Real-time fault detection for small computers*

by J. R. ALLEN

Bell Telephone Laboratories
Naperville, Illinois

and

S. S. YAU**

Northwestern University
Evanston, Illinois

INTRODUCTION

Advancing technology and declining costs have led to a sharp increase in the number and variety of small computers in use. Because small computers are readily suited for many real-time applications, a great deal of work has been directed toward simplifying the interface between the computer and its peripherals. Hardware interrupting capability and a specially designed I/O bus are required for peripheral device interfacing in a real-time environment and such things as direct memory access, data channels, and multilevel hardware and software interrupt capability are common. These machines tend to be parallel, synchronous computers with a relatively simple architecture.

In a real-time environment, fault detection can be of major importance. Much of the past work has been directed toward the design of additional hardware, internal to the computer, which allows critical feedback loops to be controlled and often inserts special registers for the maintenance task.[1–4] These techniques require that the maintenance circuitry be designed concurrently with the computer itself and have access to components internal to the computer. Many problems can arise from attempting to modify an existing computer. For example, critical timing and gate fan-out can be disturbed, and most warranties become void if unauthorized modifications are made to a computer.

Other techniques cannot be used for real-time fault detection because they require manual intervention, excessive storage, noncontiguous memory locations or excessive execution time.[5–7] Much of the previous work attempts to locate faults as well as detect them. While fault location is desirable, it is more expensive and requires much more time and storage than fault detection. Many applications do not require fault location. It may be more feasible to either interrupt the task or perform the task manually during the diagnosis and repair interval; the most important thing is to recognize that the computer may be performing the task incorrectly.

An earlier technique attempted to simulate each instruction using different instructions and comparing the simulation result with the actual result.[7] This technique requires that the computer be somewhat operational and that it be capable of comparing the two results. A means is needed for determining that the test routine is executed periodically.

In this paper, we propose a real-time fault detection scheme for small computers which is effective for faults in the central processor unit (CPU), core memory and I/O bus. It requires an external monitor which is simple, inexpensive, and interfaces to the computer's I/O bus just as any other peripheral device. The monitor periodically triggers a program interrupt, causing the computer to execute a predefine test routine. During the course of the routine's execution, several key bit configurations are transmitted to the monitor. If the computer fails to respond or if the bit configuration does not match the one that is expected, then the computer is assumed to be faulty and the device may either cause a power down or sound an alarm.

The proposed technique compares favorably to the previously referenced techniques in fault detection. Certain faults will not be detected, however, because

* Portions of this Work were supported by U. S. PHS Grant No. 5 PO1 GM 15418-04.

** Depts. of Electrical Engineering and Computer Sciences, and Biomedical Engineering Center.

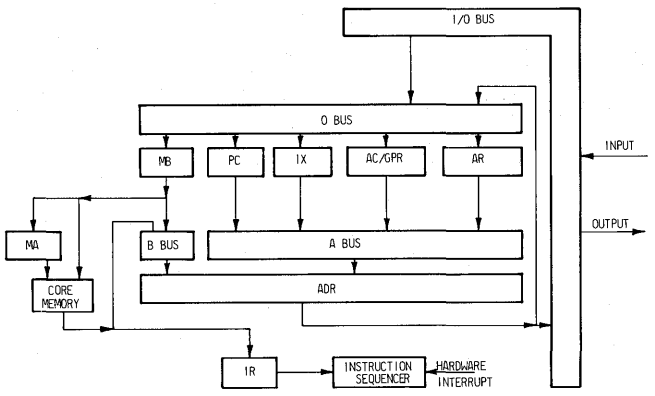

Figure 1—Typical computer architecture

internal modification to the computer is not allowed. In the past, real-time fault detection has carried a price tag comparable to the cost of the computer itself. A major advantage of this technique is that its low cost makes a great deal of fault detection possible in applications which would not previously have been cost effective. The fact that the technique assumes that no modification may be made to the computer means that many existing systems could make use of the method without extensive modifications.

