CSEE 4840
Embedded System Design
Lab 3: Peripherals and Device Drivers

Stephen A. Edwards
Columbia University

2015

Implement on the FPGA a memory-mapped peripheral that can receive communication from the ARM processors on the Cyclone V. Communicate with your peripheral through a Linux userspace program that accesses a device driver you have written.

Your peripheral should display a ball on the VGA screen at coordinates given to it through software. Your device driver should implement an *ioctl* that takes a coordinate from the user and sends it to your peripheral

1 Introduction

In this lab, you will control your own hardware from your own software, communicating through a Linux device driver. We supply a base hardware design to extend, a Linux kernel and root filesystem, a working example of a VGA peripheral that you will have to modify, and a working device driver for the existing peripheral that you will have to adapt to work with your own peripheral.

You will implement a video bouncing ball in this setting. Your peripheral will generate an VGA raster consisting of a ball at a particular location, your userspace C program (software) will make this ball bounce around the screen, and your device driver will mediate between your program and your peripheral.

2 Add the VGA Component to the Base Design

In this section, you will tell Qsys about a new peripheral component, connect it ultimately to the ARM processors, and synthesize a new FPGA configuration bitstream.

Download *lab3-qsys.tar.gz* from the class website and unpack it on your workstation. Start Quartus and open the supplied *lab3.qpf* project. From within Quartus, start Qsys (Tools→Qsys) and open the *lab3.qsys* project.

Create a new VGA_LED component and connect it to the base design. In Qsys, select File→New Component.

Under the Component Type tab, set its name to *vga_led* and its display name to *VGA LED Emulator*.

Under the Files tab, under "Synthesis Files," add the *VGA_LED.sv* and *VGA_LED_Emulator.sv* files. The first file contains the code for the memory-mapped peripheral that drives the VGA raster generator in the second. Click on "Analyze Synthesis Files." The top-level module name should now be "VGA_LED."

2.1 *Assigning Signals to Channels*

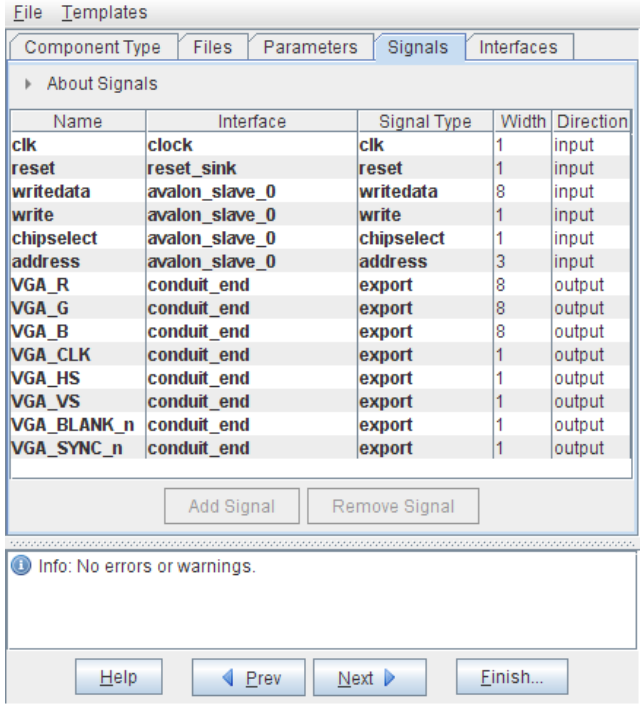
When Qsys analyzes the synthesis files, it makes some good guesses about the meaning of each signal on the peripheral, but it is not perfect. Below, you will fix these mistakes manually.

Under the Signals tab, create a new interface by selecting "New Reset Sink..." under the Interfaces column for the reset signal.

Create a new conduit by selecting "New Conduit..." under the Interfaces column for one of the VGA signals. Set the interface of each VGA signal to this new "conduit_end."

