

Block I

Apollo Guidance Computer (AGC)

How to build one in your basement

Part 1: Overview

John Pultorak
December, 2004

Abstract

This report describes my successful project to build a working reproduction of the 1964 prototype for the Block I Apollo Guidance Computer. The AGC is the flight computer for the Apollo moon landings, and is the world's first integrated circuit computer.

I built it in my basement. It took me 4 years.

If you like, you can build one too. It will take you less time, and yours will be better than mine.

I documented my project in 9 separate .pdf files:

- Part 1 Overview: Introduces the project.
- Part 2 CTL Module: Design and construction of the control module.
- Part 3 PROC Module: Design and construction of the processing (CPU) module.
- Part 4 MEM Module: Design and construction of the memory module.
- Part 5 IO Module: Design and construction of the display/keyboard (DSKY) module.
- Part 6 Assembler: A cross-assembler for AGC software development.
- Part 7 C++ Simulator: A low-level simulator that runs assembled AGC code.
- Part 8 Flight Software: My translation of portions of the COLOSSUS 249 flight software.
- Part 9 Test & Checkout: A suite of test programs in AGC assembly language.

Why build an AGC?

Early computers are interesting. Because they're simple, you can (if you like) actually understand the entire computer, from hardware to software.

The AGC is the most interesting early computer because: a) it flew the first men to the moon; and b) it's the world's first integrated circuit (IC, or microchip) computer. It also has interesting architectural features.

Original AGC:

Designed by M.I.T. in 1964

World's first microchip computer

Prototype computer for Apollo moon landing

Memory: 12K fixed (ROM), 1K eraseable (RAM)

Clock: 1.024 MHz

Computing: 11 instructions, 16 bit word

Logic: ~5000 ICs (3-input NOR gates, RTL logic)

My AGC:

Built from original M.I.T. design documents

Started November 2000, completed October 2004

~15K hand-wrapped wire connections; ~3500 feet of wire

Cost (parts only): \$2,980.

Labor: ~2500 hours

Logic: ~500 ICs (LSTTL logic)

Runs flight software (1969 program name: COLOSSUS 249)

How did I build it?

I collected original documents from libraries, technical services, and the internet. I had to pay for some (~\$350.) Mostly, they're available for free now.

I wrote an AGC software simulation (in C++) and cross-assembler (to program it). Then, I wrote test and checkout programs in AGC assembly language to check the simulator against specifications in the original documents.

I downloaded a big chunk of original flight software from M.I.T. and got it running on my simulator.

The simulator was my baseline for logic design. I put my logic design into a circuit design tool (CircuitMaker) which "captures" schematics and (digitally) simulates circuits. Using CircuitMaker, I "unit-tested" each AGC subsystem logic design (there are about 20).

Then, I assembled my subsystem logic simulations into one really huge digital circuit simulation of the entire AGC and "integration-tested" this for about 6 months. I eventually ran my entire suite of AGC test and checkout software on this circuit simulation. (The simulation was extremely slow, running at about a 500,000 : 1 rate, but very detailed).

I bought parts, mostly from JAMECO, and wire-wrapped the entire computer on 15 circuit boards in my basement. I built the "relay rack" out of 1x2 inch pine boards, rails for shelving (can you spot them?), plexiglass, screws, and spray paint.

Three AGCs

To succeed, I had to build the AGC three times. Or rather, I had to build three AGCs. Each one is a fully functional computer. Each one is a step toward building the next. The last one is the one I wanted.

Here's the "faces" of my three AGCs:

AGC #1: my C++ AGC simulator.

It has every register and logic signal in the original documentation.

I use it to develop and debug AGC assembly language code.

When I got this working, I knew I understood the AGC. I knew I could do the rest of the project.

```

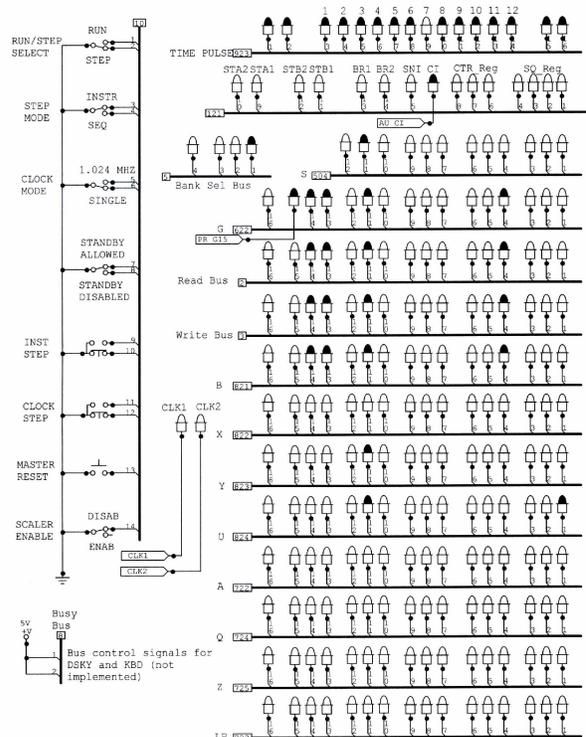
AGC4 SIMULATOR 1.15 -----
TP: STBY F17:0 F13:0 F10:0 SCL:000000
STA:0 STB:0 BR1:0 BR2:0 SNI:0 CI:0 LOOPCTR:0
RPCELL:00000 INH1:0 INH:0 UpCELL:000 DnCELL:000 SQ:00 TC TC0
CP:GENRST
S: 0000 G:000000 P:000000 (r)RUN :0 (p)PURST:1 (F2,F4)CLK:0
RBU:000000 WBU:000000 P2:0 (s)STEP:0
B:000000 CADR:000000 (n)INST:1 PALM: [ ]
X:000000 Y:000000 U:000000 (a)SA :0

00 A:000000 15 BANK:00 36 TIME1:000000 53 OPT Y:000000
01 Q:000000 16 RELINT: 37 TIME3:000000 54 TRKR X:000000
02 Z:000000 17 INHINT: 40 TIME4:000000 55 TRKR Y:000000
03 LP:000000 20 CYR:000000 41 UPLINK:000000 56 TRKR Z:000000
04 IN0:000000 21 SR:000000 42 OUTCR1:000000
05 IN1:000000 22 CYL:000000 43 OUTCR2:000000 CF: [ ]:KR [ ]:PA
06 IN2:000000 23 SL:000000 44 PIPA X:000000
07 IN3:000000 24 ZRUPT:000000 45 PIPA Y:000000 A: [ ] M:[ ]
10 OUT0: 25 BRUPT:000000 46 PIPA Z:000000 V: [ ] N:[ ]
11 OUT1:000000 26 ARUPT:000000 47 CDU X:000000 R1:[ ]
12 OUT2:000000 27 QRUPT:000000 50 CDU Y:000000 R2:[ ]
13 OUT3:000000 34 OVCTR:000000 51 CDU Z:000000 R3:[ ]
14 OUT4:000000 35 TIME2:000000 52 OPT X:000000
>
    
```

AGC #2: my CircuitMaker digital circuit simulator AGC. What you see here are front panel logic indicators for signals and registers. The remaining part: about 60 or so pages of logic diagrams; the same schematics you'll see in the CTL, PROC, MEM, and IO .pdf files that follow this overview.

This AGC literally runs about a half-million times slower than the first! I used it to debug my logic design.

When I got this working, I knew I still had a year (14 months, actually) of tedious work ahead.



AGC #3: This is the final one.

My hardware AGC is built into a relay rack to give it the same prototyped appearance as the original. The dimensions are about 3 x 5 feet.

The upper left module (CTL) is the control logic.

