

BACK TO THE MOON: THE VERIFICATION OF A SMALL MICROPROCESSOR'S LOGIC DESIGN

Hugh Blair-Smith, Richard Katz, and Igor Kleyner,

NASA Office of Logic Design, Goddard Space Flight Center, Greenbelt, MD

Abstract

The original and primary task of self-test program Smalley3 was independent verification of the logic design of the LOLA DU (Lunar Orbiter Laser Altimeter Digital Unit) microprocessor. Tasks were added to verify continuing correct operation of this central processing unit (CPU) under margin testing for supply voltage, ambient temperature, and clock frequency. Finally, an on-orbit diagnostic task was added so that any malfunctions of LOLA in lunar orbit can be identified as faults in, or not in, the CPU.

The Lunar Reconnaissance Orbiter spacecraft will be launched to the Moon in 2009 with six scientific instruments including LOLA, each containing an embedded microprocessor to perform real-time subsystem control calculations. LOLA's CPU is a small, custom-designed processor, designed to meet the mission requirements while minimizing resources. This 8-bit machine is essentially code compatible with Intel's 8085 but is implemented in modern technology, an advanced, radiation-hardened 0.15 μm gate array, with the only logic element types being a 4:1 multiplexor and a flip-flop.

This paper explains the fundamental structure of the verification task, shows how particular instructions are verified, presents a high-coverage scheme for detecting inadvertent RAM alteration, describes subsystem testing of RAM, and reviews the results of the verification effort. Some infamous CPU design flaws from both the commercial industry and aerospace flight control systems are discussed.

Lunar Orbiter Laser Altimeter

The Lunar Reconnaissance Orbiter (LRO), to be launched early in 2009, will carry six scientific instruments including the Lunar Orbiter Laser

Altimeter (LOLA). Laser altimetry will produce very detailed and precise geodetic maps (why not "selenodetic?") to support various purposes, such as where to drill for ice, good places to build a permanent base, etc. Similarly to the other instruments, LOLA's copious data stream is controlled by a Digital Unit (DU), whose CPU must be radiation-hardened, nimble though not a heavy-duty number cruncher, and above all reliable.

NASA's Office of Logic Design (OLD) has designed the "80k85" which fits into a compact corner of an Actel Field Programmable Gate Array (FPGA) to be this CPU. As the name suggests, it is logically a near-clone of Intel's 8085 microprocessor, upgraded significantly in speed and general reliability, with solid radiation hardening. Development complexity and costs were reduced by adopting an instruction repertoire of proven suitability for embedded control: not a RISC architecture, but no prodigal consumer of gates by FPGA standards. This approach also leverages existing software tools and the skill sets of their users. Software development reliability is enhanced by adding an internal interrupt trap for unimplemented operation codes.

Processor Design Verification Issues

Unlike software, the logic design of a processor (or microprocessor) is broken down into a large number of small simple design tasks that interact in a small number of relatively simple ways. It would be hard to imagine a design flaw in, say, instruction implementation, that would cause the kind of few-times-in-a-trillion hangups that occasionally freeze our desktops and laptops.

The catch is the large number of these tasks; somebody's going to nod off somewhere in the long, tedious, and repetitious path through them, and there had better be something independent, rigorous, and thorough to spot the problems. IBM and Intel (though the giants of the universe) have been caught napping, as we shall see. The other

catch is there are dimensions of design other than logical, and we'll examine an electrical design issue in a spacecraft processor that might have been spotted by such an independent-rigorous-thorough testing tool.

IBM System 4π AP-101 Long Divide, 1987

When NASA sought a more-or-less-COTS computer to be the Space Shuttle's General Purpose Computer (GPC), IBM Federal Systems Division bid a new "Advanced Processor" variation on their existing System 4π architecture which had seen service in aircraft and spacecraft. A critical requirement was that the GPC implement all the System 360 instructions with exactly the same results, to eliminate any discrepancies between the GPC's calculations and those of the 360 Model 75 mainframes in Mission Control at JSC.

In pursuit of this goal, IBM's design for the AP-101 included a Divide Exponential Double (DED) to handle floating-point numbers with a mantissa of 14 hexadecimal digits. To ensure that interrupts would be taken quickly, they made DED (and its register-access twin DEDR) interruptible within itself: it could be interrupted after the development of any quotient digit. When it was pointed out that this DED design was so slow that a subroutine using the single-precision DE could beat it [1], IBM redesigned DED to run without interruption. Perhaps because of the short time frame for this redesign (AP-101B), it was straightforward, simple, and correct.

For deployment in 1990, IBM was required to re-implement the AP-101 in more up-to-date technology, resulting in the AP-101S. Part of the upgrade was a significantly more elegant algorithm for DED and DEDR, which worked for most inputs but produced wrong answers for certain cases where the low-order part of the divisor was not zero [2]. We haven't found any documentation of what verification was performed and why it wasn't sufficient, but the discrepancy reports noted that it was "difficult to define" which inputs produced the wrong results.

