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 something more like the original computer which is 70 pounds and runs at 1.1Mhz, our implementation uses modern technology, as an open-source-able FPGA core, and will weigh 3.5 pounds and run at 50 Mhz. This will make our implementation run faster and be more portable than the original unit, 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 aim to 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 aim to 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

The use case requirements were mostly shaped around the specifications of commonly available FPGA development boards and concerned mostly on what was possible to successfully implement during the semester (our minimum viable product - MVP). This meant that our AGC Architecture would be running on 50 Mhz clock, which is a common clock speed on a lot of boards. This would allow our designs to be able to run on various development boards and be flexible as a distributable piece. Furthermore, for the AGC architecture, we will have 33 instructions and 5 I/O channels. That is the MVP architecture of what is left after we had de-scoped all the complicated and eccentric qualities of the original Apollo guidance computer. This was crucial in guaranteeing of having any chance of finishing this project in time.

For the hardware side of the project, we will have a DSKY-like display on a PCB, inspired by the original Apollo DSYKY's retro looks, therefore requirements were dictated by original's specifications. This consisted of 14 LED lamp indicators, 25 LED displays, and 19 key switches with a distinct keyboard layout. The update of these displays and the scanning of keyboard switches will be around 100 Hz, which is just fast enough to appear smooth in response, but not too fast that it will congest our I2C traffic with the FPGA.

The weight and size will be kept at a minimum (around 1 by 2 feet at 5 pounds), for ease of transport. Furthermore, we will require some protective measures (i.e. a plastic enclosure) to prevent the destruction of our boards while on display for the demonstration.

3 THEORY OF OPERATION AND ARCHITECTURE

The user will interact with the system via our PCB-based operator interface (see Figure 1), inspired heavily by the original DSKY that was mounted in the instrumentation panel of the Apollo spacecraft. 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 simulated mission metrics, initiate a lamp test, and perform simple arithmetic operations in "Calculator Mode". Under the hood is a small network of processing cores, digital circuitry, serial communication, and discrete I/O that make all of these things happen.

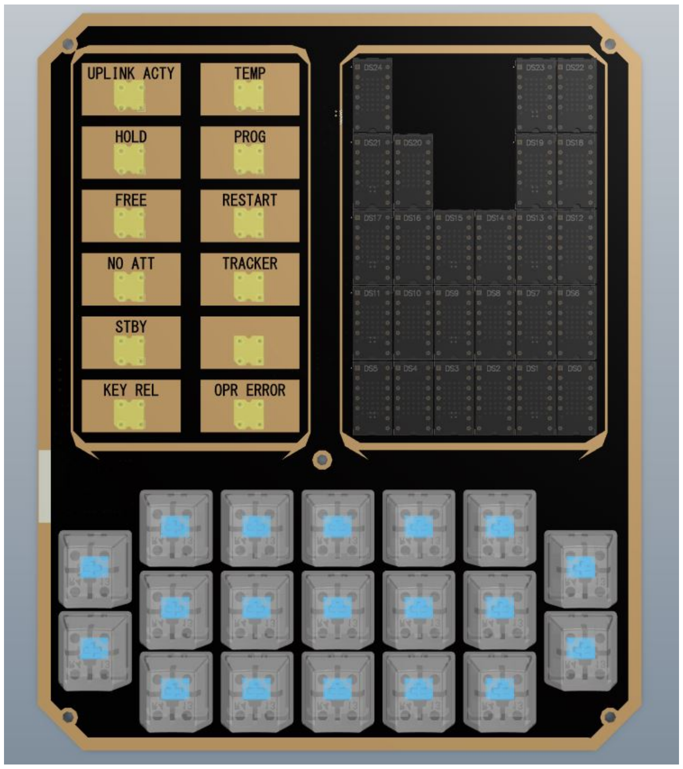
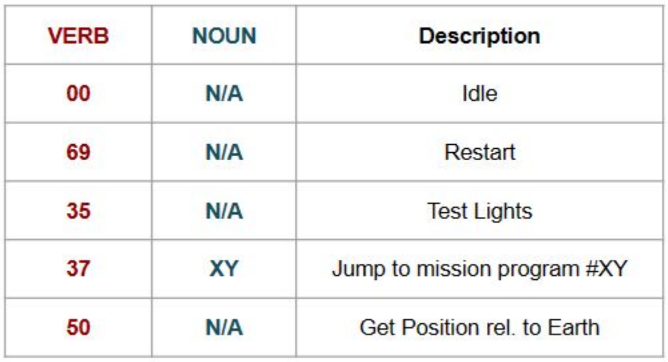


Figure 1: Design rendering of our DSKY (Display and Keyboard) user interface.

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. Most of our commands are verb-only, with the exception of that which runs a mission-oriented program, where the user must also enter a noun to indicate the particular program they wish to run



VERB	NOUN	Description
00	N/A	Idle
69	N/A	Restart
35	N/A	Test Lights
37	XY	Jump to mission program #XY
50	N/A	Get Position rel. to Earth

Figure 2: Table of pertinent verb, noun combinations.

3.2 Theory of Operation

Upon powering up the device, the user automatically puts the system in 'Idle' mode (Verb 00). 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 37 (Jump to mission program XY), has been confirmed, the user must also enter a valid noun XY, corresponding with the mission-oriented program they wish to run. Otherwise, a valid verb alone will initiate the selected function program to run.

