525.742

SOC FPGA Design Lab

Laboratory 6: Custom AXI / AXIS Peripheral for Processor Interfacing

**Goals**

1. To develop a peripheral which encapsulates your radio design, which uses standard interfaces for easy design re-use in any system
2. To use the new custom peripheral in a PS7-based system
3. To benchmark the performance of that peripheral and determine the bandwidth that we can expect when reading/writing an AXI-lite peripheral.
4. To gain familiarity with the Vivado flow for developing a custom peripheral
5. To use the PS/PL with the PS running Linux

**WARNING : this doesn’t look like a ton of work, but this will take some time to get it right. Do not put this one off and then get frustrated with the tools because you are rushing. It is very easy to get into trouble with the IP packaging flow because it requires that you understand exactly what files you need and how everything fits together.**

Note : The steps in this laboratory assume that you have the Linux boot image on an SD card, and are able to get a shell into the Zybo and connect to it via the network.

**Introduction**

At this point, you have constructed a large portion of the framework for a basic SDR, including selecting a small channel of interesting bandwidth and reducing the data rate to a manageable level. There still remain some pieces to build, but before we do that, we want to capture all of your hard work so that it can be used again, easily, by you or others. The radio you have spent all this time building, is likely not something that is easily moved to another system without effort. This effort would involve lots of configuring of GUI pieces, locating COE files and more. At the completion of this lab, that problem will be solved, all of the signal processing section of your direct-digital-downconverter will be encapsulated in one nice IP block that can be added to any system (and used!) in about 5 minutes.

**References**

In the lecture this week we will discuss packaging IP, both the concepts and the mechanics in Vivado. We will also as a class, construct a custom AXI peripheral. This peripheral (full\_radio) will be a good starting point for the creation of the IP requested in this lab.

**Custom Peripheral**

The majority of this laboratory revolves around the creation of a custom peripheral, which will be implemented in FPGA fabric and connected to the PS7 CPU via an AXI bus. This peripheral will be the DSP portion of our complete digital downconverter (i.e. not the i2c or DAC interface) and it will have the following interfaces:

* + - 1. AXI-Lite (Slave)
      2. AXI-Stream (Master)
      3. Clk (125MHz in this design), and is the clock for both AXI interfaces
      4. Resetn (active low reset)

Inside the peripheral will be those key pieces of lab 3,4,5 :

1. Lab 3 DDS (the fake-adc, real DDS)
2. Lab 5 Tuner DDS (creates a complex sinusoid for tuning the radio)
3. Lab 5 Mixer (multiplier)
4. Lab 4/5 Filter Chain (filters the tuned signal and reduces the data rate to 48ksps)

The AXI-Stream Master port of the peripheral will contain the output of the second filter as a compliant AXI stream interface (don’t forget the valid!!)

The AXI-Lite Slave port will allow the processor to access 4 registers:

1. Fake\_ADC\_PINC\_Register. This register is at the BASE Address of the peripheral (Peripheral address + 0x00). Writes to this register will set the phase increment of the fake-adc DDS. Reads from this register will return the current phase increment (i.e. what you wrote)
2. Tuner\_PINC\_Register. This register is at BASE+0x04 of the peripheral. Writes to this register will set the phase increment of the tuner DDS. Reads from this register will return the current phase increment (i.e. what you wrote)
3. Control Register. This register is at BASE+0x08 of the peripheral. The only bit implemented in the control register at this time will be reset to the DDS. When bit 0 of this register is ‘1’, the DDS will be in reset. Reads from this register return what was written
4. Timer Register. This register is at address BASE+0x0C of the peripheral. Writing this location will have no effect. Reading this location will return the value of a 32-bit free running counter which is counting clocks.

