DESIGN PRINCIPLES FOR A HIGH-PERFORMANCE SYSTEM

H. Schorr
IBM Thomas J. Watson Research Center, Yorktown Heights, NY

An exploration was made of the essential principles of a system design for executing large scientific programs. The design included a powerful highly-concurrent CPU, new channels, a storage hierarchy with relocation and overlay hardware, a new maintenance architecture and new software. The software included a global-optimizing compiler designed to produce code which realized the full potential of the machine. The operating system was split into two parts: the nucleus which contains all of the privileged code; and the peripheral service part. The objective of the nucleus was to provide primitive multiprogramming capability for an on-line data-base environment with performance comparable to ad-hoc user coding.

INTRODUCTION

This paper describes the results of an exploratory design effort which was carried out in order to develop new methods of radically improving system performance especially on large scientific calculations. Heavy emphasis was placed on simultaneously addressing software and hardware problems both at the system and logic-design levels.

In addition to high performance on scientific problems, it was assumed that good performance was also required in batch, time-sharing, and on-line data-base environments. Moreover this level of performance must be directly realizable through the use of higher-level languages. The ability to modify the operating system and, in particular, to add varied and different I/O devices was felt to be important. Accordingly, the group set out to discover the design principles of a total system centered around a powerful CPU which was to be programmed mainly in higher-level languages. The system design turned out to include:

- A CPU with a high degree of parallelism
- An advanced memory system
- Auxiliary storage devices integrated into the system
- A maintenance architecture and availability strategy
- An extensible multiprogramming control program
- A language processing system featuring an optimizing compiler.

Presented at the Symposium on Computers and Automata
Polytechnic Institute of Brooklyn, April 13-15, 1971.

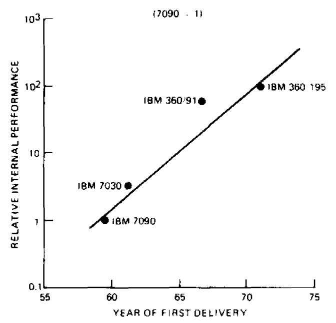


Fig. 1. Performance vs time of delivery.

II. THE CPU

If the average CPU performance is plotted as a function of time of first delivery, Fig. 1 results. For the mid 70s, then, it appeared appropriate to seek design principles that could achieve a CPU performance of 1,000 × IBM 7090 performance. CPU performance can be characterized as a function of circuit speed × number of circuits × architecture factor (Table I). It was assumed that

TABLE I. Architectural factor.

	RELATIVE PERFORMANCE	NUMBER OF CIRCUITS	CIRCUIT SPEED	ARCHITECTURAL FACTOR
IBM 7090	1	1	1	1
IBM 7030	3.5	3	1	1.2
IBM 360/91	60	13	4.2	1.1
IBM 360/195	96	13	4.2	1.7

circuit speed would improve by an order to magnitude over the 7090 so that the problem faced was to increase the number of circuits × architecture factor by two orders of magnitude over the 7090. This can only be obtained by employing some form of parallel execution—either: (1) extensions of the look-ahead principles of the IBM 7030 (Stretch) 360/91, 360/195; (2) vector or array processors; or (3) multiple instruction counters of some type. Use of the last type of parallelism *on a single* problem involved the solution of very difficult software problems. Vector and array processors were thought to be too specialized for effective use in general purpose areas. Therefore, attention quickly focused upon a single instruction counter machine with hardware controlled parallelism and an unconventional memory organization.

CPU Concurrency

The CPU was designed to achieve performance via concurrency in the operation of the following functions: instruction fetch, instruction decode, data fetch and index operations, arithmetic operators, as illustrated by Figure 2.

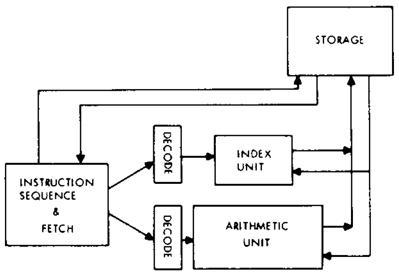


Fig. 2. CPU.

Each of the above operations themselves involved a high degree of concurrency: fetch multiple instructions—up to eight per cycle, decode multiple instructions—up to 16 per cycle, multiple load-store, index operations—up to three started per cycle, arithmetic operations—up to three started per cycle, process one branch instruction per cycle.

In the exploratory design, up to 50 different instructions could be decoded and in process concurrently (not counting storage). Multiple decoding was a new function examined by this project. By multiple decoding it is possible to initiate the execution of three index-type operations and three of the next eight arithmetic or logical-type instructions in every cycle. Eight arithmetic-type instructions are examined in each cycle to determine three executable instructions so that an instruction held up due to logical dependencies, register conflicts, memory interference, etc., does not impede the execution of any logically-independent instructions that follow it. This is especially useful in loops where the last instructions of the loop are usually dependent upon previous instructions, but where the instructions at the beginning of the loop are usually independent of those at the end.

High levels of arithmetic performance were obtained by having seven independent functional units which permitted different operations to be performed in parallel. Similarily, pipelining within the adders and multiplier [1] permitted three operations of the same kind to be in process concurrently by each one of these functional units; the arithmetic performance is summarized in Table II and the bussing given in Figure 3. For scientific calculations, double precision, in particular double precision inner product, was designed to be executed as fast as possible. Thus the bussing permits a single-precision multiply with double-precision product and a double-precision addition to be started on every cycle.

TABLE II. Summary of arithmetic performance.

	10^6 operations per second.		
FUNCTION	IBM 7090	IBM 360/91	EXPLORATORY DESIGN
Floating-Point Add	0.07	15	160
Floating-Point Multiply	0.04	6	80
Floating-Point Divide	0.03	2	12
Arithmetic Rate	0.046	11	240

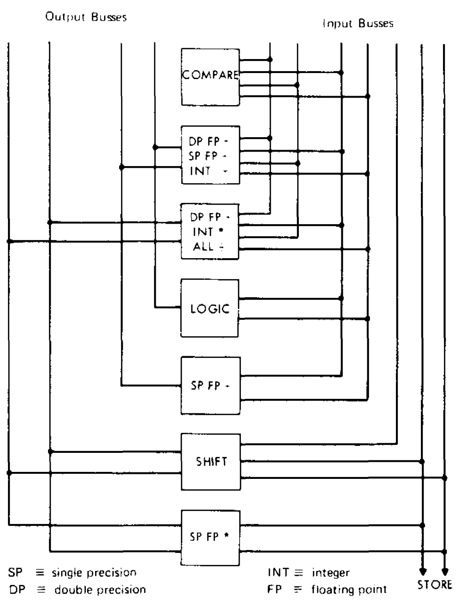


Fig. 3. *A*-unit bussing.

In addition, an attempt was made to design the bussing to minimize interference between instructions.

To supply the arithmetic unit with operands and to do the address arithmetic, a powerful index unit was designed. This unit consists of the functional units shown in Fig. 4, all of which can be pipelined.

Since a total of three index arithmetic and three floating/fixed-point operations can be initiated per cycle, a peak rate of six per cycle results.

Registers

To maintain an adequate supply of operands to sustain parallelism, the design included 31 arithmetic registers and 31 index registers. These arithmetic registers:

(1) Permit overlap of memory access with execution
(2) Hold data for multiple use
(3) Hold temporary results, constants, etc.
(4) Permit the compiler to schedule for maximum concurrency.

In addition 31 arithmetic backup registers provide further nonprogrammable operand buffers.

The number of indexed registers was chosen to permit:

(1) Permanent assignment to be made over an entire (sub) program
(2) Parameters to be left in registers to provide efficient subroutine and operating-system linkage
(3) Liberal use for basing purposes.

