To the 60's and Back

Authors: Christopher Bernard, Jae Woong Choi, Donovan Gionis
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of computing and displaying calculations in the original Apollo Guidance Computer Architecture in displays for educational purposes. Our project will be part of an exhibit that displays the wonders of the Apollo mission while conveying modern improvements in technology. While the current education displays will use original computer which is 70 pounds and runs at 1.1Mhz, our implementation uses modern technology, including an open-source-able FPGA core, and it weighs under 3.5 pounds and runs at 5 Mhz. Our implementation runs faster and is more portable than the original display, and therefore more available to the general public.**

***Index Terms*—Computer Architecure, Digital Design, Education**

1 INTRODUCTION

The Apollo space missions are among the most important accomplishments in human history. To this day, the influence of these missions can still be tangibly seen. In the words of word-renown tech entrepreneur, Elon Musk, " I think Apollo 11 was one of the most inspiring things in all of human history. Arguably the most inspiring thing. And one of the most universally good things in history. The level of inspiration that provided to the people of Earth was incredible. And it certainly inspired me. I'm not sure SpaceX would exist if not for Apollo 11." [5]. Similarly, Jeff Bezos built his company, Amazon, so that he could eventually travel to space, having been inspired by the Apollo missions [6]. We capture Apollo's historical significance and ongoing influence in a portable educational and museum display of the Apollo Guidance Computer (AGC) and its DSKY interface.

As opposed to the previous Apollo Exhibit we use a modern SoC and custom PCB to decrease the weight of the project from seventy to under five pounds. This weight reduction will make our design an easy portable exhibit that can be moved from smaller classrooms and exhibits easily making our project accessible to all. Our modern physical redesign also is advantageous over software simulators because it brings the project into the real physical world which is what museums are all about. The goal of our project is to make a physical redesign of the AGC for educational and inspirational purposes.

2 USE-CASE REQUIREMENTS

We intend for our device to be used as an interactive exhibition piece in a museum or as a teaching tool in academia. Users should be able to learn about the role played by the AGC in the Apollo program, get a feel for how astronauts interacted with the computer, and understand the influence its design had on subsequent developments in computer engineering.

The Display-and-Keyboard interface (DSKY) is arguably the most iconic original piece of the AGC system; it allowed astronauts to issue commands and provide data necessary for the computer to perform crucial guidance calculations and pilot the spacecraft. Thus, we sought to recreate the DSKY on a custom printed circuit board (PCB) with the same set of illuminated displays and push buttons as the original. This requires 14 LED lamp indicators, 25 LED alphanumeric displays, and 19 key switches.

Given that the most high-profile Apollo missions brought humans to the moon and back, it is crucial that people can use our DSKY to run programs that make necessary computations enabling a spacecraft to leave earth's orbit, attain the orbital radius of the moon around the earth, and enter an orbital path around the moon. Spacecraft actuators, such as rocket boosters that burn fuel, are out of scope given our timeline of development and the target application, but a simple external monitor can convey the results of those calculations being used to change the orbit of a simulated spacecraft relative to earth and the moon. The simulation program must display the orbits of the three bodies in real time, in a format that resembles a mission control console.

In order to convey the AGC's significance and impact in the world of computing, a full software emulation is insufficient. We decided to write a program in the original AGC assembly language that could respond to user commands issued via the DSKY, read in simulated mission data, perform the pertinent calculations, and then provide results to our DSKY and simulation display program. The assembly program would be run on a processor of our own implementation that processes a subset of the original instruction set. This requires support for 33 AGC instructions, 15 I/0 channels, one's complement arithmetic, 15-bit word size, and 16 of the original CPU registers.

Our processor must be implemented using the hardware description language (HDL) SystemVerilog, employ modern pipelining techniques in design, and be programmed to a Cyclone V FPGA. Furthermore, our system must integrate a modern PCB and serial communication over UART. Using modern digital design tools and integration techniques to implement this legacy architecture conveys to users the sheer amount of progress that has been made following, and in response to the deployment of the AGC.

The use of modern technology allows our device to be smaller and lighter than the original, ensuring ease of trans-

portation and installation. Excluding the monitor, the device must fit within a cubic foot, and weigh less than 5 lbs. A plastic enclosure is necessary to prevent destruction of hardware components through repeated use. The device should also be able to function properly in ambient temperatures around 25°C.

Illuminated displays and push buttons on our DSKY must be updated and scanned, respectively, at rate of around 100 Hz. This is fast enough to ensure users' interaction with the device is fluid and without lag, but also slow enough that congestion of UART traffic with the FPGA is avoided. 50 MHz, a common rate for FPGA development boards, is the target clock rate for all modules on the FPGA. It is significantly faster than the clock rate of the original AGC, and aids in ensuring smooth interaction with our device.

3 THEORY OF OPERATION AND ARCHITECTURE

The final setup is shown in Figure 1. The user interacts with the system via our custom PCB interface. Utilizing the interface's keypad, status lamps, and alphanumeric displays, the user is able to run and monitor the status of mission-oriented programs, view the elapsed time since power-up, or perform a lamp test. Under the hood is a small network of processing cores, digital circuitry, serial communication, and discrete I/O that make these things happen.

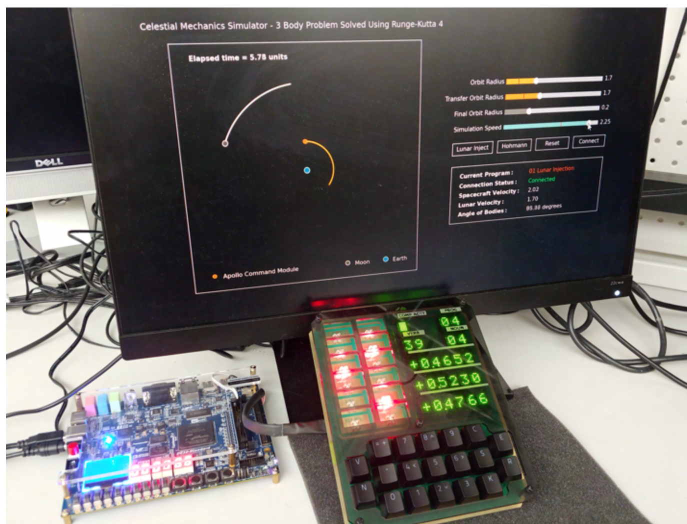

Figure 1: Final Setup of the AGC FPGA, DSKY, and the Demo Program

3.1 Nouns and Verbs: Establishing Context

While interacting with the DSKY, astronauts of the Apollo missions would issue commands and receive automated prompts in a so-called "verb, noun" format, indicating an action (verb) to be performed by/upon a device/data (noun). All possible nouns and verbs were stored as 2-digit decimal numbers that could be entered via the keypad and displayed on the segments. For example, a verb 16 (continuous display - decimal) and noun 65 (mission time) combination would result in a constantly updating display of the seconds elapsed within the mission, in decimal form. Other commands could lead to a flashing verb, noun display aimed at prompting the astronaut to manually enter necessary data.

We devised a noticeably condensed list of nouns and verbs that our users may enter to activate functions within our device's demonstration capabilities (see Figure 2). Most of our commands are verb-only, with the exception of that which runs a mission-oriented program, with which the user must also enter a noun to indicate the particular program they wish to run.

VERB	NOUN	Description
05	N/A	View mission time
06	N/A	Test the displays - *Default on Reset*
07	N/A	Clear the displays
39	PROG	Run Program Number # PROG (NOUN)

Figure 2: Table of pertinent verb, noun pairs

3.2 Theory of Operation

Upon powering up the device, the system enters 'lamp test' mode (Verb 06) by default. It is now awaiting the entry of a verb/noun command. The user must press the "VERB" key and then type the two digits of their desired verb entry. The digits will fill in the "VERB" field on the display, and when ready, the user must press the "ENTER" key to confirm their selection. Entering a noun carries the same procedure, although it must begin with a press of the "NOUN" key. If the verb 39 (Run program XY), has been confirmed, the user must enter a valid program number as the noun. Otherwise, the noun 00 must be entered.

At any time, the user may enter a verb 07 and noun 00 to halt execution and enter an idle mode, or they may enter any other valid verb/noun combination to run something else. As our CPU runs user-selected programs, the illuminated displays on our DSKY are updated with pertinent information, and calculation results are fed to the simulation display program, which uses those numbers as parameters for the moving bodies on screen. Additionally, the simulation display program provides simulated mission data to our CPU that is necessary for completing the calculations.

3.3 System Architecture

Making all of this happen is an FPGA development board, a PCB, and a personal computer. The development board is an Altera DE10-Standard with a Cyclone V SOC that contains both a Cyclone V FPGA and an ARM core. Despite initial plans, our final system does not use the ARM core, so any similar development board with a Cyclone V FPGA could also be used. The PCB is of our own custom design and consists of an ESP32 microcontroller as well as the array of keys and LED displays that make up our user interface. The FPGA communicates with the ESP32 via UART. Additionally, a personal computer is used to run the simulation display program, and it exchanges data with the ESP32 over Bluetooth. (The following content refers to the block diagram in Figure 3 on the next page.)