**Building and Demonstrating the Custom Peripheral**

For this lab, we are going to do something different, and that is to all start with the same base project. As we have briefly discussed in class, though the Vivado GUI is often the best tool for iterative development and experimentation, it is often not the way that large collaborative projects are run. Transferring full projects around is cumbersome, understanding what someone changed from one version to the next is very hard, and intelligent version control is nearly impossible. For that reason, projects are often at least stored and version controlled as scripts (think your VHDL source, and a set of instructions to make the project like “put down this IP, set these parameters, connect this to that…etc.” For this lab, we are going to all gain a little bit of a taste of that by starting with a project which uses that configuration. Now that you understand what is going on behind the scenes (because you’ve done it yourself many times) the automated version may seem even more helpful.

The project repository is located for your retrieval at <https://github.com/dougwen/radio_periph_lab.git> . If you have a git client on your machine, clone the repository, or you can navigate to that location and grab the archive itself. Notice first that the entire project is very small, and is entirely text files that are ideal for version control. *IMPORTANT : if you download this repository as a zip file, it will include a “-“ character since github names the archives as repositoryname-branch. You will definitely want to change the name so you don’t have a “-“ in your project path.*

In the repository, a few scripts are visible. In windows you can just double-click on the “make\_project.bat” file, in Linux, execute “make\_project.sh” from the command line. This script will:

* + Construct the vivado project
  + Add sources to the project
  + Draw the block diagram and set all the IP settings
  + Synthesize, Place + Route the design
  + Generate a bitfile
  + Note that we can also script the generation of a vitis project and BOOT.bin, but we won’t be using that flow for this lab. If you are looking at make\_project.bat or make\_project.sh, you can see the steps necessary to do all of this.

Note that the newly generated vivado project is in the “vivado” directory. This project is in the structure you are familiar with, and can be edited and used in the GUI. Also, the entire vivado project can easily be deleted and then regenerated if you like. This project is configured to use an ip repository named “ip\_repo” which is at the base level of the directory you cloned. That ip\_repository has two pieces of IP in it: lowlevel-dac-interface, and a new peripheral (the one we started in class) named full\_radio.

Lets explore around a little bit before we embark on any changes. Open up the newly created project in vivado, and check out the block diagram. Note that the low-level-dac interface and the full\_radio peripherals are instantiated in your block diagram as IP. The “full\_radio” is not complete of course, but it does have some working pieces, namely our “Fake ADC” DDS, who’s phase increment is controlled by a register at the base address of the peripheral, another register which always reads back 0xDEADBEEF(this is at address BASE+0x04), and 2 more registers locations that can be written and read back, so before we change it, lets make sure that it works.

1. First, if you haven’t already, configure the codec by loading the instructor bitfile “configure\_codec.bit.bin” into the PL. In case you’ve forgotten, this is done by typing fpgautil -b path\_to\_bitfile. At this point a tone should be coming out the headphones. We are done with that bitfile now, it only had one job ☺
2. Using scp or MobaXterm transfer the newly created bitfile to the zybo (design\_1\_wrapper.bit.bin is in the impl\_1 directory of your vivado project, just like a normal bitfile would be)
3. program the bitfile into the PL : type “fpgautil -b design\_1\_wrapper.bit.bin”
4. now, changing the DDS phase increment is as simple as a single memory write. We can do this from the command-line using devmem a command line tool for accessing physical memory. Lets write the value 1000 to the phase increment : **devmem 0x43c00000 w 1000. You should hear the tone at ~931Hz**
5. Lets also read the value of the next register, where we expect to see 0xDEADBEEF **devmem 0x43c00004 w**

So, it looks like our full\_radio peripheral works as designed, now it is time to actually turn it into the full radio!

Once you verified you have a known good starting point, you will want to edit the full\_radio peripheral to actually include the rest of our radio. **Edit in IP Packager** the full\_radio peripheral, to add all of the pieces that you have now mastered from lab 3,4,5. Once you have finished, be sure to **repackage** the IP.

HINT: don’t actually click Package until you have green checkmarks for all of the steps. Address any warnings which come up during the packaging step – they likely can not be ignored safely.

