筑波大学
1982.1.13
電子・情報工学系

ISE-TR-82-26

SOFTWARE-FAULT DETECTOR FOR MICROPROCESSORS

by

Kozo Itano

Tetsuo Ida

January 15, 1982

INSTITUTE
OF
INFORMATION SCIENCES AND ELECTRONICS

UNIVERSITY OF TSUKUBA

Software-fault Detector for Microprocessors

by

K.Itano* and T.Ida**

* Institute of Information Sciences and Electronics,
University of Tsukuba,
Sakura-mura, Niihari-gun, Ibaraki 305, Japan.

** Institute of Physical and Chemical Research,
Hirosawa, Wako-shi, Saitama 351, Japan.

This paper is a revised version of the paper presented at ACM-SIGSMALL Symposium on Small Systems on October 13-15, 1981.

key words: software reliability, run-time hardware support, key-lock memory protection, incremental protection code, array bound checker, program module protection.

Abstract:

For the realization of the means to develop more reliable software for microcomputers especially in real time environments, we design a hardware tool called "software-fault detector" which detects software faults such as misaccess to an element beyond the range of an array. The implementation of the mechanism for such address range checks is generally difficult in microprocessor environment, since internal registers are not readily visible to external logics. We introduce an "incremental" key-lock protection scheme into fixed microcomputer architecture INTEL 8080 because of its popularity and simplicity. In this scheme, a "lock" is a protection code associated with the storage cell, and a "key" is associated with access capability such as address. In each memory access to a cell a check is made whether a key matched against the lock of the addressed cell. In this paper, we present the details of the scheme and its analysis. Further, we present an actual hardware design of the software fault detector. Our design methodology is to realize a detector by the use of identical microprocessor 8080s, as an independent one-board module which can be connected to the memory bus of the host system.

1. Motivation

Recent LSI technology has improved the hardware cost performance of microcomputers. Increased execution speed and memory size of current microcomputers invite many sofisticated applications. As a result we are often faced with the situations where we have to develop fairly large and complex software on site (in a possibly short period of time). It is urged to provide adequate hardware tools to support on site software development and moreover to implement reliable software, especially in some applications where microprocessors are used in real time environments.

We examined the means to make microprocessor software more reliable and in this paper present a practical solution to one of the essential difficulties of software developments. We observe that common software faults [3,4] are caused by erroneous accesses such as 1) misuse of undefined variables, 2) misaccess to an element beyond the range of an array and 3) erroneous jumps. The theme of our paper is to prevent these kinds of access violation errors.

Range check of arrays may be performed by the use of software [2], but it may be impractical or even impossible to keep such check codes in production programs such as of real time process control.

The implementation of the mechanism for address range check is difficult in microprocessor environment, since internal registers are not readily visible to external logics. Page or segmentwize memory protection mechanism performed in conjunction with address translation which is employed in large computers (and also some recent 16 bit micro computers with large address space) is incapable of protecting user-defined data structures, such as arrays, stacks and procedures, because these data structures, large or small, are not necessarily allocated fixed page size.

We introduce an "incremental" key-lock protection scheme into a microprocessor architecture. As our design target we selected INTEL 8080 microprocessor because of its popularity in industrial applications. Our principal design methodology is

(1) to realize a detector using a microprocessor identical to the host processor,
(2) to realize a detector as an independent one-board module which could be connected to the memory bus of the host micropocessor system, and
(3) to realize a detector by which software of the host system is as little affected as possible.

2.Key-lock protection scheme

2.1 Basic concept of key-lock protection

A "lock" is a protection code associated with the storage cell and a "key" is associated with access capability, i.e. address. The concept of key-lock to protect memory blocks introduced to large computer systems is extended to the storage cell level concept, i.e. to specific data cells such as constants, variables, arrays, and strings.

First, for simplicity, assume that every consecutive storage have unique protection codes on the whole storage. In making access to such storage cell whose address is "a", the CPU should present the key together with the memory address "a". If the key matches with the lock, we know the access is correct.

In order to present such a key in each memory access, CPU should obtain the key associated with the contents of the storage, if the content is an address. To realize the scheme we need a special memory structure in which each storage cell contains two fields; information field, lock field. The information field is subdivided into data field and key fields. The key field is significant only when address is stored in the data field.