The Cyclone V FPGA has multiple modules "baked" into it. Of course, there is our implementation of the AGC CPU, which writes to and reads from the RAM, but it must also read data and instructions from the ROM. Both the RAM and the ROM are parameter-configured instances of the Altera 'altsyncram' IP megafunction. There's also an I/O unit consisting of input registers (output registers being internal to the CPU) and a UART interface. The UART interface contains logic for transmitting and receiving serial bytes of data, from the output registers and to the input registers, respectively. Logic of our own design transmits and receives this data at the byte level, interfacing to a bit-level transceiver that comes courtesy of Ben Marshall on GitHub. This UART interface allows our CPU to exchange data with the ESP32 on the DSKY PCB.

The ESP32 runs a program that handles key press scanning, LED display drivers, interfacing to the UART connection with the CPU, and interfacing to the BlueTooth connection with the PC running the simulation display program. I2C is used as the communication medium between the ESP32 and the DSKY's LED displays, whereas the keypad is a matrix wired to discrete I/O. Bluetooth allows the ESP32 to send commands and data to the simulation display program, some of the data being forwarded are computation results from the AGC CPU's output registers. Pertinent simulation data is also forwarded to the AGC CPU by the ESP32 to be used in computation.

4 DESIGN REQUIREMENTS

We have an assembler capable of processing AGC source code and assembling it into binary code that can be stored within the ROM on the FGPA, enabling our CPU to process programs that perform calculations and respond to user commands.

At the core of the design is our version of the AGC CPU; implemented in SystemVerilog, synthesized and placed onto the Cyclone V FPGA using Intel Quartus software. The system's functionality requires a CPU implementation that supports 33 of the 49 original AGC instructions, and they are listed in figure 4. The peripherals used to communicate to the CPU require at 5 functioning I/O channels. This is needed in order to do the calculation asked for by the DSKY operator and communicate them back to the DSKY display. Lastly, in order to achieve smooth-running demonstrations and quality user interaction with the system, we would like our CPU to support a 50Mhz clock rate which will demand a sub-200 nanosecond critical path.

The user interface in our system includes a custom-designed printed circuit board (PCB). Inspired by the original DSKY (Display and Keyboard) interface of the Apollo missions, the PCB contains 14 LED indicator lamps, 25 LED displays, and 19 mechanical key switches. Because of this we use a micro controller to communicate to these using I2C. This communicates with the AGC over UART, the DKSY user interface over I2C and the mission display monitor over serial Bluetooth.

Lastly our mission simulation display program is a python script running on an external PC. It communicates with the ESP32 over Bluetooth. The script models the three body problem and displays it on the screen. It sends information that would be captured by instruments on the original Apollo missions, and receives info related to the commanded movement of the simulated spacecraft, then updates the monitor accordingly.

Our design needs to be well protected. Ideally by transparent plastic so that the circuitry could be displayed but not harmed by children at the museum.

5 DESIGN TRADE STUDIES

5.1 Instruction Set

One trade-off we have constantly considered is the subset of the instructions that we are implementing. On a larger scale our choice is between implementing all or nearly all of the original AGC instruction set or only a small fraction of the original instruction set.

The advantage of implementing the entire instruction set is that it would make writing our demo code easier as we could completely reuse much of the original Apollo code and just tweak it to our preferences. However some of the instructions that the original AGC used difficult to implement in RTL in general and not very useful. Although we studied and debated every instruction in the instruction set we will give a few example of ones that we chose to include or not include and why we decided to do that below.

One instruction we decided to exclude is the CCS instruction that "The Count, Compare, and Skip instruction stores a variable from erasable memory into the accumulator (which is decremented), and then performs one of several jumps based on the original value of the variable." [3]. This instruction is difficult to implement because it is doing a lot at once so it would require a lot of additional hardware. It is also non-intuitive to use so we would likely not use it much in programming. Thus we determined that this instruction was not worth implementing.

Two instructions we decided to include after some de-

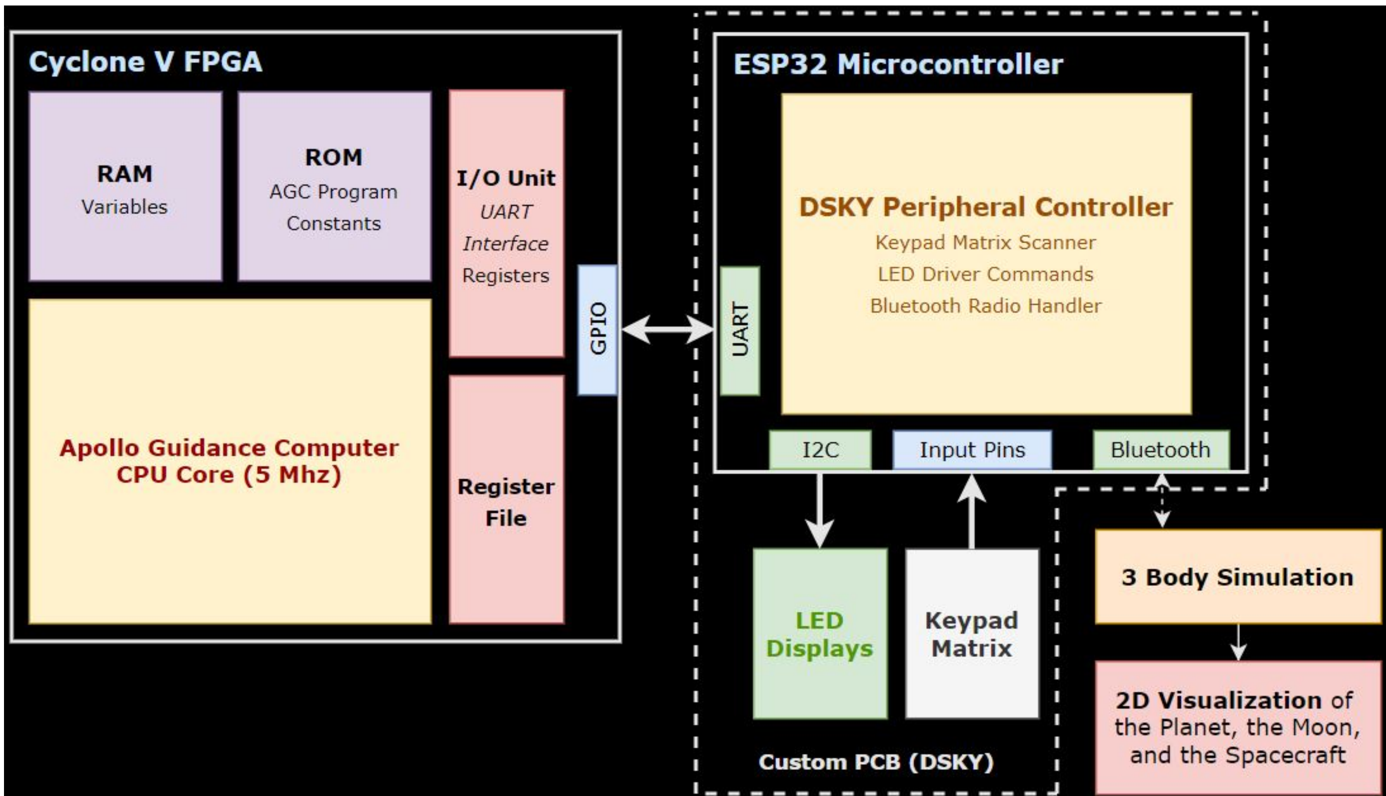

Figure 3: High level system block diagram

bate are multiply and divide. We initially were inclined not to include these instructions in our instruction set because they would be difficult to implement (note that we are using one's complement arithmetic so SystemVerilog's default multiply and divide would not suffice). However given the how much less efficient a software version of these instructions are and how these operations are foundations for any orbital mechanics calculations we decided to implement these two operations. We implemented multiply. But after some failed attempts to implement divide we decided to fall back on our back up plan and instead used inverse multiplies in the AGC code.

One problem this gave use is that the multiply instruction produced a two word product. This is a problem because we had initially opted not to add the double word store instruction so we decided to give it another thought. The RAM IP for the FPGA is dual ported. This means we could not store two words while loading one word for another instruction in the pipeline. This means that a double word load would introduce resource contention that would lead to more complexity, bugs, stalling, and a lower IPC. Thus we decided not to implement this instruction and rather do two single word stores after producing a double word product.

Another set of instructions we decided to not implement are those related to interrupts, which are out of scope given our timeline of development.

