MSC-02553

[redacted]

LOGICON

los angeles / san diego / washington, d.c.

LOGICON

los angeles / san diego / washington, d.c.

Report No. CS-7107-R0216

February 26, 1971

Advanced Software Techniques for Space Shuttle Data Management Systems

Final Report

By

Raymond J. Rubey, Wayne A. Ganzel,
Michael D. Richter, and Steven A. Vere

For

Manned Spacecraft Center
National Aeronautics and Space Administration
Contract No. NAS 9-11225

LOGICON
255 w. fifth street, san pedro, california 90731

Approved

R. Dean Hartwick, Manager
Computation and Software Department

ACKNOWLEDGMENT

This study was carried out in cooperation with personnel from NASA/MSC EB5. The assistance of Irving Burtzlaff and Donald Barron is greatly appreciated.

TABLE OF CONTENTS

1. Introduction 1

2. Summary 3
   2.1 Conclusions and Recommendations 3
   2.2 Functional Analysis 5
   2.3 Computer Configuration 7
   2.4 Computer Architecture 10
   2.5 Programming Languages 12
   2.6 Verification Tools and Techniques 13

3. Functional Analysis 15
   3.1 Inertial Reference 17
   3.2 Navigation Sensors 20
   3.3 Displays 21
   3.4 Trackers 23
   3.5 Landing Aids 25
   3.6 Primary Propulsion Subsystems 27
   3.7 Reaction Control Subsystem 28

4. Computer Configuration 31
   4.1 Centralization/Decentralization Considerations 33
   4.2 Executive Functions 35
   4.3 Preliminary Executive Designs 39
   4.4 Executive and Computer Configuration Comparisons . . 59

5. Computer Architecture 63
   5.1 Memory 64
   5.2 Execution Speed 73
   5.3 Input/Output Facilities 77
   5.4 Instruction Set 79
   5.5 Word Format 86
   5.6 Register Organization 89
   5.7 Restart and Self-Test Provisions 93
   5.8 Interrupt-Handling Facilities 96
   5.9 Summary of Conclusions 100

6. Evaluation Methodology 103

7. Programming Languages 111
   7.1 Language Suitability 112
   7.2 Compiler Suitability 115
   7.3 Language Influence on the Software Development Cycle. 120
   7.4 Summary of Conclusions 124

8. Verification Tools and Techniques 127
8.1 Engineering Simulation 127
8.2 Interpretive Computer Simulation 131
8.3 Hybrid Simulation 139
8.4 Other Verification Tools 143
8.5 Master Test Plan 148
8.6 Summary of Conclusions 155

Appendix A - Task Interaction 157

Appendix B - A Methodology for Restart Protection 169

Appendix C - Proving Program Correctness 177

LIST OF FIGURES

1.	Onboard Software Structure Outline	31
2.	Implied Executive Concept	41
3.	Polling Executive Concept	42
4.	Polling Executive Design	43
5.	Interrupt Executive Concept	44
6.	Interrupt Executive Design: Interrupt Module	46
7.	Interrupt Executive Design: Scheduling Module	49
8.	Distributed Executive/Multicomputer Concept	51
9.	Distributed Executive/Multicomputer Design	52
10.	Distributed Executive/Multiprocessor Concept	54
11.	Fixed Executive/Multiprocessor Concept	55
12.	Floating Executive/Multiprocessor Concept	56
13.	Floating Executive/Multiprocessor Design: Scheduling Module	58
14.	Floating Executive/Multiprocessor Design: Interrupt Module	60
15.	Fixed Memory Costs	66
16.	Extensible Memory Costs	66
17.	Buffering Method of Interaction with I/O Controller	78
18.	Cycle-Stealing Method of Interaction with I/O Controller	78
19.	Instruction Usage for Guidance, Navigation, and Control Programming	80
20.	Stack Operation	92
21.	Performance-Probability Functions	105
22.	Performance-Payoff Function	109
23.	Basic Simulation Model	129
24.	Typical ICS Control Loop	133
25.	Hybrid Simulation, Diagnostic Configuration	142
26.	Absolute Value Sequence	147

LIST OF TABLES

1.	Summary of Computational Requirements for Three Space Shuttle Configurations	6
2.	Computational Requirements for Three Space Shuttle Configurations	17
3.	Computational Requirements for Strapdown Inertial References	18
4.	Computational Requirements for Navigation Sensors	20
5.	Computational Requirements for One Dynamic Display	23
6.	Computational Requirements for Trackers	23
7.	Computational Requirements for Landing Aids	25
8.	Computational Requirements for Primary Propulsion Subsystems	28
9.	Executive Design Comparison	61
10.	Basic Instruction Set	82
11.	Performance-Probability Matrix	106
12.	Language Capabilities Required for Different Software Functions	113
13.	Computer Influence on Language Suitability	114
14.	Language Characteristics for Different Software Development Phases	121

1. INTRODUCTION

This report describes a hardware/software study of the Space Shuttle data management system. The study purpose was to identify at an early stage in system development those computer hardware characteristics and software development approaches that will have substantial impact on software costs and schedules. The underlying motivation was to prevent software development from becoming the pacing item in Space Shuttle system realization.

The study addressed three aspects of system development. An analysis of the overall Space Shuttle objectives, mission requirements, and baseline hardware configurations determined the nature and magnitude of the computational functions that the data management system will be called upon to perform. Tradeoff studies investigated the software advantages and disadvantages of alternative computer hardware configurations and architectures; single-computer (or simplex), multicomputer, and multiprocessor configurations were examined, together with detailed architectural features that could be used in each. Finally, support hardware/software studies investigated means of facilitating flight software development. Two areas were addressed: high-order languages in which the flight software might be coded, and simulations and other analytic tools that might be used in verifying its correctness.

2. SUMMARY

This section very briefly abstracts the study conclusions and recommendations and then goes into somewhat greater supporting detail on the individual study areas.

2.1 Conclusions and Recommendations

The functional analysis revealed that some subsystem hardware configurations and system requirements pose unreasonable computational loads. The present overall range of computational requirements is extremely wide and must be narrowed if meaningful decisions are to be made as to the computer configuration. Recommendation: The functional analysis should be continued.

Available data permitted basic computational requirements to be estimated for maximum, moderate, and minimum subsystem hardware configurations. From these requirements it was concluded that a simplex computer would be adequate for the minimum computational load, but that a multicomputer or multiprocessor configuration would be required for the moderate or maximum loads.

The computer configuration analysis showed that software costs would be lower for a centralized computer organization than for a distributed or federated organization. It was further determined that effective separation of a program's functional elements could be accomplished by software methods in a centralized organization, just as separation would be accomplished physically in a federated organization. Recommendation: A centralized computer organization should be used.

Studies of executive types as part of the computer configuration analysis showed a polling executive to be simpler to verify than an interrupt executive, but not as flexible or responsive to likely system requirements. Analysis of verification problems indicated that control of interrupts and of interrupt levels could minimize the verification problems associated with an interrupt executive. Recommendation: An interrupt executive should be utilized, suitably restricted as to interrupt occurrences and levels.

Further investigation of verification problems indicated that a simplex computer configuration would permit the simplest and easiest software verification. Recommendation: If the minimum computational requirements prevail, a simplex computer should be utilized.

A multiprocessor configuration with a distributed executive was found to result in slightly simpler software than an equivalent multicomputer, and in much simpler software than a multiprocessor with a fixed or floating executive. Recommendation: A multiprocessor configuration with a distributed executive should be utilized in the event that the computational requirements exceed the capabilities of a simplex computer.

In analyzing the architectural aspects it was determined that those most significant in terms of software costs would be adequate memory, speed, and input/output margins. Recommendation: Memory, speed, and input/output margins of at least 40% should be provided in the hardware. Effective management controls will be required to prevent this margin from being utilized for new and possibly unneeded functions.

Software costs can be reduced by from 5 to 10% if other desirable architectural features are provided. Recommendation: The computer should have the following characteristics:

- Adequate instruction set, including floating-point arithmetic
- Hardware memory locks or their equivalent
- Extensive hardware error-checking, diagnosis, and correction facilities
- Save and restart facilities
- Interrupt-control facilities

Many additional architectural aspects individually have a small effect but in combination can significantly influence software costs. Recommendation: A computer having the features listed below on the left should be selected in preference to one having the characteristics listed on the right:

Desirable	Undesirable
• Read-only/read-write memory partitioning	• Auxiliary memory
• Uniform instruction execution times	• Virtual memory
• Buffered input/output	• Interleaved memory
• Partial-word addressing	• Limited addressing range
• General-purpose registers	• Unique or unusual instructions
	• Variable-length instruction and data formats

The programming language investigation showed that the use of a high-order language will reduce software development costs, even though the

language selected might not be completely suitable for all functions. The suitability of the compiler was found to be more important than that of the language itself. This is so because developing a completely efficient, correct, and comprehensive compiler for the Space Shuttle application will take a relatively long time. Portions of the flight software -- notably the executive -- will still have to be coded in assembly language, and verification will have to be performed at the assembly language level. Recommendation: A high-order language should be used, beginning as early as possible. Attention should be concentrated on compiler development rather than on further language development or refinement.

In the verification studies, the conclusion was reached that simulations will continue to be the most important verification tools. However, the anticipated speed, size, and complexity of the flight computer configuration will result in corresponding increases in simulation execution times, which, for interpretive simulations, will approach unacceptable amounts. Recommendation: Interpretive computer simulation diagnostic methods and outputs should be improved to maximize the information obtained from each simulation run. Further, hybrid simulation techniques should be extended to yield information now efficiently obtainable only from interpretive simulations.

Another conclusion regarding verification was that each type of simulation is suitable only for particular roles, and problems detected by one type may require the use of another for adequate diagnosis. Recommendation: A master test plan should be developed and implemented covering simulation use and comparison.

Finally, simulations can demonstrate the existence of software errors but cannot absolutely prove their absence. One technique was developed for designing programs whose correctness could be proved, and another for proving the correctness of relatively simple programs. Recommendation: These techniques should be further developed, and additional techniques should be investigated.

2.2 Functional Analysis

Because hardware definition was incomplete and changing during the functional analysis, it was not possible to establish firm and detailed quantitative measures of the computational load to be imposed on the data management system. The analysis did illustrate how otherwise reasonable subsystem hardware configurations and system requirements, notably those of the docking laser and the displays, can result in an unreasonable

computational load. Sufficient data were available to permit basic computational requirements to be determined for three possible Space Shuttle configurations:

- A maximum system able to satisfy all mission requirements and providing complete support and status monitoring for all associated subsystems
- A minimum system able to satisfy the basic mission requirements but providing for minimum onboard checkout and having many functions performed by hardware distinct from the central computer facility
- A moderate system providing capabilities midway between those of the two extremes

The primary computing requirements for these three configurations are summarized in Table 1. All requirements for the minimum configuration are within the capabilities provided by existing simplex computers. However, the computational speed required for the moderate and maximum configurations cannot be provided by either existing or foreseeable simplex computers: a multicomputer or multiprocessor would be required to provide the necessary effective computational speed.

For all Space Shuttle configurations, the table indicates only the minimum necessary capabilities. It does not include provision for expansion to support new subsystems and additional mission requirements, nor does it provide allowances for software inefficiencies. Even though the 24-bit word size would be adequate for the minimum configuration, a larger word size

Table 1. Summary of Computational Requirements for Three Space Shuttle Configurations

Parameter	Configuration		
	Minimum	Moderate	Maximum
Total memory size (thousand words)	43	120	250
Word size (bits)	24	32	32
Add time (microseconds)	25	.25	.10

would nearly eliminate the need for double-precision operations and hence is preferable. As regards memory size and computational speed, the architectural aspects studies indicated that adequate margins are the most important factors in determining software costs. Capacities exceeding the minimum values listed are mandatory if software costs are to be minimized. An allowance of 40% for the inefficiency component alone would permit use of a high-order language to simplify program coding and, in addition, executive design and programming standards that simplify software verification.

The range of requirements indicated for the three possible configurations is extremely wide. It must be narrowed so that decisions concerning the computer configuration can be made in a way that minimizes the cost and difficulty of software development. Therefore, the functional analysis should be continued through the completion of the flight software development.

2.3 Computer Configuration

The software problems inherent in a computer system in which most of the computational capability is concentrated in a central facility were first contrasted with those inherent in a system in which the computational capabilities are in large part distributed to the subsystems. In the centralized system, the relatively small amount of processing performed at the subsystem level would largely be restricted to data formatting, compaction, and limit checking, and to input/output buffering. However, in a few instances -- for example, a strapped-down inertial reference unit -- more complicated computations would be performed at the subsystem level. In a distributed, or federated, system, the central computer facility would do significantly less processing; its principal task would be to control and coordinate the activities of two or more computers that were both logically and physically closely linked to particular subsystems or sets of subsystems.

A centralized system would permit a substantial majority of the software to be developed for a computer architecture that facilitates both programming and verification, and it would simplify the communication between related functional programs. A federated system, on the other hand, would greatly complicate the problem of allocating computational resources and would be less flexible. Therefore, a centralized system would result in lower software production costs than a federated system. The isolation that prevents a program in one computer from erroneously interfering with a program in another can be achieved in a centralized system by proper

The executive, whether polling or interrupt, is in any event simpler to develop and easier to verify for the simplex computer than for the multicomputer or multiprocessor. For these two configurations, significant executive design aspects are the location of the executive and tasks and the assignment of processors to tasks. In the distributed-executive scheme, which is applicable to both configurations, each computer or processor has its own executive and tasks. In the multiprocessor/fixed-executive scheme, the executive is assigned to one processor; this one executive then dynamically controls the execution of all tasks, allocating them to processors as dictated by the computational load. In the multiprocessor/floating-executive scheme, both the executive and the tasks are allocated according to the mix of tasks in execution.

Although the fixed and floating executives offer increased flexibility and performance, the greatly increased verification problems they present led to the conclusion that the distributed executive is more suitable for a Space Shuttle multiprocessor configuration. As regards distributed executives for multicomputer vs. multiprocessor, the only significant difference lies in the mechanism for communicating between tasks allocated to different computers or processors. In the multicomputer, this communication must be done through the normal input/output operations, while for the multiprocessor it is accomplished by simple, direct memory access methods.

In terms of minimizing software development difficulty and cost, the simplex configuration would be best. A simplex configuration may not be sufficient, however, if anything more than the minimum functions are to be performed by the data management system. Indeed, great increase in software development difficulty will occur if a simplex configuration is chosen that provides inadequate or barely adequate computational capabilities. Of the two remaining configurations, the multiprocessor with a distributed executive would result in slightly lower software development costs than the multicomputer. Therefore, it is recommended that a simplex computer be selected if it can provide the necessary computational capability; this is likely only if the Space Shuttle hardware configuration and system requirements are close to minimal. If more computational capability is needed, the multiprocessor configuration with a distributed executive should be chosen in preference to a multicomputer configuration.

The multiprocessor with fixed or floating executive has been excluded because of the verification difficulties these two executive designs present. Indeed, verifying the executives for all of the configurations is one of the most difficult problems that will be encountered in software development.

Enough excess computational capability must be provided so that programming standards can be employed that, although increasing the software overhead, reduce verification difficulty. That is to say, the selected computer or computers must provide an adequate margin of compuational capability so that tradeoffs between program efficiency and verification ease can always be decided in favor of the latter.

2.4 Computer Architecture

In this portion of the study, the advantages of specific architectural characteristics were identified and their impact on the cost and difficulty of Space Shuttle executive and task program development and verification estimated. Although hardware costs were not explicitly included, only those architectures that have been demonstrated or could practicably be implemented were considered.

The basic criteria influencing software development and verification difficulty and cost were defined in terms of the suitability or adequacy of the following classes of architectural features:

- Memory
- Execution speed
- Input/output facilities
- Instruction set
- Word format
- Register organization
- Restart and self-test provisions
- Interrupt-handling facilities

Minimum capability levels with respect to the first three criteria are mandatory if the software is to be produced at all. After the basic requirements have been satisfied, the most important characteristics were found to be the memory, speed, and input/output capability margins that simplify both coding and verification.

Coding and verification costs increase sharply as the memory usage and execution time approach the available capabilities. On the other hand, given sufficient memory, speed, and input/output capabilities, almost any instruction set, register organization, and word format can be used without drastically increasing software costs. Providing surplus capacity poses a problem, however. Unneeded functions may be added simply because the capacity exists, leading to an even larger program that may then have to be tailored to match what turns out to be a barely adequate computation capability. It is

therefore vital that effective management controls be imposed so that the margins provided will be utilized for reducing rather than increasing software costs.

While no other architectural aspects were found to have an impact equal to that of adequate memory, speed, and input/output margins, appreciable reductions in software cost would result from a computer design that incorporates a significant number of those remaining. As an example, providing hardware floating-point arithmetic would reduce the size of the software by up to 5%, and this reduction would translate into even greater reductions in programming and verification difficulty. As another example, providing hardware memory locks and special save and restart instructions would have a smaller impact on program size but just as significant an impact on verification ease: many operations that otherwise would have to be checked individually would be accomplished automatically by the hardware. Finally, some aspects, such as interrupt-control mechanisms, would enable the adoption of software design approaches that would otherwise be impossible.

It is recommended that the computer configuration have memory, speed, and input/output capability margins of at least 40%. It should also provide the following architectural features, each of which will affect overall software costs by about 5 to 10%:

- A suitable instruction set, including hardware floating-point arithmetic
- Hardware memory locks, memory bound registers, or similar hardware protection mechanisms
- Extensive hardware error-checking and diagnostic facilities, with appropriate feasible correction capabilities
- Provisions for saving computer registers and other status information and for restoring these registers and the status with a minimum of program steps in the event of a computer malfunction
- Capability for enabling and disabling interrupts, both absolutely and according to at least three priority levels

Besides the architectural aspects already discussed, the overall software costs are affected to a lesser degree by the presence or absence of other features. Verification problems would be eased by providing partitioning

into read-only and read-write segments, uniform instruction execution times, and buffered input/output. Capabilities for partial-word addressing and general-purpose registers would be beneficial because these features reduce data handling. With regard to the way in which an adequate memory margin is achieved, a computer architecture that used an auxiliary memory or some form of virtual memory would be less desirable than one that provided an equivalent amount of uniform main memory. Interleaved memory is similarly undesirable; even though it provided adequate speed margin, verification problems would ensue because repeatability of program execution times would be seriously compromised. Other architectural aspects such as limited addressing range, unique or unusual instructions, and variable instruction and data formats are undesirable because they require more effort on the part of both programmers and verifiers.

There are still other architectural aspects that are of even less concern in terms of software development costs, but of great concern to the hardware designer -- so much so that they may determine whether the more important software-impacting features can be incorporated at all. A facility for microprogramming is such an aspect. While the instruction set is software-important, the means of providing suitable instructions -- whether by microprogramming or hardwired logic -- is of no significant concern to the programmer.

2.5 Programming Languages

The suitability of six high-order languages for Space Shuttle onboard software development was investigated in terms of such characteristics as the amount of the problem that could easily and effectively be stated in the language, the control the language affords over computer hardware operations, and the extent that the language can be utilized throughout the software development cycle. Of the six languages, four were designed especially for aerospace applications: CLASP (Computer Language for Aeronautics and Space Programming), SPL Mk II (Space Programming Language Mk II), SPL Mk IV, and HAL. Two -- FORTRAN and PL/I -- were primarily designed for general-purpose applications. Compiler considerations were also investigated, with the most influential found to be the efficiency and correctness of the machine code generated, the diagnostic facilities provided, and the compiler development cost and time.

FORTRAN was found to have few advantages and many disadvantages for onboard software development. A smaller portion of the software could be coded in CLASP or SPL Mk II than in the more comprehensive languages HAL, SPL Mk IV, and PL/I. However, with the simpler languages object

code efficiency would be greater and the cost of compiler development smaller. The executive program would in any event have to be written in assembly language: the simpler languages lack the necessary features, while the compilers for the more comprehensive languages would be unable to generate the highly efficient object code required. Overall, it was concluded that use of any of the aerospace-oriented languages to the extent possible would reduce software development costs. The problem of compiler efficiency, the principal obstacle to the use of a high-order language, can be reduced by providing adequate memory and speed margins in the computer configuration.

Accordingly, it is recommended that a high-order language be used. Even though they have deficiencies, any of the existing aerospace-oriented programming languages would be appropriate, with their compilers being more significant than the language capabilities as such. Because of the difficulty in obtaining a completely suitable compiler, particularly for the larger languages SPL Mk IV and HAL, memory and speed estimates should provide for object code inefficiencies of about 15% and, for the reasons stated above, the executive program should be coded in assembly language. In view of the possibility that a new compiler will in some cases generate erroneous object code because of its own deficiencies, verification must continue to be done at the machine or assembly code level. Substantial verification benefits can be achieved, however, because the compiler can be so implemented as to enforce conformance with programming standards and conventions.

2.6 Verification Tools and Techniques

In this part of the study it was determined that the three types of simulations used in previous aerospace software verification activities -- engineering simulations, interpretive computer simulations, and hybrid simulations -- will continue to be the most important verification tools for the Space Shuttle onboard software. However, means of improving these tools and their use are required. A deficiency of interpretive simulations has always been the ratio between simulation time and real time, and with the much faster Space Shuttle computer system this unfavorable time ratio will become even worse. Two partial solutions to this problem were conceived. First, the interpretive simulation should contain more extensive and automatic diagnostic and information-gathering features that, at the expense of a slight increase in simulation time, would permit more to be learned from each simulation run, thus reducing the number of runs required. Second, the hybrid simulation should be improved so that it can provide much of the detailed information about internal program behavior that presently is obtained only through an

interpretive simulation. One means of making this possible is to design the flight computer so that its internal operation can be monitored by another diagnostic computer that would be a new part of the usual hybrid simulation setup.

Of course simulation tools cannot be used to prove the absence of program errors; they can only demonstrate an error's existence. In view of the large number of errors possible in the large and complex Space Shuttle onboard computer program, analytic verification methods should be developed to aid in proving program correctness and to indicate program design methods that would result in programs whose correctness could be demonstrated. Two such approaches were explored: one defining the constraints that must be satisfied to achieve program correctness when a restart occurs, and one to prove program correctness for a limited number of program structures.

It is recommended that improvement of simulation tools and techniques be pursued, and that a master test plan be defined for comparing simulation results and validating the simulations themselves. Further, the development of means for demonstrating program correctness should also be pursued and the tools integrated into the Space Shuttle software development and verification procedures.

3. FUNCTIONAL ANALYSIS

The analysis of the functions to be performed by the Space Shuttle computer system was undertaken to establish its desired characteristics in terms of the computational load to be imposed on it. Computational load can be estimated in terms of the computer memory required, the number of instructions that must be executed in a given interval, and the input/output rates that must be maintained. Also important are factors relating to the computational tasks: their relative priorities, their periodicity, the amount of intertask communication, and the number and attributes of routines shared between tasks.

The analysis began by assembling and evaluating available reference data. Where the available documentation was incomplete or inconsistent, assumptions were made as to the most probable hardware configuration. In some areas, models were constructed and the range of requirements for alternative configurations was determined; in others, experience with existing systems was sufficient to allow the expected behavior of their Space Shuttle counterparts to be determined. The most detailed analysis was performed on the guidance, navigation, and control portions of the system, since these offered both the greatest computational load and the most comprehensive source material.

It was not possible to establish firm and detailed functional requirements; rather, the functional analysis indicated the rough order of magnitude of computational requirements. At this stage of Space Shuttle system definition, the computational load ranges from one close to that of Apollo to one that large ground-based computers of today would have difficulty in supporting. As the interfacing hardware becomes firmer, extension of the functional analysis would allow requirements to be determined with greater realism.

A major computational function not analyzed in the present study is the malfunction (or error) diagnosis, circumvention, and system reconfiguration necessary to meet the fail-operational/fail-operational/fail-safe requirements. A preliminary survey of this function indicated that the software needed to implement this requirement could double the computational load, while the far greater number of possible sequences of program execution introduced by this software could increase by an order of magnitude the time and cost required for verification. The added information afforded by further hardware design definition should permit meaningful analysis of reconfiguration requirements.

Overall computing load estimates were developed for three conceptual configurations -- minimum, moderate, and maximum -- defined as follows:

- Minimum Configuration: This system provides for a gimbaled inertial unit or a strapdown system with its own processor; a basic set of sensors (e.g., star tracker or telescope and RF-only navigation); limited onboard targeting; basic telemetry without data compression; displays essentially unprocessed in the central unit; status monitoring for guidance and control and other subsystems; and an executive appropriate to a multiprogrammed simplex computer with an interpretive language.

- Moderate Configuration: This system allows for a strapdown inertial unit processed in the central computer; unitized pointing platform and docking laser (with minimum filtering); dual propulsion systems (main and orbit maneuvering) with active load alleviation; extended onboard targeting; downlink data compression; a pair of redundant unified displays; status monitoring for the extended sensor set and for single-parameter checks of interfacing systems; and an advanced simplex executive without interpreter.

- Maximum Configuration: This system assumes a strapdown inertial unit processed centrally; the full sensor complement with a state-of-the-art filter; adaptability to three propulsion systems (including turbojet) with active load alleviation; onboard targeting for arbitrary rendezvous; data analysis including pattern recognition and multiparameter trend analysis; dual, independent unified displays; full monitoring of status of avionics and all other reconfigurable systems; and a multiprocessor executive without interpreter.

The computational requirements for these three configurations are presented in Table 2. Supporting analyses for the individual hardware subsystems follow in the remainder of this section. These analyses include hardware configurations supplemental to the configurations of Table 2; and some subsystems, notably the display subsystem and the docking laser, can result in computational loads far in excess of the estimations in Table 2. That is, the data in Table 2 represent the best estimation of what actually will be required for Space Shuttle.

Table 2. Computational Requirements for Three Space Shuttle Configurations

Parameter	Configuration		
	Minimum	Moderate	Maximum
Total memory size (thousand words)	43.0	120	250
Inertial reference	2.5	15	15
Navigation sensors	2.5	20	50
Guidance, navigation, and control	10.0	40	50
Telecommunications	0.5	2	20
Display and control	1.5	15	30
Guidance and control status monitoring	1.0	8	10
Other subsystem status monitoring	10	10	40
Executive, interpreter, and overhead	15	10	35
Word size (bits)	24	32	32
Add time (microseconds)	25	0.25	0.10

3.1 Inertial Reference

Two major classes of inertial references exist, distinguished by the nature of the gyro integration. A gimbaled system implicitly integrates attitude changes by maintaining a stable member, while a strapdown system maintains its reference as a set of data in the computer, which performs an electronic (normally digital) integration. The strapdown system, with its explicit computer integration, is more computationally demanding than the gimbaled system. Its higher reliability and lower cost make it the more likely selection for Space Shuttle.

The computational requirements of four strapdown systems representative of the state of the art are summarized in Table 3. The Lunar Module Abort Guidance System (LM/AGS) was self-contained except for alignment (derived

Table 3. Computational Requirements for Strapdown Inertial References

Parameter	System			
	LM/AGS	ASST	RSS	MIT/SIRU
Memory capacity (words)	4096	2150	13,130	16,384
Memory cycle time (μsec)	5.0	4.0	1.75	0.96
Word size (bits)	18	18	24	16
Number of instructions	27	--	43	--
Add time (μsec)	10	--	3.5	1.92
Multiply time (μsec)	70	--	14	5.76
Major cycle interval (sec)	2	1-2	1	0.5
Minor cycle interval (msec)	20-40	10-20	40	10

from memory of the primary system). The Advanced Supersonic Transport (ASST) strapdown system incorporated redundant sensors and limited logic for failure detection, diagnosis, and correction. The Redundant Sensor System (RSS) employed more elaborate reconfiguration logic, increasing the software requirements considerably. Of all the four systems, the MIT/SIRU (Strapdown Inertial Reference Unit) most closely approximates Space Shuttle requirements.

The MIT/SIRU uses six gyros and six accelerometers aligned perpendicular to the faces of a dodecahedron. The computing capacity indicated for the MIT/SIRU is capable of performing the integration and formatting required of any strapdown system, plus fail-operational/fail-operational reconfiguration in its entirety. The logic to support reconfiguration for a third level of failure detection and diagnosis might be developed, but would be qualitatively different in form and would impose a very large load on the computing system. Instead of the extra software, a second SIRU may be used. That choice also protects against catastrophic failure, so is preferred. If the dual system is used, the computing load would be approximately double that shown in the table.

The primary information to be supplied by the inertial reference to the data management system is the attitude of the vehicle in inertial space. The

initial form of gyro data is the pulse representing an increment of attitude about that gyro's input axis. In any strapdown system, these pulses must be accumulated and transformed through rotation of coordinates to derive inertial attitude. The nonorthogonal arrangement of SIRU axes complicates the rotation algorithm but does not alter the basic task.

In dividing the attitude determination between local and central processors, three choices exist: local formatting only, local accumulation and formatting, or full local attitude determination. If local processing is restricted to formatting, major burdens are imposed not only on the central processor but also on the data bus, which must transmit up to 1600 pulses per second per sensor. By accumulating the pulses in a local buffer, the data rate may be reduced without adding major computation. During the 10-msec minor cycle, 16 pulses are accumulated per sensor and may be transmitted with only 4 bits. They may be packed into words of length established by the data bus design; the effective rate is fixed at 48 bits per cycle. Additional bits are required for sensor status data.

Full use of a local processor is desirable for the SIRU because the computing burden is high. The coordinate conversion must be performed every 10 msec, whether executed in the local processor or the central computer. A convenient form for the output of attitude from the local processor is a set of quaternions (four 16-bit words). Although local coordinate conversion increases the traffic on the data bus, the saving in central computing is felt to be worthwhile. The accumulated velocity increments may also be transformed into an inertial frame in the local processor (however, subsequent integration to a state vector in the local processor would require external data and thus is undesirable). The transformed velocity increment may be packed into a single 16-bit word for each minor cycle. In such application of the local processor, an interface is maintained with the data bus equivalent to that of a gimbaled platform with accumulated ΔV. Development of suitable alignment algorithms may then permit central routines to be taken from programs proved on earlier projects.

In any division of computing burden between central and local units, status data must be provided for reconfiguration. The specific data required, their rates, and their processing requirements are not yet known. It is assumed that the local processor can perform high-rate filtering in the preferred configuration and that two 16-bit words per minor cycle will suffice to maintain the central files. In the absence of local processing, both high data rates and extensive central processing would be needed.

3.2 Navigation Sensors

The attitude and state vector estimates maintained by the inertial reference must first be established and should be updated through the use of special-purpose navigation sensors. A wide range of sensor systems has been considered for Space Shuttle. The simplest configuration in terms of computing load involves radio frequency transponding from known ground locations. The heaviest computing load is that for a system employing three sensors -- star tracker, sun sensor, and horizon scanner -- mounted on a unitized pointing platform having two degrees of freedom and driven by computer commands. Computational requirements for such a system are summarized in Table 4 and were derived as follows.

Initial alignment of the inertial reference will require stabilization of the spacecraft, acquisition of the sun and the horizon by the sensors, driving the platform to put each of at least two reference stars sequentially in the star tracker field of view, and processing tracker error signals, platform azimuth and elevation, and spacecraft attitude (from the inertial reference) for each sighting. Alignment will be confirmed confirmed driving the platform to at least one additional reference star and verifying its location in the star tracker's field.

Occasional star sightings will be made under computer control to maintain alignment, and the altitude of a sequence of reference stars above the horizon will provide position data. Operationally, one or two such points may be

Table 4. Computational Requirements for Navigation Sensors

Component	Data Format	Data Rate	
		Alignment	Tracking
Platform orientation			
Analog servo	2 X 16 bits	10 sps	1 sps
Digital servo	2 X 16 bits	30 sps	1 sps
Star tracker	2 X 16 bits	discrete	discrete
Sun sensor	2 X 16 bits	discrete	discrete
Horizon scanner	2 X 16 bits	10 sps	10 sps

taken every 10 minutes, as the other tasks permit, to minimize residual errors. Each platform angle covers a range of ±30°, for which 16-bit quantization seems desirable. The horizon scanner provides only elevation deviation, which, depending on the scanner field of view, may be quantized with up to 14 bits. Similarly, azimuth and elevation deviation from the sun sensor may be quantized to 14 bits each; however, practical arguments suggest that 16 bits will be used. If the star tracker field is 5° in each axis, 16-bit quantization of its data pair appears more than adequate, yielding less than 0.6 sec error. With regard to data rate, it is assumed that the horizon scanner will provide periodic data of the order of 10 times per second. The sun sensor and star tracker will have data available continuously; the former will be sampled up to 10 times per second, while the latter need be read only once per star. These rates are marginal to obtain 5-sec accuracy with a residual attitude rate of 0.01°/sec.

Once alignment has been obtained, there appears to be no requirement to command the platform repetitively; that is, given an inertial reference, a single pair of platform commands will drive the sensors to the desired attitude through a platform servo loop. This implies a data rate of 10 samples per second. If the reliability or other cost of such a loop is unacceptable, it it will be necessary to generate incremental commands with computer-derived damping to drive the platform to the desired position. In that event, platform angles may have to be both read and commanded at up to 30 times per second to obtain the desired stability, and it may be preferable to use a platform rate command rather than an angle command as the interface. The analogy between these functions and those of the digital autopilot may be strong enough to allow some common usage of routines.

In addition to the estimates presented in Table 4, normal monitor and command functions will be required, and calibration is indicated for the horizon scanner and will probably be required for the other sensors as well. Lacking data on the mechanisms to be employed, it is estimated that a few hundred words of program may be needed for each sensor; that the star tracker and sun sensor would be calibrated once for each set of measurements (perhaps once per 10 minutes of use); and that the horizon scanner will require collection of data over several scans, perhaps 10, in every 10 minutes of use.

3.3 Displays

The display hardware is less completely defined that any of the other avionics subsystems. To estimate computer requirements, one dynamic display example -- the video image of the runaway for blind landing -- was constructed and analyzed in detail. From AWAILS or from ILS and the altitude radar,

the computing system determines the vector in inertial coordinates to the end of the runway; this may be rotated into the vehicle frame. The magnitude of the vector (distance to the end of the runway) establishes the scale of the image, while the pitch, roll, and yaw angles control the display perspective.