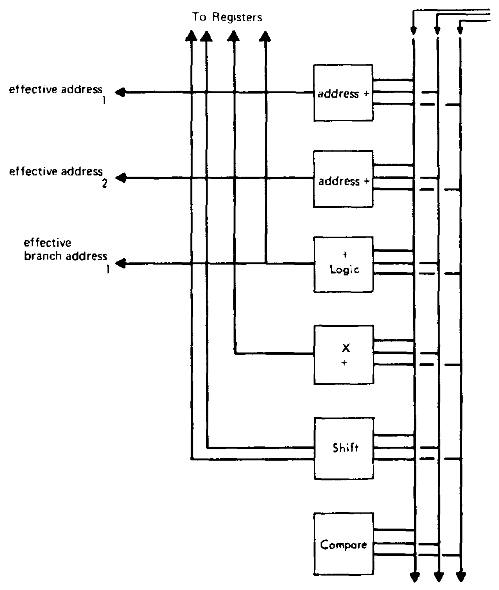


Fig. 4. X-unit bussing.

Branch Problem

In a highly-parallel machine, a branch instruction can cause a major loss of performance if the instruction-sequencing mechanism does not know which stream of instructions to prepare. This design proposed to overlap most branches via a combination of hardware and compiler by:

(1) Branch anticipation in which a conventional branch is replaced by a prepare-to-branch instruction and an exit instruction as in Figure 5
(2) Reduction of branches and exits
  (a) Branch instructions which test any function of any two condition registers
  (b) A skip instruction which causes programmer-marked instructions to be skipped at the arithmetic and index units rather than at the instruction-sequencing unit (which usually results in not delaying instruction fetching and processing)
(3) Reduction of the average branch delay via
  (a) Instruction-buffer registers which hold up to 96 instructions and which are associatively searched for a branch target
  (b) Instruction look-ahead on branches (prefetching of instructions on branch paths controlled by the history of previously-taken branches)
  (c) Index-unit instruction of a program being executed ahead of the arithmetic instructions (so that index-conditional branches can be executed in advance)
(4) Multiple-condition registers (24) which allow many tests to be made out of order with results prestored. Condition registers can be manipulated and combined by instructions.

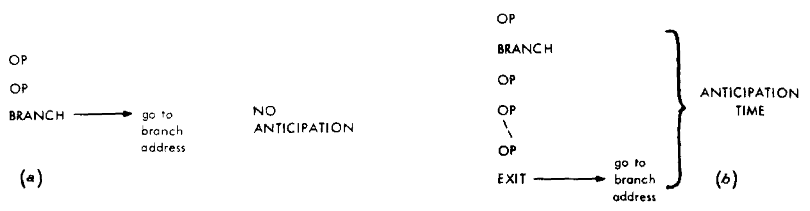
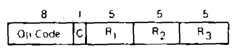


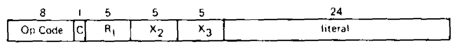
Fig. 5. Software-branch anticipation. (a) Conventional branching structure. (b) New branching structure.

Item (3) above discusses purely-hardware aids to solving the branching problem: utilization of the other three features required adroit programming or a compiler which could properly analyze higher-level-language programs. Indeed, the proposed compiler moved branches backwards in the code as far as possible, combined logical parts of IF-type statements into as few branches as possible, pushed tests forward in the code for early execution, etc.

INSTRUCTION SET (225 INSTRUCTIONS)



(a) Typical short instruction format



(b) Typical long instruction format

	Registers sources	Registers sink	Format	Variations
LOAD STORE	AXS	AXS	LS	MULTIPLE; TRUE INDEXING, LOAD AND COUNT HALF WORD
MOVE	AXCST lit	AXCST	LS	
FLOATING POINT ARITH.	A	A	LS	UNNORMALIZED, NORMALIZED OR NORMALIZED & ROUNDED; SINGLE DOUBLE OR MIXED
INTEGER ARITH	A	A	S	CONTINUED; SINGLE, DOUBLE OR MIXED
INDEX ARITH	X	X	LS	
COMPARE	AX lit	C	LS	MAGNITUDE, INTEGER, FLOAT PT., OR FLOAT PT. DOUBLE, BYTE OR MULTIPLE BYTE
SHIFT	AX lit	A X	LS	ARITHMETIC, LOGICAL SWAP, INSERT
LOGICAL (all binary)	A X C	A X C	S	
COUNT	A X	A X	S	TOTAL, LEADING ALIKE, LEADING DIFFERENT
CONVERT	A	A	S	PACK ⟷ UNPACK (FLOAT PT.) FLOAT PT. ⟷ INTEGER

CODE	FORMAT
A - Arithmetic	L - 48 bit instruction
S - Special	S - 24 bit instruction
T - IO Transmission	

lit - 24 bit address part of instruction

(c) Instruction types

Fig. 6. Instruction set.

Instruction Set

The word size was set at 48 bits. Two interaction formats were used: half-word and fullword. The proposed instruction set featured a 24-bit address literal, double indexing, index-literal loads, three-address register-to-register instructions and a skip bit. Both the index and arithmetic operations include a complete set of logical and shift operations along with operations for direct communication between the units. Double and extended-precision (via hardware and programming) integer operations were included as well as halfword floating-point loads and stores. Execute, RX, VFL and decimal instructions were not included due to the implementation difficulties; the instruction set is summarized in Figure 6.

Performance

CPU performance was simulated on two problem kernels, one a straight arithmetic-limited problem, and the second a problem which was previously branch limited are shown in Figures 7 and 8. An average performance of two instructions per cycle achieves the goal of 1,000 × 7090 average. This was usually the minimum achieved on all kernels. Note that higher peak capabilities are needed to achieve this average.

Problem Specifies

27 variable names

47 + - × ÷

1 decision point

Arithmetic Critical Path

4 +'s

3 ×'s

1 ÷

Relative Performance

7090	1
360 91	110
Exploratory Design	2500

Lagrangian Hydrodynamics Calculation

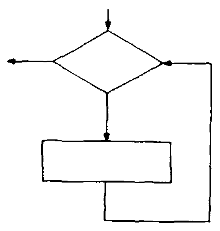


Fig. 7. Arithmetic kernel.

Interrupt

An interrupt can be considered as an asychronous branch, and thus it cannot be anticipated. It requires the current program to be stopped such that it can be restarted and the new program initiated. The system described above has as many as 50 instructions in process; an immediate stop requires a large amount of status to be saved. Rather than do this it was proposed that the hardware convert an interrupt to a branch, thus allowing instructions in process to be completed while the first instructions of the interrupt program are fetched. The cost in lost CPU processing was only on the average ten instructions while response

Problem Specifies

23 variable names

22 + − x ÷

11 decision points

Arithmetic Critical Path

8 +'s

4 x's

1 ÷

PERFORMANCE COMPARISON

	7090	360, 91	Explor. Design
Total Storage Ref.	139	57	14
Instructions/cycle	.26	.4	1.8
Time (μ sec)	643	9	.4
7090 Improvement	1	72	1608

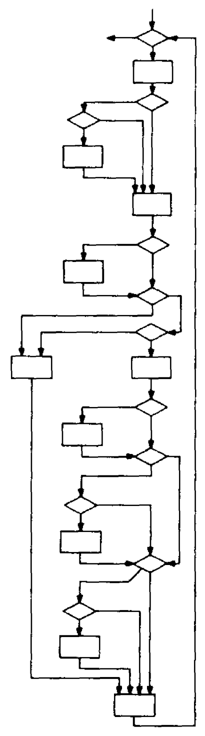


Fig. 8. Branch-limited kernel.

time could be of the order of ¼ μsec. In addition to this soft-interrupt facility a hard interrupt was included to cause an immediate halt to all instructions. This was necessary primarily to permit software action on address exceptions, etc., so that, for example, paging to large bulk memories or drums could be implemented.