5.2 Interrupts

Though interrupts would make writing software easier it would make the RTL design effort much more difficult.

The advantage of interrupts for our use case would first and foremost be allowing for simple and quickly responsive I/O. This would make the process of writing software more simple the I/O channels would not have to be regularly polled. It would also mean that we would not have to wait for a respective channel to be polled to respond but when the value was changed we could check instantly.

The disadvantage if interrupts for our case is that is would increase the RTL design effort. In order to implement interrupts we would have to devise an interrupt system that would respond to different IO channels and do different things based on which interrupt occurred. Though in theory this does not sound too difficult nobody in our group have designed such a system before and this we can only assume that implementing it will take much more time and be much more difficult than we would think. In addition to that polling often does not increase response time and often decreases it. Lastly we determined that given the amount of I/O channels we are implementing polling them would not be that difficult and therefore we decided that it would easier to poll than to implement interrupts.

5.3 Code Assembly

How to go about assembling our code was another design decision. We needed to change the assembler in order

Format	Operation	Format	Operation
AD K	A = A + [K]	*BZF K*	If A = 0; PC = [K] Else PC = PC + 1
ADS K	[K] = A + [K]; A = A + [K]	*BZMF K*	If A <= 0; PC = [K] Else PC = PC + 1
AUG K	If [K] >= +0: [K] = [K] + 1; Else: [K] = [K] - 1	*RETURN*	PC = Q
COM	A = ~A	*TC K*	PC = K Q = PC
CS K	A = -[K]	*TCAA*	PC = A
SQUARE	A = A * A	*TCF K*	PC = K
ZL	L = 0	*READ KC*	A = {KC}; Read {KC} I/O Channel
ZQ	Q = 0	*WRITE KC*	{KC} = A
DIM K	If [K] >= +0: [K] = [K] + 1; Else: [K] = [K] - 1	*RAND KC*	A = A & {KC}
DOUBLE	A = A + A	*ROR KC*	A = A | {KC}
INCR K	[K] = [K] + 1	*RXOR KC*	A = A ^ {KC}
MASK K	A = A & [K]	*WOR KC*	A = A | {KC}; {KC} = A & {KC}
SU K	A = A - [K]	*WAND KC*	A = A | {KC}; {KC} = A & {KC}
LXCH K	[K] = L; L = [K]	*NOOP*	No Operation
QXCH K	[K] = Q; Q = [K]	*INDEX K*	Next instruction in memory is executed with offset *K*
TS K	[K] = A	*EXTEND*	Next instruction is Extended (ex. MP - Multiply, SU - Subtract)
CA K	A = [K]		
XCH K	A = [K]; K = [A]		
XLQ	L = Q; Q = L		

Figure 4: Our Instruction Set

to update the memory map and change the file type. More details about our changes will be included in the system implementation. The three options we looked into were writing our own assembler, taking the output of the yaYUL [4] assembler and fitting it to out needs using a custom python script, and editing the source code of the yaYUL assembler such that it meets our needs. We were solely looking at which option we thought would be easiest to implement.

Writing our own assembler would be time consuming. We would have to right nearly all our code from scratch. However the benefit of this is we could write the script in the language of our choice and we would not have to read someone else poorly documented code.

Taking the output of the yaYUL assembler and fitting to our needs could also be time consuming. One difficulty would be reading a binary file and mapping the memory map they used to our memory map based on that information. We would also need a good understanding of how the yaYUL stored data in the binary to make sure we would be doing the right thing with the data. The benefit to this is that we would not have to write a whole assembler and that we could use a language of our choice for our script.

Out last option is editing the yaYUL assembler such that it produces the output in the MIF format required by the FPGA. The benefits to this is that it would likely require writing the least amount of code. On the other hand we would have to understand the poorly documented assembler most to take this approach and write some string code in C to produce our final output.

Our final choice was to edit the yaYUL assembler such that it produces the output in the correct file format. We implemented this and it worked.

5.4 PCB and Components Selection Trade-offs

To safely and effectively design our PCB, it was imperative that the design and components choices must be reliable and verified, while meeting our design requirements. This meant that our highest priority when deciding components were the ones of which we had prior experience with, or have extensive online documentation. Therefore, some trade-offs had to be made in the aesthetics of the design: the key switches are common computer keyboard switches and the LED displays are matrix-ed alphanumeric displays. The final 3D render (refer to Figure 5) does indeed show some discrepancies from the original Apollo DSKY. However, we considered these aesthetic deviations as acceptable, since we had to maximize the probability that our PCB will work within the remaining 2 months. This was a better design decision than choosing highly custom and non-documented components which will be much riskier and have a higher chance being nonfunctional. More details on the components BOM will be discussed in Section 8.3.

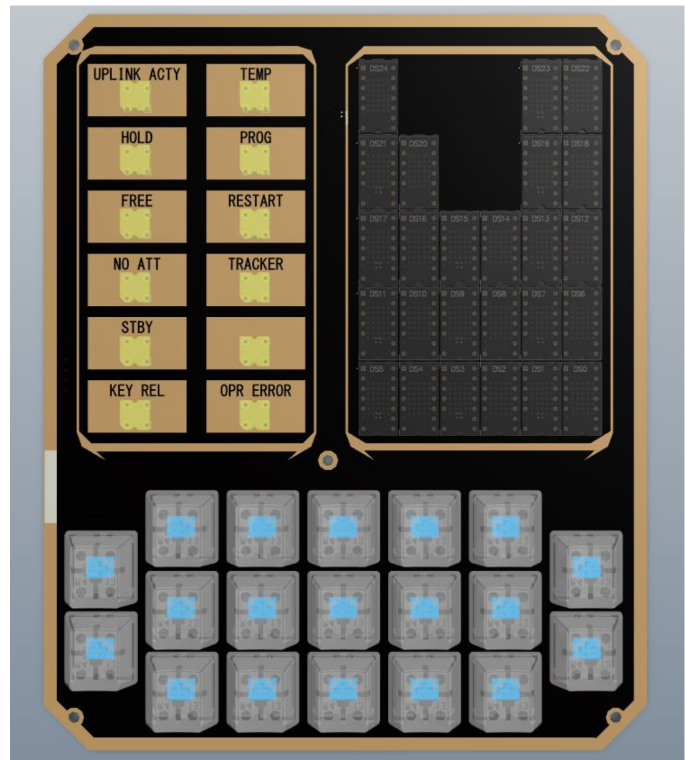

Figure 5: Design rendering of our DSKY (Display and Keyboard) user interface.

6 SYSTEM IMPLEMENTATION

6.1 Program Assembly and Loading

As discussed previously we made edits to the existing yaYUL assembler to fit our needs. For memory we used the IP blocks that Intel's Quartus produces for ROM and RAM. Those ROM and RAM blocks are meant to be used so that the FPGA knows exactly what to use to give the user the demanded functionality. In addition to this the ROM and RAM block IP block was used in simulation. In order, to put data in the RAM and ROM, the IP blocks require us to write to a .mif file and then reference that in the verilog file in the IP block. Thus we will needed to change the assembler such that instead of writing a binary file representing memory we will need to write to two different .mif files, one for RAM and one for ROM.

What will these changes entail? At a high level we need to changed 3 things. One, we wrote to two different files depending on the address. Two, we changed the memory map such that the memory matches our proposed memory map. Lastly, we changed the format. The .mif files at a high level have a line that corresponds to one address space in memory with one value noting the address in octal separated by a colon by the value at that address in octal. Thus we converted the binary values to the ASCII string octal representation. Figure 6 demonstrates this process.

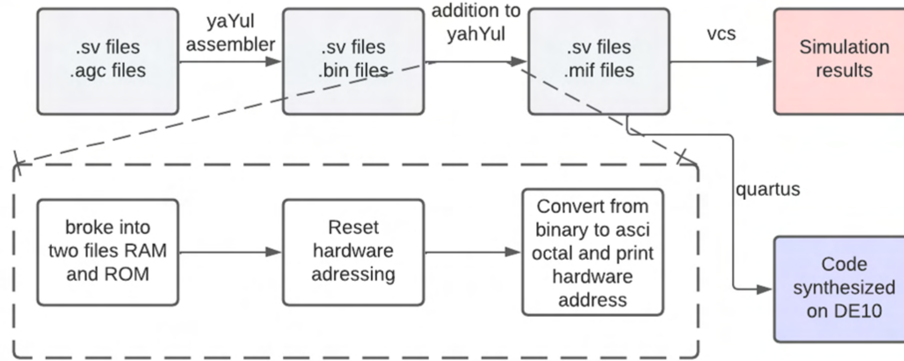

Figure 6: Assembler block diagram

6.2 RTL design

The RTL design is a core component of our AGC implementation. Our pipeline is a four stage pipeline (Fetch Decode Execute Writeback). It is a 4 stage pipeline because we are able to read from memory in decode stage and write to memory in writeback stage (therefore we do not need the memory stage of your typical 5 stage pipeline). Our pipeline implementation is shown in more detail in Figure 7. Our memory map has a 12 bit address space that is split into RAM and ROM. Additionally we use banked memory similar to the original design of the AGC. A visual representation of this is in Figure 8. Our RAM and ROM are implemented using Intel's custom IP for their FPGAs and address translation will be done in our RTL. One thing to note is we did not end up implementing the DV instruction due to difficulties in getting the exact divide behaviour expected by the AGC. Our work around was to use inverse multiplies in the AGC code instead of divides.