COMPUTER ARCHITECTURE

The proposed fault detection method may be used with a variety of small computers, all of which are very similar in architecture. Included in this class of computers are such machines as Digital Equipment Corporation's PDP-8, PDP-9/15; Hewlett-Packard's HP-2100; Honeywell's DDP-516; Data General's Nova-1200 and Nova-800; Xerox Data System's Sigma-3; and Hitachi's HITAC-10. All of these are parallel, synchronous machines with hardware interrupt and I/O bus.

Figure 1 shows a block diagram of a typical small computer. The *program counter* (PC) contains the address of the next instruction to be executed. The *memory buffer register* (MB) serves as temporary storage for information being written into and read from memory. The address of the memory cell to be accessed is placed in the *memory address register* (MA). The decoded contents of the *instruction register* (IR) controls the instruction sequencer. Other registers commonly found are an *accumulator* (AC), *general purpose registers* (GPR), an *arithmetic register* (AR) and an *index register* (IX). Most small machines make use of some subset of these registers.

Most of the logic functions are performed in the general purpose logic unit (ADR). A pulse on the hardware interrupt lead will cause the instruction sequencer to save the appropriate registers and to begin executing a different program.

ADDITIONAL HARDWARE REQUIREMENTS

This section explains the function of each portion of the block diagram for the monitor hardware shown in Figure 2.

The DEVICE SELECT modules (DS1 and DS2) are basically an array of AND gates which allow the peripheral device control signals and data signals to pass only when the proper device select code is present on the device select bus. This allows several peripheral devices to share the same I/O bus without interfering with one another. Three device codes are required, XX, $\overline{XX}$, and YY. It is desirable to have one device code ($\overline{XX}$), which is the complement of the other to verify that each bit of the device select bus can be set to both logical states.

Device codes XX and YY cause the WIRED REGISTER to be gated onto the Input Bus. The WIRED REGISTER is constructed simply by attaching alternate bits to a voltage representing a logical one and the remaining bits to logical zero. When device code YY is selected, the "DEVICE = YY" lead is enabled causing the contents of the WIRED REGISTER to be complemented before being placed on the Input Bus. Reading from both device XX and device YY causes both a 1 and a 0 to be read from each bit position. Many computers determine the status of a peripheral device by executing an instruction which causes the next instruction to be skipped if a status FF is set. By executing this instruction with device code XX the BUSY/IDLE FF will appear to be busy; if device code YY is used, the "DEVICE = YY" lead causes the FF to appear to be idle. In this way the device status testing feature may be checked.

Device code $\overline{XX}$ is used when outputting bit configurations to the monitor for comparison to the expected value. The INTERRUPT TIMER is simply a monostable FF which, after an appropriate delay, will cause a program interrupt to be sent to the computer and at the same time sets a FF, which enables the RESPONSE TIMER. The testing frequency is set by adjusting the monostable's delay time.

The RESPONSE TIMER is used to determine that the computer is taking too long to respond and may be "lost." If the RESPONSE TIMER is not reset or disabled before it "times out," an OVERTIME signal