Curiously, the HAL/S compiler developers seem to have been aware of a problem much earlier than the date on the discrepancy reports, because they didn't use DED at all, or DEDR for anything

but the remainder function DMOD, which uses short divisors (understandably), thus avoiding the problem. At least, that's the observation made by a code audit triggered by the discrepancy. A subroutine I2DEDR was substituted for DEDR, just in case there were more error cases than had been found, and the compiler was updated accordingly.

Intel Pentium P6 Core Long Divide, 1994

This is the famous Pentium bug [3, 4] that was discovered by a number-theory application developer. Intel, facing the fact that increases in processing speed aren't aided by an equivalent to Moore's Law for component density, had made a major effort to speed up double-precision divide in the P6 core, and it involved several tables in ROM. The design was correct as far as we know, but the implementation suffered from a failure to include in the ROM a significant part of one of the tables.

It's easy to see now that a thorough low-tech proofreading of the ROM tables would have prevented this embarrassment. What's interesting is that the kind of tool that is the subject of this paper has only a pretty good chance of detecting such a fault, because not all combinations of high-precision inputs can be tested, and some kind of sampling is the only feasible approach. For two 64-bit inputs, the number of combinations is 2^{128} , if each combination takes one microsecond to test, the test will run for 2^{108} seconds or about 2^{92} days or about 2^{84} years—compared with the age of the universe, about 2^{34} years! We don't know if Intel's verification suite contained the same sort of combined systematic and Monte Carlo inputs as Smalley3, but we can imagine that it might still omit the cases that depend on that table.

Having said that, we can criticize Intel for not having constructed a verification test designed to exercise every number in all those tables.

Sandia/JPL 1802 Register Interaction, 1981-6

In the microprocessor used in Galileo, a fabrication upgrade from a 2-inch wafer to a 4-inch wafer included process changes to improve the speed of a 16x16 register array. Unfortunately, this allowed a number of analog-type issues like off-nominal supply voltage and pulse delays in polysilicon lines, combined with a heavy population of

ones in certain registers, to create a digital logic fault that copied certain bits of one register to another register. JPL documentation declares that understanding the exact physical mechanism was extremely challenging, but in 1987 they were able to identify the conditions for the fault and construct a screening test to select the more robust units. They also introduced a software restriction to avoid the conditions [5].

For our 80k85 microprocessor, the approach taken by *Smalley3* seems to address this kind of problem, especially when used for testing of voltage, temperature, and clock rate margins. In fact, it found a somewhat similar fault, a stretching of fan-in/fan-out rules, which was then fixed as a logic design error.

The 80k85 Verification Challenge

The legacy of these horror stories is a top-level requirement that 80k85 verification be rigorous and thorough, and as close to exhaustive as is feasible. The architecture, though generally 8-bit, is 16-bit in places, particularly the instruction DAD (Double-precision register Add). An exhaustive test of all combinations of its two 16-bit inputs would comprise 2^{32} cases and take days to run. If the only purpose of this effort had been to create a one-time logic design verification test, that might have been acceptable, but the ability to run our verification repeatedly, varying margins or even just letting the pseudo-random number generator produce different Monte Carlo samples, was paramount.

Reaching way back for a successful model to follow, we noticed that both the Block I and Block II Apollo Guidance Computers (AGC) had self-check programs to augment the manual design verification process, and to assure at any time that all of the logic in a particular AGC was still working. That effort had the interesting side effect that an instruction EDRUPT (Ed Smalley's private interrupt) was added to the design to facilitate including interrupt logic in the test, but we didn't need to emulate that in the 80k85 development.

One mitigating factor in the verification challenge was that (again like the AGC) the 80k85 had been running, apparently successfully, for some time when the verification development began. The phased development plan for *Smalley3* was thereby

simplified, since there was no need to construct a rigid sequence of baby steps for the first few instructions tested. Instead, the phasing was simply a progression from instruction verification to the two levels of RAM testing.

While the RAM corruption detector is an integral part of the logic design verification process (to detect any unintended changes to RAM contents made by instructions), the rest of the RAM testing is really advanced burn-in and degradation detection of the RAM as a distinct subsystem.

8085/80k85 Architecture

This is an 8-bit machine that uses 16-bit addresses to access 64K bytes of RAM, either one at a time or a pair in little-endian fashion. There are 7 data registers of 8 bits:

- A (accumulator);
- B, C, D, E;
- H, L

of which, BC and DE occasionally function as 16-bit data registers, and HL frequently functions as a 16-bit indirect addressing register.

Other 16-bit addressing registers are:

- SP (stack pointer);
- PC (program counter).

There is a set of condition flags sometimes lumped as "register F," and the combination of F and A sometimes functions as a 16-bit PSW (program status word). The 8085 has an interrupt mask and a serial I/O (UART) port, which were not implemented in the 80k85.

For input/output purposes, 8-bit addresses are used to access 256 ports, which in the 80k85 are allocated 128 to input and 128 to output.

Overview of 8085/80k85 Instructions

The instruction repertoire is a rich set of simple functions suitable for an embedded controller/sequencer, hence its appeal for use in the LOLA DU. It's rich in data moving, condition-code-driven branching, Boolean, and arithmetic operations, but stops short of having multiply and divide operations—or any floating point.