Any one of several analog mechanizations may be used to support a CRT image or a projected slide of the data with frequent update. The maximum load such an analog-supported display would impose on the computing system would be four 24-bit words sent 30 times per second. Up to 40 add times may be required to format each word, yielding 4800 adds per second.

The analog support equipment, while it reduces the digital load, is both heavy and subject to failure. Replacing the analog hardware with special-purpose digital devices does not alter the problem; only centralized digital processing reduces the display penalty by providing redundancy through equipment already onboard. However, in providing the display of the example with digital processing, a heavy computing load is added to the central processor. The 30-frame-per-second rate is still required to avoid flicker, but now each point must be commanded by the central computer.

An algorithm that minimizes the central processing required was developed for generating the display. In constructing the algorithm and the example, it was assumed that only straight lines with a single intensity level would be required, even though several values of intensity may be desired for other purposes. Even so, each display frame requires six sets of four segment parameters and one inversion. For each of the 400 to 500 lines in a frame, the computer must apply the algorithm to each display line (six multiplies, six adds), determine the termini of each segment (12 adds), test each terminus for inclusion in the frame (four or eight compares), and assemble a sequence of binary words in which each bit corresponds to a point on the scan line. Typically, 15 words will be required to depict a scan line, suggesting a composite requirement for each line in excess of 6 multiplies, 18 adds, 24 compares, and 15 logical OR's. The available time, determined by the required frame-per-second rate, is 30 msec divided by 400 lines, or 75 μsec. Assuming that 10 adds require the same time as one multiply, the arithmetic operations alone require an add time of less than 1 μsec.

Table 5 summarizes the results thus derived for the analog-supported and the centralized displays. It can be seen that even a central processor with a 0.5-μsec add time would be hard pressed to handle the unsupported display when the logical operations for each line are added to the arithmetic operations, and when the projections required to provide segment information at

Table 5. Computational Requirements for One Dynamic Display

Parameter	Supported Display	Centralized Display
Bits per frame	96	200,000
Words per second	120	220,000
Adds per second	<5,000	>1,000,000

the frame level are included. Further increases in computing load may arise from the need for color display; redundant and supplemental displays; and requirements for trend analysis or other digital processing. Thus from this work it is apparent that the range of requirements for the displays is several orders of magnitude. Fortunately, analog hardware can be used to maintain a reasonable computing load.

3.4 Trackers

Three tracking subsystems were reviewed: the docking laser, the rendezvous radar, and the infrared tracker. The resulting computing load in terms of data and command rates is developed below and summarized in Table 6. It should be pointed out that no interaction between these subsystems was indicated in the pertinent preliminary literature -- although it is likely that such interaction will exist and that it will have significant computational impact.

Table 6. Computational Requirements for Trackers

Tracker	Tracker Data		Computer Commands	
	Bits/Sample	Samples/Second	Bits/Command	Commands/Second
Docking laser	24	$<10^3$	16	10^3
Rendezvous radar	48	10	16	noniterative
Infrared tracker	16	1	--	--

3.4.1 Docking Laser

The Space Shuttle requires a new type of sensor to determine the attitude of a passive base for docking. The one studied employs a modulated laser beam to determine the range along the beam and the displacement from its center of the return from each of three reflective targets on the docking ring. The modulation allows a separate range processor to establish the distance to each target within a fraction of the shorter modulation wavelength (13 ft) over a separation equal to the longer wavelength (1300 ft). Target position in the plane perpendicular to the beam is determined by electronic scanning of the detector. The beam must dwell on each target long enough to obtain a suitable signal-to-noise ratio; it is estimated that the maximum data rate from the docking laser is 1000 measurements per second.

Range is quantized with 8 bits of coarse data and 4 of fine to provide resolution of 5 in. The position of the scan at maximum signal is encoded with 6 bits in each direction. Scan command may be provided by a local processor on receipt of a discrete from the central computer, or it may be provided directly by the central computer; this implies a data rate in excess of 10^6 bits per second. Another alternative is to use some local processing to reduce the load on the data bus, decoding a 16-bit word each millisecond into a search pattern.

Reduction of the laser data is a major problem for the computer. One method was studied in which a coordinate converter, dynamic models of the Space Shuttle and target, and a Kalman filter are employed. The coordinate converter rotates the reference system into a frame with minimum filtering to reduce the total computing load. The dynamic models then estimate the attitude and position of each vehicle for comparison with sensor data, and the Kalman filter determines the best estimate of attitude and position from the past and present sensor data, taking into account detectable sensor errors. The Kalman filter is the only known means of determining docking information with the required precision. Among the operations required for its use are the inversion of a matrix presently estimated to contain over 1000 elements. Ideally, the filter would be applied to each target as detected by the laser, requiring up to 1000 matrix inversions per second. The resulting load is the equivalent of 50×10^6 additions per second, a value which cannot be handled by even a large ground computer. Only when detailed system requirements and sensor designs have been established can a practicable filter be developed.

3.4.2 Rendezvous Radar

The rendezvous radar locates an active target through scanning and detection of a microwave beam, with the electronic scan either generated in the radar

or commanded by the central computer. The scan position at maximum return locates two coordinates of the target; each coordinate may require 8-bit encoding to maintain accuracy. Sixteen bits may be provided for each of range and range rate, so that there will be 48 bits in each data set. Although lower rates have been suggested for Space Shuttle, automatic rendezvous may require up to 10 data sets per second. The phased array used on the radar is amenable to a Cartesian coordinate system, with 8 8 bits in each dimension providing scan commands. Signal lock is maintained by radar electronics, so that only one set of commands is needed to acquire and maintain lock.

3.4.3 Infrared Rendezvous Tracker

The infrared rendezvous tracker detects the thermal radiation of the base to determine polar coordinates within its field of view (5° full-cone angle about the longitudinal axis). Scanning is entirely under internal control. Data should be available at least once per second, each set consisting of the azimuth and elevation of the base, with each angle encoded to no more than 8 bits. Depending upon detector design, it may be possible to use the infrared rendezvous tracker with the sun in the field or reflected by the base. It is possible that sun-avoidance logic for the scan pattern may be required, suggesting two 8-bit words per second from the computing system to the scan control when the sun is in the field of view.

3.5 Landing Aids

Five landing aids were reviewed: altitude radar, ATC transponder, VOR TACAN, instrument landing system (ILS), and all-weather automatic instrument landing system (AWAILS). Their computing loads, as derived during the functional analysis, are discussed below and summarized in Table 7 in terms of data rates. The last three systems are complementary in principle but will have significant periods of combined operation where their data will be supplementary.

Table 7. Computational Requirements for Landing Aids

Landing Aid	Data Rates
Altitude radar	16 bits/100 milliseconds
ATC transponder	9 x 24 bps (output from DMS)
VOR TACAN	3 x 8 bps
ILS	2 x 8 bps + 1 discrete
AWAILS	3 x 8 bps

3.5.1 Altitude Radar

This landing aid determines height above the local terrain from the transit time of a 9-GHz signal. Since terrain variation will be a significant factor in the received data; smoothing will be required by the computing system. It seems likely that 16 bits will suffice for the data, and that a nominal rate of the order of 100 msec between samples will provide rapid response and effective smoothing. One possibility is that landing at the nominal site would call for removal of known terrain variation from the data, either complementing or replacing smoothing. Since the vehicle must be capable of landing at unplanned sites, such a feature for the nominal seems to be of little advantage, while it might commit a significant portion of memory.

3.5.2 ATC Transponder

The air traffic control transponder is modulated with avionics data for ground control. The following data should be required no more than once per second:

- Reference time (GMT): 16 bits
- Mean altitude above terrain (unweighted average of altitude radar data): 16 bits
- Inertial state vector (referenced to the rotating earth): 6 X 16 bits
- System status code (synthesized from individual status data): 16 bits

Other data available in the landing phase would appear to be of little value to those receiving the transponder signal, since the computing system interface is independent of the operating mode of the transponder during landing phases.

3.5.3 VOR TACAN

VOR TACAN is a pair of data sources (omnirange and TACAN) providing inflight data relative to earth-fixed transmitters. Each system employs a signal from the ground, decoded onboard into an identifying tone for the transmitter, a reference bearing, and a variable bearing as a function of vehicle position relative to the antenna. It is assumed that manual identification of the station will be employed, since the analog/digital and digital

requirements for automatic identification seem unnecessarily costly. TACAN also includes an active ranging path to its antenna. Particularly because of the short range over which VOR TACAN data are good, 8-bit encoding of each bearing signal and of range would appear sufficient. While data are available essentially continuously, sampling once per second should be sufficient.

3.5.4 ILS

The instrument landing system is used to bring the vehicle to the middle marker of the runway through detection of azimuth and elevation from a reference transmitter; a supplementary antenna signals passing of the outer and middle markers. The detected signals of glide slope (elevation) and localizer (azimuth) may be usefully encoded to about 8 bits and sampled once per second. The two marker signals should be discretes to the computing system, and require sampling at the same rate but for only the brief period near their expected occurrence. Ambiguities exist in ILS data and may require resolution in the computing system, both for display and for automatic landing. Comparison between inertial reference and ILS data should be sufficient to determine which of the five possible ILS references has been detected and to optimize the landing trajectory.

3.5.5 AWAILS

The all-weather automatic instrument landing system, employed only in the final stages of landing, provides angular interfaces equivalent to those of ILS and supplements them with measurement of range to a transponder on the runway. Again, 8-bit encoding of each of the three data inputs should suffice and sampling once per second would appear adequate.

3.6 Primary Propulsion Subsystems

The orbiter primary propulsion subsystems may employ two separate sets of engines: the main engines for the final stage of boost and the orbit maneuvering subsystem for subsequent major thrust maneuvers. A reasonable construct for operation of either engine subsystem calls for a set of pre-ignition commands over a period of seconds to minutes, an engine-on signal maintained throughout thrusting, gimbal angle commands of indeterminate rate and quantization for both degrees of freedom, and a mixture-ratio command. In the absence of data about a specific subsystem configuration, the estimates summarized in Table 8 were made for the design carrying least load on the computing system. These estimates assume that each engine may be gimbaled approximately 5° about each axis; quantization to 0.1° would be useful. A reasonable interface would allow for up to

Table 8. Computational Requirements for Primary Propulsion Subsystems

Function	Requirements
Gimbaling	4 axes: 10 bps/axis + 1 discrete
Throttling	6 (air-breathing) engines: . ? bps + 1 discrete
Start sequence	1 command (discrete) with computer countdown
Engine-on	1 command (discrete) with computer countdown
Mixture ratio	2 (rocket) engines: 1 bps + 1 discrete

10 commands per second per axis, each incrementing or decrementing the gimbal angle by one count. Initiation of thrust may use two discretes (engine sequence start and thrust start), or it may require a sequence of discretes issued on a time base; the most difficult implementation would provide discretes triggered by measurements of propulsion parameters under computing system control. Measurement of propellant quantities and flow rates provides the data for computer determination of the desired mixture ratio; the increment/decrement discrete is required no more than once per second. Each incremental signal requires a discrete which causes the controller to drive to the reference (null) position.

3.7 Reaction Control Subsystem

The reaction control subsystem employs 20 thrust chambers burning hydrogen and oxygen to generate relatively small torques and forces for attitude control and small velocity increments. If continuous thrusting is possible, the output of the computing system may be a discrete for each chamber (assuming hard-wiring rather than the use of the data bus). In that event, the signal is counted down with a tolerance of the order of 1 to 5 msec. If only a pulsed mode is available, the number of such pulses (up to perhaps 800 per second) must be transferred to the buffer in a local processor for each assembly. Two or three such items are required for each assembly, depending upon whether four or six thurst chambers are used.

Many configurations may be constructed in which computing system functions are assigned to local hardware and software. The above configuration does not allow for some modes of employing rate gyros, nor is it clearly the preferred type of interface for control surfaces in atmospheric flight. The

specific mechanism by which a command signal actuates the reaction control subsystem was not clear, and is significant in determining the software requirements. A representative computing load might involve 20 thrust-on discretes, up to six active at once, and each counted down at 400 bits per second (may be synchronous).

4. COMPUTER CONFIGURATION

The Space Shuttle onboard software may be divided into two categories: functional and executive. The functional software performs the functions dictated by the requirements of the hardware subsystems and the mission objectives. The executive software controls and coordinates the execution of the functional software in accordance with overall system requirements.

As shown in Figure 1, the functional software may be subdivided according to the functions to be performed, such as guidance, control, and checkout. Within these subdivisions it may be further categorized as to the specific computations to be performed, such as initialization and the computations performed at the minor and major cycle intervals. This final division of the functional software is made according to what will be called tasks, each task consisting of well defined computations to be performed at specified times or when specific criteria are satisfied. In general, each task will require some maximum time for execution.

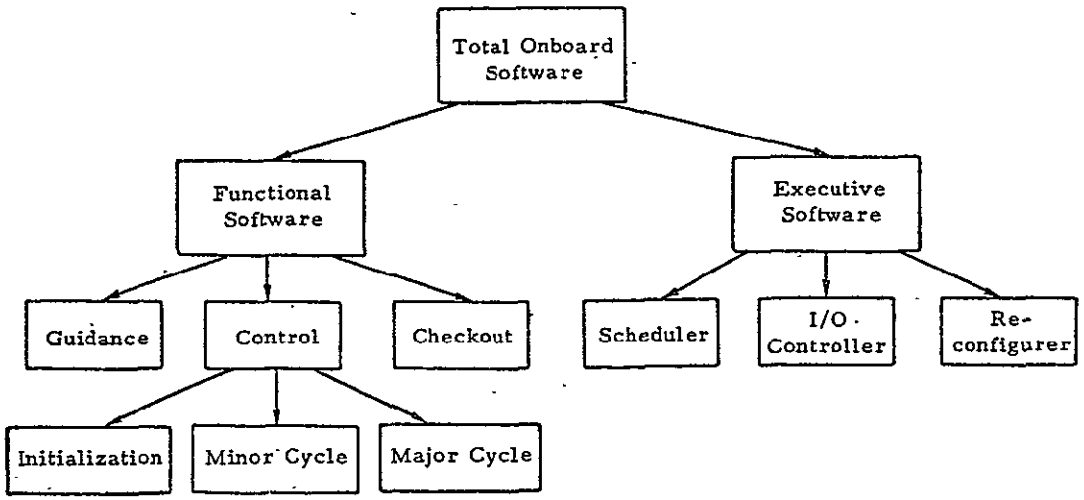


Figure 1. Onboard Software Structure Outline

Executing the many tasks will require the sharing of computer resources such as the scratchpad memory and the input/output devices. The executive software will coordinate this sharing and control the resulting operations. Like the functional software, it may be divided into tasks, such as scheduing the functional tasks for execution, controlling and performing their input/output operations, and reconfiguring the computer system according to the computational load or the health of the computer hardware components.

The functional tasks can be designed, coded, checked out, and verified independently of the specific computer configuration, except for the ways which these tasks interact with the executive system. Development of the executive, on the other hand, is intimately tied to the computer configuration, both as regards: 1) the problems a particular configuration may present in performing the task-oriented executive operations, and 2) the special executive operations that the configuration might itself demand to be performed. Therefore, the tradeoff of alternative computer configurations was approached by studying the advantages and disadvantages of the executive software.

Three alternatives were examined to determine which, for the anticipated Space Shuttle functional requirements, would be the most suitable configuration from a software point of view:

- Simplex Computer: The simplex or single computer, by far the most common configuration for both aerospace and general-purpose applications, consists of a single arithmetic and control processor, a memory or a collection of memory banks, and an input/output processor or controller.

- Multicomputer: This configuration consists of two or more simplex computers, each having its own arithmetic and control processor and memory. The separate computers, which need not be identical, communicate with each other by means of their individual input/output provisions.

- Multiprocessor: This configuration consists of two or more arithmetic and control processors connected to a common memory such that, in general, any processor can execute any program or access data located anywhere in memory.

The simplex and multiprocessor configurations are inherently centralized organizations, both physically and logically; the central computer facility performs the computational functions required by the hardware subsystems,

similarly to the way a commercial data-processing center supplies computational services to its users. A multicomputer configuration can be physically and logically considered as either a centralized system or a decentralized (or federated) system, depending on the number of computers and how the computational capabilities are allocated. In a centralized system there would be relatively few computers, each servicing part of the hardware subsystems. In a federated system the computational functions would be allocated to the computers in more of a one-to-one manner, with the operation of these subsystem level computers under control of a master computer.

Among the software aspects relevant to the tradeoff between centralized and federated systems are the disciplines enforced in software production, the cost and difficulty of software production activities, the computational loads that can be supported with equivalent hardware complexity, and the protection that might be provided from software failures. These are discussed first, followed by a look at the functions to be performed by the executive, and next by preliminary executive designs for the three computer configurations. The section concludes with a summary of executive and computer configuration comparisons.

4.1 Centralization/Decentralization Considerations

The basic option available in the physical organization of the Space Shuttle data management system is that between centralization and distribution of the computing capacity. Neither extreme is probable: concentration of digital processing in a single central computer would overload the data bus, while some central functions (e.g., steering, mission control, reconfiguration) could not reasonably be allocated to a remote processor. Thus the requirement is to optimize the separation of functions into local and central processors under the mission and cost constraints. A complete tradeoff awaits a more complete functional analysis, and this itself awaits spacecraft design decisions. However, many important considerations are apparent at this time.

In the extreme represented by a fully centralized system, the computer is divided into processors whose number and usage are established by the instantaneous load. All data are fed over the data bus by analog/digital converters, with only minimal buffering to allow noninterfering, serial transmission. Fully resolved data are received at the subsystems, so that only format conversion is required to provide the stimuli needed for their checkout. The data rates associated with the fully centralized configuration would approach an order of magnitude increase over those attainable with more extensive local processing. That penalty increases not only the

capacity required of the data bus, but also the input/output load on the central processor.

In the extreme represented by a fully federated system, all processing possible at the subsystem level is performed in dedicated processors located with the interfacing hardware; the central computer performs only the tasks involving the integration of data from several subsystems. A major advantage of such a design is the discipline it forces on defining the interfaces between software located at the subsystem level and the integrating software of the central computer. In addition, the powerful computing capability at the remote locations minimizes data rates and should save both cost and weight in the distribution system. However, the replication of routines among the local processors would be great, so that each would have to bear an overhead comparable to that of the entire centralized system. Memory and computation capacity would be committed to executive functions and to task programs and subroutines that are handled more efficiently in the centralized system. Furthermore, it is unlikely that the architectural characteristics of the smaller local processors would be conducive to easy programming. As a result, a considerable amount of software would have to be developed for hardware that is less than ideal from programming and verification points of view.

An essential computational function is the reduction of masses of data (e.g., accelerometer pulses, supply voltages) to significant information (e.g., steering commands, status displays). The partitioning of tasks to central and local processors is essentially a process of optimizing the allocation of data reduction to the different sites. The total data reduction load may be assumed to be independent of how it is allocated. Data compression, which represents an element of the filtering the system will be required to perform, would have to be executed either explicitly or implicitly in the central processor if local capability were omitted. Once such elementary capabilities as accumulation of accelerometer pulses are assumed in the local processors, there is a tendency to add others to reduce the load on the central unit.

With each level of compression at the local processor, less raw data need be forwarded to the central unit. At some point, data among subsystems must be correlated; at that stage, central processing is indicated. Typical of that case is the determination of steering commands from engine, accelerometer, and gyro data. A fully decentralized system would reduce the role of the central unit to that of performing multisource data analyses. Performing these analyses will require powerful processing units and an extensive memory regardless of how the functions are distributed. In the federated configuration, some local processors will be

comparable with the central processor in magnitude. If the local processors are physically and functionally remote from the central processor, any redundancy requirements must be reflected in them separately. Consequently, a redundant system grows rapidly more expensive with increasing decentralization.

The relationship between cost and the degree of decentralization may be illustrated by the following example of two configurations. In one, a processor is provided for each of N subsystems and an additional equivalent processor is provided to handle interprocessor communication. If system requirements dictate survival of any two failures in processing units, this configuration requires 3(N+1) processors. The other, more centralized configuration provides the same total computing power, but allocated such that each processor supports three subsystems and the balance of the required processing is provided by the central computer. This configuration would require N+(2/3N+1)+2 processors. The more centralized configuration needed to accomplish the same computing is therefore composed of 4/3N-2 processors fewer than the other, which is significant even if there were only three subsystems (8 units against 12). The same argument is relevant with respect to the location of memory modules. Memory advantages also exist in that portions of code would have to be repeated in the decentralized system, whereas in the centralized system they would merely be accessed repeatedly.

The principal argument against centralization deals with the reduced cost of verification associated with breaking the code into separate programs. Decentralization enforces modularity by removing the interfacing routines to separate processors. The same attention to software modularity and interface requirements can and should be applied to the centralized system. Programs can be separated to minimize unwanted interaction through a combination of software design techniques and computer architectural features, just as effectively as separation is achieved physically in the decentralized system.

4.2 Executive Functions

The system executive will be required to perform the following four basic functions, each of which is discussed individually in what follows:

- Allocate resources
- Perform and coordinate I/O operations
- Maintain real-time control
- Preserve system integrity

4.2.1 Resource Allocation

This function takes on different forms depending on hardware design. The resources available may include memory banks or modules, secondary memories, I/O devices, and processors. It is the management of these resources, including routing of processing through the program elements, that prompts the use of the term "executive." Essentially, the functions performed under resource allocation are those of controlling and managing the computer hardware as dictated by the needs of the software.

At one time, flight software requirements were such that an inflexible sequencing of task execution could be tolerated: programs written for ballistic missiles required no executive. The Space Shuttle represents the opposite pole in that the sequence of task execution will not be predictable before flight. While some operations may always be performed before others (e.g., boost guidance precedes reentry maneuvering), most tasks are interleaved as a function of hardware subsystem activity.

Accommodating such a flexible sequencing environment greatly complicates resource allocation; very often a task request will occur when the available computer hardware is already allocated to performing other tasks. In one sense the job of resource allocation is greatly simplified when there are relatively few resources to allocate; for example, the allocation of processors to execute tasks in a configuration with only one processor is obvious. Resource allocation is obviously simplified if there is a surplus of resources; in this case the allocation can be accomplished inefficiently yet still satisfy hardware subsystem requirements. For example, with a large number of processors it would be possible to initially allocate tasks such that none assigned to the same processor would ever need to be executed at the same time.

Most tasks require the use of erasable memory for intermediate storage and retention of input and output data. Committing memory by task without sharing may in some cases require more memory than is available; so memory may have to be allocated under the control of the executive. Similarly, if a mass memory device is used for storage of program elements required only during given mission phases, its management would be another resource allocation function. The actual reading and writing of the mass memory would be accomplished by the executive's I/O elements under this management.

4.2.2 I/O Performance and Coordination

Among the things the executive will be required to do in this category of functions will be executing the I/O operations indicated by task-issued I/O directives, monitoring the I/O operations to resume task execution when the indicated data have been received or transmitted, and sequencing the I/O operations so that both the task requirements and those of the hardware subsystems are satisfied even if the I/O directives from the tasks appear to conflict. In some simple systems the performance of I/O operations is left to the functional tasks themselves. The resulting distribution of I/O activity throughout the software introduces much duplication and requires that any task about to perform I/O be aware of possibly conflicting I/O being performed by all other tasks. The method is undesirable for a software system as complex as that for the Space Shuttle, and particularly undesirable because the preponderance of the I/O in this system uses a serial data bus. The sequencing of data on the bus must be very carefully managed to insure not only that the appropriate data are sent or received, but also that this is accomplished in a timely fashion to satisfy the software system's real-time constraints.

4.2.3 Real-Time Control

Some tasks will have relatively flexible real-time constraints: it will be permissible for them to be executed at any time during some fairly long interval. Examples are initialization for maneuvers that may not be scheduled to take place for some time, performing onboard targeting computations, and checking some vehicle hardware subsystems. For such tasks the important thing is that they be done rather than the precise time at which they are executed. These tasks comprise the background computation. Other tasks will have to be performed in response to some specific stimulus within some short period of time, and for these the executive must make certain that the real-time constraints are fulfilled. Examples of such tasks are those involved in making the proper responses to controller or keyboard inputs of the crew and those that, in response to an error signal, reconfigure the system to minimize the effect of the error.

Thus the executive must allow for the maintenance of real-time control between task software, hardware subsystems, and the vehicle. This can be accomplished in many ways. One is for the executive to conduct a poll of hardware devices and software lists to determine the tasks to perform; another is for the executive to respond to requests for task execution made via hardware or software interrupts. When such an

interrupt occurs, the executive must identify the source of the interrupt and its associated task and priority; compare its priority with that of the task in execution; and, if the new task is of higher priority, cause the current task to become dormant and the new task to begin execution.

Another type of real-time control that will have to be exercised by the executive is the scheduling of tasks that must be executed periodically. Examples of such tasks are the guidance and navigation major cycle computations and the flight control minor cycle computations. This can be accomplished by interrogating a real-time clock to determine when the proper interval has passed or by responding to an appropriate periodic interrupt.

The usual approach to maintaining real-time control is to establish a priority scheme in which tasks that must respond very quickly to certain stimuli are given a high priority and the background tasks a low priority. In a priority scheme of this sort, periodic tasks must have dynamically changing priorities, either in actuality or in effect. Normally the periodic task has a priority so low as to prevent its ever being executed, and at the appropriate times its priority is increased. Its high priority is maintained while it is being executed, after which it returns to its former low state. Similarly, many background tasks may require dynamic priority adjustments. For example, computer self-test might normally have a low priority; and if it has not been completed in some specific interval its priority would be increased. This would continue to occur until either it was executed or its continuing high priority indicated that something was consuming available computer resources to an unexpected and possibly erroneous degree.

4.2.4 System Integrity

Because of the variety of things to do, the situation may occur in which tasks to be performed require more than the time available. (One example of how this can be detected was just described.) When this condition exists, a system overload has occurred. The executive must be able to resolve these conflicts and do only those things that are mandatory for the correct function of the system. It can resolve the conflicts by reassigning priorities, aborting low-priority tasks, or stretching out the time allotted.

The executive must have the ability to detect some types of faults in external devices and in the computer on which it is executing. It must also periodically schedule tasks to perform diagnostics on the devices and on itself to ascertain any problems and take corrective procedures bypassing the elements in error. In case a malfunction in the computer system

is detected, the executive must be able to communicate with a backup computer and switch to it in such a manner that the vehicle continues and maintains its scheduled mission.

Finally, the executive must be so designed, constructed, and verified that there is no question that it can perform its functions. It must be such that adding or deleting tasks does not require extensive executive modification, and must also be so written that adding or deleting hardware devices similarly is easily accommodated without making more likely the occurrence of software errors within the executive itself.

4.3 Preliminary Executive Designs

The choice among executive designs is a function of the mixture and complexity of the functional tasks, the design of the digital hardware, and the relative value of execution time and memory overhead. In general-purpose computation, the executive may be called the operating system; the generality of its use imposes a high cost in system performance that must not be borne by the special-purpose executive for the data management system. However, some overhead penalty should be anticipated to allow enough flexibility to provide system expansion as the Space Shuttle evolves. Optimization of executive design is a function of the range of requirements anticipated. The initial requirements determined during the functional analysis activity were adequate for the initial study of executive design. While further definition of functional requirements will allow refinement of the preliminary designs described here, it is not expected that the conclusions will change.

The principal tradeoffs in executive design are the means by which task execution is initiated, the flexibility permitted in task allocation, and the residency of the executive. The task-to-processor allocation and executive residency tradeoffs of course apply only to the multicomputer and multiprocessor configurations.

Three alternatives exist as to the means of task initiation:

- Implied Executive: Each task, when completed, transfers control to the next task to be done.
- Polling Executive: When one task's execution is completed, the executive selects the next based on a poll of those waiting to be done.
- Interrupt Executive: A task is interrupted during execution to perform a more important task.

To simplify the comparison, preliminary executive designs for the three task initiation alternatives were developed for the simplex configuration only.

Three allocation and residency alternatives exist for the multicomputer and multiprocessor configurations:

- Distributed Executive: Tasks are divided into fixed sets, with each computer or processor executing only those tasks allocated to it under control of executives permanently resident in each computer or processor.

- Fixed Executive: Any task can be performed by any processor, but the executive functions are always performed by the same processor.

- Floating Executive: Any task and the executive functions can be performed by any processor.

The second and third alternatives would be practical only for the multiprocessor configuration, for executing on one computer a task located in the memory of another would obviously be very difficult, if not impossible. Therefore, distributed executive designs were developed and compared for both the multicomputer and multiprocessor configurations, but fixed and floating executives were investigated for the multiprocessor only.

4.3.1 Implied Executive/Simplex Computer

The implied executive (Figure 2) does not exist as a distinct program. Rather, executive functions are performed by the individual tasks. Each task must know which task follows and transfer control to it for execution. If various tasks are permissible, the burden is on the executing task to pick the correct one and start its execution. Then the logical decisions to determine what to do next, properly the function of the executive, have to be added into the task coding itself, thus complicating the task coding, reducing its independence from the computer configuration, and obscuring the actual program structure. It also becomes necessary to verify the program as a complete assemblage of tasks rather than to do extensive independent checking of the tasks and the executive.

Although it may be suitable at the local processor level, the implied executive is not considered a serious candidate for the central computer facility

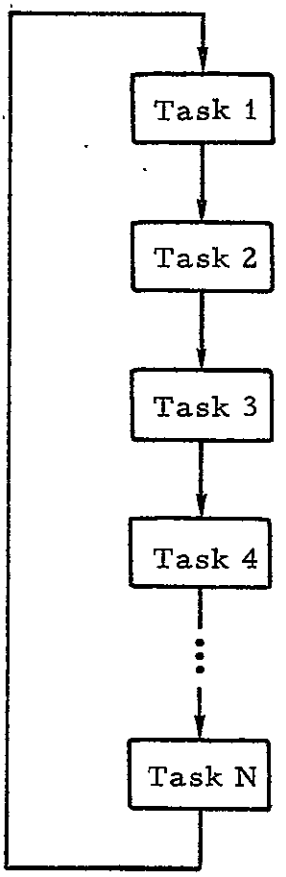


Figure 2. Implied Executive Concept

for the Space Shuttle; the multitude of events occurring, at times simultaneously, simply cannot be handled by a series of tasks executing consecutively. The main purpose for describing the implied executive here is to use it as a base for measuring the relative complexity of other executive designs.

4.3.2 Polling Executive/Simplex Computer

If the computer configuration chosen has no hardware-generated interrupts, an executive that interrogates the external devices for activity would be a likely choice. After each task is completed, control is returned to the executive. It then determines which task to do next by performing a poll of

the external devices and examining a software queue of tasks awaiting execution. Figure 3 illustrates the concept and Figure 4 shows a high-level polling executive design.

In this design, the executive is entered at the completion of every task. The first thing the executive does is poll the input and output devices and execute appropriate I/O tasks if required. Next, the current time is determined by reading a hardware clock; this is necessary since, in general, the time since the executive was last performed is unknown owing to the varying times the tasks themselves will take to execute. Then queues of time- and input/output-dependent tasks are examined to determine whether any particular task should be performed. Examples of such criteria, which may be applied singly or in combination, are the reaching of the time at which a task should be performed, the completion of a lengthy output operation, or the receipt of an input directive to perform a specific task. If the appropriate criteria are met, the indicated task is placed in the queue of tasks to be executed and, unless a periodic task, removed from the criteria-dependent task queue. Finally, the task with the highest priority in the execute queue is selected, it is removed from the execute queue, and the executive transfers control to it.

