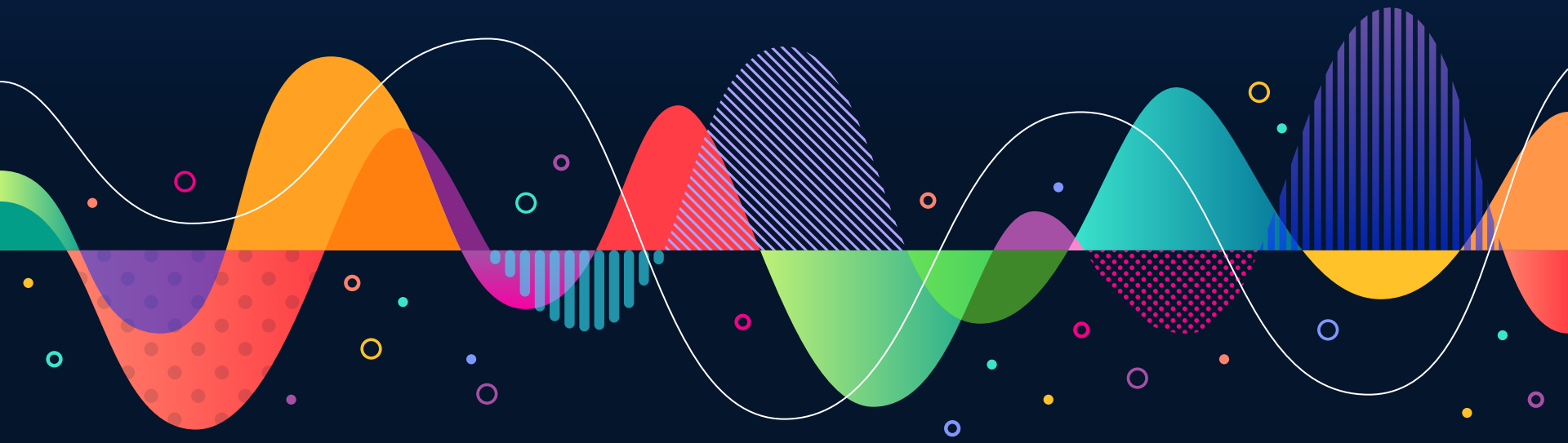
Hyperspectral Image Compression & Decompression On FPGAs

Final Demo Presentation



Team HEALTH (Hyperspectral Earth Acquisition of Longwave Thermal Heat)

is:

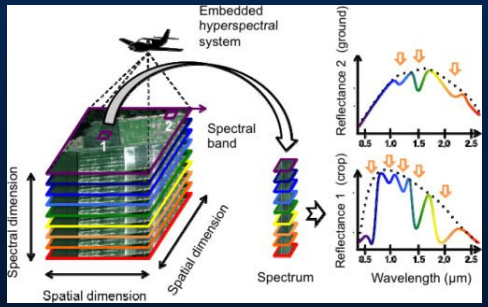
Brytni Richards

Dylan Vogel

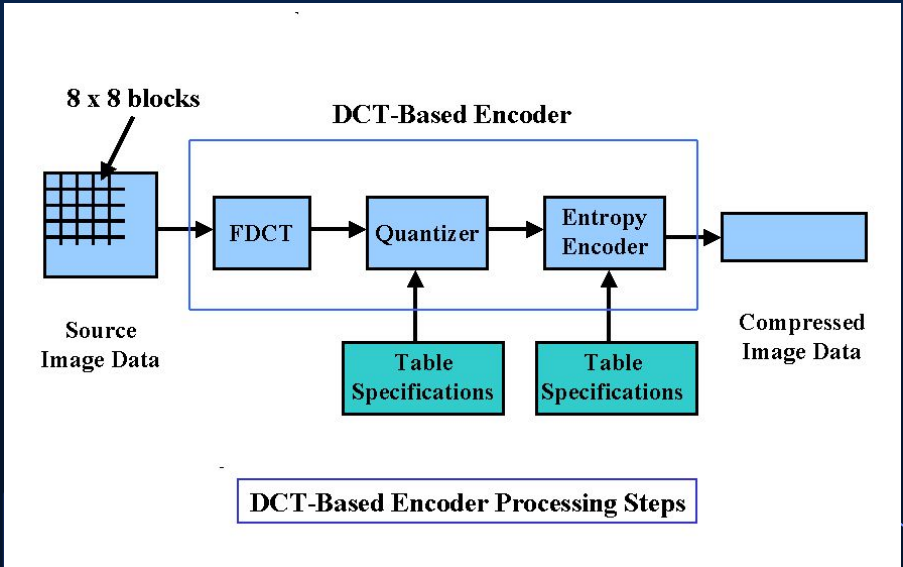
Lorna (Xi) Lan

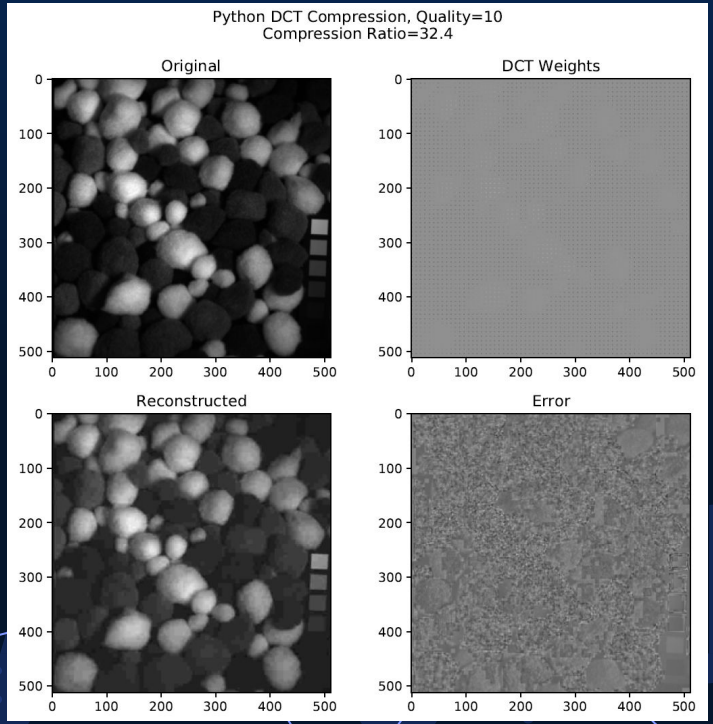
Background on Topic

- Explanation and Representation
  - 3D hyperspectral image compression and decompression pipeline
  - Accelerated DCT compression with FPGA
- Motive and Purpose
  - Utilization on a small satellite payload
    - Time/size/memory/RF constraints
  - Quickly decompress a large image dataset
  - Demonstrate feasibility and assess difficulty of design