Instructions are one to three bytes in length, with all operation coding within the first byte. The second byte of 2-byte instructions is immediate data or an I/O port address; the second and third bytes of 3-byte instructions are direct RAM addresses.

In principle, this implies 256 distinct operation codes, but 10 are unused and one (DAA, Decimal Adjust) is not implemented in the 80k85 because it is dedicated to decimal arithmetic. However, there are only 70 functionally distinct instructions, many with parametric variations such as 3-bit register addresses and 3-bit condition code tags. The extreme case is MOV, with only 2 bits of op code and register addresses of 3 bits for both source and destination. One of the 3-bit addresses specifies, not a register, but indirect access to a byte of RAM via the HL pair.

The 80k85 implements four of the five 8085 interrupts, but does not implement their priority scheme.

Functional Grouping of 80k85 Instructions

To allow design verification to focus on particular functional areas, or even test incomplete 80k85 units, ten independently testable functional groups of instructions were defined:

1. NOP and single-byte transfers;
2. Double-byte transfers;
3. Single-byte arithmetic binary operations;
4. Double-byte arithmetic binary operations;
5. Single-byte Boolean binary operations;
6. Assorted unary operations;
7. Transfers of control (except HLT);
8. Stack operations;
9. Data input & output operations;
10. Interrupt management & illegal op codes,

where “binary” and “unary” refer to the number of inputs to a given operation. Verification of HLT is obviously a very special case, but it is addressed.

Introduction to Smalley3

The primary purpose of *Smalley3* is to validate the fidelity of the 80k85’s behavior, when executing

Intel 8085 instructions and interrupts, to the “gold standard” of 8085 function exhibited by the Harris 80C85RH, except as noted herein. The 80k85 is an image, in the Actel RTAX-S FPGA, created by NASA’s Office of Logic Design (“OLD”) to allow 8085 code to run on a machine much faster and more radiation-hardened than Intel’s original chip. Intel’s chip, never radiation-hardened, was introduced in the mid-70s and has been out of production for years. The radiation-hardened Harris processor is also out of production. A secondary purpose is applicable after the logic design validation is complete; it exploits the fact that *Smalley3* is a self-test program running on the 80k85. *Smalley3* can be rerun as desired to verify the continued correct operation in laboratory stress testing and even in lunar orbit. To serve this second purpose, two tests are included that exercise the parts of the 64KB RAM not occupied by *Smalley3* code.

The lunar orbit environment, which has determined several programmatic factors in the requirements for and design of *Smalley3*, is the application of the 80k85 as the CPU for the LOLA DU (Lunar Orbiter Laser Altimeter Digital Unit) in the LRO (Lunar Reconnaissance Orbiter) to be launched in 2009.

Smalley3 is named in honor of Ed Smalley of MIT’s Instrumentation Laboratory (now the Charles Stark Draper Laboratory, Inc.), who in the 1960s developed somewhat similar self-test software for the Apollo Guidance Computer (“AGC”) at the heart of the Primary Guidance Navigation and Control System (“PGNCS”) used in both the Command Module and the Lunar Module of the Apollo spacecraft. Ed’s software, which we hereby retroactively name *Smalley1* for the Block I AGC and *Smalley2* for the Block II AGC, verified the function of all the AGC’s instructions to catch any faults caused by degradation of the hardware.

The bulk of *Smalley3* validates the logic design of the 80k85, that is, the “netlist” that determines how the FPGA’s multiplex gates and flip-flops are interconnected to functionally emulate an actual 8085. In order to catch sneak paths, fan-in and fan-out problems, and subtle cross-talk issues, its function goes well beyond simply checking that each instruction does what it is supposed to do, by verifying also that it has no side effects on the state

of the CPU or the I/O ports. *Smalley3* also verifies the absence of side effects that corrupt RAM locations which should be unaffected by the instruction under test; this is done for just 2 locations that may be closely related to what that instruction does, and at greater intervals for the entire 64KB SRAM. Finally, there are two tests of the approximately 55KB of SRAM not needed for *Smalley3* code. One, called “*Yozzle*” in memory of a Honeywell 800/1800 tape drive diagnostic from the ’60s, stresses the memory by forcing very frequent bit value reversals. The other applies the theory of pattern-sensitive faults (“*PSF*”) to catch any cross-talk between each bit of memory and the bits immediately adjacent in two dimensions.

In the development laboratory environment, the 80k85 operates under the control of Bench Check Equipment (“*BCE*”) which functions as a console. In a flight or all-up system test environment, all *BCE* commands are implemented through uplink.

In the LOLA flight configuration, the 64KB SRAM is overlaid from EEPROM pages; several of these contain copies of the operational flight program, and another contains the approximately 10KB *Smalley3* program. When it is the active overlay, *Smalley3* controls the entire 64KB address space of the 80k85.

Top-Level Design of *Smalley3*

A *Smalley3* run consists of one or more Test Cycles, either a specified number or indefinitely until a fault or a manual stop occurs. Because the pseudo-random seed is not re-initialized between Test Cycles, multiple cycles do generate different sequences of random data. The RAM subsystem tests (*Yozzle* and *PSF*) occur only at the end of an automatically halted run.