While the device is running a selected function program, the user may enter a verb 00 to exit and return to 'Idle' mode, or any other valid verb/noun command to exit the current program and run something else. Some of the mission programs (for a verb 37 command) read simulated sensor data at input ports connected to the Cyclone V's Arm core and write output data to the same Arm core. The Arm core runs a program which will provide simulated data, representing what would have been driven by simple binary sensors or more complicated devices like inertial measurement unit (IMU), to our AGC CPU which will use this information in pertinent mission programs the user chooses to run. This program will also read a subset of the AGC CPU's output channels, update an output status display on an external computer monitor, and update the simulated sensor data as a reaction to these outputs.

3.3 System Architecture

Making all of this happen is a system consisting of two PCBs. The first is an Altera DE10-Standard Development Board with a Cyclone V SOC that contains both a Cyclone V FPGA and an Arm Core, on which we will be running embedded Linux. The second board 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. These boards communicate with each other via an I2C bus. (The following content refers to the block diagram in Figure 3 on the next page.)

Within DE10-Standard's Cyclone V SOC, the Arm Core with Embedded Linux will run the simulated I/O data program. Data will be exchanged between it and our FPGA AGC CPU via an on-chip AXI bridge. This brings us to the FPGA portion of the SOC, on which is implemented our AGC CPU, and an I2C interface module containing logic of our own design and some Intellectual Property from OpenCores. The CPU has a set of I/O channels divided between the AXI bridge interface to the Arm Core and the I2C interface module which transcieves serial data over a 2 wire bus connected to GPIO pins on the DE10-Standard.

I2C is the serial communication medium we have selected for data exchange between the AGC CPU and the DSKY-inspired user interface. The first bus is between the AGC CPU and the ESP32 microcontroller, located on our custom PCB. The CPU's display update commands and the microcontroller's keypad status data are exchanged over this bus. The second bus is used by the microcontroller to drive the individual LED displays, which are I2C slaves. The keypad, on the other hand, employs a matrix format using some of the microcontroller's discrete inputs.

4 DESIGN REQUIREMENTS

We will need an assembler capable of processing AGC source code and assembling it into binary code that can be put on the FGPA chip. This is necessary because the code must be ran on the FPGA in order for the calculations to be made and displayed on our DSKY to our users.

At the core of the design is our version of the AGC CPU, implemented in SystemVerilog, synthesized on a Cyclone-V FPGA. The our system's functionality requires a CPU implementation that will support 33 of the 49 original AGC instructions, and they are listed in table 1. The peripherals used to communicate to the CPU require at least 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 microsecond critical path.

The user interface in our system will include a custom-designed printed circuit board (PCB). Inspired by the original DSKY (Display and Keyboard) interface of the Apollo missions, the PCB will contain 14 LED indicator lamps, 25 LED displays, and 19 mechanical key switches. Having a physical DSKY will allow the user to interact with the system, making our design more impactful and exciting, as compared to a simple software simulation. This is crucial in producing an effective exhibition piece.

Our design will also need 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 instruction. 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 debate 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.

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 is the interrupt related instructions. This is because we decided to not implement interrupts. We will discuss that decision next.

Our final list of instructions we are implementing will be included in the another section.