Special Registers

To further aid the handling of interrupts and also for supervisor calls, the set of status and control bits that define the state of the machine were to be held in twenty 24-bit registers called special registers. Their function is given in Table III. Most of these could only be accessed while in privileged mode. The special registers permit fast switching between a problem program and the supervisor. By means of the alternate keys, programs could share storage-protected files.

I/O Module

The system was designed to work with standard System/360 I/O devices. A channel response to an initiation/termination sequence can take up to 32 μsec

TABLE III. Special registers.

NUMBER	NAME	LENGTH IN BITS	PRIVILEGED
0	Condition	24	no
1	Problem Exception	21	no
2	Problem Mask	21	no
3	Supervisory Exception	11	yes
4	Supervisory Mask	11	yes
5	Problem Normal Key	12	yes
6	Problem Alternate Key	12	yes
7	Supervisory Normal Key	12	yes
8	Supervisory Alternate Key	12	yes
9	Interruption Return Address	24	yes
10	Effective Branch Address	24	yes
11	Machine State	16	yes
12	Cycle Count	24	yes
13	Instruction Count	24	yes
14	Timer	24	yes
15	External Signal	24	yes
16	General Purpose	24	yes
17	General Purpose	24	yes
18	General Purpose	24	yes
19	General Purpose	24	yes

and it was decided not to have the CPU idle during such a sequence. The proposed solution was to add an I/O module which buffers initiation/termination sequences and thereby execute them concurrently with CPU computation as shown in Figure 9. This required the operating system to keep track of the fact that, for example, a Start I/O has been issued to a channel, but that it has not yet been accepted by the channel. Data was communicated to or from storage

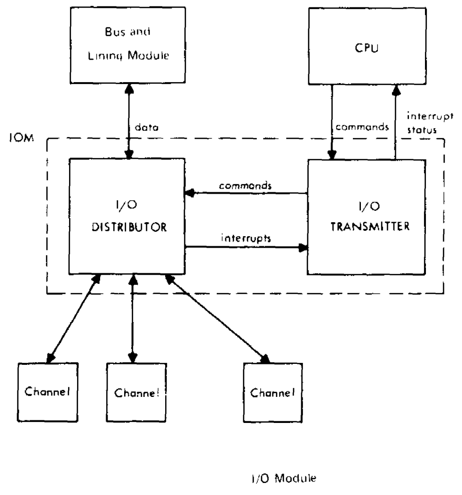


Fig. 9. I/O module.

without either CPU or I/O-transmitter (IOT) action so that both were free. On termination, the channel signalled the IOT which collects channel status before interrupting the CPU.

III. COMPILER

The CPU was a highly-concurrent CPU and its proper programming was ensured only if all of its features—multiple registers, arithmetic-functional units, condition registers, etc., were used properly. While this can be done by the assembly-language programmer, it was felt that the major amount of both the systems and application programming would be done in higher-level languages. At the start of the project, an effort was mounted to provide an optimizing compiler that would realize the full potential of the machine design. Such a compiler should also reduce global-execution redundancies and produce code comparable in size and execution time with hand-written code while producing correct answers of the same accuracy with no new side effects. It was also realized that a quick compiler was needed for debug runs, one-shot problems, etc.

Compiler Structure

It was readily apparent that an optimizing compiler for this machine was a major undertaking and that it should not be duplicated in full for every high-level language. It was therefore decided that the optimizer should work from a common internal language (IL) and that a front-end translator be added to the system for each language as required (see Figure 10). The proposed internal language was very machine dependent—its operators and operands were equivalent to machine operations, but there were no restrictions as to the number of registers that may be used. In addition, the IL was convenient for processing and contained: (1) information to make it traceable; (2) Relative data locations and extents; and (3) Any optimization directives that were included by the programmer. By having an IL it was felt that good code for many languages could be obtained by the relatively-easy process of adding a translator; this made the overall system more efficient. Moreover, it might be possible to combine programs (perhaps originating from different source languages) at the IL level. It also provided a standard base for other tools such as debugging statements, flow diagramming, etc.

It soon became apparent that for efficiency reasons, instructions should be rearranged to avoid bottlenecks and not presented in the unaltered order given by the output of, for example, a right-to-left scan. It was also observed that many of the optimizations in compilers, e.g., common-expression elimination, are basically independent of the target machine. Thus the optimizer was divided into two parts: one machine independent, and one machine dependent, as depicted in Figure 10. To minimize the interaction between parts, it was soon found that an adequate supply of registers was needed.

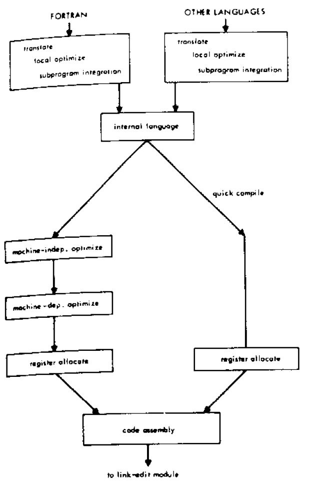


Fig. 10. Compiler.

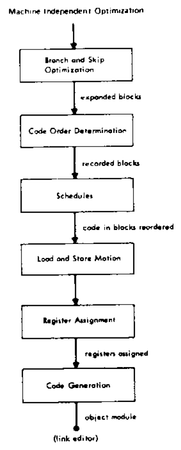


Fig. 11. Machine-dependent optimizer.

Machine-Independent Optimization

The major goal of the machine-independent part was to apply all optimizations to an entire program and not only to an inner loop [2,3]. The first part of the machine-independent optimization was to determine the basic topology of a program in terms of basic blocks of code (one input point and one exit point). Optimization then proceeds from the innermost block outwards. Optimization utilized include global common-subexpression elimination, code motion, strength reduction (e.g., changing subscript-expression calculations to index-register increments in loops), constant propagation (index variables replaced by constants wherever possible), and dead-variable elimination (the deletion of unused definitions and instructions). The sequence of machine-independent optimize operations was as follows:

(1) Basic block finder
(2) Eliminate common subexpressions in basic blocks
(3) Propagate constants of singly-defined variables
(4) Control flow analysis
(5) Global common-subexpression elimination and code motion
(6) Chain uses and definitions globally
(7) Reduction in strength process
(8) Global-constant propagation
(9) Dead-variable and expression elimination.

Machine-Dependent Optimizations

Machine-dependent optimizations included:

(1) Skip and branch analysis
(2) Scheduling
(3) Register allocation.

```
W = G + M * K * R * (U + Y + T * B (Z + H + C * N * (X + F + D * J)))
L = P * E * V/P
O = L * A + V * E/J
```

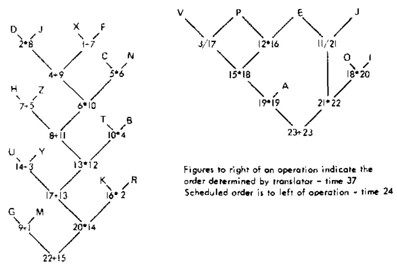


Fig. 12(a). Dependency tree of a sample problem.