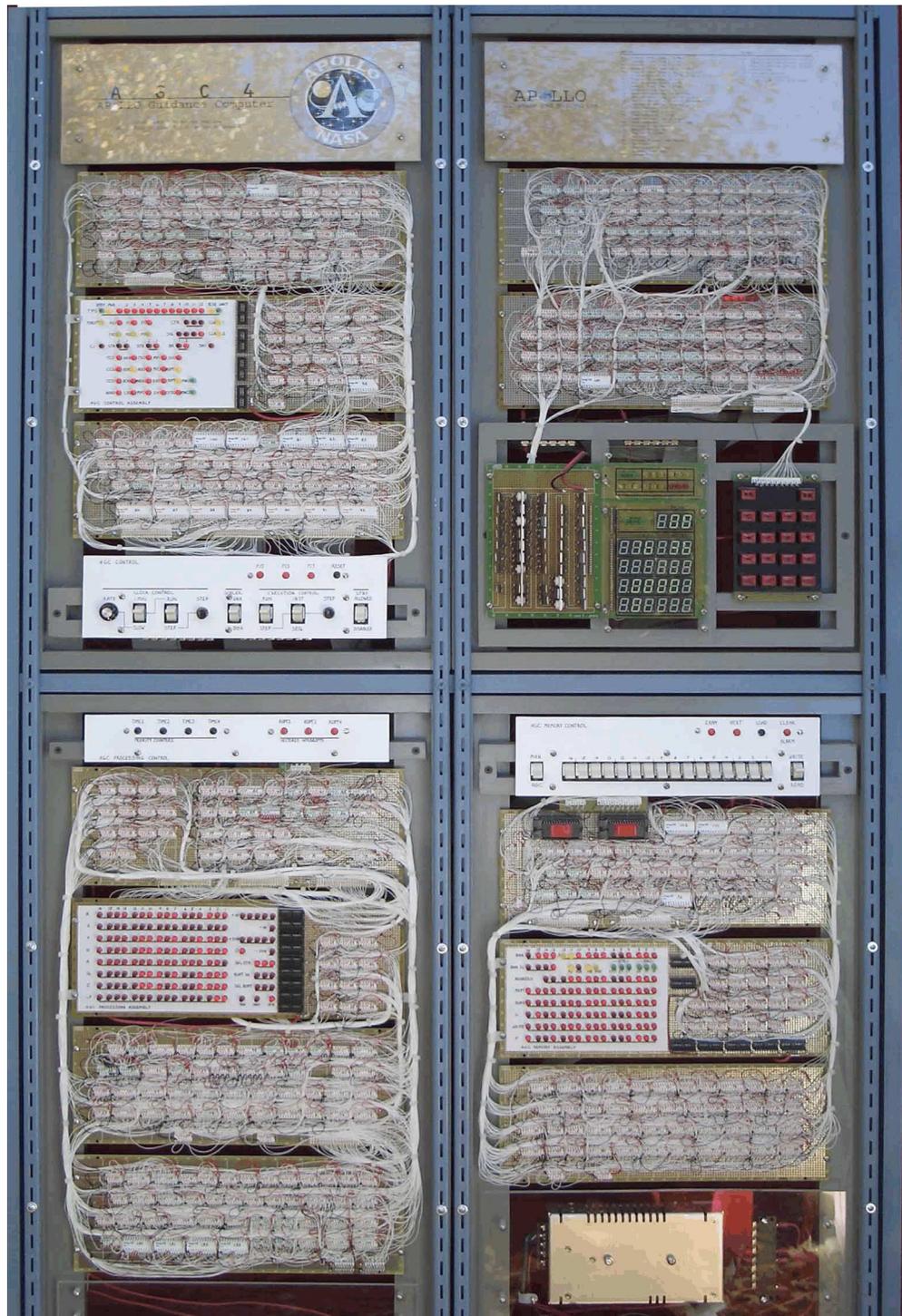
The lower left module (PROC) is the CPU.

The lower right module (MEM) is the memory.

The upper left module (IO) has the inputs and outputs. The keyboard and display (DSKY) for user input are on the upper-middle-right.

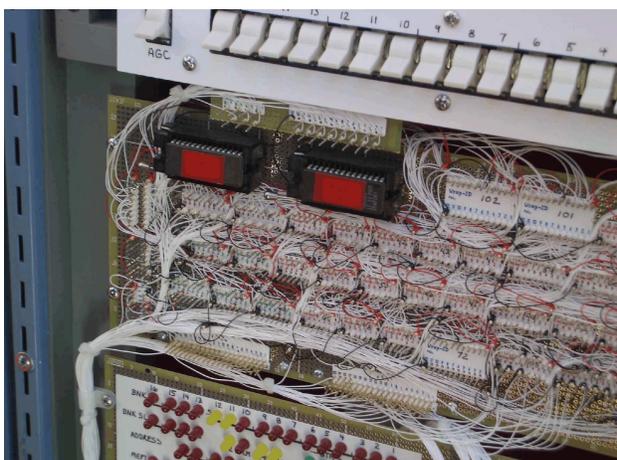
The white KYNAR wires look like cobwebs to me.

This was photographed in my back yard. Can you see the blue sky and tree branch reflections in the upper panels?



Here I am, plugging COLOSSUS 249 flight software EPROMs into ZIF sockets.

We lugged the AGC out of the basement and took it outdoors for these photographs. It's in the backyard, propped against the back of my house.



A close-up of COLOSSUS EPROMs. I put red tape over the quartz windows to keep them from erasing themselves.

A bought the toggle switches a few years ago from a surplus vendor. They were so cheap, I bought a 100 of them. Aren't they great?

I really like flashing lights and control panels. There's no way to debug something this complex without them.

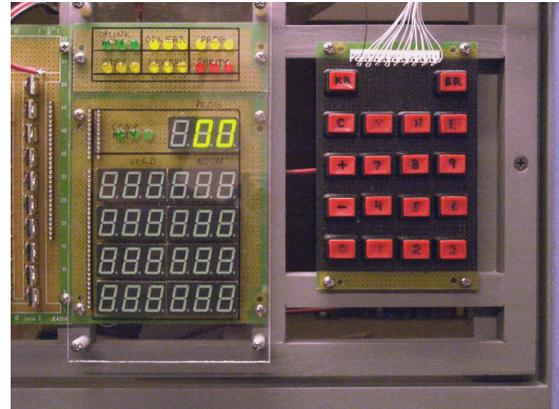
I bought LEDs in bulk from the same place that sold me the switches.



AGC Demonstration

Here's a demonstration of my AGC. It's running my version of the COLOSSUS 249 flight software load:

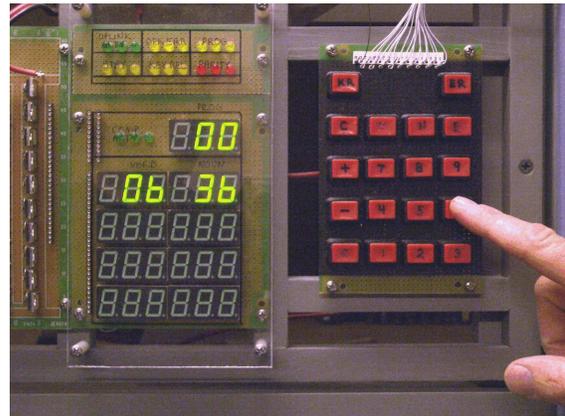
At power-on, the AGC initializes to major mode 00 (POO).