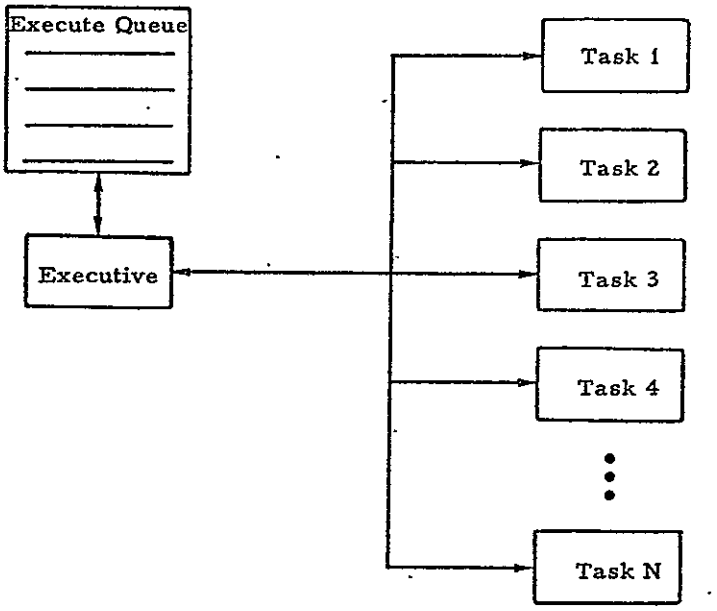


Figure 3. Polling Executive Concept

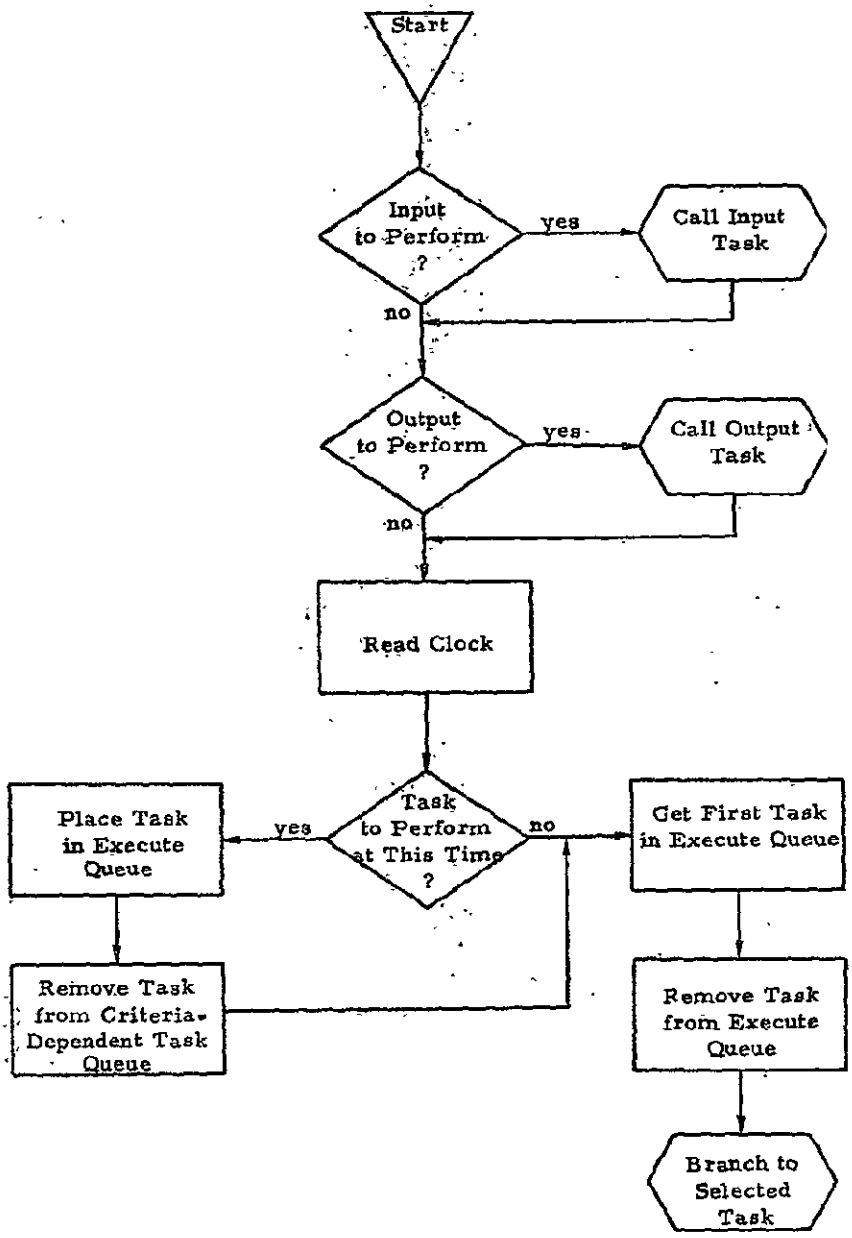


Figure 4. Polling Executive Design

Since there are no hardware interrupts, the polling executive must be executed at an interval such that the fastest device in the system -- most likely the timer or clock -- is serviced when needed. This requirement would necessarily force all tasks to be written in such a manner that they finish executing in the allotted interval, which could be as little as, say, 20 msec. Thus a task in this case may consist of a section of coding that is only a part of the overall function to be performed.

The polling executive has the advantage of being repeatable, that is, under given circumstances it would perform in the same manner. This feature is of considerable importance in verification. However, the timing considerations and the constraints they impose on task program design and development have a substantial effect in raising software costs. Further, an attempt to modify existing tasks or insert new ones would cause the timing to be off unless a completely new analysis were performed.

4.3.3 Interrupt Executive/Simplex Computer

The interrupt executive, shown conceptually in Figure 5, is similar to the polling executive as far as the queue of tasks awaiting execution is concerned.

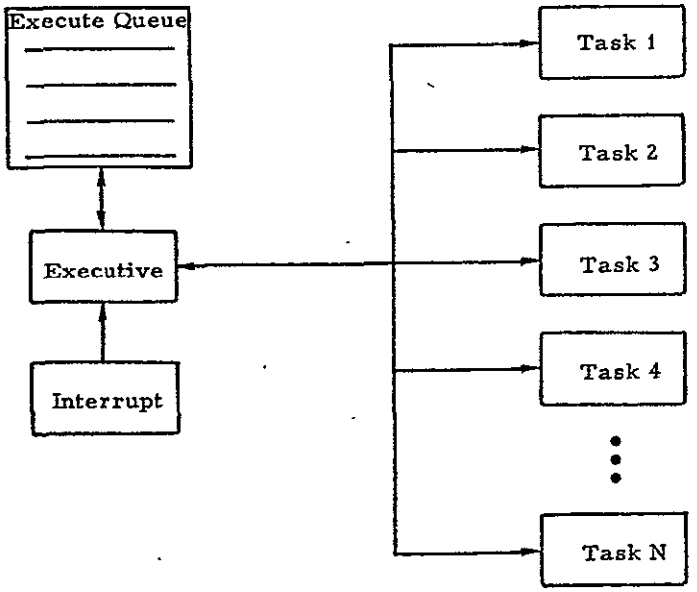


Figure 5. Interrupt Executive Concept

However, rather than waiting until a task is completed before deciding what to do next, the decision is made whenever an interrupt occurs, which can happen -- and does -- while a task has been but partially completed. The interrupt carries with it an indication of why action is to be taken: this may be that some interval has passed, that some external device is ready to transmit or receive data, or that some hardware component or subsystem has failed. The first thing the executive must do is "service" the interrupt.

Interrupt-servicing must be able to execute at the expense of a task or job currently executing, and when the interrupt has been serviced the interrupted task or job must be able to resume at its point of suspension. Therefore, all registers used by the interrupt-servicing routine must be saved and restored in the event that they were being used by the interrupted task. Interrupt-servicing difficulty may be compounded by allowing interrupts of higher priority to occur while operating and servicing a lower priority interrupt. Allowing the interrupt-servicing routine itself to be interruptible requires that it be made reentrant, that is, able to execute a section of code before it has completed its previous assignment. Further, the time consumed in performing interrupt-servicing and scheduling functions must be minimized; for interrupts may occur very frequently, multiplying the effect of any executive inefficiencies.

The interrupt executive is composed of at least seven distinct functional modules as described below. The first of these, the interrupt module, is entered whenever an interrupt occurs. The interrupt module may in turn call any of the three input/output processing modules. The fifth module to be described, the scheduling module, is the portion of the executive to which each nonexecutive task transfers control when completed. The last two modules, entered either from the tasks or the executive, perform functions such as inserting new tasks and performing diagnostic checks.

1) Interrupt Module: Before determining the cause of and the response to an external hardware interrupt, this module (Figure 6) saves all volatile registers; to allow for reentrancy, there is a save area for each interrupt in the system. The module then determines the cause of the interrupt and takes the path directed. A clock interrupt results in updating time and interrogating the time-dependent task queue; if a task were scheduled, it would be removed from this queue and placed in the execute queue. If the new task had a higher priority than the executing task, the scheduling module would cause the new task to begin; if lower, the interrupt module would restore all registers and branch back

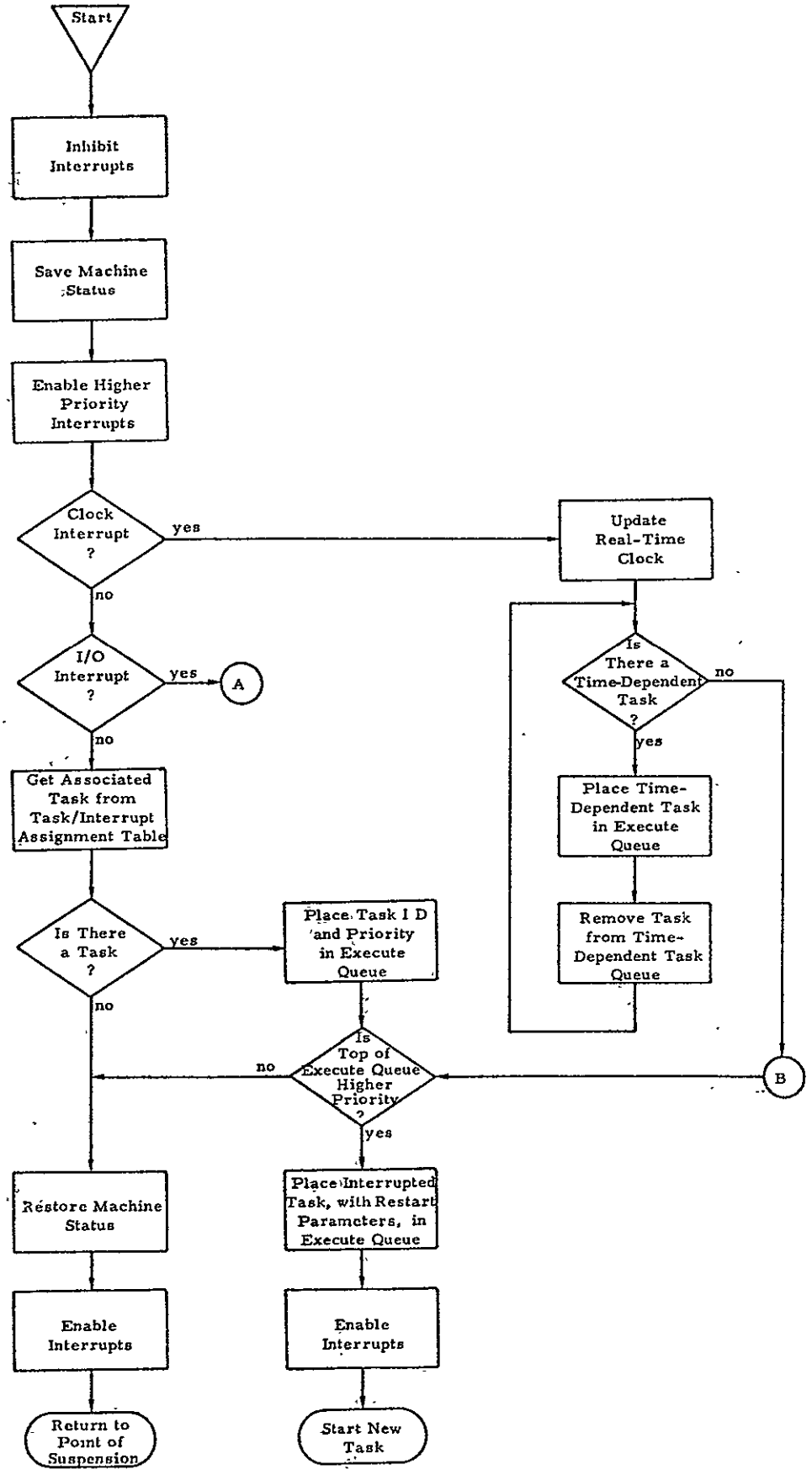


Figure 6. Interrupt Executive Design: Interrupt Module

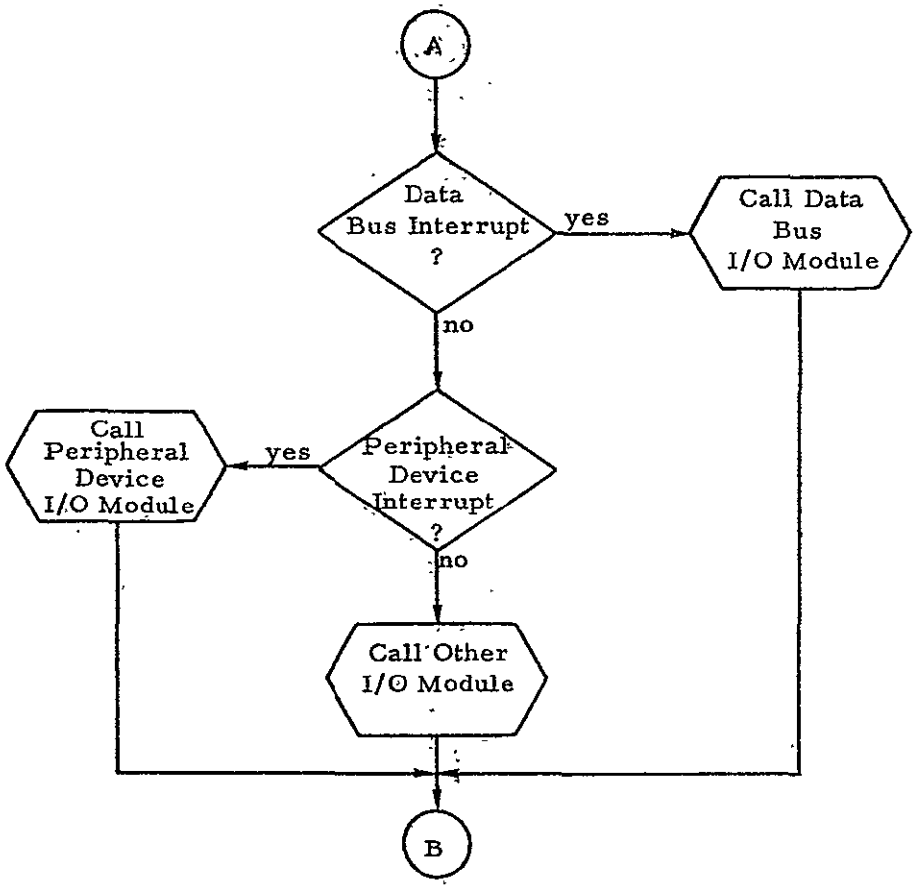


Figure 6. Interrupt Executive Design: Interrupt Module (continued)

to the suspended task. Similarly, the appropriate executive modules would be executed for other interrupts.

2) Peripheral I/O Module: This module handles I/O requests to peripheral devices such as digital displays, mass storage devices, and the like, providing the programmer's interfacing software with them. It initiates requests and responds to and resolves interrupts associated with these devices. In case of errors, retry attempts will be made; if still unsuccessful, this module will notify the executive to take corrective action.

3) Data Bus I/O Module: This module performs the functions necessary for communication via the I/O data bus. It

coordinates all activity on the bus, schedules the I/O such that optimum use is made of it, handles all interrupts associated with it, and monitors for any error conditions that may exist. Corrective action in case of failure or trouble is directed to the error recovery module.

4) Other I/O Module: All other system I/O is lumped into a package handled by this module. Falling in this group would be any analog or digital I/O and any I/O such as pulse counters or counters of some other sort.

5) Scheduling Module: After a task has completed its function, it branches to this module (Figure 7), which then accesses the execute queue for the next task. If the task in queue is a new one, the scheduling module branches to it to begin execution; if it is a previously interrupted task, the scheduling module must reset the registers before performing the branch to the task in question. If no task is waiting, the scheduling module starts the idle/self-test task. Before accessing the execute queue, the scheduling module inhibits interrupts, then enables them before branching to the next task; this ensures a graceful transition from task to task.

6) Task Support Module: This module provides the functional tasks with the ability to perform such operations as the insertion of tasks in the execute or time-dependent queues. Requests for temporary memory blocks are also handled by calls to this module, which can remain flexible for the addition of new functions suited to be included within the executive structure.

7) Error Recovery Module: This module has the responsibility to ensure corrective action in the event of hardware or software error conditions. Included in this module are self-test and diagnostic routines such that error conditions can be ascertained and recovery procedures implemented.

The interrupt executive has a distinct advantage over the polling scheme with which tasks need to be completed in a short interval before returning: the interrupt executive is able to respond to real-time events as they occur. That is, it can accept externally caused events, suspend task operation, and perform any function necessary to respond successfully to the event. The only delay would be in saving any volatile registers such that the system can

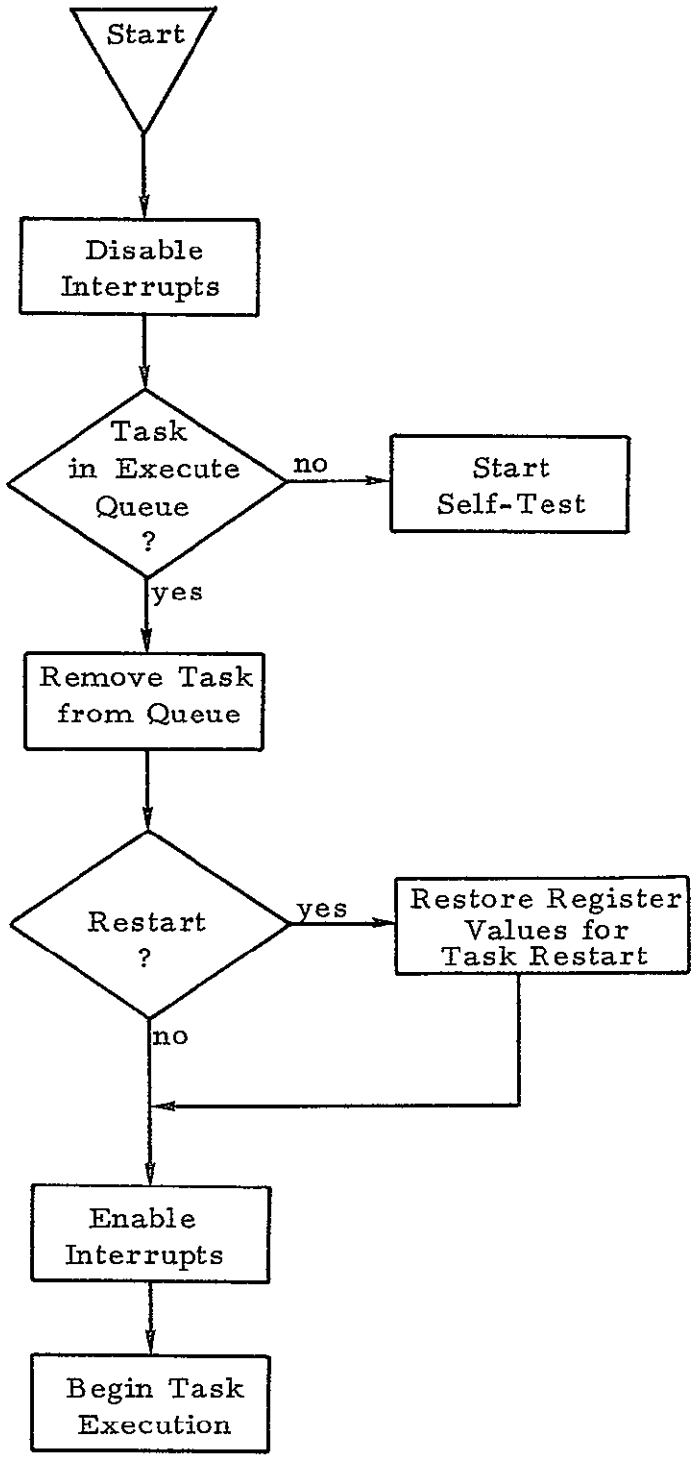


Figure 7. Interrupt Executive Design: Scheduling Module

resume execution at the point of suspension. One minor disadvantage is that during the response to an interrupt, the task currently executing is temporarily delayed. Another is that the time needed by the executive to preserve machine status must be included in executive overhead. The potentially most significant disadvantage, and one that requires discussion, relates to ease of verification.

A major verification problem with any software system consisting of a large number of tasks lies in assuring that the required interaction between tasks is correct and that no unwanted or erroneous interactions happen. As an example of the type of problem that can occur, consider the case in which one task is computing a vector and before it is done it is interrupted by a task that uses the same vector. It would be satisfactory if the interrupting task obtained either all the old values for the vector elements or all the new values, but wrong for it to obtain a mixture of old and new elements. What is needed in such a case is a means for locking out the interrupting program until all the new values have been computed. A similar, more serious problem -- and one often more difficult to detect -- occurs when a task places the results of its computations in some task intercommunication region, but before they can be used by the task for which they are intended they are overwritten by a third task.

This potential problem of task interference, while it also exists for the polling scheme, appears to be greater for the interrupt scheme owing to its greater complexity. Noninterference between two tasks can be assured for the interrupt executive by proving that the tasks can be executed in parallel, with no differences in the results except for those that are a consequence of deliberate and valid task interactions. For the polling executive, noninterference can be assured by proving that executing the tasks in any sequence produces the same results, except for the deliberate and valid task interactions. To permit comparing the verification difficulties for the two executives, the necessary conditions for assuring that no interference exists were analyzed. The analysis, presented in Appendix A, indicates that the polling executive is not enough simpler than the interrupt executive to permit a significant relaxation of the required amount of verification.

The keys to making an interrupt executive manageable with respect to verification are to require absolute adherence to standard task intercommunication mechanisms, restrict the number of interrupt levels, employ the selective inhibiting of interrupts over selected regions of task coding, and provide for restarting interrupted programs only at selected, consistent, safe places in the task coding. Use of these techniques

for an interrupt executive can greatly reduce verification problems, thus bringing it more into line with a polling executive as regards verification ease.

4.3.4 Distributed Executive/Multicomputer

In a multicomputer system, tasks would be permanently allocated to individual computers. With a two-computer system, for example, a likely allocation would be for one computer to execute guidance, navigation, and control tasks and the other to execute display and communication tasks. An executive is required for each of the computers, and it is assumed that the two executives will be similar. Continuing with the example, the display computer needs to access the current state of guidance and control to allow update of the displays, while the guidance and control computer needs to communicate to obtain the latest command information from the crew. Thus the executives require the ability to accomplish computer-to-computer communication; this is done through the computers' I/O provisions as indicated in Figure 8.

The need for intercommunication requires an additional module to be implemented in the interrupt executive design previously described for the simplex computer. Figure 9, which is a replacement for the indicated portion of Figure 6, shows how this module is included. The intercomputer communications module would perform the transferring of data or messages between computers. The transfer may be accomplished by a simple I/O device, or if greater sophistication and speed were needed it might involve some form of direct coupling.

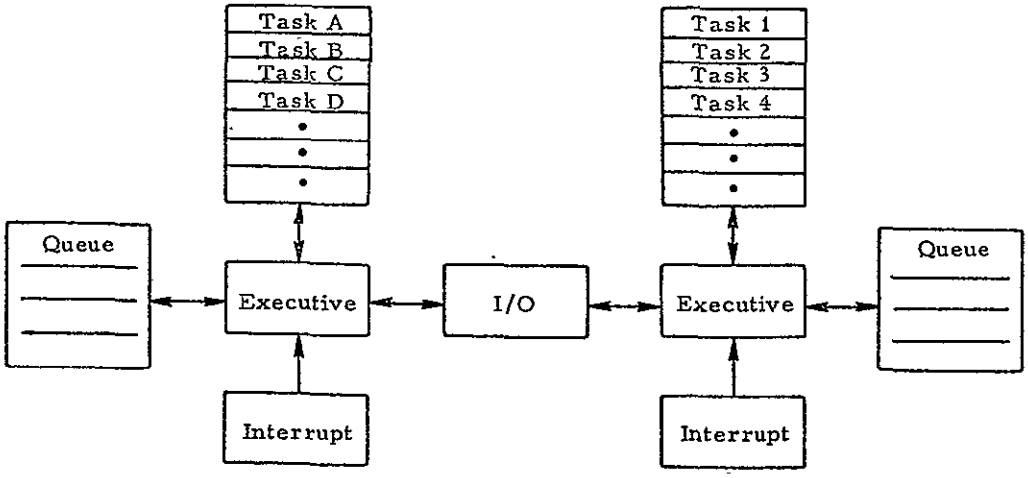


Figure 8. Distributed Executive/Multicomputer Concept

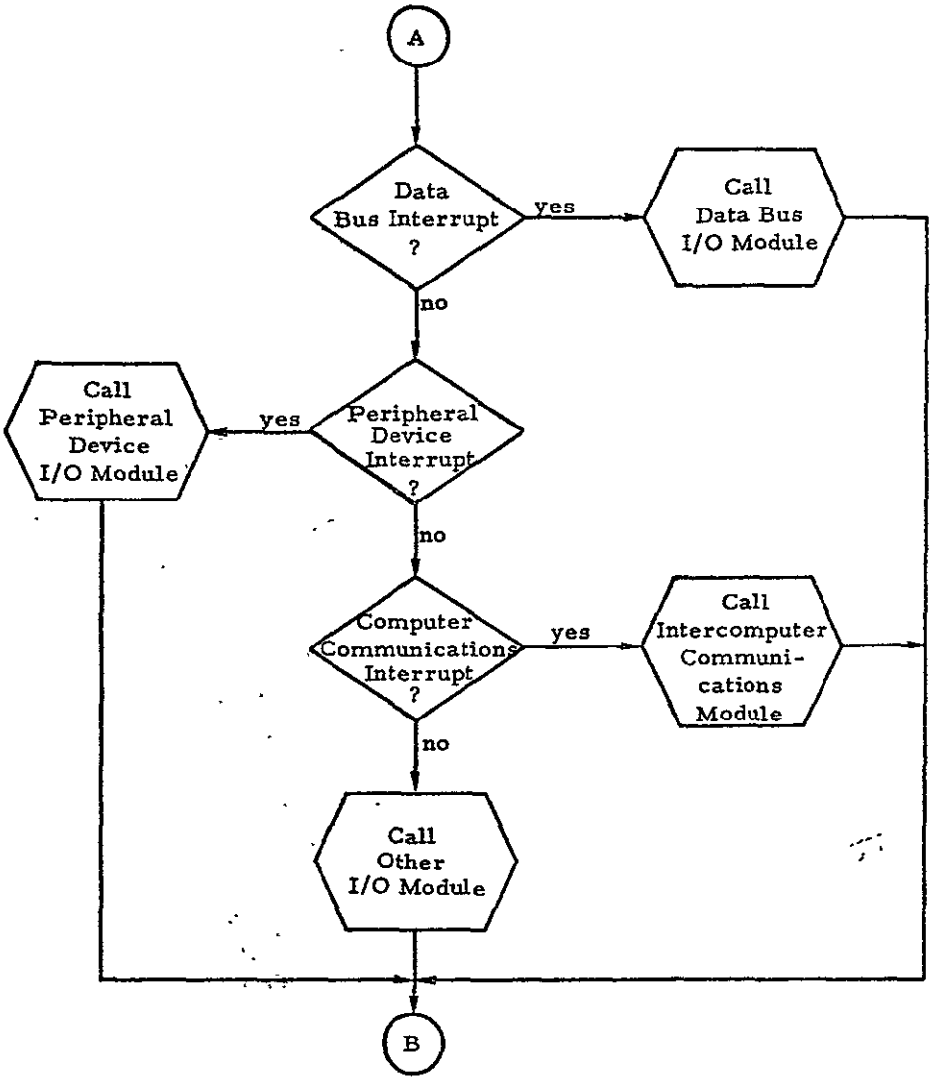


Figure 9. Distributed Executive/Multicomputer Design

The principal advantage of a multicomputer approach is the increase in system capacity it provides. The tasks can be logically divided into almost independent groups, with each group implemented on its own computer and executed as an entity. An interupt directed to one computer in no way interferes with execution on the other. This leads to another advantage: with the reduced probability that any given task will be interrupted, overall system response time increases. Of course the communication required between computers is a disadvantage in that a more complex executive is needed. Veryifying this executive is more expensive because

it becomes necessary to verify two or more executives running concurrently on different computers. Also, it is harder in this case to achieve repeatability.

4.3.5 Distributed Executive/Multiprocessor

In the multiprocessor distributed executive approach (Figure 10), the tasks would be allocated to particular memory modules and a processor permanently assigned to each partition. The partitions would be autonomous entities except for a particular portion of memory in each that would be accessible to all processors, providing an intercommunication block so that data and commands or requests could be passed from one partition to another at the cost of a memory cycle. Each partition would also contain an executive that would oversee the scheduling and servicing of interrupts associated with its resident tasks. A logical breakdown would be similar to the multicomputer approach: one partition assigned to guidance and control tasks and the other to display and communication tasks. The multiprocessor does not require the computer-to-computer communication necessary to a multicomputer configuration. The executive design could be the same as that for the interrupt executive on the simplex computer, with an additional module to access the common area in each partition. In the unlikely event that there is but little intercommunication between tasks, the necessary accessing controls could be handled automatically by the assembler or compiler, eliminating this module.

While similar to the distributed executive for a multicomputer, the distributed/multiprocessor has the advantage that intercommunication between computers is no longer needed. Transfer of data between systems is now done only by memory references through the appropriate executive module, simplifying the executive design. Because the memory is continuous, the executive offers more flexibility with regard to the partitioning of tasks and memory allocation. Its disadvantages are the same as those of the distributed/multicomputer executive; the repeatability problem exists, as does the problem of verifying multiple executives executing concurrently. In all, the distributed/multiprocessor is slightly better.

4.3.6 Fixed Executive/Multiprocessor

In this approach (Figure 11), one processor would serve as a dedicated processor for running the executive and the others would be allocated to task execution under control of the executive. The interrupt executive discussed earlier would serve as the base, with added coding to assign processors when a task is scheduled. When executing a task, the assigned

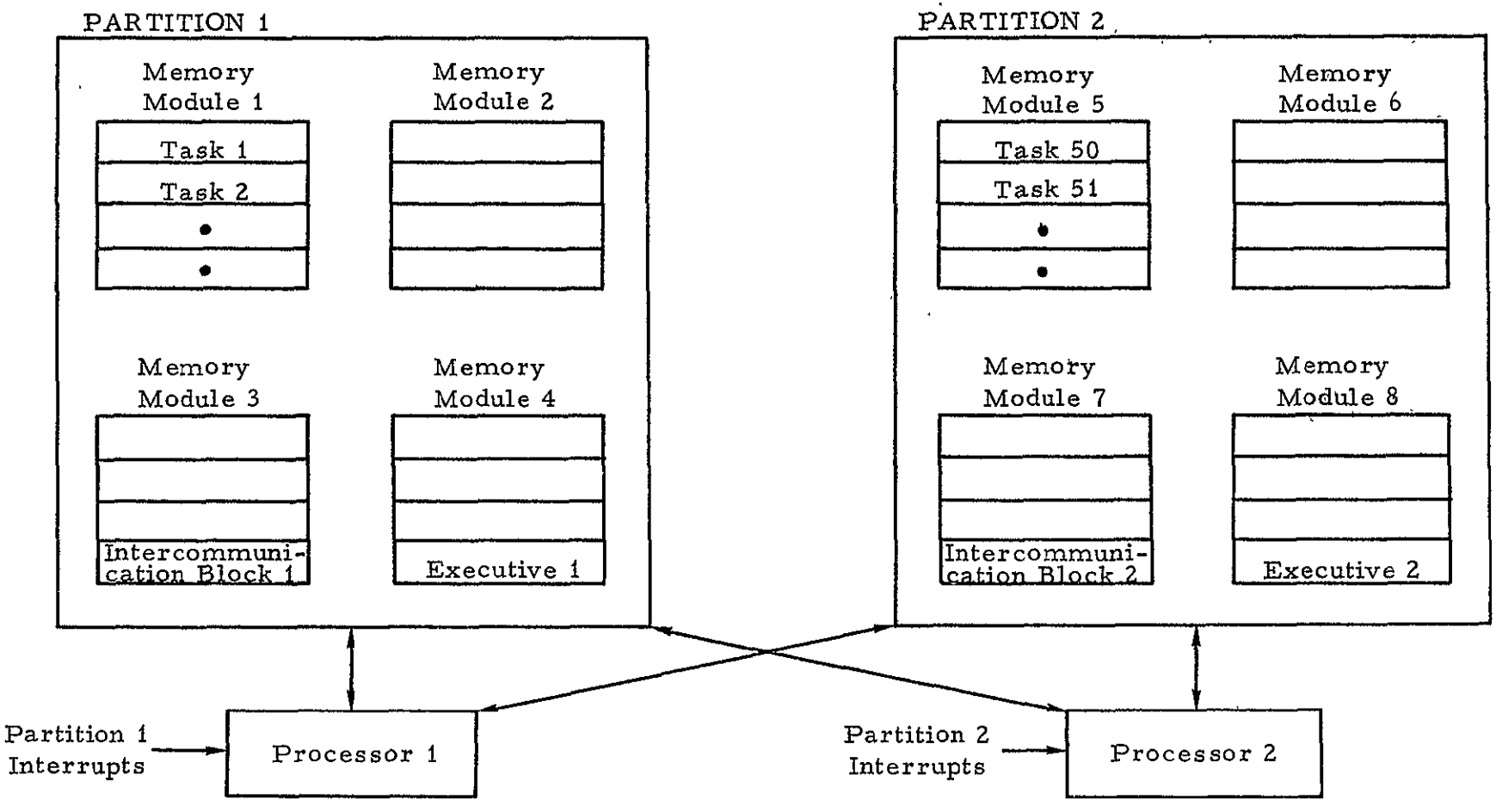


Figure 10. Distributed Executive/Multiprocessor Concept

processor would be limited to the area of core in which the task resided. A common access block, available for reference by all processors, could be used for temporary storage or for the passing of data from task to task. The use of this block would increase as the system evolved and would become more complex as new tasks and functions are added. However, the complexities introduced by the need for ever greater task intercommunication would impact only the executive and the common access block.

A fixed executive implemented on a multiprocessor offers advantages over any of the previously discussed executives. One is that processors are assigned to high-priority tasks such that the task is not interrupted when an external interrupt occurs. This means, of course, that high-priority tasks run to conclusion once started. This improves system throughput and response time. Flexibility, capacity, and failure handling are greatly improved because of the processor units available. If a timing problem develops, it may be possible to add another processor. If an error occurs in the processor assigned to any task, it is relatively simple to assign another processor to it. Thus, the backup problems facing the simplex and multicomputer configurations are more easily solved in this multiprocessor with fixed executive. This alternative does have verification disadvantages, however: since a task could be assigned to any processor at any time,

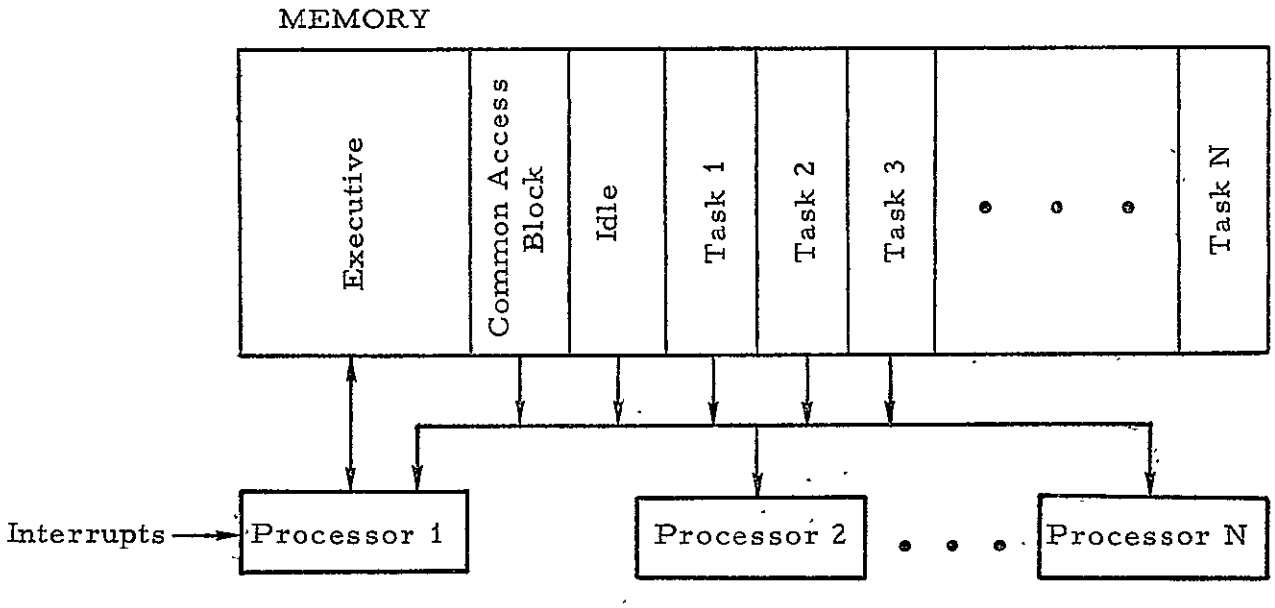


Figure 11. Fixed Executive/Multiprocessor Concept

repeatability cannot be assured from run to run. Thus response time and throughput advantages are gained at the expense of verification difficulty.

