525.742

SOC FPGA Design Lab

Assorted Topics for Lab 9, linux SDR

**Flashing an SD Card with the Provided Linux Image**

Preparing an SD card to boot linux is very much like preparing your SD card for booting your previous PS7-based projects. This time however, you don’t need to create any of the files (i.e. BOOT.bin…etc.) since they will be provided for you. At the link provided in the lab you can download 4 files which can be placed at the root level of your standard FAT32 formatted SD-card. These files are the familiar BOOT.bin, and then 3 other files which contain U-boot instructions, the linux kernel image, and a zipped copy of a root filesystem (all the files that will be there in your linux build when you login). The filesystem will be unzipped, and placed in a RAMdisk, so you will in general be running out of DDR (the SD card won’t generally be used)

Place all the files on your SD card and you should be ready to go.

(Note – the provided BOOT.bin does NOT have a “bit” file for the PL, the PL will remain unconfigured until you program it later) This approach doesn’t work well of course if there are devices that are essential to boot that are in the PL, but in our case, the entire linux system runs on the PS7 itself, and our PL will just be extra peripherals (like our radio) that we access from user-space.

**Booting from the SD card.**

Place the SD card in the slot, and make sure to have your jumpers set to boot from SD. Power on the board (note, you can reboot at any time by hitting the PS-SRST button if you don’t want to power-cycle). The boot process will be printing out progress to the USB serial port (not the PMOD, but the one on J12…. Remember, we don’t want any dependencies on PL for this part). You should see 10-20 seconds worth of booting on the serial port, and then eventually you will get a prompt:

root@doug\_linux:~#

Here you can type standard Linux commands like “echo Hello”, “pwd”…etc.

Poke around for a little bit, and see what is there.

**Network Connectivity**

This build of linux (like most) has a full featured network interface that won’t require any of the fancy steps we talked about in class. The board is setup by default to auto-negotiate speed with whatever it is connected to, and to get an IP address by requesting one via DHCP. (This is likely how your computer is setup as well). The most ideal setting to use here is to simply plug the Zybo into a spare port on your home router, where it will get an IP address and be accessible to you on your home network (and it will also have access to the internet). This is by far the best choice. If you don’t have a home router to plug into, then you can direct-connect the Zybo to your computer with just a little bit of extra effort. This isn’t as handy as freely roaming your house and talking to the Zybo while your board is sitting somewhere else, but it will work! Both of these approaches will use the USB network adapter provided in the kit (or a network adapter on your PC)

*Option 1: Both computer and Zybo have static IP*

With this approach, your computer will have a static IP address, set to whatever you like (ex. 192.168.30.1, 192.168.30.2). Set the IP address on your computer to be static using the windows control panel. To change the Zybo IP, wait until it boots, and then type the following at the terminal prompt (using the serial port of course)

ip addr add 192.168.1.2/24 dev eth0;

*Option 2 : Zybo remains unchanged, PC has static address and runs a DHCP server*

This may be more trouble than it is worth, but you can run a DHCP server on your windows machine using a tool like this : . This will allow your Zybo to get an IP address directly from your computer when it powers up.

If you can’t ping or access your Zybo and you think you should be able to, you can always check the state of things from the serial port, by typing the command “ifconfig”, which will tell you if it has an IP address, and what it is.

eth0 Link encap:Ethernet HWaddr 00:0A:35:00:1E:53

inet addr:192.168.11.218 Bcast:192.168.15.255 Mask:255.255.240.0

inet6 addr: fe80::20a:35ff:fe00:1e53/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:3368 errors:0 dropped:0 overruns:0 frame:0

TX packets:767 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:421639 (411.7 KiB) TX bytes:105646 (103.1 KiB)

Interrupt:34 Base address:0xb000

**Getting Shell Access to the Zybo via Ethernet**

Using TeraTerm, MobaXterm or putty, you can get a terminal connection to the Zybo over the network by ssh’ing to it with a username of “root”. The password for login will be “root”. Then you should get a login prompt just like what you got on the serial port. To get the IP address of the board, you can type “ipconfig” in the serial terminal. (Most routers will give your board the same address each time, so you won’t always have to remember it)

eth0 Link encap:Ethernet HWaddr 00:0A:35:00:1E:53

inet addr:192.168.11.218 Bcast:192.168.15.255 Mask:255.255.240.0

inet6 addr: fe80::20a:35ff:fe00:1e53/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:3368 errors:0 dropped:0 overruns:0 frame:0

TX packets:767 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:421639 (411.7 KiB) TX bytes:105646 (103.1 KiB)

Interrupt:34 Base address:0xb000

Note that you can do this right from the Command Window in Windows if you like, as ssh and scp are built in there as well:

ssh root@ip\_address\_of\_your\_zybo

**Transferring Files**

If you are using Mobaxterm, you can drag and drop files back and forth from your computer to the Zybo board. In the background, Moba will be using SCP to transfer the files. This can be very useful, because **most directories on the Zybo are not persistent across boot, so you don’t want to generally be using the Zybo as the place to store your work during development**. An exception is /media/sd-mmc..., which is the SD card itself. Practice moving a file back and forth to the Zybo.