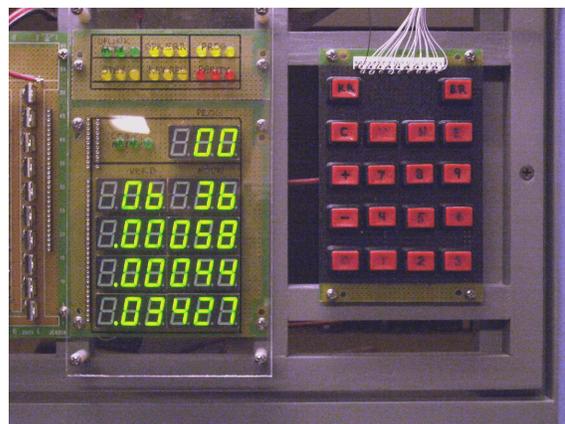
Display CM clock:

<VERB> <0> <6> <NOUN> <3> <6> <ENTER>

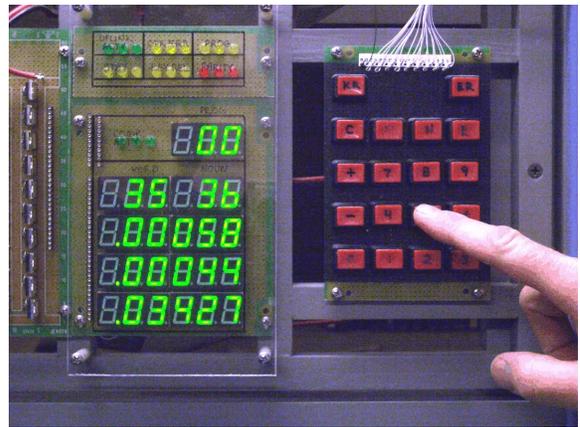
The red buttons were from Radio Shack.



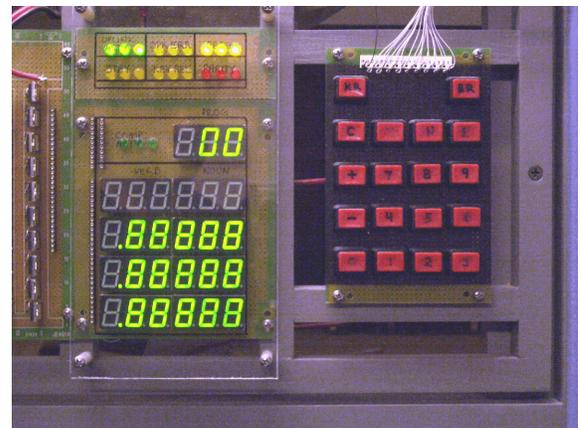
The AGC samples and displays the CM clock. 58 hours, 44 minutes, 34.27 seconds.



Test display lights:
<VERB> <3> <5> <ENTER>

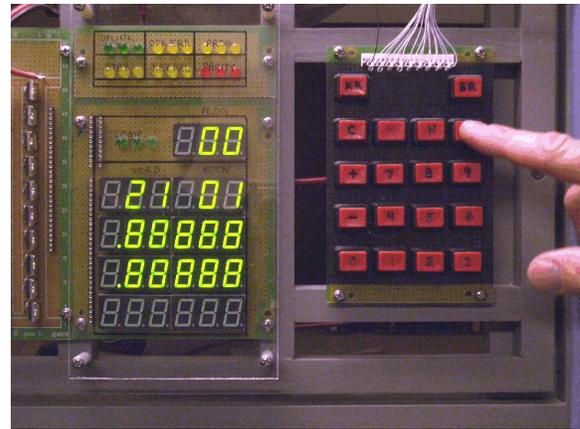


All DSKY lamps and display segments illuminate for 5 sec. The Verb and Noun displays flash. After 5 sec, the DSKY lamps extinguish. The 88888 displays are left in the registers.

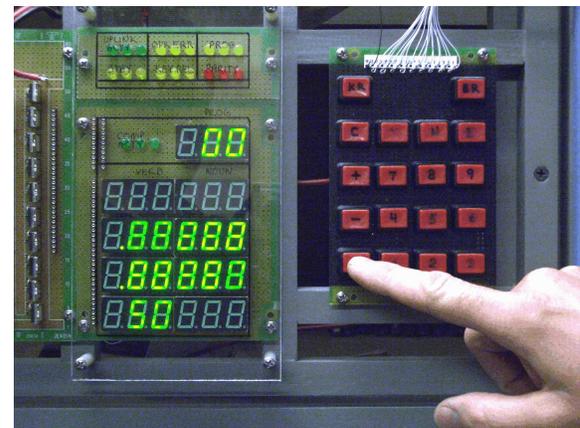


Load component 1 for dataset at octal address 50 with octal 123:

<VERB> <2> <1> <NOUN> <0> <1>
<ENTER>

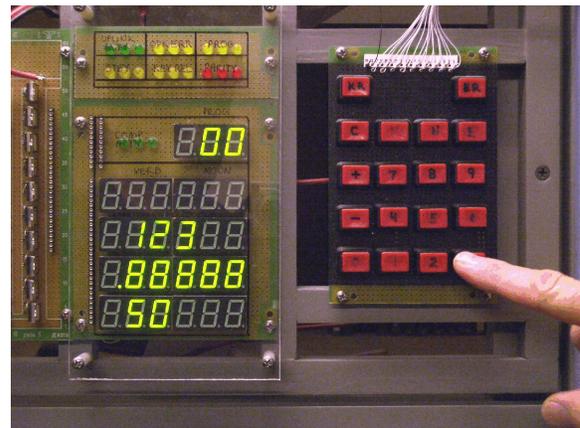


Verb/noun display flashes: waiting for address
<5> <0> <ENTER>

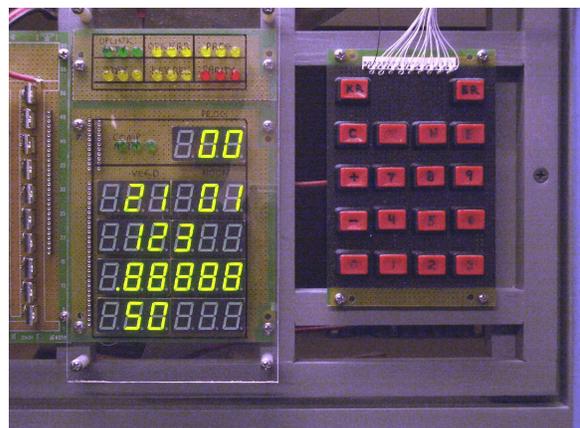


Verb/noun display flash continues: waiting for
data

<1> <2> <3> <ENTER>

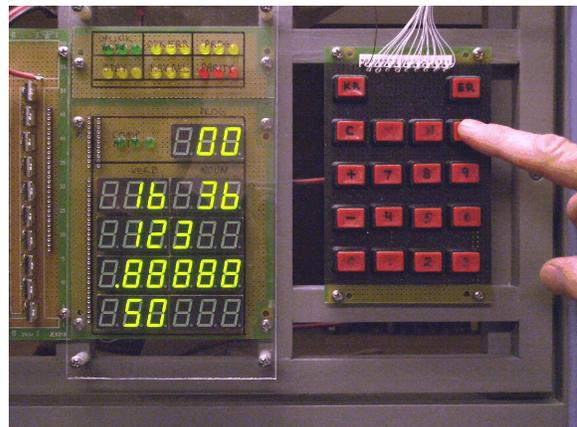


Octal word from R1 is loaded into memory at
address 50.