6.3 PCB Design

The DSKY PCB, developed using Altium Designer 16, was designed tp provide a professional looking device for display and exhibition purposes. By doing so, we can utilize the full manufacturing capabilities of our PCB manufacturer and use silkscreen/solder-mask to print high quality labels and designs. This can be seen in the 3-D render in Figure 5. The end product also will have a laser cut acrylic front panel, thereby displaying the classic qualities of the original DSKY.

The portions of the schematics are shown in the appendix Figure 14 and a screenshot of the layout is shown in Figure 15. Every two LTP-305G LED displays are handled by an IS31FL3730 I2C display driver, which takes care of all the process for lighting up the multiplexed LED segments with brightness control. The R50RED-F-0160 LED lamps on the left are also controlled by the same driver. They are all connected to a common I2C bus, which is handled by the ESP32 micro-controller acting as the I2C master. The Cherry MX keyboard switches are multiplexed and connected directly to the ESP32, which will conduct periodic scanning of the keys to detect key presses. Diodes are attached to the inputs of each keys to prevent "ghosting," which occurs when two key switches on the same row are pressed and cause a third "ghost" switch to be registered. Finally, two I2C bus pins from the ESP32 are broken out to the connectors so that they can be connected to the FPGA and used for communication.

6.4 Peripheral Controller Software

The Peripheral Controller is the ESP32 microcontroller, which primarily handle detecting key presses and displaying the LED display segments through appropriate I2C commands to the display drivers. The ESP32 microcontroller is a WIFI/Bluetooth module, chosen for its ease of use, low cost, being modular, and having a mature toolchain (Xtensa GCC). The Bluetooth capabilities were used to communicate simulate mission data and calculations to and from the demo Python simulation software.

There are three major program flows, as shown in Figure 9. There are two interrupt service routine (ISR) flows which govern the handling of displaying the LED segments and the keyboard scanning. For the first ISR, an UART interrupt will be triggered upon receiving display values from the AGC FPGA. The ISR will parse the UART packets and then send appropriate I2C commands to all the IS31FL3730 LED drivers, which in turn will drive the 25 LED displays and 14 lamps. The second ISR will be triggered around every 100Hz using a timer, and it will periodically scan each 4 rows to determine if any key switches have been pressed. If any key switches are pressed, the main program will construct a VERB/NOUN pair and send an appropriate command back to the FPGA to be parsed. The ESP32 will also be responsible for error handling (i.e. invalid key presses), of which appropriate warning lamps will be indicated and the packets will be thrown out. A separate Bluetooth task handles the receiving and the sending of mission data and corresponding calculations.