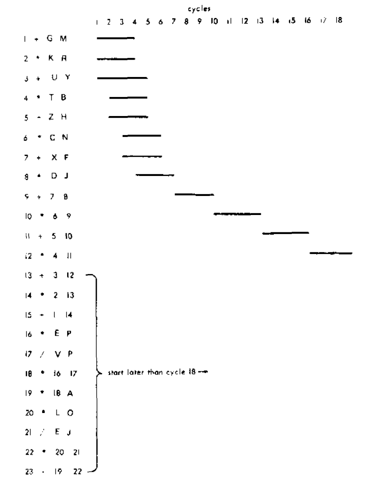


Fig. 12(b). Unscheduled sequence

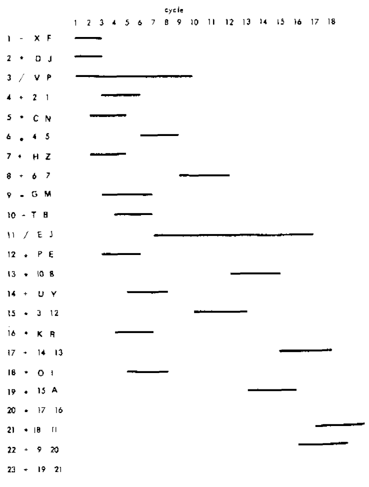


Fig. 12(c). Scheduled sequence.

In skip and branch analysis, basic blocks are merged to improve the schedule and eliminate transfers, and blocks are reordered to reduce transfer frequency. In scheduling, code sequences are ordered to minimize execution time. Finally, register limitations are considered and the necessary loads and stores introduced. A flowchart is given in Fig. 11 and the result of scheduling a problem shown in Figure 12. Other optimizations, especially those to smooth subroutine linkages, will be included.

FORTRAN Language Extensions for Systems Programming

Early in the study the question of a possible implementation langage for programs was considered. At the time this decision was made, code produced by PL/I compilers was not suitable. The decision was made to use an ASA FORTRAN which: gave access to most machine facilities; permitted more-complex data structures and part-word variables to be handled; and, interfaced with many primitive operation-system services. Specific features included: machine operations using symbolic variables (not registers), partial-word variables, storage allocation and hierarchy-level control, memory-mapping control, (e.g., interleaving, overlay, boundary alignment, partial-word packing), macros, new I/O, based variables, optimization directives, subroutine-interface control, recursive subroutines, debugging statements, and expanded definitions of expressions (if, then, else, $\cdots$).

IV. STORAGE HIERARCHY

As the proposed CPU was to operate in a general-purpose environment, it required support by a large auxiliary storage with an adequate data-transfer rate. Based on a study of time-sharing installations, 7090 job shops, and 7030 scientific usage, an average of one bit of I/O is required for each CPU instruction executed. Assuming that the goal of 1,000 times the 7090 was achieved, and assuming that the 7090 executed up to 160,000 instructions per second [4], then the system would require 160×10^6 bit/sec on the average. This represents the total average I/O required by the system, and consists of: (1) System programs; (2) Temporary storage (intermediate results); and (3) The jobs themselves (procedures and data).

In modern systems the first two components of the I/O can be supplied from drums or local core storage (LCS) in which the systems programs and intermediate results are stored. It remains, therefore, to introduce the job I/O from lower-level storage. How large is this component? This is unclear at this time, but it was estimated that about $\frac{1}{4}$ of the total average I/O, 40×10^6 bit/sec, should be an upper bound in scientific installations. If tapes alone are used to supply the job I/O, then a total of 80 units in continuous usage (mount, read, demount at top speed) would be required. The operators' problems in an installation such as the above would be immense. More importantly, conversational and real-time jobs require a response time better than that provided by an operator-serviced library. Reduction of operator intervention requires an increase in the size of on-line auxiliary storage. The required capacity of this on-line storage was estimated to be 10^9 words.

Staging

Storages capable of economically providing this capacity are characterized by low data rates (400K byte/sec) and slow access time ($\frac{1}{20}$ to 5 sec). If the CPU is directly multiprogrammed from such storage devices, a large number of jobs and care are involved. For example, consider a simple model involving identical jobs each of which executes 250,000 instructions between I/O requests for 25K bytes of data. As the CPU was assumed to perform 160×10^6 instruction/sec, it would have required the multiprogramming of at least

$$\left(\frac{1}{20} + \frac{25,000}{400,000}\right) \div \left(\frac{250,000}{160,000,000}\right) = 72 \quad ,$$

jobs to keep the CPU fully utilized even if no allowance is made for I/O interference. Assuming each job to require 200K bytes of electronic storage for program and data, this implies a core storage requirement in excess of

$$72 \times 200{,}000 = 14.4 \times 10^6 \text{ [bytes]} \ .$$

Appreciable reductions in this excessive core-storage requirement could be achieved only by significant improvements in I/O-access times and transfer rates. Devices with these improved characteristics such as fixed-head files (FHF) were far too expensive to use for the entire on-line storage of 10^9 words, however. This led to a multilevel on-line storage design in which jobs were *staged* from lower-cost devices to fixed-head files such as the IBM 2305 before moving to core.

Note that it is necessary to use much-larger blocking factors when moving data from lower levels to the FHF then when moving from the FHF to core. This is easily seen if we again use the simple model of 25,000 bytes per I/O request after the execution of each 250,000 instructions. If data are moved to the FHF in blocks of 25,000 bytes and used only once, then the number of accesses to lower-cost online storage is required to be at least

$$\frac{160{,}000{,}000}{250{,}000} = 640 \text{ per second} \ .$$

This access rate could not be economically satisfied by devices with access time in excess of $\frac{1}{20}$ second and larger blocking factors were thus required.

Core storage is required to buffer transfers from lower levels of on-line storage to the FHFs. The costs of staging, therefore, include:

(1) Core storage capacity for buffering
(2) Increased data flowing to and from core storage.

In general, the design of a multilevel-storage hierarchy is a complex multi-parameter-optimization problem involving the following: number of levels, capacity (and therefore cost) of each level, data transfer rates between levels, blocking factors, access rates, organization of hardware and software to control transfers between levels.

The storage hierarchy which was settled upon as a design goal is given in Figure 13. Note that two levels of electronic storage were specified: a small high speed storage (HSS); and a larger, slower, main storage (MS). This decision was made for reasons analogous to those which led to the use of multiple levels of on-line storage. The following comments all apply to the hypothetical hierarchy of Figure 13.

(1) Use of FHFs reduces the level of multiprogramming (and hence the electronic storage required for job buffering) by a factor of three approximately
(2) System programs are specified to be core resident
(3) The FHF should be used in a sector-queuing mode with fixed-length

sector sizes. That is, accesses must be sorted by rotational position as in Figure 14 and multisector requests made. If this is done, a queuing delay (including rotational delay) of

$$\frac{1}{2} + \frac{\lambda}{2(1-\lambda)} = \frac{1}{2(1-\lambda)} \text{ revolutions} \quad ,$$

is introduced where λ is the request rate specified as a fraction of the maximum data rate of a FHF reading continuously.

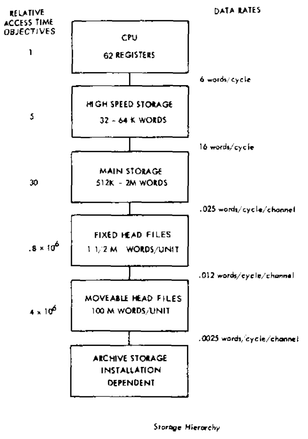


Fig. 13. Storage hierarchy.

