**Apollo Guidance Computer (AGC) Block III: Overview**

*Hugh Blair-Smith, MIT Instrumentation/Draper Lab, 1959-1981*

Full disclosure first: there never was a Block III AGC in the Apollo era. There were three generations of AGC designed by MIT’s Instrumentation Laboratory, all featuring 15-bit words plus a parity bit, but their naming was, shall we say, idiosyncratic:

1. AGC3 was so named because it was “Mod 3C,” the third generation of design for an unmanned U.S. Air Force spacecraft to make a photo reconnaissance of Mars, a program killed by the birth of NASA.
2. AGC Block I was originally named AGC4, as the successor to AGC3. The Block designation was imposed to make us stop improving AGC4 and hold most of our creative ideas for the next generation. Block I flew only in unmanned flights.
3. AGC Block II was used in all manned flights in the Apollo program, and in all Apollo Command Module flights ferrying astronauts to and from the Skylab space station, and in the Apollo-Soyuz Test Program.

All of those models used a modified Harvard architecture, with separate physical memory units for instructions and data. The modification was the one created years earlier by Johnny von Neumann: the Read-Only “fixed” program memory could also contain data words, and the Read-Write “erasable” data memory could also be used for instructions. The reasons for two distinct memory technologies were compactness and reliability. The fixed memory was woven like a mass of transformer wiring whose information content could not be corrupted by anything less than physical destruction. Erasable memory, built as a coincident-current ferrite core memory, could be altered by electromagnetic events, but obtained some protection from a sort of “firewall” that disabled all such changes when an abnormal surge occurred—like the lightning strike in the Apollo 12 launch.

Word Length Considerations

In an environment where size, weight and power consumption all had to be minimized, and where speed requirements mandated parallel processing of the bits in each word, it was crucial to make the data word length just enough. For many variables used in trans-Lunar navigation, a little over eight digits of precision was required, equivalent to about 28 bits. But for “housekeeping” data, like timing intervals and addresses to access each word in a memory of 4K up to maybe 32K words, half of that was enough. So it made sense to make the standard word length 16 bits including parity, allowing for 14-bit signed or 15-bit unsigned numbers, and use double precision wherever needed for navigational variables. The length of a typical instruction certainly didn’t need to be 30 bits, and could be made, with a little ingenuity, to fit into 15 bits. A 3-bit operation code field would accommodate 8 different instruction types including multiplication, a snug fit but usable. A 12-bit address field would provide random access to 4096 words of the two memories combined, which seemed adequate if not generous. All that fit together well enough while we thought the job would be much smaller than it turned out to be.

Even at that level, the instruction-word format was a little too snug. Mainstream computers of the time devoted a few bits of each instruction word to controlling address modification: select from a few index registers, from which numbers could be added to or subtracted from the address field, and usually some way to indicate indirect addressing, reaching into erasable memory to get the actual address to use. But after allocating 3 bits to op code and 12 to address, there was nothing left over for such purposes.

Architecture of Memory and Addresses

Maybe 3584 words of fixed and 512 of erasable would have been enough for an unmanned mission to photograph another planet. But soon enough, the realities of manned travel to the Moon became clearer. When, for hardware reliability reasons, we had to change technology to the still-infant art of integrated circuits, we took the opportunity to make what felt like generous increases. We doubled erasable memory to 1024 words, and more than tripled fixed memory to 12K words, providing a new “bank-switching” feature to time-share 1024 addresses among 10 banks of 1024 words of fixed. That left 2048 words of fixed memory with unambiguous assignments to physical program storage, essential for central multi-purpose parts of the program; to that last part we gave the illogical-sounding name of “fixed-fixed.” We devised a similar trick to share certain op codes among different instructions, and were thus enabled to add subtraction and division to the repertoire. That was the original AGC4.

At that time, the idea of a Lunar lander Module was just being born, so we had no precise idea of what its computer should do, but we did insist that for reliability of both hardware and software the two spacecraft should use the same type of computer. But even on the Command Module side, it was time to double the word count of fixed memory to 24K, upping the number of switchable 1024-word banks from 10 to 22. With no other changes in instruction repertoire or logic, that became AGC Block I.

