CSE467: Project Phase 1 - Building the Framebuffer, Z-buffer, and Display Interfaces

Vincent Lee, Mark Wyse, Mark Oskin

Winter 2015
Design Doc Due Saturday Jan. 24 @ 11:59pm
Design Review Due Tuesday Jan. 27
Code Due Tuesday Feb. 3
Demo Due Thursday Feb. 5

1 Introduction

Now that you have a better understanding of how the graphics pipeline works and you have implemented a model of each pipeline stage, you will start implementing the GPU in hardware. Your first task will be implementing the z-buffer, frame buffer, and VGA interface. The rest of your pipeline will remain in software and communicate to the hardware over a memory-mapped interface. In following phases, you will implement the programmable GPU cores and move more logic into hardware.

2 DE1-SoC Overview

Starting with this phase, we will be using an Altera DE1-SoC board (Revision D). The centerpiece of this board is a Programmable System-on-Chip (SoC) that contains an ARM Cortex A9 Hard Processor System (HPS) and an Altera Cyclone V FPGA on the same chip. Figure 2 depicts most of the peripherals you can access from the SoC.

The HPS and FPGA are tightly coupled through an interconnet network, which allows them to share peripherals and data. The peripherals most important to us are the VGA port and 64MB SDRAM that are connected to the FPGA fabric. You may also be interested in using the 7-Segment Displays, Buttons, and 10 LEDs for debugging since they are directly connected to the FPGA fabric and easy to use. The following sections will provide basic details on the VGA Output and SDRAM. For more extensive information on the board, please consult the DE1-SoC User Manual (Rev C/D), linked below in Additional Resources. Section 3.6 provides details on the FPGA peripherals that you will be using:

2.1 VGA Display Core

The DE1-Soc Board has a 15-pin D-SUB connector setup for VGA output. The VGA signals are directly exposed to the FPGA fabric on the SoC, and you will be implementing a hardware module to communicate over this interface. An overview of the FPGA to VGA DAC interface that you will be connecting to is shown in Figure 2.1. The VGA protocol consists of several communication signals, but is actually relatively simple.

Note that you will have to use the specific VGA display clock which operates at 25.125 MHz. If you choose to use a higher frequency clock for the rest of your application logic you will need to employ clock domain crossing FIFOs.

A decent description of the VGA signaling and timing can be found at:

`http://javiervalcarce.eu/wiki/VGA_Video_Signal_Format_and_Timing_Specifications`