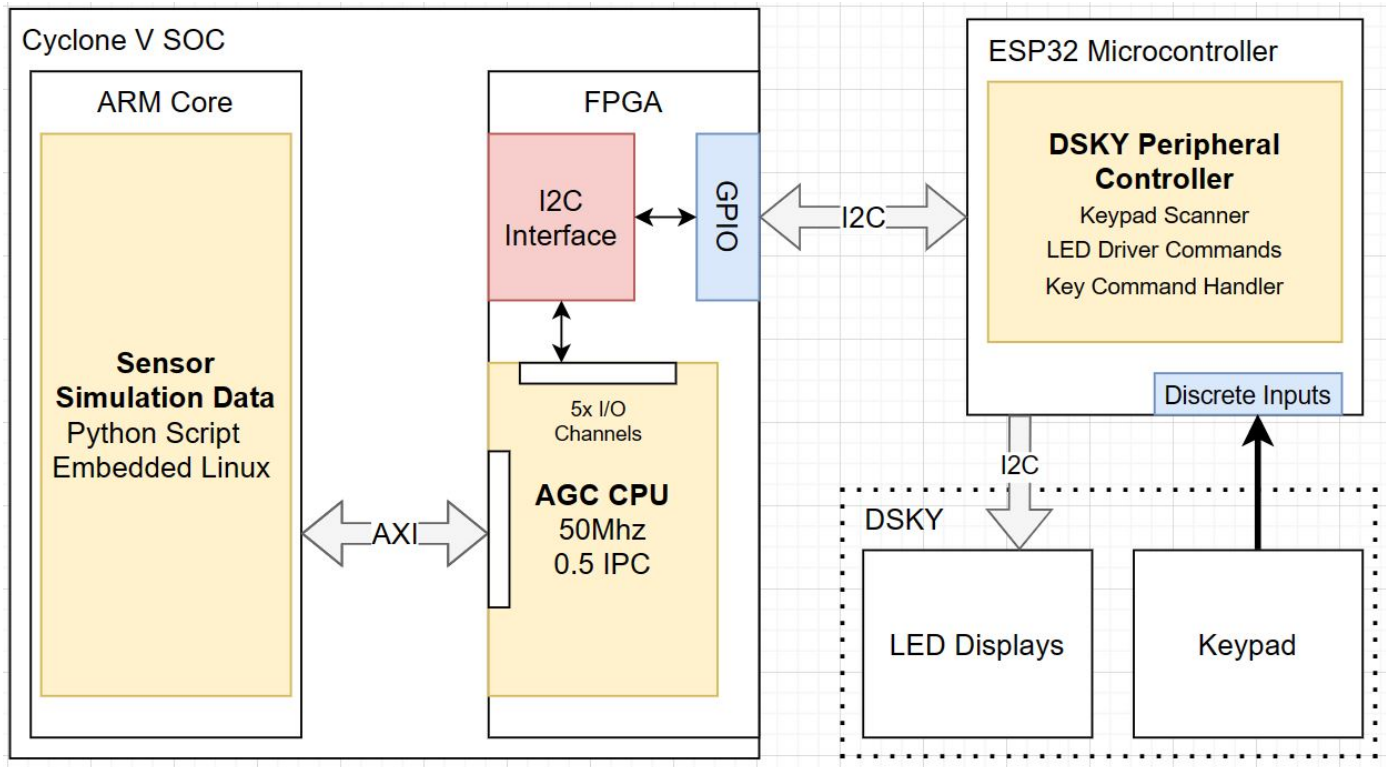


Figure 3: High level system block diagram.

Table 1: Instruction Set

Instruction	Description	Instruction	Description
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	$A = \tilde{}A$	TC	$PC = K$; $Q = PC$
CS K	$A = -[K]$	TCAA	$PC = A$
SQUARE	$A = A * A$	TCF K	$PC = K$
ZL	$L = 0$	READ KC	$A = \{KC\}$;
ZQ	$Q = 0$	WRITE KC	$\{KC\} = A$
DIM K	If $[K] >= +0$; $[K] = [K] + 1$; Else $[K] = [K] - 1$	RAND	$A = A\&\{KC\}$
DOUBLE	$A = A + A$	ROR	$A = A\|\{KC\}$
INCR K	$[K] = [K] + 1$	RXOR	$A = A \wedge \{KC\}$
MASK K	$A = A\&[K]$	WOR	$A = A\|\{KC\}$; $\{KC\} = A\&\{KC\}$
SU K	$A = A - [K]$	WAND	$A = A\|\{KC\}$; $\{KC\} = A\&\{KC\}$
DV K	$(A, L) = (A, L)/[K]$	LXCH K	$[K] = L$; $L = [K]$
NOOP	Nothing	QXCH K	$[K] = Q$; $Q = [K]$
INDEX K	Next instruction is executed differently	TS K	$[K] = A$
EXTEND	Next instruction uses extra code to interpret	CA K	$A = [K]$
MP K	$(A, L) = A * [K]$	XLQ	$L = Q$; $Q = L$
		XCH K	$A = [K]$; $K = [A]$

5.2 Interrupts

As mentioned in the last section we decided not to implement 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. 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 choice for now is to edit the yaYUL assembler such that it produces the output in the correct file format. However if that becomes more difficult than it looks we may change to one of the other methods.

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 1) 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.

6 SYSTEM IMPLEMENTATION

There should be a subsection for each of the subsystems as shown below.

6.1 Program Assembly and Loading

As discussed previously our plan is to make edits to the existing yaYUL assembler to fit our needs. Our plan for memory is to use the IP blocks that Intel's Quartus can produce for use 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 can be used in simulation. In order, to put data in the RAM and ROM IP the block require us to write to a .mif file and then reference that in the verilog file in the IP block. Thus we will need 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 change 3 things. One we will be writing to two different files depending on the address. Two we will be changing the memory map such that the memory matches our proposed memory map. Lastly, we will change 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

