Network-Attached Laser Projector Final Report

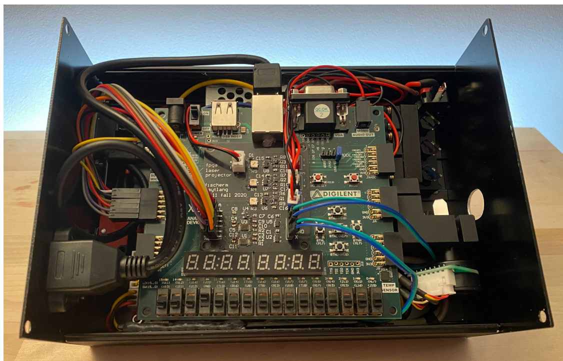
1st Fischer Moseley
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA, USA
fischerm@mit.edu

2nd Jay Lang
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
jaytlang@mit.edu

***Abstract*—We present a design for a Network-Attached Laser Projector implemented entirely in hardware on an FPGA fabric, which utilizes a novel parallel-stack UDP offload engine to connect to a local area network (LAN), and stream full-color vectorized images to an RGB laser projector. This hardware networking stack interfaces with the laser control module directly, and user packets are processed in real time, allowing for total system throughput exceeding 100 megabits per second. We implement this design using a custom laser module and the Nexys 4 DDR FPGA, evaluate its performance and quality using custom trajectory generation software to stream images over a custom application layer network protocol, and discuss potential areas for future expansion and improvement.**

***Index Terms*—Digital systems, Field programmable gate arrays, Computer networks, Diode lasers, Optical projectors**

I. Physical Construction (Fischer)



Apart from the interlocks, the enclosure also contains the following:

- An IEC203 connector for 120V power, mounted on the rear panel.
- An Ethernet connector, mounted on the rear panel.
- A micro-USB connector, mounted on the rear panel.
- The Nexys 4 DDR board, with the custom laser/galvo board connected to a PMOD connector.
- A bipolar, $\pm$15V power supply for driving the laser galvos and laser diodes.
- A pair of galvonometer drivers. The galvonomoters have closed-loop control, so these boards implement some kind of control loop in analog electronics.
- A galvonometer module, which includes a set of mirrors mounted orthogonally such that the laser position can be varied in the x and y directions.
- A RGB laser module. The multicolored output is formed by combining 660nm, 520nm, and 450nm lasers, which does not create a true full-color output, but it is sufficient for our purposes. The beams are generated by individual lasers and combined with dichromatic mirrors into a single colinear beam. This is driven by the custom board connected to the Nexys board's PMOD port. The output of the laser module is directed into the galvonometer's input before being reflected out of an aperture on the front of the module.

II. Image Processing (Fischer)

To pre-prepare images and videos for display via the laser projector, images need to be vectorized and converted into a path of points the projector can understand. This processing currently takes place in Python using the OpenCV toolkit, and consists of a few steps:

- Rescaling. The image processing script is meant to handle any arbitrarily sized input, so it rescales everything to a uniform 512x512 square before any processing begins. This is done to keep the generated trajectory small and quickens the actual processing.
- Canny Filtering. A greyscale map of the image is generated, and then processed by a Canny Filter to detect edges.
- Trajectory Planning. The canny filter only outputs a rasterized image, not lists of connected points. OpenCV's `connectedComponents` function is used here, which returns lists of pixels on the same contour. These lists don't have any order to them, so they must be resorted such that adjacent pixels come after each other in the list. This process is incredibly slow as it occurs in Python instead of C++, like the rest of the OpenCV backend. Once the contours have their pixels properly reordered, a nearest-neighbors algorithm is run to find the next contour to add to the trajectory. The final trajectory is passed on to the next step.
- Recolorization. Each point in the output trajectory is colorized by sampling the color of the same (x, y) point

in a blurred version of the original image. It was found that blurring the image before sampling helped to produce a more predictable color of the contour, as blurring effectively averages neighboring pixel colors.
- Network streaming. Each point is encapsulated in our custom application layer network protocol, and then sent out via the conventional OS socket API at the FPGA. Earlier versions of the system also supported receiving information over a direct Ethernet link; The Scapy Python library was used for crafting and sending such packets.

