Notes comments M. Hamilton

JAN 27 1970

AGC Features Leading to Increased Software Costs

Copies to
P. Adler
D. Densmore
D. Eyles
D. Millard
P. Mimno
A. Hopkins
A. Green
B. Ellison

Inte[illegible] been expressed in features of future airborne computers, in order to take advantage of lessons learned from the Apollo CSM and LM PGNCS software activities. Before becoming encumbered with the present plans for such future computers, such as those for the "space shuttle", a list of desirable design features for such a device has been assembled below. Lacking any details of how the computer will fit into the spacecraft design, and indeed what the system design will require (page 17 of Aviation Week for 19 January 1970 indicates that OMSF is to issue RFPs on 21 February for this purpose), the general Apollo-class of activity has been assumed.

No attempt is made to assign functions to the computer hardware as opposed to supporting software, since these decisions require knowledge of relative costs of each approach (the AGC analogy would be a requirement for "double precision vector cross products", which of course is in the software of the interpretive language, although a programmer can essentially treat the VXV operation as a "hardware" capability).

From the viewpoint of Apollo background, design characteristics of a future airborne computer should include the following (in roughly decreasing priority):

1. Ample erasable memory

One of the major causes of software difficulty in Apollo has been a limited amount of erasable memory. This has caused difficulties in a variety of areas (even such situations as use of R31 with MIDFLAG set), and is probably a major obstacle to the realization of the full advantages of higher-level program languages. Manual assignment of variables to particular cells, and modifications, requires a vast amount of knowledge as to the design of the complete program, hence a high skill level and considerable experience on the part of the people involved.

This is a good example of the hardware/software conflicts that can arise, since erasable memory requires more volume, more power, and imposes heat dissipation problems considerably in excess of those for fixed memory. Thermal operating regions can also be stricter, and reliability assessments would favor a given amount of fixed memory over a somewhat lesser amount of erasable memory. One possibility would be to have about half the memory be fixed, and the other half optionally be used for variables or program steps (this is done in the AGS from a packaging, heat, and power consumption viewpoint, and hasn't been too drastic a constraint, in spite of the "very fixed" nature of "fixed" memory).

2. Floating point capability

Although the number of post-release problems in Apollo software due to scaling difficulties has not been large, the amount of manual analysis and checking that must be done to insure that scaling problems are avoided has been considerable (and for unanticipated problems, overflow can cause difficulties that might be avoided if floating point were used). As observed by F. Clark of TRW when discussing future computers, "I wonder if they will decide that floating point is here to stay yet."

Floating point is another good example of hardware/software conflict, since clearly the floating point hardware is more elaborate than fixed point, and since clearly software "can" be written to achieve the same effect. One

case in Apollo where floating point would have been useful was for measurement incorporation of the rendezvous radar rate data. Program performance study led to the inclusion of a "quasi-floating point" normalization in this area, with a subsequent improvement in software performance. In general, floating point does not "improve accuracy", however, since the bits consumed to specify the location of the "binary point" could be used to good advantage to provide additional precision for the operand. Next to erasable memory, determination by some automatic means of proper scaling for quantities is probably the major obstacle to the use of higher-level program languages. Automatic schemes can be dreamed up, of course, but whether they always compute (A + B - C - D) in the proper order, for example, to avoid overflow isn't so clear, especially if B and C are always negative and A and D positive: should not rearrange terms.

3. Sufficient fixed memory

A limit on fixed memory is, from the point of view of a software manager, a very desirable thing, since this puts an upper bound on the amount of work that has to be done, and helps drive home to his customer that the "computer can do anything but not everything". Sizing of fixed memory is undoubtedly a rough job, and applications for fixed memory probably will grow until the memory is filled (exclusion of P37 "inside the sphere" in the CSM is a good example of something nice to have for Apollo that we don't have because of memory problems). Probably the best arrangement is some sort of ability to add additional memory modules as new requirements become identified: to some extent, this capability is in the AGC due to the unused values of SUPERBNK, but this approach restricts somewhat the "random" access capability, thus relegating relatively independent functions (such as Pinball) to the orphan banks of memory. In the picturesque vocabulary of one MSC individual, the "software becomes the garbage can for all the hardware's problems", and this tendency presumably will be true in the future.

4. Moderate execution time

For the sake of preflight validation (and such applications as crew trainers), the execution time capabilities of the computer should not be "spectacular": a factor of no more than 6, for example, improvement over Apollo (giving an add time of 4 microseconds or more) should be tolerated. This may allow serial as opposed to parallel arithmetic (saving hardware), and in any case reduces circuit speeds with a gain in reliability ("ruggedness") as a hopeful byproduct. The software cost reduction here is similar to that gained under item 3 by "sufficient" (as opposed to "ample") fixed memory: by reducing the amount of activity that needs to be going on in the computer, more attention can be paid to that which remains, and proper decisions can be made on what is really needed vs. what is done because "it doesn't cost anything".