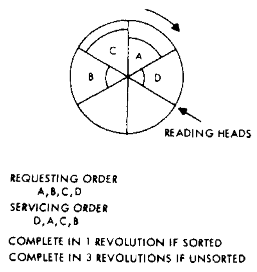

C
A
B
D
READING HEADS
REQUESTING ORDER
A,B,C,D
SERVICING ORDER
D,A,C,B
COMPLETE IN 1 REVOLUTION IF SORTED
COMPLETE IN 3 REVOLUTIONS IF UNSORTED

Fig. 14. Queue sorting.

Thus for $\lambda = 0.5$, to obtain a page, a delay of one revolution results. A plot of delay vs data rate is given in Figure 15. The same curve for FIFO-access method is also plotted and shows the much-greater data rates obtained by sorting requests. To benefit from such a strategy requires the operating system and problem programs to issue multiple-sector requests. Three fixed-length sector sizes were to be accommodated by the proposed operating system: they are 256, 1024, and 4096 words.

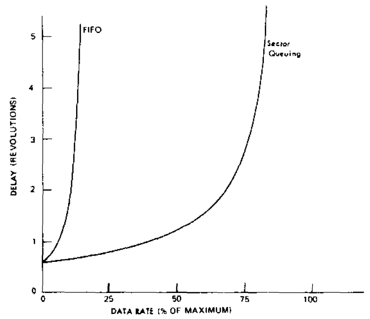


Fig. 15. Fixed-head file (FHF) queuing.

HSS-MS Storage Allocation and Addressing

The HSS was to have been relatively small but would have had the best access time of any storage; for efficiency the CPU must execute problems primarily out of HSS. The number of programs that have to be multiprogrammed requires a large amount of storage that can be economically supplied by using MS. The use of a two-level storage introduces several problems:

(1) Two addressing structures
(2) Two memories to allocate
(3) Transfer between HSS and MS must be done efficiently
(4) The overlay problem for programs greater than the size of HSS.

The system proposed to avoid these problems by using hardware that makes the two-level system look like a one-level system [5]. Storage is connected together by a buss and lining module (BLM) as shown in Figure 16. The BLM provides *virtual addressing* of HSS-MS: the address used to identify data is independent of the location of that data in the storage system; it also provides *storage protection.*

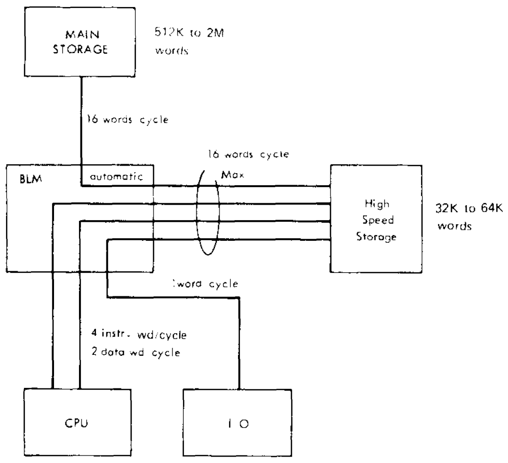


Fig. 16. Bus and lining module (BLM).

The addressing structure is shown in Fig. 17 and it should be noted to both CPU and channels.

Virtual Addressing

The set of virtual addresses is logically divided into 64-word pages. Any page may occupy *any* 64-word segment of MS. Pages are allocated in MS under the control of the operating system. Another storage, located in the BLM and called the directory, associates virtual to physical addresses. The directory works in units of 1, 4, 16, or 64 pages (64, 256, 1024, or 4096 words).

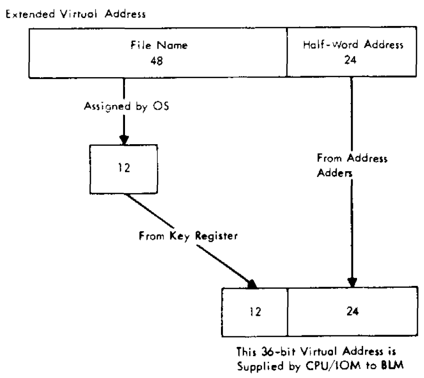


Fig. 17. Addressing structure.

Lining

The unit of transfer between HSS and MS is not a whole page but part of a page called a *line*. Since only part of a program is required at a given execution of a task, transferring only this required part of a program cuts down on the amount that must be transferred between HSS and MS. If the line size is one word then only what is needed is transferred and the effective program size is at a minimum. Unfortunately, hardware and programming constraints tend to require a larger line size. To reduce interference and utilize bandwidth the line size should be a multiple of the physical word size of MS. Similarly, it seems desirable to be able to utilize sequential data from MS. If the data is being processed at one word per cycle and the net access time to MS is c CPU cycles, then c words should be fetched from MS at a time. To minimize hardware, the line size should be large, i.e., the amount of control, and the size of the table which identifies the line in HSS is inversely proportional to line size. A line size of 16 or 32 words seemed to achieve the right balance; 32 words was chosen.

Assignment of Lines to HSS

The two extreme ways of assigning lines to HSS are: (1) Restricting a line to one fixed 32-word slot; or (2) Allowing it to occupy any available 32-word slot. The former results in too many conflicts due to lines competing for the same slot while the latter requires either an associative memory or the search of a large association table before accessing data. These solutions are too slow and expensive. The proposed choice was to allow a line to reside in either of two fixed 32-word slots; the storage utilization closely approaches that of an associative memory. This permits *simultaneous* access of both the data and the association table without excessive conflicts; raw HSS and bussing performance is degraded by no more than 20%. However, the HSS bandwidth required is doubled. The lining and MS to HSS transfer (swap) mechanisms are illustrated in Figures 18 and 19. The choice of which line to be swapped is based on a hot/cold replacement algorithm.

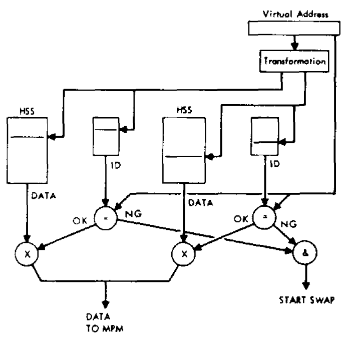


Fig. 18. Lining mechanism. Four accessing mechanisms operate concurrently: 2 for data fetch or store; 1 for I/O fetch or store; and 1 for instruction fetch.

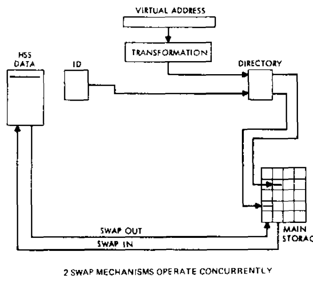


Fig. 19. Swap mechanism.

Size and Characteristics of HSS

The HSS would have had some unique physical characteristics; among them were that the cycle time is the same as the CPU cycle time. This permits a storage request to be made to a module on every CPU cycle and thus a 16-module system permits 16 requests per cycle to be serviced if there are no conflicts. The proposed system could make 14 requests per cycle (7 logical requests). At this rate, a queuing delay of less than $\frac{1}{2}$ cycle results.

The size of the HSS was larger than the buffer stores provided in other systems. This is due primarily to two factors:

(1) Almost all the time (better than 99%) accesses must be made to lines already present in HSS or else performance falls off drastically. The attempt of the system is to have memory look like it has the capacity of MS with the access time of HSS

(2) I/O and its control resides in HSS and requires considerable space, approximately 10K. Thus HSS should be available in 32K and 64K word sizes.