This process can be be seen in the following images:

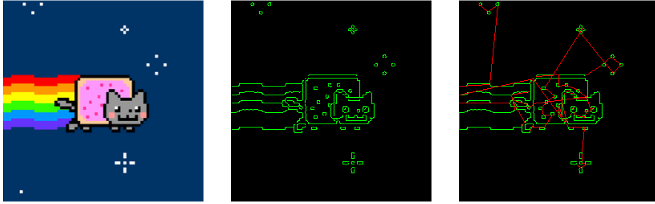


Fig. 1. Output of trajectory planning. On the left, the source image. In the center, the image to be rendered by the projector. On the right, the image to be rendered, but including the jumps between contours in red. Although the actual output is colorized, the trajectory here is shown as monochrome for clarity.

The image processing script can process either a video, a static image, or a webcam input by specifying a number of command line options. It was also possible to export the resulting output into either a `.coe`, `.csv`, `.png`, or `.traj` file, which was incredibly useful for debugging.

III. Networking Offload Engine (Jay)

The networking offload engine is a sophisticated, parallel-stack, pure-hardware device which maintains a connection to an Internet Protocol Version 4 (IPv4) gateway, and allows application layer data to be channeled to other hardware modules on the FPGA fabric over UDP.

The design of this module is broken into several parts, corresponding with the appropriate layer of the canonical OSDI model. We introduce these components from the bottom up.

A. The Physical Layer

The FPGA comes with an Ethernet chipset implementing the IEEE802.3 fast Ethernet standard, thus rated for 100 Mbps full-duplex operation. The chipset exports a number of configuration registers to the FPGA, in addition to implementing the RMII specification.

B. Media Access Controllers

We implement a Media Access Controller (MAC) layer to complement the Ethernet physical (PHY) chipset on the Nexys board, per the IEEE802.3 standard. Separate modules are devised for reception and transmission of packets to support full duplex operation, each translating raw Ethernet II packets back and forth from Ethernet frames. In order to populate the Frame Check Sequence (FCS) and verify it against received packets, an Ethernet checksum (CRC32-BZIP2) module is implemented. As is the case in modern NICs, the FCS is shedded after it is verified, in addition to other Layer-1 specific structures such as the Ethernet preamble.

This module is small, but difficult to test *in vivo* due to the complexity of the RMII interface and the resulting number of partitions on input signals. To test this module, we utilize the popular sniffer Wireshark to view raw packets which contain a passing FCS, and additionally configure a network card on the controlling machine to discard the FCS regardless of its correctness. A custom Ethernet protocol was implemented to accelerate this process; when a packet of this type is received, it along with its data is echoed back to the sender and never processed by the rest of the networking subsystem.

C. Address Resolution

The Address Resolution Protocol (ARP) is implemented above the MAC layer according to RFC 826 [1], to enable discovery and static protocol addressing for the LAN-connected FPGA. At compile time, a MAC address and desired IP address are specified within the system logic, and the host LAN is configured to allow static IP addressing outside of its DHCP range (if necessary).

The ARP implementation utilizes a single element table to store protocol and hardware addresses of the gateway. As such, the system is capable of not only responding to ARP requests for its own address, but updates its own table mapping dynamically in accordance to the algorithm specified within the RFC.

To simplify implementation, the ARP module examines a single large buffer (implemented as an array of byte registers and likely synthesized into block RAM), looking for relevant identifiers at appropriate offsets. As this is extremely difficult to replicate in simulation, the full dynamic system is tested via external sniffing (thorugh Wireshark etc.) and fuzzing, in conjunction with several different router configurations.

D. The Internet Protocol

A subset of IPv4 is implemented according to (the infamous) RFC 791 [2], and the header checksum logic is implemented in a separate module according to RFC 1071. When an IP packet is passed from the MAC subsystem, the IP block verifies the header checksum using this module and also checks to ensure that the packet doesn't utilize additional options, is not fragmented, and encapsulates a UDP packet. If none of these conditions are true, the packet is dropped.