Each Test Cycle in a run tests a specified subset of the ten functional groups. Within each functional group, testing is performed on a specified subset of its functionally distinct operations. For each distinct operation, testing covers a specified subset of its parametric variations as defined by all 8 bits of the first byte (e.g., distinguishing MOV A,M from MOV D,H). For each such parametric variation, testing used 16 systematic data value sets and a specified number of pseudo-random data sets.

All this flexibility, which in the event has been lightly used, seems complex but is driven quite simply by tables for each level. The specifications mentioned above are performed mostly by patching the tables, and in some cases by initializing a few input ports.

The RAM corruption check can be run at any of these levels (as noted above), but in any case runs once at the end of each automatically halted run, whether or not a fault was detected. This supports a major goal of presenting a maximum of information for analysis whenever a fault stops a run.

Input operations are tested by periodic loads of the input ports generated by the *BCE* at intervals that are regular but look truly random to the 80k85 instruction flow. When read by the *IN* instruction during such a load, the value read may be old, new, or a mixture. Output operations depend on reading the values back from the relevant output port.

In accordance with the designed interrupt schedule in the spacecraft, the four types of interrupts are triggered by the *BCE* in a regular sequence, separated to prevent any problems from the lack of priority and maskability. This regular sequence, like that of input data, looks truly random from the point of view of instruction flow.

Design Considerations in *Smalley3*

Ideally, every instruction would be tested in every possible machine state, but realistically what’s possible is a considerable variety of central register and RAM states, comprising not only the registers and RAM locations involved in the instruction, but also the other registers. This variety is described below under *Systematic or Random Data Environment*.

Other principles governing instruction testing include watching as widely as is practicable for unintended side effects, minimizing the inherent “conflict of interest” when a computer tests itself, and presenting for analysis enough data to handle multiple errors arising from a single fault. These are described below under *Instruction Testing*.

Rigorous and thorough testing of input/output operations has many aspects that don’t figure in

other instruction types. These get a section of their own, *Input/Output Testing*.

Similarly, external interrupts are a major special case, described below under *Interrupt Testing*.

Detection of unintended corruption of RAM contents required some innovative developments described below under *RAM Corruption Detection*.

Finally, there is a section covering the *Yozzle* and *PSF* tests under *RAM Subsystem Testing*.

Systematic or Random Data Environment

The artificial machine-state data that varies the environment in which each instruction test runs is placed into all possible register pairs before the test, with the obvious exception of PC. There is also a restriction on the range of values placed into SP, to prevent stack operations or interrupts from stepping on Smalley3's code or scratch locations. This is a case where the flags and accumulator are treated as a pair, so that the arbitrary values are imposed on the condition codes as well as the ordinary registers.

Artificial machine-state data is also used for the address of a pair of bytes in RAM and for their initial contents. The address is in a restricted range of values to avoid stepping on the stack or on Smalley3's code or scratch locations. Furthermore, 3-byte instructions are made to use this address, and 2-byte instructions are made to use artificial data as either immediate data or an I/O port address, as appropriate.

Systematic Artificial Data Sets

The goal here is to include sixteen of the most "interesting and edgy" patterns of bits: heavy on the ones, heavy on the zeros, alternating ones & zeros, etc. For efficiency, each pattern is generated as a pair of bytes even though sometimes only half the pattern is used. Each systematic data pattern is placed in all the machine-state registers and the selected port and RAM locations, subject to the constraints on SP and the RAM address, so that whatever is interesting about it will apply to as much of the total machine state as possible. In systematic mode, Smalley3's Content Engine composes sixteen patterns of 16 bits from the following categories:

- All 16 bits the same;

- Alternating 8 zeros with 8 ones;
- 4 zeros, 4 ones, 4 zeros, 4 ones;
- Alternating pairs of zeros and ones;
- Alternating zero and one bits.

Pseudo-Random Artificial Data Sets

An 8-bit Linear Feedback Shift Register (LFSR) is implemented in Smalley3's code, with special-case logic to prevent 00000000 from being a lockup state, to cycle in a non-obvious sequence through all the 256 states of one byte. This function is called a Pseudo-Random Number Generator (PRNG), and is called twice by the Content Engine in random mode to produce each pseudo-random 16-bit pattern.

In contrast to the systematic mode, each register pair, RAM location, etc. gets a different PRNG value, applying required constraints as appropriate. The seed value is never reset; that's why repeated Test Cycles get different values for each particular instruction test, increasing the odds of finding the rarest and most obscure data pattern sensitivities.

Instruction Testing

Each instruction test must verify that the instruction does everything it is supposed to do, and nothing else. Any given instruction properly affects only a small part of the machine state (often, just one register), but its unintended side effects could affect any part of the total machine state. As we said before, we had to limit the scope in which unintended side effects are detected during the instruction test:

- All the central and special registers;
- The most relevant two bytes of RAM, which are the top 2 bytes of the stack when appropriate, or just two bytes arbitrarily chosen when none are relevant;
- The most relevant I/O port, or just one arbitrarily chosen when none is relevant.