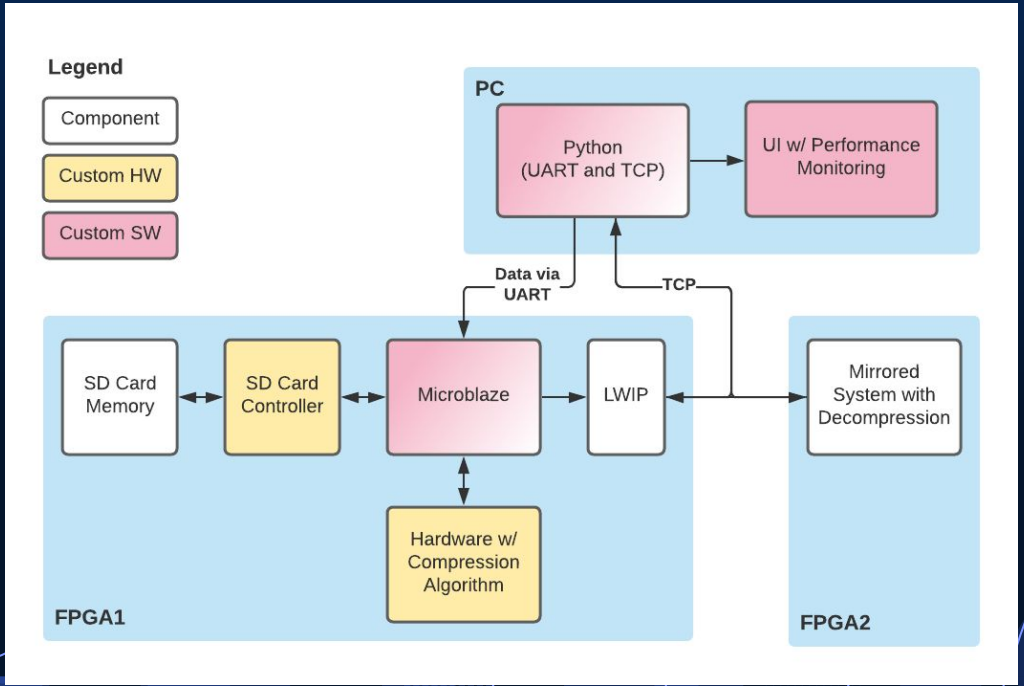
Compression Algorithm





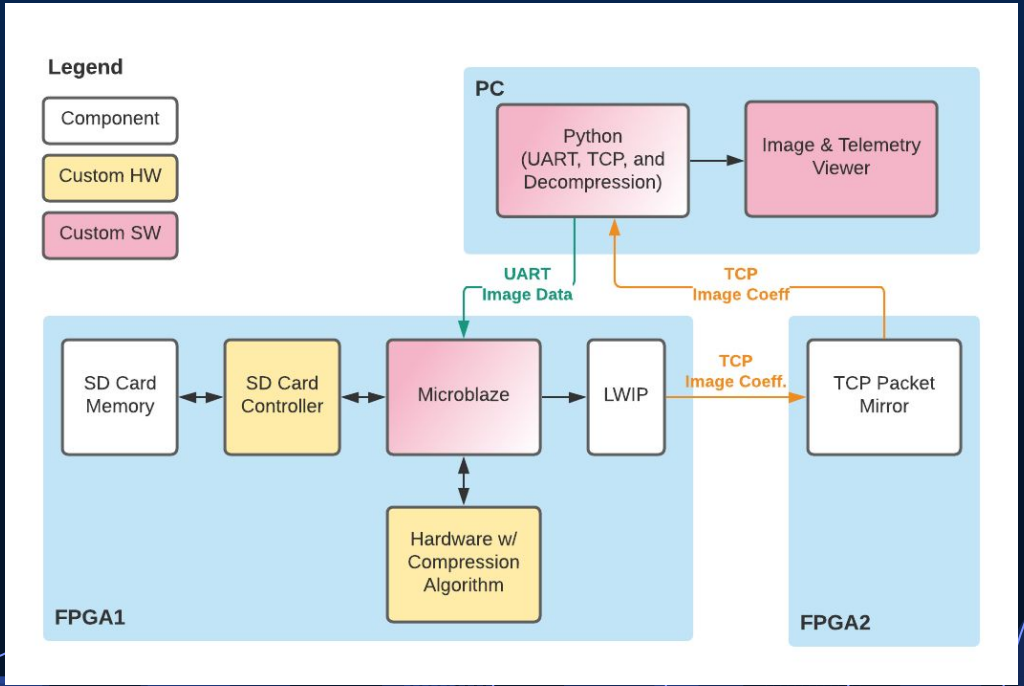
Initial Goals and System Block Diagram

- Simulate a remote sensing satellite mission with RF link bandwidth and large data cube constraints
- Transfer images from PC to FPGA1 over serial (UART) and compress the images
- Transfer compressed DCT coefficients over TCP to FPGA2 for decompression
- FPGA-FPGA TCP communications stands in for the RF link
- Host PC mimics a TT&C link which would also be present in a real system
- PC collects and visualizes performance telemetry



