University of Pennsylvania
Department of Electrical and System Engineering
System-on-a-Chip Architecture

ESE532, Fall 2019 — I/O and Energy Milestone — Wednesday, November 6

Due: Friday, Nov. 15, 5:00PM

Group: Integrate I/O. Measure energy. Writeup design progress (Items 1–4).

Individual: Energy estimate. Writeup on energy and measurement and estimation. (Items 5, 6).

1. Integrate I/O and validate operation with the I/O in place. Report the maximum real-time throughput the current design can sustain.
2. Move some part of your design onto the FPGA for acceleration.
   Writeup should identify what you moved onto the FPGA, how you validated it, and how you tuned it. Identify the current throughput achieved and the current bottleneck(s).
3. Turn in a tar file with your I/O integrated and FPGA accelerated code to the designated assignment component in canvas (one per group).
4. Turn in a tar or zip file with binaries to support execution of your code to the designated assignment component in canvas (one per group).
   (a) The tar (or zip) files should package up the `projectdir/debug/sd_card` directory for your encoder implementation.
   (b) Your encoder should take inputs from the ethernet link.
   (c) Your encoder should store the encoded result in `/compress.dat` on the SD Card.
5. Measure and report the energy to encode the Linux source code on the two implementations corresponding to Problems 1 and 2 (if you have had a chance to explore multiple design options using the FPGA for acceleration, measure the highest performing design that you have currently found.)

   Note: the request here is for energy consumption (Joules) not power consumption (Watts). If two designs run in different times, power alone does not tell you which is most energy efficient.
6. Estimate the energy to encode the Linux source code for the same two designs based on metrics provided by the Xilinx tools.

We don't expect significant FPGA acceleration on this milestone, but we do want you to become familiar with how to measure and estimate the energy including that of the FPGA mapping so you will be prepared to characterize your more mature designs.

Measure and Compute Energy

To measure energy, you will need to measure both current into the Ultra96 board and the end-to-end runtime. From those two measurement and knowing voltage at which the board is supplied, you should be able to calculate energy. To measure current, you will use the lab power supplies in Detkin along with a suitable power connector rather than your normal transformer. *We are swapping out one board per team to provide you with a power supply with connections you can use with the Detkin power supplies. Please exchange your board in Office Hours.*

Estimate Energy

You can obtain the power by opening the Vivado project file generated by SDSoC in Vivado (like we did before to show the block diagram) and looking in the *Project Summary*. The power consumption is given as *Total On-Chip Power* in the *Power* section at the bottom right. The same section also has a tab called *On-Chip* that shows the dynamic and static power. Dynamic power is further divided into clocks, signals, logic, BRAM, and PS blocks.

Ethernet Input Integration

Next we will describe how data will be sent through ethernet using the client and server, the packet layout, Petalinux, and other relevant information for getting started to receiving real-time data. We provide the files `server.cpp`, `server.h` and `client.c`. The contents and usage of these files are described below.

Model

Our basic model will be communication between two Ultra96 systems over ethernet. One system will send packets at a fixed rate. A second will receive the data and compress it. The Figure 1 and Figure 2 below show the setup and cabling. Since the first system is sending data at a fixed rate, it is necessary for the receiver to compress the data at that rate or data will be lost. We provide the code for the sender (`client.c`). Your project is connected to the receiver (`server.cpp`).

What is Petalinux?

Instead of running stand-alone configurations as we have to date for this project, we will run on top of a Linux operating system for the project.

Petalinux is Xilinx's build tool for putting a Linux operating system on an Ultra96. It will be leveraged to gain access to stock I/O drivers to send and receive ethernet packets. PetaLinux is widely used in the industry for Xilinx's FPGA solutions.