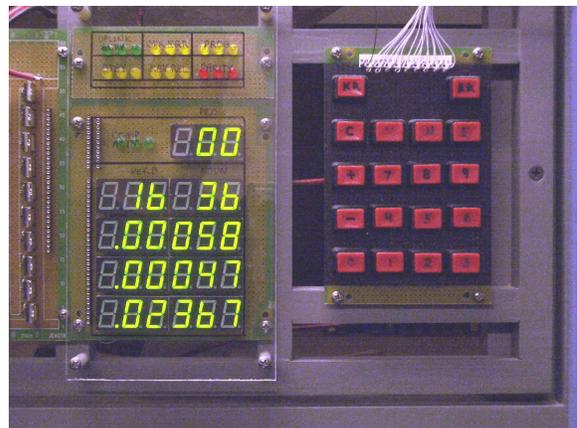


Start a monitor program to continuously display elapsed time from the CM clock:

<VERB> <1> <6> <NOUN> <3> <6>
<ENTER>



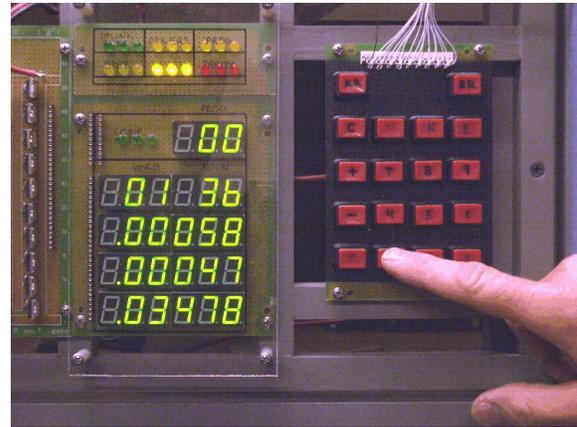
The monitor program starts to continuously display the CM clock, updating the display about 1 second intervals. The time shown is 58 hours, 47 minutes, and 23.67 seconds.



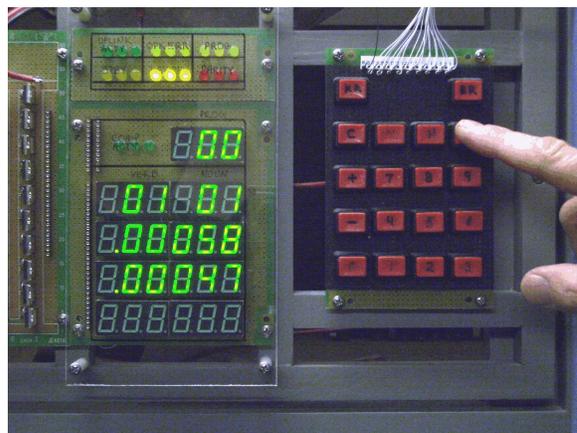
Display component 1 of dataset at octal address 50:

<VERB> <0> <1>

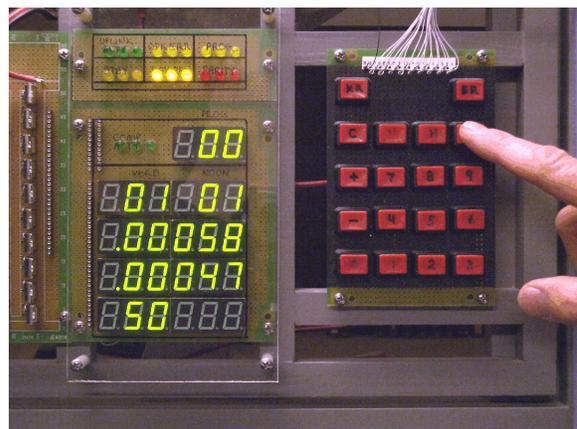
The "key rel" light flashes because the CM clock monitor program has been suspended.



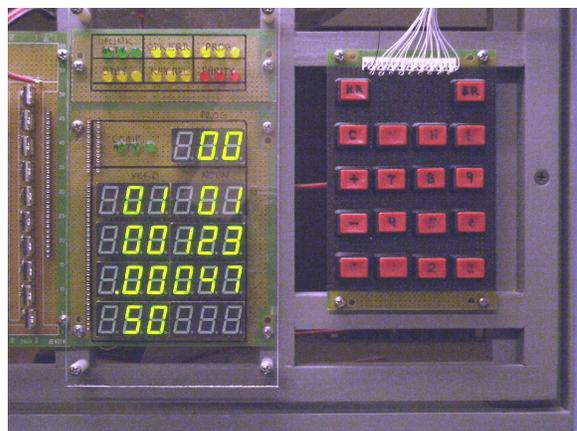
<NOUN> <0> <1> <ENTER>
verb/noun display flashes: waiting for address



<5> <0> <ENTER>

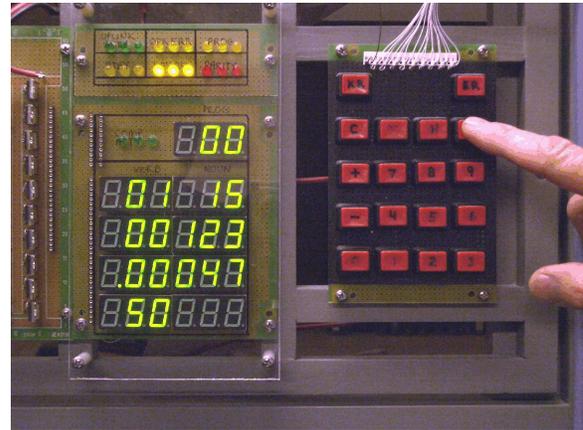


The octal word from address 50 is displayed in R1. In a previous operation, we had set the word to 123.



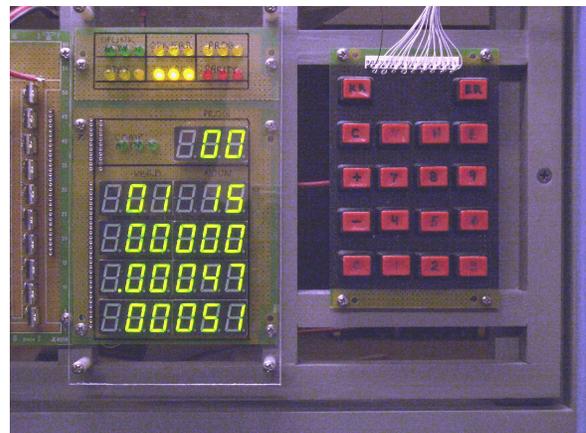
Increment the address:

<NOUN> <1> <5> <ENTER>

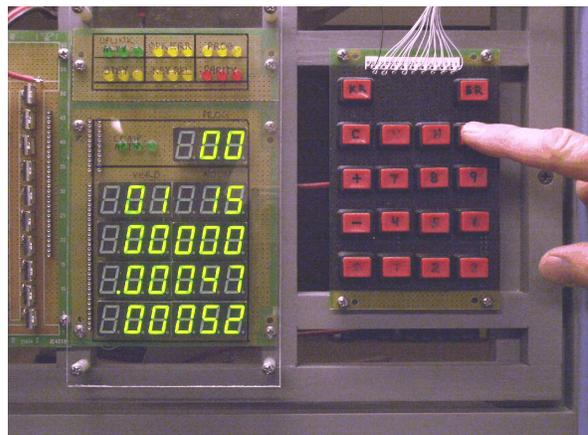


The octal word in address 51 is displayed in R1,
incremented address in R3.