Set the Signal Type of each VGA signal to "export."

Your signals show now appear like the list on the right. If errors remain, the next steps should resolve them.



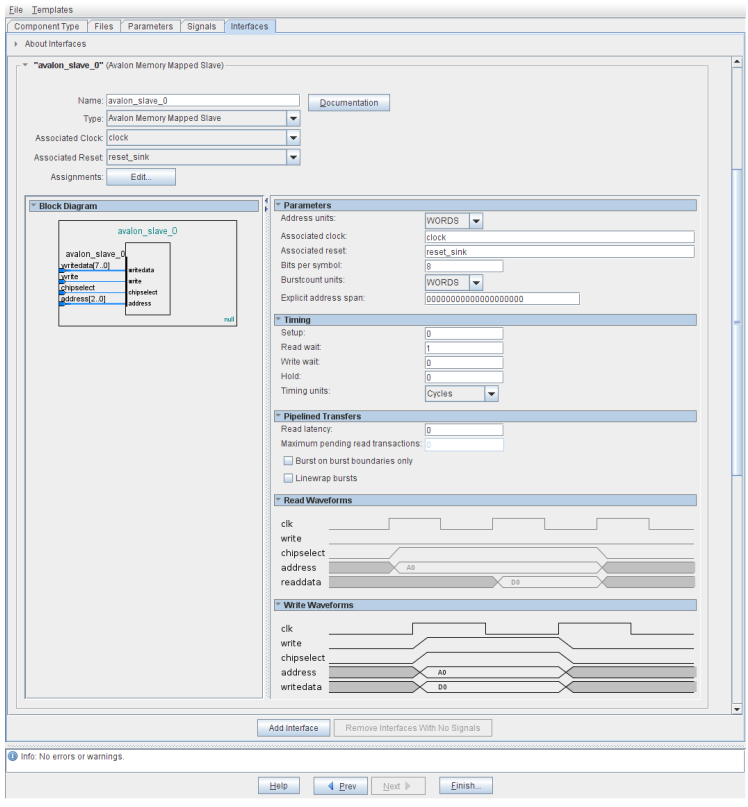
Under the Interfaces tab, click on "Remove Interfaces with no signals."

Set the associated clock of "reset_sink" to "clock."

Set the associated reset for "avalon_slave_0" to "reset_sink."

The *avalon_slave_0* interface should now appear as it does on the right.

Once you have resolved any errors or warnings, click on "Finish" and save the component. This creates the file *vga_led_hw.tcl* to record metadata about the new component.



2.2 *Connecting the VGA Component*

Now that Qsys knows about your custom component, you will connect it to the rest of your design.

In Qsys, add an instance of the new *VGA LED Emulator* component by selecting it under "Project" in the library and clicking on the green +. By default, it will be named *vga_led_0*.

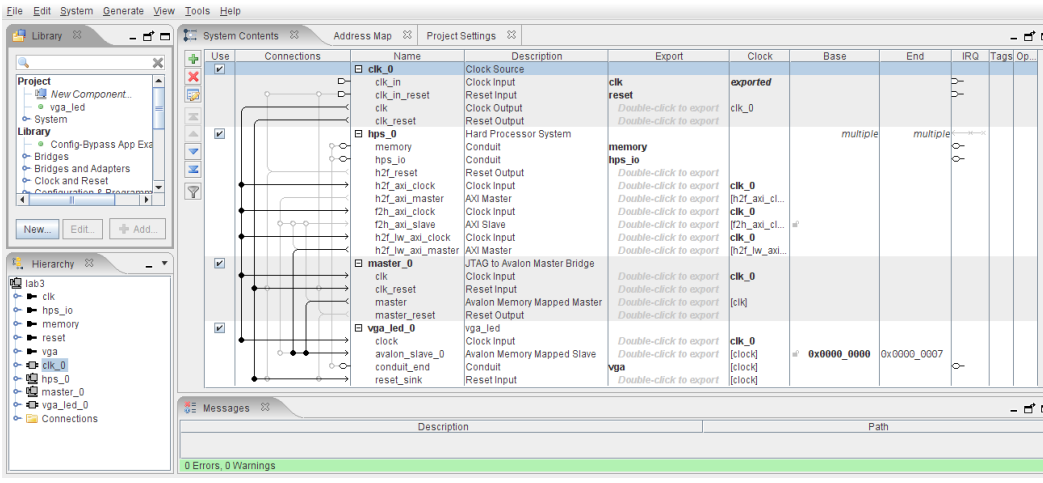
On the new *vga_led_0* component instance, connect the clock to *clk* from *clk_0* and connect the *reset_sink* to *clk_reset* from *clk_0*.