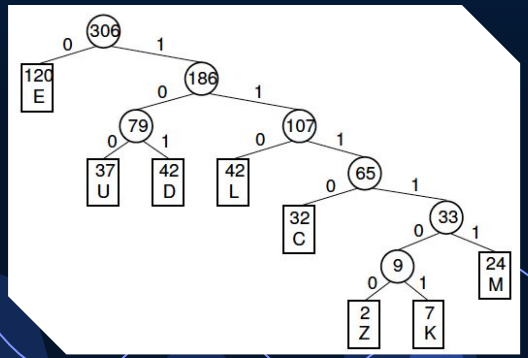
Final System - Overview

- Mostly the same, although we'll get into some low-level differences later
- Note that FPGA2 has been replaced by a TCP mirror. We ended up de-scoping the hardware decompression due to the complexity of the compression alone
- Python now performs image decompression
- Signal paths, UI modified slightly to indicate that telemetry flows unidirectionally from FPGAs to PC



Problems We Encountered (0/2)

- Underestimated the amount of time required for integrating the compression pipeline
- Likewise, de-scoped Huffman encoding of the DCT coefficients
- SD card speeds are slow (spends ~a second processing the write and AXI transfer takes time)



Problems We Encountered (1/2)

- Underestimated the amount of time required for integrating the compression pipeline; various bugs with the DCT block itself, Xilinx AXI DMA, and Xilinx AXI Stream FIFO. All tested individually (and working!) but not working together
  - **Change:** set up second FPGA as TCP mirror and use Python for decompression
  - **Result:** only build compression block, although still a significant amount of work. Decompression block could essentially be "slotted in" since FPGA 2 otherwise has the desired architecture
- Likewise, de-scoped Huffman encoding of the DCT coefficients
  - **Result:** coefficients could be further minimized by using variable-width encodings for each range of values, although adds significant complexity for small gains in our system

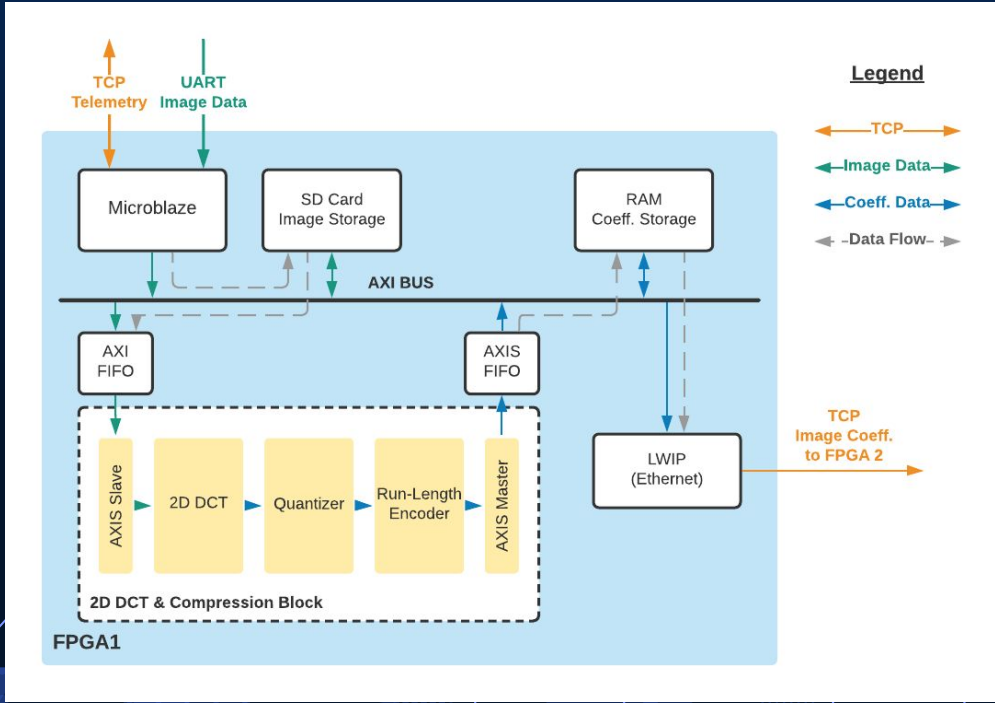
Problems We Encountered (2/2)

- SD card speeds are slow
  - **Change:** Decreased demo picture size to accommodate demo time restrictions
  - **Result:** Demo doesn't display entire dataset picture implementation but still illustrates the basic system without optimizations

SD card sample waveform on a 25MHz clock

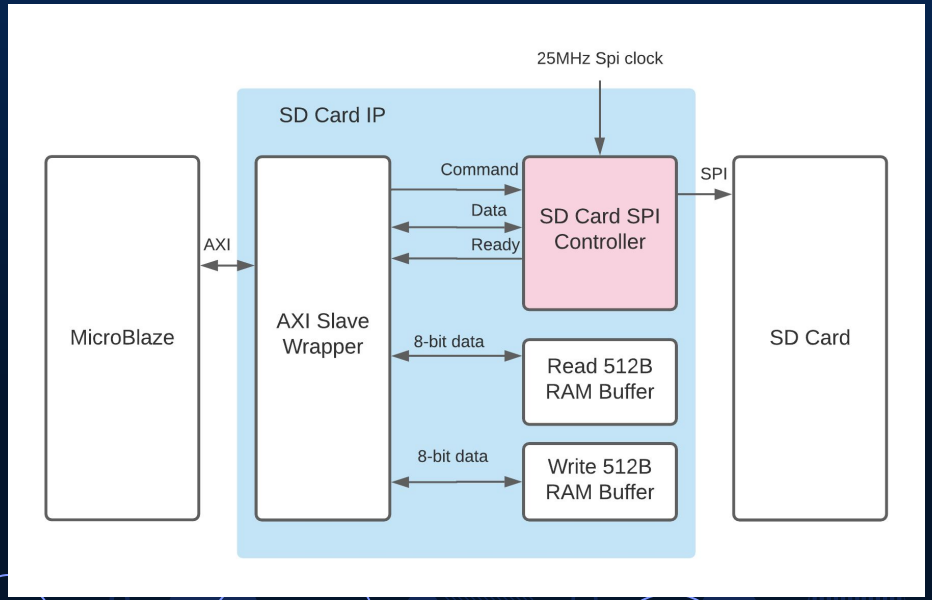
Closer Look: FPGA1 & DCT Block

- Python sends original image to FPGA1 over UART in block order.
- Microblaze writes pixel values to SD card
- Microblaze passes pixel values to AXI FIFO, which sends/receives AXI Stream from DCT
- DCT calculates compressed coefficients and writes back run-length encoded values
- Microblaze assembles coefficients according to a packet format we defined and transmits to FPGA2