<ENTER>

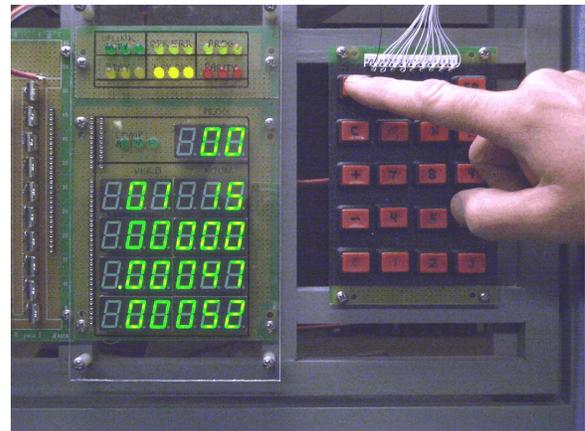


The octal word in address 52 is displayed in R1,
incremented address in R3.



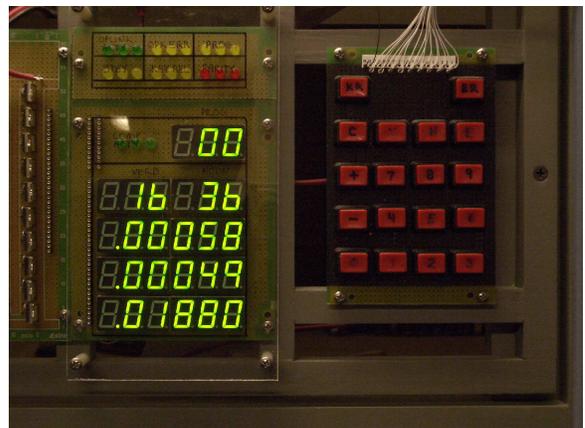
Resume the CM clock monitor program:

<KEY REL>



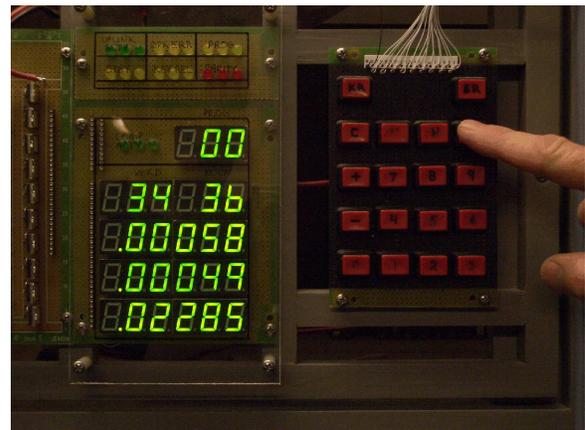
Verb 16, noun 36 reappears, along with the clock display. The key release light goes out.

(I'm not sure why the photos suddenly turned darker; I think someone must have turned off a light in an adjoining room)



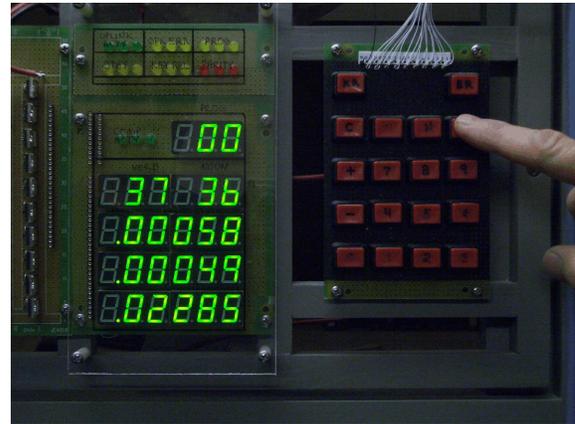
Terminate the CM clock monitor program:

<VERB> <3> <4> <ENTER>

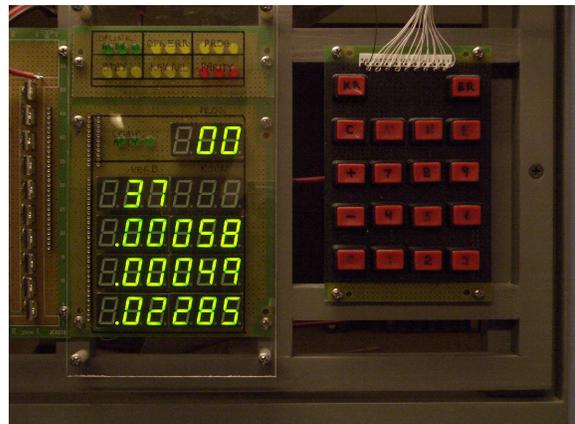


Change major mode to P00:

<VERB> <3> <7> <ENTER>

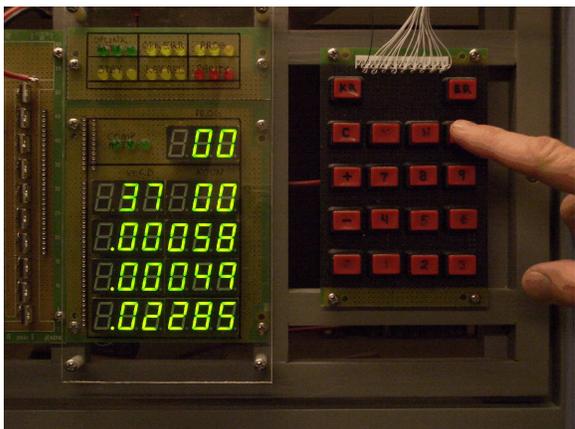


Verb/noun display flashes: waiting for major mode.

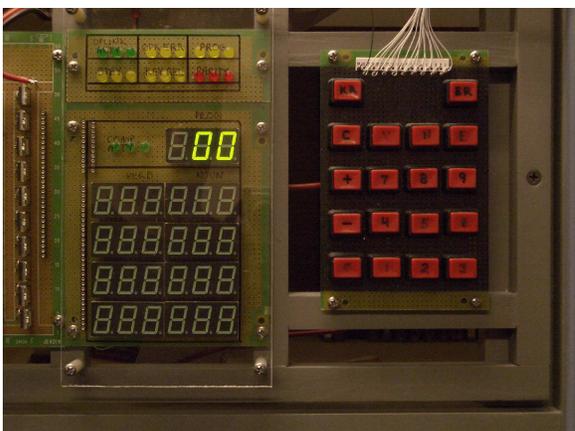


Enter major mode P00:

<0> <0> <ENTER>



AGC starts P00.



AGC Project Diary

October - November, 2000:

Thinking about building a computer. Decided to reproduce a minicomputer from the 1960's or '70's. Started gathering technical documentation on DEC PDP 8 and Data General NOVA.

Found a book in the local library (Ceruzzi, "A history of Modern Computing") with a page on the AGC. Decided to build a NASA flight computer; Gemini or Apollo. Gemini interests me more, because it's the earlier. Downloaded Tomayko's "Computers in Spaceflight."

December 2000:

My search for Gemini computer documentation on CASI and NTIS is fruitless. IBM federal systems division is gone. Found some interesting AGC documentation. Asked for, and received, E.C. Hall's "History of the Apollo Guidance Computer" for Christmas.

January - February 2001:

Decided to build an AGC. Ordered about \$300. of technical documents from CASI and NTIS. Discovered you can't judge a document by its title.

Sent e-mail to Goddard Space Flight Center; got a copy of the AGC Basic Training Manual, a programmer's manual for the Block II. Went to CompUSA and bought a Microsoft C++ compiler. Starting to build a AGC Block II software simulation.

Sent an e-mail to Draper Labs (former MIT Instrumentation Lab where AGC was designed) asking for R-393 (Block I "Logic Description"). This might be the key document. Draper responds by sending R-393 (free!).

JACKPOT!

Abandoning Block II simulator. I am building a Block I AGC.

March - May 2001:

Rapid progress on the Block I simulation. Wrote an AGC cross-assembler in C++; it produces object code readable by the AGC simulator. Generating test code to debug AGC basic instructions.