4.3.7 Floating Executive/Multiprocessor

The floating executive (Figure 12) allows the executive as well as the tasks to be executed by any processor and is a logical extension of the fixed executive. In this scheme the executive is considered another task or set of tasks, and would of course have the highest priority of all. Briefly, the executive would assign a processor to a particular task only when that task was about to be executed. A processor's execution of one task could be interrupted and that processor assigned to another, higher priority task. When a second processor completed execution of its assigned task, it could be assigned to the interrupted task; thus execution of a low-priority task might eventually be accomplished by many processors in succession. Executing the executive tasks themselves to accomplish processor allocation would be initiated either on completion of a functional task or on receipt of an external interrupt. The processor selector, either hardware or a combination of hardware and software, is required so that when an external interrupt is received only one

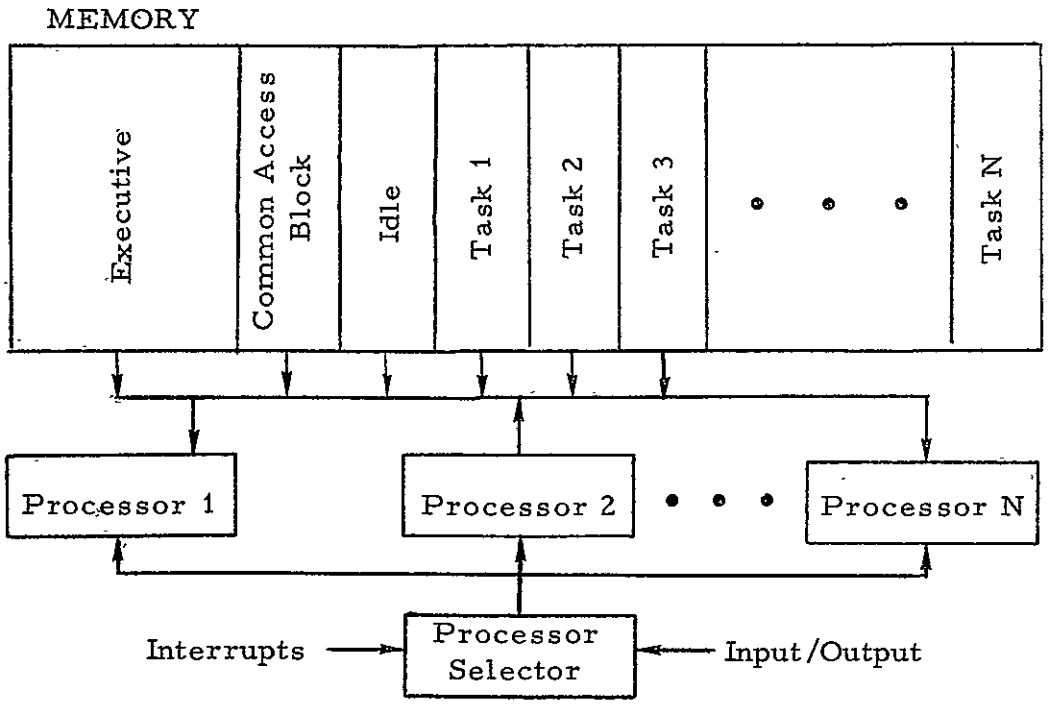


Figure 12. Floating Executive/Multiprocessor Concept

processor would be interrupted in executing a functional task. Task intercommunication would be controlled by the executive, utilizing a common access block to contain the shared data.

At some times there may be more processors available that tasks to perform; when this happens the special idle task would be activated to perform self-test and associated functions. It would be the function of the executive to make certain that the execution of this idle task be rotated among the processors and that even in times of peak computation loading the self-test portions of the idle task be performed occasionally.

Like the interrupt executive for the simplex configuration, the floating executive would be composed of many modules. Of these, the scheduling and interrupt modules are the most important. The scheduling module, for which a high-level design is shown in Figure 13, is executed at the completion of any non-executive tasks. This module must be reentrant: more than one processor may complete its task at about the same time, which means that the scheduling module may be executed any number of times nearly simultaneously. The scheduling module examines the queue of tasks to be performed; if one is waiting, it removes the task from the queue, relinquishes the queue, and begins execution of the task. If no task is scheduled, the idle/self-test task is begun and the execute queue is relinquished for further update. In either case, the processor/task priority indicator is updated to reflect current assignments, making it possible for the processor selector to route any interrupts to the processor performing the lowest priority task.

A high-level design developed for the interrupt module is presented in Figure 14. This module handles the dynamic assignment of processors to tasks in response to interrupts. It is executed whenever an interrupt occurs and, like the scheduling module, may be executed by any processor. The first thing that it does is suppress interrupts until the current machine state can be saved in a pushdown stack; after this, higher priority interrupts are enabled. Next the type of operation to be performed is determined and the appropriate operations performed, either by the interrupt module itself or by other executive modules. If the interrupt indicates that a functional task should be executed, that task is assigned to a free processor if there is one; here, a free processor is defined as one that is executing the idle task. If no processor is free, the priority of the interrupting task is compared with that of the task that was being executed by this processor before the interruption to do the executive functions. If the new task has a lower priority than the interrupted task, the new one is added to the execute queue, with its place in the queue a function of its

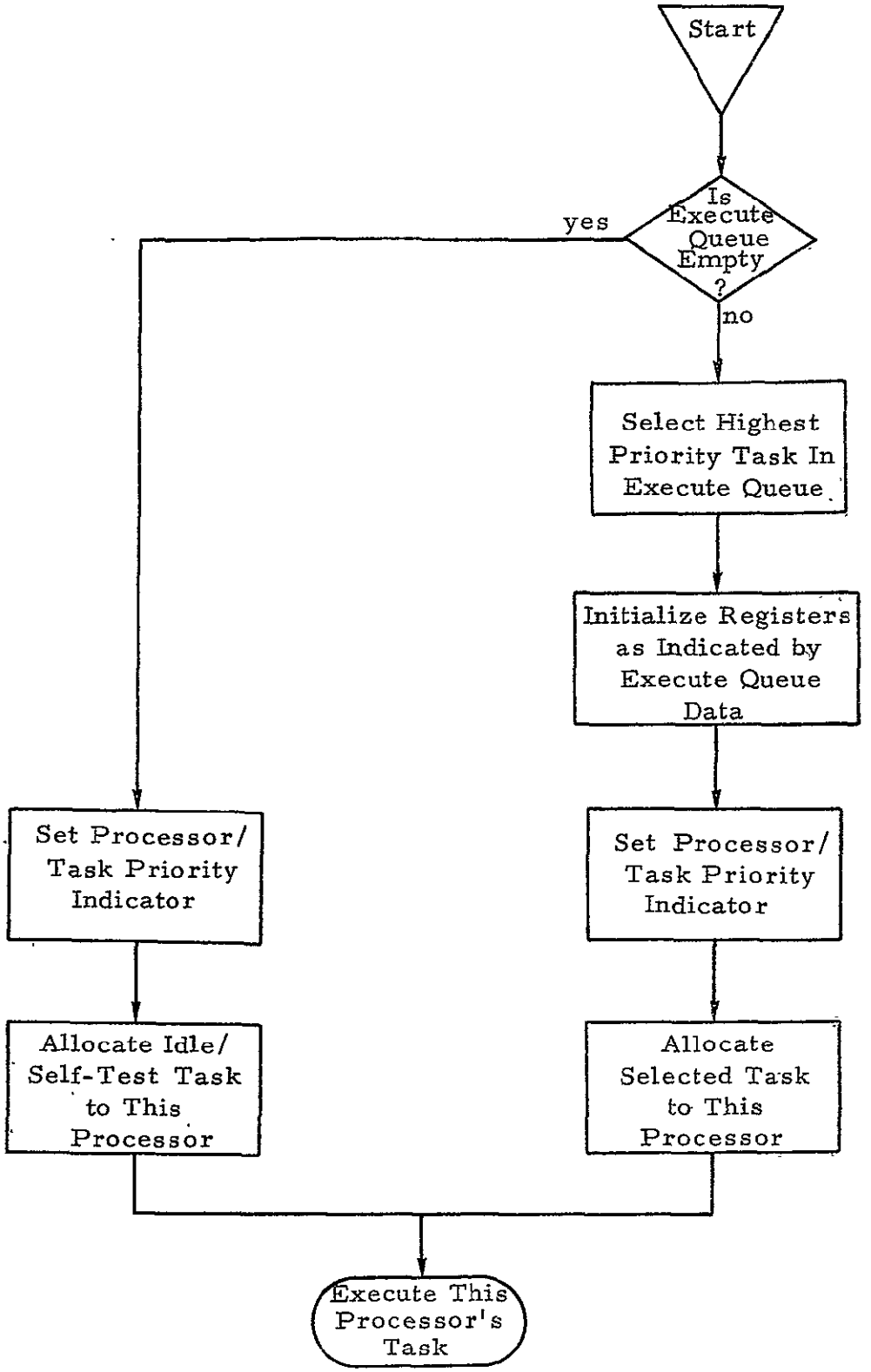


Figure 13. Floating Executive/Multiprocessor Design: Scheduling Module

priority, and the interrupted task is resumed. If the new task's priority is higher, the interrupted task is added to the execute queue along with information about machine state needed to resume its execution later, and the new task's execution is begun. The processor selector is then set to indicate which processor is currently performing the lowest priority task. When the next interrupt occurs, that processor will be the one to be interrupted.

The above description contains several simplifications. First, it allows for the scheduling of only one task in response to an interrupt; further elaboration is required to schedule tasks in processors other than the one already interrupted to do the executive operations. Second, it relies on random characteristics of the tasks themselves and their scheduling to schedule the idle task in all processors within an allotted period; in practice, the executive will have to assign the idle task to a processor in the event that that processor has not done the idle task recently, even if doing this results in unnecessary reallocation of another task or a slight violation of the priority structure. Finally, the high-level description does not illustrate the provisions required for computer reconfiguration in the event of a processor's hard failure; this will require additional queues for describing processor status.

One advantage of the floating executive/multiprocessor approach is that all pieces of the system are continually being tested for correct operation. All processors are performing as identical parts of the total system, greatly simplifying the job of removing one processor from the active list if it fails. The floating executive offers still more advantages over the fixed executive with regard to throughput efficiency. Its response time is the best of any executive design because each processor is always working on one of the highest priority tasks. However, verification is much more difficult than for any of the other alternatives because of the much greater uncertainty as to which processor is doing what and the many combinations that can occur.

4.4 Executive and Computer Configuration Comparisons

Executive design criteria were defined and the performance of the preliminary executive designs estimated. The results are summarized in Table 9; the higher the numbers, the better the performance.

For the first criterion, size and complexity of the executive programs themselves, the implied executive is by far the best and the floating executive the worst. The distributed executive for the multiprocessor is

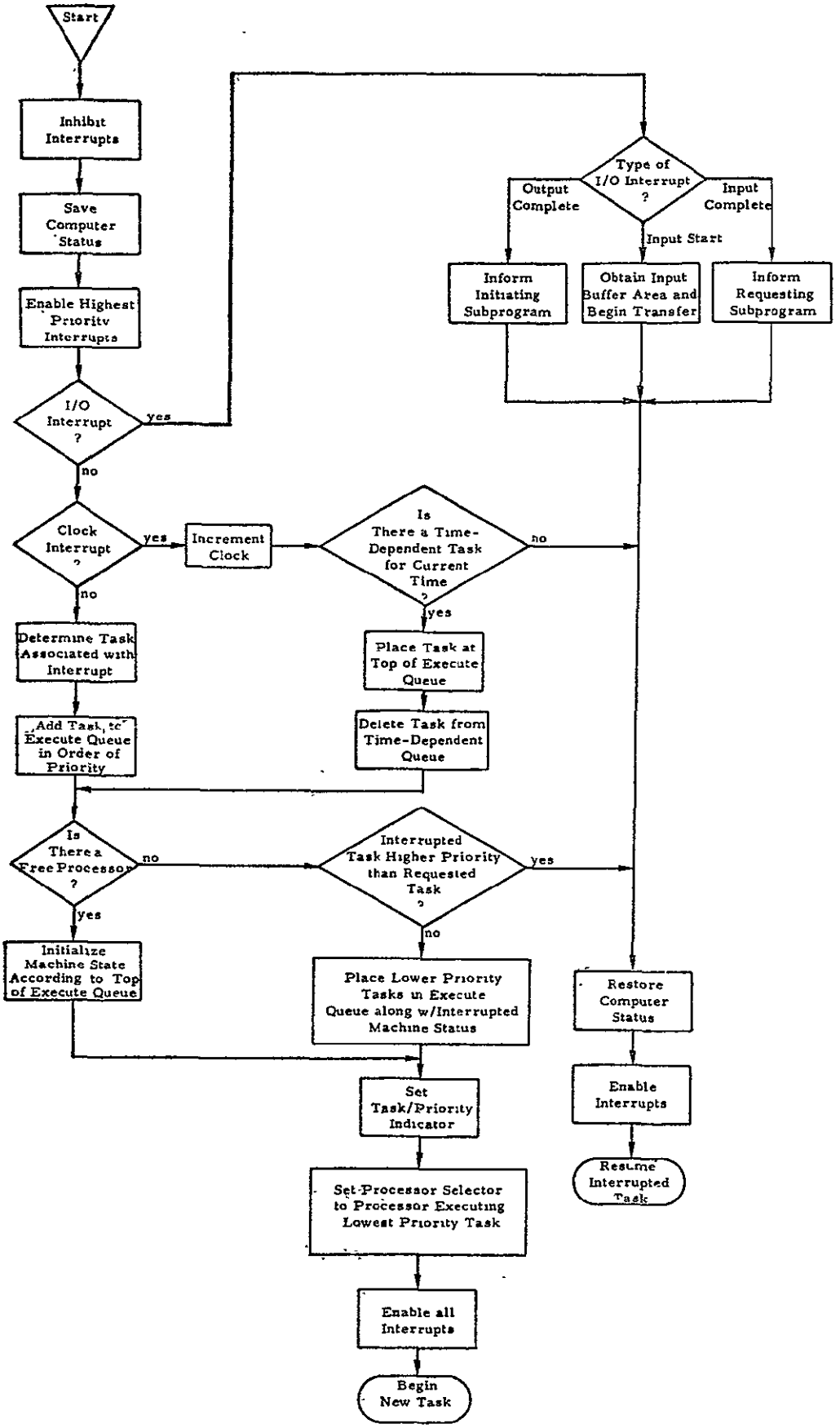


Figure 14. Floating Executive/Multiprocessor Design: Interrupt Module

Table 9. Executive Design Comparison

Criterion	Simplex			Multicomputer	Multiprocessor		
	Implied	Polling	Interrupt	Distributed	Distributed	Fixed	Floating
Size and complexity	40	20	10	7	8	5	4
Overhead	40	12	10	10	10	9	8
Capacity	1	8	10	20	20	20	25
Response time	1	5	10	11	11	12	14
Flexibility	0	5	10	11	13	15	20
Failure handling	0	0	10	10	10	15	20
Verifiability	20	15	10	7	7	3	1

slightly smaller and less complex than its counterpart for a multicomputer because the use of direct memory access rather than normal input/output simplifies task intercommunication. Estimates for the second criterion, executive overhead, follow much the same pattern, with little significant difference between polling, interrupt, distributed, and fixed executives. Executive capacity, which is a measure of the number of unique tasks that can be accommodated without requiring any executive modifications, is significantly better for the multicomputer and multiprocessor configurations than for the simplex computer. Executive response time of the interrupt scheme is much better than that of the polling executive. Flexibility, the capability of being easily modified to accommodate new system requirements, and failure handling, the capability to dynamically react to and compensate for hardware failures, are likewise significantly better for the interrupt executive. Finally, verifiability follows the same trend as size and complexity, with the simplex polling executive better than the simplex interrupt executive, much better than the multicomputer and multiprocessor distributed executives, and very much better than the fixed or floating executives.

It is concluded that the simplex configuration will lead to significantly simpler software than the multicomputer or multiprocessor configurations. Compared with a polling executive, the advantages an interrupt executive offers with regard to capacity, response time, flexibility, and failure handling compensate for its greater verification difficulty. The multiprocessor is slightly preferred over the multicomputer configuration if the distributed executive alternative is selected. Although the multiprocessor with the fixed or floating executive offers many advantages as far as capacity, response time, flexibility, and failure handling, the verification problems are so great that these executive designs should not be utilized for the Space Shuttle.

5. COMPUTER ARCHITECTURE

The software effect of the Space Shuttle computer configuration will largely be felt in the executive program, which, while the most difficult part to design and verify, will constitute only some 10% of the total onboard software. The effect of architectural features will impact the executive program and the individual task programs as well. Although no specific architectural feature will have as much impact on software design and production as will the choice of configuration, each one will apply to much more of the software.

Basic criteria for determining software impact are discussed in this section in terms of the suitability or adequacy of the following classes of architectural features:

- Memory
- Execution speed
- Input/output facilities
- Instruction set
- Word format
- Register organization
- Restart and self-test provisions
- Interrupt-handling facilities

The discussion here applies only to software. Considerations of hardware availability, cost, and reliability will be equally important in selecting a particular architecture. As the succeeding discussion will show, some of the detailed architectural aspects will be important to the overall software cost impact, while others, although important for other reasons, will have a small effect on cost.

The criteria, because they apply to the total onboard software, also have implications as regards the computer configuration. In particular, significant architectural differences are possible between the computers at the local and central levels in a federated configuration. In such a configuration, a significant portion of the software will be for the local computers. If these local computers are unsuitable in terms of, say, their instruction set, then the overall software effect will be increased in proportion to the amount of local level software.

Before proceeding with the discussion of individual architectural aspects, it should be pointed out that for a given set of functional requirements there

is a level of performance with respect to the first three -- memory, execution speed, and input/output facilities -- that must be satisfied if adequate software is to be developed at all. Uncertainty in Space Shuttle avionics subsystem hardware and mission requirements definition has not permitted this minimum level of performance to be determined in the present study. Whatever level is ultimately determined, capabilities in excess of the minimum will have definite advantages in reducing software costs.

5.1 Memory

Memory size, characteristics, and access methods are all extremely important in programming. Software production is greatly facilitated when these features are suitable to both the application and the programming techniques employed.

5.1.1 Memory Size

Memory size is the single most important hardware characteristic affecting software development. If memory is inadequate, even simple programs become difficult to write, with many iterations of reading and memory reallocation to ensure that all routines will fit. The immediate result is that the programmer is forced to concentrate on efficient memory utilization and not on the problem being solved.

A very serious complication resulting from inadequate memory, especially in verification, is usage of the same memory locations for temporary storage of unrelated data items created by separate tasks. This is possible, in theory, if the first task retrieves its temporarily stored data item before another task attempts to utilize the same location. The second task must retrieve its data item before the first task utilizes the same location again. Like bigamy, this scheme works only if perfect separation can be guaranteed. In practice, the common sharing of memory leads to unwanted and unpredictable interaction between otherwise independent tasks. The problem of memory sharing also applies to the sharing of subroutines that can be accessed from two or more tasks. If the routine cannot be duplicated in both tasks, either it must be demonstrated that the one task never attempts to reference the common subroutine before the other task has finished with it, or the subroutine must be designed to be reentrant. Such memory-sharing problems can exist for both polling and interrupt executives, but are more difficult to solve for the latter.

Memory locks are one hardware feature that can be used to alleviate interference problems. However, their use in controlling the accessing of data

that must be shared should not be complicated by also using them in situations where data sharing is not actually required. Certainly memory locks must not be considered a substitute for adequate memory; for where memory is already restricted, the additional memory needed to operate them often prevents their use.

It is very difficult to predict just how much memory a set of routines will require. In the present study, the minimum and maximum memory size estimates (Table 2) varied by nearly an order of magnitude. Further, the prediction process, or "sizing," becomes harder the more the new system differs from existing systems, and of course the Space Shuttle system will be unprecedented in many respects. The answer to the sizing problem is threefold. First, the functional analysis activities should be continued up to the point where actual onboard software coding has been completed to ensure that proper actions can be taken to keep memory size compatible with software functional requirements. Second, the computer architecture should have an extensible memory organization as discussed later. Third, sufficient surplus memory should be obtained initially. The amount of surplus depends in part on the uncertainty in the software functional requirement; in view of the present uncertainty, a surplus of 100% would not be excessive. Even when the subsystem hardware configurations and software requirements are firmly established, a surplus of at least 40% should be provided. This will ensure sufficient memory to permit a high-order language, with the attendant compiler inefficiencies, to be utilized.

One problem with surplus memory is that its very existence causes the addition of software functions which, although not required, are added to the system because it is possible to do so. Preventing this unneeded software growth requires both strong management and a recognition of the positive benefits that surplus memory can offer to the reduction of software costs.

5.1.2 Extensible Memory

An extensible memory capacity is highly desirable for both hardware and software efficiency. With such a system, memory capacity is varied by adding or removing memory modules (or banks), which may range in size from 4K to 32K words. With extensible memory, the capacity can be adjusted to hold the software, rather than requiring the software to be adjusted to fit the memory. Any computer with fixed memory size is undesirable; it is almost certain to be either too small or too big. If it is too small, programming costs rise sharply. If it is too big, a penalty is paid in excess hardware and excess weight. A broadly extensible memory consisting of small modules need never be too big or too small by more than 5-10%.

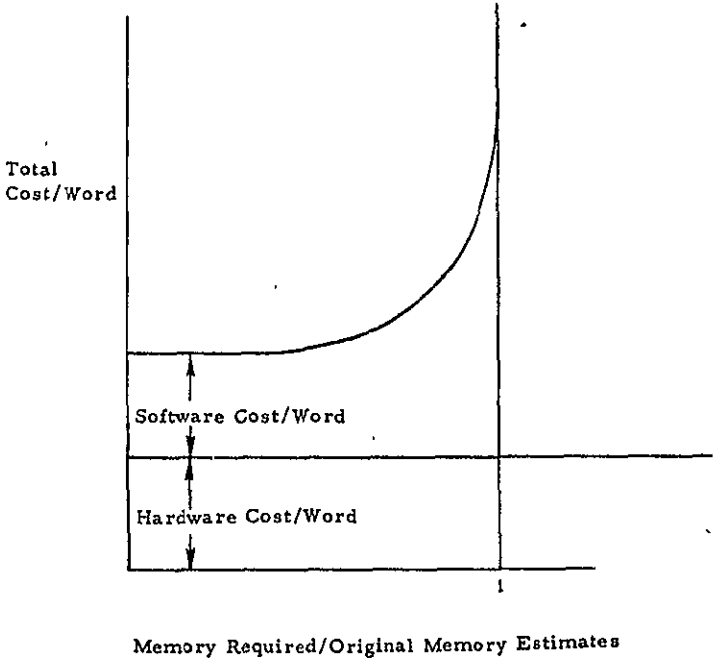


Figure 15. Fixed Memory Costs

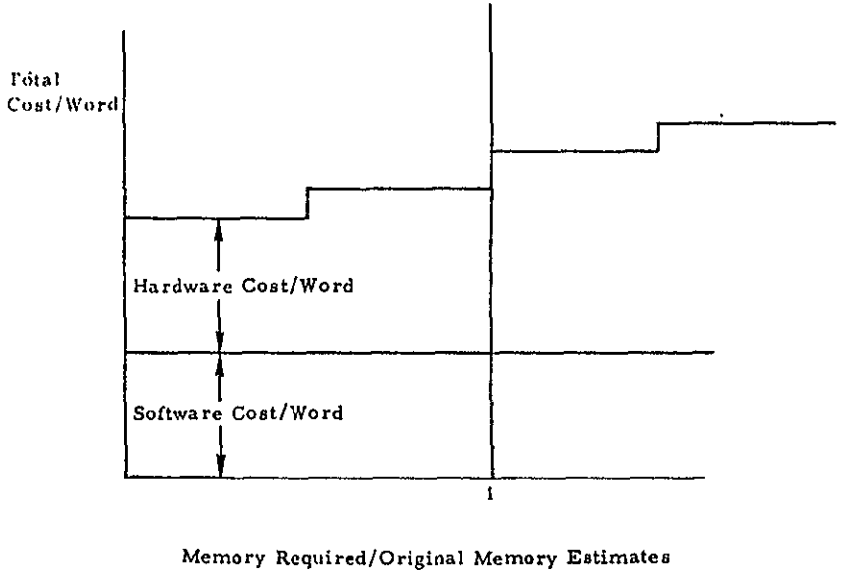


Figure 16. Extensible Memory Costs

Figure 15 illustrates, for a fixed memory size, the relationship between total memory cost per word and the ratio of actual memory size to the original memory size estimated and purchased. The hardware contribution to total cost for such a memory configuration is constant, but the software contribution increases greatly the more closely the memory required approaches that available. Figure 16 illustrates the total cost per word for an extensible memory architecture. With this architecture, the software costs involved in coding a given function can be relatively constant because enough memory can always be added to avoid the problems induced by inadequate memory. The hardware cost would be higher for an extensible memory, and would increase in discrete jumps as more memory had to be added to maintain the fixed software cost. However, the sharp increase in software cost is avoided, so that total cost per word is lower when actual memory required closely approaches or exceeds original estimates. Thus the availability of the extensible memory feature would greatly reduce the sensitivity of Space Shuttle software costs to program size underestimation.

5.1.3 Auxiliary Memory

Supplementing the main memory of many large computing systems is a large, relatively slow auxiliary memory. Software development is simpler and less costly without an auxiliary memory, but the hardware cost and weight advantages of supplementing the central memory can be significant. A detailed analysis of the requirement for auxiliary memory is dependent on full definition of Space Shuttle mission requirements. Study of the available data suggests that an all-main-memory arrangement may be desirable because:

- The executive routines must be largely, if not entirely, resident in main memory at all times.
- At peak load, the bulk of all lines of code concerned with mission operations will be in main memory at once.
- During the most active mission phases, most of the mission control program will almost certainly be required to be in main memory at the same time.

5.1.4 Read-Only/Read-Write Memory

Hardware reasons -- reliability considerations, reducing electrical power consumption, and so forth -- may exist for dividing the memory

into read-only and read-write portions. Temporary data items and variables are allocated to read-write, and instructions and constants to read-only memory. The partitioning has little consequence to programming efforts so long as there is enough of each type. But if the program will not fit in the read-only section, or if all variables, tables, etc., will not fit into the read-write section, the effects are the same as those of having insufficient memory: development and verification costs are both increased because routines must be tightly packed, causing increased interaction between unrelated routines.

If all memory is read-write, there is a small penalty in confidence (or in cost of establishing confidence) in software quality because a task's program or constants can be erroneously overwritten by another program. In the case of read-only memory, verification must make allowance for attempted erroneous writing; even though erroneous writing is inhibited, it must be searched for as indicative of program malfunction. That is, while a stored constant will not be destroyed by the erroneous attempt, subsequent reading of the location by the faulty routine will not give the value that that routine expects. For a very large centralized system, the absolute protection afforded vital tasks (such as the executive or those that perform the control functions) from being destroyed by other tasks that have errors may justify less complete verification of the "unimportant" tasks without reducing confidence in the computer's ability to perform the vital ones. The reduction in verification effort, however, is not great, and any reliance on minimizing the effects of errors should not deter efforts to find them.

5.1.5 Virtual Memory

In a computer system with virtual memory, the memory apparent to the programmer is much larger than the physical size of main memory. Generally, main and virtual memory are divided into pages consisting of a fixed number of words, commonly 500-1000. Only a fraction of the total number of virtual pages resides in main memory at one time. If a routine attempts to access a word in a page that is not in main memory, that page is brought in from mass storage and a page not recently used is removed.

While generally aimed at time-sharing applications, a virtual memory might be considered for the Space Shuttle as an alternative to a fixed memory size or an extensibility feature. Programmers would not need to be concerned with running out of physical main memory, only virtual memory. This feature would also facilitate restarts or rollbacks after failures not

involving the mass storage device. When a page is brought into main memory, its image remains on the mass storage device. If a fault occurs, this page may be recopied into main memory and the program reinitiated at the point where the page originally entered main memory. In effect, a paged virtual memory performs main memory saves automatically as the pages are swapped from main memory onto the mass storage device.

In spite of its theoretical appeal, virtual memory has two overriding disadvantages for the Space Shuttle. First, the average memory access time is slowed considerably because of the time spent in swapping pages. Second, execution times vary unpredictably, depending on which pages happen to be in main memory. This, in turn, depends on how many other routines are running concurrently. Hence timing varies noticeably, limiting predictability and repeatability, both of which are very important to verification. Accordingly, it is concluded that virtual memory is an undesirable feature.

5.1.6 Interleaved Memory

Memory interleaving is commonly used in simplex computers and multiprocessors to minimize the average memory access time. With conventional distribution of addresses, assuming a computer having several memory modules, addresses 1, 2, 3, etc., would fall in module 1, which might be loaded down with requests for words while others remained idle. With one form of interleaved address distribution, address 1 would be located in module 1, address 2 in module 2, and so forth; in general, for a memory of n modules, address m is in module 1, where $i = m \pmod n$. Thus interleaving randomizes the pattern of accesses to memory modules.

On a simplex processor, interleaving is a desirable feature. Interleaving is invisible to the programmer; he can ignore the fact that it exists and still gain the speed advantage. That is, interleaving could save him the trouble of distributing program and data and variable storage areas so as to minimize access time. The variation in access time is predictable, so that interleaving does not make verification more difficult.

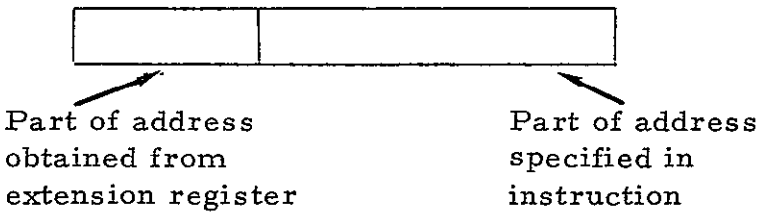
A different conclusion is reached for interleaved memory in conjunction with a multiprocessor configuration. Assuming that different processors had access to the same memory module, the execution time of one processor could vary depending on what functions another processor was performing. This would then make execution times vary, thus limiting predictability and repeatability. Since some of the effect of interleaving is lost anyway when two processors can access the same memory, it is concluded that interleaving should not be used in the general multiprocessor case.

5.1.7 Nonrandom Access Memory

Even though rotating disk or drum memory machines are still in use in some applications, it has been assumed in the previous discussions that the main memory is of the random access type. Nonrandom access implies variable access times and is considerably slower than random access. In many cases the programmer is required to structure data and program flow in a highly unnatural manner in order to decrease the average access time. This can be a major programming burden. Hence a computer with a nonrandom access main memory should simply not be considered for the Space Shuttle.

5.1.8 Addressing Range

In some computers it is not possible to directly address the entire memory. Rather, addresses in instructions represent only part of the true address, and, as indicated in the diagram below, the addressing hardware must concatenate an extension or base register onto the left end of the address specified in the instruction to obtain the complete address.



The purpose of this procedure is to conserve memory. Since addresses are shorter, instructions can be shorter; hence a smaller word length may be feasible.

Sometimes the use of extension registers indicates a kludged machine resulting from an initial design in which the memory proved too small for the intended application. The memory is enlarged, and the extension register is added to permit the larger memory to be addressed.

While offering no software advantages, the use of extension registers poses several disadvantages for software development. The extension register(s) must be loaded and unloaded by the programmer. A Logicon study of guidance, navigation, and control programming showed that 10% of the instructions involved manipulation of extension registers. In other cases where the average-size program module was well within the addressing range and there was little intermodule communication, less than 3% of the instructions were

devoted to manipulating the extension register. Based on these observations, it is expected that the penalty in Space Shuttle software costs would be between 3% and 10%, with the most likely value being around 6%. A secondary disadvantage is that programmers are induced to try to group instructions and data so as to minimize extension register manipulation. Not only does this add a further constraint to programming, but the grouping may warp an otherwise straightforward program and make it harder to verify. The conclusion is that a computer requiring extension registers to address its full range is undesirable from the software viewpoint.

5.1.9 Indirect Addressing

Normally an address specified in an instruction is the address of that instruction's operand. Some computer architectures also permit indirect addressing: the address specified in an instruction can be the address of the location containing the address of the operand.