5. Restrained interrupt structure

Arrival of interrupts at unexpected times has caused AGC software problems in the past, such as the "ghost" R36 at ignition and some unbuffered downlink information for radar data in the LM. In addition, of course, there was the Apollo 11 counter interrupt "overload", leading to renewed emphasis on just how much of a TLOSS safety factor should be reflected in the onboard software.

This area too is a hardware/software conflict, but of a slightly different type than some of the others. Here, software in some sense is made "easy"

by several different interrupts (four based on time, for example, whereas one, except perhaps for 10 ms phasing and least increment considerations, might have been enough), but possible "sneak paths" are compounded too. It seems mandatory that some sort of synchronism to real time (other than by equalization of computing delays for different program branches) be available similar to the "waitlist" capability in the AGC now. Whether this is some periodic event such as the AGS 20 ms interrupt or a more elaborate variable delay as in the AGC now is a hardware option. Telemetry and manual inputs could be keyed off of this interrupt(as is done now in AGC for the LM thruster fail inputs, for example).

If counter interrupts are continued, a method should be available to have the software disable them (as seems to be provided, although not used, for the uplink by bit 6 of channel 13). A nagging problem in software validation is that cannot predict before the flight the exact sequence in which the program computations will be executed during flight (i.e. just when the display interface routines will be entered with respect to when a PRO response to flash for engine ignition was supplied), and counter interrupts are certainly a contributor to this uncertainty. Due to the hardware savings which they can achieve, however, perhaps their continued use is justified.

6. Flexible input/output capability

As part, perhaps, of the "garbage can syndrome", it is natural that additional interfaces for the computer will become identified as the flight program progresses. One interface that was handled easily was the PRO button (which in an "original" design might have been another key code, but for ease in retro-fit was made a contact closure), and another one, not handled so nicely, was the addition of a rate display capability on the FDAI in the LM: it would have been desirable to drive the separate rate needles directly, so that both rate and attitude error displays could be provided. A degree of flexibility in the I/O would also help to dispose of space surplus computers for applications in which their fixed memory is not a disadvantage.

One possibility for doing this would be a separately packaged I/O unit, connected to the computer "main frame" by a restricted number of lines. If this approach had been in use in Apollo, the channel 13 "stall" routine, which can rob as much as 5 ms of execution time, perhaps could have been implemented more easily in the hardware (although this particular problem, of course, in retrospect should have been handled by an interconnection spec).

7. Input/output override capability

A hardware capability that might simplify the software job is an input/output "override" capability. In some cases this has been handled by software design changes (such as deletion of checks of the stage verify bit); in other cases, the computer program is vulnerable to failures of the interface (such as the stuck mark button before H2 over-riding the ROD input,or a improper RR Auto mode input keeping the Apollo 11 descent problem from being avoided). These items are perhaps special cases of the general philosophy of "give the user/ programmer control over the machine if he wants to use it."

8. Do some software design before freezing the hardware

Evidence that this was done for the Apollo computers is one of the most impressive features of the PGNCS design. The EDOP register, for example, which shifts bits 14-8 to bits 7-1, probably was included solely for use with the interpretive language, where it saves execution time over performance of a similar function by order code means, although it also has been handy since the transition to decimal verbs and nouns. Whether a special purpose register such as this is included in NACA (Next Airborne Computer Attributes) hardware or not isn't a question that can be decided now: the philosophy that led to having this register in the AGC design, however, is one that must be encouraged. The alternative of a jazzy hardware design regardless of the software headaches (such as the Air Force Titan II computer that saved a diode for a price of hellish divide) cannot be allowed to prevail.

9. Consider the problem of in-hardware program debugging

Regardless of the general purpose computer simulations that may be made, problems will probably still be found only after the program is running on the actual hardware. These have proven at times to be extremely difficult to track down (such as a peculiar Executive System difficulty in an early LM-1 program release or, perhaps, one of the R36 "ghost" appearances in more recent programs). For the AGC, a special hardware unit ("Project Trace" in early 1968) perhaps has been obtained, but perhaps for future systems a penalty in software proficiency should be paid for the sake of easier debugging later (whatever happened to that V53 flash in the CMS entry simulations in late October 1969, by the way?).

10. Avoid excessive software burden due to restarts

If restarts are felt to be a consideration for NACA, then the burden of recovery from them should be removed from the software. This item probably belongs earlier on the priority list than here, in view of the amount of special programming effort that has to be spent (even for such "elementary" actions as decrementing velocity-to-be-gained by accelerometer outputs) in allowing for restarts and recovering from them, the difficulty of systematic testing, and the number of post-release difficulties that have been discovered. One approach might be to forget about them, since proper programming to handle them seems another area making use of higher-level languages less beneficial than might otherwise be the case.

11. Other items

As merely a sort of checklist, the following items should also be considered in evolving the design of a future airborne computer:

a) Crew training techniques

b) Crew interface/override requirements

c) Mission-critical computer actions (e.g. burns or aborts)

d) Software management procedures ("sealing", for example)

e) Software validation procedures

f) Software documentation

g) Programming language to be used (or imposed)