Closer Look: SD Card

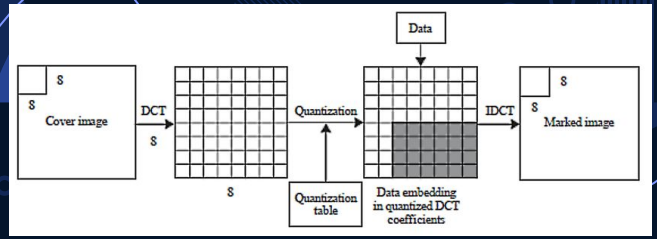
- SD Card SPI Controller was borrowed from [1]
  - Made some minor modifications to work for the newer SDHC/SDXC card version
- For Microblaze, SD module is treated as memory mapped IO
  - Commands are executed based on AXI Slave register values
- SD card read/writes data in 512-byte blocks
  - Data is stored in a RAM buffer



[1] Gim P. Hom. 6.111 Introductory Digital Systems Laboratory. Fall 2015. Massachusetts Institute of Technology: MIT, http://web.mit.edu/6.111/www/f2015/tools/sd_controller.v.

System Components

- SD Card:
  - **We built:** SD controller modifications for newer version, RAM buffer for write/read data
  - **We borrowed:** SD card SPI controller, AXI Slave interface template
- Networking:
  - **We built:** UART transfer between host PC and FPGA, TCP pipeline and a simple server
  - **We borrowed:** UART, LwIP low-level drivers
- DCT
  - **We built:** 1D DCT stage, transpose double-buffer, quantization stage, zig-zag decoder, run-length encoder, and AXI-Stream interface
  - **We borrowed:** Xilinx AXI Stream FIFO block and C drivers

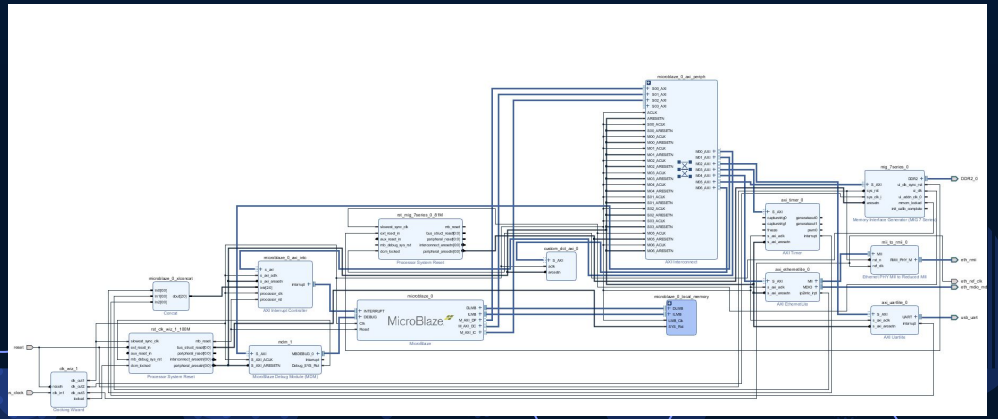


System Components

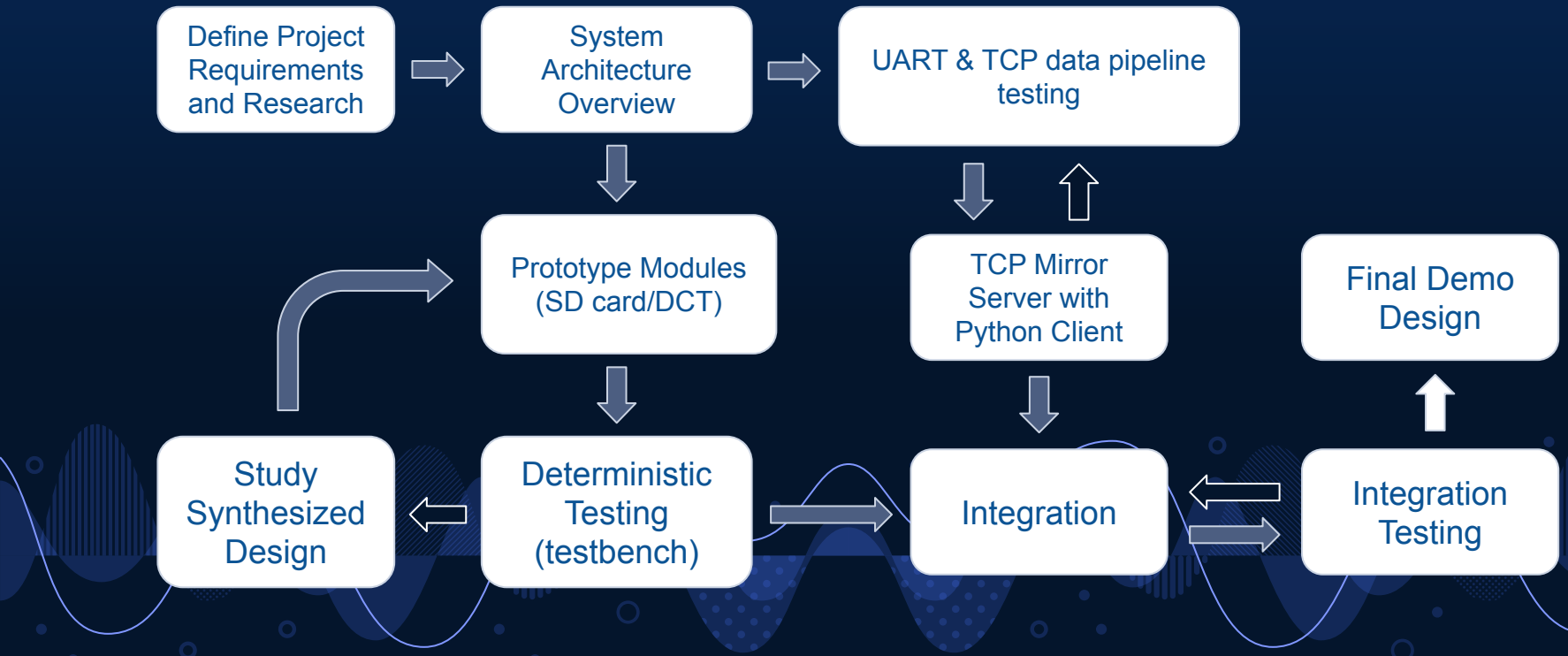
- Microblaze architecture:
  - **We borrowed:** Microblaze BRAM, MIG, AXI Interconnects, LWIP, UART, etc.
- Visualization & Python Decompression
  - **We built:** all of it, using curses, matplotlib, socket, and numpy Python libraries

```
Showing Telemetry Data
----------------------------------

Bandwidth Telemetry: 1000
Compression Ratio: 4000
```