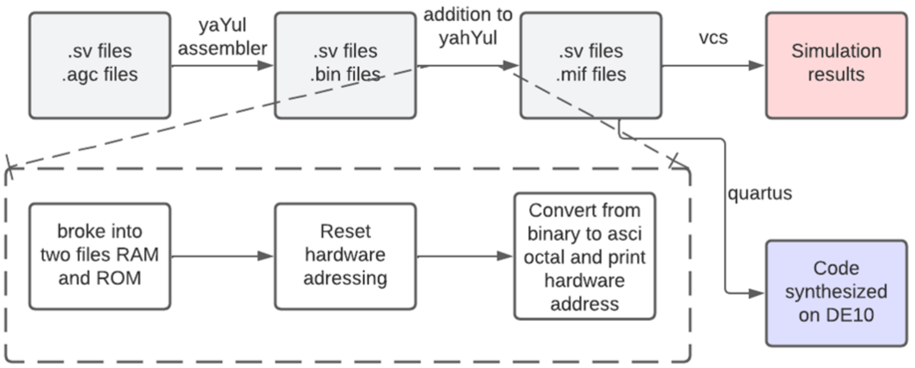


Figure 4: Assembler block diagram

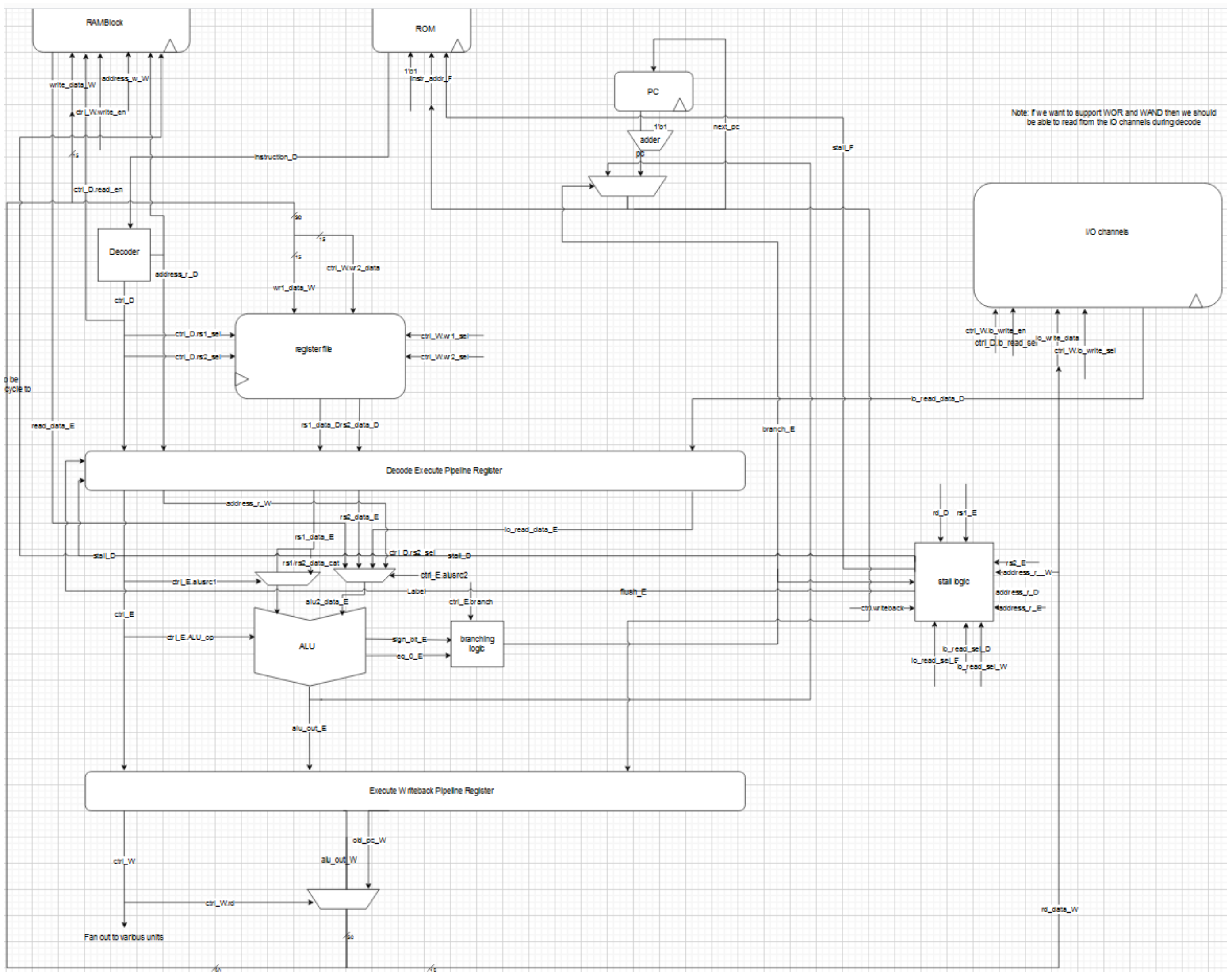


Figure 5: Pipeline Diagram

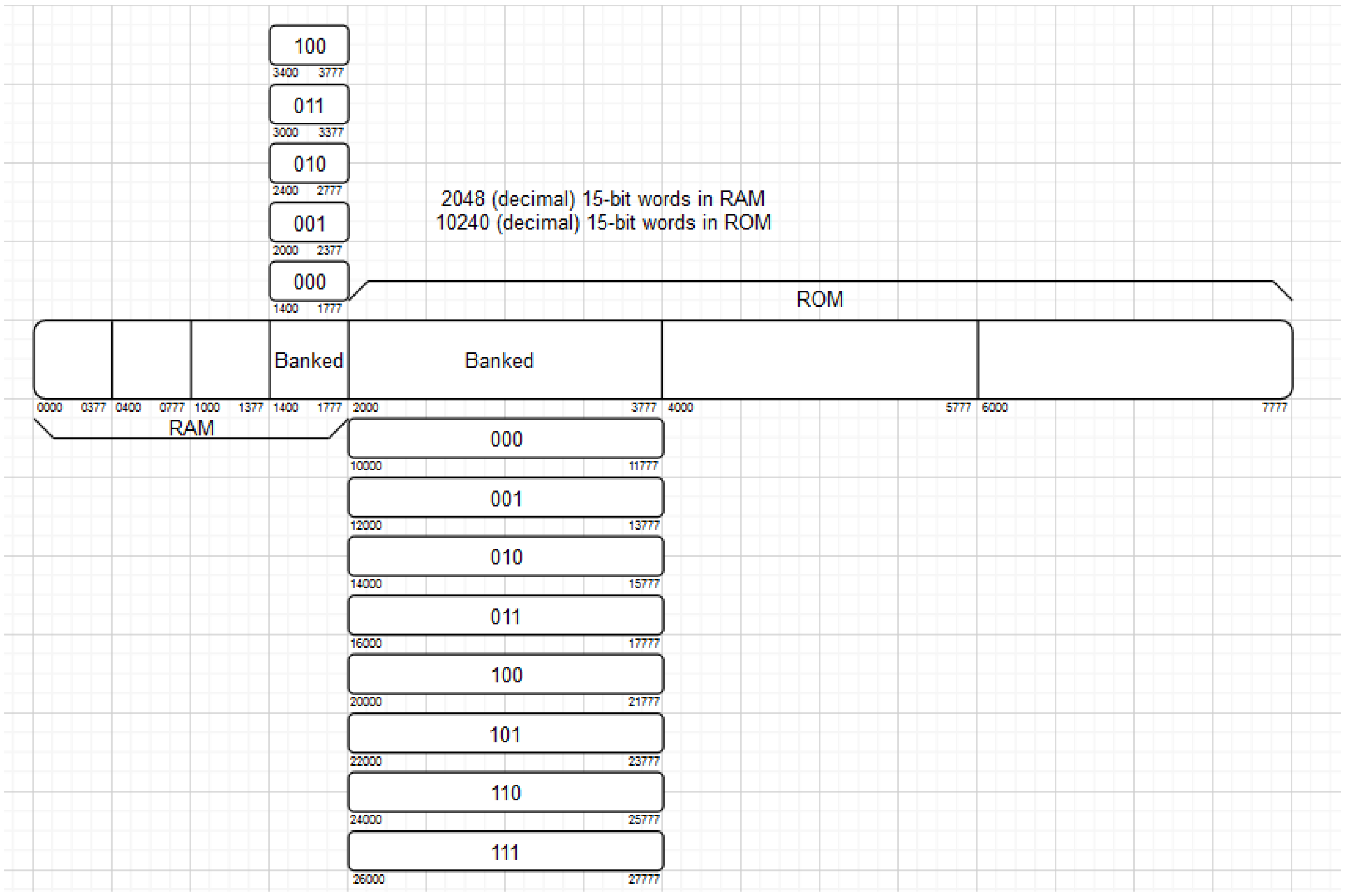


Figure 6: Memory Map

address in octal. This means we will have to convert the binary values to the ASCII string octal representation figure 4 demonstrates this process.

6.2 RTL design

The RTL design is a core component of our AGC implementation. Our pipeline will be a four stage pipeline (Fetch Decode Execute Writeback). It will be a 4 stage pipeline because we will be 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 5. Our memory map has a 12 bit address space that is split into RAM and ROM. Additionally we will be using banked memory similar to the original design of the AGC. A visual representation of this is in Figure 6. Our RAM and ROM will be implemented using Intel's custom IP for their FPGAs and address translation will be done in our RTL.

6.3 PCB Design

The DSKY PCB, developed using Altium Designer 16, was designed such that the PCB itself would act as the front facing panel of the DSKY. 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 1. The end product is an professional looking DSKY panel at minimal cost, while still retaining the classic qualities of the original DSKY.

The portions of the schematics are shown in the appendix Figure 9 and a screenshot of the layout is shown in Figure 10. 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 will primarily handle detecting key presses and displaying the LED display segments through appropriate I2C commands to the display drivers. Note, although the ESP32 microcontroller is a WIFI/Bluetooth module, which is not required for this device, it was still chosen for its ease of use, low cost, being modular, and having a mature toolchain (Xtensa GCC).

There are three major program flows, as shown in Figure 7. 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 I2C interrupt will be triggered upon receiving display values from the FPGA. The ISR will parse the I2C 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.

6.5 Demo Software on Linux SoC and the AGC

The programs running on the AGC 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 to be implemented are listed below.

- **00** Monitor Mission Time
- **02** Orbital Plane Transfer
- **04** Calculate Escape Velocity
- **01** Hohmann Orbit Transfer
- **16** Conduct Lunar Insertion
- **05** Calculator Demonstration