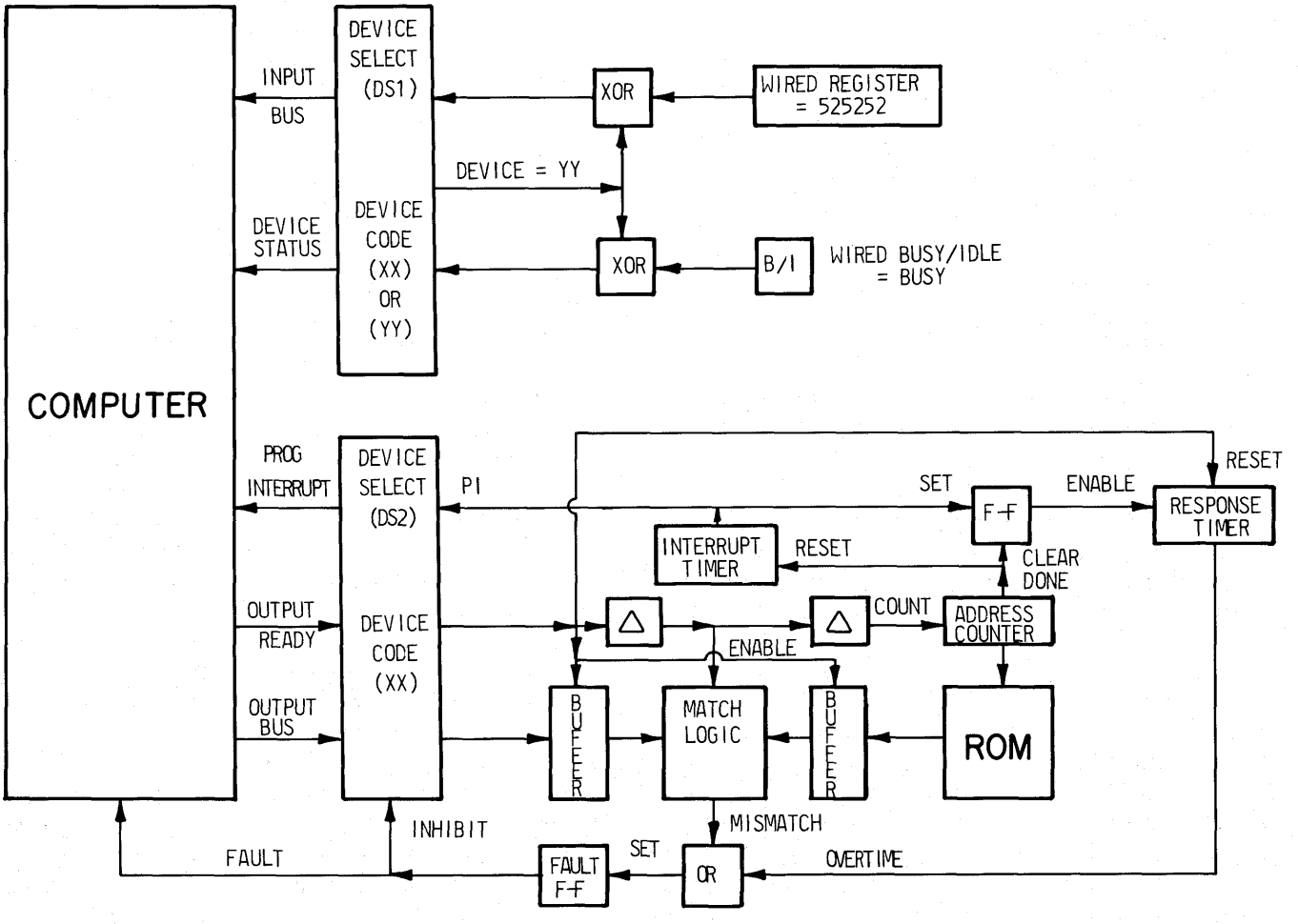

Figure 2—Block diagram for monitor hardware

is generated, indicating that a fault has been detected. A circuit to perform the RESPONSE TIMER function is shown in Figure 3.

The ADDRESS COUNTER is simply a cyclic counter designed to count modulo N, where N is the number of responses expected during a normal test. When the counter resets, a DONE signal is generated which disables the RESPONSE TIMER and resets the INTERRUPT TIMER. The ADDRESS COUNTER is incremented after each output from the computer to sequentially address the contents of the read-only memory (ROM). The ROM need not be large and may be built economically from a diode array.

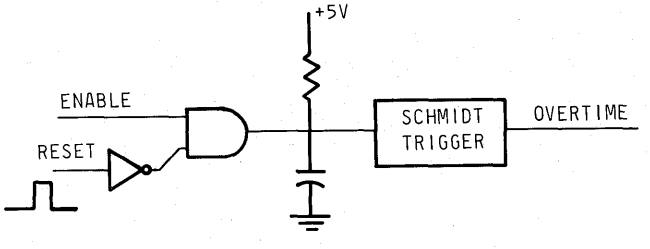

Figure 3—Response timer

The response from the computer and the expected response, read from the ROM, are both buffered and compared by the MATCH LOGIC. When enabled, the MATCH LOGIC basically OR's together each bit of the XOR of the two buffers to produce the MISMATCH signal. An OUTPUT READY signal from the computer is used to load the BUFFER, enable the MATCH LOGIC, reset the RESPONSE TIMER, and increment the ADDRESS COUNTER, all after appropriate delays.