Design Process - Project Development



Design Process - Human Collaboration

- Collaboration
  - Regular meetings twice a week
  - Clear division of labour
  - Individual modules were tested before integration with the rest of the system
- Other Important Factors
  - Started integration early (~2 weeks before final demo)
  - SVN repository for individual parts & integration
    - Version control

What We Learned

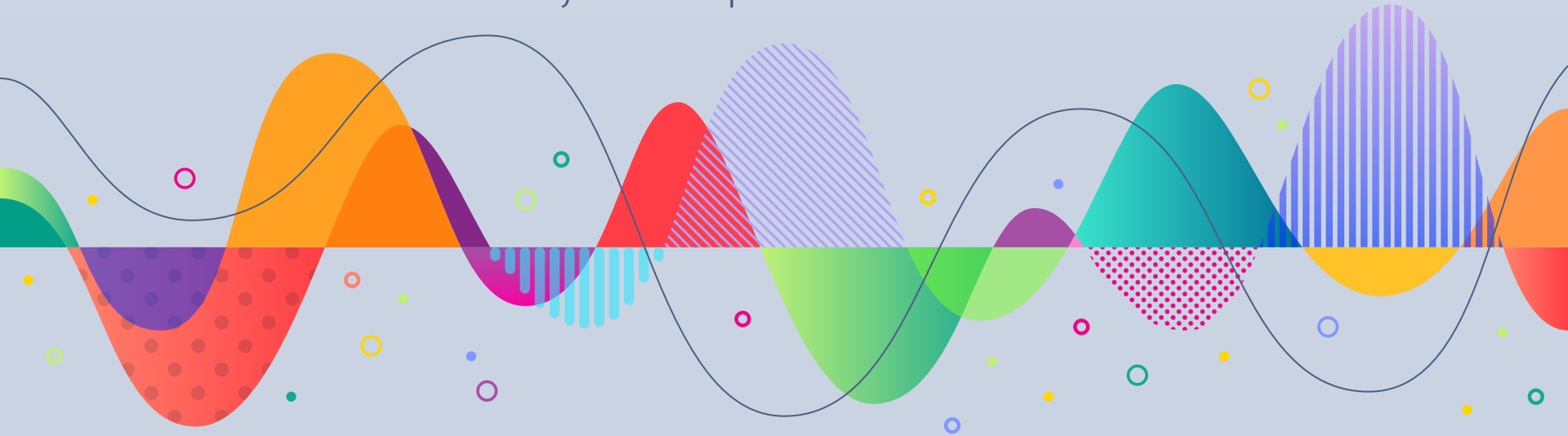
- **Brytni**
  - I learned how to use the Verilog tools and platform, along with the SD card communication protocol details. Additionally, I found that self-written IP modules are much more flexible than the Xilinx provided ones
- **Dylan**
  - I learned a ton about Verilog and FPGA design. In hindsight, a smarter debugging approach might have been to build each DCT component as AXI Stream blocks
  - I also learned that “working” Xilinx components should be tested early to avoid surprises
- **Lorna**
  - I learned how to write communication packets and callback functions
  - I (re)-learned all the C programming practices and bare-metal programming, had a lot of fun fiddling with interrupts and driver code
- **Together we learned:**
  - You can never start integrating early enough

DEMO TIME!