HINT: when you package the IP, let it create an archive of the IP for you. In that zip file will be only the files that are needed for the IP, and nothing else. Useful to have if you accidentally clutter up the ip directory with a bunch of other files some way or another.

Once you have repackaged the IP, you should be prompted to update the IP in your base design; however if not, be sure to run “Report IP Status” to start the update process. Since this is the only part of the design that needs to be changed, you can now rebuild the entire design to the bitfile stage. With this bitfile programmed into the PL, the PS will be able to control tuning and frequency by register writes. Note that the project has a system ILA in it which you can use to observe both the AXI transactions to the custom peripheral, and the AXI-S data flow from the custom peripheral. Once your bitfile has been created, lets test it out, first interactively:

1. Make the .bit.bin file, transfer it to the zybo, and load it with fpgautil.
2. Is the counter present ? Run this line a couple times and see if the value is incrementing : **devmem 0x43c0000c w**
3. Set the tuner DDS phase increment to 30000000 : **devmem 0x43c00004 w 30000000**
4. Set the fake ADC to phase increment 30001000 : **devmem 0x43c00000 w 30001000**

If the results are expected, it is time to run the full test suite. Part of the repository (under src/linux\_software) is a C program named test\_radio.c can be used to test the radio.

1. Edit the test\_radio.c file to print your name at the beginning of main()
2. Edit the test\_radio.c file to change the print\_benchmark function to actually compute the actual read-speed in **Megabytes/second**
3. Transfer the test\_radio.c file to the Zybo if you haven’t already
4. Compile the code : **gcc test\_radio.c -o test\_radio**
5. Execute the code : **./test\_radio**

The test radio code tunes the radio to 30MHz, and then plays a little test song by injecting tones near 30MHz into the radio from the fake adc. Finally. the program will then do a benchmarking loop, whereby it uses the timer register to measure how many clocks it takes to transfer a fixed amount of data through the peripheral. It reads the timer 2048 times. The number of clocks that has elapsed between the first and last reads can tell you exactly how much data the processor can read from a peripheral using this method. This is an important number – because it determines the types of things that we can do with this type of peripheral.

**What to Turn In**

1. Your entire project packaged as a script, just like you received it. Attach a zip file to the submission which should not include the generated vitis or vivado directories. I should be able to take your directory and run it all the way to the point of generating a .bit.bin that I can use for testing. *Though it isn’t required,* 10 bonus points are given for skipping the zip file and locating it in a git-repo where I can clone it. The way I imagine this, the only things that change from the project you grabbed are:
   1. Some changes and new files in the ip\_repo/full\_radio folder. Note, if this is MegaBytes in size, you’ve likely done something wrong.
   2. Test\_radio.c now has your name in it, and has your code for calculating throughput in MegaBytes/second
2. A short video which shows the program executing on your Zybo. Make sure that the result you measured for the data rate from the peripheral (in Mbytes/sec) is visible on the video, and that I can hear that beautiful song playing in the background… does it sound familiar to any of you?
3. Clone the repository
4. Execute “construct\_project.bat” or “construct\_project.sh”, this will make a new directory (“vivado”) with a complete vivado project in it. This project will contain the PS, an ILA, the lowlevel\_dac\_interface, and a skeleton radio\_peripheral, the one we worked on in class.
5. Open the project in vivado to see the design. As mentioned, this design already contains a custom peripheral which is the starting point for your complete radio peripheral. Right click on it, and “Edit in IP Packager” to edit your IP and make it how you want.
6. After transferring all of your DDC from lab5 into the peripheral, repackage it
7. Update the bd (report ip status)
8. Make a bitfile (either manually in the gui ,or by running impl\_project,bat (or impl\_oroject.sh for linux users)

At this point, your hardware is complete, and it is time to test the peripheral with some software. This time, instead of exporting to Vitis and writing software that way, we will be running some software