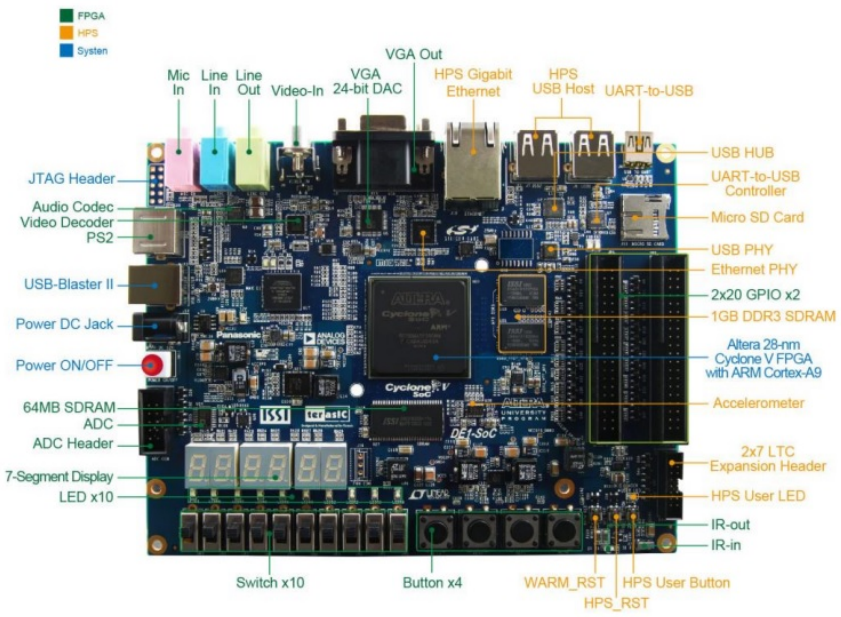

Figure 1: DE1-SoC Board

2.2 FPGA SDRAM

Also attached to the FPGA fabric is an externally located 64MB SDRAM (32Mx16) memory chip. This memory chip is connected to the FPGA via a 16-bit data bus, control lines, and 13-bit address bus. This memory will hold your frame buffer(s) and z-buffer. The FPGA to SDRAM interface is displayed in Figure 2.2. The timing and control signals for memory chips is significantly more complicated than those for a VGA connection. Therefore, you will be using a pre-made SDRAM controller module from the DE1-SoC example code and will be responsible for connecting your logic to the much simpler interface exposed by the SDRAM controller.

3 Design Tools

This section will provide a very brief overview of the design tools you will be using for the remainder of the project. The first tool you will be working with is Quartus II v14.1, which manages the SoC project. The second tool you will use is the Altera SoC EDS, an Eclipse based IDE for compiling your C code to run on the ARM HPS. Both of these tools should be installed on the lab bench machines. They are also available for download if you wish to run them on your personal machines.

3.1 Quartus II

Quartus II is a hardware development tool allowing integration of the FPGA logic, HPS, and peripherals (VGA, SDRAM, etc.). Quartus II is similar to many IDE's you have likely worked in, except that your programming will be done in Verilog RTL to describe hardware. We do not have an exhaustive tutorial for using Quartus but we will provide some basic pointers below. We will also provide a skeleton project with the VGA and SDRAM interfaces exposed, the SDRAM controller code, and an HPS-to-FPGA interface that we suggest you start from. You do not have to start from these files but we recommend that you do.

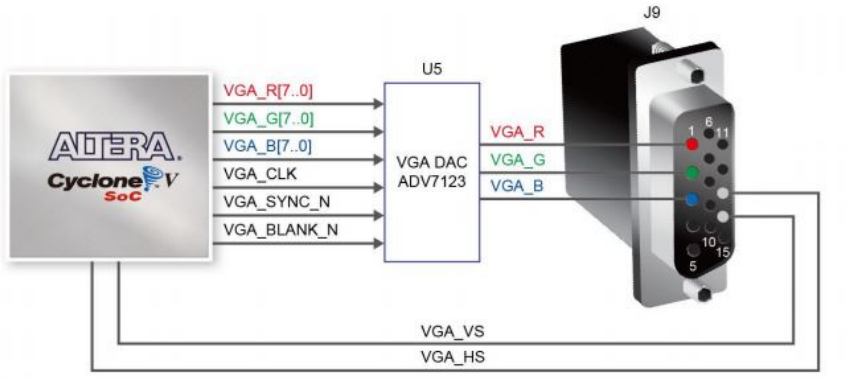

Figure 3-21 VGA Connections between FPGA and VGA

Figure 2: VGA Display Interface between FPGA and Display Connector

3.1.1 Basic Layout

The basic layout of Quartus II is shown in Figure 4. There are a few windows or "Views" of note that are provided by default.

The "Project Navigator" located in the top left is used to track all fiiles associated with the project. Here you will find several tabs: "Hierarchy", "Files", "Design Units", "IP Components", and "Revisions". Roughly speaking the "Hierarchy" tab shows the module instantiation tree from your top level mode (kind of like a call graph but with verilog RTL modules). The "Files" tab tracks all the files associated with this project including the .sdc file and other files you may add. The "Design Units" tab tracks the modules you have added to the design. The "IP Components" tab tracks the modules that have been added to your project using the IP core generator which we will discuss later.