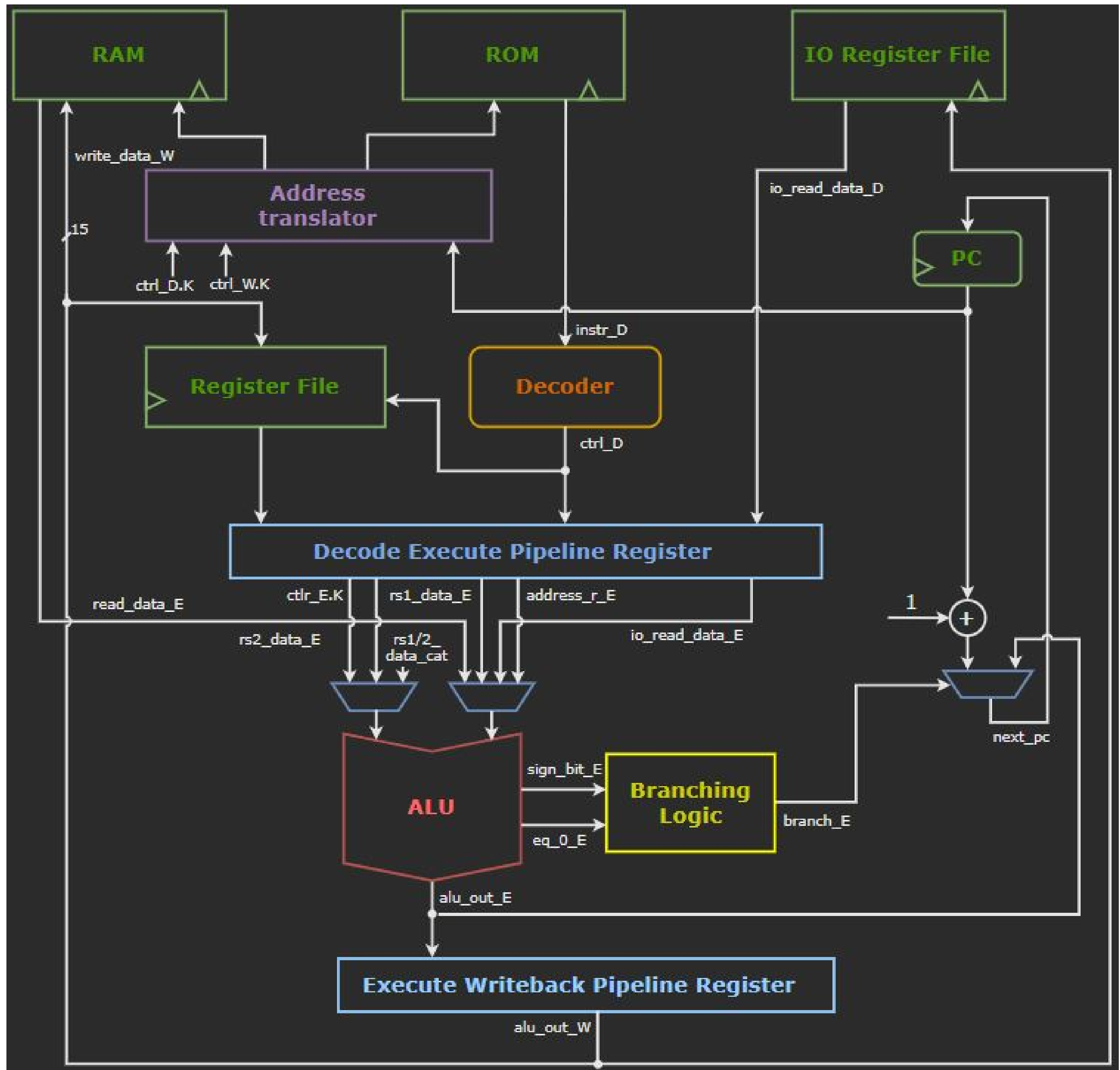

Figure 7: CPU Pipeline Diagram

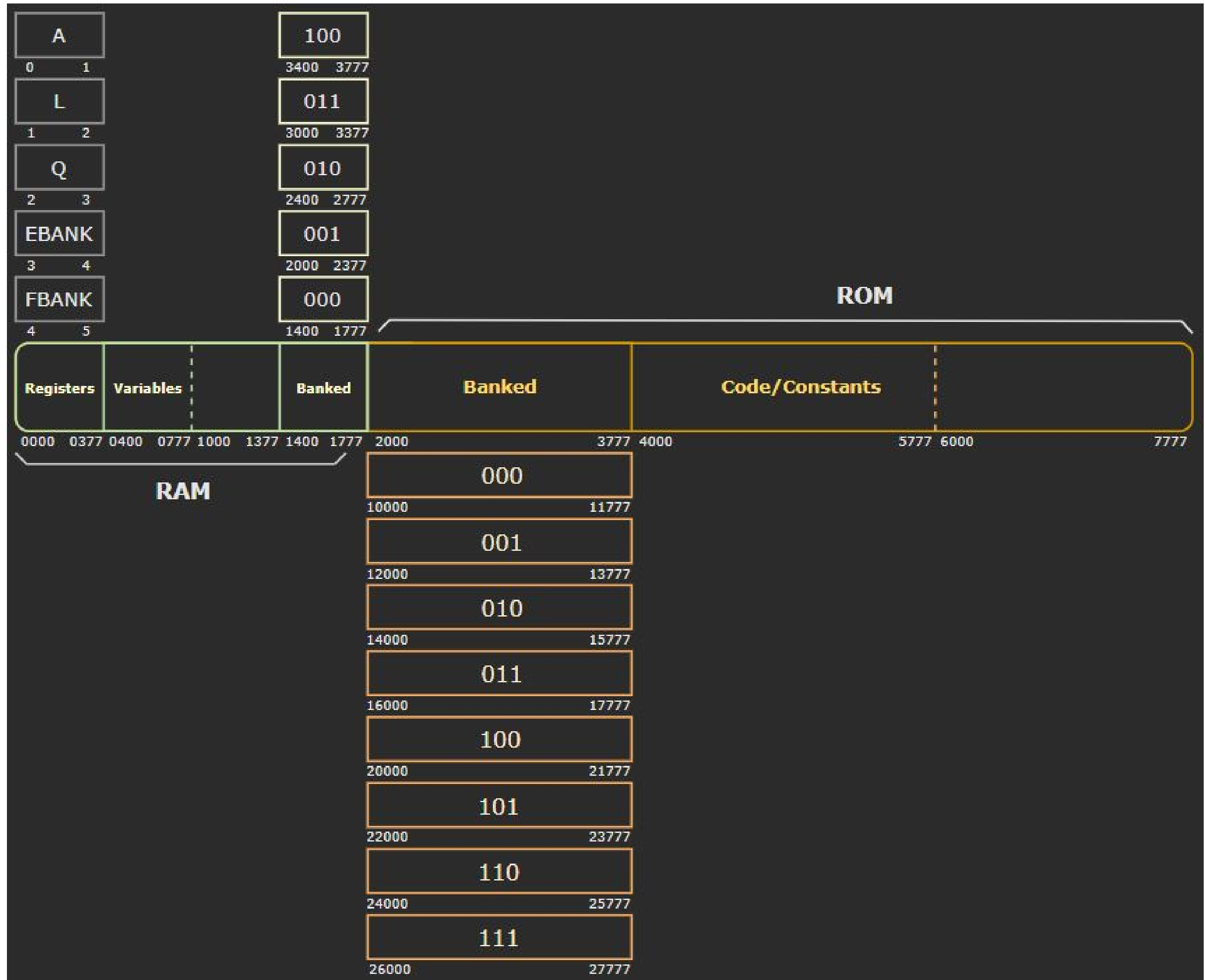

Figure 8: Memory Map of our implementation

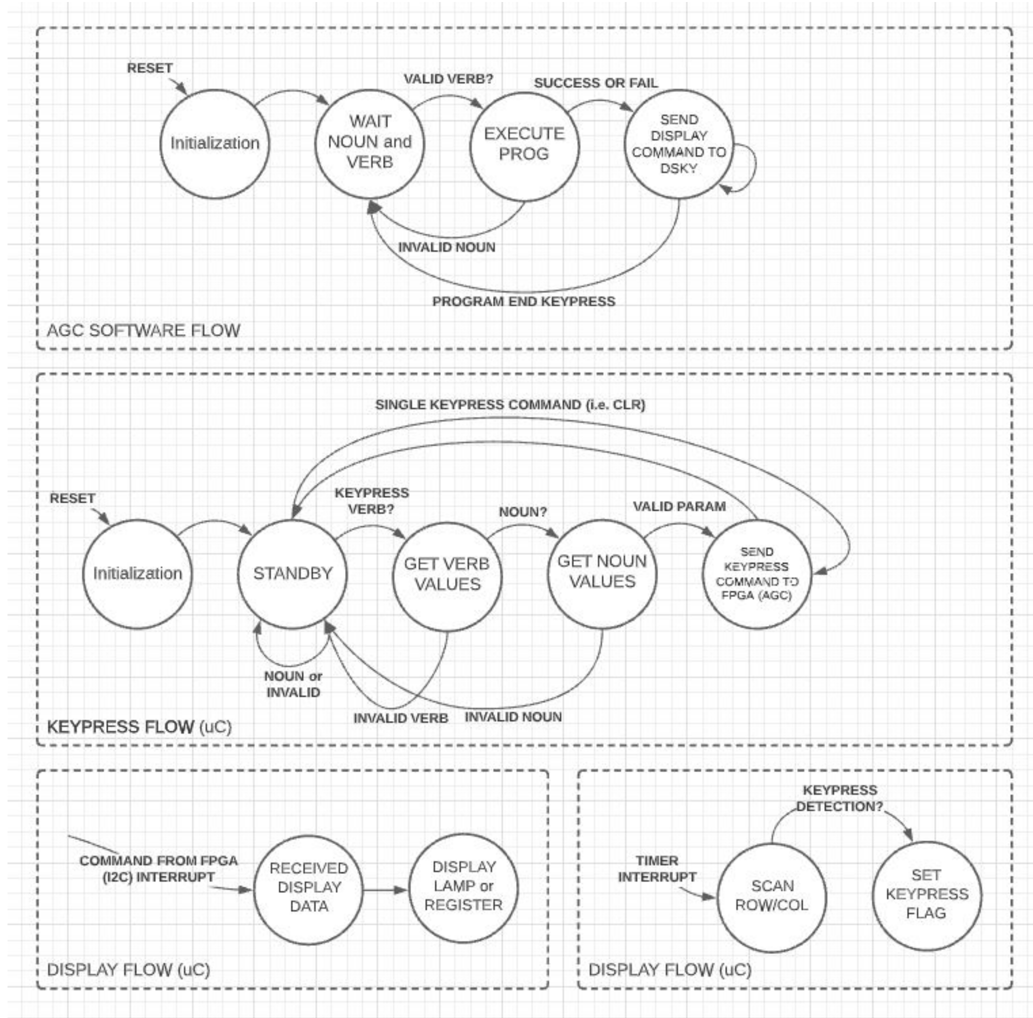

Figure 9: High level FSM of the Peripheral Controller

6.5 Python Demo Software and the AGC Assembly Program

The programs running on the AGC, written in the implemented ISA Assembly, will primarily revolve performing calculations required for basic orbital mechanics. This will require math functions, such as sine, cosine, and square root. Implementing these functions in Assembly will be very challenging, but we can re-purpose the original Apollo's implementations of these math functions. Apollo missions such as *Comanche 055* (Command module programs used for Apollo 11) and *Aurora 12* (Test vehicle) already have implemented these functions in AGC assembly, which are simple enough to be executable on our architecture [2]. The basic orbital mechanics functions and time-keeping programs implemented are listed below.

- **00** Calculate Escape Velocity
- **01** Hohmann Orbit Transfer
- **02** Calculate Initial Burn for Lunar Injection
- **03** Calculate Final Burn for Lunar Injection
- **04** Calculate Alignment Angle for Lunar Injection
- **01** Orbital Plane Transfer

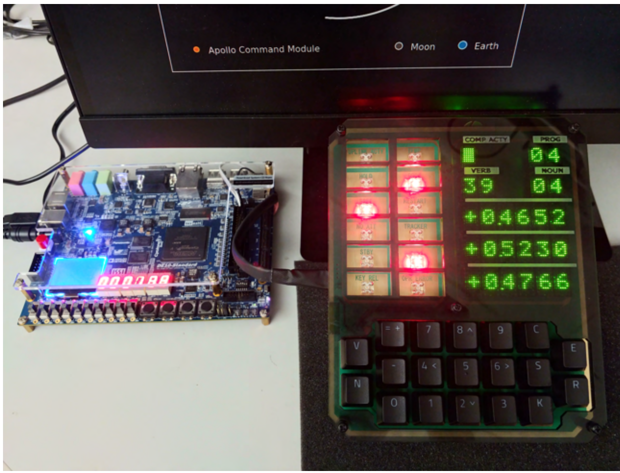

Figure 10: Close up of the FPGA DE10 board connected to the DSKY, currently running the Lunar Injection program

The orbital mechanics calculations primarily revolve around using Orbital Plane/Phase Transfers to calculate delta-v:

$$\Delta v_i = \frac{2\sin(\frac{\Delta i}{2})\sqrt{1-e^2}\cos(\omega+f)na}{(1+e\cos(f))} \tag{1}$$

where e is the orbital eccentricity, ω, is the argument of periapsis, f is the true anomaly, $n = 1/P$ is the mean motion (orbital period frequency), and a is the semi-major axis. Also Hohmann Orbital Transfer equations will be used for orbital mechanics involving increasing/decreasing orbits or performing lunar injection:

$$\begin{aligned} \Delta v_1 &= \sqrt{\frac{\mu}{r_1}}\left(\sqrt{\frac{2r_2}{r_1+r_2}}-1\right) \\ \Delta v_2 &= \sqrt{\frac{\mu}{r_2}}\left(1-\sqrt{\frac{2r_1}{r_1+r_2}}\right) \end{aligned} \tag{2}$$

where Δv_1 is the first burn required and Δv_2 is the second burn required for changing orbitals from orbital radius r_1 to r_2, with $\mu = GM$ being the standard gravitational parameter. More extensive documentation can be found at the referenced website [1].