Connect the *avalon_slave_0* port on *vga_led_0* to both the *h2f_lw_axi_master* port on the *hps_0* component (this is the lightweight bus from the *arm* processors) and to the *master* port on *master_0* (this will allow you to write its registers from the System Console—see below).

Double-click to export *vga_led_0*'s *conduit_end* in the Export column. Set the name of the export to *vga*.

Run System→Assign Base Addresses to assign the base address for the *vga_led_0* peripheral.

The System Contents tab should now look as it does below.



Once you have resolved any errors or warnings, run Generate→Generate to have Qsys generate all the files Quartus needs to synthesize to make the design work.

2.3 *Connect the VGA Peripheral to its Pins*

The VGA peripheral you just created needs to communicate off-chip through pins. To do this, add the following connections within the instance of *lab3* near the end of the *SoCKit_top.v* file:

```
.vga_R (VGA_R),
.vga_G (VGA_G),
.vga_B (VGA_B),
.vga_CLK (VGA_CLK),
.vga_HS (VGA_HS),
.vga_VS (VGA_VS),
.vga_BLANK_n (VGA_BLANK_n),
.vga_SYNC_n (VGA_SYNC_n)
```

The lowercase signal names are part of the *conduit_end* you named "vga" when you connected the component to your design. They are being connected to named pins.

Compile your project in Quartus to produce the *output_files/SoCKit_Top.sof* file.

3 Use the System Console to Verify Your Peripheral

While we will eventually communicate with our peripheral through the Linux environment, it is often easier to check the hardware without software in the way.

Altera provides the System Console: an interactive Tcl environment that provides direct access to system busses. If necessary, start Quartus and compile the design you generated with Qsys in the previous section.

Download to the FPGA your newly created system with the VGA LED emulator peripheral: run Tools→Programmer from within Quartus and download the *output_files/SoCKit_Top.sof* file to the board as you did in lab 1.

If you connected the outputs on your new peripheral to the appropriate pins, the board should display the image on the right on the VGA monitor attached to the SoCKit board.

3.1 Running the System Console

Back in Quartus, run Tools→System Console→System Console. It should start up, report that it discovered some JTAG and USB connections, that it "auto-linked" to SOCKit_Top.sof, and note that a script (*system_console_rc.tcl*) does not exist, which is harmless.

In the Tcl Console sub-window, type

```
source syscon-test.tcl
```

This should load and run the *syscon-test.tcl* script that was provided for you in the *lab3-qsys.tar.gz* file. If all is well, it should report

```
Started system-console-test-script
Opened jtag_debug
Checking the JTAG chain loopback: 0x01 0x02 0x03 0x04 0x05 0x06
Sampling the clock: 100100101001
Checking reset state: 1
Closed jtag_debug
Opened master
Closed master
```

The script establishes contact with the JTAG debugging chain, establishes that the chain works by pumping a short sequence of numbers through it, verifies that the clock is toggling (your sequence may be different: all is well provided you see both 1's and 0's), resets the bus, then writes a test pattern to the registers that should change the display to what is shown on the right.