The "Tasks" panel access to tool flow executions such as compilation, RTL simulation, etc. This is where you will fire off compilations of your design when you are ready to put them on the board. Note that there are several other flows in the drop down menu for the "Flow" option which allow execution of different tool flows.

The "IP Catalog" panel is the core generator menu which is used to generate stock IP cores such as RAMs, DSPs or FIFOs so that you don't have to build these primitives yourself. These will be useful in this phase of the project as we you will need to synthesize memories for your buffers. The generation process will be explained briefly later.

The middle section of the Quartus tool is usually used as the workspace where you can view files and develop your Verilog modules.

If any of these views mysteriously disappear, you can revive them by going to "View→Utility Windows".

3.1.2 Developing in Quartus

The Quartus development environment is similar to your standard IDE minus the fact that you're designing in Verilog RTL. We encourage you to consult the Quartus User Manual if you encounter roadblocks which can be found at:

`https://courses.cs.washington.edu/courses/cse467/15wi/docs/Quartus_II_Intro.pdf`

Figure 3: 64MB SDRAM FPGA Interface

3.1.3 Qsys

Qsys is a system integration tool available within Quartus II. Qsys is the tool used to instantiate connections between the HPS and FPGA fabric in your design. Building system components in Qsys allows the tool to handle details of component interconnect and lets you integrate components at a higher level of abstraction. The skeleton project contains an HPS system, clock generator, and an Avalon Memory-Mapped (Avalon-MM) FIFO in a Qsys system. The Avalon-MM FIFO's input side is memory-mapped into the HPS address space, and it's output is exposed to the FPGA fabric logic. This allows direct communication from code running on the HPS to the logic in the FPGA fabric you will be implementing. Please refer to Chapter 5 of the Quartus II Handbook Volume 1: Design and Synthesis.

The Quartus II Handbook can be found at:

`https://courses.cs.washington.edu/courses/cse467/15wi/docs/Quartus_II_Handbook.pdf`

3.1.4 Generating IP Cores

Over the course of the project, you may need to generate primitives such as large memories, FIFOs, and DSP cores within the FPGA fabric. Most FPGA toolchains have specialized hardened or automatically generated core options for these primitives so you don't have to build them yourself. The Quartus toolchain provides a core generator for these primitives cores in the "IP Catalog" view. The IP Catalog contains core generators for these primitive cores; you simply need to provide the core parameters and the generator will build the core for you. Note that these IP cores are different than what you can instantiate in the Qsys system described above.

To initialize this process, expand the IP Catalog and navigate to the core that you want to generate. You can also type text into the search box to search the available IP, such as entering "fifo" to easily access the FIFO IP. Select the core you want and press the "Add" button. You will be prompted where you want to save the resulting core file that you generate, it might help to have an IP directory in your project directory. Once you enter this information, a core generator wizard will appear and guide you through the rest of the core generation steps. When you finish, you should end up with a new core in your "IP Components" tab of the "Project Navigator". You can now freely instantiate the IP core in your design.