If either a MISMATCH or an OVERTIME signal is produced, the FAULT FF is set. This inhibits any further output to the monitor, thus preserving the contents of the ADDRESS COUNTER and the two buffers as an aid for the diagnosis of the fault.

The FAULT signal may be used in whatever way seems appropriate; it may stop the computer, sound an alarm, or trigger an interrupt to call some form of self-diagnosis program, which may attempt to actually locate the fault.

All circuits may be built using conventional methods and commercially available logic gates. The total cost of the hardware components is estimated at approximately $250.

GENERAL GUIDELINES

Small computers seldom have a large amount of core memory, often as little as 4,000 words. Often, external storage is limited to paper tape, which requires manual intervention to be read. In order for a real-time fault detection scheme to be useful in these circumstances, its memory requirements should be small enough to allow it to remain resident in a very small core memory and still leave room for the system programs.

A large number of faults are severe enough to either stop execution or at least to make sequential instruction execution impossible. No additional effort need be made to detect such a fault; the resulting failure to produce the required sequence of responses will cause the fault to be detected. However, faults which affect only a single bit position of a bus or register can be among the most troublesome. This type of fault may allow execution to continue, perhaps indefinitely, without an indication the fault exists.

Certain portions of the computer are very difficult to check under the assumed restrictions. Input-Output leads, other than those used by the monitor, cannot be checked without adding more hardware for each set of leads. Such Input-Output leads include Direct Memory Access, Data Channel, Console Controls, and special device channels. These devices usually appear as a data bus input which cannot be controlled. It can, however, be determined that none of these inputs is stuck at a one level as this would always cause the output of the bus to be one.

The instruction control circuitry cannot be checked directly. There are no instructions which access most of the control leads. Therefore, the correct operation of a control lead can only be determined by checking conditions caused by that lead, such as the complementing of a register. A thorough check of the control circuitry, under the given constraints, would be lengthy. It can, however, be determined that most of the gating signals can be produced. This can be done by executing every instruction type and checking to see that the proper data transfers have taken place. This method would intuitively seem to be very effective and can be accomplished in a short period of time.

Because the test routine is periodically required to change the contents of memory locations outside of its own boundaries, the interrupt facility must be disabled during the execution of the test. This is necessary to insure that all locations are restored before control is released by the test routine. Since it is undesirable to lock out high-priority tasks for a long period of time, the execution time of the routine should either be very short or the routine should be segmented to allow the servicing of high-priority tasks.

There are a number of ways to cause the external monitor to recognize that a fault has occurred. One of the most reliable ways is to design the test in such a way that the computer will output an unexpected bit configuration in the event of a fault. This technique is demonstrated by the following example. The accumulator is first loaded with a bit pattern A and then caused to skip over an instruction which would place A on the I/O bus. After the skip, the AC was changed to a pattern B and output to the monitor. If the skip did not occur, pattern A would have been output and would not have matched the expected pattern B.

Another method is to cause the program to "loop" in the event of an error. The normal sequence of output bit patterns will then cease and the monitor will recognize the cessation of response as an indication of a fault. Use of these techniques reduces the read-only memory requirements for the monitor.

TESTING TECHNIQUES

This section describes several techniques, which may be used to test the various computer components.

Registers

Registers in parallel synchronous machines commonly resemble the configuration in Figure 4. Although the actual circuitry may vary, the function is very

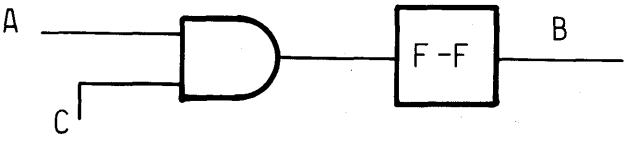

Figure 4—Flip-flop

simple and well defined. When lead C becomes enabled, the information on lead A (usually a major bus) is gated into the FF. When lead C is disabled, the FF retains the state of lead A. A variation of this type of register requires a fourth lead which clears the register prior to the enabling of lead C which then OR's the state of lead A into the FF. Verification that each bit of a register can be set to both logical states completes a functional test of the FF. A single possibility remains—a stuck at 1 fault on lead C. This fault is very severe because lead C is common to every bit of the register. Having lead C stuck at 1 would cause lead A to be fed directly into the FF and would totally incapacitate the register. In most cases the state of lead A changes many times between the setting and the reading of a register, thereby, destroying the original contents. If this does not happen automatically, an effort should be made to cause it to happen.