Designed the simulator to map as closely as possible to the R-393 hardware registers, buses, microinstructions, and control signals. Broke the simulation into 20 subsystems that map to AGC hardware. I will eventually use the simulator code to guide my hardware logic design.

Finished emulation of all 11 normal and extracode instructions. Wrote my first Block I assembly language program! Starting to simulate the priority counters and interrupts.

June - August 2001:

Finished v1.0 of the Block I simulator. Wrote a suite of AGC assembly language test and checkout programs: TECO1 checks basic instructions; TECO2 checks extracode instructions; TECO3 checks editing registers. Everything works, including interrupts and counters.

Found MIT website with AGC history and interesting online documentation.

E-mailed Eldon Hall, designer of the AGC, telling him about my project. His reply was gracious and encouraging. Wrote many emails to others asking for Block I source code.

I order a 2.048 MHz crystal from DigiKey. My first hardware purchase.

September - October 2001:

Can't find any original Block I source code, except tiny fragments in the documents. Recoded my own EXEC, WAITLIST, and interrupt handlers using specifications from R-393 and others. I'm starting to become a good AGC programmer. Now my simulator can multitask!

Discovered Block II flight software (COLOSSUS 249) is now downloadable from MIT. 300+ pages of blurry assembler source code listing. Is that an eight, a six, number zero with a slash through it, or the letter "O", or maybe "G"? Printed a hardcopy. I think I can make most of it out.

The second half of COLOSSUS is missing! The missing part has the INTERBANK COMMUNICATION, EXEC, and WAITLIST. E-mailed MIT. Their reply: they don't have the missing portion.

November - December 2001:

Tried to reproduce DSKY code using flowcharts from Green's "Keyboard and Display System Program". Green calls it PINBALL GAME. Very confusing. Started writing a C++ simulation from Green's flowcharts. Things are becoming clearer.

Located PINBALL in my COLOSSUS fragment. Abandoned effort to code my own. I will use the real thing.

January - February 2002:

Retyped most of COLOSSUS PINBALL back into machine-readable form. 95% is already Block I instructions; recoded the remaining 5% into Block I equivalent. Finished all the octal display verbs (1-5) and decimal verb (6) and the load verbs (21-25), but they're only tested for octal loads so far. Noun tables are stubbed, but I can manually set them for different scaling.

Integrated PINBALL into my EXEC and WAITLIST. Coded up a few missing pieces for interbank communication. Also had to code a math library (yikes). But it works.

The AGC simulator is running at about 1/30 the speed of the original. I need to speed it up.

March - May 2002:

Bought a new, faster PC. The simulator now runs at about 1/5 speed. Recoded some simulator pieces. Now its 1:1.

Finished PINBALL. All regular verbs (0-39) work. Also, normal nouns 1, 2, 3, 9, 15, 26, and 36. Very cool. My favorites: verb 16, noun 36, a monitor routine to continuously display the AGC clock, and verb 35 which does the lamp test.

Now that I have some proficiency, I am reluctant to leave AGC software, but it's time to start hardware logic design.

June - December 2002:

Decided to use 74LS logic family. Started designing AGC logic, subsystem-by-subsystem, using the C++ simulator code as my guide. Began with MBF. Posted a subsystem block diagram on the wall. I'm coloring in blocks as I finish each subsystem design.

Entered logic designs into the CircuitMaker PC tool. Using CircuitMaker's digital circuit simulation to unit-test each subsystem.

Struggling with ALU and read/write bus design. The original AGC OR'ed everything onto the bus, and the default bus state was logical zero: sometimes, they read the bus without writing to clear registers. Other times, they wrote multiple registers simultaneously to OR the data. These tricks won't work with tri-state bus drivers. I identify where tricks are used and add ALU logic to handle the cases. My ALU sort-of feeds back on itself. Confusing, but it should work.

Logic design the old way: Karnaugh maps, excitation tables, and bubble-pushing. Fun, sort of.

Logic design is now finished, except for the DSKY. Unit tests are done.

I start ordering parts from JAMECO. The first order is for more than 200 ICs.

January 2003:

DSKY logic design is now finished and unit tested in CircuitMaker. All blocks on my diagram are colored in. Will the subsystems work together?

February - May 2003:

Using CircuitMaker to integrate subsystems. Diagrams for each subsystem are hooked into CircuitMaker "macros"; rectangular components with all inputs and outputs for that subsystem. "Wired" all subsystem macros together. Will it run? I call it AGC2, to differentiate it from the C++ simulator, which I now call AGC1.

Now I have two AGCs! When I build the hardware, that will make three.

Started debugging a TC instruction, the simplest, in AGC2. Worked it through, step-by-step, fixing logic and interface errors. Finally, it works. Debugging the remaining 8 basic instructions.

Massive snowstorm; snowed in, 3 days straight. Lots of time for AGC2 debugging and testing.

I estimate my power budget and order a 15A 5V supply. More ICs and sockets are ordered, too.

Sick of hand-assembling test code for AGC2. Wrote a version of the cross-assembler that produces object code for AGC2. Broke TECO1 into 8 parts; one for each instruction. One-by-one, I get all portions of TECO1 to run on AGC2.

Broke TECO2 and TECO3 into pieces and got them to run on AGC2 also.
INTEGRATION TESTING IS DONE!

How to build it? There are too many subsystems and interfaces.

June - August 2003

Grouped the 20 subsystems into 4 "assemblies" (soon to be called "modules"): I/O, CTL, PROC, and MEM. This is more manageable.

Wrote C++ code that checks the netlists produced by CircuitMaker for fan-out. Established a limit of 8 TTL loads, and added buffers where necessary. Added buffers between the 4 modules to make the fan-in for each module/module interface 1 TTL load.

Stuffed IC sockets into 13 circuit boards; each board is 13"x5". What connectors and cables to use between boards? Between modules? Should I worry about bus termination? What kind of chassis?

Decided to build each module as a small relay rack. Board-to-board connections inside each module are wire-wrapped--no connectors. Between modules, 40-pin IDE cables, mating to 30-pin wire-wrap connectors, are for module/module interfaces.

Too lazy to pull 500 IC sockets and redo the boards. I'll work in connectors and additional buffers where I can. Better buy the longest IDE cables (3 feet). More worry about bus termination.

Module/Module interfaces are now defined: 6 IDE cables.

Built a rack for the I/O module out of 1x2 pine boards. Spray-painted gray; it looks pretty good. Hired my engineering student son to wire-wrap some I/O module logic during summer vacation. He wires most of the DSKY. It lights up and multiplexes, but operation is erratic; set aside for now.

September - December 2003:

Built control panels for PROC, CTL, and MEM modules by mounting switches on a wooden frame. Used thick styrene plastic for a faceplate, hand-lettered with indelible marker. It doesn't look too bad.

Built indicator light panels for PROC, CTL, and MEM by punching holes in styrene plastic with a push-pin and then shoving LEDs through the plastic into the PCB. Hundreds of LEDs; my thumb has a blister.

Built 3 more relay racks for the PROC, CTL, and MEM modules. Laid all the boards out on the racks. Will the IDE cables reach? Yes, but in some cases, barely.

Bought an EPROM programmer. Learned Motorola S-format. Wrote yet another version of the cross-assembler that outputs S-Record (s2f) format object code. Burned EPROMs with TECO1, TECO2, TECO3, and the AGC flight software.

Modified the C++ simulator (AGC1) so it dumps its microinstructions (subsystem CPM-A) in Motorola S-format. Burned EPROMs for CPM-A. Created a special version of AGC1 that reads CPM-A S-format microinstructions to verify the tables and reads AGC object code in S-format to verify the cross-assembler.