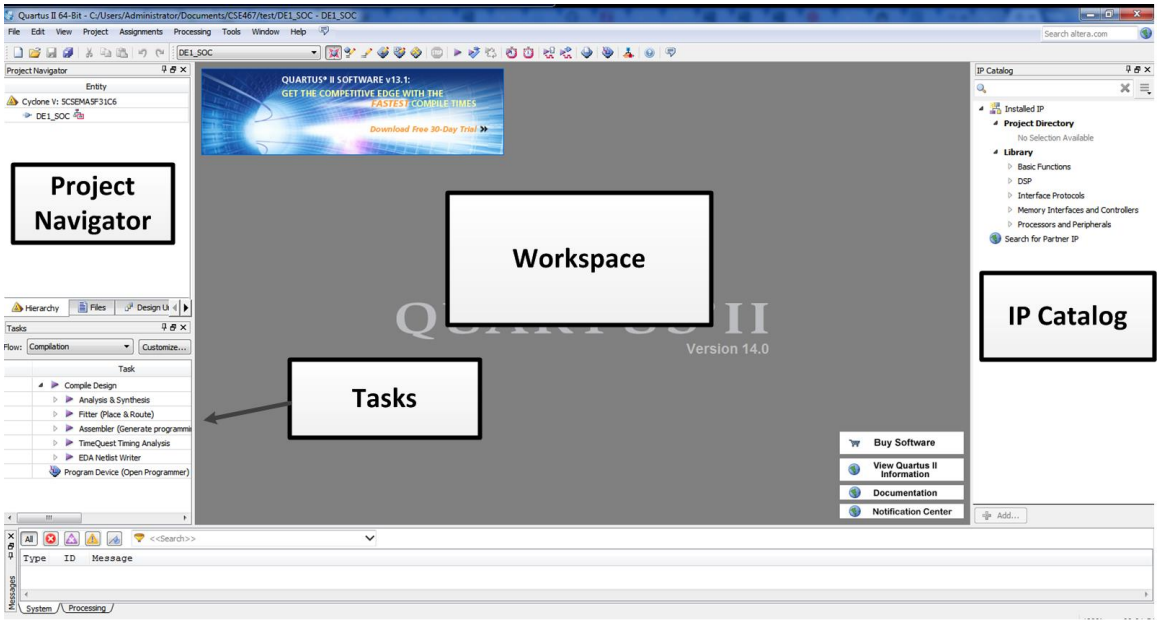

Figure 4: Initial Quartus Project Configuration

3.2 Altera SoC EDS

The other tool you will use is the Altera SoC Embedded Design Suite (EDS). The SoC EDS includes the ARM Development Studio 5 (DS-5) Eclipse based IDE for software development and cross-compilation. We will be using ARM DS-5 to develop and compile code that will run in Linux on the HPS side of the system. Refer to Chapter 3 of the My First HPS-FPGA document found in the DE1-SoC CD-ROM download found on the DE1-SoC Resources link below. ARM DS-5 also contains many features that you will not need. Further details on the Altera SoC EDS can be found at the link in "Additional Resources" below. The SoC EDS user guide will likely be very helpful, especially Chapter 5 detailing ARM DS-5 Altera Edition.

3.2.1 Communicating with the FPGA

Communication between the HPS and FPGA will be handled through memory mapped connections. Specifically, the skeleton project comes with an Avalon-MM FIFO that has its input exposed as a memory mapped device in the HPS side. Its output is exposed from the Qsys system to the rest of the FPGA fabric, and has a standard FIFO interface. A special header file that is automatically generated from the Quartus project specifies the address of the FIFO. To use the interface, you just need to include this header file and then mmap() it into your code. Please refer to the examples in the DE1-SoC CD-ROM download for examples on communicating over memory mapped interfaces.

4 Additional Resources

If at any point in the project the documents we provide are inadequate, you can most likely find the answers in one of the Altera manuals for the DE1-SoC board. Please consult the manuals before asking the TAs for assistance.

4.1 Documentation

We are using board revision D which you will need to download the appropriate packages. The DE1-SoC User Manual contains detailed information about the board including pin assignments which you may need to adjust the constraints file. The DE1-SoC Getting Started Guide should contain the instructions required to program the SD card to run Linux on the board. Follow the instructions in section 5 of the Getting Started Guide to get your Linux distribution up and running. The DE1-SoC CD-ROM (Rev C/D) contains many examples of using the HPS and FPGA components on the board.

4.2 Working Remotely

The Altera Quartus tool chain and Altera SoC EDS are available for download online from the Altera website. To use the limited version of Quartus for lower end boards such as the Cyclone V we are using, the web version should be sufficient. Once you obtain the software, you should be able to install it on your local machine and be good to go since it doesn't require a license (only the more heavy duty FPGAs require that). You may also need to install USB-Blaster drivers; instructions for which can be found in the DE1-SoC Getting Started Guide.