To check a programmable register, such as the accumulator, the register is first loaded with a bit configuration. The contents of the register are then output through the I/O bus to the external monitor. The programmable register is then loaded with the complement of the first bit configuration and its contents again output to the monitor. This procedure also checks the I/O bus drivers, and the outputs of each bus between the accumulator and the I/O bus.

Not all of the registers generally used in small machines are directly accessible under program control. In these cases a variety of techniques must be used in order to infer that the register can be set and read correctly. Five such registers were described in the introduction.

One of these registers is the arithmetic register (AR). Because the contents of the general purpose registers is commonly stored into AR so that AR may be used as the operand, many tests of the general purpose registers require the data flow to pass through AR, testing it at the same time. This commonly happens in the output instruction itself, meaning that no extra effort is required to check AR.

The second nonprogrammable register is the memory buffer (MB). This register is used to contain the data word core memory read-write operations; it must be capable of being set and cleared in order to correctly access core memory. If two complementary words can be read from memory, then the MB is operative.

The third nonprogrammable register is the instruction register (IR) which contains the OP code. This register is different from the above registers in that its contents are never gated to a point at which it could be displayed. In this case a set of instructions may be selected in such a way that together they incorporate both logical states for each bit. If it can be verified that each of these operations can be performed successfully, then it may be assumed that IR is operative.

The fourth nonprogrammable register is the memory address register (MA). This register is tested in the following manner: the contents of the sequence of addresses (0, 1, 2, 4, 8, . . . , 2^{n-1}, where n is the number of bits in MA, is saved in a test routine buffer area in real core for future restoration. Second, two complementary flags X and $\bar{X}$ are written into location 0 and read back. The flag X is left in location 0. Another flag (Y) is then written into each of the other n locations. If, after the n write operations, the contents of location 0 is still X, then the MA register is operative and the n+1 locations used for this test should be restored. To see this, assume that a bit (A) of MA is stuck at either 1 or 0. This is not to say that bit A is the only inoperative bit, but only one of possibly several. In this case, when an attempt is made to write into location 0, the flag will actually be written into another location (B). (If every faulty bit is stuck at 0, then B=0.) Later when an attempt is made to write into location 2^A, this data will also be written into location B, overwriting the original flag. In general, location B will be overwritten once for every inoperative bit in MA. When an attempt is made to read from location 0, the overwritten contents of location B will be returned, which will no longer be the flag originally placed there. It should also be noted that a fault in the address decoding circuitry or the memory itself cannot mask a fault in the MA.

The fifth nonprogrammable register, the program counter (PC), may be tested in much the same way as the MA register. The first flag to be written into location 0 is the binary configuration of a "JUMP TO A" instruction. Into the remaining n locations is written a "JUMP TO B" instruction. After the MA register has been checked, a transfer is made to location 0. This causes the PC to be loaded with a binary zero and the "JUMP TO A" instruction there to be executed. At location A, a signal is made to the monitoring unit that the transfer was successful and the contents of location 0 are changed to a "JUMP TO C" instruction. The code at location C will cause the program to loop, or output an unexpected word which will cause an error condition to arise. The test routine then proceeds to transfer to each of the locations 1, 2, 3 . . . 2^{n-1}. Each transfer should cause a return to location B, which simply continues the sequence. However, as with the MA register, if any bit of the PC is inoperative, then the "JUMP TO C" instruction will be executed and an error condition generated.