The System Console can be an invaluable debugging tool to verify the operation of the hardware without the interference of (potentially flawed) software. For your project, I suggest you write a similar script to exercise your hardware before you embark on software.

4 Communicate with Your Peripheral Through Software

Once you are satisfied your hardware peripherals work properly by testing them with the System Console, it is easier to configure the FPGA during the boot process rather than with the Quartus programmer.

Enter the *output_files* directory of your Quartus project and from the command-line, run

```
quartus_cpf -c SoCKit_Top.sof soc_system.rbf
```

to convert the .sof file generated by Quartus to an .rbf file that our boards download and program into the FPGA as part of the boot process (e.g., as done in lab 2).

Copy soc_system.rbf to the */sockit/lab3-XXX* directory on your workstation, where XXX is your or your partner's UNI (e.g., se2007).

Now, turn on the board, connect to its console as you did in lab 2 using

```
screen /dev/ttyUSB0 57600
```

select your lab3-XXX image to boot and make sure the FPGA is configured as part of the boot process, displays "4840LAb3" on the VGA display, and delivers you to a root prompt (e.g., `root@linaro-nano:~#`).

4.1 Compile and Run the Sample Program

Download *lab3-sw.tar.gz* from the class website and unpack it in your workstation's */sockit/lab3-XXX/root/root* directory.

Compile the device driver and user program, install the kernel module, and verify that it works. This should look like

```
root@linaro-nano:~/lab3-sw# make
make -C /usr/src/linux SUBDIRS=/root/lab3-sw modules
make[1]: Entering directory '/usr/src/linux'
  CC [M]  /root/lab3-sw/vga_led.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/lab3-sw/vga_led.mod.o
  LD [M]  /root/lab3-sw/vga_led.ko
make[1]: Leaving directory '/usr/src/linux'
cc     hello.c   -o hello
root@linaro-nano:~/lab3-sw# insmod vga_led.ko
root@linaro-nano:~/lab3-sw# ./hello
VGA LED Userspace program started
initial state: 3e 7d 77 08 38 79 5e 00
current state: 39 6d 79 79 66 7f 66 3f
VGA LED Userspace program terminating
```

```
root@linaro-nano:~/lab3-sw# rmmod vga_led
```

"make" compiles the kernel module (*vga_led.ko*) and the userspace program (*hello*).

"insmod"' loads the generated kernel module. In the supplied device driver, doing this should change the display.

The *hello* program is a userspace program that communicates with the *vga_led* device driver primarily through the *ioctl* system call. It opens the device, reads its state, writes its state, and animates the display for a little while.

"rmmod" removes the kernel module, which is necessary any time you modify and recompile the module.

5 What to Do

Modify the hardware and software in the skeleton you have been provided to display a bouncing ball. Change both the interface and contents of the hardware peripheral so that it displays a stationary ball at a software-controllable set of coordinates. Like the segments of the faux LED display, have the peripheral respond to writes to one or more addresses that control the location of the ball.

Adapt the provided device driver to communicate with your peripheral. E.g., create an *ioctl* that sets the coordinates of the ball.

You will need to modify the */sockit/lab3-XXX/socfpga.dtb* file to pass information about your new peripheral to the kernel. Modify the provided *socfpga.dts* file and replace the *vga_led* entry with yours. Compile it to a *.dtb* file, by running on the SoCKit board,

```
/usr/src/linux/scripts/dtc/dtc -O dtb -o socfpga.dtb socfpga.dts
```

Write a userspace program that bounces the ball by repeatedly communicating the new coordinates to your peripheral through your device driver.