Since a lock is set at the time when the storage is allocated, the lock field cannot be modified by the non-priviledged programs but is only modified by the storage allocator. On the other hand, since the "key" is associated with the address, the key is loaded into the CPU register and is stored into the

storage together with address.

2.2 Uniform protection code

First we consider assigning the same protection code to the locks of every cells belonging to the single consecutive area. Hence, different areas have unique different protection codes. A general principle of storage access operation is (cf. Figure 1(1)):

(1) CPU has the head address "a" together with key "k" to the area A as (a, k),
(2) and the CPU calculates real address (a+i, k) by indexing "i",
(3) then, CPU makes access to the cell(a+i) and simultaneously checks whether k equals to the protection code of the cell(a+i).

When there are N different areas, we need log N bits of protection codes to distinguish the areas uniquely. If there is not sufficient memory bits for the protection codes, complete protection cannot be done. However, the probability of detecting erroneous access is shown to be very high even with limited number of bits.
With k bits in handling N areas, the probabilities of error detection q is given by the following formula [1] :

$$q = \frac{1 - \frac{1}{2^k}}{1 - \frac{1}{N}} \qquad \text{where } N > 2^k$$

$$\text{and } q = 1 \qquad \text{where } N \leq 2^k .$$

Probabilities of typical cases are given in table 1.

2.3 Incremental protection code

Although the uniform protection code is very simple to implement, construction of the hardware based on the uniform protection code is difficult on microprocessor. To be concrete, to attach auxiliary processors which operate independently from the host CPU under different program control, but yet check the detailed behavior (in particular memory accesses) of the host CPU is difficult to realize, because internal logic and states of the host CPU is not visible to external world. Hence, we develop a more suitable scheme for the detector which is implemented by microprocessors identical to the host processor as described later. We transformed the scheme into a compatible one using "incremental" protection codes.

In handling M cells of area A whose head address is "a", we assign protection codes to the locks of each cells as:

lock of cell(a+j) = k + j,

for $0 \leq j \leq M-1$,

where k is a base protection code for area A.

The term "incremental" is named after the property that the incrementally increased protection codes are assigned to the contiguos cells in this way. The essential feature of the incremental protection codes is that the increment of the key is the same as the increment of the address, and we can perform the equivalent checks to the one by the uniform protection codes. A general principle of storage access operations are (cf. Figure 1(2)):

(1) CPU has the head address "a" together with
head key "k" to area A as: (a, k),
(2) and CPU calculates the real address and the real key
by index "i" as (a+i, (k+i modulo K)),
where K is defined later,
(3) then makes access to the cell(a+i), and checks
whether (k+i modulo K) equals to the protection
code of the cell(a+i).

We note that in the case of the incremental protection codes we cannot distinguish areas simply by assigning different values to the base protection codes to each area. For example, the areas A1 and A2 in Figure 2(1) cannot be distinguished. We analyze the conditions for detection more closely.

On any two different areas A and B, "a" and "b" are the head addresses of areas A and B, and "L_a" and "L_b" are the base protection codes assigned to the head cells of areas A and B. When N diffrent areas:

$$A_0, A_1, \ldots, A_{N-1}$$

exist on the storage in the same time, the condition of error detection is given as:

$$L_i \neq (L_j + (a_i - a_j) \text{ modulo } K),$$

where a_i is the physical head address of the area A_i,

and L_i is the protection code assigned to the head

cell of the area A_i.

The sufficient condition for the complete detection is $K \geqq N$.

Although a_i is fixed when the area is allocated, we can

determine L_i $(0 \leqq i \leqq N-1)$ systematically as follows:

(1) First, choose L_0 properly.

(2) Compute L_i as:

$$L_i = ((a_i - a_{i-1}) + L_{i-1} + m) \quad \text{modulo } K,$$

where $1 \leq i \leq N-1$ and m is relatively prime to K.

Most simple "m" is 1, and we may rewrite L_i as:

$$L_i = ((a_i + i) + (L_0 - a_0)) \quad \text{modulo } K.$$

This relation gives a simple systematic method generating base protection codes for unique identification of all areas as shown in Figure 2(2).