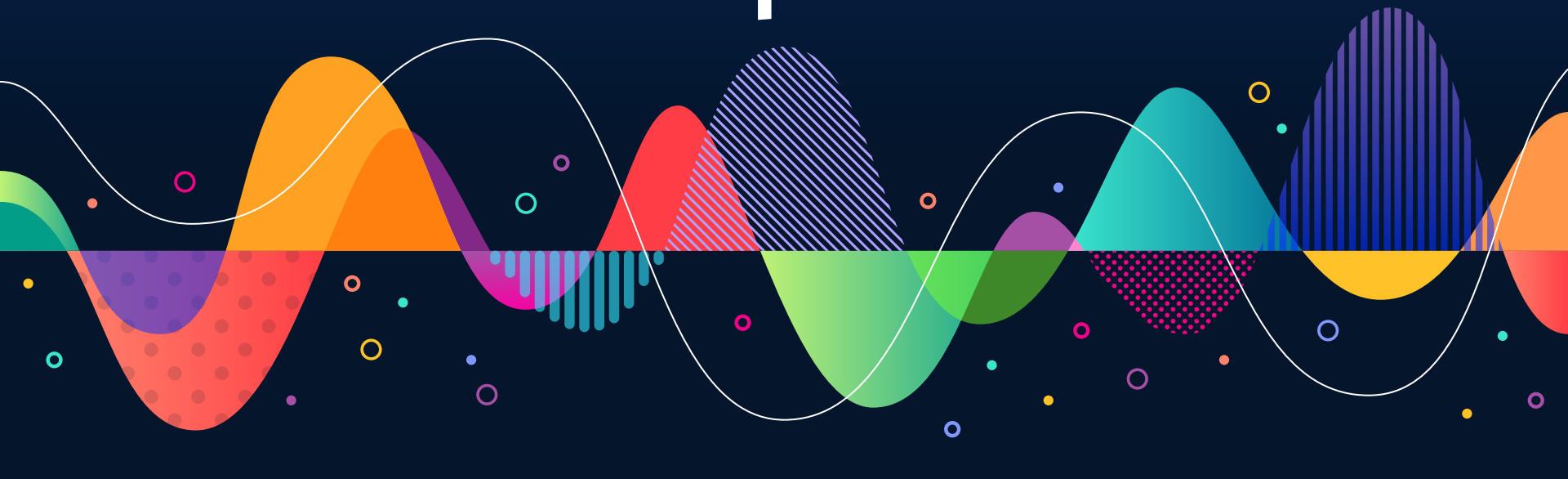
- **First part of the demo (DONE)**
  - Transfer 256x256 image via UART
  - Write to SD card
- **Second part of the demo**
  - Read from SD card (showing first 10 read values for verification)
  - Pass pixel values to DCT block
  - Store coefficient from DCT block
  - Transfer coefficients over TCP to second FPGA
  - Pull data from second FPGA via PC client Python
- **Third part of the demo**
  - Show a decompressed image we pulled from running the DCT block individually

Q&A Time!

Any and all questions welcome.