Initialize, Predict, and Verify Machine State

Smalley3's scratch locations include three instances of the limited machine state, called PRE, POST, and FOUND. The PRE values are of course filled in from the artificial data sets described above. The POST values are initialized from the

PRE values, since any one instruction is supposed to affect only a little of even the limited state.

Then ad hoc logic for each parametric variation of each instruction type predicts what the instruction should do, and updates the POST values accordingly.

After the instruction is executed (under strict control to keep it from gaining control of the machine), the actual state of the machine goes into the FOUND values. Verification consists of comparing FOUND to POST values over the entire limited machine state.

Principles for Prediction

Ideally, each instruction's results would be predicted by code that includes no instances of the instruction under test. In a self-test program, this is naturally not possible, but a substantial step in this direction was nonetheless achieved in *Smalley3*. In several cases, the predicted results are obtained from tables entered with the input data as arguments. Specifically, there is a routine called *Blackadder* that predicts the results of all addition and subtraction operations using 256-byte tables whose addresses are aligned so that entering them is nothing but setting the L register (lower half of indirect address). The only way the 80k85's adder takes part in this process is the unavoidable one of incrementing PC.

Prediction of the results of Boolean operations loops through the 8 bit positions, shifting as required and using the condition codes to control branching, but includes no Boolean operations.

Verification and Analysis Support

Smalley3's scratch locations are laid out in a concentrated area of low memory so that a small memory dump will supply as much information as possible about what error or errors are induced by a fault. The PRE, POST, and FOUND instances described above are in locations aligned to facilitate manual comparison. There are also a number of variables called BADS that contain the exclusive OR of the FOUND and POST instances, to exhibit clearly which bits are wrong. These are arranged in a tree to facilitate navigating to the error bits: a master byte FBADS shows not only which flag bits are bad but uses non-flag bit positions to point to other BADS data. One of these is RBADS, which is simply a set of bits indicating which registers

contain discrepancies, thereby pointing in effect to whichever BADS value actually exhibits the discrepancies.

Special Note on "Testing" HLT

In a self-check program, there is no way to make HLT do anything but halt the machine, which means there's no hope of filling in the FOUND variables. However, the test engineer has some control over what the machine state is when HLT is used to bring a test run to an automatic end, because the same sort of setup of PRE and POST variables is done before such a HLT. Observation of the machine's actual state when halted can be compared against the POST values. Variations in the manual setup of the next run (initial random seed, number of random data sets per instruction, etc.) will force changes in the PRE-POST setup for the next run's halt. You can't control what the new values will be, but they will almost certainly be different.

Input/Output Testing

The customary way to test I/O functions is to write output ports and incorporate some kind of wraparound in the BCE to allow those values to be read back in and compared to the original outputs. In this project, the BCE couldn't do that, so it doesn't contribute to output testing at all.

The 80k85's 128 input ports can be initialized by the BCE, and are updated periodically with a mixture of systematic and pseudo-random data produced by the BCE. As noted above, the updating of a particular port can be caught in a partial state by an IN instruction, so software logic had to be introduced to discard half-baked data.

Systematic and Random Input Data

When just one or a limited number of Test Cycles are run, the BCE supplies only pseudo-random data, but in an open-ended test run, it supplies a few systematic data inputs first and then the pseudo-random data. The systematic data is all zeros into all ports followed by all ones into all ports, which is certainly a minimum set.

Late in the project we noticed that these would not allow detection of a wiring error that reversed two bit positions in some port(s). We identified 3 more systematic data sets that would provide this coverage but did not have time to incorporate them.

Verifying Input Data

Verifying the input of systematic data is straightforward because *Smalley3* knows a priori what the values should be. No such knowledge can be feasibly provided for random data, so we borrowed an old magnetic tape SEC-DED technique from the 1960's, Honeywell 800/1800's "Orthotronic Control®" in which each port's data obeys a parity rule and there is a longitudinal checksum for each bit position over the 128 input ports as a whole. This allows *Smalley3* to identify a single bad port, and even a single bad bit within it; if there are multiple errors, at least one bad port is identified. To gain a little extra coverage, the parity rule for each random data set is the reverse of the one used in the previous set.

Asynchronicity Issues in Testing Inputs

The replacement of one data set in the input ports by another, though regular and straightforward from the BCE's point of view, is a long asynchronous process from *Smalley3*'s point of view. Our approach is to assign the highest-numbered input port, 7F, as the longitudinal checksum and arbitrarily rule out 00000000 as a checksum value. The BCE, when ready to replace an input data set, first zeros port 7F as a signal that the input ports as a whole are not in a stable state. *Smalley3* samples port 7F often enough to see the nonzero-to-zero transition before reading any port that will be affected. Then it suspends reading until port 7F becomes nonzero again, which is the BCE's signal that the new update is complete.

Because the ports are updated in order of ascending port addresses, all of ports 00-7E are guaranteed to contain new stable values when the nonzero checksum appears. Thus the "half-baked data" issue arises only for the checksum itself, but this doesn't matter because the input testing logic starts reading the new data set at port 00, so the checksum is long since stable when it finally gets read.