The demonstration program was written in Python, and displays a physics simulation based on the 3-body problem. This was achieved by solving the differential equations related to the gravitational forces between the three bodies (the Earth, the Moon, and the spacecraft) using a 4th order Runge-Kutta ODE solver. The results were plotted graphically to demonstrate orbital mechanics and the calculation results of the AGC. The full setup can be seen in Figure 1. The clean and aesthetic graphics of the orbiting bodies are crucial for exhibitions, as they serve primarily to visually demonstrate the capabilities of the AGC.

7 TEST & VALIDATION

7.1 Tests for Verifying our RTL meets the ISA

In order to meet the architectural specification we first focused on making sure our RTL meets the specification in simulation. To do this we wrote a test case for each instruction we implemented, which will make it 34 tests. We wrote an infrastructure in SystemVerilog that displays the register values of each register at the end of the test. We then compared that to the simulation results of an software simulator of the AGC in order to make sure we are hitting the architectural specification. We passed the test for every instruction we tested after debugging with the exception of DV which we decided to remove from our design. Thus we passed 34 of the 34 tests that we decided was necessary by the end.

7.2 Verifying our RTL meets our threshold for IPC

In order to verify our AGC processor had an IPC above .5 we had to add some additional hardware to our system. We added counters to count every cycle and count every instruction that finished. Than we ran it on one of our demo programs to retrieve the amount of instructions per cycle. We had an IPC of .8095. This significantly exceeded our target of .5 and exceeded our expectations. It exceeded our expectations because the EXTEND instruction being inserted between many instructions increased the distance between read after write dependencies to be greater than

the hazard distance. Because of that it stalled far less frequently than expected.

7.3 Verifying our RTL meets our threshold for our critical path/frequency

To clarify the frequency that the processor can run at is the inverse of the critical path and thus we only needed to verify one of them to check if we met our constraint for both. When synthesizing a design on Quartus it can tell us of the critical path of the design. It said our critical path was 590 ns which is significantly longer than our 200 ns critical path goal. Because of this we decreased our frequency and in order to make our clock scaling as simple as possible we scaled it to 5 Mhz from our original goal of 50 Mhz. Quartus said our critical path was our writeback from the register in writeback stage to the Intel generated RAM. Thus the issue was that the Intel generated RAM had too long of a critical path. However, once we were able to measure this our system was nearly fully implemented and it seemed to not affect the response time to the user in a significant way. Thus we decided it best to relax our constraints for frequency and critical path because the original constraint was tighter than necessary given our use case requirement of having a smooth user experience.

7.4 Verifying our RTL meets our threshold for the amount of LUTs used

Once again the amount of LUTs used is given by Quartus once compilation was complete.The amount of LUTs is 2271 which is significantly less than out maximum target of 110,000. Thus we meet the constraint. We expected to meet our constraint as our constrain was necessary to make sure our core could fit on the FPGA and did not reflect our expectations. Thus though 2271 was less LUTs than expected but is certainly reasonable.

7.5 PCB Manufacture and Verification

The design review verification consisted of using Altium Designer 16's Electrical Rule Check (ERC) and Design Rule Check (DRC). The only errors encountered were minor silkscreen overlap. The PCB was built and tested at Techspark PCB Labs at Ansys Hall, Carnegie Mellon University. The components were placed using the aid of *ProtoPlace S* pick-and-place machine, and reflowed in the *ProtoFlow S Reflow oven.* Through hole components were manually soldered by hand using a soldering pen. The final piece was verified for soldering via microscope inspection. The correctness of voltages and the I2C packets were validated using a 4-channel oscilloscopes and protocol analyzers (Figure 12). Also, the temperature of the components were verified using an FLIR thermal imaging camera to check for overheating of components. The verification results are as follows: Maximum temperature of around 40.8 degrees Celsius, well within the +85 degrees Celcius maximum; I2C Bus Congestion: 31.8% Congestion (68.2% Idle) for 400kHz bus, well below 100% congestion; UART Bus Congestion: 42.4% Congestion (56.6% Idle) for 115200 Baudrate, again well below 100% congestion.

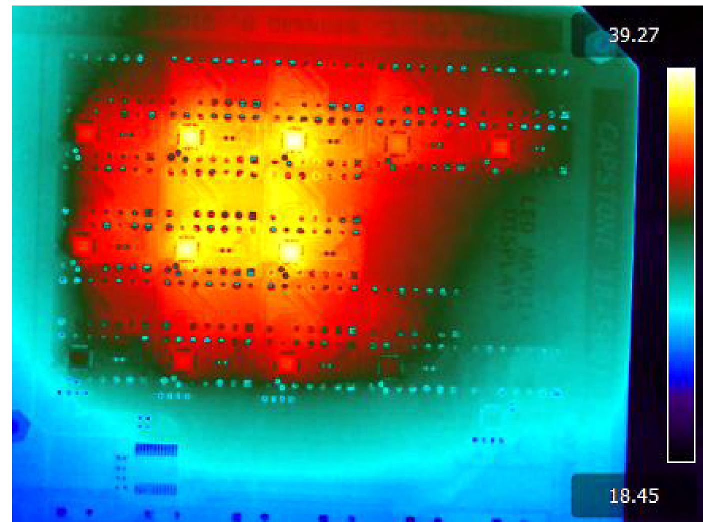

Figure 11: Thermal image of the DSKY PCB LED Drivers

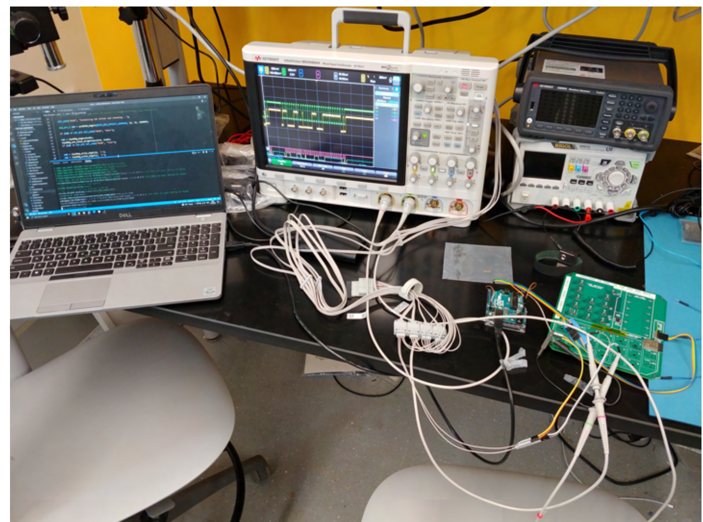

Figure 12: Functional verification of the I2C bus using an Oscilloscope

7.6 Software Calculations

The verification of the software relies primarily on checking the correctness of the software calculations, especially due to rounding errors and the fact that our AGC is using 15-bit single precision fixed floating point. The final results were verified to be within +/-0.5% of the predicted results for Escape velocity, Hohmann Transfer, and Lunar Injection calculations. These results were also functionally verified to produce correct orbital injections and transfers in the physics simulation.

8 PROJECT MANAGEMENT

8.1 Schedule

In general the main changes between our planned schedule and our final schedule is that we have more detail for tasks that were at the end of the project. For example we added a task for the final video and poster. Our gantt chart is included in the references in the back figure 13.

8.2 Team Member Responsibilities

Christopher Bernard: Primarily responsible for the assembler, the RTL testing infrastructure, RTL testing and the RTL coding with the exception of the ALU, address translate unit, register file, and UART receiving unit. Secondarily responsible for making sure the assembly works on the final project by helping debug others code.

Donovan Gionis: Primarily responsible for making the ALU, Address translate unit, register file, and UART recieving unit.

Jae Choi: Primarily responsible for making the PCB of the DSKY, the ESP32 coding, and the display monitor, and for writing the assembly code.

8.3 Bill of Materials and Budget

The bills of materials for the components (excluding the PCB cost) for building 2 and a half pieces are shown in Table 1. The cost of building the PCB and the solder stencil comes to 186.24 USD for 5 boards. Therefore, the total cost is 317.07 + 185.24 = 503.31 USD, and with shipping costs included the total amounts to around 530 USD. Therefore, we are well below the allocated budget of 600 USD and have plenty of spare components for repairs.

8.4 Risk Management

One key risk we had is the PCB not arriving on time. In order to manage that risk Jae started work on the PCB shortly after the semester started. Another risk is that none of us have written an assembler before. To manage that risk Christopher has started working on the assembler earlier than initially planned. One risk we had to mitigate is after our divider in our ALU failed to work. Another risk we have is that generally our project contains a lot of work so if anything takes longer than necessary our whole project may not finish.