Both the incremental protection code and the uniform protection code require the same size of code fields to uniquely identify the areas. Therefore, the power of error detection on both schemes are the same. In the case that the number of bits for the key-lock memory is limited and the relation $N \leq K$ is not satisfied, the probabilities of error detection is also the same. The incremental protection code scheme can be implemented with much difficulty by multiple processors as is shown later, because the scheme is based on the assumption that programs running in the host system can be tightly coupled to the control programs for auxiliary processors which manipulate keys and locks. We take advantage of the fact that mostly the

same program can be obeyed by the host and auxiliary processors alike.

2.4 Program module protection

Program modules such as procedures can be considered as a kind of consecutive storage areas. If we assign protection codes to the locks associated with the instruction words of such modules, the key-lock protection mechanism can also work in the instruction fetch cycles in the same way as the data protection. Keys are stored in the key memory associated with address relating to the jump and call instructions shown in Figure 5. These keys are loaded into the program counter of the key processor on the execution of such jump and call instructions.

3. Hardware organization of the detector

Figure 3 shows a modular software-fault detector connected to the address bus of the host system. The detector consists of two microprocessors, auxiliary functional logics and three memories: lock memory, key memory and flag memory.

All these three memories are realized as independent banks, and can be addressed in parallel with the main memory by the host processor. Key and lock memories are used to hold the keys and the protection codes respectively, and the flag memory is used to hold a single identification bit which specifies whether the

associated data of the main memory is address. We use two identical microprocessor 8080s as a key processor and a flag processor respectively, and outside of the microprocessor a comparator is provided for high speed checking of the key and the protection code. The data lines of these processors are connected to the specific memories through a multiplexer, because the instruction opcodes are supplied directly from the main memory during the fetch cycle of the host processor.

The three microprocessors (main, key and flag) should be synchronized by the same clock system, and the same instruction opcode (the first byte of the instruction) should be given at the same time. Although most instructions are sent to these three microprocessors, some special instructions are mapped into harmless ones such as NOPs for the flag processor. Therefore, the execution time of such a processor may differ from the others. In order to synchronize such processors, "wait" states are inserted in the M1 cycle (the first instruction fetch cycle) by controlling the memory ready status.

The use of the identical microprocessors is essential in modular design of hardware, because the processors have the same registers, instruction decoders and execution timing controls in their own LSI chips. Hence, these LSI chips can make our design of hardware simple and compact.

4. Principles of operations

4.1 Basic operations

(1) Opcode fetch cycle

The first action of instruction execution is the opcode fetch. At this fetch cycle, the following operations <a-d> are performed in parallel:

<a> The host processor outputs the contects of the program counter to its own address lines, and all the four memory banks are accessed by this same address simultaneously.

<b> The host processor reads a byte of data (opcode) from the main memory.

<c> The key processor outputs the contects of its own program counter (procedure-wise key) to its own address lines, and the comparator checks the equality between this address and the data (protection code) read from the lock memory. If an error is detected during the comparison, an interrupt occurs to the host processor.

<d> The key processor reads the same byte of data (opcode) from the main memory; during this cycle, data bus of the key processor is switched to the data lines of the main memory.

(2) The other cycles

The other cycles of the execution may be either instruction operand fetch cycle or execution cycle. In these cycles, the following operations <a-d> are performed in parallel:

<a> The host processor executes the opcode; if necessary it may access to all the four memories by the same address which is the following: the program counter, registers, or internal working address register.

<b> The host processor reads or writes the data from or to the main memory if necessary.

<c> The key processor executes the same opcode as the host processor; it outputs the corresponded key to its address lines. Then, the comparator checks the equality in the same way as the previous case.

<d> The other actions of this key processor to the data depend upon whether the content is a key or not. In the case that the data is a key, the key processor reads or writes the key from or to the key memory.

Otherwise, in the case of read operation the key processor reads data from the main memory (as described in later example), and in the case of write operation it writes nothing to the key or main memory except that the operation is performed to the stack area. The decision whether the content is a key or not is made by the flag processor and the flag memory.

(3) Incrementation of the keys

Once the address is loaded into one of the registers of the host processor, address arithmetic operations such as increments or index additions are usually performed on such addresses. In parallel with such operations, the key processor performs the same operation on the corresponded keys in order to retain the consistency between the key and the address. This is simply done by the execution of the same instruction. The incremental protection code is designed to match with such calculations of keys.

(4) Invalid key operations

In our scheme, arithmetic operations on addresses are prohibited other than a±i or i±a where a is address and i is integer. (Address should be transformed into integer by a - address '0', if general arithmetic operations on addresses are required.) The flag processor checks whether the invalid operation was performed on the address.