Indirect addressing is nonessential but convenient. One application is in returning from a subroutine. For example, if any routine calling subroutine XI placed its return address in cell 251, the return from subroutine XI can be uniformly accomplished with a "JI 251" instruction, meaning jump to the address contained in location 251. A second application is in accomplishing computed GOTOs. For example, consider the function GOTO(A, B, C, D)I. Suppose pointers corresponding to labels A, B, C, D are listed in order beginning with address T. I is loaded into index register 6 and then the instruction JI, 6, T is executed. The effect is to add the contents of register 6 to T, obtaining the address of the cell containing the address to which the jump is made. A third application is in communicating values to a subroutine. Suppose a vector A(1), . . . , A(10) is to be passed to a subroutine as an argument. The straightforward method is to copy all 10 values into a region of memory associated with the routine. If indirect addressing is possible, only the base address of A need be passed to the subroutine. If the base address is placed in BA, instructions such as ADDI 6, BA will fetch the desired values from vector A. (The use of indirect addressing is likely to be indicated by a bit in the instruction, rather than as distinct instructions. Thus ADDI is really an add instruction with the indirect addressing bit set.)

All of these operations could be accomplished using indexing alone. Indirect addressing is merely a convenience, one that may save less than 1% in execution time and memory requirements or in software development costs. On the other hand, it is an invitation to intricate and overly clever coding, which can increase verification difficulty more than the relatively minor savings in programming effort realized. Assuming that programming standards were

established to alleviate the verification problem, it is expected that an indirect addressing capability would not affect overall software costs one way or the other.

Some computers provide cascaded indirect addressing: an additional indirect addressing bit in every word signifies whether the remainder of the word is an address of an operand or the address of another address. Thus the programmer may reference an operand through the address of the address of the address of -- ad infinitum -- the operand. Such cascading is even less useful than indirect addressing, and may complicate the development of diagnostic and verification tools. For these reasons cascaded indirect addressing is considered undesirable for the Space Shuttle.

5.1.10 Bound Registers

In the earlier discussion of read-only memory, the protection afforded a vital task from inadvertent destruction was outlined. A more flexible and effective way of doing this, and more, is to limit the region of memory accessible by a given program element by using upper- and lower-bound registers. If a routine tries to access a word whose address lies outside the limits contained in these bound registers, execution is halted and control is returned to the executive.

Bound registers would be useful during software checkout and verification and would assure that vital software functions are accorded proper protection regardless of malfunctions in other functions. Any routine which tried to fetch or write outside its legitimate region of memory would easily be detected so that the problem could be corrected or its effect minimized. Without this feature, errant routines would be able to sabotage the data and programs of other routines, causing errors that might be hard to trace or that could destroy vital instructions or data. In a very centralized data management system the use of bound registers would prevent propagation of hardware or software failures to other parts of the system, just as would be accomplished in a decentralized system through hardware separation at the computer level. Again, although this feature would simplify verification somewhat, it cannot be used as justification for a haphazard verification effort.

If easily controllable through software, bound registers may also make it easier to prevent unwanted interference between tasks. What is needed is the capability for one task to inhibit any other task from reading from or writing into a particular region of memory. If any other task attempted to perform the inhibited operations, its execution would be suspended until

either the particular bounds were removed or the executive program resolved the conflict. The provision of such powerful and flexible means of controlling task access to common data items can simplify software production and verification by accomplishing in hardware what would otherwise have to be done through software.

5.1.11 Memory Locks

Memory locks, an extension of the bound register feature, generally apply to smaller memory blocks. Whereas memory bounds establish the regions that a particular program can access, memory locks determine the programs that can access a particular region. A memory lock may designate whether the program's access is to be read-only or whether both reading and writing are permitted. Memory locks are conventionally implemented by having a keyword for each memory region; any program attempting to access the data in that region must supply a matching keyword.

As with bound registers, hardware memory locks would be useful during Space Shuttle software checkout and verification. However, they are cumbersome to use for regulating interaction of logically related routines unless designed specifically for the purpose. Software locks are more practical for such situations: the domain of a software lock can be tailored to individual groups of data, while the domain of a hardware lock usually cannot.

Hardware locks having the ability (already described for bound registers) to delimit the areas of memory accessible to a particular task and to suspend task execution and signal the executive when these limits are exceeded can eliminate much of the need for their software counterparts, with an attendant savings in software production and verification costs. Procedures for the use of hardware and software locks are discussed in detail in Appendix A.

5.2 Execution Speed

The speed of the processor will have a profound effect upon Space Shuttle program development techniques and on ease of program verification and modification. Unfortunately, the contribution of individual architectural aspects to the effective speed of a particular computer cannot be completely determined without actually developing the total program. However, several measures may be employed to approximate a computer's speed, including:

- Memory Cycle Time: the average access times for instructions and data located in the memory

- Add and Multiply Times: the execution times for relatively typical, simple, but commonly used instructions

- Composite Instruction Times: the execution time for a mix representative of the instructions that will be employed for the actual program

- Kernel Problem Times: the time to perform common basic operations such as matrix multiply and polynomial evaluation

- Benchmark Problems Times: the time to execute a collection of programs to solve limited but representative hypothetical problems

The advantage of the first two measures is their simplicity; their disadvantage is their inaccuracy. That is, they do not give a true indication of effective speed because they ignore the many other architectural features that affect speed, among them the memory size, operand addressing structure, instruction set, and register organization. The advantage of the last three measures is their greater accuracy; their disadvantages are their greater complexity and the greater amount that must be known about the problems the program is to solve.

At this stage of the Space Shuttle's development it has not been possible to define representative instruction mixes, kernel problems, or benchmark problems. This should be done when the software requirements become more complete, and the results used to evaluate the speed capabilities of proposed computer architectures. This section outlines the expected effect of adequate computational capability as determined by the recommended approach and proposes computational capability margins. It also describes the effect of computer architectures in which execution times are variable, depending not only on the instructions used but also on the data being manipulated.

5.2.1 Computational Capability

An absolute requirement exists for the processor to perform highly critical computational bursts within the allotted time. Beyond this, any additional available time can be very fruitfully used to permit conformance with desirable programming practices and standards and to reduce the costly attention that otherwise would have to be paid to program optimization.

As stated above, a computer's effective speed depends on many of its architectural characteristics. Computation can be speeded, for example, by reducing the amount of looping, but with a resultant increase in program size. Or, incorporating highly specialized instructions, such as square root, could reduce the number of instructions required to perform a particular function, again reducing the computational time for a specific task. These examples illustrate a classic programming tradeoff: that between the memory space and execution time required to accomplish a specific function. It is costly for a programmer to have to concern himself with whether space optimization or time optimization should be used for the task he is coding. Rather, he should be able to proceed in any reasonable and natural way. Even if forced to make the appropriate memory sacrifices to achieve the needed effective speed, the payoff is slight; speed savings over the entire program are expected to be less than 5%, even with the devotion of considerable effort.

To look at the question from another viewpoint, providing a high basic computational speed is a way to reduce the importance of other architectural features -- such as the number and power of instructions, the word size, and the register organization -- because it permits techniques such as interpretive subroutines to be used for operations that cannot otherwise be easily done. In general, an adequate speed margin will reduce software costs because it:

- Provides a Safety Factor: This may well be necessary if computational requirements are initially underestimated or are subsequently enlarged to make a software compensation for unanticipated hardware problems.

- Eases Programming: A limited computational speed leads to a requirement for difficult and costly time optimization.

- Eases Verification: An adequate speed margin permits a program to be designed and coded with close attention to clarity and organization and conformance with standards, enabling it to be tested far more easily, and also allows software self-checking and diagnostic features to be incorporated.

- Eases Maintenance: The programming approaches and routines can be developed and utilized in a more general way, in many cases obviating the need for changes or reducing the difficulty of making them.

It is recommended that the Space Shuttle computer provide a computational capability of 100% over the estimated maximum execution time. The minimum execution time margin should be 25%; and this margin is considered adequate only if the computational load can be accurately estimated and is not expected to change significantly during the software system's lifetime. A multiprocessor configuration can to some extent permit smaller speed margins if adding new processors to the system is relatively easy. This is analogous to the use of extensible memory in that the additional processors extend the effective speed of the computer system. However, the advantages of adding processors are not without their corresponding software costs: if the distributed executive design is chosen as recommended, some redesign and recoding of the existing task communication executive modules will be required, and some of the added computational capability will be lost because another, albeit identical, executive program will have to be supported.

The 100% margin is strongly recommended to eliminate the need for optimization or to allow the program to be expanded to perform twice its estimated functions. That this margin is not unreasonably high can be demonstrated by citing what happened with the Titan III space launch vehicle in the mid 1960s. In this instance, the flight controls were designed to operate at a frequency of 20 msec, and the onboard computer provided a 20% safety margin based on the estimated computation. This margin proved to be insufficient, even with extreme code optimization. The onboard flight control program was eventually redesigned and recoded; the major change was a reduction in computation frequency from 50 to 25 cycles/major cycle. A large computational speed margin would have eliminated a considerable amount of the extra effort and cost expended in redesign.

5.2.2 Uniform Execution Times

Inability to predict precise program timing, caused in a complex program by the many logic paths, produces great verification difficulties. The problem is greatly aggravated if instruction execution times are not completely known. One architectural feature that can prevent their being known is a variable-length multiply, in which the multiply time is a function of the contents of the multiplier. Variable instruction timing in association with interrupts makes it extremely difficult to predict precisely where interrupts will fall, thereby making it difficult both to predict and to test possible combinations. The problem can be somewhat alleviated by careful program design, but only at the cost of placing a considerable burden on the program designer for marginal gains of computation time. More discussion concerning

interrupts follows in Section 5.8; here the recommendation is made that the architecture of the computer be such that all computation intervals, of any size, be decidable.

5.3 Input/Output Facilities

It is convenient to regard I/O activities as data transfer activities. If the Space Shuttle central computer facility has two or more separate computers, it will be necessary to transfer data between them. If local processors exist at physically remote points, it will be necessary to transfer data between the central facility and these processors. If bulk memory devices are used, either disk or tape, data transfer will take place between them and the central facility. In discussing the Space Shuttle system, "I/O" and "data transfer" are therefore considered synonymous. A block is defined to be a contiguous set of memory words and a data transfer to be a copying of a block from one memory (disk, core, tape, register) into another memory. All that should really be needed to accomplish such a transfer is:

- Location of the first word in the "source" block
- Location of the first word in the "sink" block
- Length of the block (number of words)

In some cases, one or more of these items of information may be implicit. For example, if the I/O involves tapes, the block length may be fixed and so need not be specified. However, in many computer systems a programmer must do more than just specify these three basic items of information. From a software standpoint, a measure of the convenience in using I/O facilities is how much more a programmer has to do to actually accomplish a data transfer. How data transfer is accomplished is highly variable, and the commands and conventions involved are generally merely an arbitrary set established by the hardware designers. It would be unrealistic to try to enumerate all possible conventions and the software load they would impose.

Data transfer is commonly much slower than other computer operations. The Space Shuttle computer system will not be an exception. For example, a 1-MHz serial data bus would require 32 msec to transmit a 32-bit word, if overhead is ignored. A computer of reasonable speed can execute 10 or 20 instructions in this length of time. It is easy to see how a simplex central computer could spend all its time just on I/O if its CPU itself controlled data transfers and remained idle while the transfers were taking place. In light of the amount of data transfer that must be accommodated within the Space Shuttle, an independent I/O controller is required to perform the task on command from the CPU, leaving the CPU virtually free to perform other processing.

Two methods of interacting with the I/O controller have been used in the past: buffering and cycle stealing. The buffering approach is indicated schematically in Figure 17. The CPU transfers at high speed the information to be transmitted into a buffer memory associated with the I/O controller, then commands the controller to execute the I/O operation and proceeds with its own computation. The controller transmits the contents of the buffer memory word by word at the slower speed required by external devices. The controller may signal the CPU in some manner, for example by interrupting it, when the I/O operation is completed.

Figure 17. Buffering Method of Interaction with I/O Controller

The cycle-stealing alternative is shown schematically in Figure 18. Here, the I/O controller directly accesses main memory. Effectively, main memory serves as a buffer. The CPU merely tells the controller the memory address of the block of information to be transmitted and then proceeds with its own computation. Because of the slow speed of I/O rates, the controller needs to access the memory much less frequently than does the CPU; and when it requires a word for transmission, it preempts the CPU, that is, steals a memory cycle away from the CPU. Thus the execution of the program running when the cycle stealing takes place is delayed by one memory cycle every time the I/O controller steals a cycle. This can cause the executing program to lose repeatability. The points in time when the I/O controller steals a cycle and their frequency may vary depending on external circumstances such as bus or device availability. Transmission errors may require that a word be retransmitted; and this can result in an apparently random variation in timing of any program running while the cycle stealing is taking place.

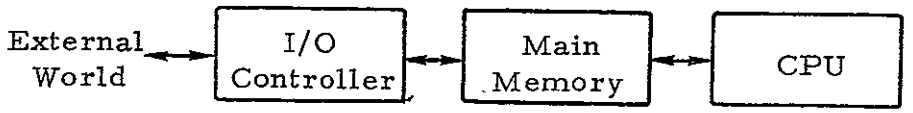


Figure 18. Cycle-Stealing Method of Interaction with I/O Controller

For a system using a 1-MHz data bus and a 32-bit word length, cycle stealing would occur no oftener than once per 32 μsec. If the main memory cycle rate is 1 μsec, the I/O controller will then steal no more than one out of every 32 cycles. (If I/O overhead such as repetition and parity operations are considered, the fraction of cycle stealing can be estimated as even lower.) If main memory is divided into modules, the probability of interference between the I/O controller and the CPU is further reduced. If there are N modules, the interference probability would be 1/32N, assuming that the CPU sequentially accesses each memory block every cycle. (With a large number of modules, it may be possible to treat one of them as a de facto buffer, making a cycle-stealing I/O system look like a buffered I/O system.) A reasonable number for N for Space Shuttle would be about 10, which would give a probability of interference of about .003. It can be concluded that the magnitude of the interference will be small and that speed degradation due to cycle stealing is negligible. However, a difference of just one cycle time can be enough to cause loss of repeatability for software testing and verification.

The tradeoff from a software point of view is between the slight added executive complexity required to handle buffer filing (which might be illusory if a whole memory block were treated as an I/O buffer under the cycle-stealing scheme) and the loss of repeatability encountered with the simpler cycle-stealing system. Since loss of repeatability is a serious complication for the software testing phase of development, the buffer option is judged to be more desirable.

Another factor affecting programming complexity is the method of queueing requests for shared I/O facilities. From a software point of view, the simplest is to have the queueing done by the I/O controller(s). A program would merely execute an instruction requesting an I/O operation and wait for a completion flag to be set in main memory. If the I/O controller(s) cannot queue requests, or if there is no I/O controller, the competition for shared facilities may have to be handled by software. The I/O device may be tested with a "test busy" instruction which causes a branch if the specified unit is in use. Priority conflicts between separate routines must then be handled by an executive routine.

5.4 Instruction Set

The Space Shuttle computer's instruction set will be the architectural feature of most continuous concern to the programmers. If there were no restrictions on memory size and task execution times, almost any required functions could be accomplished with a very small instruction set, albeit at a considerable increase in effort.

One measure of instruction set suitability is the number of instructions required to implement a set of benchmark programs; writing such programs for all candidate computer architectures and instruction sets and comparing the results would indicate the one for which software costs are likely to be lowest. This approach, while a valid one, could not be applied at this stage of Space Shuttle development owing to the difficulty of defining representative benchmark programs and the cost and time required for their implementation.

Many of the functions to be accomplished by the Space Shuttle data management system have counterparts in software developments for current space and missile systems. The usage of instructions in these applications was examined to provide a baseline from which the suitability of instruction sets for the Space Shuttle was evaluated. Figure 19, showing the frequency of instruction occurrence by type for a guidance, navigation, and control program, is typical of the kind of information utilized. The very high usage of load and store instructions in this application is immediately apparent; and substantial reductions in the percentage of input/output instructions, for example, would not have nearly as great an effect as a small reduction in load and store instructions. The number of shifting instructions is roughly half the number of arithmetic instructions, indicating the penalty paid in this architecture for the lack of floating-point arithmetic. Analysis of these and other data concerning the frequency of instruction usage has indicated the areas of most concern in satisfying the instruction suitability criterion.

Figure 19. Instruction Usage for Guidance, Navigation, and Control Programming

5.4.1 Basic Instruction Set

In addition to a minimum set of instructions without which accomplishing any but the most elementary functions becomes very difficult, an effective computer architecture includes those others that, although not as important as the mandatory instructions, will accomplish frequently needed elementary functions without requiring extraneous or redundant information or producing troublesome side effects. Table 10 shows the basic instruction set comprising these mandatory and highly desirable instructions. Some instructions, such as those for input/output, restart provision, and floating-point arithmetic, are treated separately because of their considerable individual impacts. It is assumed that the instructions for addition, subtraction, multiplication, and division manipulate operands consistent with the word length estimates presented in Table 2, that is, 24 bits for the minimum hardware configuration and 32 bits for the moderate and maximum configurations. If a shorter word length, say 16 bits, is provided, then double-precision arithmetic would be included in the basic instruction set.

Different architectures have different numbers and categories of programmer-usable registers into which the programmer should in general have the capability to load and store. Since the number of such registers is indeterminate until a computer architecture has been selected, the number of mandatory and desirable instructions is a function of N, the number of such registers. These numbers are 11 + 2(N) for the mandatory instructions alone and 30 + 2(N) for the entire basic set.

To determine suitability of the basic instructions provided by a particular computer, its instruction set should be compared with Table 10. If the computer's instruction set does not contain all the indicated mandatory instructions, it would be rated unsatisfactory and assigned a numerical value of 0. The occurrence of each basic instruction in a computer's repertoire would give a value of 1 to be added to the mandatory 11 + 2N. The highest score that could be achieved on the basic instruction set criterion is 30 + 2N. It is estimated that a computer with an instruction set scoring 11 + 2N would have 5-10% higher software costs than one with an instruction set having a suitability of 30 + 2N. Instruction suitabilities between these extremes would result in software costs in proportion. Instructions in addition to the basic 30 + 2N would have less cost impact than fewer instructions than the basic set. It is not likely that even a very large number of relatively elementary instructions in addition to the basic set would reduce software costs by more than 5-10%.

Table 10. Basic Instruction Set

Instruction	Category		Comments
	Desirable	Mandatory	
Add		X	
Subtract		X	
Multiply		X	
Divide		X	
Load (to register)		X	Number of instructions a function of number of registers
Store (from register)		X	
Register exchange	X		
Arithmetic left shift	X		
Arithmetic right shift	X		
Logical left shift	X		
Logical right shift	X		
Circular left shift		X	Either of these two instructions
Circular right shift		X	
Multiple register left shift		X	
Multiple register right shift		X	
Logical "and"		X	
Logical "or"	X		
Logical "exclusive or"	X		
Complement	X		
Return jump		X	
Unconditional transfer		X	
Skip/transfer on positive		X	Any one of these seven instructions
Skip/transfer on negative		X	
Skip/transfer on zero		X	
Skip/transfer on nonzero		X	
Skip/transfer on overflow		X	
Skip/transfer on equal		X	
Skip/transfer on not equal		X	
Store zero	X		
Increment	X		
Decrement	X		
Increment and transfer on condition	X		

5.4.2 Floating-Point Arithmetic

For the minimum Space Shuttle configuration it is estimated that roughly 60% of the onboard program will be devoted to arithmetic calculations; for the maximum configuration the proportion will fall to about 40%. In performing fixed-point arithmetic calculations, approximately 10% of the coding is devoted to rescaling operations and similar data manipulations that would not be required if floating point were provided. Thus eliminating the need for this data manipulation by providing floating-point addition, subtraction, multiplication, and division instructions would be of obvious benefit.

Floating-point capability does introduce some problems of its own, however. Testing for zero and for equality between two floating-point operands is complicated. Floating point also complicates modulo arithmetic, which is often used for calculations involving angles. It is expected that much of the data received from and transmitted to other vehicle subsystems will continue to be maintained in fixed point, thus requiring the proper conversions to be performed. Weighing the disadvantages against the far greater benefits attainable, it is estimated that floating-point instructions would reduce the size and complexity of the total onboard software by about 3-7%. This reduction in size and complexity would be directly translatable into equivalent cost savings.

In some systems that lack hardware floating point, the equivalent arithmetic operations are accomplished by means of subroutines written in a special machine-like language; a software interpreter causes their execution. This software approach to minimizing the effect of a hardware deficiency is undesirable for several reasons. First, it reduces effective computer speed by a factor of 10 at a very minimum and in some cases by a factor of over 100. Second, the subroutines and interpreter add to the size of the onboard program and to verification problems. Finally, the use of the interpretive mode greatly complicates the multiprogramming of independent tasks, for if one task is in the middle of executing a floating-point subroutine, any interruption by another task using that same operation must be prevented or else the subroutine must be made reentrant. Thus interpretive floating point should in no way be considered an acceptable substitute for hardware floating point.

Another approach to minimizing the lack of hardware floating point would be to utilize a higher-order language that allowed the range and precision desired for each fixed-point operand to be specified for arithmetic operations, and a compiler that automatically allocated proper scalings and inserted the

required scalings adjustments. Shifting the burden of fixed-point arithmetic problems from the programmer to the language and compiler in this way would be of some benefit, but would eliminate neither the analysis burden of determining range and precision nor the verification burden of demonstrating their adequacy and correct implementation. Furthermore, developing a compiler to accomplish automatic scaling allocation and readjustment is very difficult. Experience with the recently completed CLASP compiler for a fixed-point onboard computer indicates that compiler cost would be increased by at least 25% owing to the added complexity. And with the current state of the art in compiler construction, considerable help from the programmer in scaling allocation is still required to maintain precision and accuracy. Rather than relying on a language and compiler to obviate the need for floating-point arithmetic, hardware floating point is mandatory if a high-order language is to be workable on Space Shuttle.

5.4.3 Multiple and Subroutine Instructions

Some computers provide instructions that singly accomplish something that would otherwise require many instructions. Two categories may be defined:

- Multiple instructions by which the same elementary operation is repeated several times -- for example, a block copy instruction that moves many data items from one area of memory to another

- Subroutine instructions by which an operation is performed that would otherwise require a subroutine composed of different basic instructions -- for example, an instruction that computes the sine for the operand

Generally, multiple and subroutine instructions take much less time to execute than the sequence of basic instructions for which they are substituted, but much longer than any of the basic instructions individually.

The long execution time of multiple and subroutine instructions can introduce problems if the system also utilizes interrupts. While interrupts are not generally permitted to occur during an instruction's execution, it may not be permissible to lock out all interrupts for the length of time a multiple or subroutine instruction takes to execute. There is no software solution to this dilemma; and serious hardware and software complications are likely to arise if interrupts occur during instruction execution. Still

another problem, particularly with multiple instructions, is retaining the capability of restarting when a computer error is detected. If a computer error occurs during the execution of a multiple or subroutine instruction that reads from a memory block and writes back into that same memory block, restarting will not be possible unless the complete original memory block has been saved.

Unquestionably, multiple and subroutine instructions save memory and execution time compared with equivalent subroutines composed of basic instructions. Their utility is not great, however, because of their relatively infrequent usage. For example, when a sin/cos instruction was provided in a hypothetical computer, the guidance, navigation, and targeting program's execution time was reduced by 0.1% and its memory size by 51 instructions; a cross-product instruction reduced execution time by 2.7% and program size by 61 instructions. This was the type of program that, in the Space Shuttle application, would make the heaviest use of these subroutine instructions. The very small number of instructions saved indicates the small impact that subroutine instructions would have on overall Space Shuttle software costs. The execution time savings is somewhat more significant, although when the total software is considered, subroutine instructions would reduce execution time by less than 1%. It is concluded that the expected memory and execution time savings are not sufficient to offset the problems that multiple and subroutine instructions would introduce.

5.4.4 Unique Instructions

Nearly every computer has instructions unique to that computer alone. These instructions are quite complex, having many of the attributes of multiple or subroutine instructions, and accomplish operations appropriate only to a restricted type of problem; a well-known example is the "convert by replacement from the accumulator" instruction of the IBM 7094. The virtues of unique instructions and the means of employing them are not obvious; hence their usage is generally confined to experienced and inventive programmers and their frequency of occurrence is measured in tenths of a percent or less. The best that could be said about unique instructions is that they cause no harm, and even this is not always true. Unique instructions complicate verification because usually there is not a clear relationship between the computer operations that will be performed and the programming specification. Similarly, they complicate the creation of automatic verification tools.

It is not possible to describe all instructions that could fall into this category because of the special characteristics of each. The major indication is the

occurrence of an instruction that has no counterpart in any other computer and whose suitability to the class of problems being solved is not apparent. Frequency of occurrence by itself is not a reliable indicator; some instructions, such as restart and interrupt disable, may not be used very frequently but are essential when they are used.

5.5 Word Format

The format of instruction words and the type and format of data words all influence the relative ease of using a computer's instruction set. From the functional analysis described in Section 3, it was concluded that the data word length for moderate and heavy Space Shuttle processing loads should be 32 bits, based on the expected precision required. The following paragraphs discuss some of the more probable of the many formats possible for a basic word size of 32 bits and their effect on software implementation.

5.5.1 Instruction Words

The most common and straightforward alternative is to have instruction and data word lengths identical. This provides considerable ease in allocating instructions and data words throughout core. Typically, aerospace computers have instruction words that are either consistently shorter than the data words (the UNIVAC 1824) or are of variable length (the IBM 4 PI). The primary motivation for making instruction words shorter than data words is to permit full utilization of core memory. For example, if it is assumed that:

- full memory addressing to 256K words of core is allowed
- the instruction repertoire consists of 128 instructions
- the processor has seven index registers

then 28 bits would be needed to encode the above information. Thus if the basic computer word size is 32 bits, there would be at least 4 unused bits in all instruction words. Instructions such as shifting and those loading directly from the address field would require even fewer bits.

If the computer architecture permitted, it would be possible to fit 8 instructions into 7 basic computer words if these unused bits are employed. Such an architecture is not likely. Another alternative, and one that has been employed on some aerospace computers, is to reduce the number of bits allotted for memory and index register addressing and for instruction operation codes to allow shortening the instruction word. For example, the main memory addressing range could be reduced to 2K, permitting an instruction word length of 21 bits and allowing 3 instruction words to be packed

into 2 data words. Even more drastic, the memory addressing range could be reduced to 256, the number of operand-code-distinguishable instructions to 64, and the number of index registers to 3, permitting an instruction word size of 16 and allowing 2 instruction words to be packed into 1 basic computer word.

Even though the total hardware memory size may be reduced, such trimming of instruction formats to achieve a high instruction packing density in the basic computer word is very undesirable from a software point of view. The problems associated with the restricted memory addressing ranges have already been described (Section 5.1.8). Having to determine the instruction to be executed from bits in the instruction word other than the operation code or from some previously executed operation is similarly undesirable because the programmer often will have to specify more information in coding than would otherwise be necessary, and the verifier will have more things to demonstrate correct.

On the other hand, allocating instructions and data according to purely logical reasons, without having to follow coding restrictions regarding use of half-words and positioning of word or segment boundaries, makes it considerably easier both to develop the program and to train others to use and to modify it. Of course, verification advantage is gained in that the number of things that have to be checked is reduced. The computer code is more direct, avoiding peculiarities that might arise from packing; as an example, the use of partial-word instructions of the Honeywell 701P often requires insertion of do-nothing instructions because instructions cannot be broken across word boundaries. Thus it is concluded that the most effective instruction word format for Space Shuttle software development is one in which all instructions occupy a full computer word. There is no software advantage in having partial-word instructions, while there are noticeable disadvantages.

5.5.2 Data Words

Four data types are expected to be required for Space Shuttle programming: floating-point numbers, fixed-point or integer numbers, logical vectors, and alphanumeric strings. Ideally, the computer instructions used in manipulating each type should allow the data to be addressed directly. At the present time the relative amounts of each variety of data to be used in the data management system cannot be accurately estimated. Clearly, the actual mix will determine the utility of having special instructions for each data type.

For the vast majority of mathematical computations, floating-point numbers are highly desirable. The format must be fixed if the floating-point operations are to be executed as single instructions. The number sign, fraction,

exponent, exponent sign, and (in some instances) flag bits are extracted by the processor from the addressed word according to a fixed convention. Flag bits can be used to specify such things as options for overflow and underflow procedures. This seems to be an unnecessary level of sophistication for this application.

Fixed-point and integer arithmetic will still be highly useful even if a floating-point capability is provided. They are especially useful for non-equation-processing functions, such as defining and modifying contents of index registers, logic control parameters, looping control, and so forth. The general capabilities possessed by most computers are sufficient here. One helpful architecture feature for use when handling small integers is partial word control.

The final two data types are logical vectors and alphanumeric data. Logical vectors are simply strings of 0's and 1's. Ideally, it should be possible to manipulate strings of arbitrary length using only a small set of logical instructions. Alphanumeric data are characteristically of variable length; the natural unit is the single alphanumeric character. Editing and manipulative operations should ideally permit character-level addressing of strings of arbitrary length.

For any of the four data types there are advantages in being able to read or write portions of a data word without being affected by or affecting the rest of the word. This is accomplished by providing the capability to address half-words, quarter-words, or bytes. The ability to divide data words into smaller addressable segments has a different software impact from that of dividing instructions into half-words, providing additional capabilities and flexibility in data declaration as opposed to imposing burdens on the programmer. Partial data word capability facilitates masking of logical information, processing of Hollerith information, modification of branching, and so forth. Generally, this should be done on the basis of an orderly division of the word length resolved; for example, the division of a 32-bit word into 16-bit half-words and/or 8-bit bytes. This capability is extremely useful and is recommended.

An extension of this technique is to make a computer capable of handling a completely variable data word length. Computer "word length" is commonly defined as the number of bits retrieved by a single memory fetch. With variable-field-length addressing, the physical word length is of no great concern to the programmer: the memory appears to be a continuous horizontal string of bits. A data item is accessed by giving the address of the leftmost bit plus the length (in bits) of the item. Data of arbitrary length

may thus be stored or retrieved from any point in memory. There is no need to pack and unpack small items in a single word; long items need not be artificially divided into words; and no effort is required to extract subcomponents of a data item. Thus the programmer is free to let the actual representation reflect his internal conception of the data. For convenience, a fixed default word length may be provided for use when the specification of variable field lengths becomes burdensome.

This feature is considered undesirable for the Space Shuttle data management system. Not only would it provide relatively little additional utility in developing software, but it would also impose heavy burdens. It would require complicated and detailed interfaces to be defined between programs sharing data of different lengths and would increase verification difficulty because of the greater possibility for errors in interfaces and the many types of data manipulations that would have to be checked.

5.6 Register Organization

Registers are involved in nearly all basic computer functions. A large number of registers, interacting in many complex ways, increases both programming and verification difficulty. A minimum set may be too restrictive, necessitating additional data manipulation to accomplish simple functions, and this, too, will increase programming and verification difficulty. Factors influencing register suitability are reviewed here; it turns out that none of them has a major impact on software costs. Registers used to perform input/output operations are not included; these more properly designate a convenient way to interface the processor rather than serving as a means of facilitating problem solving using the computer.

5.6.1 Multiple Registers

From the viewpoint of facilitating software development, the optimal computer architecture in terms of registers is one that provides a sufficient number of general-purpose registers that can be used to perform all functions of the accumulator, quotient registers, index registers, and masking registers. Thus registers could be used as multipliers and multiplicands, or dividends and divisors, or to contain a bit mask to be used in performing logical arithmetic. Also, index registers could be included in these general-purpose registers, allowing index quantities to be computed and used without requiring intermediate load and store instructions. Not only would it be convenient for programmers to have several of these general-purpose registers, but providing several could also result in a savings of memory and execution time. In a typical guidance, navigation, and control

computer having a single accumulator and a single index register, it can be expected that roughly 40% of all instructions are register loads and stores. Providing several general registers would greatly reduce the number of temporary stores and recalls that would otherwise be required.

This is not to say that the number of registers should be expanded indefinitely. At some point, very little additional gains are made, and too many registers make it difficult to verify and change the software. For example, if the practice is followed of keeping items in registers through long coding sequences, information can be lost by inappropriate transfers. Another problem presented by large numbers of registers -- securing them in the event of an interrupt -- can be simplified by appropriate computer design; for example, by having blocks of registers that can be selected by a single instruction.

For the Space Shuttle, it is desirable that the registers be general purpose, thus providing flexibility, power, and simplicity for program development. An examination of the code produced by compilers for computers having multiple arithmetic registers indicates that seldom are more than six required to do a reasonable job of register allocation. Coding in assembly language often results in the use of as many arithmetic registers as are available, usually because of tricky coding or carefully tailored optimization, both of which should be avoided if software costs are to be minimized. Additional registers can probably be usefully provided up to about 10; more than these will provide only marginal gains in software development, while increasingly complicating verification. Finally, a rapid and simple hardware means should be provided for securing the contents of registers; this might take the form of a single instruction that saves all registers or one that switches between blocks of registers.

5.6.2 Index Registers

An index register is one whose contents can be automatically added to an address specified in an instruction, resulting in a new effective address. Two categories can be distinguished. True index registers are used optionally, that is, an instruction must specifically request that the address in the instruction be indexed. A base register's value is added to every address, whether data or branching.

Index registers are usually applied as follows:

- Accessing elements of vectors, matrices, and tables
- Looping control
- Many-way decision branches

- Providing reentrancy of routines
- Providing relocation of routines

The use of index registers for the first three applications is familiar and has no unusual requirements. The programming ease afforded in these applications indicates that the Space Shuttle computer should incorporate index registers; if it does not, some very similar substitute must be provided, particularly to facilitate vector, matrix, and table operations.

The fourth application, reentrancy, requires that each incarnation of a routine have its own data region. This is easily accomplished if the base address of the new data region can be loaded into an index register, and if the address of each datum can be augmented by the contents of that register. Since reentrant routines use the same program, it is important that the contents of this register not be added to addresses of branch instructions in the reentrant routine. Although in many cases the use of reentrant routines may be desirable, relying on such an indexing feature as the sole mechanism to enable reentrancy will lead to many verification problems. The fifth application, relocation, utilizes a base register. With this feature, a routine may be loaded into any part of memory and is relativized by loading the base register with the base address of the routine.