Requirements for the Lunar Module system became known, including those for a Digital AutoPilot (DAP)—so another quantum leap was needed. We doubled the size of erasable again, to 2048 words with a little bank-switching scheme of its own to time-share the top 256 addresses of a 1024-word address range among 5 “E-banks.” We jumped the capacity of fixed memory to 36K, of the same physical size but triple the density of AGC4’s original 12K, and overflowed the capacity of our fixed bank-switching scheme, which wasn’t prepared to go beyond 32K. So we cobbled up a kludgy thing called a “superbank” field that would have the capability to support the last 4K, and if required ultimately, another near-doubling of the size of fixed memory. If the bank numbers were like telephone area codes, superbank numbers were like country codes. In the instruction architecture, we extended the time-sharing trick to cover all the op codes, allowing for 16 instructions with 3-bit op codes, and then noticed that not every instruction type needed a full-size address. Those that addressed only erasable needed only 10 bits for addresses, allowing the op code size to grow to 5 bits for those types; and those that addressed only Input/Output channels used 9-bit addresses and could have 6-bit op codes. That, with a total of 34 instruction types, was AGC Block II, and we couldn’t justify an effort to make yet another quantum leap within the Apollo program. It was quite marvelous enough to have been allowed two great leaps.

What Next?

Nobody used the term “Block III” in those days, even though we nosed around looking for a good place to put one, like the Space Shuttle for instance, or a space station. But the for-profit mainstream of the computer industry had by then acquired the technology and the political strength to nudge MIT out of the computer-inventing business.

Still . . . what would it have been like to create an AGC Block III? Something with just a little bit of upgraded technology perhaps, but with enough greater ingenuity to make the programming a lot easier and the running speed about three times as fast. In retirement, I’ve found that question nibbling at the parts of my brain where nearly all the details of AGC architecture will live for as long as my synapses connect one neuron to others.

Word Length Too Short for Instructions

The effect of inadequate word length also increased complexities of programming in the interpretive language we developed to allow programmers to think in vector terms, such as vector dot and cross products, matrix multiplied or divided by vector, etc., plus such features as a pushdown list for double precision or vector data, and two index registers. The software interpreter provided for instructions of 7-bit operation codes and 14-bit addresses in separate physical words, again not quite comfortable with the size of fixed memory.

Top priority goes to relieving the excessive pressure on the AGC’s native instruction word format, giving more bits per word to make the programming a little less “warped.” Since the packaging of AGC hardware was organized by 4-bit groups, it makes sense to consider lengthening the fixed-memory words to 20 bits including parity. Then, to preserve the von Neumann architectural mandate, it would seem appropriate to make erasable memory the same size, but that would increase a lot of logic by 25% without answering any need of the data. Accordingly, I moved further in the direction of Harvard architecture, with fixed memory words 4 bits longer than erasable memory words. Most of the Central Processing Unit (CPU) logic handles data, and that can continue to be packaged as four blocks handling 4 bits each. The only part of CPU logic that would have to think in a 20-bit length is the interface with fixed memory. To appease Johnny’s ghost one way, it isn’t too hard to find ways to get 15-bit constants out of fixed memory. For the other way, I use the fact that running code out of erasable is rarely done, only in emergencies really, and doesn’t have to run at anywhere near the same speed. To achieve that option, I invented a second kind of interpreter program I call a Native-mode Interpreter (NI). More about that later.

Exploiting the Harvard Architecture

Given a true Harvard architecture, what’s the next low-hanging fruit? Since most instructions make two references to memory, one for data and one for fetching the next instruction, this architecture allows them to run at the same time except for a few special cases. That’s one doubling of speed. Another comes from my assumption that both memories can be assumed to run at twice the speed of the Block II memories, that is, with a 6-microsecond Memory Cycle Time (MCT) instead of 12 microseconds.