All of the above registers are found in most com-

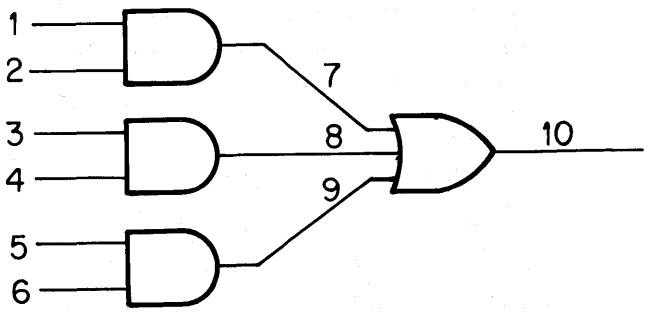

Figure 5—Bus structure

puters and the methods described are generally applicable.

In the methods for checking MA and PC, it is important to insure that addresses 0, 1, 2, . . . 2^{n-1} do not lie in critical parts of the test routine. This is simply a matter of convenience; if this restriction is undesirable, a slightly more general algorithm can be used to allow complete freedom in the location of the test routine. Since all of the n+1 locations lie in the first 2^n locations, the program could be loaded anywhere in the second half of the memory.

Data bus

A computer data bus typically consists of some variation of the circuit shown in Figure 5. Although there are many ways to implement this function, using different types of logic gates, nearly all faults will behave as though one of the leads numbered 1-10 has become frozen at one logical state. Certain varieties of logic allow the OR function to be accomplished simply by tying the outputs of gates A, B, and C together. This arrangement also satisfies the above statement. One input of each AND gate is usually common to all bit positions and functions as a gating lead for transferring data onto the bus.

A data bus may often be directly accessible to a number of peripheral devices for such features as direct memory access or a data channel. When this happens, one or more of the AND gates in Figure 5 may have inputs which cannot be controlled without interfering with certain of the peripheral devices. There will, therefore, be a few of the AND gates which will not be checked.

For the remaining AND gates, the objective is to verify that each input and output can assume either logical state. This can be done by gating both a 1 and a 0 onto the bus from every AND gate.

Control circuitry

Unlike a data bus or a register, the circuitry which controls the gating and timing for the instruction execution is neither simple in function nor standard in design.

The approach taken here is to include segments of code in the test routine which will exercise each instruction and check to see that it is performed correctly. This thoroughly tests the instruction decoder by providing it with all possible inputs. The control circuitry is the most difficult part of the computer to check because there is virtually no means of direct communication between this circuitry and the I/O bus. All tests must be made by performing an operation and verifying that the correct operations have taken place. It is possible, however, that something totally unexpected may happen in addition to normal operation of the instruction. To detect a fault such as this would require an extensive test for every instruction. There is no method known, at present, which would produce such a test using only the I/O bus with no hardware modifications to the computer. If such a method were available, it is likely that the storage requirement and execution time would limit its usefulness for a real-time environment. Fortunately, faults which occur in the control circuitry are usually quite severe and grossly affect the instruction execution. Therefore, a thorough exercising of each instruction appears to be the best approach to this problem and may be relied upon to detect the majority of such faults.

The length of the test can be reduced by studying logic diagrams in an attempt to find sections of logic used by more than one instruction. In some computers, for example, indirect addressing is handled by the same microinstruction cycle in every instruction. It would not be necessary, therefore, to test indirect addressing with each instruction.

Correct operation of certain instructions may be assumed if execution of the test routine is not possible without them. For example, it may be assumed that the JUMP instruction is operative because the first instruction executed after a hardware interrupt is a JUMP to the routine which is to handle the interrupt.

Although the approach used in this case is largely intuitive, if a little time is spent familiarizing oneself with the computer's logic and timing, the number of

faults which can escape detection can be greatly reduced.

Logic function and condition logic

Most Boolean operations may be easily tested by exhaustion of their respective truth tables.

