Pre-Alpha Build Report - Lycan

Video Link

https://youtu.be/jokJ8e4beo8

Effort

The pre-alpha build milestone caps off a long period of research and planning for our team. We spent the first few weeks of the project defining parameters and creating a plan, which is outlined in our Design Plan Draft. After that milestone, we began the early stages of development on our host computer software and learning System Verilog. We chose to utilize System Verilog because of it being an industry standard, and will be a useful skill for our team members to have in their future careers. We spent at least a dozen hours communicating over text, Zoom, and in-person to determine the appropriate circuit blocks needed for the device, the interfaces of those modules, and how they would signal each other. This work is outlined in documentation/planning.md in our Github repository. Overall, each team member has worked about 6-8 hours on writing RTL code and testbenches for Lycan, representing the starting implementations of our design.

For this milestone, our goal was to have a proof of concept for communicating between the Lycan FPGA and the host computer. This required implementing the controller for our FT601 USB to FIFO interface IC, the bus arbiter that controls when peripherals are able to send data back to the host computer, and a loopback peripheral that will receive any data transmitted by the host computer and immediately send it back to the host computer. These three components will allow us to prove that our board can receive and transmit data over USB before we implement more complex functionality for future milestones. More details about our current implementation can be found below.

Architectural Elements

External Interface

The FTDI UFT601 USB to FIFO development board is connected to a computer using a Micro USB 3.0 cable. The dev board was procured in the two weeks leading up to this milestone, and the cable was received this week. A Python script (see our Github repo in the software folder) was created around the ftd3xx Python package, which communicates using the d3xx driver to the FT601 chip on the dev board. The FT601 is our primary interface between Lycan and the host computer. The script has methods for reading the FIFO, writing a packet (32 bits) to the FIFO, and writing multiple packets to the FIFO (one at a time). The script also identifies the connected device, and prints the information, such as serial number, for future use.

Persistent State

At the moment, Lycan does not store any persistent state on-device. The main state of the device is the current peripheral instantiated in each reconfigurable block and the configuration registers for each peripheral. Since the main goal of this milestone was to have a proof of concept for communicating between Lycan and the host computer, we have so far only designed a loopback peripheral which requires no state. Additionally, we are waiting to implement partial reconfiguration until we have the basic architecture finished and multiple peripherals have been designed. Thus, we have no persistent state at the moment since our device does not yet require it.

Internal Systems

For this milestone, we designed three major circuit blocks. The first block is a loopback peripheral. Right now, we have it set up so that 8 peripherals can be enabled on the board. Each instance of a peripheral in our system will have designated local FIFOs to hold their data while it is waiting to be sent out of Lycan to the host computer processing the data. These FIFOs are 256 bytes each and are designed to send out signals for when they are empty, 75% full, and completely full. Additionally, they have control signals to say when they are allowed to send data out.

To allow multiple peripherals to share the USB bus to send their data back to the host computer, we have designed a bus arbiter circuit. This arbiter uses a barrel shifter round robin approach to ensure none of the peripherals starve and they all get even access to the bus under normal conditions. It does this by using the inverse of each peripheral's rx_fifo_empty signal as a "bus request" signal. These request signals are used as the inputs to a barrel shifter which determines the priorities of each peripheral's request signal. From there, a priority encoder determines which peripheral's RX FIFO is currently not empty (requesting access to the bus) that has had the longest time since last access. This is sent as a 3 bit grant signal that informs the peripheral with that address that they can put data onto the bus. Additionally, to handle the case where some FIFOs might have a lot more data that needs to be transmitted out, there is a signal for rx_fifo_almost_full which should any of the peripheral FIFOs report that they are more than 75% full, these request signals will take priority and round robin will proceed only for FIFOs that are almost full.

Finally, we have designed a controller for the FT601 USB to FIFO IC. The controller at this stage manages whether the host computer can transmit data to the peripherals or if it will instead allow peripherals to send data to the host computer. To manage this, we use signals that provide the current state of the USB FIFO including usb_tx_full and usb_rx_empty. These are inherent to the design and are described in the spec sheet for the FT601.

Information Handling

Communication

A communication protocol between the host computer and the FIFO has been finalized, and is outlined on our Design Plan Draft and documentation/planning.md in our Github repository. We plan to send 32 bit packets to and from Lycan using the FT601 interface IC. Every packet includes the address of the peripheral it is addressed to/from, whether the packet contains configuration data or data samples, and how much valid data is contained in the packet. These packets can then be directly transmitted to the peripherals that they are addressed to using a demultiplexer. The peripherals are then responsible for parsing packets from the host. When a peripheral has data that it wants to send to the host, it creates its own 32-bit packet, which is then queued in a local FIFO to be sent through the bus arbitration system (see the Internal Systems section).

Integrity & Resilience

We utilize try-except structures within the Python code to make sure that we catch possible errors within the d3xx driver module, or errors relating to “None” object types. We added custom exception descriptions to better explain the issues. Further, we raise exceptions when the results are unexpected. For example, if during a write command no data was actually written to the FIFO, an exception is raised and the program is halted. In the future, we will use further error checking and data sanitization to ensure that packets being sent to the FIFO are within spec and error-free.