The second biggest problem in AGC experience was the somewhat desperate time-sharing of erasable memory locations between programs that had to keep from stepping on each other’s data. Doubling the erasable from 2048 to 4096 words should be enough, especially if we can operate them without any bank-switching. Then: do we still need 36K words of fixed memory, for a total of 40K? Since the words are longer and each instruction word can therefore get more work done, we may well find that 32K words are enough, and with 15-bit addresses, those can be accessed without bank-switching. That makes a total of 36K words, compared to a total of 38K in Block II, but the addressing scheme can be simpler because of the Harvard architecture. The 32K words of fixed can be addressed as 00000-77777 octal (i.e. in 15 bits), and the 4K words of erasable can be addressed as 0000-7777 octal (i.e. in 12 bits). The apparent overlap of addresses is not a real problem because each instruction’s architecture causes any reference to memory to be dedicated to one memory or the other. If it turns out that 32K of fixed isn’t enough, bank-switching can be re-introduced, allowing addresses 00000-57777 octal to be fixed-fixed, and using addresses 60000-77777 octal for any number of 8K-word fixed banks.

Allocating Blocks of Erasable Memory to Jobs

The AGC’s celebrated priority-driven executive raises its own kind of time-sharing issue regarding the relative small erasable areas allocated to different jobs at different times. Every active job requires a “CoreSet” of a dozen words to maintain the job’s state after it yields the machine to a higher-priority job, until it is again the top priority. Any active job that includes interpretive code requires in addition a “VAC area” of a few dozen words to maintain the state of its pushdown list of temporary vector and scalar results, and other parameters used by interpretive code. As any scholar of Apollo 11’s infamous program alarms knows, the Block II executive provided eleven CoreSets of 10 words each and five VAC areas of 43 words each. Every reference to a VAC area had to use the address modification feature to direct the referencing to that job’s allocated VAC area, which made such code bigger and slower than if it hadn’t needed such modification. To avoid a similar problem with CoreSets, there was a permanent additional “current” CoreSet area to which the appropriate job’s CoreSet was copied whenever that job woke up, and from which the data was copied to the job’s CoreSet when it was being put to sleep. An efficient way to avoid both problems is to use a smaller version of the bank-switching logic. Erasable memory locations 7400-7777 octal are reserved for sixteen CoreSets of 16 words each, always referenced using addresses 7400-7437 octal, which invokes the use of a 4-bit CS register to determine which CoreSet is being used. Also, erasable locations 0000-0777 octal are reserved for eight VAC areas of 64 words each, always referenced using addresses 0000-0077 octal, which invokes the use of a 3-bit V register to determine which VAC area is being used. There is still a kind of address modification going on in these situations, but (once set up by the executive loading the CS and V registers) it’s done by the hardware at no cost in running time or program space to the jobs. Erasable memory addresses 1000-7377 octal, a total word count of 3328, have unambiguous assignments to their corresponding memory locations.

Notation for Negative Numbers

Another recurring concern with the design of Blocks I and II was the use of ones complement for negation, despite the fact that input data representing angles tends to be in twos complement. That had been chosen so that numerical and Boolean negation would be done by the same op code, a saving of much more significance in AGC3 and AGC4/Block I than in Block II. With more op codes available in Block III, I have standardized on twos complement for numerical negation.

Instruction Architecture for Block III

With all that in mind, what should the 19-bit instruction formats look like?

There are just two purposes for 15-bit fields: fixed-memory addresses used only by transfer-of-control instructions, and data constants to be supplied to the program. That leaves 4 bits for operation codes, but of those, one bit must be reserved to distinguish this class of instruction from all others. And in fact, eight op codes of this type are enough if condition codes are used to capture the state of the register last used. Four op codes are conditional and unconditional transfers of control of the ordinary sort, using a three-state condition code (positive, zero, negative) to direct the conditional ones. Two are devoted to generating 15-bit immediate data, one to the Accumulator (A) directly, and one to any instruction that has an interesting use for it. The last two ops are a type called “Execute,” for which the transfer of control is valid for only one or two instructions, after which control reverts to the instruction after the Execute. Many architectures include an Execute that does one instruction, but because we want ready access to double-precision constants in the Harvard-architecture environment, I have provided a “DEXEC’ instruction that can execute two of those immediate-data instructions, thereby placing a two-word constant into A and the Lower accumulator (L). It has other uses, but that was the driver.