Smalley3 continues to loop through the input ports, using the SEC-DED logic each time until port 7F again becomes zero, at which point the current loop is abandoned to await the completion of the new data set. This leaves one narrow crack of a "half-baked data" problem, if the sampling of port 7F happens to coincide with the zeroing of it in such a way that the IN instruction finds some but

not all of the one bits still there. That situation is perceived as a longitudinal checksum error, so an apparently bad checksum is read again to see if it has settled out to zero, in which case the current port loop is abandoned as above.

Verifying Output Data

Using the regular data sets from the Content Engine, *Smalley3* loops through the output ports 80-FF, writing systematic or random data and then reading it back. That leaves uncovered the correctness of the wiring from the output ports to the outside world; we identified but didn't have the resources to implement a way to gain such coverage within the BCE's functional limitations. It would involve the BCE remembering the last data set it put into the input ports, and *Smalley3* would copy each input data set into the corresponding output ports, leaving it up to the BCE to read the output ports and decide whether that data matched the last input data set.

Interrupt Testing

Interrupts are triggered by the BCE at regular intervals, and are separated in time so that they can be assumed to arrive "one at a time" and no interrupt will arrive while another is still in progress, but each arrival seems to the 80k85 instruction flow to be at a completely random time. *Smalley3* retains some control by planting its own addresses in the interrupt vector, but the PRE/POST/FOUND architecture for testing instructions doesn't apply. Verification consists of checking that each interrupt uses its correct target location in low memory and saves and restores PC and SP correctly (since an interrupt is essentially an externally triggered CALL). More important, we verify that it does not corrupt the progression of programmed logic driven by *Smalley3*, even though it necessarily makes one modification by leaving the "resume address" in RAM next to the top of the stack.

RAM Corruption Detection

This feature uses the 512 highest addresses in RAM to maintain row and column XOR-style checksums for all 64K bytes of RAM, where "row" means 256 consecutive locations and "column" means 256 locations whose addresses are 256 apart. The idea is that any one-byte corruption can be

located by discovering checksum errors for its row (low half of its address) and column (high half of its address). Furthermore, the check can reconstruct the corrupted byte and therefore shows which bits are bad. Row 254 (addresses FExx) contains checksums for 256 columns each covering row positions 0-253, and row 255 (addresses FFxx) contains checksums of every row including itself.

An interesting sidelight of the checking routine is the fact that the loop that XORs all the bytes that contribute to a given checksum cannot modify any part of RAM while it runs, and must maintain all of its state information in central registers.

The actual coverage obtained by this method is a little less than the full 64K bytes, because it would use too much time to construct the checksums often enough. Coverage of *Smalley3*'s scratch locations is negated by a scheme of shadow locations that prevent those locations from making any contribution to the checksums. That is, each scratch location is shadowed by 3 other locations, one in the same row, one in the same column, and one at the intersection of those two shadows. Each triad of shadows is copied from its scratch location at the beginning of the run, before the checksums are initially generated. Then when it's time to perform a check, each triad is updated from its scratch location (which would hide any corruption of the scratch location itself), but some coverage is regained by verifying that the 3 shadows agree with each other, showing that none of them has been corrupted since the previous check. A scheme of restorations applies to stack and other RAM locations outside the scratch area which have been properly modified by instruction testing.

There is also some logic to ignore any stack area location that may have been affected by an interrupt. Having built all this logic to handle scratch locations and made it work, we have to wonder whether it was the best use of resources, compared to a much simpler scheme to generate and check checksums of all of RAM except the scratch area. For the purposes of this paper, it serves as an interesting demonstration of what's possible; for purposes of the spacecraft, it does no harm and does obtain a little coverage in the scratch area.

RAM Subsystem Testing

Two tests, philosophically distinct from the design verification focus discussed so far, were added for the flight software. They are addressed to electrical design issues: the ability of the RAM to withstand abnormally high rates of state reversal, and possible crosstalk between bit positions that are "adjacent" in some sense.

Stress RAM by Rapid Bit Reversals: Yozzle

The name *Yozzle* is taken from a Honeywell mainframe diagnostic used in the 1960s to stress a magnetic tape drive's servos and tape handler by "yozzling," i.e. reversing the direction of the tape's motion as rapidly as possible.

There can be only one occurrence of *Yozzle* per run, following the final occurrence of *Elliot-Ness* and preceding the one occurrence of *PSF*. Like *PSF*, it operates on the contiguous majority of RAM addresses above *Smalley3*'s program code, that is, about 54KB out of the total of 64KB.

The essence of this stress test is to fill the subject RAM with alternating bytes such that every odd location contains the one's complement of the even location preceding it, and then reverse all 16 bits of each such location pair repeatedly, as rapidly as possible, for a parametrically varying number of repetitions. This is done by making each such pair the head of the stack, loading the one's complement of the pair's initial state into registers H and L, and using the XTHL (exchange top of stack with HL) instruction to do the reversals.