(5) Stack operations

Although the values in the key processor should not be stored into key memory usually, there is one exception. We have to save both key and value onto the stack in the case of stack operations such as PUSH.

4.2 Examples of the basic operations

(1) Simple reference of an array element

First, consider that CPU loads the contects of the second byte of the array AA into A-register. The program may be written as P1:

LXI	H,AA	I	LXI	H,base key of AA
INX	H	I	INX	H
MOV	A,M	I	MOV	A,M

(i) host processor (ii) key processor

Program P1. Reference of the second element of array AA

In this case, when the host processor executes the instructions as program P1(i), the key processor executes the instruction as program P1(ii) in parallel. However, we note that the program P1(ii) does not exists explicitly on the memory. On the third step of this program, the key processor would present the contents of its own HL register to the address lines. This is the key to be compared with the lock of the second byte of the array AA. An example of object codes for the program P1 is shown in Figure 4.

(2) Indexed access

In the program P2, the CPU loads the contents of the fourth byte of array AA into A-register. But the value 3 is given in the DE register as as index. If we could not load this value 3 into the DE register of the key processor at the second step, the key processor could not produce a proper key in the later memory access. Therefore, we load value 3 into the DE register of the key processor at step 2. To perform this, the value (* in the program P2(ii)) must be transfered from the main memory data bus to the key processor data bus in the proper timing. This switching is done by the contents of flag memory; that is, the data bus is connected to main system in the case that the flag indicates "not address", and connected to the key memory in the case the flag indicates "address". Thus, keys and values are mixed in the key processor. A flag processor is used to prevent this confusion.

LXI H,AA	LXI H,base key of AA	LXI H,address flag
LXI D,3	LXI D,3 (*)	LXI D,value flag
DAD D	DAD D	DAD D
MOV A,M	MOV A,M	MOV A,M
(i) host processor	(ii) key processor	(iii) flag processor

Program P2. Reference by index access

(3) Programming restriction on parameter passing

In our scheme, we can pass the key of the parameter to the called subroutine by the use of registers as parameter. However, in the case the parameter address is passed by the program counter, mostly by return address, with 8080 microprocessor we have no mean to pass the proper key of those parameter area to the subroutine, even when we assign the different protection codes to such parameter area. There are two choices for this case:

(i) We restrict the programming technique or code generation convention, and we pass the parameter address through a register as shown in Figure 5.

(ii) Or, we abandon the runtime check; that is, we do not distinguish the parameter area from the instruction space.

However, the first strategy is to be preferred to improve the software reliability.

5. Concluding remarks

We designed the detector based on the INTEL 8080 microprocessor as a hardware tool to improve the software reliability, and further to increase software cost performance. Our detector reqiures additional hardware resources such as processor chips and extra memories. However, the cost would be minimal, considering the fact that the cost of software development occupies large part of the total system development cost, especially in the case of a dedicated embedded system.

In our scheme other additional hardware will make further protection possible. For example, an additional one bit flag memory to each memory cell can detect the illegal reference of undefined values of a variable [7]. If we apply the bound checking mechanism to limit the range of pointer (address variable), we could incorporate a more complex range checking including subrange limitation. Further, a mechanism which can monitor dynamic behavior of the program [5,8,9] would profitably be incorporated as hardware tools for the realization of the powerful debugging environments.

References:

[1] K. Itano and T. Ida,
Hardware array bound checker on tagged architecture,
ISE-TR-80-16, University of Tsukuba (1980).

[2] N. Suzuki and K. Ishihata,
Implemantation of array bound checker,
Proc. of 4th ACM Symposium on Principles of
Programming Languages (1977).

[3] G.J. Myers, Advances in computer architecture,
John Wiley and Sons (1978).

[4] G.J. Myers, The art of software testing,
John Wiley and Sons (1978).

[5] L.G. Stucki, New directions in automated tools for
improving software quality,
in 'Current trends in programming methodology',
Vol. 2, Program Validation,
Prentice Hall, (1977), 80-111.

[6] C.V. Ramamoorthy, R.E. Meeker, and J. Tunner,
Design and construction of an automated software
evaluation system,
IEEE Symposium on computer software reliability,
New York, (1973), 28-37.