The large class of instructions that refer to erasable memory use 12-bit addresses, leaving 7 bits for non-address purposes. Of those 7 bits, one bit has already been dedicated to distinguishing this class from the 15-bit instruction class, leaving 6 bits. Without room to allocate 2 or 3 bits to identify index registers, I chose to devote one bit to signify indirect addressing, leaving 5 bits to distinguish up to 32 instruction types in this class. That is ample, and allows for a few special cases that don’t use a 12-bit erasable-memory address. Shifting in the double-precision accumulator (A,L) needs only 5 bits as a count. Input/Output (I/O) can refer to any of 512 channel registers with addresses 000-777 octal in 9 bits. That leaves room for up to 512 operations that don’t use any sort of address field; some of that logical space can serve instructions that are not implemented in the hardware at all, by pointing to an array of routines in fixed that emulate them.

Address Modification at Runtime

As in Block II, a special instruction type allows any location in Erasable to serve as an index register. For Block III, I’m standardizing on one of the names used there: NDX. It has the same general functionality: fetching a 15-bit word from Erasable, adding it to the next instruction, and executing the result as the next instruction. Two new refinements:

One is to limit the use of the result to just the length of address field appropriate to the instruction, ignoring any overflow out of that field. Thus, NDXing an immediate-data or transfer-of-control instruction (15-bit address) will use the 15-bit sum to replace the raw address, but will not affect the op code. NDXing an Erasable-referencing instruction (12-bit address) will use only the rightmost 12 bits of the sum, and will not affect the op code. NDXing a shift or Input/Output instruction (9-bit address) will use only the rightmost 9 bits of the sum, and will not affect the op code, though in shift instructions it may change a nominally left shift into a right shift or vice versa. NDXing a no-address instruction will have no effect whatever, and presumably will never be done.

The other refinement involves the intersection between NDXing and indirect addressing (available only in 12-bit-address instructions). Since I’d expect indirect addresses that point to tables to occur more often than tables of indirect addresses each pointing to only one thing, NDXing of an instruction with indirect addressing does not affect the indirect address itself. Instead, it holds its argument until the Erasable memory cycle in which the direct address is fetched, and adds the argument to that, then trimming the sum to 12 bits since the direct address must refer to Erasable. If you want a name for that style of addressing, you could call it “indirect indexed.”

The opposite mode, that might be called “indexed indirect,” can be achieved by NDXing an NDX instruction which in turn affects an instruction with zero in the address field.

Call-Return Linkage for Subroutines

A particular concern about sharing erasable memory between jobs involves return addresses. If Job A calls subroutine X which in turn calls subroutine Z, and Job B calls subroutine Y which also calls subroutine Z, in which word W should Z keep the return address it must use when done, to make sure that W’s use by B doesn’t step on W’s use in A? Many architectures use a pushdown list for return addresses, and enforce on the programmers a degree of compliance with principles of structured programming, which is often described by an oversimplification: “Always CALL, never GOTO.” I’ve specified that there’s a separate such list for each job, resident in its CoreSet, to cover cases where a job has to hand the machine over to a higher-priority job and will get it back later. Any program interrupt operates like a CALL, and uses the return-address list of the current job to push and pop its own return addresses, and by structured rules always pops that list back to its status quo prior to the interrupt.

As a practical matter, non-calling go-tos have to appear in certain kinds of places in the code. To keep the required number of transfer-of-control instruction types within limits, such instructions in Block III do not determine what is a call and what is a go-to. Instead, that distinction is made within the code being given control. If the first instruction called is Push [previous] Return Address (PRA), that is the beginning of a subroutine. If it’s anything else, the return-address data is discarded and the transfer is treated as a go-to.