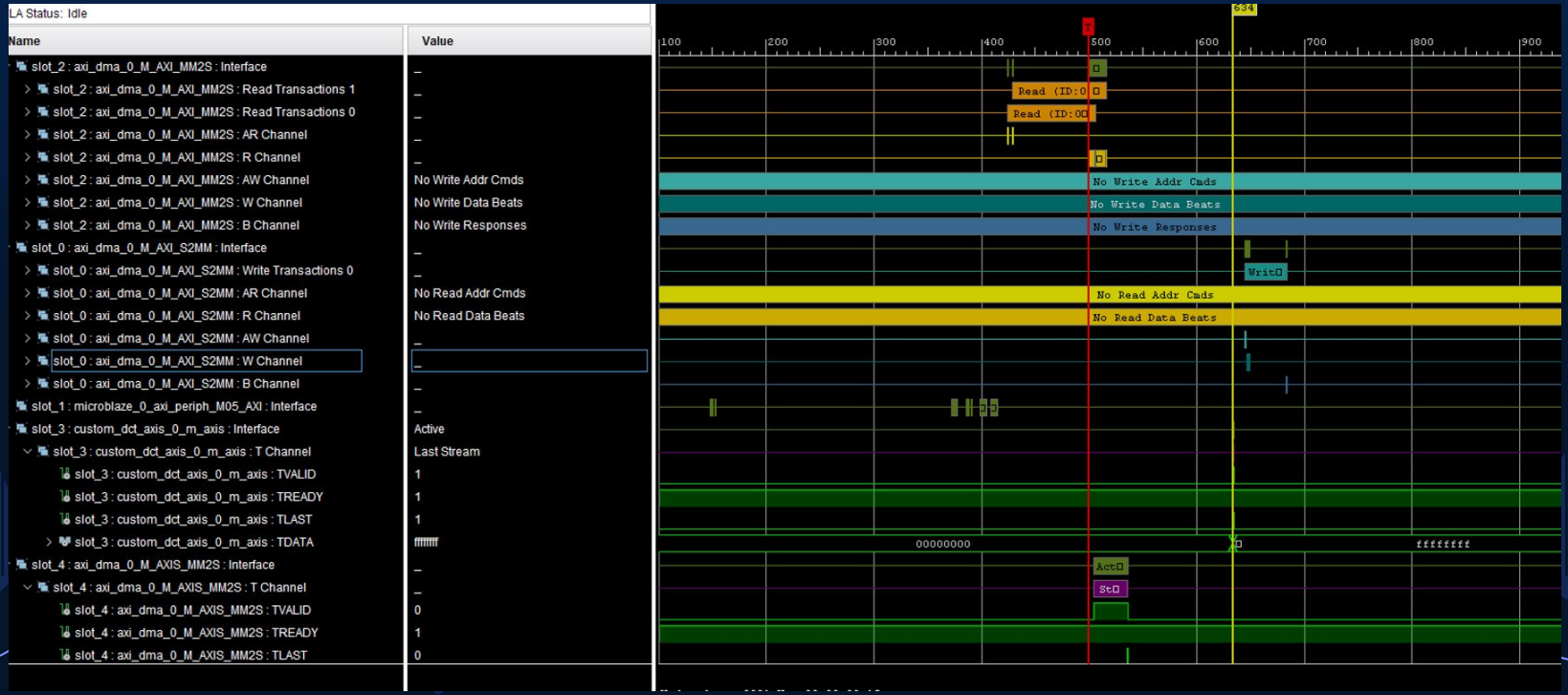


Backup Slides

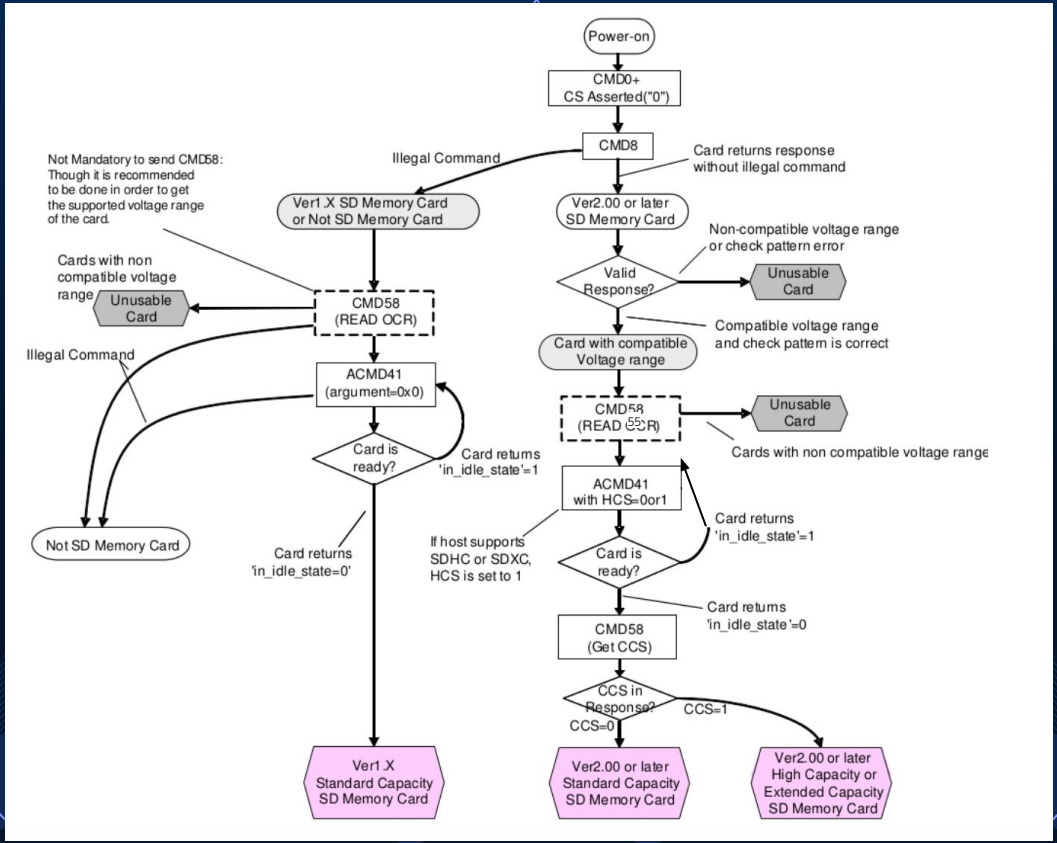


DCT Debugging Waveforms

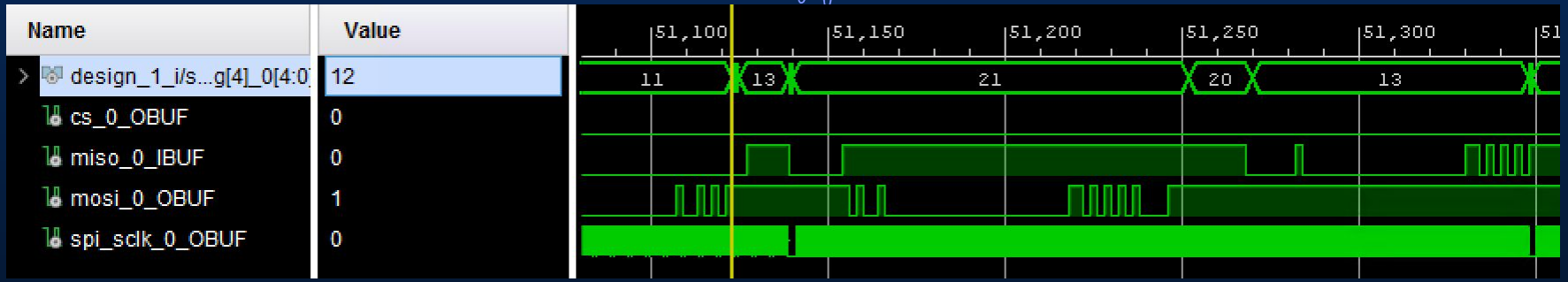
ILA Waveform of our block transferring stream data back