Any failure discovered by the YOZZLE test places data in RAM, overlaying certain locations that are normally used by the instruction testing but do not contribute information to the dump analysis. This data includes the address where RAM failed, the intended content, and the found content, each two bytes.

Pattern-Sensitive Testing of RAM: PSF

This test is based on John P. Hayes, *Testing Memories for Single-Cell Pattern-Sensitive Faults* [6]. As will be seen, it cannot be a rigorously exact implementation of the documented test because of the 80k85's architecture.

Hayes's notation is based on the "cell," that is, storage in RAM for a single bit, and on a tiling "neighborhood" consisting of a particular cell and 4

adjacent cells, immediately left, right, above, and below the cell under consideration. In the RAM occupied by *Smalley3*, the memory range to be tested is approximately the same as for *Yozzle*; however, there must be a whole number of 256-byte rows. The lowest available whole row is 2700-27FF, so the number of cells in the address range is 444,416 (55,552 bytes).

The documented test seems to assume the addressing of individual bits in a RAM chip, which is a normal environment for RAM testing in a chip foundry, but the 80k85 architecture addresses RAM bits 8 at a time. We assigned to the words “left” and “right” the meanings that are (mostly) intuitive in 80k85 architecture:

- left and right bit positions within a byte;
- to the left of the leftmost bit position of a byte is the rightmost bit position of the next-lower-addressed byte;
- to the right of the rightmost bit position of a byte is the leftmost bit position of the next-higher-addressed byte;
- but there is a wraparound rule to prevent the next byte (whether higher or lower) from going outside the row xx00-xxFF.

We assigned to the words “above” and “below” similarly intuitive meanings:

- above a bit position in any byte is the same bit position in the byte whose address is 256 higher than the subject byte;
- below a bit position in any byte is the same bit position in the byte whose address is 256 lower than the subject byte;
- but instead of a vertical wraparound rule, we restrict the “top” row FF00-FFFF to contain bits “above” bytes in row FE00-FEFF in this sense,
- and we restrict the “bottom” row 2700-27FF to contain bits “below” bytes in row 2800-28FF in this sense.

Even though *Smalley3*'s *PSF* reads and writes bits 8 at a time, we believe that the fault coverage is substantially the same. The logic is too complex to try to summarize here; interested parties can consult

the reference or even obtain the heavily commented *Smalley3* source code from OLD at GSFC.

Summary and Conclusions

Technical Summary

The fact that *Smalley3* occupies the low 9.3 kbytes of RAM not only demonstrates how much function can be packed into a modest-sized assembly-language program, but also suggests how software reliability suitable for spacecraft can be obtained (in part) by staying away from the gigabytes of OS and related infrasoftware that are common to PCs. Of course, the absence of any requirement for a graphical interface helps too.

Only 5.8 kbytes are executable code; 2.7 kbytes are tables, and 1.1 kbytes are variables. Those add up to more than 9.3 kbytes because some of the initialization code gets overlaid by variables.

Running time for each instruction test cycle with a maximum set of Monte Carlo initial states for each instruction is 14 seconds. Each cycle of the RAM corruption detector “*Elliot-Ness*” takes about 0.1 second; these can be commanded to occur at 6 different frequencies, but only the higher frequencies consume significant running time, 1.8 hours maximum. The *Yozzle* test runs for about ¼ second, but the *PSF* test takes about 500 seconds. All times are based on a 4 MIPS processing rate.

Programmatic Observations

The 80k85, a small 8-bit microprocessor with no instructions as complicated as even multiplication, seems to present a simple problem in design verification. However, the requirements for rigorous and thorough testing cast quite a long shadow for any computer, no matter how simple:

- largest practicable variety of initial machine states for each instruction;
- detection of largest practicable variety of unintended side effects of any instruction;
- detection of largest practicable variety of interference between programmed and unprogrammed activity (i.e., interrupts and arrival of inputs);

- verification of input/output wiring harness configurations;
- excitation of RAM components in ways that normal software wouldn't induce.

Despite the complexity to fulfill these requirements, the development of *Smalley3* itself was fairly straightforward thanks to a good-quality 8085 simulator usable at the development site. Most of the bugs in *Smalley3* code resembled 80k85 faults and were readily resolved by using the dump analysis features addressing such faults, with a major assist from the simulator's instruction-level trace. However, the fidelity of the 8085 simulator to the 80k85 was less than complete and had to be worked around. Also, test runs often had to incorporate artificial simplifications of long and complex runs because the simulator couldn't record more than 1 million instruction executions.

Some of the complexities could have been reduced or avoided by enhancements of the 80k85 and BCE design:

- 80k85 design could have been more of a clone of the 8085 design without hurting objectives;
- BCE could have supported wrap-back of outputs to inputs to facilitate I/O testing;
- BCE/80k85 interface could have provided an interlock to prevent the reading of an input port in a "half-baked" (partially updated) state.