Program and Single-Cycle Interrupts

One of the complexities of AGC software testing was the fact that program interrupts could occur anywhere except where inhibited by an INHINT instruction which remained in effect until a subsequent RELINT instruction. In the Space Shuttle’s Flight Computer Operating System (FCOS), we found a way to “quarantine” all the hazardous effects of such interrupts, so that they couldn’t do much of anything until the fault tolerance logic was ready for them. That required the mission software to perform explicit permissions for those interrupt “consequences” at intervals not to exceed some standard, which is very straightforward to verify in testing. Such a rule wouldn’t work well in a development environment, where unruly code gets into loops or other mischief, but the Shuttle, like Apollo, was an operational environment whose software has been thoroughly sanitized.

For Block III, I have defined an instruction called RUPT for programmers to insert in their code at suitable intervals. When no interrupts are waiting, it occupies its one memory cycle and goes on. But if one or more are waiting, it gives control to the highest-priority one, exactly like a subroutine call except for making it return to the RUPT instruction when done. That makes all waiting interrupts get done in a batch, in order of decreasing priority. As in Block II, each interrupt performs a very brief “task” which may or may not change the priority of a job, but the actual handover from one job to another is held off until the current job comes to a higher-level explicit permission. Because the occurrence of interrupts is limited to known “clean” points in the logic flow, very little of the CPU’s state has to be saved and restored, saving some time in the interrupt routines.

Single-cycle interrupts, like those in Block II, commandeer individual cycles of erasable memory, usually to increment or decrement an interface variable. But since this architecture allows fixed and erasable cycles to run simultaneously, it’s beneficial to make these erasable cycles occur in memory cycle times where no erasable action normally takes place, as in branches, immediate data, etc. That way, they get their erasable-memory cycles without slowing the program down at all. My design provides fallback logic to let them occur between instructions “before too long” when there are too many consecutive cycles of erasable use for the program.

High-Level Interpreter

Block II’s interpretive language simplified life for the many programmers who were primarily engineers trained in disciplines other than computer science—up to a point. It let them think and write in terms of double-precision scalars and of 3-axis vectors and 9-element matrices in which all elements were in double precision. However, the way in which interpretive op codes and addresses had to be squeezed into 15-bit words made them acquire proficiency in forming code into a non-intuitive variation on the “Polish-prefix” notation invented in 1924 to use data-pushdown lists for temporary results. The oddest thing was that the 7-bit op codes, packed two to a word, were not all immediately adjacent to their respective address words and, in reading the code, you had to keep straight which address went with which op.

Seeking to make the Block III interpreter follow the spirit of the Block II while taking advantage of the longer words of fixed, I found myself in a similar predicament. There is the same need to reserve the leftmost bit to distinguish op code words from address words, leaving 18 bits. All addresses have to reserve 2 bits for index register access, and most have to use another bit to indicate indirect addressing. There are more details, but the point is that packing op code and address into a single 19-bit word is no more feasible, for interpretive-level instructions, than before. However, it is now possible to pack three op codes into a word with a bit left over to distinguish it as an op word. Each op is 6 bits instead of the 7 of Block II, but they don’t have to take part in signaling index register use, and a very small minority are augmented by another 6-bit group as a parameter. The extreme case is a family of flagbit operations, which use two parameters to perform any of sixteen operations on any one of sixty global flagbits. The first 6-bit group is the op code, just distinguishing flagbit operations from all others. The second group is the number (1-60) of the flagbit to be used. The third identifies the operation to be performed, and since most of those are branches based on the incoming value of the flagbit, an address word is needed as well.