4.3 Links

- DE1-SoC Resources (Rev C/D): `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=836&PartNo=4`
- Alter SoC EDS Downloads: `http://dl.altera.com/soceds/`
- Quartus II Downloads: `http://dl.altera.com/?edition=web`

5 Getting Started

5.1 Loading the Skeleton Project

To load the skeleton code and project first download the project archive file from the course website and place it in your workspace directory. Then launch Quartus II and open the project by choosing "Restore Archived Project..." under the "Project" menu item. Select the .qar (Quartus Project Archive) you downloaded earlier and choose a location to restore the project to.

5.2 Adding Source

Adding source can be achieved by either creating new source in Quartus or adding existing Verilog RTL files to your project. New source can be created through "File→New..." and existing source can be added under "Project→Add/Remove Files in Project..."

5.3 Compiling Your Design

To compile your Quartus design, select "Processing→Start Compilation". This will kick off the compilation process. It may take some time for compilation to complete, especially as your design grows.

5.4 Programming the Board

Compilation will generate a ".sof" file that can then be programmed onto the board. Programming the board begins by selecting "Tools→Programmer". This opens the Programmer window. Assuming the DE1-SoC Board is plugged in to your machine, it should auto-detect the board. If it does not, you may have to click on "Hardware Setup.." and select the DE1-SoC from the dropdown. Then return to the main Programmer

window and select the "Auto Detect" button. This will display a pop-up dialog where you should select "". Add your ".sof" file by right clicking on the FPGA device and choosing "Edit→Change File" to navigate to and select your file. Finally, check the "Program/Configure" box and press "Start" to program the device.

5.5 Booting Linux & Transferring Files

The DE1-SoC board contains an ARM Cortex A9 processor that is capable of running Linux. We will provide microSD cards preloaded with a Linux image that runs on the board. Please refer to Chapter 5 of the DE1-SoC Getting Started Guide for a quick tutorial on booting Linux. You can skip Section 5.2 since we did that for you.

You will be booting into Linux to run code that communicates with the FPGA fabric. At the end of the project the communication will be loading the input object file into the FPGA memory. For now, you will be sending data to fill the frame buffer or z-buffer to be rendered over the VGA port.

6 Project Tasks

Now that you have some basic knowledge of the DE1-SoC system, we will detail your tasks for this phase of the project.

6.1 Z-buffers and Frame Buffers

The z-buffer and frame buffer are the final stages of the graphics pipeline that interface directly to the display core. As explained in the previous phase of the project, for a given pixel coordinate (x, y) the z-buffer is responsible for storing the depth of the pixel currently in the frame buffer at the coordinate (x, y). The frame buffer simply stores the pixel color values for each coordinate (x, y) in the frame. The values of the frame buffer are then read out by the display core and output to the screen. In current GPU systems, the z-buffering and frame buffering are implemented in hardware so we're going to do the same.

To implement each of these two buffers, you will need to use off-chip memory; in our case we will be using the 64MB SDRAM attached to the FPGA since the FPGA fabric itself does not have enough on-board memory. We are providing an SDRAM controller for you to use, but you will need to write code that transfers data from the HPS coming out of the Avalon-MM FIFO to the SDRAM.

Each frame is sized to 256 x 256 pixels x 16-bits per pixel but the VGA resolution we are using is 640 x 480. To make driving the VGA display reasonable, we recommend you allocate enough buffer space to support a full VGA frame as opposed to just the 256 x 256 frame. The z-buffer is only 8-bits per pixel, but has the same number of rows and columns as the frames. We recommend that you implement your z-buffer and frame buffer as separate modules to make your life easier.

It is up to you how you want to interface these modules with the GPU core which we will build later, but keep in mind that the framebuffer must adhere to the interface governed by the display core.