To mitigate the largest risk (just not having enough time for everything) we worked ahead of schedule. We were able to do this by putting massive amount of time on spring break into the project to get a couple weeks ahead. Then worked to stay ahead. We ended up finishing the technical part of our project before the final presentation. One large risk we had to mitigate was the RTL design we synthesized on the FPGA does not work (and would therefore be very hard to debug). To manage this risk we spent considerable amount of time setting up a testing infrastructure using VCS in simulation and debugged it in simulation. And when we put it on the FPGA it only had a couple minor bugs.To mitigate the risk of the PCB not working Jae ordered enough parts to make two separate boards just in case he failed with the first one. To mitigate the risk for the divider not working our work around was to not implement it and use inverse multiply instead. We also mitigated the risk of having the I2C bus not work between the AGC and the ESP32 by switching to UART because it is a simpler protocol. Another thing we had to manage was that we could not hit our frequency/critical path constraint so we had to loosen our constraints. Beyond that because we started early we were able to get through our project according to plan.

9 ETHICAL ISSUES

Ultimately, we want our device to convey the AGC's historic role and significance to its users. Thus, it is crucial that the system functions as designed for every user. Our lengthy verification process ensures the reliability of our device once deployed. However, in its role as a museum exhibition piece, the repeated use of the device opens it up to risk of physical damage and malicious tampering. The plastic enclosure around our DSKY should theoretically prevent physical damage and attempts at changing the code on the ESP32, but it is up to the exhibiting museum to ensure that the FPGA development board and simulation display program PC are behind a barrier to prevent changes to the logic.

An educational piece isn't worth much unless it may be readily used by the masses. We feel that our design keeps the overall unit cost at a minimum, and the use of an external computer and development board means that if a potential customer already owns such items, they could simply purchase the software intellectual property necessary to configure those devices for use in the overall system. We believe these features would permit even financially disadvantaged students to have access to our device and the knowledge it shares.

10 RELATED WORK

Our two main competitors are the official Apollo mission display and a yaAGC simulator. The Apollo mission display has the full original Apollo computer and the command capsule. The benefit ours has over this is that ours is smaller and more portable and can be mass produced. The yaAGC software simulator is only a simulator can be put on any computer but does not have any physical hardware to be displayed at the museum.

11 SUMMARY

Our design takes a modern approach on an inspirational mission reflecting both the past but also how much more is

Table 1: Bill of materials (Components)

Description	Quantity	Manufacturer Part Number	Unit Price	Total Price
LED RED CLEAR T/H	26	R50RED-F-0160	1.21	31.46
SWITCH PUSH SPST-NO 0.01A 12V	40	MX1A-E1NW	0.9076	36.30
IC INTERFACE SPECIALIZED 24SSOP	3	PCA9548ADB	1.61	4.83
LED MATRIX 5X7 0.3" GREEN	50	LTP-305G	2.246	112.30
IC MATRIX LED DRIVER AUDIO 24QFN	27	IS31FL3730-QFLS2-TR	1.2168	32.85
CONN HEADER SMD R/A 6POS 2.54MM	3	M20-8890645	0.83	2.49
RES 10K OHM 1% 1/10W 0603	10	RC0603FR-0710KL	0.024	0.24
CAP CER 0.1UF 50V X7R 0603	50	CC0603KRX7R9BB104	0.0324	1.62
SOLDER PASTE NO-CLEAN 63/37 5CC	1	SMD291AX	14.99	14.99
RX TXRX MOD WIFI TRACE ANT SMD	3	ESP32-WROOM-32E-N16	3.6	10.80
RES 4.7K OHM 1% 1/10W 0603	100	RC0603FR-074K7L	0.0097	0.97
DIODE GEN PURP 75V 150MA SOD323	40	1N4148WS-HE3-18	0.212	8.48
KEY CAPS FOR CHERRY MX SWITCHES	1	PBT KEYCAPS	29.74	29.74
3.3V POWER SUPPLY	1	DE10-STANDARD	30	30
DE10 FPGA SOC ALTERA DEV BOARD	1	DE10-STANDARD	0	0
				317.07 USD

possible in the future. Our modern design also makes our project smaller and more portable. Our modern take on the AGC would prove to be an interesting museum exhibit and it's small size would make it easy to move. It could even tour schools.

11.1 Future Work

Given that only a small percentage of logical elements on the Cyclone V FPGA were used, logic handled by the ESP32 could be moved over to hardware for the sake of improving performance in a further iteration. Hardware offers more opportunities at parallelism than software. With less work performed by the program on the ESP32, perhaps a lower-power, more affordable processor could be used in its place.

11.2 Lessons Learned

Though our project was a success there were somethings we now know that could have made it better if we had known to begin with. One would be synthesizing as soon as you can so you can know the critical path of you system before it is too late to fix it. Another thing we learned was to have very clear communications between the people working on different levels of the stack about how each part will function. Lastly it is very important to plan every part of the project before implementing it. We did this for almost every part so it went smoothly. But the parts in which we did not plan previously went far less smoothly because we did not plan them.

Glossary of Terms

- AGC – Apollo Guidence Computer
- CPU - Central Processing Unit
- DSKY - Display and Keyboard
- FPGA - Field Programmable Gate Array
- IMU - Inertial Measurement Unit
- IPC - Instruction Per Cycle
- PC - Personal Computer
- PCB - Printed Circuit Board
- RAM – Random Access Memory
- ROM – Read Only Memory
- RTL - Register Transfer Level
- SOC - System on Chip
- .mif file - Intel's file format for their FPGA Memory.
- .agc file - File format of AGC assembly code
- .sv files - File for SystemVerilog code.

References

[1] Robert A. Braeunig. *Orbital Mechanics*. 2013. URL: `http://www.braeunig.us/space/orbmech.htm`.

[2] Ronald Burkey. *Comanche 055 Single Precision Routines.AGC*. 2009. URL: `https://github.com/chrislgarry/Apollo-11/blob/master/Comanche055/SINGLE_PRECISION_SUBROUTINES.agc`.

[3] Ronald Burkey. *Virtual AGC — AGS — LVDC — Gemini Programmer's Manual Block 2 AGC Assembly Language*. 2021. URL: `https://www.ibiblio.org/apollo/assembly_language_manual.html`.

[4] Ronald Burkey. *virtualagc*. 2022. URL: `https://github.com/virtualagc/virtualagc`.

[5] Jeffrey Kluger. *Elon Musk Told Us Why He Thinks We Can Land on the Moon in 'Less Than 2 Years'*. Aug. 2019. URL: https://time.com/5628572/elon-musk-moon-landing/.

[6] Annie Palmer. *Jeff Bezos looks to life beyond Amazon after historic space ride*. 2021. URL: https://www.cnbc.com/2021/07/20/jeff-bezos-looks-to-life-beyond-amazon-after-historic-space-ride.html.

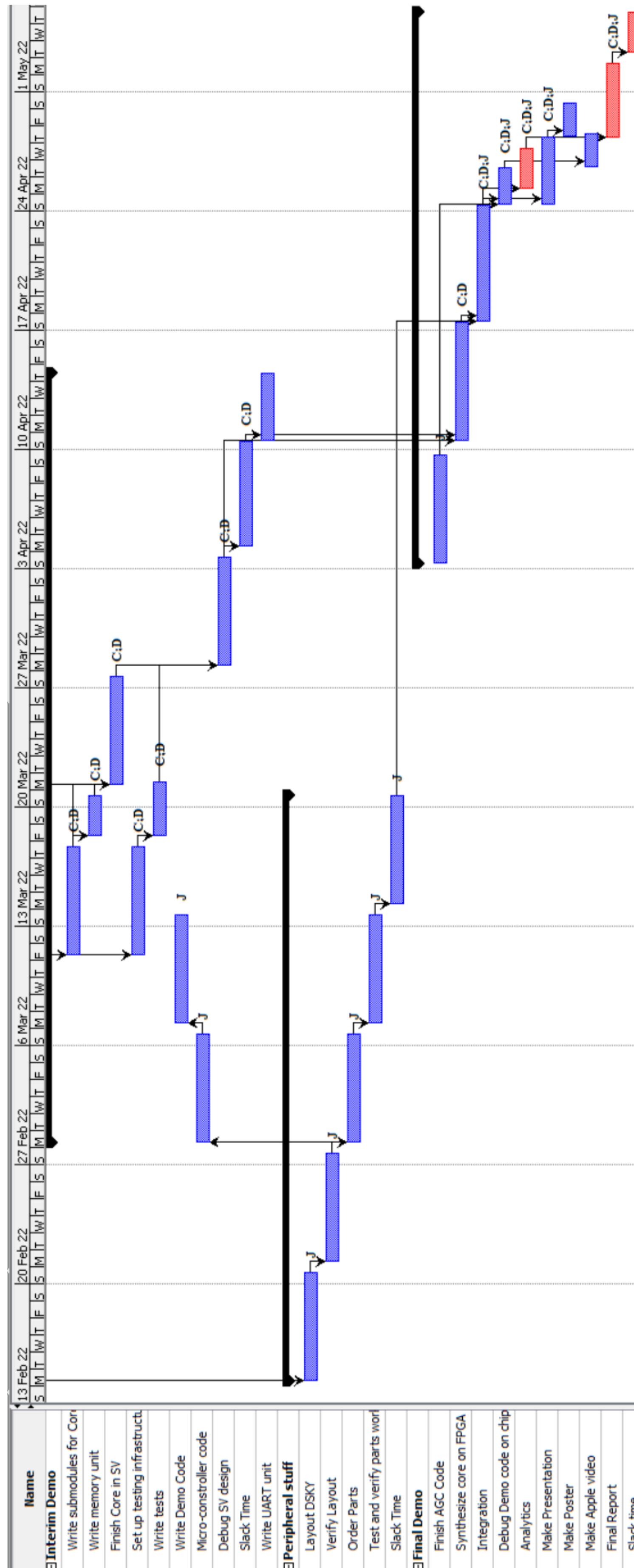

Figure 13: Our full gantt chart.