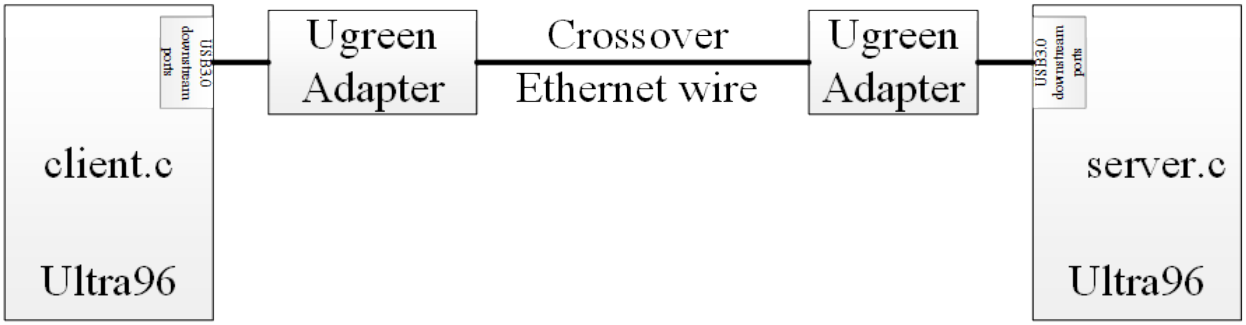

Figure 1: Ethernet Diagram

Figure 2: Physical Interconnection

Since you are running a full Linux operating system, you will also get a Linux command shell (which you can access through the SDx console) and can execute Linux commands interactively from the terminal. In some of the instructions that follow, you will interact with the Linux command shell.

MultiCore

We will be running the Linux operating system. Linux runs a symmetric multiprocessing (SMP) scheduler on the APUs (the A53 cores). This means that processes will go on separate cores based on the scheduler when they are ready to run. You will recall that in Homework 3 you utilized the 4 cores, by compiling 4 separate elf files and programming each core. With Linux this is not necessary. You can simply use the `fork` system call inside your program. This system call creates a new process that will now run. For example, in your C file you could have something as follows.

```
void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)
          LZW();
     else
          SHA256();
}
```

If you would like to bind certain processes to certain cores then you may look into setting set the CPU affinity of a task to bind it to a specific core. Otherwise It will not impact your design. For more information on CPU affinity you can read here: https://www.linuxjournal.com/article/6799

If you are running multiple processses and are curious about how tasks will be divyed up you may run the Linux command `top` from the command shell followed by "1" this will give you a breakdown of your CPU utilization. See https://unix.stackexchange.com/questions/146085/top-command-on-multi-core-processor

Packet Layout

We will be sending data via UDP data grams. Linux supports the UDP protocol and receiving packets from the client can be done easily using Linux IP. The code provided will direct you on how to setup your compression pipeline to listen as well as handle incoming packets.

The Maximum size of a packet will be 16K Bytes. The header of the packet will be 2 bytes consisting of a done bit denoting that all of the data has been transmitted as well as the length of the data contained inside the packet.

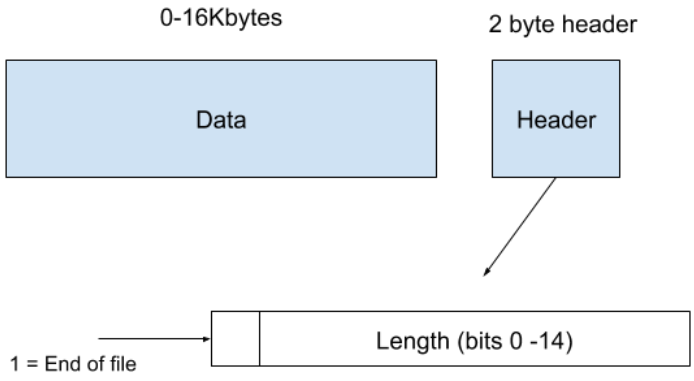

Figure 3: Packet Layout

Code Repository

To view examples of how to use the server, look at the example in this repository: https://github.com/micallef25/SDSoc-Examples-Fall2019

To view examples of how to use the client, look at the example in this repository: https://github.com/micallef25/ESE532CLient

For the server, you will be interested in the following two branches **emulated stream** and **fullstream**.

The branch **fullstream** represents a working model of receiving ethernet packets over the wire and sending it to the PL. You will be specifically interested in the files `server.cpp` and `server.h`.

These are the files provided and can be directly copy and pasted into your project to set up your server pipeline to receive data. The rest of the repository can serve as an example of one could implement the design.

At a high level you need to call the function `setup_server()` once.

