ORGFX - getting started

Per Lenander, Anton Fosselius

June 4, 2012

1 Introduction

This document is intended to make it easier to get started with the ORSoC graphics accelerator component. It contains some useful links and instructions on how to get the example code working.

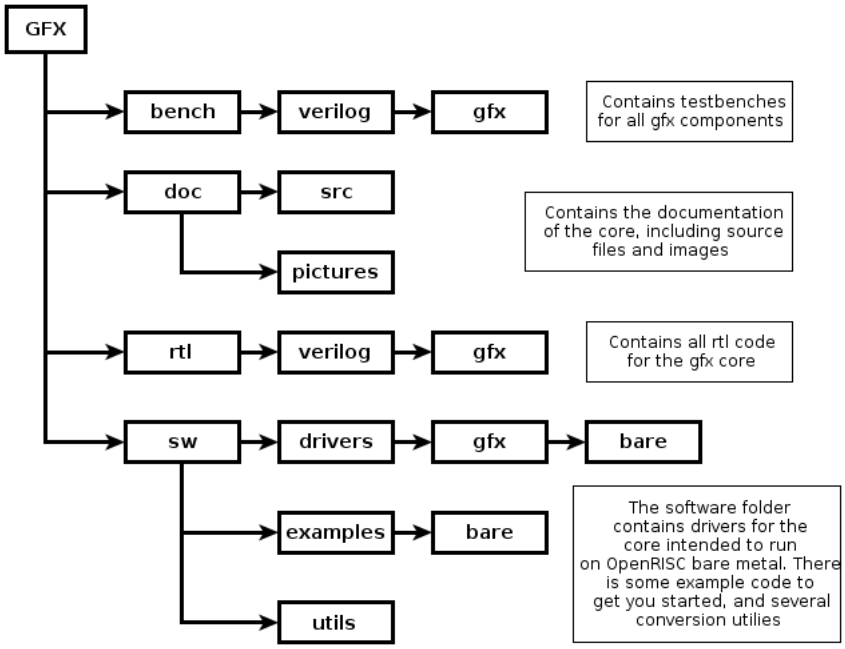

Figure 1: Directory structure of the ORSoC graphics accelerator.

2 Project structure

The project source is structured as seen in figure 1.

3 Hardware

The **rtl** folder contains all the code needed to instantiate the ORGFX IP Core.

The component has 3 main wishbone interfaces:

- A wishbone slave interface for reading and writing registers. Every registry write will be added to a circular FIFO instruction queue.
- A wishbone master interface for reading from memory (requires at least a read-only memory).
- A wishbone master interface for writing to memory (requires at least a write-only memory).

All functionality of the component is accessed by writing instructions to the registers. Only writes to the control register can potentially start drawing operations.

Refer to the specification document for a full explanation of all registers, their default values and how they are used.

The component can be modified to extend the size of the instruction queue by changing the *fifo_depth* parameter. It is not advised to change the other parameters, as their function is not fully tested.

4 Software

4.1 Drivers

ORGFX comes with a layered software driver in the **sw/drivers** folder.

The basic driver consists of:

```
orgfx.h
orgfx.c
orgfx_regs.h
```

It is possible to set the base bus address of the graphics accelerator in the regs file. The **orgfx.c** file is the only part of the driver that can interact with the hardware.

The extended driver (convenience functions and better surface management) consists of:

```
orgfx_plus.h
orgfx_plus.c
```

The extended driver is required for most of the advanced API. Memory is managed statically in the driver, so the driver must be modified if support for a larger number of surfaces is desired.

There are four sets of advanced API that all build on the functionality of the basic and the extended driver:

```
orgfx_3d.h
orgfx_3d.c
orgfx_tileset.h
orgfx_tileset.c
orgfx_bitmap_font.h
orgfx_bitmap_font.c
orgfx_vector_font.h
orgfx_vector_font.c
```

These supply convenience functions for doing more complex operations in a small number of function calls (such as drawing a 3D mesh). Most of these should be used with the output of the *utilities* described below.

4.2 Examples

In addition to the drivers, the **sw** folder contains **examples** and **utils**. The examples contain a few sample implementations with some basic functionality. The examples can be built by using the supplied Makefile. By default it builds the examples for hardware (requires or32-elf- toolchain to build for OpenRISC. The Makefile assumes that the or32-elf tool **bin2binsizeword** is in the path).

4.3 Utilities

There are four useful utilities supplied with ORGFX: the spritemaker, meshmaker, bitfontmaker and vector font maker programs. Build all the utilities with the Makefile in the **utils** directory.

All the utilities take some form of input file and generates a header file containing all the information needed for the drivers. These files also contains a function for initialization specific to the resource contained. Calling this function creates the necessary structure that the drivers can use.

The spritemaker program takes an image *filename.png* and creates an output file *filename.png.h* (the program also support various other file formats[1]).

```
./spritemaker filename.<jpg/png/bmp> [bpp=16]
```

The meshmaker program takes a Wavefront .obj file *filename.obj* and generates the file *filename.obj.h*.

```
./meshmaker filename.obj [bpp=16]
```

The bitfontmaker program takes an image containing a 256 character bitmap font where each letter is *glyphsize* pixels large. Th program takes an image *filename.png* and generates *filename_font.h* (supports the same image formats that spritemaker does).

```
./bitfontmaker filename.<jpg/png/bmp> [bpp=16] [glyphsize=32]
```

The fonter program takes a vector font *filename.ttf* and generates *filename_font.h*.

```
./fonter filename.ttf
```

5 Emulator

Along with the hardware implementation, there is a software implementation of the graphics accelerator written in SDL. The software implementation consists of the file:

```
orgfx_sw.c
```

and replaces the files:

```
orgfx.c
orgfx_plus.c
```

From the examples folder, type:

[1] http://www.libsdl.org/projects/SDL_image/

make sw

This command will generate a software implementation of all the examples (requires the SDL and SDL_image libraries). While the implementation is not a cycle-for-cycle perfect emulation, it does make it significantly easier and faster to develop applications for the device.

6 Example implementation

To follow the instructions you will need to have Xilinx ISE installed and have a Digilent ATLYS board available. You will also need Git and the OpenRISC toolchain. Git can easily be installed in Debian based Linux distributions like Ubuntu with the following command:

sudo apt-get install git

The example implementation based on ORPSOCv2 can be found on:

https://github.com/maidenone/ORGFXSoC

To build the project you will also need to download the ORGFX project located at:

http://opencores.org/project,orsoc_graphics_accelerator

First checkout the code from github with the following command:

git checkout git://github.com/maidenone/ORGFXSoC.git

Now move into the folder of the ORGFX project and edit the make file located in sw/examples/bare:

cd PATH_TO/orsoc_graphics_accelerator/trunk/sw/examples/bare/

Use gedit or a texteditor of your choice to edit the Makefile

gedit Makefile

Now replace the path to the example implementation from the path:

../../../../orpsocv2/boards/xilinx/atlys/backend/par/run/

To the same path in your ORGFXSoC folder.
Now build the example code in sw/examples/bare/ with make

make

Move to the following directory in your ORGFXSoC directory:

cd orpsocv2/boards/xilinx/atlys/backend/par/run

and finally run make

make

Now a .mcs file have been generated, Now program your atlys board with Digilent Adept. Chose the Flash tab and in the "FPGA programming file" field browse for the .mcs file that the makefile generated. The mcs file will be located in the par/run folder.
Hit the program button and you are done.