These steps ensure the integrity of incoming data, and additionally ensure that the packet conforms to only our desired subset of IPv4. While this imposes some restrictions upon incoming data (e.g. width cannot exceed the Maximum Transmission Unit (MTU) of the underlying physical-layer medium), we have found that we implement a sufficient subset of IPv4 to where all major operating systems and all tested router hardware convey packets in a manner our device can understand.

Note that the Internet Control Message Protocol is not implemented, so bad IPv4 packets (e.g. bad checksum, exceeded time to live, etc.) are simply dropped rather than relayed back to the sender.

E. User Datagram Protocol

We implement the User Datagram Protocol (UDP) on top of our IPv4 receive layer in accordance with RFC 768 [3]. The checksum is currently omitted, in order to provide a constant-time depacketization capability, but the necessary logic to compute this is implemented according to RFC 1071 (using the same checksum logic as the IPv4 layer). Data at this layer is completely depacketized once received, and if the incoming port matches compile-time configuration, data is passed on to the display controller.

F. Performance Characteristics

The network stack is novel for a number of reasons: most significantly, since it is implemented in pure hardware, it is capable of depacketizing data in a bounded number of clock cycles. This number varies depending on the presence of certain IPv4 options, but ultimately falls well beneath the interframe gap required by IEEE802.3. This implies packets can be sent along to the laser projector at 100 Mbps reliably - in addition, since the transmission layer is implemented in parallel to the reception layer, full-duplex operation can be properly taken advantage of and impending transmission doesn't impact the integrity of incoming data.

Furthermore, assuming an IPv4 header length of 20 bytes, the system is theoretically capable of computing all checksums and performing all depacketization within 6 clock cycles. This easily puts it under the required time to implement the gigabit Ethernet standard, and potentially allows the implementation of more complex transport-layer protocols (e.g. TCP) using existing lower infrastructure.

IV. Laser Display Module (Fischer)

A. Framebuffer

As data is streamed off of the network and into the projector, incoming sets of (x, y, r, g, b) points are buffered such that the display controller can later scan through them and write them to the drive electronics. Originally we considered streaming directly into the framebuffer, but the incoming packet stream is of variable speed, meaning that the drive electronics could read from the framebuffer before the network stack is finished writing to it. This would produce a distorted frame.

To mitigate this, we use a pair of BRAM banks inside the display controller. When packets are being received, the network module will write into one BRAM bank, waiting for it to fill. Once the end of the frame is reached, the host computer will signal the FPGA to exchange the banks. The module will then save the current BRAM address as the end of the frame, and then toggle an internal `bram_select` line to indicate that the incoming and outgoing BRAM banks have been swapped. Points are then written out to the drive electronics from the freshly filled BRAM bank, and the newly decomissioned BRAM bank is made available for new points to be recorded into. Each BRAM bank is 20,000 addresses deep and 64-bits wide.

B. Packet Structure

The data enclosed in the packets is 64-bits wide, and follows the following structure:

cmd	x	y	r	g	b

- `cmd`: The control signal the FPGA uses to determine when to switch BRAM banks in the framebuffer. This field is set to 0x01 when data is being streamed in, and set to 0x02 when the frame is complete and the BRAM banks should be flipped. This field is 8 bits wide so that the entire packet would be 64 bits wide, which is conveniently the width of our BRAM buffers. This enables us to save the entire packet into BRAM without worrying about rearranging the packet.
- `x`: The position of the laser beam in the x direction. The DAC that feeds the drive electronics is 16-bit, so this field is also 16 bits wide.
- `y`: The position of the laser beam in the y direction. The x and y channels are identical, so this field is also 16 bits wide.
- `r`: The intensity of the red light at the point. Most images use 8-bit color anyway, so using 8 bits here seemed reasonable.
- `g`: The intensity of the green light. 8 bits wide.
- `b`: The intensity of the blue light. 8 bits wide.

These packets are stored in this same format in the 64-bit wide BRAM banks. It is also worth noting that the cmd parameter is only respected on incoming packets, and packets being read out of either BRAM bank ignore this parameter.