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

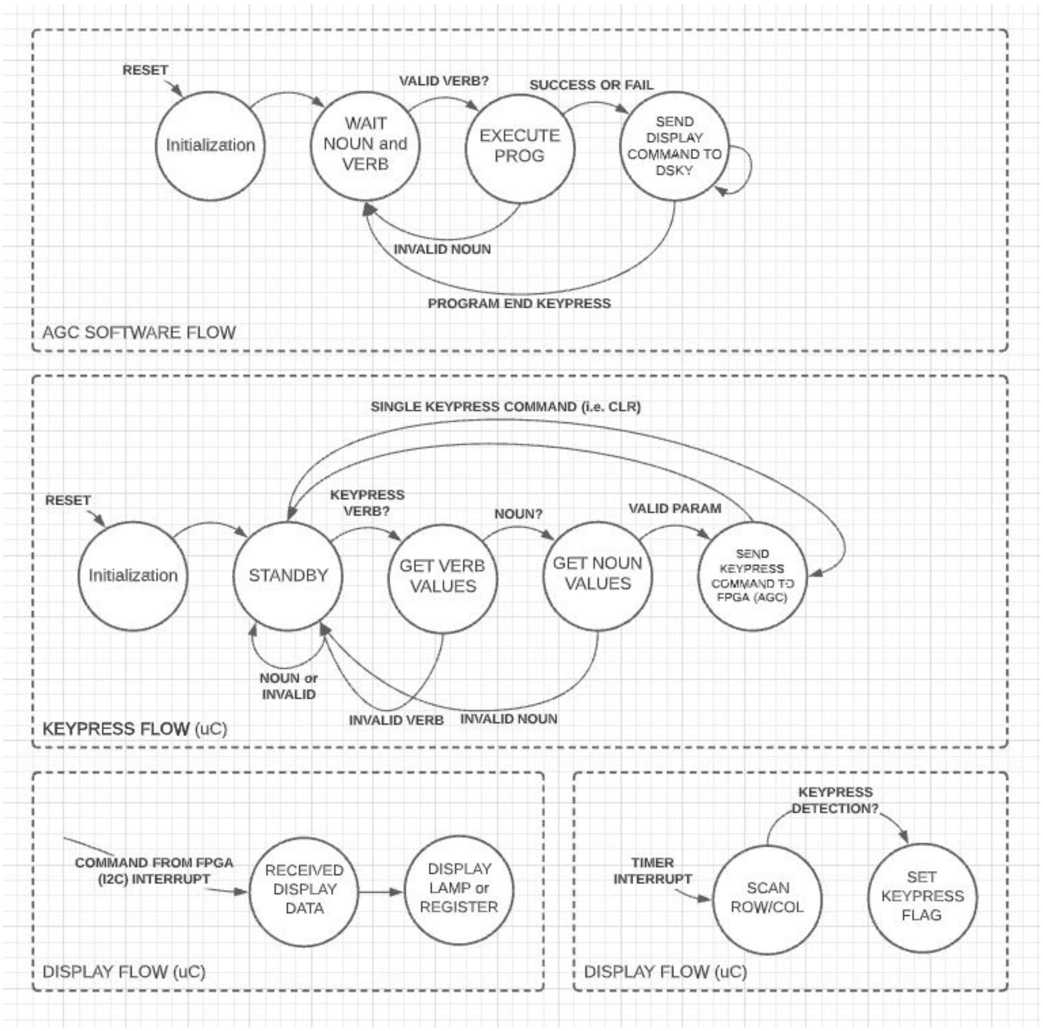


Figure 7: High level FSM of the Peripheral Controller

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].

7 TEST & VALIDATION

7.1 Tests for Verifying our RTL meets the ISA

In order to meet the architectural specification we will first focus on making sure our RTL meets the spec in simulation. To do this we will write a test case for each instruction we are implementing, which will make it 33 tests. We will have a infrastructure in SystemVerilog that will display the register values of each register at the end of the test. We will then compare that to the simulation results of an online simulator of the AGC in order to make sure we are hitting the architectural specification.

7.2 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 will be built and tested at Techspark PCB Labs at Ansys Hall, Carnegie Mellon University. For the surface mount components, solder will be dispensed using the *Protoprint ZeliFlex QR MT Solder Paste Printer*, the components placed using the aid of *ProtoPlace S* pick-and-place machine, and reflowed in the *ProtoFlow S Reflow oven.* Through hole components will be manually soldered by hand using a soldering pen. The final piece will be verified for correct soldering of the components using the *TruView X-ray analyzer* and also checked via microscope inspection. The correctness of voltages and the I2C packets will be validated using a 4-channel oscilloscopes and protocol analyzers. Finally, the temperature of the components will be checked using an IR thermometer/camera to prevent overheating of components.

7.3 Software Calculations

The verification of the software relies primarily on checking the correctness of the software calculations. Therefore, the final results must match within 0.1% of the predicted results to prevent rounding errors that can impact our simulation demo.

8 PROJECT MANAGEMENT

8.1 Schedule

The general break down of our schedule is that we will be working on the PCB while working on this document. We will then work on setting up the test infrastructure and writing and debugging the RTL. Next we will write the software while synthesizing our project on the FPGA. Then finally we will integrate. Our complete schedule is included in the back as figure 8.

8.2 Team Member Responsibilities

Christopher Bernard: Primarily responsible for the assembler and RTL testing infrastructure and co-responsible with Donovan on making sure the RTL works. Secondarily responsible for making sure the assembly works on the final project.

Donovan Gionis: Primarily responsible for making the sub-modules and co-responsible with Christopher on making sure the RTL works. Secondarily responsible for making sure the assembly works on the final project.

Jae Choi: Primarily responsible for making the PCB of the DSKY and the peripheral interface 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 2. The cost of building the PCB and the solder stencil comes to 186.24 USD for 5 boards. Therefore, the total cost is 287.07 + 185.24 = 473.31 USD, and with shipping costs included the total amounts to around 500 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 is working on it now at the cost of having others pick up his slack on the report. 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. 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 this plan we have several things we can cut out and still have a project. We could write less assembly code (have less programs for demo) and we could simplify our pipeline design while relaxing our frequency and IPC constraints. If the PCB has errors, bodge wires will be used to discretely fix the traces. If the worst happens and the PCB goes up in flames, then we will opt for a software DSKY demo. If our critical path is longer than 200ms, we will weigh the cost/benefits of spending more time optimizing or opting for a slower clock. If some of non-essential

Table 2: 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
DE10 FPGA SOC ALTERA DEV BOARD	1	DE10-STANDARD	0	0
				287.07 USD

instructions are costing us too much time/effort, we will descope them if necessary. Finally, if we are running out of time, we will focus on creating a simple but compact demo that demonstrates the absolute key features of the DSKY.

9 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.

10 SUMMARY

Our project uses a DE-10 SOC and a custom DKSY PCB. Our design takes a modern approach on an inspirational mission reflecting both the past but also how much more is 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.

Our upcoming challenges will include getting all of the parts of our project to work together in one. Each component of what we are working on will be time consuming so time management will be key for us to finishing everything on time in order to have a finished display for people to interact with.

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
- 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.

[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`.