6.2 Design Documents

All teams will be required to submit a design document prior to design reviews. As with all large projects, it's always good to have a battle plan before you go to war with the FPGA. There is no required structure for the design document but it must address the following:

- Identify the design problem components that you are trying to solve - What are you building? What do each of the components do?
- Outline and present your solution to the problems identified in the previous step
- Explain why your solution is sufficient and correct
- An explanation of your testing strategy - both unit and integration test plans

- Supplement explanations of your proposed system solution with appropriate block diagrams - we expect to see at least a high level block diagram of each module you intend to build

Design documents will be graded on clarity, conciseness, and effort. Note we are not looking for perfect optimal solutions as there are multiple solutions for this project. We really just want you to think through the solution and put it on paper so that you have a plan when you go to build the design. The design document must be submitted to the Catalyst dropbox as a PDF document and reviewed by the TAs before the actual design review meeting.

Late design documents will be accepted up until 24 hours prior to the design review. If you are submitting the document late, you MUST email the design document to the TAs directly in addition to submitting it to Catalyst.

6.3 Design Reviews

All teams are required to schedule a design review with one of the TAs in addition to submitting a design document. During the design review, you will be asked to identify the problem components, present your solution, and justify why your solution is sufficient. You are free to reuse your design document, or bring additional presentation materials when explaining your solution. Design reviews will be 15 minutes long and be graded based on your teams ability to effectively communicate your solution.

6.4 Demonstration

You will be required to schedule a demonstartion with your TA to show a working solution (or just show us in lab sometime before the deadline). During this demonstration, you will need to show that your implementation works by displaying an image of your choice to the screen. If your implementation is not working, you will need to prove to us which components of the design work properly. This can be done either by showing us an alternate but convincing demonstration or by executing test harness simulations.

7 Deliverables

Your job is to build the z-buffer, and frame buffer cores, and then interface them to the display core. The deliverables for this project phase are as follows:

1. Submit a design document addressing each of the points in the "Design Documents" section prior to the design review.
2. Conduct a design review with one of the TAs prior to implementing the project
3. Schedule a demonstration with your TA to show your working z-buffer and frame buffer interfaced with the display core - the display should show something after you program the board and configure your design (however you do that is up to you)
4. Submit working code with all appropriate test harnesses and README files

8 Code Submission

You are required to submit a copy of your code to the Catalyst dropbox. This submission must be a tar file labeled as *project*_1_$*name*_1_$*name*_2.*tar* where *name*_ and *name*_2 are the UW ID's of each team member. If you are working alone, omit the second name. At the top level of this submission, you must also include a README file which contains the first and last names, student ID, and uw.edu emails of each team member. Only one member of the team is required to submit the tarball of the code.

9 Tips and Tricks

Here are a few tips and tricks the TAs think may be useful when completing this checkpoint:

1. Use hierarchy and abstraction. In hardware design these are really the only ways of managing complexity in a reasonable manner.
2. Make sure to implement thorough unit testing. If you build your design hierarchicially (which you should), it's very easy for one of the smaller components to break everything so it's important to make sure the smaller components work.
3. Use integration testing to test your design and prove to yourself that your design works. Integration tests don't always have to be in Verilog. In fact, it is sometimes easier to generate test vectors in a scripting language or from a model like the one you implemented in the previous phase of the project.
4. Run regression tests frequently. This can help you catch bugs before you implement too many changes which can make it difficult to track down which change broke your design.
5. Use version control and make sure to tag working commits. This can save you the trouble of having to manually look for a revertable commit in the event something goes wrong.
6. Take breaks. While it's tempting to try and throw a bunch of continuous hours into fixing that one bug, it sometimes helps to stop and come back to it when you feel like your spinning your wheels.
7. Extra pairs of eyes. As with all large projects, it can be useful to have an extra pair of eyes (partner or classmate) to help you find bugs that you might not have thought about yourself. Discussion solutions with another person can go a long way in helping you navigate some of the more nasty bugs that may arise.