January - April 2004

Powered up, and started debugging the partly completed I/O module. Corrected a design error and a few minor wiring errors; cleaned up some cold-solder joints. It now works reliably. Finished wiring the I/O module. It's difficult to test, but it seems to work.

May - September 2004

Wired the power connections for all sockets on the CTL module; added a bypassing capacitor for every 2 packages, and a 10uf tantalum capacitor for each row of ICs. Wired the LED lamps and drivers, and then the logic for each subsystem. Plugged all the CTL chips into the sockets.

Discovered all control signals from CPM-A were inverted: the EPROM tables were generated from AGC1, which uses positive logic. The hardware AGC uses negative logic, because TTL inputs float high; I wanted floating inputs to assume the inactive state during assembly and test. Pried the EPROMs out of their sockets, bit-flipped the tables, erased and reprogrammed the chips, and reinserted them. Now it works.

Completed wiring for the other modules. Now to hook them up.

September - October 2004

Built a large rack to hold the 4 modules. Screwed the 4 modules into the rack and hooked up the IDE cables. Powered it on. Everything lights up. No smoke. Amazing! It runs part of TECO1, getting through dozens of instructions including subroutine calls, before bombing out.

Trying to debug the AGC by burning test EPROMs, single-stepping the clock, and comparing the results to the AGC1 C++ simulator. It's acting flaky. Could it be a supply problem?

Tore out my flimsy power distribution; replaced it with heavy aluminum bus bars. Supply lines look cleaner on the scope, but the operation is just as flaky as before. Maybe worse. It's bombing out in different places. Is there some common element?

Common element: the problem always involves read bus microinstructions. The ALU sets a default state on the read bus if no other subsystem is using it. My bus arbitration scheme stinks: if a read signal is asserted, the ALU disables its default state, but propagation delays cause both drivers to enable simultaneously for a brief period. Is this the problem?

I kludge-in logic to disable the read bus during CLK1. This gives the read bus logic time to settle. I add propagation delay to the CLK1 input to TPG so the bus is disabled before the TPG state transition occurs. Will this fix the problem?

No. It gets farther along, but still bombs out on read bus operations. It's time to download internet advice on bus termination. I add 220/330 ohm passive termination to the read bus input in the ALU. IT WORKS!! TECO1 and TECO3 run flawlessly.

TECO2 bombs out while testing the DV (divide) instruction. It produces different results than the AGC1 simulator in the tests that involve division by zero. Do I care? I decide I don't.

I load the COLOSSUS EPROMs. The AGC flight software hangs at first; but I add some passive termination to the "busy" bus driver (for the read bus, of course) and then it works flawlessly too. The project is finished (except, I have to write up these reports!)

AGC Block I Diagram

I constructed this diagram in the first months of my project. It shows the AGC subsystems as boxes. I gave them three-letter names. My diagram is derived from a similar block II diagram I found in R-700.

I used this diagram to organize my simulator and logic designs. But, when it became time to build the AGC, I felt this diagram had too many interfaces; too much complexity.

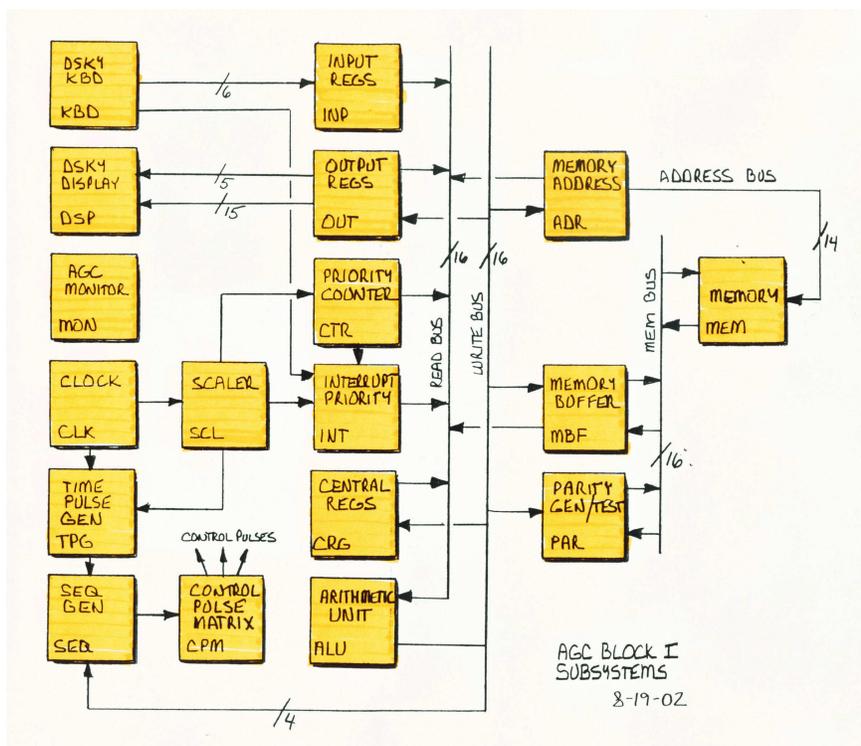
To simplify things, I grouped the subsystems from this diagram into 4 modules.

The KBD, INP, OUT, and DSP subsystems were external interfaces, so I grouped them into an IO module. I'll tell you about it in part 5 of this report.

The ADR, MEM, MBF, and PAR subsystems dealt with memory, so they went into a MEM module (described in part 4).

The MON, CLK, SCL, TPG, SEQ, and CPM subsystems generated the timing and control pulses that run all the other subsystems, so I gathered them into a CTL (control) module (in part 2).

What remained? The ALU, CRG, INT, and CTR subsystems. I put them in a PROC (processing) module (part 3).



How my AGC differs from the original

For my purposes, the original AGC is described in a M.I.T. document called R-393 (A. Hopkins, R. Alonso, and H. Blair-Smith, "Logical Description for the Apollo Guidance Computer (AGC4)", R-393, MIT Instrumentation Laboratory, Cambridge, MA, Mar. 1963).

Logic Design

The original AGC4 was built almost entirely from 1964-era 3-input NOR gate ICs; about 5,000 of them. Original gate-level logic designs are not available.

Logic for my replica was recreated using specifications in R-393, and architecture information/diagrams from R-700. Since R-393 defines (in detail) AGC registers, register transfers, microinstructions, and most control pulses (logic signals), the architecture of my replica closely mirrors the original to a low level.

The logic design for my replica uses late 1960's-early 1970's 74LS TTL logic, which affords about a 10-to-1 reduction in package count. Flip-flop and register chips are used instead of constructing each element from NOR gates.

The replica successfully runs those portions of the original COLOSSUS flight software that have been loaded onto it.

Clock

The original AGC4 divided its 2.048 MHz clock down into 4-phase 1.024 MHz signals to drive asynchronous, level-triggered, register logic.

My replica divides the 2.048 MHz clock into a 2 phased, non-overlapping 1.024 MHz clock to drive synchronous, edge-triggered register logic. Phase 1 of the clock (CLK1) steps a sequencer (TPG), which sets up the control signals. Those signals have time to propagate (settle) between phase 1 and phase 2. Phase 2 (CLK2) clocks the registers using the control signals established in phase 1. Data transfer occurs at phase 2.

Timing Pulses and Control States

The original AGC4 operation was controlled by 12 non-overlapping timing pulses (TP1 - TP12) generated from a 4-bit gray code counter. Two SR FFs (called R1 and R2) were used to generate 3 states to control standby operation (STANDBY, POWERON, RUN). R-393 identifies a start/stop logic subsystem which gates the time pulse generator to run and halt AGC4 for ground test purposes but the internals were not defined.