[7] J. R. Ehrman, System design, machine architecture,
and debugging, SIGPLAN Notices, 8(1972), 8-23.

[8] J. R. Kane and S. S. Yau, Concurrent software
fault detection,
IEEE trans. Software Engineering, SE-1, 1(1975), 87-99.

[9] H.J. Saal and L.J. Shuster,
On measuring computer systems by microprogramming,
Microprogrammin and Systems architecture:
Infotech State of the art reports, Berksire, England:
Infotech (1975), 473-489.

Figure captions:

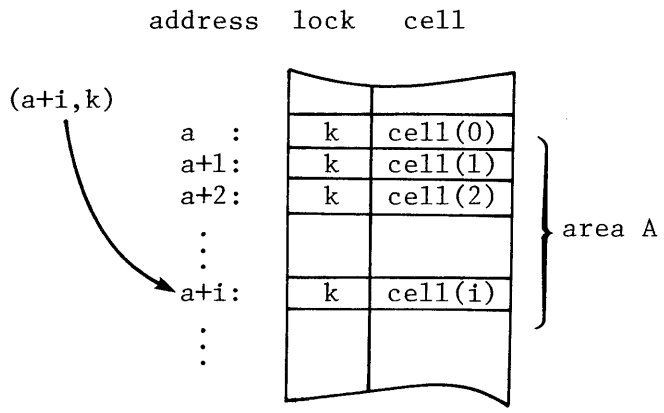
Figure 1. Basic key-lock protection scheme.
Figure 2. Incremental protection codes with two bit lock field.
Figure 3. Hardware organization of the modular software-fault detector.
Figure 4. An example of object codes of program P1.
Figure 5. Procedurewise protection scheme and parameter passing.

N \ k	1	2	3	4	8
20	0.526	0.789	0.921	0.987	1
50	0.510	0.765	0.893	0.957	1
100	0.505	0.758	0.884	0.947	1
200	0.503	0.754	0.879	0.942	1
500	0.501	0.752	0.877	0.939	0.998
∞	0.500	0.750	0.875	0.938	0.996

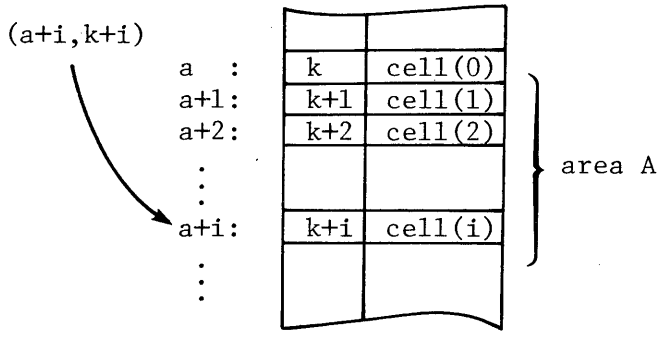
k = number of bits of protection code

N = number of areas

Table 1. Probabilities of error detection

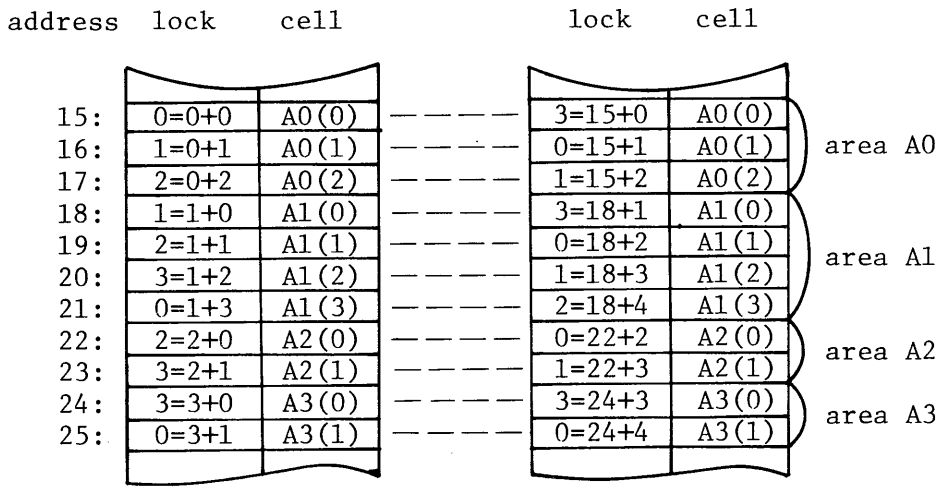


(1) Uniform protection code



(2) Incremental protection code

Figure 1. Basic key-lock protection scheme.