All that clearly addresses one big driver of interpreter design: minimizing memory space used for code. Equally clearly, it doesn’t help a guidance-and-navigation engineer read the code, and in fact complicates it seriously relative to Block II. My solution is to let the programmers see a much more conventional-looking program in source-code format, and to shift the complications to the assembler software, which winds up being about a third of the way from assembler to compiler. Reading this source code, you’d think you were looking at assembly code for a von Neumann-type machine that has all the bits it needs for an op code, on the left, and an address, on the right—and furthermore supports all the functionality of a pushdown list for data, whether vector or scalar. Some op codes will lack addresses, like PUSH. Any op code that calls for fetching an operand from memory will be allowed to pop it off the pushdown list if desired, and in such cases the address in source code is just a minus sign representing a short dash. Operands that are constants are usable in all such cases. Subroutine linkage at the interpreter level is quite similar to the native level, including a separate pushdown stack for return addresses (two words each, here) and a PRA op to push each one down, converting a go-to retroactively into a call. Looking at an assembler output listing, you’ll see how each line of source code has been made into as little as a single 6-bit op code, or as much as three 6-bit codes and an address word or even two, spread over as many as four object code words. And all packed in as tight as logic will permit.

There is no thought of providing a way for interpretive code to be run out of erasable, partly because nobody in Block II programming ever requested such a thing. However, the same cannot be said for occasionally running native instructions out of erasable.

Native-mode Interpreter (NI)

I felt it necessary to compensate for the major disadvantage of the Harvard architecture by providing a way, in emergency situations, to run native instructions out of erasable. The method depends on a software program, probably resident in ground control systems when the application is space flight or even aviation. What it must do is translate each 19-bit object code word into a 2-word set including a brief op code in one 15-bit word and an address in the other, to place into erasable memory. Then the software installed in the Block III machine must include a “Native Interpreter” (NI), which picks up these fragments and dispatches them to be executed by the relevant native instructions, but without ever giving up control. I’ve written enough of the NI to feel confident about it.

CPUArchitecture

Block III has more central registers than Block II, but a design goal is to implement it in no more than twice the count of gate-equivalents. Unlike Block II, there aren’t separate packages containing only two 3-input gates, and I can’t guess what upgrades to that line of integrated circuits might then have been available. The logic packages usable for such hardware today contain Field Programmable Gate Arrays (FPGAs) containing large numbers of general-purpose logic units. Like Block II, none of the central registers have parity checking, allowing the 16-bit data word length to provide a duplicated sign bit for graceful handling of temporary overflow conditions. The following list is in alphabetical order for easy reference, showing width in bits, and providing cross-links as required.

A/16 – Accumulator for single precision and upper half of double precision [see L]

AL/32 – Not a distinct register: just A and L considered together for shifting etc.

B/16 – Buffer for multiple internal purposes [see C]

C/16 – Not a distinct register: just the ones-complement outputs of B

CO/1 – Carry out from 16-bit adder sum [see U]

CS/4 – Select one of 16 CoreSets as active; addressed as a channel [see V]

D/15 – Short-term register for immediate data and other internal purposes

DACT/1 – Discrete to signal that D has been set up but not yet used

DEXD/1 – Discrete to signal that D is specially active for a DEXEC sequence

DRA/1 – Discrete to signal that D contains a return address (used only by PRA)

E/16 – Memory local register for Erasable; parity processing both directions [see SE]

F/20 – Memory local register for Fixed; parity check when filled from memory [see SF]

IR/14 – Interrupt Requests, a prioritized array of discretes; addressed as a channel

L/16 – Lower accumulator for double precision; lower half of product (MP and MPAD),
 lower half of dividend, then remainder (DV). Mostly unsigned, has rounding bit

Q/16 – Quick-turnover register for short-term data; word counter in copy-blocks

RA/15 – latest Return Address, for instant return from subroutine [see RAPO/RAPU]

RAPO/5 – Return-Address POP pointer; addressed as a channel [see RAPU, RA]

RAPU/5 – Return-Address PUSH pointer [see RAPO, RA]

SC/9 – Select Channel: holds address to identify active channel [no local register req’d]

SE/12 – Select Erasable: holds address to drive RAM [see E]

SF/15 – Select Fixed: holds address to drive ROM; incremented content readable [see F]

U/17 – Not a distinct register: just the adder output; 17th bit is CO [see X, Y]

V/3 – Select one of 8 VAC areas as active; addressed as a channel [see CS]

X/16 – First latch for augend as input to adder [see U, Y]

Y/16 – Second latch for addend as input to adder [see U, X]

Z/15 – Instruction location counter, ROM address of (usually) next instruction