My control states are integrated into one state machine with 16 states. The 12 timing pulses (TP1 - TP12) are implemented as states along with 2 additional states for standby operation (STBY, PWRON). The remaining 2 states (RLSE, WAIT) handle front panel switch inputs for single stepping.

Interpolation of Gaps in Control Logic

R-393 defines control pulses for TP1-TP11, but does not define control pulses for TP12. Interfaces between interrupts, involuntary counters, and the main control logic are not well defined. For this reason, logic in the gaps in R-393 had to be interpolated, based upon functional requirements.

Number and Address of Involuntary Counters

The number of involuntary counters and their address range is ambiguously defined in R-

393. Table 1-1 in R-393 says AGC4 has 20 (base 10, I assume) counters. This is supported by Figure 2-5 which numbers counters from 1-20. However, Table 3-1 which shows AGC special registers assigns the counters to addresses 0030 through 0056 (base 8) which translates to 23 (base 10) counters. And Table 5-1, section D gives the counter addresses from 0034 for OVCTR through 0056 for TRKR R, which translates to 19 (base 10) counters. So, its unclear whether AGC4 had 19, 20, or 23 counters and whether the counter addresses start at 0030 or 0034.

To resolve the ambiguity, I set the starting address for the counters to 34 (octal), which is the starting address used in the Block II AGC (COLOSSUS program listing). For convenience, I only implemented counters that were used by the AGC EXEC and WAITLIST, the real-time clock, and the overflow (5 in all). These are:

address	counter
34	OVCTR
35	TIME2
36	TIME1
37	TIME3
40	TIME4

I also used the Block II ordering of TIME1 and TIME2, for compatibility with the COLOSSUS flight software. In the AGC4, TIME1 is at the lower address.

Address of Interrupt Vectors and GOPROG

In the Block II AGC, GOPROG (the starting address for execution) is at the bottom of fixed memory at address 4000, and the interrupt vectors follow at 4004, 4010, 4014, etc. By extension, it would seem that the Block I AGC GOPROG would be at the bottom of fixed memory at 2000, followed by interrupt vectors at 2004, 2010, 2014, etc. However, R-393 has the interrupt vectors starting at the bottom of fixed memory at 2000 (according to the RUPT3 sequence on pages 3-67/3-68). R-393 doesn't identify the address for the Block I GOPROG.

For compatibility with the COLOSSUS source code (which I translated and cross-assembled for Block I), I set GOPROG to 2000, with the interrupt vectors beginning at 2004.

Number of Interrupts

The original AGC had 5 vectored interrupts: UPRUPT, ERUPT, KEYRUPT, T3RUPT, and DSRUPT. UPRUPT was used for uplinked commands; ERUPT was generated when hardware errors were detected. Since I didn't plan to use these interrupts, I eliminated the hardware that supports them from my replica.

My replica implements the remaining 3 interrupts: KEYRUPT, T3RUPT, and DSRUPT (T4RUPT).

Priority Counter sequences

The original AGC4 implemented 3 sequences: PINC, MINC, and SHINC. PINC increments counter locations in memory; MINC decrements them. SHINC implements a shift operation used to shift telemetry bits into the AGC. After 16 bits were shifted into an AGC word, an interrupt UPRUPT was triggered, so the AGC could pull the telemetry word out of the counter.

Since I'm not receiving telemetry, I didn't implement the SHINC subsequence or the UPRUPT interrupt.

Memory

AGC memory is 16-bit words, organized into 1024 word banks. The lowest bank (bank 0) is erasable memory, originally core, but implemented in my replica as static RAM. All banks above bank 0 are fixed memory (originally implemented as rope core, but implemented in my replica as EPROM). AGC4 initially had 12K words of fixed memory. My replica has 15K.

Buses

The original AGC OR'ed everything onto the bus, and the default bus state was logical zero: Registers were sometimes cleared by reading the default state off the bus. Other times, several registers were simultaneously read to OR the data. Because these tricks won't work well with tri-state bus drivers, I identified these instances and added logic to the ALU to handle them.

Flight Software

The original Block I flight software is not available (at least, to me). The Block II software (COLOSSUS 249) is available. Block II is an extension of the Block I instruction set. However, most of the Block II software was originally coded as Block I, so translating the Block II code back to Block I only involves changing about 5% of the instructions back to their Block I equivalents. This is what I did.

Back in 2002, only a portion of the COLOSSUS Block II code was available. Some key portions, such as the EXEC and WAITLIST, were missing. I coded these parts in, using information from the M.I.T. documents and the interfaces and functional requirements implied by their use in the portions of COLOSSUS that were available.

Sources

Many of these sources are now (2004) available (free!) online at <http://hrst.mit.edu/hrs/apollo/public/>

- R. Alonso, J. H. Laning, Jr. and H. Blair-Smith, "Preliminary MOD 3C Programmer's Manual", E-1077, MIT Instrumentation Laboratory, Cambridge, MA, Nov. 1961.
Useful information on AGC4's predecessor: AGC3. AGC3 had fewer instructions (8 vs. 11) and a shorter instruction cycle (8 vs 12 timing pulses). It is primarily helpful for its presentation of AGC Block I programming techniques and examples.
- A. I. Green and J. J. Rocchio, "Keyboard and Display System Program for AGC (Program Sunrise)", E-1574, MIT Instrumentation Laboratory, Cambridge, MA, Aug. 1964.
Flowcharts for DSKY software; no source code. Gives input codes for the DSKY keyboard and the output codes for display characters and registers.
- E. C. Hall, "Journey to the Moon: The History of the Apollo Guidance Computer", AIAA, Reston VA, 1996.
Nice information on the AGC development history with photos; R-700 (also by E.C. Hall) is a better technical summary.
- E. C. Hall, "MIT's Role in Project Apollo, Volume III, Computer Subsystem", R-700, MIT Charles Stark Draper Laboratory, Cambridge, MA, Aug. 1972.
An excellent overview of the AGC; more technical than Hall's "Journey to the Moon" book. It contains an excellent diagram of the Block II register architecture and a nice diagram of a bit-slice of the register/bus logic. My copy from NTIS is somewhat poor quality. There is also a useful discussion of the AGC test and checkout software.
- A. Hopkins, "Guidance Computer Design, Part VI"
Extracted from some (unknown) larger document. An excellent overview of the Block II AGC with emphasis on the I/O circuits. A very useful discussion of number representation and overflow handling in the AGC, which is unconventional.
- A. Hopkins, R. Alonso, and H. Blair-Smith, "Logical Description for the Apollo Guidance Computer (AGC4)", R-393, MIT Instrumentation Laboratory, Cambridge, MA, Mar. 1963.
My primary source. It has a nearly complete specification of the AGC4 (Block I) instruction set, register transfers, and control pulses. Information on the logic design is largely absent, however. There are some internal inconsistencies and gaps in the definition of the control logic: particularly at TP12, in the memory cycle, and at the intersection of the control logic with the interrupts and involuntary counters. Unfortunately, there are few diagrams; its mostly text and tables. There are also some examples of double-precision math routines.
- B. I. Savage and A. Drake, "AGC4 Basic Training Manual, Volume I", E-2052, MIT Instrumentation Laboratory, Cambridge, MA, Jan. 1967.
The software manual for the Block II AGC. It has a fairly complete presentation of the instruction set, but lacks example code.

Now, what?

There's 8 more parts to this report. Download and read the parts you're interested in.

Perhaps you want to build your own AGC. You can use my software and logic designs, or develop your own. There's lots of room for improvement in my work. You could use it as your starting point.

If you like, you can contact me at: agcproject@msn.com