Following this you can makes calls to the function `getpacket()` to receive your next data gram. Please note that your buffer passed into this function needs to be able to handle the maximum payload size plus the two byte header. Please also note that the `recvfrom()` Linux call is blocking. This means that your process will be blocked until a datagram has arrived. Depending on your design this may be ok. If you do not want to block you may look into the `select()` system call as an alternative. This will allow you to check if data has arrived in your socket and take actions accordingly.

You are of course free to write your own application.

For more information on how to receive the data you can refer to the man page.

https://linux.die.net/man/2/recvfrom

The branch **emulated stream** represents a working model of how you can emulate receiving packets from a client. In the past, students encountered many errors they needed to debug when sending data in a streaming manner. This is an example of how you could test streaming to help facilitate debugging.

Configuring Sender

You will likely need to slow down the client's data transfer to begin with. If your system cannot keep up with the pace of the sender packets will be dropped.

To configure the sender, when you start the client from the Linux shell, it takes arguments as shown:

```
./client.elf -s 5 -f file -i ip_address
```

Useage example is:

```
./client.elf -s 5 -f prince.txt -i 192.168.0.100
```

`-s` option specifies the sleep time or delay between packets (in microseconds)
`-i` option specifies ip address to send to
`-f` option specifies what file to send

For Problem 1 (and for later times where you characterize your throughput), you should adjust the `-s` argument until your design fails. Report your maximum throughput as the throughput associated with the smallest value of `-s` on which your design successfully receives and correctly compresses the input. Measure the actual throughput by measuring the time it takes for the client to send the file. You can use `/bin/time` to measure the time.

We currently believe the upper bound on the throughput achieved by a placeholder receiver is around 800 Mb/s limited by the Ugreen ethernet-to-USB interfaces. This will be your actual target instead of the 1 Gb/s stated in the project handout. That is, we're not asking you to exceed the speed supported by the ethernet-transceiver, but we are asking you to match it.

Ways to Test

In an effort to help facilitate debugging there are a slew of way's to build up to a real-time ethernet stream that can be configured.

If you find it easier to debug using your bare-metal application or want to collect a trace of your application you may emulate streaming data into your system. This can be show in the branch **emulated stream**.

Setup and Run

The button that you have been using to turn on your device should also be used to turn off the device. Please connect to the Linux shell console to watch it start and shutdown properly. Please do not abruptly unplug the device as you could corrupt the sd card image. *You will need to exchange you original SD card with the SD card (Linux image inside) from the TA during the office hour.*

Create a SDx project by using the Ultra96 Linux platform from https://github.com/vagrantxiao/PetaLinux_ESE532

To test your compiled application all you need to do is to flip the boot mode from JTAG to SD card. Following this copy the contents in the SD card folder onto your SD card. These contents are located here: `projectdir/debug/sd_card`. The files you should see are: `BOOT.bin, README, server.elf, server.elf.bin` for server project and `BOOT.bin, README, client.elf, client.elf.bin` for client project.

If this is your first time running anything from the Linux OS on the Ultra96 board, run the following commands in the Linux shell on the Ultra96:

```
mount /dev/mmcblk0p1 /mnt
mkdir /usb1
cd /mnt
cp ./libsds_lib.so /usr/lib
./server.elf
```

If it is not your first time you can run these commands:

```
mount /dev/mmcblk0p1 /mnt
cd /mnt
./server.elf
```

In the Linux shell, you can assign an IP address to the Ugreen ethernet2USB adapter by executing commands.

```
ifconfig eth0 192.168.0.100
```

You will need to run `ifconfig` on both the client and server. So, maybe you run the above on the client and on the server run:

```
ifconfig eth0 192.168.0.200
```

Testing with Linux Host

For testing, if you have a Linux machine you can also run the client on your laptop to send data to your Ultra96 server. You will need to adjust the IP addresses you use accordingly.

Roundup of New Hardware

1. Ultra96 with modified power connector (one per team; pickup in Office Hours)
2. SD Card with Linux (at least 2 per team, maybe all 3; pickup in Office Hours)
3. Ethernet crossover cable (1 per team; distributed in lecture 11/6/19)
4. Ugreen Ethernet to USB interface (2 per team; distributed in lecture 11/6/19)
5. USB 3.0 cable (1 per team (initially; distribute in lecture 11/6/19) – may not need this, but at some points looked like it might be necessary.

Questions

If anything is unclear please post on piazza or come to office hours, and we will be glad to assist.