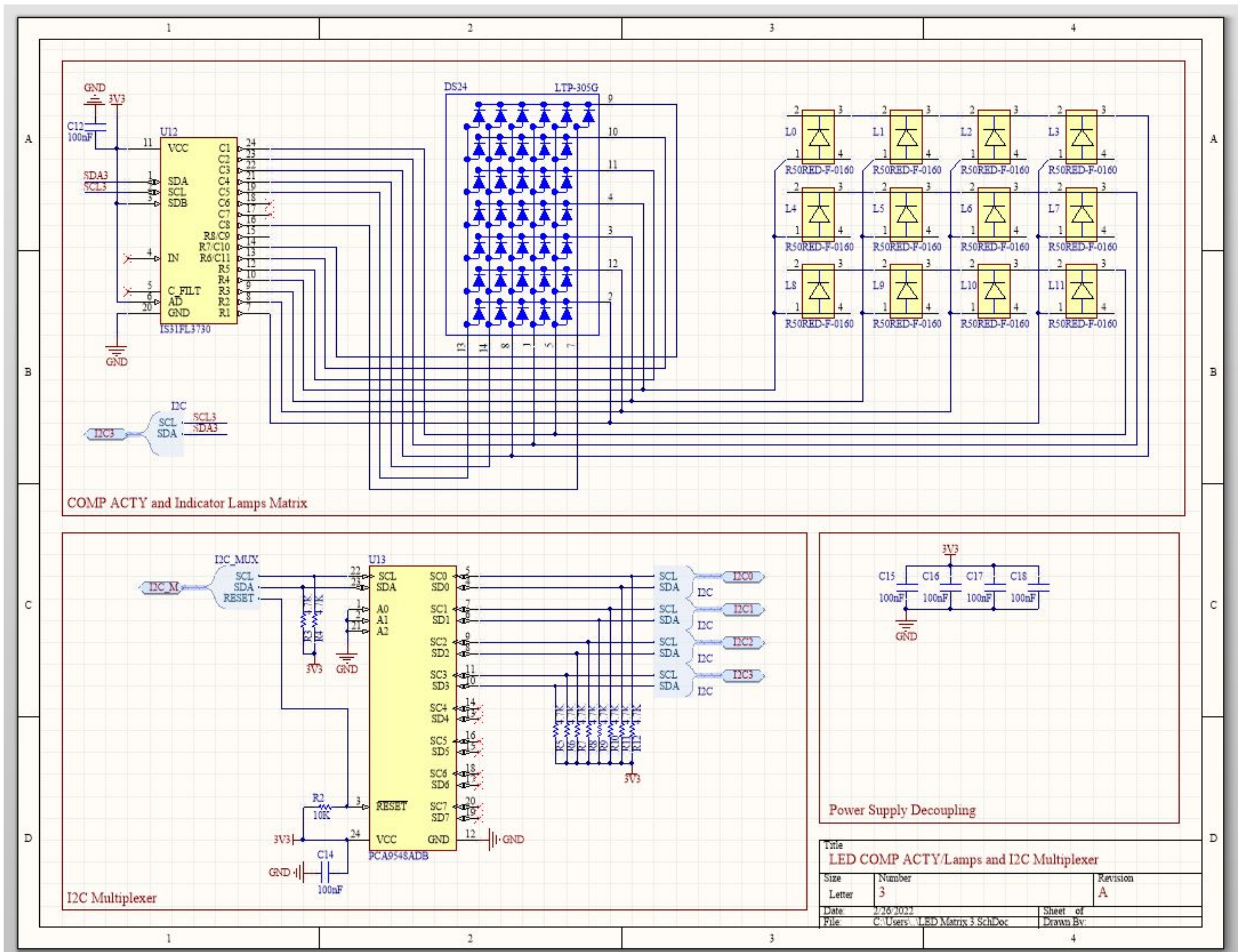

Figure 14: DSKY Schematics for the portion of the LED Lamps and Display

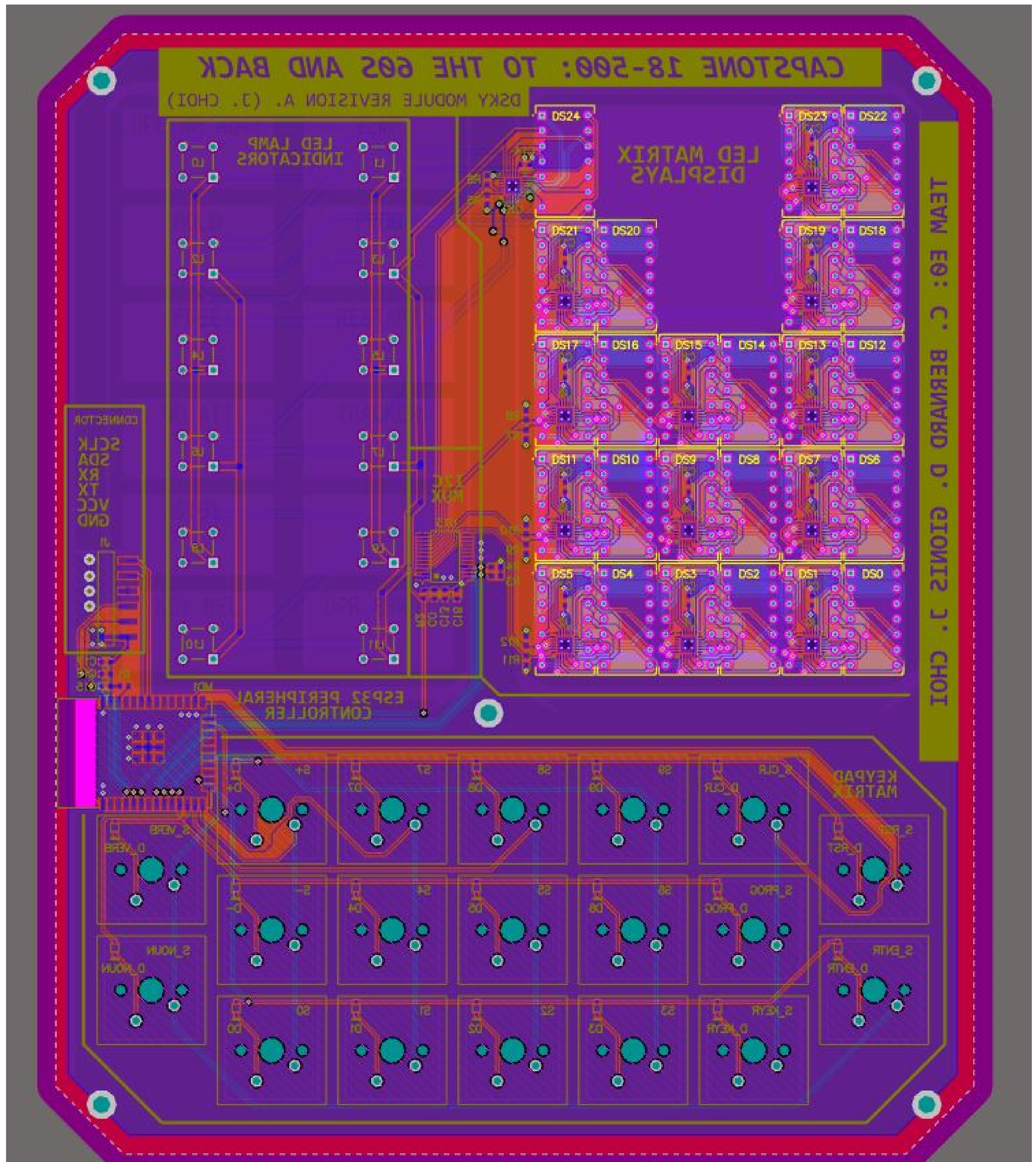

Figure 15: DSKY PCB Layout as seen from the bottom