From command-line on windows or linux : scp file\_you\_want\_to\_transfer root@ip\_address\_of\_zybo:filename\_you\_want\_it\_to\_be\_on\_zybo

**Writing a simple C Program**

To develop code for the PS, there are many choices, but the easiest way is to compile code directly on the PS itself (gcc is installed). To get started, use “vi main.c” on the platform to edit a new program, or edit this on your computer and copy it via scp/Mobaxterm.

#include <stdio.h>

void main()

{

printf("hello linux world\n");

}

Then compile the program from the command line : “gcc main.c -o hello”

This will make an executable program named “hello”, which you can execute by typing : “./hello”

**Simple Multithreading**

Now that we have an OS, it may be convenient to make use of some of the multi-tasking features to make our software clearer and/or easier to develop. As an example, streaming data via ethernet is a rather separate thing than our user i/o which will be collecting commands for tuning...etc. Since sometimes collecting user data from the terminal is easier if we allow it to be blocking, it could be very valuable to have the ethernet streaming happening in its own thread, operating independently. The OS can manage the task switching between the two, and to us it can look like they are operating in parallel. Here is an example program that you can try which shows how easy it is to spin up multiple threads doing different things inside a program. Compile the below code with gcc main.c -o threadtest -lpthread

(the-lpthread tells the compiler to link the the threading library)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

void \*myThreadFun(void \*vargp)

{

while(1) {

sleep(1);

printf("Printing from Thread \n");

}

return NULL;

}

int main()

{

pthread\_t thread\_id;

pthread\_create(&thread\_id, NULL, myThreadFun, NULL);

while(1) {

printf("printing from main\n");

sleep(2);

}

}

**Loading a PL design into the PL.**

You can continue to use your cable and the hardware manager to program bitfiles into the FPGA if you like. Eventually however, you will want to (it is required for this lab) to program the bitfile directly from the PS instead of having to have Xilinx software connected. This can be done by running a script on the board called “fpgautil”. Unfortunately, this utility doesn’t actually use the exact same format of bitfile, it wants a modified version. To make that modified version, a “bin” file, you need to run a Xilinx tool called “bootgen”. The procedures for making your “bin” file are shown below. **The radio\_periph\_lab make\_project.bat script does this automatically as the last step, so the below is for information**

First make a .bif file with name of your choosing, I will choose doug.bif, it should contain only the below text, where the file listed is the name of your bitfile for the design, whatever that is. In this example, the name of the top level bit is design\_1\_wrapper.bit

all:

{

design\_1\_wrapper.bit

}

then execute this command :

bootgen –image doug.bif –arch zynq –process\_bitstream bin

There is probably also a GUI way of doing this. The result is a file with the .bit.bin extension, which you can transfer to your zybo board and use to program the FPGA PL. There is an example .bit.bin file which you can load to test this functionality. This bit.bin file won’t do much, but it will put an AXI GPIO hooked to our LEDs at location 0x43c00000 and more importantly, it will program the codec for you. You won’t need to worry about doing that part for this lab (we’ve done that enough)

fpgautil –b thebitbinfilename

Upon doing this, your done light should go on and the FPGA PL is now programmed

**Sanity Checking the PL design using the command line**

To see that the expected design is present in the PL, you could write a bunch of software to read from it, but we will get to that later. The easiest thing to do first is to just read/write to the FPGA registers via a tool called “devmem”. For instance, if you have loaded the instructor bit.bin file which is on the SD Card : configure\_codec.bit.bin, there is a GPIO hooked to LEDs at address 0x43c00000, so we can access registers in it and control the lights by just typing:

devmem 0x43c00000 w 3

To write 3 to the LEDs

If my radio peripheral were in the memory map at 0x43c00000, I could do this to read the timer:

devmem 0x43c0000c w

I could do it again and watch the timer value change. Or, I could ask linux to read that register over and over again watch the whole thing continually loop:

watch –n 0.1 devmem 0x43c0000c w

Stuff like this is always a really good thing to do in order to make sure you understand what peripherals are in your PL and see them working. Please note though, this is a command-line interactive checkout, it is not intended to be a production solution to reading data from and controlling your peripherals – it is really slow!

**Accessing the PL peripherals in C using /dev/mem**

In an ideal world, one would have drivers in Linux for the peripherals that you have hooked to the PS. There are lots of ways to approach this, and all are beyond the scope of this class. In our case, we would just like to get memory access to our peripherals in as simple of a way as possible. Fortunately, a device in linux exists for this : /dev/mem. Opening this file will allow us to get a pointer to our peripheral, where we can read and write registers on the peripheral.

Here is an example:

#include <stdio.h>

#include <sys/mman.h>

#include <fcntl.h>

#define RADIO\_PERIPH\_ADDRESS 0x43c00000

void main()

{

int mem\_fd = open("/dev/mem", O\_RDWR | O\_SYNC);

void \*map\_base = mmap(0, 4096, PROT\_READ | PROT\_WRITE, MAP\_SHARED, mem\_fd, RADIO\_PERIPH\_ADDRESS);

volatile int \*radio\_base = (volatile int \*)map\_base;

printf("Current Timer is: %d \n", \*(radio\_base+3));

printf("Current Timer is: %d \n", \*(radio\_base+3));

}