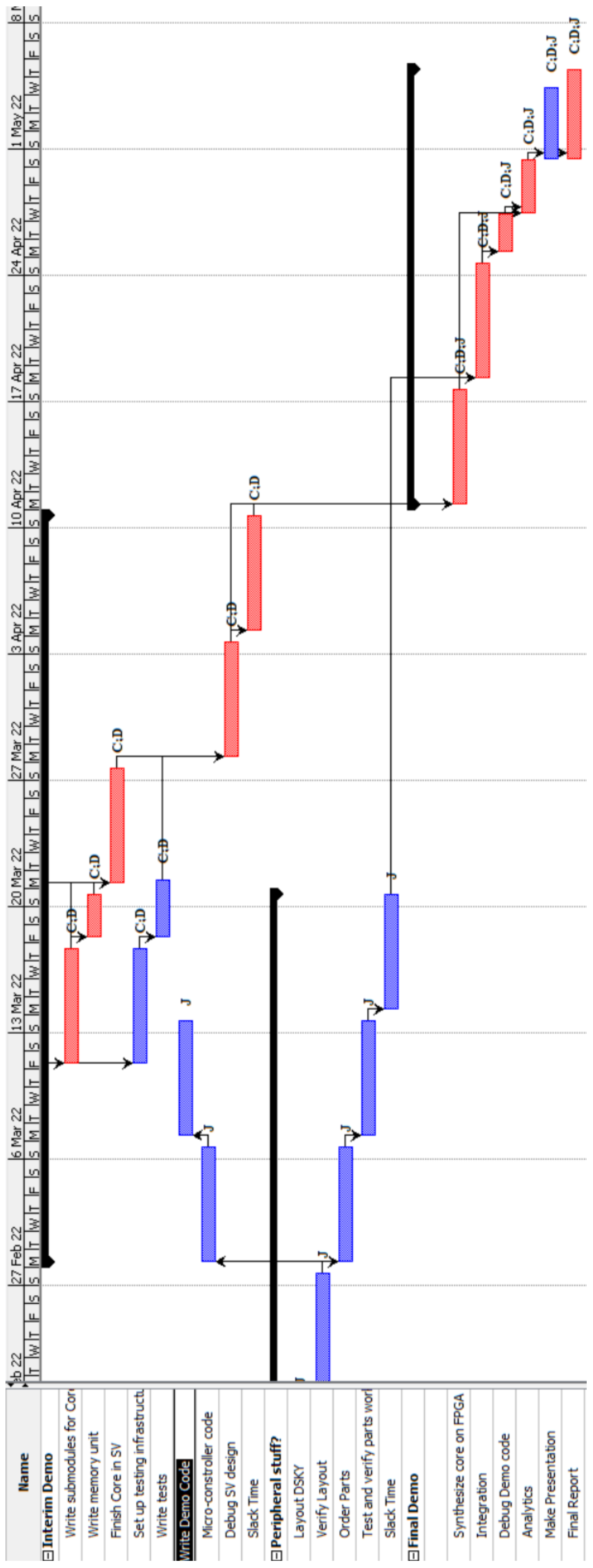


Figure 8: Our full gantt chart.