Reentrancy and relocation of routines facilitate program development only if complicated program structures are involved. Because little use of such structures is envisioned, the benefits obtainable from the last two applications of index registers are not great. Certainly the overall Space Shuttle software design should not attempt to relocate program elements dynamically; even if an auxiliary memory were used, the program elements loaded as a function of mission phase should always be loaded into the same memory addresses. Thus while index registers will not be required for reentrancy or relocation reasons in particular, their other applications are sufficient to justify their inclusion.

5.6.3 Register Stacks

One or more hardware-implemented register stacks, together with related machine instructions, can simplify or eliminate many load-and-store chores usually required in performing subroutine calls and complicated arithmetic and logical expressions. The Honeywell 701P is an existing aerospace computer which has such an architecture.

Basically, the implementation, shown in Figure 20, uses three registers (R1, R2, and P) coupled with a section of main memory. From a logical

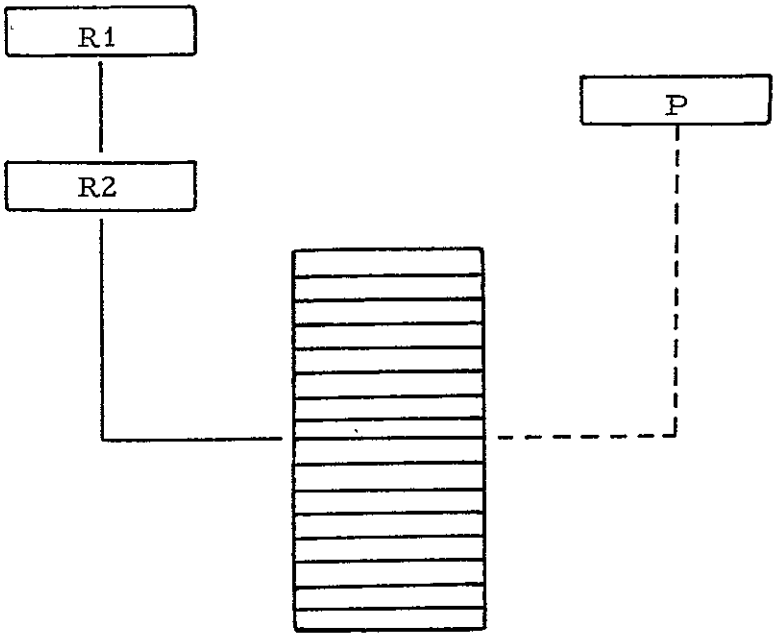


Figure 20. Stack Operation

point of view, R1 and R2 are the top two words in the stack. A load from memory brings a new word into R1 after first:

- Incrementing the pointer in P
- Moving the contents of R2 into the word in main memory indicated by P
- Moving the contents of R1 into R2

A store into memory reverses the procedure. Thus the load and store instructions respectively push down and pop up the stack. Operations such as "add" or "logical and" are performed on the top two words in the stack and the result is returned to the top of the stack. This concept retains data available in the stack for sbusequent use. If variables can be propitiously ordered, arithmetic can be performed with a minimum of loads and stores. Another slight advantage comes in terms of compiler operation, as there can be a closer correspondence between the compiler's intermediate language and machine instructions.

Subroutine transfers and returns can also be performed more rapidly and more easily if special subroutine branch and return instructions are provided in conjunction with a hardware stack. A return pointer and other necessary information are placed on a stack before transferring to the subroutine, and removed to effect the return. Adding one level of indirect addressing in conjunction with the stack enables all task routines and subroutines to be made automatically reentrant. As a routine or subroutine goes into execution, either through a subroutine call or initiation by the task scheduler, a marker is placed on the stack and the parameters and data for the routine are entered above it. Addressing of data local to the routine is relative. Prior to data fetch, the address of the marker is added to the relative address to obtain the absolute address. Essentially, the details of reentrant programming have been built into hardware.

With regard to whether the stacked register computer offers any advantages for Space Shuttle, it can be argued that it represents a way of making some software gains over a computer having a conventional architecture. However, most of these gains would not be significant compared to conventional designs having an appropriate set of instructions and register design, such as have already been recommended. It is concluded that the selection of either a stacked register machine or a more conventional design is relatively unimportant in terms of software implications.

5.7 Restart and Self-Test Provisions

Maintaining computer system integrity and detection of malfunctions is essential for the Space Shuttle computer. Restart and self-test are two classes of activities for performing these functions.

5.7.1 Restart

Aerospace computers are susceptible to transient errors caused by random events such as power supply surges or dips and radio-frequency interference. Such transient errors usually result in unwanted changes to the contents of volatile registers or other volatile storage devices. If the contents of volatile registers have recently been stored away in nonvolatile memory, it often may be possible to restart the computation by reloading them (including the instruction counter) from nonvolatile memory. This type of restart is a method by which software can recover from a large class of transient errors.

The minimum hardware provision required is some mechanism for detecting transient errors. In the simplest case this detection merely

causes an interrupt. If this is the only provision, the programming necessary to provide for the possibility of a restart is tedious: at frequent intervals the contents of all volatile registers must be stored away individually in nonvolatile memory, and an executive routine must be written to reload the registers when the transient error interrupt occurs. On a machine for which protection from such transient errors is desirable or mandatory, special provisions should be made both for storing the contents of volatile registers and for reloading them after an error is detected. For example, the Honeywell 701P has a save instruction which causes the volatile registers to be stored automatically. When a fault is detected, the volatile registers are automatically reloaded with the values stored by the last save instruction and execution is resumed without aid from software. The programmer is required to insert the appropriate save instructions, but the burden of doing so is much less than that of saving all volatile registers using the conventional store instructions. A method for inserting save-for-restart instructions into a program to guarantee correct operation after a restart is discussed in Appendix B.

To ensure that restarts will work after an error at any point, the save-for-restart instructions must appear frequently in the program code. Hence the delay encountered when a restart is performed will be the equivalent of only a few instruction times. However, saves could easily account for 10% of the total number of instructions, and thus represent a considerable load both on execution time and on main memory capacity. Furthermore, the problem of determining the proper points at which to insert the save instruction is not trivial. The conclusion is that a computer that is susceptible to transient errors and requires restart protection is less desirable than one that is not susceptible. If restart protection must be provided, it is very desirable that it be performed by a powerful save-for-restart instruction.

Of course, a hard failure can also trigger a restart. In this case, the machine might loop indefinitely if there were no other provisions. Commonly this problem is handled by a mechanism that counts the number of times a program has restarted at a given point. After a fixed number of restarts have failed, logic steps in to halt the restarting mechanism, possibly causing an interrupt indicating a hard failure.

5.7.2 Self-Test and Fault Diagnosis

From a software point of view, the best computer would be one that either did not require self-test and fault diagnosis or performed these operations automatically.

From a reliability point of view, it is not desirable that the only way of detecting faults be by means of software self-test. Self-tests cannot be performed continuously, and faults occurring during the intervals between tests could be calamitous. It is highly desirable that fault detection be performed continuously by hardware. Computers have been designed that automatically detect and automatically correct all failures in their hardware. An example is the STAR computer being developed at The Jet Propulsion Laboratory. For such a computer the burdens of reliability considerations, self-test, fault diagnosis, restart protection, etc., are removed from software development. This is an extremely desirable feature.

If the Space Shuttle computer cannot do all of the required error detection, correction, and diagnosis by hardware mechanisms, it is preferable that as much as possible of the error detection be done by hardware, leaving the required diagnosis and correction to be performed by software. This is consistent with the philosophy that simple tasks that must be performed at high frequency are best performed by hardware, while complex tasks of lower frequency are best performed by software. With such an architecture, hardware would generate an interrupt when an error was detected. The executive system would then transfer the current job to a spare unit, and diagnosis and cure would be performed on the failed unit. In the absence of complete hardware self-testing, software self-test is commonly used as a background task; that is, self-test is performed when all necessary tasks are completed. This ensures that the machine is operating correctly when a new work cycle begins. In the multiprogramming environment, self-test could thus be treated as the lowest priority task.

One means of performing self-test is by comparing the results of the same computations performed on two or more computers. The following example shows how the hardware efficiency of a triply redundant system may be compared with that of a doubly redundant system. The triply redundant system votes on all outputs. Errors are detected as a disagreement between one computer and the other two. The effective utilization is one-third; that is, if the machine were perfectly fault-free, only a third of the actual hardware would be needed. The doubly redundant system consists of two computers, each comparing outputs. Failure is detected by a disagreement in their outputs. Both computers must then self-test to determine which is actually correct. Because of the delay this job introduces, every task must be scheduled such that even if it is delayed by the self-test routine, all deadlines are met. For the periodically cycled tasks, if the computations require A units of time

and self-test requires B units, then the utilization is (1/2)(A/A+B). Hardware efficiency is higher for the triply redundant system only if 1/3>(1/2)(A/A+B), which holds only if the ratio A/B is greater than 2. Whether this ratio is actually greater than 2 will depend on the machine speed, the speed of the diagnostic routines, and the computations that must be performed at the highest frequency.

5.8 Interrupt-Handling Facilities

On real-time computers, an interrupt mechanism is commonly used to inform the executive that a request has occurred for processing a task or group of tasks. Interrupts ensure that the task in execution is always the important one. They are properly the concern of the executive system designers and ought to be invisible to programmers of non-real-time tasks.

Interrupts present verification problems, many of which have been discussed in Section 4. One partial solution is to minimize the number of interrupt occurrences possible during program execution and to reduce the amount of operations that must be performed by software when interrupts occur. This can be done by providing interrupt types suitable both to the interrupt source and the tasks invoked; an appropriate hardware priority structure and effective interrupt control mechanisms; automatic interrupt identification; and automatic computer status preservation mechanisms. Each of these desirable architectural features is described below. These features are very important to the reduction of software costs; further, serious architectural shortcomings with respect to interrupt features may result in software whose correctness cannot be feasibly demonstrated.

5.8.1 Interrupt Types

Three types of interrupts can be distinguished for the Space Shuttle application:

- Internal Interrupts: those generated within the computer itself to indicate the computer's and software's operation. Examples of the source of such interrupts include the completion of input/output operations, invalid attempts to access memory regions protected by locks, and hardware-detected redundancy and checksum errors.

- External Interrupts: those generated by other subsystems to indicate changes in their status and demands for servicing by the central computer facility.

- Clock Interrupts: those generated by hardware timers to indicate that a specified interval has elapsed or a specific time has been reached.

As regards internal interrupts, their number can be limited by requiring explicit checks in the software; however, such checks increase software complexity at the expense of interrupt simplification. Since most operations initiated by internal interrupts are relatively simple and in large part should be performed by the executive, this type of interrupt should not be eliminated entirely.

The number of external interrupts is a function of both the number of subsystems and the rapidity with which the central computer must respond to subsystem demands. External interrupts can be greatly reduced if not eliminated by having the communication of subsystem requests for processing received by the central computer through an executive poll of the subsystems. By having the executive ask the subsystems if processing is required rather than having the subsystems interrupt the current processing to tell the executive that some other processing is also required, significant reductions can be made in the number of interrupt occurrences and the associated verification effort. Only external interrupts which require very rapid or immediate processing, such as signals indicating the catastrophic failure of a subsystem, should therefore be tolerated.

The third category of interrupt is vital to maintain synchronization between the software and real time. Typically, a precision hardware clock generates an interrupt at a regular interval. The longest time that this interval can be is that of the most frequently executed periodic function. It is desirable that the interval not be very much shorter than that of the most frequently executed periodic function, or many unnecessary interrupts will have to be processed for every one that has any significance. One very desirable architectural feature is an alarm clock timer in which the interrupt interval is set by the executive, probably by storing a value in a special register. After an interval of the specified length has elapsed, an interrupt is generated. Aperiodic tasks that must be performed at some fixed future time can be scheduled using this type of timer. The alarm clock timer can substitute for the interval timer by simply storing the desired interval value back into the register after each timer interrupt. It can also serve as a watchdog to prevent certain tasks from taking too much time.

In a multiprocessor, only the processor that should do the indicated operations or the one with the lowest priority task should be interrupted for any of the three interrupt types. It is desirable that the processor to be interrupted be chosen by hardware means without interrupting another processor to perform an executive routine, since doing so would increase executive overhead and increase the number of combinations that must be verified.

5.8.2 Interrupt Priority Levels

In the Space Shuttle computer system there will be at best several dozen different situations that must generate interrupts. Five or six priority levels would suffice to distinguish the relative urgency of the interrupt requests. On the highest level might be computer hardware failure interrupts caused, for example, by parity errors. A slightly lower level might indicate manual hold, timer errors, and discrete command errors. Another level might indicate normal interval timer interrupts and requests for analog I/O. A fourth level might include console I/O interrupts and CRT-generated interrupts. A fifth level might indicate analog/digital conversion I/O interrupts.

One approach is to have the executive handle all the priority considerations, with interrupts occurring regardless of the priority of the tasks they might invoke. A much more desirable approach is to have the hardware participate in priority control. This is accomplished by having interrupts not immediately occur if an interrupt of higher priority is being processed. Instead, the new interrupt would be delayed by hardware mechanisms until all higher priority functions were complete; then it would occur immediately. The advantage of this approach is obvious in terms of both reducing executive overhead and simplifying verification, for interference with high priority tasks is eliminated. Utilizing this feature requires some software discipline, for task termination must be done by a method that the hardware can recognize. This can be more easily accomplished if automatic save and restore mechanisms such as those described in Section 5.8.4 are also provided.

5.8.3 Interrupt Disable and Enable

Another means of minimizing interrupts is to provide the facilities for selectively disabling and enabling interrupts under software control. Some operations may be so critical that they cannot be interrupted even to add a new task to the queue. An example would be a high-speed transmission such as a disk transfer; this might have low priority until it actually begins but must then have very high priority, since processing of new interrupts might cause transmission errors or information loss. On the other hand, really critical interrupts, e.g., those indicating hardware failure within the computer, should never be disabled. During the priod in which interrupts are disabled, information indicating what has occurred during that period must be saved automatically. When the interrupts are enabled again, this information is consulted, and if an interrupt has been suppressed it now occurs immediately. Thus the only effect of the disable/enable

sequence is the delay of lower priority interrupts. Providing this disable/enable capability is very important in reducing verification difficulty in that the areas that must be verified for interrupts can be minimized.

5.8.4 Status Save and Restore

From a software viewpoint, one of the most important aspects of an interrupt mechanism is the method by which processor status is preserved when an interrupt occurs. Some method must be employed for saving the state of the machine's registers at the time of the interrupt and for restoring this state when the interrupted program is resumed. The basic approaches to the problem of status preservation are:

1) Let the interrupting program (the executive interrupt handler) decide what to save and restore.

2) Automatically (by means of hardware) store all machine registers in a designated region in main memory -- possibly with a stack convention to allow interrupts on top of interrupts.

3) Provide two or more sets of internal registers. When the interrupt occurs, a pointer is modified to indicate that a different set of registers is to be used.

4) If the machine is such that save instructions are employed to prepare for potential restarts, no auxiliary saving of registers is necessary. After the interrupt is sensed, execution may be reinitiated from the last save point.

Option 1 is the least desirable but is representative of conventional flight computer architecture. Checking that the necessary registers are saved and restored has consequently been a major verification chore in the past. Option 2 makes no demand on the system programmer, and the overhead time per interrupt is constant. Option 3 likewise makes no demand on the system programmer, unless there is the possibility of using up all the sets of registers. Then software may have to be added to prevent register overflow. With many possible executive designs, the interrupted task will not always be resumed after the interrupt has been processed. If this is the case, even with Option 3 it would be necessary to store the contents of the registers in a region of main memory associated with that task. Thus, a register store instruction would be very useful. For both Options 2 and 3 a complementary register fill instruction would be useful in reinitiating a

partially completed task. Beginning with the location in memory specified by the address, this instruction would proceed to fill the machine's registers from main memory. Options 2 and 3 are equally desirable since they impose no software load. Choice should be made on the basis of speed requirements. Option 2 is slower but probably cheaper. Option 3 is faster but probably more expensive.

Option 4 deserves separate consideration because of its unique impact on software design. If the save/restart feature is provided, the use of the same feature for restart after an interrupt involves little or no addition to the hardware. The program's response to an interrupt would be equal to the fastest of the other alternatives; however, resuming the execution of the interrupted program would effectively take much longer than with the other alternatives because execution is resumed at a point preceding the interruption. This back-tracking has one advantage, nevertheless, for restarts from an interrupt are always made at defined places in the program. Although Options 2 and 3 may overall be superior to Option 4, there will be substantial advantages to Option 4 if the actual hardware implementation of Options 2 and 3 fall short of the ideal cases described above.

5.9 Summary of Conclusions

A conventional computer architecture, unique only in its provisions for restart and interrupt processing and having large memory size, speed, and input/output capability margins would result in the lowest software development costs. These margins are the most important characteristics; an excess capacity on the order of 40% is suggested to facilitate software verfication, maintenance, and growth.

An extendable memory or the ability to add processors provide means for tailoring the computer's capability to the requirements and thus are desirable. An auxiliary memory is undesirable because of its attendant software complications and verification difficulty and is not recommended unless there is a substantial difference between the total memory and peak mission phase requirements. Such a difference does not appear likely at this time. It is also important that the instructions have the capability of addressing all of memory without requiring the use of special extension registers and that there be provisions for selectively restricting a program's access to designated memory regions, either through bound registers or hardware memory locks.

For a conventional computer with an accumulator, quotient, and index registers, the total number of basic instructions identified is 36. In addition, hardware floating-point arithmetic capability should be provided. Powerful or unique instructions that either repetitively perform the same operations as the basic instructions or replace software subroutines do not offer any substantial benefits to compensate for the problems in restart protection and software verification that they might introduce. The format of instructions should be compatible with data formats, and architectures which attempt to increase the effective memory size by packing more than one instruction into the basic data word size should be avoided. A general-purpose multiple-register organization in which the same register can be used as an accumulator, quotient register, or index register depending on the instructions referencing it is desirable because it will reduce the number of tempory data loads and stores.

Where possible, the detection of hardware errors, both hard and transient, should be accomplished by hardware mechanisms. Similarly, error diagnosis and correction should be done by hardware where possible. Where software mechanisms are required, such as for reloading volatile registers after a transient failure, the computer architecture should facilitate their construction. In the event protection of volatile registers against transient failures must be done with software, one important feature is a save-for-restart instruction that, appropriately located throughout the program, establishes safe points from which restarts can be made.

Desirable interrupt features include hardware priority control in which lower priority interrupts are delayed during the processing of higher priority interrupts, and interrupt enable and disable commands that can lock out selected interrupts during execution of critical program segments. Periodic or interval timer interrupts are required to maintain software synchronization with real time. The interrupt periods must be compatible with the task execution frequencies. Finally, the saving of machine status when an interrupt occurs and its restoration after the task invoked by that interrupt has been completed should require a minimum of programmer effort.

None of the desirable architectural features appears to pose any hardware design problems although many, such as hardware floating point, hardware memory locks, and a flexible interrupt control mechanism, are not generally found in aerospace computers today. On the other

hand, many of the undesirable features are; among them are restricted memory addressing, minimal instruction sets, and different instruction and data word lengths. The evaluation of specific proposed computer architectures, particularly for off-the-shelf hardware, will require a balancing of their respective advantages and disadvantages with reference to the criteria described in this section. A methodology for performing such comparisons is described next.

6. EVALUATION METHODOLOGY

Studying the software impact of particular configurations and architectures is, by itself, useful. First, it indicates early in hardware definition which approaches can simplify production without incurring additional hardware costs, in effect stating to the computer hardware designers what is important to the software analysts. Second, for a particular hardware choice it indicates where software design and production problems are likely to occur, so that attention can be devoted to solving them. Finally, studying the software impact of hardware choices is the first step in selecting the most suitable hardware, or in comparing proposed alternatives. The next step is to define an evaluation methodology that enables differing and conflicting effects to be amalgamated into a meaningful estimation of total software impact.

Two relatively simple ways exist for using the evaluation criteria. One is to estimate performance qualitatively with respect to the individual criteria and mentally integrate the subjective evaluations into a qualitative estimate of the particular computer system's overall software performance. Such a qualitative approach can lead to valid decisions. Among its disadvantages are the difficulty of communicating individual value judgments about criteria importance and performance. This approach also inadvertently emphasizes the problems that have most recently confronted the evaluator, and solutions that are appealing largely because of their technical interest and sophistication.

Another way of evaluating a computer system is to assign weights indicating the relative importance of each criterion and to determine a single quantitative measurement for each proposed system's performance with respect to all criteria. The system's expected performance is then determined by summing over all criteria the product of the individual weights and quantitative performance measurements. For example, the instruction set suitability criterion could be assigned a weight of 10. If it were determined that including floating-point arithmetic doubled performance with respect to instruction set suitability, a computer that had this feature would receive a score higher by 20 than one that did not. Such a simple weighting scheme has several deficiencies. Some of them are inescapable in a quantitative model that attempts to deal with criteria that usually are viewed only qualitatively. Others result from the simplicity of the model: it is not always possible to measure a proposed computer system's performance on one criterion using a single number.

Uncertainty about the software effect of system characteristics may make a range of possible values a better indicator of expected performance than a single value. For example, the effect of a restricted operand addressing range in the instruction format depends on module size, programming techniques employed, and data organization. There is some small probability that a restricted operand addressing format might affect programming costs only negligibly, and another small probability that it would increase programming costs by as much as 10%, with the most likely effect falling between the extremes. For cases such as this, a probability distribution can more accurately depict expected performance than can a single value.

Another limitation of a simple weighting methodology is that the benefits obtainable from increased performance are not always directly proportional to the increase, as is implied by using constant weighting functions. For example, a memory surplus is desirable: it reduces the need for optimizing object code, sharing commonly used subroutines, and overlaying unrelated data. Up to a certain point its advantage is proportional to its amount; beyond that point additional memory affords no advantages; and at some point additional memory may increase software costs because inessential functions are added largely because it is possible to perform them. Hence to more accurately depict the benefits of performance with respect to the criteria, nonlinear weighting functions must be accommodated.

Thus the evaluation methodology developed during this study has two major inputs, both of them based on the evaluation criteria. One is expected performance with respect to each criterion. For the reasons discussed above, it may not always be possible to establish precise performance with respect to a criterion; rather, a range of possible performances will be more accurate. Thus, in the general case, the probability of each level of performance is the input function. These performance-probability functions are required for each evaluation criterion and proposed computer system. They are represented mathematically by the expression:

$$P_{ij} = f(y_i \mid A_j)$$

where

A_j = the ith computer alternative being examined

y_i = the jth evaluation criterion

P_{ij} = the estimated probability density of y_i for A_j

The performance-probability function P_{ij} represents a discrete probability function where the measure of performance with respect to the evaluation criterion y_i is discrete, and a continuous probability distribution function where the measure with respect to y_i is continuous.

An example performance-probability function is shown in Figure 21, indicating a range of performance for architectures with floating-point arithmetic. Overall performance with respect to instruction set suitability is measured in terms of the number of instructions needed to code the onboard program, with a suitability of .95 indicating that 5% fewer instructions would be required compared to an architecture having a suitability of 1. Performance of a fixed-point architecture is 1 on the scale chosen; that of the floating-point alternative is between .93 and .97 on the same scale, with all values in between equally likely. Any convenient scale may be chosen for any criterion; in this example the scale is based upon the relative program sizes. Lower numbers on this scale indicate greater suitability.

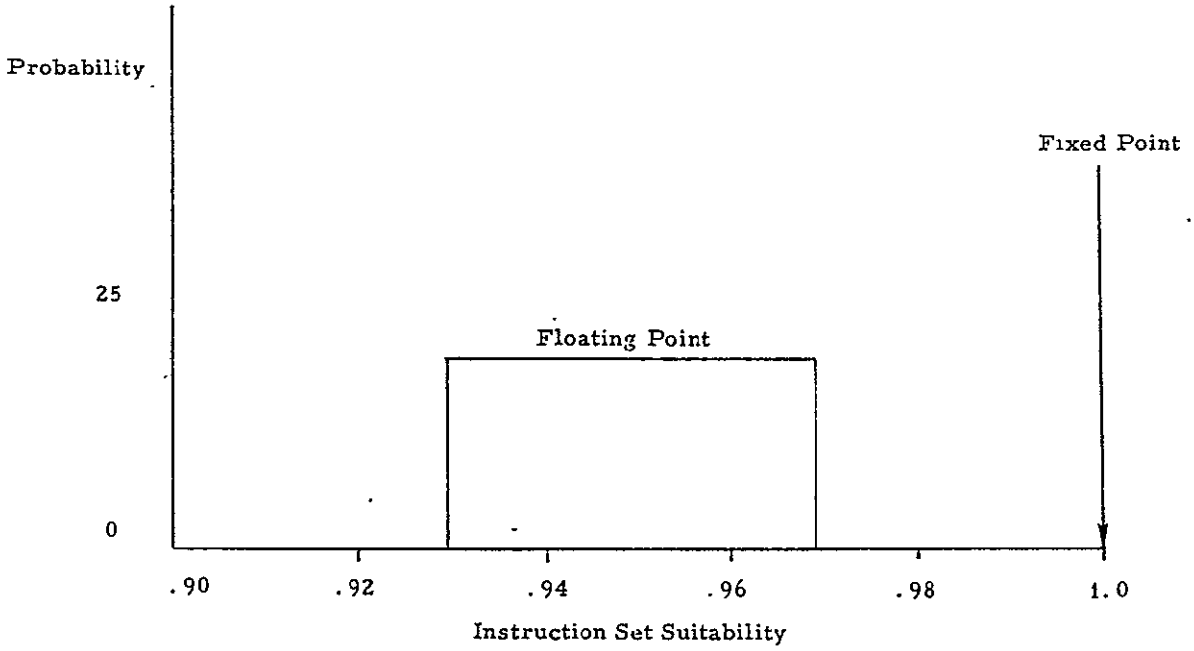


Figure 21. Performance-Probability Functions

Defining performance-probability functions for each alternative permits a performance-probability matrix to be constructed as shown in Table 11. The criteria appearing in the matrix for the evaluation of computer architectures would be those in the eight categories developed in the preceding section; the examples shown in Table 11 are for the evaluation of memory suitability. The elements of this matrix, either single-valued functions or probability distributions, represent the expected performance of each alternative with reference to arbitrary scales.

A further extension of the performance-probability matrix concept can be made, although in the current study the information needed for its application was not available. This extension involves constructing performance-probability matrices for each foreseeable set of Space Shuttle development environments and requirements. For example, separate performance-probability matrices would be constructed to indicate the difference in criteria performance when the same computer architecture is used for both large and small Space Shuttle avionics configurations. While many elements of these matrices might be the same, it is probable that a significant number would be different. For example, the effect of floating-point arithmetic on performance depends on what percentage of the total computational load consists of strictly numeric calculations. The larger the avionics system, the smaller the likely percentage of numeric calculations: the functions that will always have to be performed -- such as guidance, navigation, and control -- are largely numeric, while a significant portion of the additional capabilities of the large system would be devoted to such things as more powerful monitoring, diagnostic, reconfiguration, and display functions involving a higher percentage of logic and nonnumeric data manipulation.

Table 11. Performance-Probability Matrix

Evaluation Criteria	Expected Performance			
	Alternative 1	Alternative 2	• • •	Alternative j
Memory margin	P_{11}	P_{12}		P_{1j}
Memory addressing range	P_{21}	P_{22}		P_{2j}
Memory locks	P_{31}	P_{32}		P_{3j}
•	•	•		•
•	•	•		•
•	•	•		•
Criterion i	P_{i1}	P_{i2}		P_{ij}

The next input required in the evaluation methodology enables the information represented through the arbitrary scales of the performance-probability functions to be transformed into units that permit summation of performance with respect to the total set of evaluation criteria. This is done by using performance-payoff functions to determine overall computer impact on software development cost. For complete accuracy the software impact should also be measured in terms other than just cost; examples of such additional dimensions are the software schedule requirements and software effectiveness, flexibility and verifiability. To simplify the evaluation process, these aspects can be considered by including -- in the total development cost -- the cost of conforming to a fixed schedule and achieving uniform levels of effectiveness, flexibility, and verifiability. Thus, for example, the difference in verification ease for two systems could be determined by comparing the costs incurred for each system to obtain the same confidence level in correct software functioning. Selecting cost as the uniform scale for measuring the value of performance facilitates comparisons between distinct aspects of software impact. However, it should always be recognized that a simplification -- in effect analogous to comparing apples to oranges by comparing the respective costs of the nutrients they contain -- has been made. For many purposes such simplifications are justified, although the decisions made on this quantitative basis must also be compared with subjective, qualitative evaluations.

The evaluation methodology has two uses. One is determining the software impact of a specific computer configuration and architecture and of relatively small variations. The major difficulty with this use is in establishing an absolute scale to measure costs associated with each level of criteria performance. To establish a meaningful scale it is necessary to know not only that providing a feature such as floating point is good as far as reducing software production costs and schedules, but how much reduction can be expected. The other use is in comparing the software impact of two or more distinct configurations and architectures. Establishing an absolute measurement scale for each criterion is not as important for this use, for a relative scale is almost as suitable. What is required in this case is a single measurement scale, whether relative or absolute, that is valid for all criteria. For example, such a uniform scale would be needed to compare the software impact of two proposed computers, one having a suitable interrupt structure but an unsuitable restart provision and the other with these two characteristics reversed.

To a large extent establishing an absolute scale depends on the final size and complexity of the total Space Shuttle software. As already indicated,

for example, the software cost impact of a floating-point capability depends on the amount of arithmetic operations to be coded. Final determination of the acceptable hardware cost of providing floating point thus depends on the functions to be performed. As outlined in Section 3, the size and complexity of the total avionics system software cannot be determined until the avionics system hardware configurations are more completely and permanently defined. Thus an absolute scale for measuring performance with respect to the evaluation criteria cannot be established at this point.

A relative scale can be defined and would be suitable for comparing the performance expected for different configurations and architectures and as a base for establishing an absolute scale when further information about avionics hardware configurations and functional requirements is available. With a relative scale, the software cost impact of an architectural feature would be measured in terms of the expected percentage reduction in total software cost, rather than the expected dollar amount that would result from an absolute scale.

Performance-payoff functions are plotted with the ordinate having the same units as the performance-probability function and the abscissa being either absolute or relative cost. An example is shown in Figure 22 to illustrate the relationship between instruction set suitability and relative program development cost. Expected performance of architectures utilizing floating-point and fixed-point arithmetic has been shown in Figure 21; Figure 22 illustrates the effect any of these possible levels of performance will have on program development cost. The relationship is linear in the region around 1, but does not drop to 0 as the number of required instructions, and therefore the instruction set's suitability, approaches 0. As the number of instructions increases greatly on the other side of the curve, the required software development effort increases even more rapidly.

In this simplified evaluation model the performance-payoff functions are considered to be independent of the particular computer configuration and architecture. Thus the cost benefit of increasing instruction set suitability is independent of the particular architectural details leading to that increased suitability. A performance-cost vector is defined in the same format as the performance-probability matrix, with each element C_i of this vector representing the performance-payoff function associated with the ith evaluation criterion. Like the performance-probability matrix, this vector can be extended in another dimension to indicate the effect of various foreseeable Space Shuttle development environments.

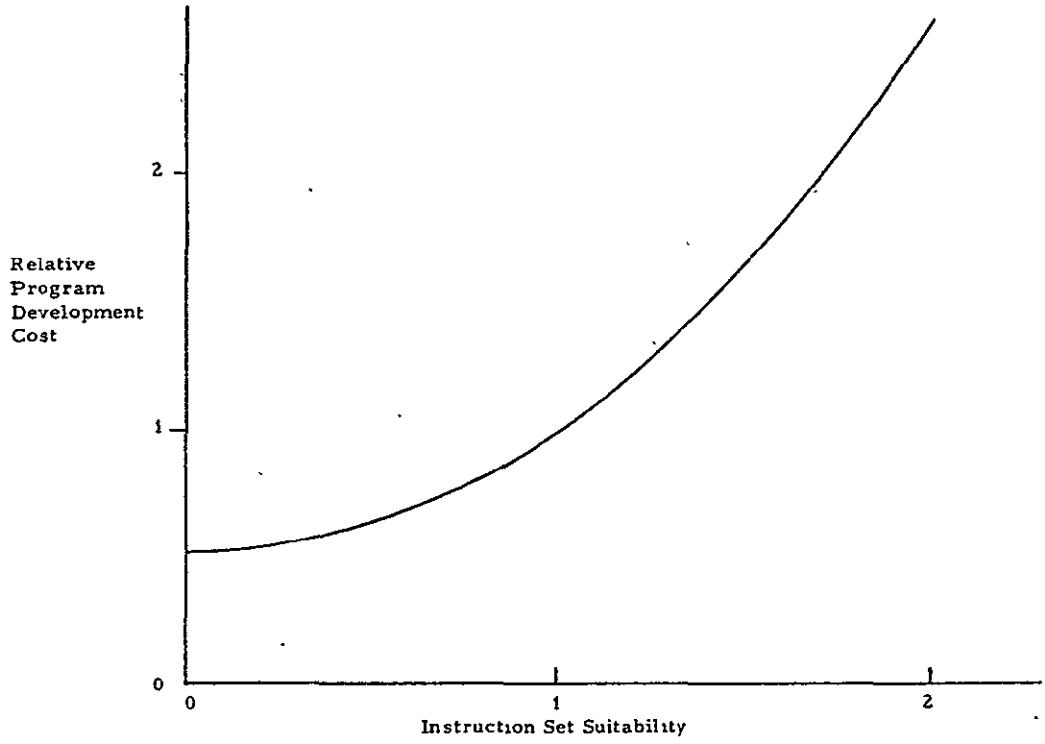


Figure 22. Performance-Payoff Function

When the performance-probability and performance-cost functions have been established, it is a relatively simple mathematical step to compute the expected software cost impact of each alternative computer system. This computation is described by the expression

$$E(C_{ij}) = \int_{-\infty}^{\infty} P_{ij}\ C_i\ dy_{ij}$$

where $E(C_{ij})$ is the expected software cost for the ith computer alternative and jth evaluation criterion. The total software cost impact of the computer configuration and architecture is the sum of the applicable individual criterion cost impacts. The computation above reduces to the simple weighting evaluation approach when C_i is a linear function and P_{ij} is a single-valued function.

