAM. And, for now, I just build a 640×480 example. If we want to test other image sizes, we should use control panel.

To fetch sink images by control panel,
1. Fetch sink images by control panel, store it as "/fpga/ALTERA_DE2/test_GFP_process/scripts/img_i.dat"
   Address Box should be 12C000
   Length Box should be 12C000
2. If needed, change the parameter in "dat2jpg.m" according to image size, for example,
   HON = 640;
   VER = 480;
3. Run "dat2jpg.m" in Matlab, you will see

Attention should be paid on the following staffs.
1. It can be observed that there are several abnormal dots in the fetched images. This is because SDRAM will lose
data during the programming process.

2. Here, 12C000 is expressed in half-word address, which means the corresponding byte address is 258000. The byte address for a 640×480 images is 640×480×4, namely, 12C000 in hexadecimal. It is not equal because of the mapping relationship of our SDRAM controller is different from control panel. I will try to fix it later. Nevertheless, this platform is still useable.

3. THIS PLATFORM IS NOT FULLY TESTED, PLEASE CONTACT ME IF THERE IS ANYTHING WRONG!
V. Version

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2015.09.11</td>
<td></td>
<td>Leilei Huang</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>