6 What to turn in

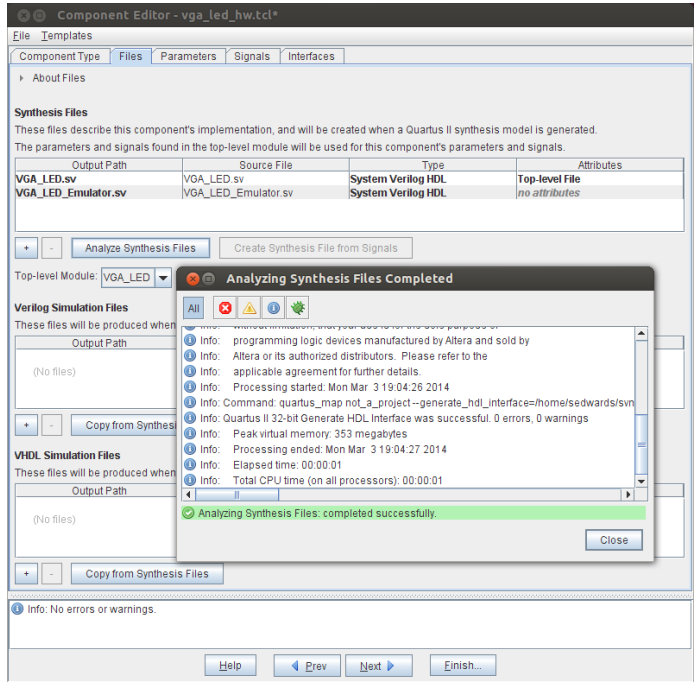
Find an overworked TA or instructor, and show him/er your bouncing ball. Once s/he is satisfied, collect just the files *you wrote or modified* for this lab in a directory called "lab3," make a tarball with *tar zcf lab3.tar.gz lab3*, and submit that via CourseWorks. This should include the SystemVerilog for your peripheral and source for your device driver and userspace program.

Do not submit everything in your lab3-qsys directory: it is too big.

7 Qsys Hints

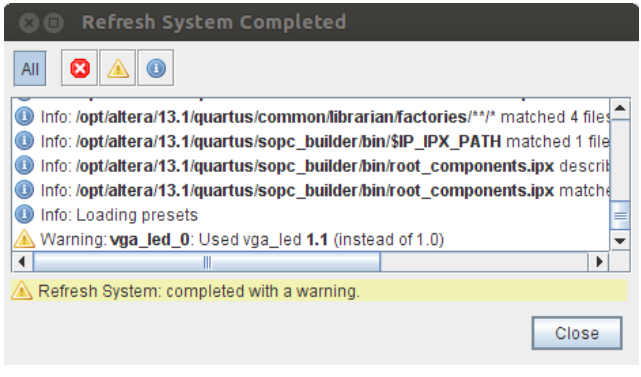
7.1 Editing the Source of Your Qsys Component

If you modify the SystemVerilog for your component (e.g., to fix a bug), you need to regenerate your system in Qsys before re-running Quartus. Open Qsys from Quartus (Tools→Qsys), open your *.qsys* file, select your component under "Project," and click "Edit." This should bring up the Component Editor window.



Click on the "Files" tab and then "Analyze Synthesis Files." Once your files compile successfully, click on the "Component Type" tab, increase the version number, click "Finish," and "Yes, Save" to save the change and return to the Qsys main window.

In Qsys, select File→Refresh System (or just press F5). It should complete with a reassuring warning indicating the version of your component has changed. Hovering over the instance of your component should also indicate its version has changed.



Now, select Generate→Generate… to instruct Qsys to regenerate your system so Quartus can recompile it.

7.2 Don't Edit Copies

Do not edit the files in the *synthesis* directory (e.g., in *lab3/synthesis/submodules*). These are copied or automatically generated by Qsys and will be overwritten the next time Qsys runs.

7.3 Verilog For a System Instance

Qsys can automatically generate a Verilog template for instantiating your system. Select Generate→HDL Example… then copy-and-paste the sample. You will need to edit the names of all the "`connected-to-`" signals to complete the connections. The instance of lab3 in the *SoCKit_top.v* file was generated in this way.

7.4 Viewing Components as Blocks

Select a component and then View→Block Symbol. This shows the interface to a component.