Scalability Considerations

What if the processor had a 16-bit, 32-bit, or 64-bit architecture? There would certainly be a larger number of functionally distinct instructions, and they would include more complex functions like multiply and divide, floating point, possibly even vector and trig functions and decimal arithmetic. That much would produce an increase in the amount self-check code, and require enormously larger look-up tables, but no great change in qualitative complexity. However, the introduction of base and index registers would complicate the behaviors of all instructions and increase the variety of initial machine states.

We would expect vastly greater amounts of RAM, which would help with the larger tables

(though probably not enough), and would take considerably more time to check. There might also be a more complex I/O architecture with, say, programmed channels instead of just ports.

This challenge could perhaps be eased if the designs of the bigger computers included some Built-In Self-Test logic. For example, in 1970 the MIT Instrumentation Lab designed a SIRU (Strapdown Inertial Measurement Unit) controller whose arithmetic instructions calculated both direct and complement results and compared them. Still, that doesn't overcome the need for design verification and operational checking of the Built-In Self-Test logic. The boldest argument against such logic that we can recall was made by Seymour Cray, who left parity checking out of the design of the CDC 6600 because, as he said, you never can tell whether your checking logic is working.

Achievements and Utilization

In support of the 80k85 development phase, *Smalley3* made three major contributions:

- Caught a failure to carry a design detail into the netlist—this affected CMP B (Compare Accumulator vs. B register), which evidently hadn't occurred in the testing of application code;
- Identified a weak point in the electrical design of non-80k85 parts of the FPGA chip, based on a fan-out limit violation—seen in low-voltage tests that made a particular instruction fail;
- Exposed a poor margin in CPU-memory slew rate during high-temperature testing.

These achievements motivated NASA to apply *Smalley3* also to the operational phase, so that any suspicion regarding the LOLA DU's operation on lunar orbit can be investigated to see whether the root cause is a fault in the 80k85. This application was the impetus for adding the RAM subsystem testing.

Controversial Observation

A Minority Opinion on a Design Point

This is a minority opinion in the sense that only one of the three authors (Hugh Blair-Smith) embraces it. In a nutshell: the full power of external interrupts is more trouble in an embedded system than it's worth. Mainframes and other highly multi-tasking systems need interrupts to arbitrate among execution streams from many independent sources, some of which can be expected to do bad things like infinite loops. In a small embedded processor, the number of tasks is limited and well known, and their interaction is specified and designed in great detail. That means that frequent sampling or polling for external events, at points convenient to the logic, becomes feasible.

In the Space Shuttle GPC software, the rules for synchronizing redundant-set machines require essentially this scheme. Interrupts exist, but they don't do anything until all GPCs in the set agree on what's the highest-priority thing to do. These agreements are called "sync points," and one of the rules is that any routine that runs longer than a millisecond must perform a programmed sync point to effectively poll the interrupt subsystem.

The full flexibility of interrupt logic makes design verification harder and the possible paths through the code more numerous and less well known. Having said that, this is a controversial viewpoint, partly because considering backing away from interrupts is something of a cultural shock in real-time control, and partly because it can be mistaken for a plea for excessively rigid "washing-machine cycle" multi-tasking logic, the sort of thing that would not have survived the Apollo 11 program alarm (1202) caused by an unforeseen torrent of DMA cycles. Task priorities, which did save Apollo 11, do still matter.

References

1. Blair-Smith, Hugh, c. 1972, personal recollection of conversation with IBM Federal Systems Division engineers and submission of the subject subroutine to them, Cambridge MA.
2. Miller, Jim et al, c. 1973-2005, HAL/S Compiler System Specification, United Space

Alliance (but originated by Intermetrics, Inc.), pp 3-16 and 6-12 (which references CR11164 and DR106660), Houston TX,

http://klabs.org/richcontent/software_content/hal_s/hal-s_compiler_system_specification.pdf

3. Nicely, Thomas R, 1994, e-mail communication to whom it may concern, Lynchburg College, Lynchburg VA, <http://www.emery.com/bizstuff/nicely.htm>
4. Peterson, Ivars, 1997, Pentium Bug Revisited, Math Trek posted on MAA Online (Mathematical Association of America), Washington DC, http://www.maa.org/mathland/mathland_5_12.html.
5. Stott, F., 1987, Register Interaction in the 1802 Microprocessor (Interoffice Memo 514-B-761-87, with PIP No. 201 attached), Jet Propulsion Laboratory, Pasadena CA, http://klabs.org/DEI/Processor/1802/register_interaction/stott_memo/stott_memo.htm.
6. Hayes, John P, 1980, Testing Memories for Single-Cell Pattern-Sensitive Faults, IEEE Transactions on Computers, Vol. C-29, No. 3, March 1980, pp 249-254, available through http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1675556. This article seeks to generalize the somewhat more accessible *Efficient Algorithm for Testing Semiconductor Random-Access Memories* by R. Nair, S. Thatte, and J. Abraham, *ibid*, Vol. C-27, No. 6, June 1978. The latter cites Hayes's earlier work, *Detection of Pattern-Sensitive Faults in Random-Access Memories*, *ibid*, Vol. C-24, No. 2, February 1975, but criticizes the method set forth there as lacking significant coverage.

27th Digital Avionics Systems Conference
October 26-30, 2008