As a general rule, small computers have a ripple-carry adder. Because the circuitry is simpler, this form of adder is easier to check than one using carry-look-ahead. To check an adder, it is necessary to verify that each bit position can both generate and sink a carry. Both of these tests may be made simultaneously for all bits by adding a word of alternating ones and zeros to itself ($25252_8+25252_8$) and then adding that same word's complement to itself ($52525_8+52525_8$). It is also necessary to verify that each bit position can propagate a carry. This is accomplished by adding 1 to 77777_8. Adding $77777_8+77777_8$ verifies that every bit position can simultaneously generate and sink a carry. The remainder of the truth table may be verified by adding $00000_8+00000_8$, $77777_8+00000_8$, and $00000_8+77777_8$.

It is desirable to consult the logic drawings in order to determine how to test the overflow and conditional transfer logic. Although the implementation of this logic varies between machines, it is quite straightforward and simple to check.

Core memory

The core memory is, perhaps, the functional element which will fail most frequently. This may be attributed to the requirement for high-power circuits operating under closer tolerances than the conventional logic

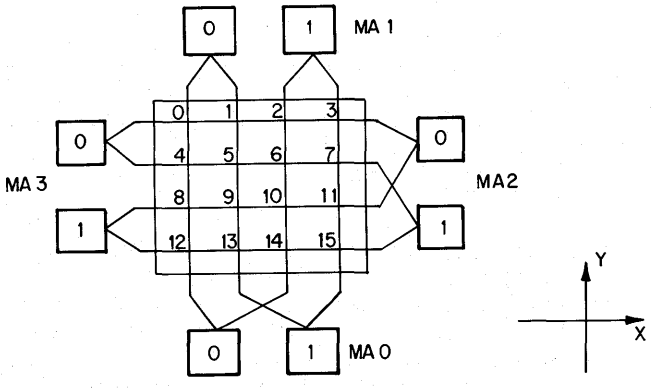

Figure 6—Core memory bit slice

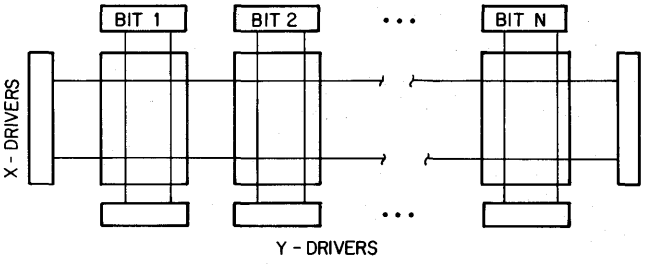

Figure 7—Memory addressing structure

gates used in the rest of the computer. Many faults which occur in a core memory are the type which may be present for a long time before being discovered, such as a fault which affects only a rarely used part of memory. The ability to read and write at every address does not mean that the memory is free from faults; a faulty address decoder may read or write at two locations simultaneously.

Small computers commonly use a 2-1/2D arrangement for the memory address decoding.[8] This method relies on a coincidence of current on an X-axis lead and a Y-axis lead to select a core to be written or read. To do this the address bits are divided into two fields, which independently control the X and Y current drivers. In addition, each of these fields is further divided into two subfields. For example, if there are n bits in the field which selects the Y lead, then half of these bits will select one of $2^{n/2}$ current sources; the other half will select one of $2^{n/2}$ current sinks. Since every source is connected to every sink, the independent selection of one source and one sink selects one of the 2^n Y-axis leads. Different sources and sinks are used for the read and write operations.

Figure 6 depicts the addressing scheme for a 16-word memory requiring a 4-bit MA. Only a single bit position of the memory word is shown. Other bits of the same word are selected by replicating the Y-axis drivers for each bit position as shown in Figure 7.

During the write cycle, only the Y-axis drivers corresponding to bits which are to be set to 1 will be enabled. Because the read and write operations require currents in opposite directions, separate drivers are required for each of these operations. This means that a fault may affect the read operation and not the write operation and vice versa.

The test procedure for checking the address decoder is to select one of the four subfields, having length m, to be checked first. The contents of each of the 2^m locations obtained by selecting all combinations of these m bits and holding the remaining bits constant

are stored in a buffer area. Each of the 2^m locations is then set to zero. Into one of the locations is written a word of all ones. Each of the remaining locations is checked to see that its contents is still zero. The original location is then checked and cleared. This process is repeated for each of the 2^m locations. Upon completion of the test, the contents of the locations are restored and the test is repeated for each of the remaining three subfields.