C. Display Controller

Once a complete framebuffer has been assembled and filled with packets following the above structure, its contents are sent to the drive electronics. For galvonometer control, this takes the form of two SPI-connected DACs, with the output buffered by a pair of unity-gain opamps. SPI was chosen for the interface because initial testing revealed that I^{2}C was much too slow to update the DACs fast enough to produce a non-flickering image, even when running at 400kHz clock speed. Using SPI also allows us to skip the address byte at the start of every transmission, increasing throughput, and using concurrent SPI buses also allows us to leverage the parallel nature of the FPGA for a tangible speed improvement.

Originally we considered using a non-unity gain amplifier stage on the output of the DACs to use more of the dynamic range of the galvonometers, which supposedly accept a voltage input between $0-15V$. Ultimately, this was decided against as documentation on the laser modules was virtually nonexistent, and it wasn't worth risking damage to the laser driver. This had the unexpected benefit of requiring the mirrors to undergo less displacement to produce the same image, effectively reducing

acceleration on the galvonometers and making a less fuzzy image. This does come at the expense of throw distance, and the projector must be nearly 15 feet from the screen in order to produce an image with any discernable detail. We considered this tradeoff reasonable to prevent any accidental damage to the hardware.

On the laser side, a PWM signal was used to control the intensity of each laser. We planned on using DAC-based output stage originally for controlling laser power, but it was deduced that a PWM signal could be varied faster than the x and y galvonometers could be, and PWM would suffice. This wouldn't be possible on a traditional microcontroller because of the difference in clock speed, and so a high-frequency PWM approach better highlights the unique nature of the FPGA and enables a less elaborate output stage.

Each channel of PWM output from the FPGA is fed into a potentiometer that sets the current on an opamp-based constant current source. This potentiometer can be adjusted to vary the maximum output power for compliance with EHS safety restrictions. The constant current source uses a NPN transistor to regulate the current through the laser diode, and was chosen instead of a MOSFET because of the absence of any gate-source or gate-drain capacitance. Since the BJT is a current-controlled amplifier, parasitic voltages cannot accumulate across the gate capacitance and accidentally turn on the laser diode if the laser driver is tampered with. The lack of gate capacitance also enables a faster switching time, and the lack of any (significant) switching losses.

This design was implemented on a custom PCB, designed in Altium and manufactured by PCBWay.

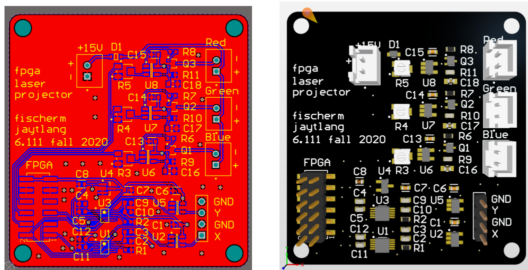


Fig. 2. The 2D and 3D renders of the board in Altium.

V. Retrospective

In hindsight, we learned a few rather important lessons during the development process:

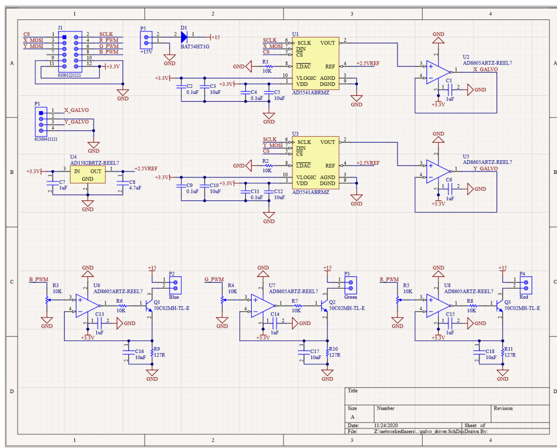
- Simulations are incredibly useful when the input space is limited. For instance, most of the display control was verified with simulation before a bitstream was ever generated, and even then it worked the first time on hardware. This saved a significant amount of time.
  However, simulation of modules with more complex input spaces (e.g. the Media Access Controllers) can get unwieldly, especially when the variation of these inputs over time is critical to assessing system correctness. For this reason, it's often useful to utilize an alternate method of design verification - in our case, the NIC FCS discard combined with the custom Ethernet-layer echo protocol proved invaluable.
- Doing trajectory planning properly is super hard. There's some rather fancy graph-based algorithms for computing a proper path through a set of points, but we didn't feel like spending a bunch of time on that really captured the spirit of 6.111 - and so we consider our nearest-neighbors algorithm sufficient. Those algorithms would take longer to run than our current system (which is slow enough as is) and would probably require porting to a compiled language to make it run in any reasonable amount of time. This would be the most useful improvement to the image processing if more time was had.
- System interfaces are anathema to large systems - whether they lie in software or hardware. Accordingly, if two team members are implementing two different parts of the system, picking a well defined interface, establishing it clearly, and utilizing it throughout design iteration is invaluable. As we developed our system, we determined a suitable interface between the network offload engine and the display controller fairly early on - and obviously the interface between the network offload engine and software image generation is well defined through tens of RFCs. This way we were able to re-integrate the design several times utilizing the same interface, in a bug-free manner and a (surprisingly) rapid pace
- Don't buy sketchy USB Ethernet adapters off the internet, because sometimes they don't implement the slower 100 Mbps standard correctly. Make sure you've got old and established hardware, or better yet, an actual Ethernet port on your device you can manually modeset with `ethtool`.