V. I/O CONTROL AND CHANNELS

The efficient processing of I/O requires handling periodic arrivals of partial words, buffering them to some multiple of MS word size and then transferring them into MS. Rather than provide separate assembly registers for a large number of channels, it was better to add HSS capacity and let it be shared between I/O and programs; a line was selected as the I/O-buffer size. The transfer of data from buffers would then be carried out automatically by the BLM hardware via the line-replacement algorithm. The strategy of always accessing I/O through HSS also eliminates any copying back from HSS to MS before carrying out an I/O operation. I/O and its control is a very important operation in any system; the lines in HSS allocated to I/O are accessed with greater frequency by I/O than those accessed by other programs. This results in low probability of being swapped out and consequently I/O overruns were held to acceptable levels.

Channels

The proposed channels were to be more-directly controlled by program than in most IBM systems. That is, there is no hardware realization of software-implementable features such as chaining, queuing, or searching. Implementation of these features via program is facilitated by the availability of HSS and a very-powerful CPU. This means was chosen in order to reduce the channel cost. For example, if a large average data rate of between $\frac{1}{2}$ to 2 bits (depending upon the job mix) of I/O per instruction executed is to be accommodated, then for $1,000 \times$ 7090 and the higher I/O average-data rate, $1,000 \times 160,000 \times 2 = 320 \times 10^6$ bit/sec must be concurrently handled by the channels. The instantaneous rates would be higher. A factor of two in excess channel capacity is needed for staging and another factor of two is needed to limit queuing delay. Thus a total channel rate of 1280×10^6 bit/sec would be needed. Typical channels were limited to 10×10^6 bit/sec so that 128 channels would be needed. A high speed channel was designed which by transmitting a 48-bit word could achieve a maximum data rate of 400×10^6 bit/sec. Up to 64 standard and up to 32 of the above high-speed channels were allowed.

Greater use of program control permits a channel design utilizing minimal hardware; every four channels would share adders and control circuitry. In operation, each channel executes only a single command and then interrupts the CPU; chaining and queuing were to be provided in software. It should be mentioned that the study examined several alternate paths, including an I/O processor and high-speed channels which automatically handled queuing. These were rejected because of cost considerations and because they would have resulted in several other problems such as higher I/O overruns, storage allocation, cooperation in the use of commonly-accessed queues, the need for a separate I/O processor having its own HSS and operating system, etc. Analysis also showed that neither of these two approaches saved a great deal of CPU time, and in any case CPU time did not appear to be a limiting factor.

VI. OPERATING SYSTEM

Control over the I/O and the storage hierarchy in a multiprogramming environment were the key problems to be faced by the operating system. Also, the operating system would support terminal job entry, remote entry, real-time applications, new installation specified I/O devices and the ability for an installation to replace IBM-supplied systems programs by their own. It was felt that the best way to provide these features was to divide the operating system into two parts: a nucleus and peripheral services. The nucleus would consist of those services essential to the entire community of users, that is, the intersection of all their needs. This nucleus interfaced directly with the hardware and converted it into an extended machine in which users were isolated from one another but still retained sufficient control over the hardware so that they could maximize its use for the solution of their particular problems. The user, for example, was allowed to control the type of storage he would like to have pages of his file in and also control their transfer up and down the storage hierarchy with predictable response time.

The peripheral service routines were a user of nucleus services (and are no different from any other user) usually oriented towards providing maximum user convenience. For example, compilers, job-control-language interpreters, peripheral I/O, sequential and partitioned-access methods, etc., are peripheral service routines. These routines can be added to, replaced, etc., but any such changes are written in nonprivileged code and must obey the nucleus-interface conventions (as well as any peripheral-service conventions).

This rigid division of the operating system, as characterized in Fig. 20, provides for the separation of the users into two classes: those who are principally concerned with response and efficiency; and those concerned with convenience, adaptability, and versatility. The nucleus thus incorporated a strict set of rules and provided a set of services each of which had a predictable response time. Automatic-allocation schemes such as demand paging or data look-ahead buffering (as provided in sequential-access methods) would not be provided.

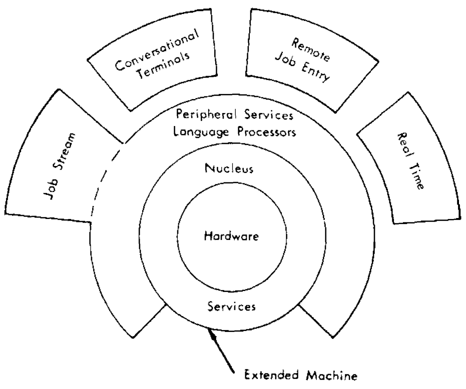


Fig. 20. Conceptual view of operating system.

Nucleus Services

The nucleus can be characterized as consisting of interrupt handling, task management, basic data management of the storage hierarchy and an I/O interface, and can be depicted pictorially as in Figure 21. The nucleus is entered only via SVCs and interrupts. The SVCs and interrupts are routed by the process dispatcher to the appropriate nucleus component (or to a task). The basic part of the operating system is the process dispatcher which has to operate disabled and was designed to operate with a minimum number of instruction executions. The logical consistency of the rest of nucleus is maintained by means of a set of locks on important facilities and nonreentrant pieces of code. An important part of the nucleus is the basic data management which catalogs, creates, destroys, and moves files within the storage hierarchy. A file was identified by a 48-bit file name. This file name was permanently associated with the file and was the only means by which a nucleus user could address it. The file name identified the file but did not refer *at any time* to the physical locations occupied by the file. The physical locations were specified in a *file index* which was maintained by the nucleus. Thus the name by which the programmer addressed a file did not have to be changed when movement between levels of storage hierarchy occurred and when a physical assignment was made within a level; this was also true for both MS and HSS. The use throughout the system of virtual addresses resulted in a simplification of the addressing structure since it did not reflect the levels of the hierarchy and it also provided for relocation throughout the system.

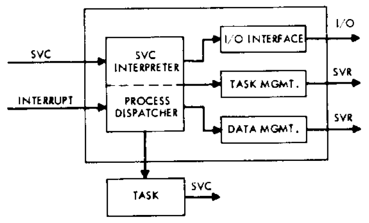


Fig. 21. Conceptual view of nucleus.

A file had as *attributes*:

(1) Its storage level within the hierarchy
(2) Its accessibility—whether it is read only or read/write.

These attributes could be assigned to *pages* or blocks of pages within a file.

In addition to a file with its physical addresses, the file index also specified other attributes of the file. The file index could be physically separate from the file itself. In general when active it should be on a faster device than the file itself so that a file access appears to be only slightly longer than the access time of the slower device. The collection of all file indexes constitutes the internal catalog which can be accessed *only by nucleus routines*. The basic nucleus data management services were: create file, move file index to storage level X, move data block to storage level X, copy data block X into data block Y.

Tasks

A task was considered to be an independent entity for the purposes of accounting and servicing requests. Tasks were created by other tasks *only* with no mother-daughter relationship as in OS/360. However, it was possible to have SVCs routed to another task rather than the nucleus. A task could communicate with another task via explicit commands and this same mechanism was used by the nucleus to communicate with a task. Each task is assigned to a *partition* which was assigned a certain percentage of the systems resources. This percentage was usually guaranteed to a partition and was not a static assignment, i.e., at any given instant a partition may have no resources assigned to it. A partition which at a given time has received less than its allotted amount of resources has priority over other partitions that have full allotments. Tasks were assigned priorities within partitions and were assigned resources on the basis of their previous utilization of those resources (a variation of exponential scheduling).