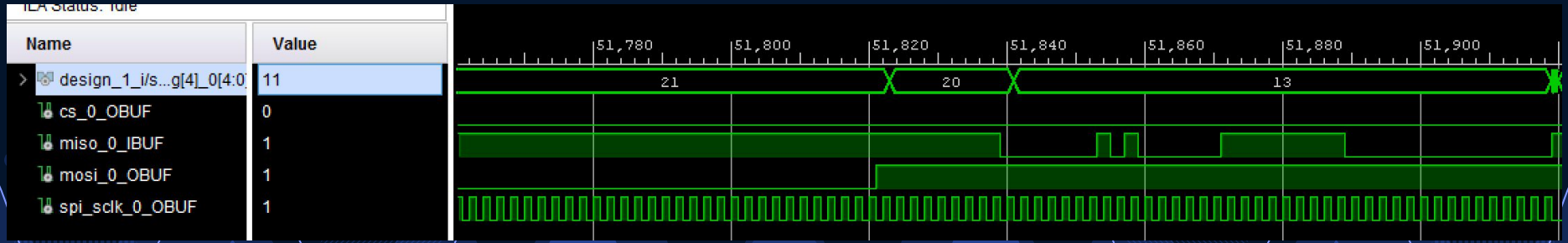
SD Card Command Flow



SD Card Debugging Waveforms

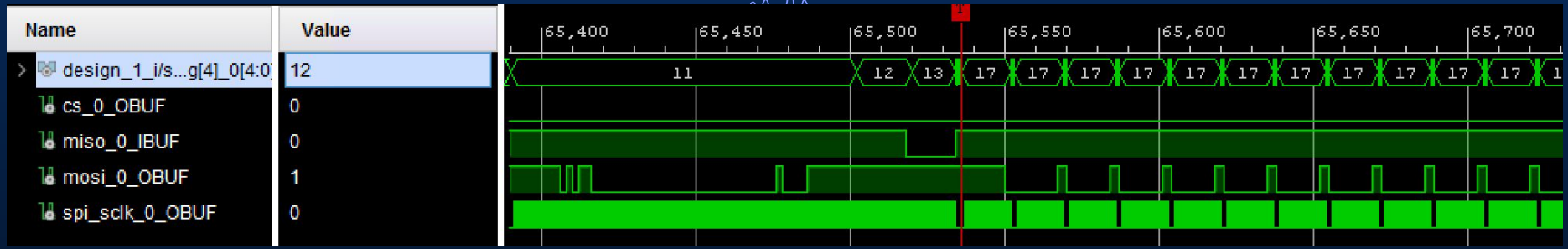


CMD8 - Valid return response: mirrored

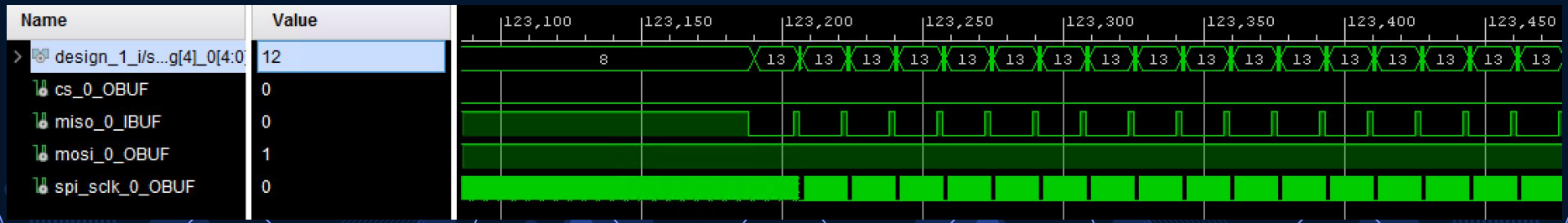


CMD58 - Card (voltage) operating conditions (currently busy state)

SD Card Debugging Waveforms



Writing ones to SD card



Successful read back from SD card

Implementation Plan

Brytni: SD card implementation & data handling, 25% of performance visualization

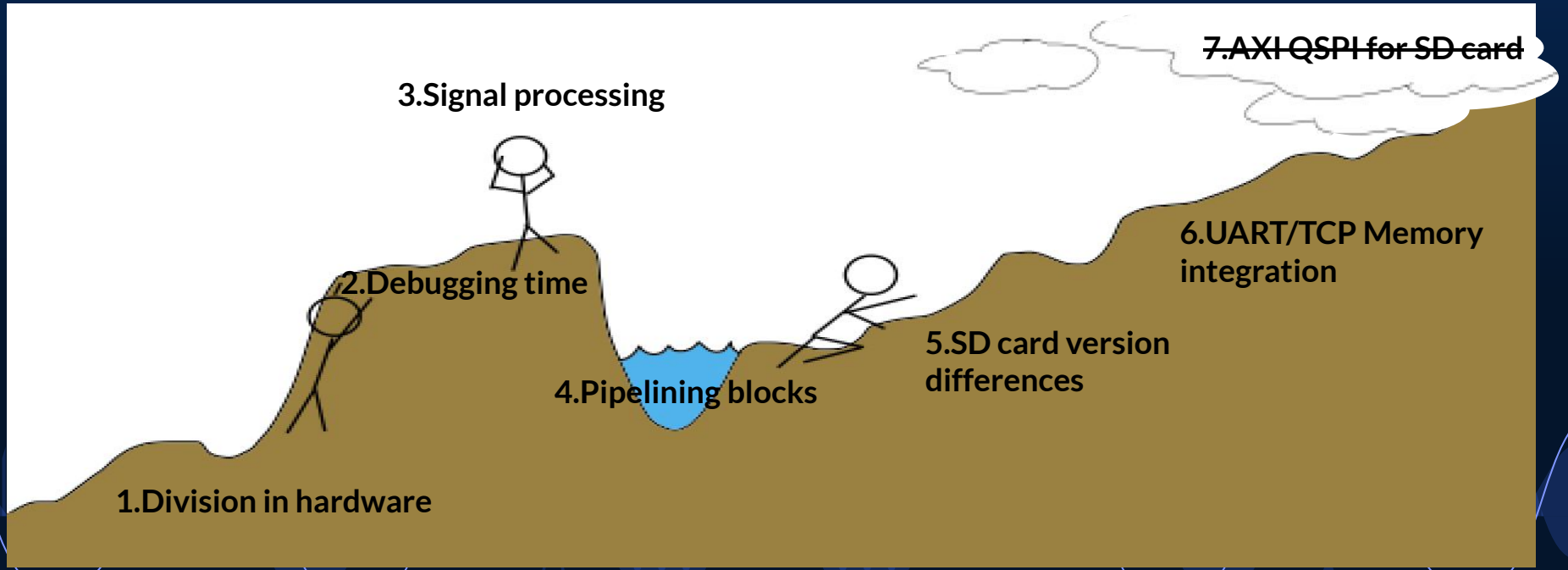
Dylan: Image compression/decompression blocks, 75% of performance visualization

Lorna: Network connectivity (FPGA-FPGA and FPGA-PC TCP/UDP, UART to PC)

All: System integration (hard)

Milestone #	High-Level Target
1	Research phase
2	SDIO/SPI physical layer implementation, DCT running in testbench, PC to FPGA via UART
3	SD card testbench, visualization started, initial network implementation done
4	Begin integrating SD card, microblaze, and compression/decompression blocks.
5	Finish SD card image transfer and network pipeline, pull performance monitoring stats
6	Road to final systems integration, time for exceed goals

Challenges Encountered



Future Work

- Complete write and read SD card functionality (finished)
- Integrate mass storage with SD card
- Finish integration of 2D DCT custom block with AXI interface and data transfer (Master and Slave interfaces, probably)
- Have the full data transmission chain done (and RAM actions in between)
- Get a working terminal visualization going