7. PROGRAMMING LANGUAGES

Two general language categories can be distinguished: assembly or machine-oriented languages that require the statement of problem solution in detailed and very computer-dependent terms, and high-order languages that allow more general, problem-oriented, and computer-independent terms. Languages in the first category have many disadvantages. Extra programmer effort is required because of the increased number of commands that must be written. Greater possibility of coding errors exists because of the unneeded complexity and unwanted side effects of many of the commands. Also, formidable communication barriers are raised by the difference in appearance between the language commands and a statement of what they are to accomplish. High-order aerospace-oriented languages with suitable compilers are just now being developed. Their usage for Space Shuttle software development was investigated with a view to minimizing coding, verification, and maintenance costs. The investigation focused on three principal aspects:

- Language suitability
- Compiler suitability
- Language influence on the software development cycle

These aspects are individually discussed below, followed by the conclusions reached. Language suitability in particular -- and, where applicable, the other two aspects -- was studied with reference to the following six existing high-order languages:

	Special Purpose	General Purpose
Minimal	CLASP SPL Mk II	FORTRAN
Comprehensive	HAL SPL Mk IV	PL/I

Of the four special-purpose languages, CLASP (Computer Language for Aeronautics and Space Programming) and SPL Mk II (Space Programming Language Mk II) are almost identical, are relatively limited, and are intended for small, fixed-point aerospace computers. SPL Mk IV and HAL are omnibus languages intended to provide very comprehensive capabilities

for large aerospace computers and for general-purpose computers as well. The two general-purpose languages, FORTRAN and PL/I, are the respective minimal and comprehensive counterparts of the aerospace languages.

7.1 Language Suitability

Ideally, a high-order programming language is machine and application independent; that is, any such language should be suitable for any potential computer system and for any tasks it might be called upon to implement within that system. In practice, this ideal is never achieved. This does not mean that a language unsuitable in only some respects cannot be used; given sufficient time and resources, almost any language can be used to implement any programmable function. However, software development difficulty will be increased, schedules lengthened, and attainable level of confidence in program correctness reduced in proportion to lack of language suitability.

Language capabilities required for straightforward and efficient implementation of anticipated software functions are shown in Table 12, together with the estimated percentage of the program devoted to each function for the minimum and maximum Space Shuttle hardware configurations. The percentages reflect the fact that arithmetic capabilities become less important and logical and Boolean capabilities more so as hardware complexity increases. The effect of this shift toward logical and Boolean capabilities is magnified in that the software itself also becomes larger to support the increased hardware configuration.

Of the six languages considered, FORTRAN provides the fewest capabilities for implementing all of the required functions. Even for a minimum hardware configuration, it lacks adequate data description facilities, fixed-point arithmetic modes, flexible arithmetic and logical decision statements, program structuring facilities, and adequate hardware interaction statements. CLASP and SPL Mk II would be adequate for the majority of the first two classes of software functions but inadequate for implementing the Space Shuttle display, communication, and executive software. HAL, SPL Mk IV, and PL/I provide almost all of the necessary capabilities; HAL and SPL Mk IV -- the two special purpose languages -- are the most suitable because they lack unneeded features and emphasize either simplifying or automating capabilities commonly used in aerospace applications.

The relationship between computer architecture and language features is indicated in Table 13. It might be expected that the comprehensive special-purpose languages would incorporate those features necessary to make them

Table 12. Language Capabilities Required for Different Software Functions

Software Function	Percentage of Program per Function		Language Capabilities
	Minimum Hardware	Maximum Hardware	
Guidance, navigation, and control	60	40	Arithmetic data description, expression evaluation, and decision facility Array, matrix, and vector operations Looping control Subroutine and subprogram definition and calling
Status monitoring and checkout	15	40	All of above plus: Logical and Boolean data description, expression evaluation, and decision facility
Display and communication	15	10	All of above plus: Message and format description Character and bit manipulation
Executive	10	10	All of above plus: Hardware interaction I/O processing Program structuring File management

Table 13. Computer Influence on Language Suitability

Computer Architecture	Affected Language Features
Memory structure	Storage allocation and overlay facilities Optimization directives
Word format	Data declaration Multiple-precision arithmetic
Register organization	Hardware directives Register allocation
Instruction set	Fixed-point arithmetic Interfacing with machine language
Restart/interrupt facilities	Restart protection Real-time control Program structuring
Input/output facilities	I/O statements Data formatting and conversion Data buffering

suitable to almost any computer configuration and architecture. However, HAL and SPL Mk IV both emphasize computer independence and presently contain few of the features that allow for often-needed direct control of specific computer operations; these remain to be provided when the languages are implemented for an actual computer and application. CLASP and SPL Mk II are capable of providing the necessary computer-dependent language features only for small and relatively simple aerospace computers comparable to those for which compilers have already been developed (IBM 4PI CP-2 and UNIVAC 1824). FORTRAN's hardware-oriented language features are limited and directed to computer characteristics and applications quite unlike those expected for Space Shuttle. Of all the languages considered, PL/I is the most suitable insofar as its computer-oriented features are concerned; this is evidenced by its effective use in complex executive system programming and real-time applications, of which the MULTICS and SABRE projects are examples.

7.2 Compiler Suitability

Even the most suitable language requires an equally suitable compiler if it is to be a practical tool for software production. The criteria against which a compiler may be evaluated to determine its suitability are relatively objective and may be stated as follows:

- Object code efficiency
- Object code correctness
- Diagnostic capability
- Compilation time
- Cost and availability

The first criterion is a measure of the additional computational capacity that will be required because of the way in which the high-order language is translated into machine code, while the second refers to the validity of the translation process itself. Certain levels of efficiency and correctness are mandatory if use of a high-order language is to be practical. The third criterion, diagnostic capability, relates to means of improving and simplifying verification. The last two refer to the costs incurred in obtaining and using the compiler -- costs that must be offset by reduced cost of onboard software production if high-order language usage is to be economical.

7.2.1 Object Code Efficiency

Even a very large and powerful Space Shuttle computing system will have memory capacity and execution speed limitations imposed by cost, size, weight, and power consumption constraints. In all aerospace software developments to the present, machine or assembly language coding has been used, through which the programmers, by applying their skills, knowledge, and considerable effort, can produce highly optimized code. The programs produced by such methods can be considered to be standards against which programs written in a high-order language can be compared. Comparing such hand-tailored code with compiler-produced code almost always results in the latter's requiring more memory and longer execution time. This difference between hand-tailored and compiler-produced code is a measure of a compiler's efficiency.

Compilers differ widely in their efficiencies. In one study performed by Logicon, three different compilers for a single language -- FORTRAN IV -- were shown to differ by 100% in the time required to execute the same program and by 50% in the memory required, even though the capabilities of the computers on which the compilers were implemented were comparable. Furthermore, differences as great as 300% were observed in comparing the

programs produced by compilers of two different languages -- FORTRAN IV and PL/I -- even though the source programs had the same functional capability, design, and structure. Obviously, the hardware penalties associated with such severe inefficiencies are unacceptable; a compiler for the Space Shuttle must do a better job of generating optimum machine code.

A compiler can be designed to perform many different types of optimization; a partial listing will indicate their nature and the compiler complexity that can result, and will also serve to draw attention to the approaches that a good programmer uses automatically. This listing, which follows, is divided into local and global optimization categories. Local optimization aims at improving the translation of the programmer's statements into object code in small, detailed, and often machine-dependent ways. Global optimization attempts to improve the code by making changes to the structure of the program itself. There are many more proven methods for local than for global optimization, and their implementation is easier and their effect more apparent.

Local Optimization Methods

1) Eliminate common subexpressions

2) Evaluate expressions at compile time

3) Minimize use of intermediate storage by reordering computations

4) Maintain frequently used operands in registers

5) Eliminate redundant instructions

6) Reduce operator strength (e.g., substitute addition for multiplication)

7) Combine several instructions into a single instruction

8) Simplify storage alignment in structures

9) Substitute open subroutines for references to closed subroutines

Global Optimization Methods

10) Eliminate calculations whose results are not used

11) Transform nested loops into single loops

12) Move code to less frequently used blocks

13) Move invariant computations outside loops

14) Replace a program segment with a shorter or faster equivalent

A heavy emphasis on local optimization is mandatory for a Space Shuttle compiler. Global optimization does not usually accomplish anything that the programmer could not have done himself by establishing a better program organization or using more suitable language features. Also, with global optimization the resultant object program -- even though it produces the desired results -- can look substantially different from the original high-order language source program and the programmer's concept of it. This complicates verification and maintenance in that program changes can have a far greater effect than could have been anticipated before compilation of the changed program.

Experience with compilers that perform a great deal of optimization is relatively limited. One exception is the Logicon-developed CLASP compiler recently delivered to NASA/MSC. This compiler employs optimization techniques 1 through 8 and 10 from the list above. Benchmark programming performed to compare the compiler-generated code with equivalent assembly language coding revealed two things. First, it is possible for a compiler to generate code very closely approximating hand-written assembly code. Second, the ability of the compiler to do this depends strongly on the programmer's knowledge of the optimization techniques employed and the best way of utilizing them. Whereas a programmer experienced with the compiler and its methods of optimization produced a program essentially identical in size and execution time to its assembly language counterpart, a programmer lacking familiarity with the particular compiler produced a program some 75% slower and 80% larger.

Another factor affecting object code efficiency is the size of the programming language. Developing an optimizing compiler for a large language is a good deal more difficult. This is demonstrated by experience with the FORTRAN IV Level H and PL/I Level F compilers for the IBM 360/65. The FORTRAN compiler produced object code of relatively high efficiency quite early in its development, while the PL/I compiler went through many versions over a 5-year period and even today does not do as complete optimization as its FORTRAN counterpart. It is estimated that with a relatively small language and considerable effort spent to include extensive optimization

features, a Space Shuttle compiler can be developed to generate object code requiring only 10-15% more execution time and memory than the best assembly language program. With a large language, including equally comprehensive optimization capabilities would be difficult, if not impossible, and inefficiencies upwards of 25% can be expected.

7.2.2 Object Code Correctness

The translation from source language into target computer object code must be correct. Achieving compiler correctness has been a major problem with many commercial compilers; in some cases the period between initial distribution and error-free operation has been greater than 2 years. Unless a very small and simple language is specified, it can be expected that the same thing will happen with a Space Shuttle compiler. The chief consequence of using a compiler in which a high level of confidence has not yet been established is that increased efforts are required in program development to detect and correct compiler errors promptly. No degradation in program quality need be feared because of compiler errors, however, because the threat of compiler-induced errors requires that verification activities be conducted at the assembly or machine language level.

7.2.3 Diagnostic Capability

A high-order language can contribute significantly to software development by making it possible to detect software errors before verification. Most compilers detect syntax errors that prevent unambiguous or meaningful interpretation of language statements. With somewhat greater difficulty, other diagnostic capabilities can be implemented to detect programming convention violations, such as the unauthorized referencing of data items; program logic errors and inconsistencies, such as loops which do not terminate; and data incompatibility as in mixed-mode floating- and fixed-point arithmetic operations. The more diagnostic capabilities provided in the compiler, the greater are the chances that errors will be detected early during software development. A related requirement is that the compiler should generate extensive listings correlating the source and object code and describing the contents of symbol tables and dictionaries, cross-reference tables, and memory allocation maps.

7.2.4 Compilation Time

In the commercial programming environment a significant portion of the computing load is consumed by compilation, and as a consequence considerable effort is spent on reducing compilation time. This is one reason

for relatively low efficiency of the object code: it would be a poor trade-off to save, say, a minute of execution time at the cost of increasing compilation time by 3 minutes. For a Space Shuttle compiler the opposite will be true; here, compilation time will be secondary to object code efficiency. What may be expected when optimization is accorded high priority is shown by the IBM 360/65 FORTRAN IV Level H compiler. The user of this compiler can select the level of optimization. When most of the local and global optimization techniques listed in the previous section are enabled, efficiency improves by a factor of up to two and compilation time increases by a factor of three.

Typical commercial compilers operating on the typical large-scale general-purpose computer compile about 500 source statements per minute. For Space Shuttle, the required optimization will reduce the number of statements compiled to somewhere between 50 and 100 statements per minute. A rough conversion factor of 6 to 8 object instructions generated per source statement indicates that for the total Space Shuttle onboard program, assuming the moderate hardware configuration, the compilation time will be on the order of 15 to 30 minutes. This would be excessive. However, by doing most of the compilation at a module or task level, total compilation time required for software development using even a relatively time-consuming compiler will be practical. Thus even compilation times of the magnitude suggested above will not have a significant impact on the practicality of a high-order language for Space Shuttle.

7.2.5 Cost and Availability

Compilers are large, complex programs: those for simple languages consist of upwards of 50,000 instructions, while those for large languages are even larger and more costly than the proportional increase in language size might indicate. It is apparent that the Space Shuttle compiler may be as large as or larger than the onboard program to be created. In commercial applications the cost of developing a large and complex compiler is justified by the great number of programs that will be compiled during its useful lifetime. The cost of a Space Shuttle compiler cannot be spread over more than a few complete onboard program developments; further, extensive optimization capabilities -- not normally included in commercial compilers because of their development cost -- will be required. The cost savings attributable to the use of a high-order language thus must be substantially greater than the relatively high, concentrated expense of compiler development. The magnitude of the software required for the moderate and maximum hardware configurations indicates that this will be the case for Space Shuttle. In addition, using the language can have a beneficial

effect throughout the software development cycle, thereby further increasing the overall cost savings.

7.3 Language Influence on the Software Development Cycle

The language capabilities and features discussed in Section 7.1 principally relate to the extent to which onboard programs for Space Shuttle can be developed without resorting to extensive machine- or assembly-language-oriented coding. The benefits of using a suitable high-order language for coding this software, while significant, can be even greater if the language can also be used in the other phases of the software development cycle. Language capabilities necessary to each development phase, including programming, are summarized in Table 14. The capabilities called out in the table are in many cases contradictory. Thus, although comprehensive general-purpose capabilities are desirable for program specification development, this requirement is at odds with a straightforward and simple compiler implementation to enhance suitability for the programming phase. Similarly, it would be desirable for the programming phase if there were few restrictions, but for checkout and verification it is important that programming standards be enforced. Because of these and other unavoidable contradictions, any language will be a compromise insofar as its applicability to all development phases is concerned.

To describe how a high-order language can be gainfully used throughout the software development cycle, some specific approaches are outlined below. Many of the features described are not now provided in the six candidate languages; the discussion indicates the type of desirable language extensions.

Assuming that a high-order language is to be used for developing the operational software, one advantage in using it also for the programming done in the program specification development phase would be improvement of the specification itself. As a result, the specification could contain the actual programs that satisfy its requirements to the extent necessary at that phase of the development. This would be particularly useful if the programs were organized on a modular basis according to the functions to be performed.

As an example, consider how a guidance and navigation program would be developed if the specification were enriched in this way. The program used to develop the requirements for performing guidance and navigation would very likely be unnecessarily sophisticated with regard to some functions (e.g., trigonometric subroutines) that are readily available from utility program libraries. At the same time, it would be too elementary in other

Table 14. Language Characteristics for Different Software Development Phases

Development Phase	Necessary Language Characteristics
Program specification development	Comprehensive general-purpose capabilities Easy interface with other software tools Amenable to program modularity Intelligible to nonprogrammers
Programming	Few restrictions Easy to learn and use Concise and natural Straightforward and simple compiler implementation
Checkout and verification	Extensive syntax and semantic error detection facilities Maintains program modularity Enforces programming standards Minimum error-provoking or ambiguous features
Maintenance	Self-documenting Isolates changes and minimizes their effect Extensible Hardware and application independence

functions (e.g., steering command calculation and position determination) that must be developed from scratch and for which greater detail is not required for mission planning purposes. The program specification thus would contain not only a selected mission profile but also a program which generates such a profile. Continuing with this program that demonstrates that mission requirements can be met without precisely describing how to meet them, the too-sophisticated and too-elementary modules would be replaced by modules representing the exact algorithms to be used. As substitutions are made, a more and more complete picture of the actual

trajectory to be flown for various vehicle and environmental conditions will develop. These algorithms, coded in the high-order language, would form the core of the final guidance and navigation program whose development is completed during the programming phase. The activities of this phase will be directed in large part towards the program's computer-dependent aspects; the actual algorithmic solution will have already been presented in a manner which makes its incorporation in the final onboard program a relatively simple task.

The use of a common language in all phases of onboard program development requires that the language and ancillary software possess certain capabilities. Information is added at each stage of the development, and both must allow its acceptance simply and completely yet should not complicate earlier phases because of its absence. As an example, the required range and precision of each variable will not be known at the beginning of program specification development. The initial analysis thus must be conducted with ranges and precisions far greater than will be required. Through simulations conducted with these excess values, those values needed to meet mission requirements will be established. During the programming phase the appropriate changes will be made to the program developed in the earlier phase to maintain these actual values, given the word length and arithmetic characteristics of the actual flight computer.

Programs written during the initial information-gathering phases will be designed to execute on large-scale general-purpose computers because of their suitability to the computational tasks, while the final onboard program must execute on the intended special-purpose aerospace computer, or a simulation of it. This would imply that two compilers would be needed, and their development is one approach that can be taken. However, the structure of most compilers suggests a different, more cost-effective, and far more versatile approach. Most compilers have two main stages: a syntax analysis stage in which the source language is translated into a computer-independent intermediate language, and a code generation stage in which the intermediate language is translated into the computer-dependent language of the intended machine. In this approach, compilation of programs written in support of specification development would be terminated at the end of the syntax analysis stage. Execution of the intermediate language programs thus obtained would be performed through the use of an interpreter. This would be like the onboard computer simulation used later, but more powerful and flexible in control over the details of arithmetic operations, and potentially a good deal faster.

The power and flexibility inherent in this approach results from the capability of the compiler/interpreter user to specify the numeric representation of variables and parameters manipulated by the program and to change this representation with only minimum changes to the program, if any. The utility of this concept, which has been called dataless programming, has been examined through an experimental programming language. Dataless programming allows the user to specify the procedures for manipulating variables and parameters independently of their representation. If during program specification development he wished to study how a control algorithm functions for various word sizes, he would simply change the number of digits declared for each variable and parameter utilized in the algorithm but would not modify the procedures implementing the algorithm. Similarly, all programming done during early development phases would utilize floating point for variables and parameters; only during the actual programming phase would a restriction to the fixed-point mode be introduced if made necessary by the onboard computer's architecture.

The use of the dataless programming approach, while freeing the analysts from concern over the details of range and precision requirements, memory allocation, and similar computer-dependent aspects, does have some disadvantages that must be examined. First, there will be an increase in computation time to execute programs in an interpretive mode. However, this is compensated for by a reduction in the compilation time, since only the syntax analysis stage is executed, and by the fact that fewer computer runs are required, since the information gathered from individual runs increases.

A second possible disadvantage is that errors might exist in a program and might remain undiscovered as long as it is being executed interpretively, to be found only when machine language code is generated. It turns out that the common-language compiler/interpreter approach actually reduces the possibility of such occurrences. In the first place, the errors in translation from one language to another are eliminated. Second, the syntax analysis stage would be identical for both the interpretive and the machine code versions, as would be the error-checking facilities built into this first stage; therefore, differences between the results of program compilation would be fewer for this system than if two separate compilers were developed. Any remaining error situations are largely due to errors or deficiencies in the code generation stage of the compiler and, as such, can therefore be minimized through attention during compiler debugging and experience with the system.

A third disadvantage is the possibility that utilizing the results of one phase in another phase with relatively little repetition of the work will reduce the amount of independent verification. It has been argued that the duplication

of effort in reprogramming for each phase is beneficial in that it reduces the likelihood that errors will be transmitted undetected from one phase to another. This reliance on redundancy is a costly way to achieve confidence; a better solution is to provide within the system powerful program debugging aids that become a part of the onboard computer program itself and are carried with it through the development process. Providing and using such debugging facilities can eliminate much duplication in verification activities, just as duplication in the programming activities is eliminated by the use of a common language. System analysts can, by including debugging statements in the program specification, precisely specify the tests they want performed and the reports they want generated in later phases of program development. These tests would be performed and the reports generated automatically, with no further action required by the programmer.

7.4 Summary of Conclusions

No existing language meets all of the conflicting requirements that must be fulfilled if Space Shuttle software costs are to be held to the minimum. What is wanted is a language that is simple and easy to use but completely capable, is computer independent but offers full control over all computer capabilities, and is very cheap to implement in a compiler but generates efficient object code. It is unlikely that such a language could be defined; any attempts to develop a perfect Space Shuttle language are likely to result only in a language having new and different compromises.

Of the six languages examined, FORTRAN is the most clearly unsuitable. The only advantages it offers are simplicity, the existence of extensive experience in writing compilers for it, and some suitability for use in other phases of Space Shuttle software development. It provides very few of the needed capabilities: less than 50% of the required Space Shuttle onboard software could be written in it.

CLASP and SPL Mk II are suitable for a greater percentage of the onboard software development; about 70% of the software for a minimum hardware configuration and 50% for a maximum configuration could be written in either language. Compilers for aerospace computers have been written for these languages, and the object code they generate indicates that a reasonable degree of efficiency can be expected. Each language is small enough that a Space Shuttle compiler can be developed without significant problems or excessive cost or schedule impacts. However, these languages would not be suitable for use in all phases of the software development cycle unless modified.

Substantially all of the onboard software could be developed using the comprehensive languages HAL, SPL Mk IV, and PL/I, with the first two being slightly more suitable. These languages would also be applicable for use throughout the software development cycle, particularly if they were enriched with debugging and verification features. Compiler development is likely to be more costly than the Space Shuttle alone could justify. Compiler development time could become critical to onboard software development and would certainly preclude using these languages in earlier software development phases unless off-the-shelf compilers were utilized. Furthermore, the efficiency of the generated object code is likely to be poor until all implications of language complexities are fully understood and the problems they engender are adequately solved.

The lack of a language that neither has deficiencies in its capabilities nor poses compiler implementation difficulties prevents making an unqualified recommendation. In general, any of the special-purpose languages -- CLASP, HAL, or either version of SPL -- offers enough benefits to warrant its use in Space Shuttle onboard software development. It is expected that some portions of the program, particularly the executive, will have to be written in assembly language. This is so for the minimal languages because they do not provide all of the necessary facilities, and for the comprehensive languages because the object code efficiency would be too low.

8. VERIFICATION TOOLS AND TECHNIQUES

The software and hardware used to test and verify the proper operation of onboard software are costly but vital elements in the development process. Past systems have relied heavily on simulation for software testing and verification. One reason for this is the impossibility of performing adequate testing in the operational environment; another is that simulation provides increased visibility into and control over the details of software behavior and its interaction with other subsystems.

Many of the disadvantages of simulations, such as the development cost, the difficulty in achieving a proper blend of simplicity and fidelity, and the time required to use them, will become more pronounced because of the expected greater complexity in Space Shuttle software and hardware compared with past systems. For these reasons improvements in the major simulation types - - engineering, interpretive, and hybrid -- are required, and new software testing and verification tools must be developed to supplement their use.

8.1 Engineering Simulation

The first tool to be employed in Space Shuttle software development is the engineering simulation. It is continually used during software development, and the vehicle and environmental models it contains are used as the basis for similar models in the later interpretive computer simulations.

The engineering simulation runs on a general-purpose computer and typically exercises a limited environment and a subset of the flight equations to perform parametric studies and design tradeoffs. An accurate simulation of this type usually runs faster than real time, with the ratio of real to simulated time dependent on the accuracy with which the system is modeled. Within its constraints, an engineering simulation is economical to build and use.

While the engineering simulation is rapid and inexpensive, it is characteristically imprecise in its representation of the digital or interfacing hardware, and is seldom designed to even approximate software performance in meaningful detail. For deriving or verifying gain settings, approximations, targeting errors, and related factors, coarse modeling is acceptable. As sophistication is added to the models, two paths tend to be followed. Some systems add sophistication to the environment model beyond the point where the imprecision of the computer model makes environmental accuracy meaningful; others increase the accuracy of the computer model until the drawbacks of the ICS are achieved without its

advantages. Consequently, judgment must be used to determine where and when an engineering simulation should be used and what the simulation should include if the best compromise between sophistication and usability is to be achieved. Each engineering simulation must also be constructed so that its behavior and results can be easily correlated with the results of other engineering, interpretive, and hybrid simulations. This requires that:

- Its abilities and limitations are sufficiently understood to determine the range of tests for which it is suitable.
- Its output formats are consistent with those derived from finer models so that comparisons can be made.
- Its use is not extended beyond its known capabilities.

Many existing engineering simulations are suitable or can be modified for Space Shuttle software development. Their use would save simulation development costs and, more important, reduce the time and cost required to become familiar with simulation behavior.

During the development of Apollo, a highly successful engineering simulation was built integrating equation-level flight programs with a generalized trajectory package. This Apollo Reference Mission Program (ARMP) provided a complete analytic tool for parametric studies and for determination/verification of flight program parameters. The need for lunar operations in ARMP entailed significant complexity that is not required for Space Shuttle. Competitive simulations exist in modular form which may be evaluated to select a baseline engineering simulation for the Space Shuttle. For this project, the primary new requirements are extensions to simulate the atmospheric flight phases and to achieve input/output compatibility with the other simulations to be used. To assure adaptability to changing mission requirements, modularity of the simulation is essential; similarly, the need for confidence in test results suggests the use of a baseline program already developed.

The basic engineering simulation model at the highest level of abstraction is shown in Figure 23. The three blocks at the top of the figure represent the major components of the guidance, navigation, and control subsystem. The top center block, Data Processing and Guidance, represents all the functions carried out by the onboard computer and its software. The solid blocks enclosed in the two dotted blocks labeled Kinematics and Kinetics represent vehicle dynamics. The bottom block, External Environment, represents everything not included in the others, such as man/computer communication and anomaly generation.

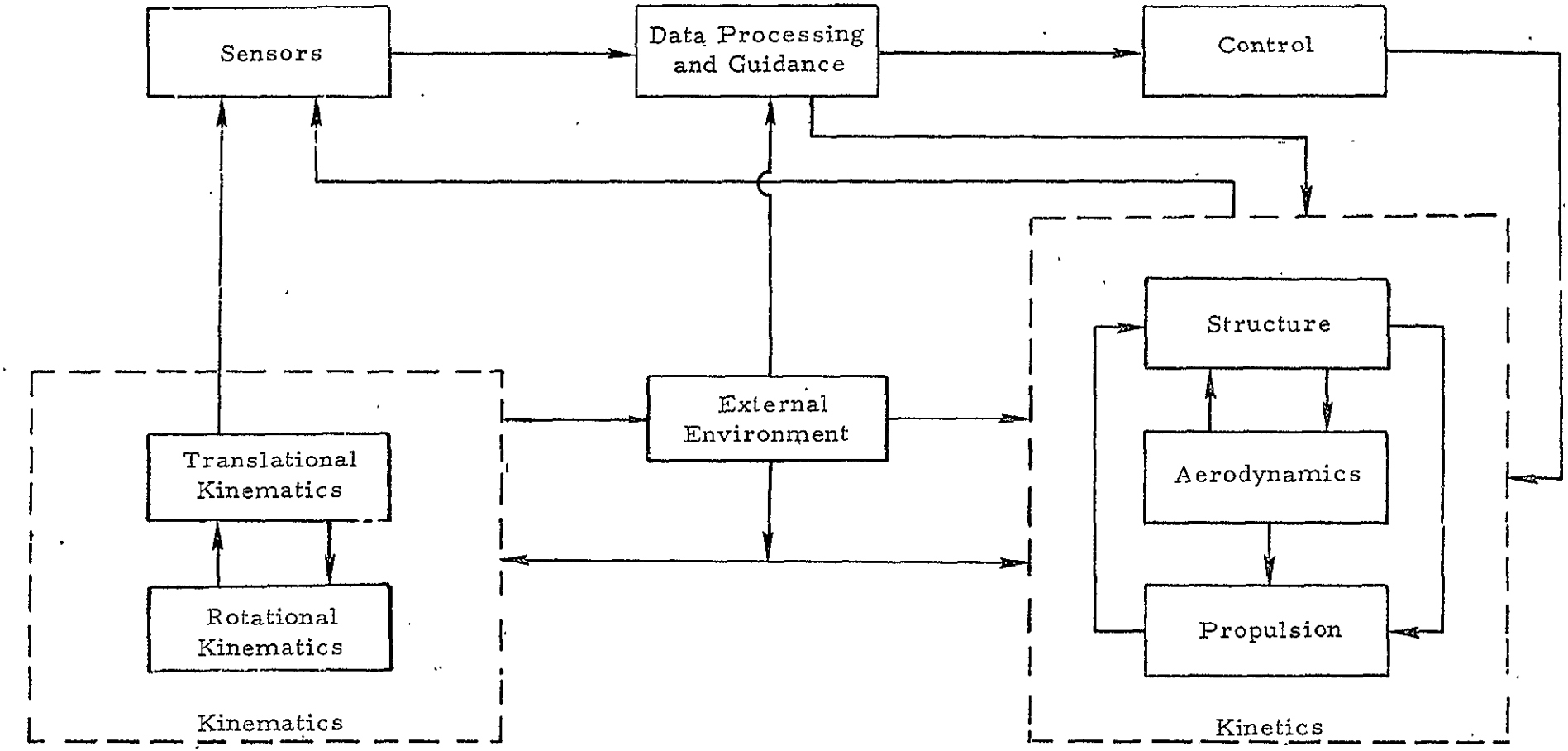


Figure 23. Basic Simulation Model

These functional blocks can be used to clarify the terminology applied to simulations to be used in Space Shuttle software development. A six-degree-of-freedom (6D) simulation would include a model for each of the solid blocks, and detailed submodels for most of them. A three-degree-of-freedom (3D) simulation would not include an authentic representation of rotational kinematics and those portions of the sensors and control models that relate to rotational dynamics. In an engineering simulation, the data processing and guidance model will not necessarily completely represent the functions performed by the onboard computer, but instead will represent a form of the guidance equations more convenient to the programming language and architecture of the machine on which the engineering simulation itself runs.

In a 3D simulation, the three degrees of freedom consist of the three independent coordinates that define the vehicle's translational motion. (A rigid body is assumed in 3D and 6D simulations.) A 3D point-mass simulation is a true three-dimensional simulation in that the vehicle is treated as a point mass and no rotational motions enter into the computations in any form. The fast execution speeds of these simulations make them ideal for planning studies, gross sizing and tradeoff analyses, and those portions of detailed analyses that do not depend on rotational effects. In a 3D zero-moments simulation, center of pressure, center of gravity, and a central thrust application point are located within a vehicle coordinate system as functions of mass distribution, overall engine locations, and aerodynamic characteristics. Turning moments produced by aerodynamic forces are computed, and a single thrust vector is applied such that the total turning moments acting on the vehicle equal zero. The vehicle's attitude is specified independently in terms of attitude rates or attitude angles.

The 6D simulation, like the 3D zero-moments simulation, treats the vehicle as a three-dimensional rigid body but includes three additional degrees of freedom in the form of rotational motion in three dimensions. The vehicle control system is simulated to at least the lowest response frequencies, and the integration step size is correspondingly diminished. The increase in running time of a 6D simulation as compared to a 3D simulation is due in part to an increase in computational complexity, but is even more a result of a higher frequency for control system calculations. The differences in computational frequencies between 3D zero-moments and 6D simulations can be of the order of 50:1 (1/2 sec to 10 msec). Hence using a 6D simulation in its pure digital form is a time-consuming process. Whereas 3D simulations run faster than real time on computers of the

UNIVAC 1108 class, 6D simulations usually run considerably slower. The more heavily used 6D simulation programs have been generalized to allow greater flexibility and growth potential, but these very features have tended to place additional burdens on the user, whose primary interest is in being able to perform his runs with a minimum of extraneous manipulations and delay. Also, even with the added complexity and smaller step sizes of the 6D simulations, the digital models may not simulate high-frequency-response characteristics with the fidelity required by control system users. On the other hand, the all-digital 6D simulation provides repeatability, flexibility of execution, accuracy, extensive data output capability, and the capability to monitor and trace variables.

8.2 Interpretive Computer Simulation

Engineering simulations contain imperfect models of the computer system that, although adequate for such things as program specification development and software sizing, are inadequate for onboard software checkout and verification. The need for a very accurate computer system model for checkout and verification suggests that the actual onboard computer be employed, and this is done in the hybrid simulations reviewed in the next section. However, in many cases the actual computer hardware is not suitable for software debugging and verification owing to its lack of diagnostic information and options in execution. Also, the actual computer may not even be available early in the software development cycle. Hence it is generally necessary to employ a simulation that faithfully represents the condition of every register in the onboard computer before and after the execution of each instruction and that also maintains a simulated-real-time clock. Such a register-by-register simulation can be implemented through either the translator or the interpreter method.

With the translator method, the computer program, which is originally in the onboard (target) computer language, is preprocessed to produce an equivalent program in the simulator (host) computer language. The subsequent execution of the new program on the host computer is equivalent to execution of the original program on the target computer. When the translation is faithful, the host computer code will include all operations necessary to represent the target computer's word length, instruction functions, and timing. When absolute fidelity is not required, simplifications such as neglecting word-length differences are made. The number of host computer instructions per target computer instruction depends on the architectural differences between host and target computers and the faithfulness of the translation. In any case, the translator method tends to create host computer storage problems owing to the expansion of the

program when translated. On the other hand, the speed of execution of the translated program is not reduced by repeated examination of each target computer instruction, as is the case with the interpreter method.

The interpreter method also involves a one-to-many transformation of the computer program code. However, instead of producing an entirely new program for subsequent execution on the host computer, the ICS takes the target computer code as input and faithfully executes it by means of calls on subroutines that produce exact copies of the target computer registers and set a simulated-real-time clock after each target instruction. Since the ICS interprets the target computer code at execution time rather than executing a translated code, it is inefficient in that it retranslates each target computer instruction each time it is encountered. On the other hand, this procedure facilitates the handling of modifiable instructions, interrupts, and diagnostic functions during execution.