I/O Interface

The I/O interface includes the I/O drivers for members of the storage hierarchy, i.e., FHF, MHF, and archives. These devices are completely under the control of the nucleus, have fixed formats, and constitute the on-line data base. Other devices such as tapes, unit-record equipment, removable MHF disk packs, terminals, etc., cannot be completely under the nucleus' control because of the nature of the devices, data formats and their use which are usually unpredictable. These *external* I/O devices are controlled and protected by using the nucleus to interpret channel programs generated by the user. The nucleus replaced logical channels by physical channel numbers, translated data identifiers into MS virtual addresses, implemented System/360-like chaining and collected channel, control unit, and device status for return to the user. The nucleus did not take responsibility for correct use of the channel or error recovery.

TABLE IV.

MACHINE	EXTENDED MACHINE
Data Identification	X-Data Identification
12-Bit Keys	48-Bit File Name
24-Bit Logical Address	24-Bit Logical Address
Operations	X-Operations (SVCs)
Branch	(a) Direct X-Branch
	(b) Indirect X-Branch for Protection of
	● Entry Points
	● Reading
	● Modification
$\left.\begin{array}{l}\text{Load}\\ \text{Store}\end{array}\right\}$ MS to Reg.	$\left.\begin{array}{l}\text{Move}\\ \text{Copy}\end{array}\right\}$ MS to Aux. Storage
I/O OPS (Privileged)	Symbolic Channel Programs

Extended Machine

The user of the nucleus would see an extended machine (Table IV) consisting of: files, tasks, unprivileged machine operations, extended machine operations (nucleus service requests) implemented through SVC instruction, and symbolic I/O channels. This gave the user effective access to all system resources without using privileged code. All other systems programs and all user programs operated in this environment.

Peripheral Services

These programs were designed to support large production jobs concurrently with small foreground jobs. Jobs could be entered into the job stream via spooled I/O with multiple entry (card readers, terminals, tape). Peripheral service routines were categorized as extended-data management, extended-task management, and external-device support. The main emphasis of extended-data management was the support of higher-level languages; it contained routines to provide basic and queued sequential-access methods. It was planned that each user would have his libraries whose main purpose would be to identify his files and to perform translation of alphanumeric file-editing facilities. Extendedtask management included routines for user command-language interpretation, standard-error recovery and debugging procedures, and operating-environment supervision. The latter included gross-scheduling, accounting, operator, and job-stream supervision routines. External-device support routines included spooling, tape-handling, and terminal-support routines.

Additions to the peripheral service routines to support conversational terminals and remote job entry, would have consisted of additional language facilities, interactive-terminal control, peripheral-processor support, and telecommunication support.

VII. MAINTENANCE ARCHITECTURE

There are several areas of maintenance that were addressed as part of the architectural design of the system. These were:

(1) Error control—the determination that an error has occurred, how it has propagated, and prescription for its repair
(2) Protection of the operating system—protect essential parts of the operating system against error and, attempt to insure the integrity of the files; facilitate fast re-IPL
(3) Checkpoint/restart—provide user function so that the user can use additional measures for maintaining the integrity of his data beyond those that the system provided
(4) Availability—reconfiguration of the system when a hardware failure occurs to provide a lesser amount of computing capacity
(5) Repair—discovery and location of a faulty, replaceable part.

It was felt that because these areas strongly interacted with other system-architecture areas they must be included in the initial overall design.

Error Control

Hardware error control for I/O consisted of specifying burst-error correction for FHF and standard byte parity for System/360 I/O devices. For HSS and MS, byte parity was also specified rather than the single-error correction possible with extra memory bits. The latter was eliminated because it did not appreciably increase availability, it did increase cost, and it affected the design and performance of the CPU; furthermore it causes some errors to be miscorrected.

For the CPU it was felt that parity did not provide a complete solution to error control; parity could provide detection on the data paths but not elsewhere and at great cost in both circuits and performance. Other hardware devices were similarily rejected. It was decided instead to propose running periodic diagnostic tests under the operating system. However, byte parity was maintained from the I/O devices through the channels, I/O module, and BLM since both ends, storage and devices, maintained parity.

Some error control is obtained via the nucleus. When an error occurs the nucleus can replace a file by a copy of it which has been maintained lower in the storage hierarchy. In general such copies should be maintained until the CPU has finished processing the MS copy. At this point the space occupied by the lower-level copy is returned to the free list. Of course, if the file has been changed in MS then there is nothing the nucleus could do but notify the user. The user of course can increase the likelihood of correctability by explicit update of lower-level copies. Work on providing user checkpointing did not proceed past this point.

The protection afforded to user data also aids in the preparation of the operating system since most operating-system tables are cataloged as part of the user data. The nucleus was protected against catalog loss by keeping a transaction log and by providing (redundant) identification of the file with the file. Re-IPL is facilitated by keeping a copy of nucleus on FHF to ensure quick reinitialization. Most of the effects of system software bugs were isolated by the separation of the nucleus from the peripheral services.

Availability

Failure of the CPU, BLM, or I/O transmitter would have caused the system to go down. In a system with 64K of HSS and more than 256K of MS a storage failure would not have caused unavailability. The system operated with half of a 64K HSS disabled with no software adjustments required. Similarly a system can function with all but one 256K block of MS disabled. System transition to these states is facilitated by the use of virtual addressing. All FHF and other I/O devices should be accessible by alternate paths.

A duplex system in which CPU, IOM, BLM, and HSS were duplicated was studied in a preliminary way. This augmented system would have been designed to operate in a reconfigured mode less than 200 hours a year and would be unavailable about one hour a year based on typical hardware-failure projections.

Repair

High availability can be achieved partly through rapid and semiautomatic repair. A failed, replaceable part could be automatically found by using a maintenance computer to compare all the triggers of the machine with simulated values of these triggers. The maintenance computer would either run the simulation or communicate with a remote computer running the simulator (which could be an identical system). The isolation strategy is flowcharted in Figure 22. It was our judgment that this technique will isolate 93% of the hardware failures thereby reducing manual intervention and keeping the duration of unscheduled interruption (DUI) to approximately two hours as shown in Figure 23. The hardcore required to run diagnostics consists mainly of logout of all control triggers, input to the instruction buffer and I/O-module register, and the maintenance register as shown in Figure 24. It is the ability to keep this hardcore to a minimum which allows the failure-detection program to proceed without manual intervention and this, in turn, helps keep the DUI to two hours. The time to physically replace a failed component is in the order of ½ hr. The replaceable part would generally be large and complex, e.g., a BSM of storage or a board of logic.

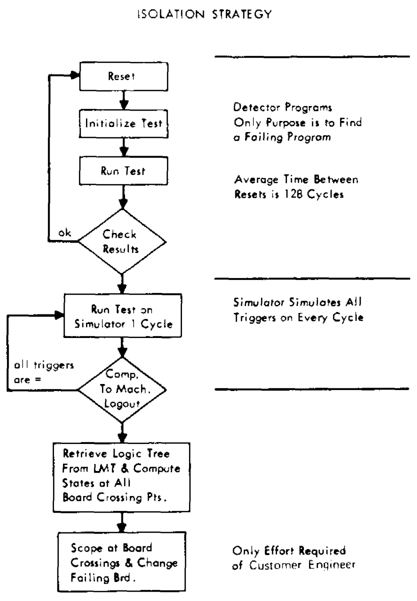

ISOLATION STRATEGY
Reset
Initialize Test
Run Test
ok
Check Results
Detector Programs Only Purpose is to Find a Failing Program
Average Time Between Resets is 128 Cycles
Run Test on Simulator 1 Cycle
all triggers are =
Comp. To Mach. Logout
Simulator Simulates All Triggers on Every Cycle
Retrieve Logic Tree From LMT & Compute States at All Board Crossing Pts.
Scope at Board Crossings & Change Failing Brd.
Only Effort Required of Customer Engineer

Fig. 22. Isolation strategy.