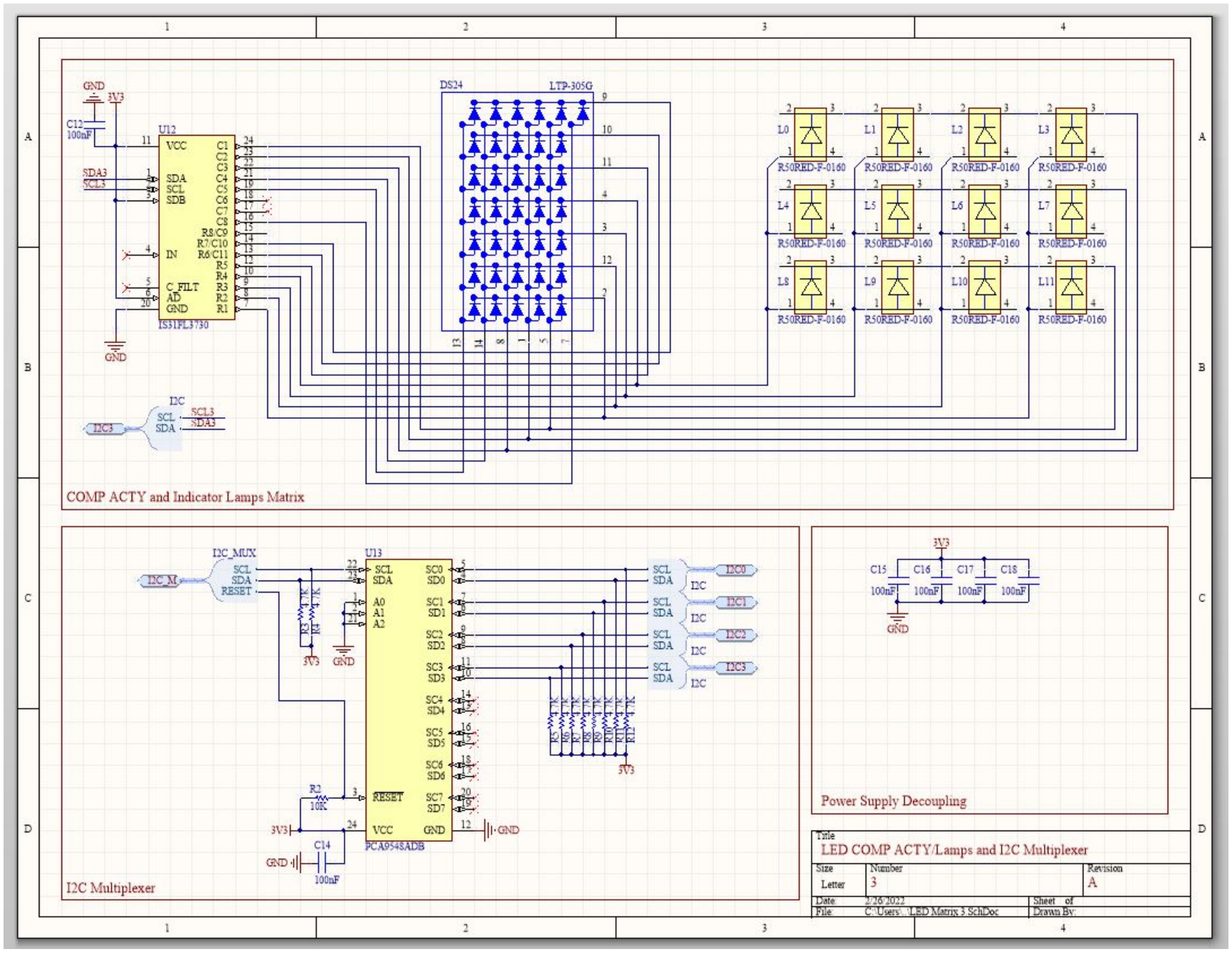


Figure 9: DSKY Schematics for the portion of the LED Lamps and Display

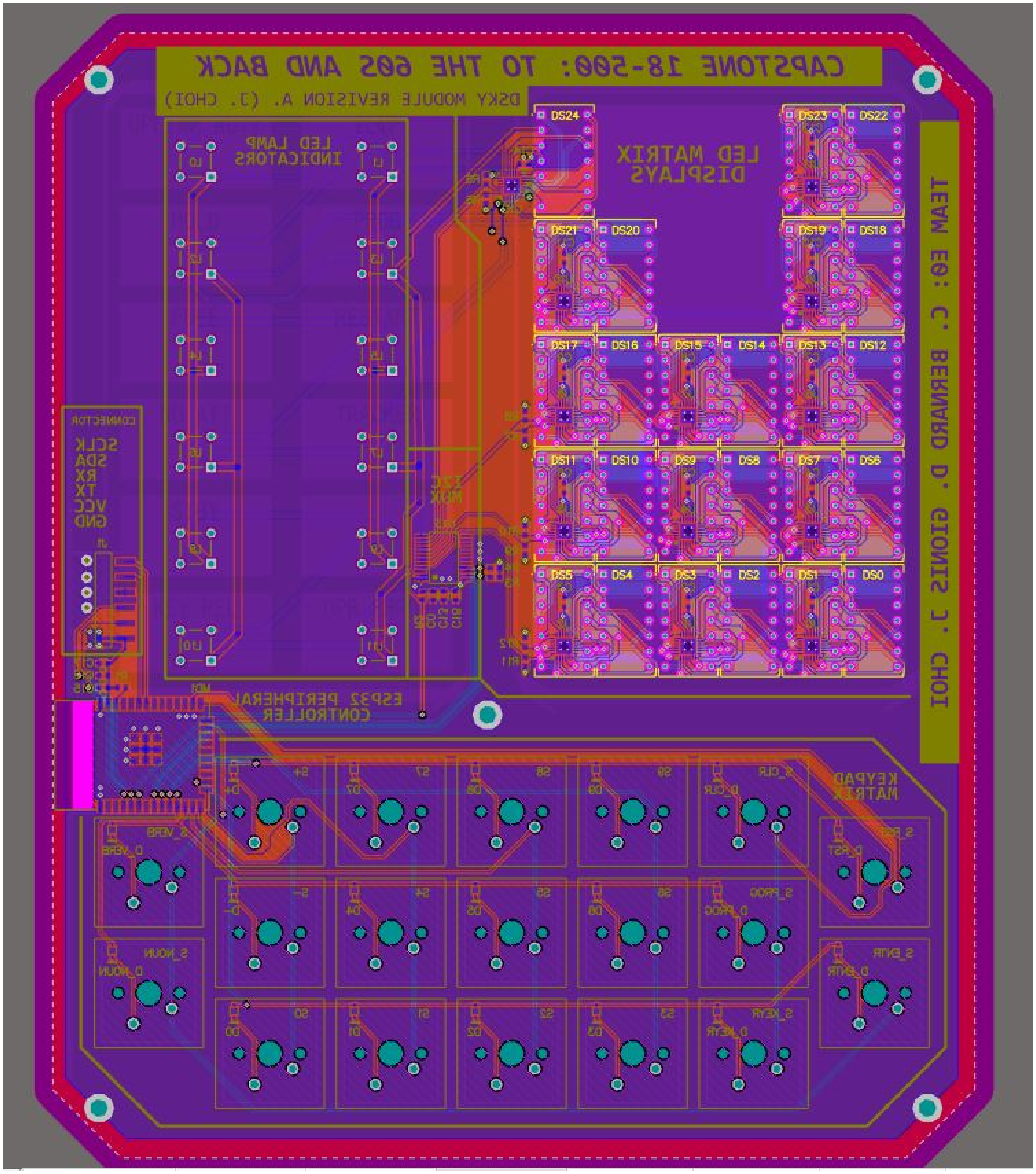


Figure 10: DSKY PCB Layout as seen from the bottom