VI. Appendix A - Laser Safety and Procedures

Lasers are sketchy. Especially cheap, poorly-documented ones procured from the Internet. The RGB laser module that has been procured is a little short on trustworthy specifications, and as a result MIT's Environmental Health and Safety office (EHS) has been consulted to operate the laser safely. To ensure compliance with their requirements, the laser:

- Does not output more than 5mW on any channel. This classifies it as a Class 2M laser.
- Is mounted inside a tamper-proof, interlocked enclosure that cuts power to the laser when opened. This is accomplished with a pair of microswitches in series with the 120V supply inside the enclosure.
- Has been tested by EHS to verify that the laser output spectra is not harmful to humans.

VII. Appendix B - Driver Board Schematic



VIII. Appendix C - Getting Networking to Work

In order to hook the NALP up to your LAN (or my networking stack in general), you'll have to do the following:

- *Optional:* Assign a MAC address to the FPGA. Currently I have my FPGA co-opting a MAC address from a Raspberry Pi I own, for ease of debugging packets by brute-force equality. You might want to change this (especially if I'm on campus or have my raspi plugged in on your network), or better yet, set up your own little range of MAC packets nobody else has claimed yet. This is pretty easy to do, and currently defined in `include/offsets.svh` as well as `hdl/mac_rx.sv`.
- Assign an IP address to the FPGA. This one's pretty simple - figure out the DHCP range of your network and stick the IP address (`include/offsets.svh`) outside of the DHCP range but within the subnet. This should work out of the box...if not, send the IP address you've chosen some ARP requests (e.g. with `arping`) and verify that the system responds to ARP requests. Doing this will force the gateway to recognize the device as well, which might not happen the first time around.

  This might be more complicated on MITnet. Registering a static IP with IST should work without further configuration, and if that's too much work, a Linux machine can be configured as a router, and thus as a gateway to MITnet. This isn't too hard to set up if the Linux box has 2 NICs - there's lots of documentation online - and should work if you port forward your desired port on that Linux machine to the FPGA. You'll need a static IP if you want to talk to this thing over the wider, non-MITnet internet though.
- If you're sending packets over the internet rather than just a LAN (MITnet doesn't fall into this category), you'll have to port forward through your local router.
- *Optional:* Change the UDP port this thing listens on if you feel like it. This is also in `include/offsets.svh`.

Note that if you just wanna do Ethernet-level networking, no router configuration or other black magic is required, besides modifying my network stack to do just that. Just plug your board in and start sending packets, using the Media Access Controller modules and a shared packet buffer without any of the Internet Protocol / Address Resolution business. Ethertype 0x1234 is used for echo functionality, so try sending packets there (e.g. with raw sockets or Scapy) to sanity check your configuration.

References

[1] An Ethernet Address Resolution Protocol. RFC 826, November 1982.
[2] Internet Program Protocol Specification. RFC 791, September 1981.
[3] User Datagram Protocol. RFC 768, November 1982.