VIII. SIMULATION AND CHECKOUT SYSTEM

To aid in the design of this new system and the checkout of the software, various simulators were needed. Those used in the hardware design included an instruction simulator [6], a cycle-by-cycle CPU simulator, and BLM and storage

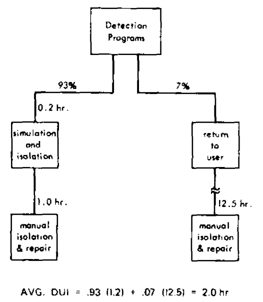

Detection Programs
93%
7%
0.2 hr.
simulation and isolation
return to user
1.0 hr.
12.5 hr.
manual isolation & repair
manual isolation & repair
AVG. DUI = .93 (1.2) + .07 (12.5) = 2.0 hr

Fig. 23. Duration of unscheduled interrupt obtained by reducing hard-core and isolation strategy of Figure 22.

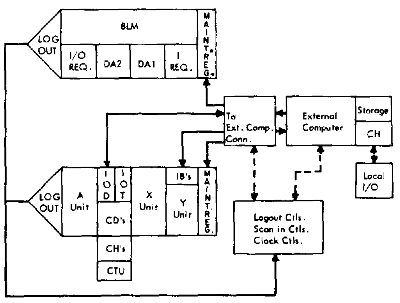


Fig. 24. Maintenance and data paths.

simulators. In addition, simulation of the FHF and of the storage hierarchy was performed at the system level. In the storage hierarchy simulation, a simple language was developed to describe job profiles in terms of storage requirements, I/O requests, and CPU instructions to be executed. These jobs were then processed by the system and various activities such as I/O and CPU busyness and storage requirements were plotted as a function of a time; in this manner both steady-state and transient behavior were studied. One result was that the scheduler should give I/O-limited jobs priority over CPU-limited jobs. That is, whenever an I/O request for an I/O-limited job is completed, the CPU should be switched to that job. Otherwise, it was found that I/O-limited jobs bunched up and eventually the system had no other jobs to work on. This resulted in the CPU becoming idle for long periods of time.

As software and hardware were being designed simultaneously, the only way to check out the software was to extensively debug and integrate it on a simulator, in this case on the System/360 under OS. Since the software itself (except for parts of the nucleus) was to be written entirely in Extended FORTRAN, the simulator had to include a compiler for this language. This compiler, called A360 FORTRAN, generated CPU code which served as input to a link editor, loader, library, etc., and finally the simulator itself. These were written in System/360 assembler and FORTRAN languages. The flowchart of the simulator is shown in Figure 25. The simulator control could interpret an elementary Job

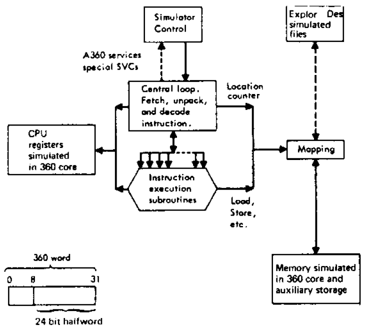

Simulator Control
A360 services special SVCs
Central loop. Fetch, unpack, and decode instruction.
Location counter
CPU registers simulated in 360 core
Instruction execution subroutines
Load, Store, etc.
Explor Des simulated files
Mapping
Memory simulated in 360 core and auxiliary storage
360 word
0
8
31
24 bit halfword

Fig. 25. Simulator flowchart.

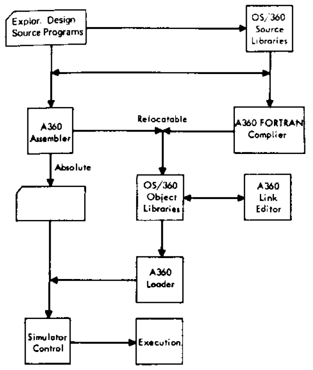


Fig. 26(a). Prenucleus operation of simulator system.

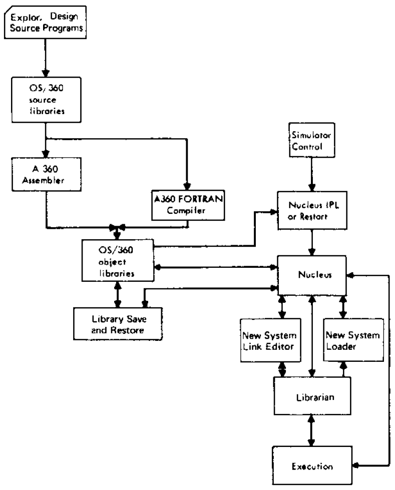


Fig. 26(b). Nucleus operation of simulator system.

Control Language (JCL) which allowed the initialization of a simulation in a specified state. The save, restore, etc., functions included permitted the results of a simulation at a given point of simulated execution to be saved and reinitialized later at the same point.

The use of the simulator was to proceed in an orderly manner; first the nucleus, the A360 FORTRAN compiler, and assembler were to be debugged. The entire system was to operate as shown in Figure 26(a). Rather than employ extensive scaffolding, it was decided to debug peripheral-service routines under the actual nucleus itself as depicted in Figure 26(b). This, while perhaps slower, would insure correctness of all interfaces and nucleus functions. The debugging of an extensive software package under simulation was felt to be feasible only when: (1) A higher-level language was used to write the software; (2) The simulator used was quite extensive; (3) The simulator would run under a modern system such as OS/360; (4) Most scaffolding was eliminated; and (5) There was more time to do the job than is usually the case.

CONCLUSION

The exploratory design of the system permitted all aspects of hardware and software tradeoffs to be studied by one group of people. This flexibility did not result in identifying as many functions in the hardware to aid the software as one's preconceived notions might have thought. For example, a machine to compile, simulate, and emulate was designed and rejected because of cost/performance and system-balance reasons. However, MS relocations, HSS lining, a large bulk memory (MS) and FHF record-ready features were considered essential hardware aids if a successful modern multiprogramming monitor was to be written. It was also noted that improved I/O (including an IBM 3330 with higher data rates) microfilm printers, and better archive devices would be needed to balance the system, particularly in noncomputation-bound environments.

The experiment was terminated after the major ideas had been fully explored and the knowledge obtained transmitted to product-development groups.

ACKNOWLEDGEMENT

The author was the manager of the group in which the major portion of the work described in the paper was performed. The ideas presented represent the contributions of all of these people. Particular mention should be made of F. Allen, R. Arnold, J. Beatty, P. Dauber, C. Freiman, C. Galtieri, D. Helman, R. Robelen, D. Rozenberg, and E. Sussenguth. J. Cocke, while not a member of the department, was a major contributor. The author's manager, J. E. Bertram, also deserves special thanks for his guidance and his technical contributions.

REFERENCES

[1] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J. Res. Devel.*, **11**, No. 1, 34-53 (January 1967).

[2] F. Allen, "Program Optimization," *Ann. Rev. in Automatic Programming*, **5**, 239-307 (Oxford, England: Pergamon Press, 1969).

[3] J. Cocke and J. Schwartz, "Programming Languages and Their Compilers," 2nd Edition, Courant Institute of Mathematical Science, New York University (1970).

[4] K. E. Knight, "Changes in Computer Performance," *Datamation*, 40-54 (September 1968).

[5] J. S. Liptay, "Structural Aspects of the System/360 Model 85 II—The Cache," *IBM Sys. J.*, **7**, No. 1, 15-21 (1968).

[6] D. P. Rozenberg and R. F. Savage, "A Proposal for the Computer-Design Process Based on Multilevel Simulation," IBM T. J. Watson Research Center, Yorktown Heights, New York, Report RC-3167 (December 2, 1970).