(1) Incrementing the base protection codes

(2) Assignment of the protection codes by $L_i = a_i + i$.

* All additions are made in modulo 4.

Figure 2. Incremental protection codes with two bit lock field.

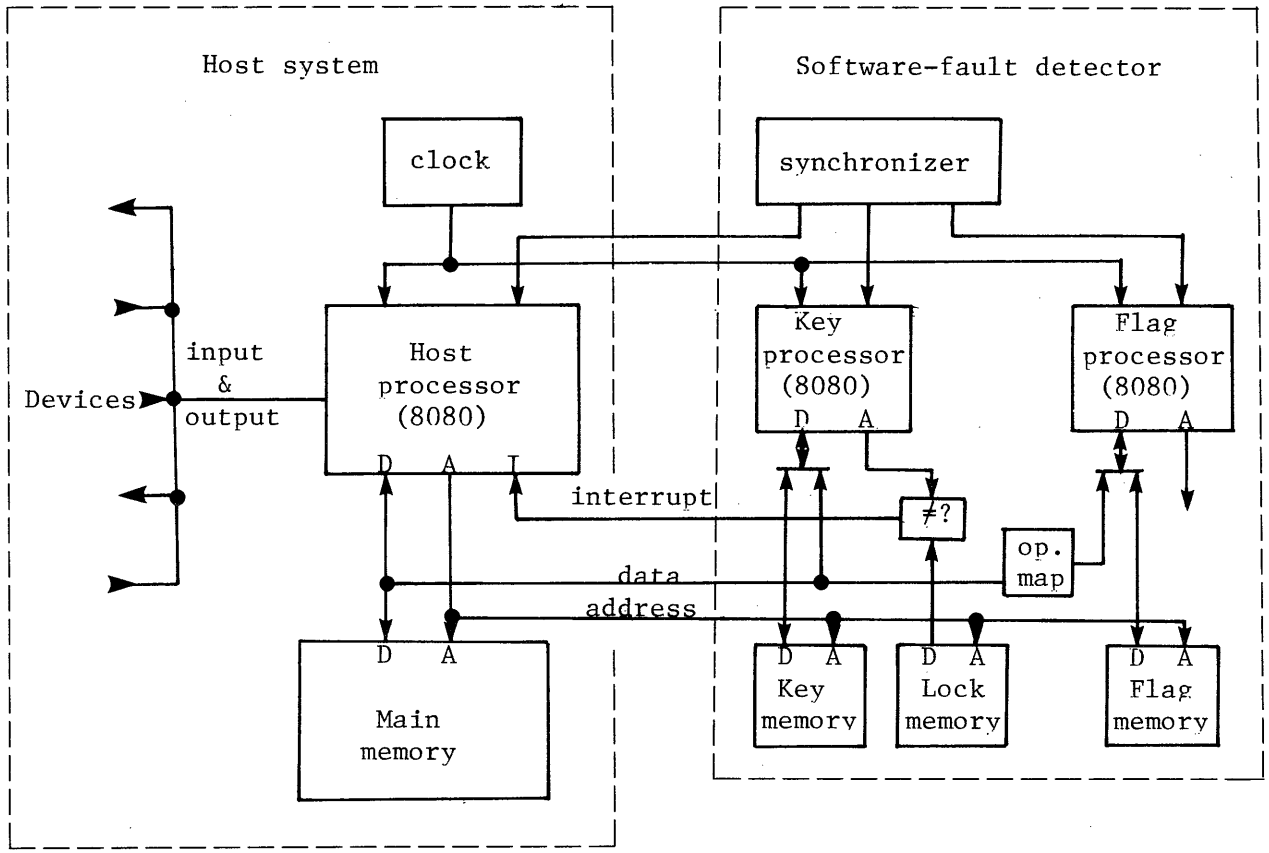


Figure 3. Hardware organization of the modular software-fault detector

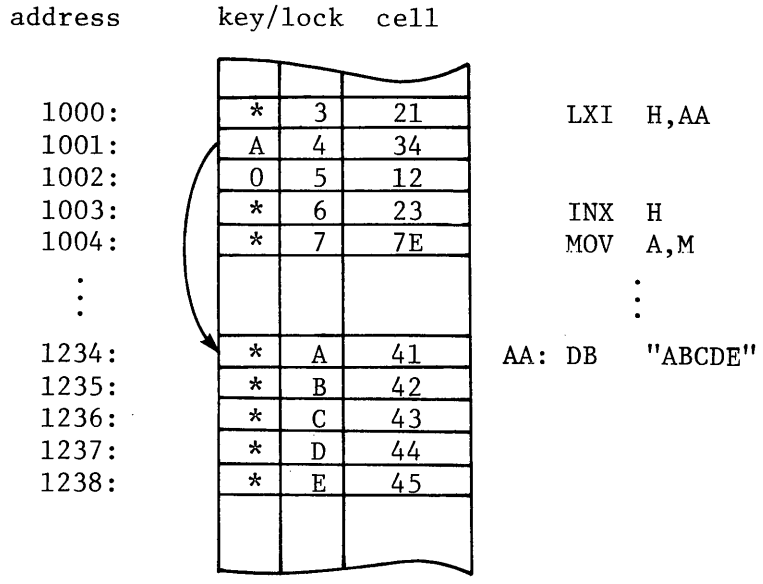


Figure 4. An example of object codes of Program P1.

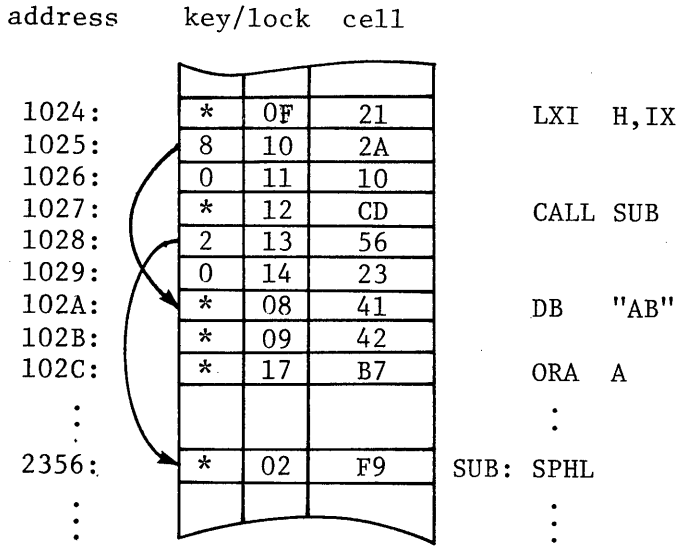


Figure 5. Procedurewise protection scheme and parameter passing.

INSTITUTE OF INFORMATION SCIENCE AND ELECTRONICS
UNIVERSITY OF TSUKUBA
SAKURA-MURA, NIIHARI-GUN, IBARAKI, JAPAN

REPORT DOCUMENTATION PAGE	REPORT NUMBER ISE-TR-82-26

TITLE

Software-fault detector for Microprocessors

AUTHOR(S)

Kozo Itano

Tetsuo Ida

REPORT DATE	NUMBER OF PAGES
January 15, 1982	27
MAIN CATEGORY	CR CATEGORIES
Computer systems	6.2, 6.3

KEY WORDS

software reliability, run-time hardware support,
key-lock memory protection, incremental protection code,
array bound checker, program module protection.

ABSTRACT

For the realization of the means to develop more reliable software for the microcomputers especially in real time environments, we design a hardware tool called "software-fault detector" which detects software faults such as misaccess to an element beyond the range of an array. The implementation of the mechanism for such address range checks is generally difficult in microprocessor environment, since internal registers are not readily visible to external logics. We introduce an "incremental" key-lock protection scheme into the fixed microcomputer architecture of the INTEL 8080 because of its popularity and simplicity. In this scheme, a "lock" is a protection code associated with the storage cell, and a "key" is associated with access capability such as address. In each memory access to a cell, a check is made whether a key matched against the lock of the addressed cell. In this paper, we present the details of the scheme and its analysis. Further, we present an actual hardware design of the software fault detector. Our design methodology is to realize a dector by the use of identical microprocessor 8080s, as an independent one-board module which could be connected to the memory bus of the host system.

SUPPLEMENTARY NOTES