To justify this test, assume that one of the input leads to a read current source were stuck at 0. This would mean that at least one bit in the set of test words could not be read as a one and would be detected by the test. It is also possible that an input to a write current source could be stuck at 0. Because of a core memory's destructive read-out, the corresponding bit, or bits, could be read once and then never rewritten as a 1. Again, at least one bit of the set of test words would be stuck at 0 and the fault would be detected.

Suppose now that an input to a read current source were stuck at 1. This means that one of the 2^m addresses will select two read current sources at the same time, dividing the read current. The result will now depend upon the tolerances of the memory. The divided currents may not be sufficient to cause a core to switch. At least one test bit would then appear to be stuck at 0 and the fault would be detected. The divided currents may, however, each be sufficient to switch a core. During its execution, the test will cause a 1 to be written by using lead A. Later, when an attempt is made to read a 0 by using lead B, lead A will also be selected and a 1 will be read. Similarly, an input to a write current source may be stuck at 1. If the resulting split in the write current is insufficient to switch a core, the inability to write into certain test locations will be detected as before. If each half of the divided current is sufficient to switch a core, the writing of a 1 into some location C will also write a 1 into some location D. This extra 1 will be detected when location D is checked for a 0. Because this test is considerably longer than the previous tests, it may be desirable to partition the test. A very natural partition would be to check each group of current sources separately.

The only way to check individual cores in the memory is to read and write using every location. Although this must be a lengthy test, some time saving can be realized if the locations are checked by reading a word and writing its complement back into the same location. If the complement can be read back, the word is good. This method makes use of whatever is already in the memory location and, therefore, saves the time which would have been required to save the contents to the location and initialize the location. This test is normally not included in the test program because of the time required for its execution.

SUMMARY

This procedure has been applied to a DEC PDP-9 computer with two 8K core memory modules. The test routine requires 550 words of core memory and a maximum of 8 milliseconds per pass. The time needed to test the core memory increases with the size of the memory itself; but, by segmenting the test so that only a portion of the core is checked during each pass, it is possible to increase the memory size without increasing the amount of time required for a pass. All tests of the CPU itself are made each pass, but 12 passes are required to completely test the memory. The hardware monitor requires 58 words of read-only memory and solely determines the frequency at which the tests are made. This frequency may be adjusted according to the work load on the computer.

This technique would seem to have many applications on small machines which have previously avoided fault detection because of the cost or the need to make hardware changes to the computer.

ACKNOWLEDGMENTS

The authors would like to thank Messrs. Gary F. DePalma, G. Wayne Dietrich and J. S. Tang of Northwestern University for helping in the implementation of this fault detection scheme on the DEC PDP-9 computer.

REFERENCES

1 E G MANNING
On computer self-diagnosis: Part I—Experimental study of a processor
IEEE Trans Electronic Computers EC-15 pp 873-881 1966

2 E G MANNING
On computer self-diagnosis: Part II—Generalizations and design principles
IEEE Trans Electronic Computers EC-15 pp 882-890 1966

3 R W DOWNING J S NOWAK
L S TUOMENOKSA
No 1 ESS maintenance plan
Bell System Technical Journal Vol 43 pp 1961-2019 1964

4 K MALING E L ALLEN JR
A computer organization and programming system for automated maintenance
IEEE Trans Electronic Computers EC-12 pp 887-895 1963

5 C V RAVI
Fault location in memory systems by program
Proceedings of AFIPS Spring Joint Computer Conference pp 393-401 1969
6 M S HOROVITZ
Automatic checkout of small computers
Proceedings of AFIPS Spring Joint Computer Conference pp 359-365 1969
7 T R BASHKOW J FRIETS A KARSON
A programming system for detection and diagnosis of machine malfunctions
IEEE Trans Electronic Computers EC-12 pp 10-17 1963
8 P A HARDING M W ROLUND
Bit access problems in 2½D 2-wire memories
Proceedings of AFIPS Fall Joint Computer Conference pp 353-362 1967