Figure 24 shows a typical ICS control loop. The first step is to obtain the target computer instruction whose execution is to be simulated. This instruction is then decomposed into its component parts, such as the operation code, operand address, etc. Based on the operation code, the control loop then branches to the appropriate subroutine for simulating the instruction's execution. The instruction subroutine can be very simple, particularly if the instructions for the host and target computers are very similar. The simulated-real-time clock is usually updated by the instruction subroutine because the target computer instructions vary as to their execution time; this may even be a function of the operands being manipulated. The instruction subroutine returns to the control loop and the simulated-real-time clock is examined to see if it is time for an interrupt, an I/O update operation, or further vehicle and environment simulation computations.

All of the steps to this point in the control loop are directed towards duplicating the target computer's behavior, with little information being generated for the programmer or analyst beyond that which he could obtain by observing the actual target computer executing the same program. A typical ICS requires the execution of approximately 25 host instructions to perform the operations just described in the simulation of a single target computer instruction. The coding of the ICS control loop and the instruction subroutines is a relatively simple and well-defined task for a typical onboard computer; the major problems are in the input and output interfaces. If these tasks were all that were required in the development of an ICS, the total job would take only a few manmonths.

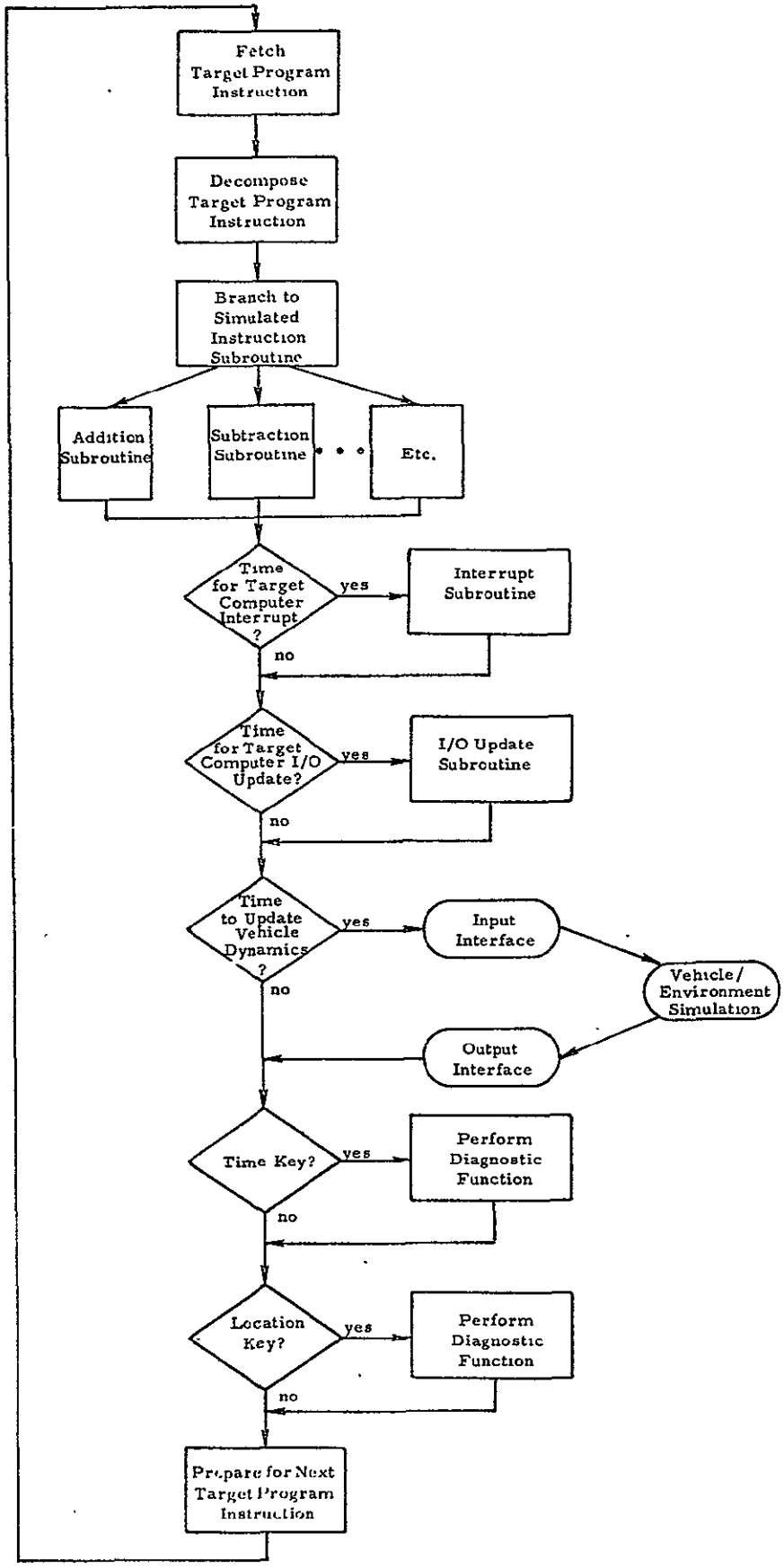


Figure 24. Typical ICS Control Loop

The remainder of the operations outlined in Figure 24 enable the ICS to be a powerful debugging and verification tool, and require most of the ICS development effort. These operations consist of testing both time and location keys to determine whether any diagnostic functions should be performed. A location key is associated with a particular instruction in the onboard computer program; only the simulated execution of that particular instruction results in the indicated diagnostic function being performed. A time key, on the other hand, is not associated with a particular instruction; the indicated diagnostic functions are performed whenever the simulated time reaches a certain value, regardless of the instruction being executed. Time keys may also be periodic in that they both invoke a diagnostic function at the current time and set a new time key which invokes a diagnostic function at some later time. Examples of diagnostic functions include the printout, in convenient formats, of register and memory contents; the checking and comparison, within specified limits, of register and memory contents; and the injection of controlled errors into input or intermediate parameters.

While a majority of the ICS development effort is devoted to the diagnostic portions, most instructions will not invoke any diagnostic functions. The execution time penalty paid for these extensive diagnostic facilities is due almost entirely to the testing of the location and time keys. This testing can be accomplished with about five host computer instructions.

ICSs are employed either open loop or in closed loop with the effects of vehicle and environmental behavior supplied by the appropriate elements of the engineering simulation. In the open-loop mode, ICSs are used in coding and checkout to perform simple tests on small routines, instruction sequencing, and merging of program modules. In the closed-loop mode, they are used extensively for detailed onboard program testing and accuracy assessment. In closed loop with an engineering simulation, the ICS often offers a better diagnostic capability than the actual target computer and is easier to maintain. Such a system has complete repeatability and can be used to create artificial stimuli to force the program into seldom-used branches when desired.

The speed of an ICS is slower than real time. If the selected Space Shuttle computer fulfills the computational requirement of the moderate load outlined in Section 3 (120,000 words, 0.25 msec add time), the increase in speed compared with the Apollo Guidance Computer will have a great impact on the practicality of extensive ICS utilization.

To facilitate discussion of these matters, the basic simulation speed is defined as follows:

$$\mu = \frac{T_{s(p)}}{T_p}$$

where T_p = time required to perform real-world process p (real time)

$T_{s(p)}$ = time required to simulate p on simulator s

For ICS considerations, μ can be considered to be a function of the following target and host computer relationships:

μ_s - the ratio of host computer speed to target computer speed

μ_n - the average number of host computer instructions (per target computer instruction) to simulate a target computer instruction

μ_d - the average number of host computer instructions (per target computer instruction) to perform diagnostics

μ_e - the slowdown due to the vehicle and environmental simulation (simulation time to real time)

The approximate functional relationship between the above factors is

$$\mu = \mu_s (\mu_n + \mu_d) + \mu_e$$

A more precise formulation for μ might require that a least-common-speed quantum and a least-common-instruction quantum be defined to more faithfully represent the relation between μ_s and μ_n. For example a more accurate μ_n cam be computed as a weighted average of the number of basic host computer cycles required to simulate the various target computer instruction types, as follows:

$$\mu_n = \frac{\sum_{i=1}^{m} w_i \sum_{j=1}^{n} a_{ij} b_j}{\sum_{i=1}^{m} w_i}$$

where m = total number of target computer instruction types

n = total number of host computer instruction types

w_i = weighting factor that reflects the usage frequency of the ith target instruction type

a_{ij} = number of times the jth host computer instruction type is used in simulating the ith target computer instruction type

b_j = number of host computer cycles to perform the jth host computer instruction type

Average values for μ_n appear to fall within a range of about 15 to 30, depending on the severity of differences between host and target computer architectures. Certain assumptions are inherent in these considerations, e. g., host computer word length is greater than or equal to target computer word length. The best μ_n values can be expected when host and target computers come from the same "family," such as the UNIVAC 1830 and 1108 or the IBM 4PI and 360. When the target computer is a true subset of the host computer, μ_n will approach unity.

For the Space Shuttle it is very unlikely that μ_n would ever equal 1 regardless of the compatibility between host and target computers. Even when the instruction sets are compatible, slight differences in timing and input/output characteristics, interfacing with a vehicle and environmental simulation, and providing comprehensive location- and time-keyed diagnostic functions will necessitate executing essentially all the control loop functions outlined in Figure 24. To be effective an ICS must be more than just a simulation of the target computer; it must provide more capabilities for verification. Therefore, a host computer must provide more than just instruction set compatibility if substantial improvements in basic simulation speed are to be obtained. A microprogrammable host computer could achieve this; however, full employment of the microprogramming features would require that the basic control loop as well as the instruction subroutines be coded in the microprogramming language rather than in the basic machine, assembly, or higher order language used to develop the conventional ICS. Such a microprogrammed ICS would be desirable as far as the improvements possible in the μ_n ratio. However, no large general-purpose computer available today appears to offer the flexibility needed for ICS development while retaining all characteristics necessary for the other programs it must execute, such as the system executive, assemblers and compilers, and the vehicle and environment simulation.

Another approach to reducing the total time of simulations is to reduce the μ_s ratio by employing faster host computers. Unfortunately, the speed of general-purpose computers is not likely to increase in proportion to the increase in the hypothetical moderate Space Shuttle computer over the Apollo Guidance Computer, indicating that even if the fastest general-purpose computers are employed there will be an increase in simulation time owing to the μ_s ratio.

Two remaining factors in total simulation time are μ_d and μ_e. To decrease μ_d (the average number of host computer instructions performing diagnostic functions), a corresponding reduction in the power, amount, and completeness of the diagnostic functions performed would be required. To allow this to occur would be self-defeating because the principal advantage of the ICS is its diagnostic capability. As the discussion to follow will demonstrate, more rather than less diagnostic capabilities should be employed; therefore μ_d will increase slightly rather than decrease. Reductions in μ_e (vehicle and environment simulation time) are tied to the accuracy of the models employed, although some savings may be possible by using 6D simulations only for the mission phases and program testing where 3D cannot be used. Hence no significant changes in this factor are expected.

Based on estimations of target computer speed for the moderate Space Shuttle computer system presented in Table 2, and assuming a host computer comparable to the UNIVAC 1108, a basic simulation speed of 100 can be expected. Even with the most optimistic values for μ_s and μ_n and a nearly compatible and very fast host computer, the basic simulation speed would remain as high as 20. The corresponding simulation speed for the Apollo Guidance Computer was about 3.

Because of the anticipated and unavoidable increase in ICS execution time for the Space Shuttle, it is imperative that improvements be made in utilizing the information obtained from each simulation, thereby reducing the total number of simulations that must be run.

While ICSs contain numerous aids for analysis, the full potential of automated aids has not been realized. For example, one of the functions that must be verified is that a flight program issues discretes at precise points along the trajectory. This is typically checked by having the ICS print out the value of time associated with the discrete issuance, and having an analyst manually confirm that the discrete issuance was timely. If it was not, he must determine why, make a fix, and start again. The checking of discrete times could be automated, thereby saving the time required to make the check. (It could

also provide for the display of information concerning the discrete occurrence, and if the issuance were bad could stop the run automatically, thus saving computer time.) In general, any function that is amenable to checklisting for testing purposes is potentially amenable to this type of treatment.

Another type of aid that can be considered involves the use of ICSs in program development. During the early debugging stages the programmer is not particularly interested in many of the powerful ICS debugging features. He would be interested however, in inserting simple driving data and hand-computed comparison data at intermediate points. In the event that the comparison failed, the programmer would then be able to trace information back to the last valid comparison point.

It appears that only where ICSs are heavily used in a production mode for verification does their cost, in terms of computer time, approach the cost of the user-analyst. In most other instances the user cost is more, sometimes (e.g., in code development) considerably more. Therefore, to utilize ICSs effectively, ways must be found to reduce their running time, and even more important, to increase the information obtained from each simulation run and to reduce the number of simulations producing no information. Thus a definitive need exists to shift to the simulation the work now performed by analysts.

In the preceding section the use of a high-order language was recommended for onboard program development. By integrating the development of the compiler for this language with the ICS, debugging and verification can be greatly facilitated. Among the improvements gained is the ability to perform these tasks using the same symbology as that used in program development. Another is the ability to have the tests constructed and performed in one stage automatically repeated in later stages. The integration of the compiler and ICS requires directives that until now have been used to control ICS operation be made part of the high-order language. Some of the candidate Space Shuttle languages, notably CLASP and SPL, already contain directives that allow debugging functions to be iniated through the language itself; these functions are rudimentary but can serve as the base for the powerful capability needed. The debugging aids, although included as part of the source program, generate input for the ICS that cause it to perform the required functions. They increase the information obtained about the program's behavior during simulation, but do not affect its behavior as such.

In the design of debugging aids, provision must be made to include the functions required during all program development phases: specification development as well as programming. The general form of the functions that should be included can best be described by providing an example.

The capability of signaling that a computed quantity exceeds specified limits is one necessary function. Assume that during specification development a requirement is established that vehicle acceleration, as determined by the program, must not exceed some maximum value. This requirement could be included in the source program by a statement of the form SIGNAL ERRORA WHEN VDOT IS GREATER THAN 200. In processing this command, the compiler would transmit information to the ICS that would cause the setting of flags such that whenever VDOT was calculated a comparison would be made between it and the critical value 200. Whenever it exceeded 200, the message labeled ERRORA would be printed by the ICS. Similarly desirable debugging functions would make it possible to describe timing limitations on program execution, or to describe invalid program execution sequences. The analysts responsible for specification development should be able to describe, within the format of the debugging aids, as much of the requirements imposed on the onboard program as possible. These computer-intelligible statements would be made a part of the program included in the specification. Then the appropriate checks to verify conformance with the specification would be made automatically on all subsequent executions of the program in the ICS mode.

8.3 Hybrid Simulation

A hybrid simulation incorporates digital and analog devices and, in many cases, the actual avionics system hardware. The onboard computer is represented in the hybrid simulation by an exact hardware equivalent. The simulation of the vehicle and environment is provided by a combination of digital and analog computers, together with any actual avionics hardware that itself is to be tested or that is needed to provide realism because computer models are inadequate. Interfaces between the onboard computer and the vehicle and environment simulation are similarly provided by digital or analog devices or, when appropriate, by actual avionics system hardware. Other hardware, such as recorders and displays, is incorporated in the hybrid simulation to aid in observing the simulation's behavior and to record intermediate and final results. Generally, such recording and display devices are oriented more towards observing and recording the parameters measuring vehicle and environment behavior than towards providing any insight into the operation of the onboard computer software. Finally, other peripheral hardware is provided for loading

the onboard computer with instructions and constants and determining the contents of computer memory and registers, although in a static rather than a dynamic sense.

Where a significant amount of actual avionics system hardware is employed, the effect of its behavior on the overall system is usually represented more accurately than in any other simulation. However, when actual hardware is incorporated into the hybrid simulation, it is often necessary to modify either other avionics hardware elements or the signals generated by them to adequately represent flight conditions in the laboratory. In that event, particular attention must be given to the fidelity of the altered element, since the effects of a modeling error may be both significant and subtle.

Because it employs the actual flight computer and some avionics system hardware, a hybrid simulation can run in real time. For some purposes, such as subsystem testing and crew training, this real-time behavior is very desirable, if not absolutely mandatory. It demonstrates the correct interaction between hardware and software, eliminating most uncertainties that may still exist when only an idealized model of the hardware can be employed, as is the case with the interpretive computer simulation.

There are significant economic and practical advantages for using the hybrid simulation rather than the ICS; these derive from its more rapid execution time. For the larger Space Shuttle hardware configurations it will be possible to perform many more hybrid than interpretive simulations in a given period of time. Since the more times a program is executed correctly, the greater the confidence is in its overall correctness, hybrid simulations will be used extensively for Space Shuttle software verification.

The major deficiency of a hybrid simulation as compared to an interpretive simulation is that much less information can be obtained about details of program behavior. Whereas with an interpretive computer simulator the contents of an onboard computer register are easily obtained at any time during program execution, and the value found can be printed in any convenient format without interfering with normal onboard program execution, with a hybrid simulation internal computer registers can be examined only when the onboard program's execution is halted. Thus with a hybrid simulation the great majority of software diagnosis must be accomplished by examing the inputs and outputs of the computer and postmortem dumps of its memory and registers.

Exact duplication of hybrid simulation results is very difficult to achieve. This further complicates its use in software verification, for it may not be possible to reproduce the symptoms of potential problems observed. To achieve simulation repeatability, three things are required: input recordability, output recordability, and input specifiability. Input and output recordability mean that all computer inputs and outputs can be recorded in a format suitable for both the programmer-analyst and subsequent machine processing. Input specifiability means that it is possible to duplicate the inputs exactly, with regard to both numerical values and temporal sequencing. Repeatability and specifiability , particularly the latter, are very difficult to achieve in a hybrid simulation.

In essence, because of its speed a hybrid simulation allows many more simulations to be run, increasing the chances of uncovering software problems. It is a poor tool for diagnosing these problems, once found: it does not provide sufficient information about internal software behavior and repeating the conditions that spotlighted the problems is difficult.

The diagnostic capabilities of the hybrid simulation could be improved if the onboard computer could operate in conjunction with a monitor computer. In such a setup, the monitor computer has greater access to internal computer operations, can simulate onboard computer internal operations to which it does not have access, and can record the detailed information obtained for later formatting and output. The operation of the monitor computer must not be allowed to interfere with the operation or real-time synchronization of the onboard computer. The onboard and the monitor computers must, of course, have facilities to permit monitoring in real time; this requires that the onboard computer possess capabilities not utilized for flight operations. The diagnostic-oriented design of the onboard and monitoring computers combination can solve many problems not solved in current designs.

Although the specific details of the required design characteristics have not been examined, a few examples will be presented to indicate their nature. One approach, illustrated in Figure 25, is to have the monitor computer share the onboard computer memory with the onboard processor. The monitor computer can receive and record the inputs and outputs of the onboard computer; detect and duplicate the execution of each onboard computer instruction during program execution; and control the onboard computer clock, enabling the monitor computer to start or stop it when non-real-time operation is tolerable. The monitor computer's processor would have to be

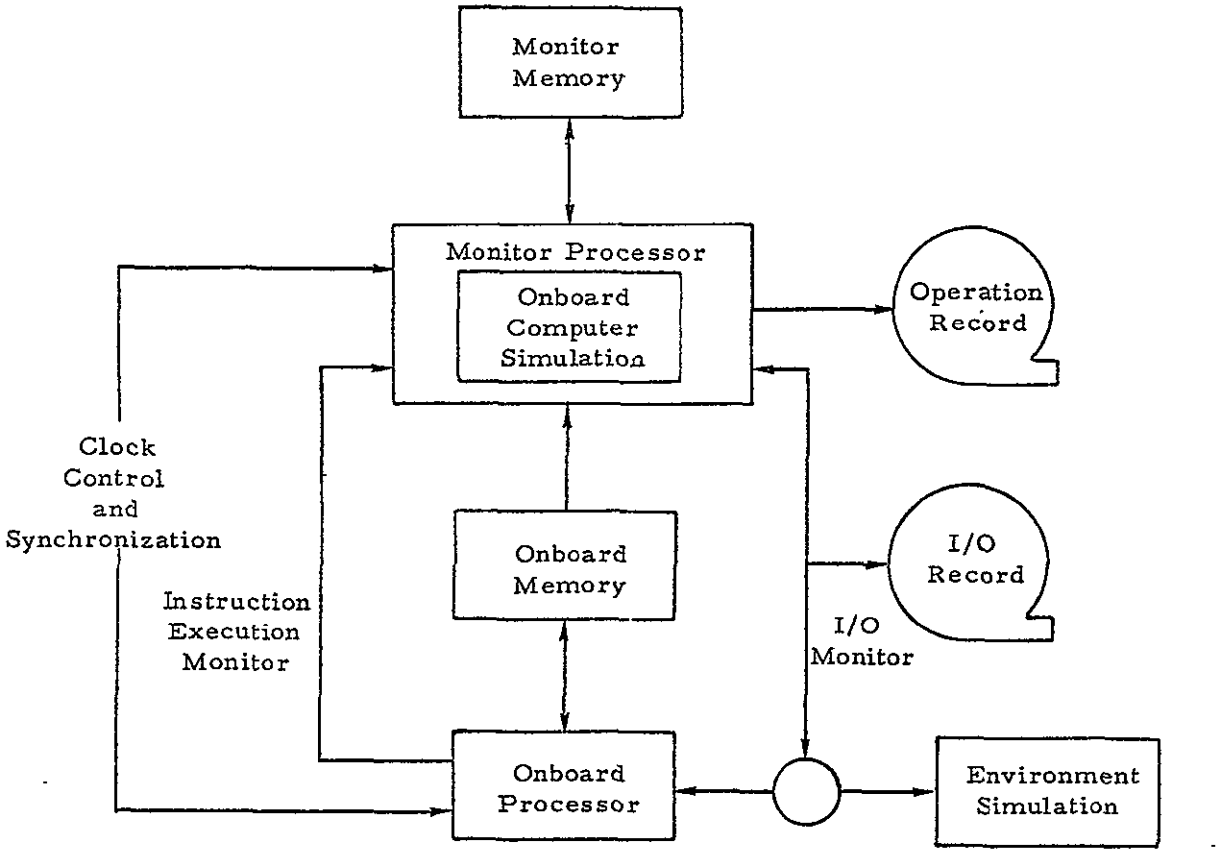


Figure 25. Hybrid Simulation, Diagnostic Configuration

significantly faster than that of the onboard computer, enabling it to simulate the onboard program's execution for instructions not involving memory reading or writing. The simulator of the onboard computer, which is part of the monitor computer, is no less accurate than an interpretive computer simulator, but would be considerably simpler for several reasons. First, it needs a minimal control loop since it receives, from the onboard processor through the instruction execution monitor, an indication of what instructions it should be simulating. Second, it has access to all of the inputs, outputs, and memory of the onboard computer. Finally, since the monitor computer is specially designed or adapted to operate in conjunction with the onboard computer, the two can be identical as to instruction set and other architectural characteristics, greatly simplifying development of the onboard computer simulation.

A difficulty in implementing such a diagnostic hybrid simulation lies in enabling the monitor computer to access the onboard memory without interfering with the real-time behavior of the onboard computer. One way of accomplishing this is to have an onboard memory with twice the effective speed needed as far as the onboard processor is concerned. Another way, although less desirable because it may introduce differences between the hybrid simulation's and actual computer's behavior, is to employ the double-speed memory in the hybrid simulation only and to use a similar, but slower, flight-qualified memory in the actual vehicle. In simulation, then, every second memory access cycle would be assigned to the monitor processor, while during actual flight every second memory access cycle would be a do-nothing operation. This effective reduction in available memory cycle time is a great price to pay for achieving verification suitability; and a detailed examination of other hardware alternatives to achieve the same end is required. An example of a less powerful scheme that should be studied for the onboard hardware is to allow the monitor to access the onboard memory only during the intervals when the onboard processor is not accessing it, such as when long multiply and divide instructions are being executed. Another scheme is to maintain a duplicate of the onboard memory within the monitor memory, updating it by means of the information received from the instruction execution monitor, with the monitor processor directly accessing the onboard memory only in non-real-time situations.

The development of such a hybrid simulation diagnostic configuration would require extra hardware to be added to the basic onboard configuration, since no discrepancy can be tolerated between the onboard computer's behavior in the hybrid simulation and its behavior during an actual mission. One example, in which this hardware has a faster memory than is actually utilized, has been discussed; other examples include the connectors, amplifiers, and isolation circuitry needed to accomplish the desired data paths. The benefits obtainable by such a hybrid simulation diagnostic configuration are the greater amount of software testing and verification made possible in real time, and the corresponding reduction in the reliance that must be placed on interpretive computer simulations to completely diagnose problems first observed using hybrid simulations.

8.4 Other Verification Tools

The simulations discussed in the previous paragraphs have several disadvantages. From the standpoint of verification, the main one is that they can indicate the presence of software errors but cannot prove their absence. Thus simulation may lead to the detection of an error through the observation of an irregularity in the output, but the failure to observe any

irregularities may simply be due to the fact that the right combination of error-inducing input values and timing has not yet come about. In the past, this deficiency of the simulation as the principal verification technique has been compensated by performing very careful and detailed visual checks of the software (often called desk- or sight-checking), and by also performing such a large number of simulations that most error-producing combinations will be exercised.

The anticipated increase in the size and complexity of the Space Shuttle onboard computer program will make the task of performing visual checks much more difficult and, coupled with an increase in the ratio of simulation time to real time, will reduce the completeness of the simulations that can be performed in a reasonable time. Therefore, verification approaches other than using simulations must be investigated.

One such type of verification tool, the automatic flowcharter, has already been utilized in some verification activities. This tool produces flowcharts indicating the operations performed by, and the logical structure of, the onboard computer program. The flowcharts it produces are similar to those made by a programmer-analyst, but more regular and without human errors. An automatic flowcharter suitable for verification is similar to flowcharting programs intended for software documentation purposes; these are offered by many software vendors. Unlike documentation flowcharters, a verification-oriented flowcharter accepts as input either machine or assembly language rather than a high-order language, and can neither rely on nor utilize the comment fields in producing an acceptable and readable flowchart. The flowcharts it produces are compared with the program specification, and where discrepancies are detected, the reason for and consequence of the discrepancy are determined either by visual examination of the program or by exercising the suspect area with the appropriate simulation.

Because it must perform a fairly complete code analysis to produce its output, the automatic flowcharting program can easily be augmented to perform repetitive checking operations. One example of such an operation is the generation of a comprehensive cross-reference list containing the names and addresses of all program variables, the program instructions referencing each variable by name, the means of reference (e.g., direct addressing, indexing, or indirect addressing), and the type of referencing (i.e., reading or writing). Where the same address has multiple references with different names, as would be the case when data regions are overlayed, the cross-reference list would name the program elements sharing the common location to aid in finding any data conflicts between otherwise independent elements.

The advantages of automatic flowcharters as compared to simulations are the reduced computer time they require in use, the facility they provide for early checking of program segments that will normally be executed late in the mission, and the intelligibility of their outputs to nonprogrammers. Although the use of an automatic flowcharter requires the preparation of a correct, detailed, and complete program specification to serve as the comparison standard, this requirement should not be viewed as a disadvantage; such a programming specification should be prepared in any case. Constructing a practical flowcharter requires that restrictions be placed on the use of tricky or obscure code sequences in the program to be checked, but this merely means enforcement of good programming practices and should not be viewed as a disadvantage.

The most significant real disadvantage of flowcharters, and one that prevents them from becoming the primary verification tools, is that the technique has not been used enough to provide the level of confidence obtained by observing that the program actually operates correctly during simulation, even though the number of simulation runs is necessarily limited. This is reasonable: given the current state of their development, it is not expected that automatic flowcharters would be able to reveal all of the software errors in a given Space Shuttle computer program. In particular, omissions or ambiguities in the programming specification and error in program timing, interaction, and sequencing may not be uncovered. The only way to overcome this disadvantage is to utilize the automatic flowcharter in parallel with conventional verification simulations. If, as expected, using the automatic flowcharter results in the discovery and diagnosis of many errors before they are detected in simulation, its worth will be demonstrated. For this reason it is recommended that an automatic flowcharter be developed and used early in the Space Shuttle program verification process. The cost of developing the tool, while not negligible, would be repaid by the fewer number of simulation runs otherwise required for verification. The automatic flowcharting program can execute on any conventional large-scale digital computer available; the difference in suitability between computers is largely due to the availability of graphics support software. Although flowcharts can be produced on a conventional line printer, greater readability and conciseness requires a graphic output device such as a high-speed digital plotter or a CRT/microfilm display and recorder.

As regards still other verification concepts, two general approaches have been explored. One would minimize the manual comparison of machine-produced flowcharts against the programming specification by mechanizing the comparison, just as the creation of flowcharts is mechanized. This

would require error-free and onboard-program-independent encoding of the program specification itself into a form suitable for internal computer representation. Such an approach is roughly equivalent to producing two versions of the program in different languages, one of which is machine-oriented, and comparing the two. Except for the reduction in manual effort, this approach does not offer any verification advantages for Space Shuttle software development; indeed, it would require the additional step of demonstrating the correctness of the way in which the programming specification was encoded. Only in a limited area is the performance of such automatic comparisons recommended: checking the values of program constants and discovering code sequences that violate established conventions or ground rules can best be accomplished using automatic verification tools.

The other verification concept referred to above is the proving of program correctness by analytic methods. Basically, this consists of establishing the state and environment of the computer before the program is executed, determining the state of the computer after the program has been executed, and comparing the final state with the desired state. The description of the computer's final state includes not only the variables that were computed and retained in the computer memory but, more importantly, all variables that were output. In contrast to simulation, actual numeric values of variables would not be determined; instead, general expressions representing the relationships between variables would be constructed. For example, consider the program illustrated in Figure 26. By simulation it would be possible to establish that if the input variable at point a were -2, then the value of X at point b would be +2. Using simulation alone, without further analysis of the program itself, many test cases would be required to establish that the X at point b is always equal to the absolute value of A. A program correctness proof for such a flowchart would assert that X at point b would be equal to A if A at point a is greater than or equal to zero, and would be equal to -A otherwise. This assertion can then be replaced by the equivalent assertion that X at point b is equal to the absolute value of A at point a.

The advantage of such a program correctness proof is that the assertions as to program behavior are valid for all data values, whereas simulations can be used to rigorously establish program behavior only for a limited number of data values. Of course in the simple example provided, the behavior for all data values can be inferred from a limited number of test cases; in a more complex program this is not often the case.

Analytic program correctness proofs are described further in Appendix C. The theory of program correctness proofs for relatively ideal cases has

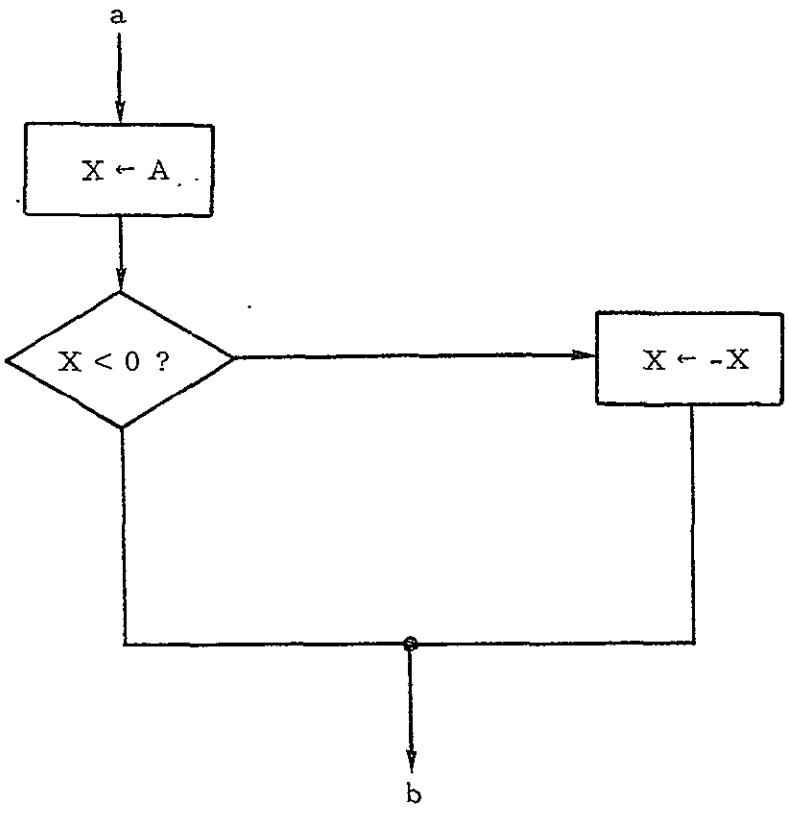


Figure 26. Absolute Value Sequence

been studied for several years. The concepts outlined in Appendix C are a distillation of the findings until now, together with some new refinements that are a necessary first step in applying past work to the Space Shuttle. As indicated in the appendix, it is now possible to prove the correctness of small but fairly complex programs by manual methods. The next step should be to determine the limits, with regards to program size and complexity, within which these manual techniques are practical. The principal problems with current program correctness proof techniques are the complexity of the proofs developed and the considerable manual effort required to produce them. One reason for their complexity is that many of the assertions about a program's behavior do not relate to specific aspects of the programming specification, but result from internal program operations. The complexity can be minimized by intelligent selection of the significant assertions by the person developing the program correctness proof, but this only increases the manual effort required. Concern

with the amount of manual effort stems not only from its cost. What is even more important is the possibility that an error will be made, leading to an assertion that an incorrect program is correct.

Completely automatic methods for proving program correctness probably cannot be developed soon enough to be of benefit in Space Shuttle verification. A more promising approach, within the Space Shuttle time scale, is to develop a computer program for checking manually generated proofs. No saving in verification effort would result, but the possibility of error could be greatly reduced. One way for performing machine-checking of manual proofs is to insert assertions of program behavior in appropriate places in the program to be examined and then have the computer check whether the assertions are valid.

Although program correctness proofs are a new and untested verification technique and cannot be expected to replace simulations as the major verification tools, it is recommended that the approach be further developed. First, it will supplement simulation activities, and experience in its application may allow reductions in the number of simulations that must be performed during verification. Second, the insight gained into program behavior will indicate program design approaches that facilitate verification, whether by correctness proofs or simulation. The methods presented in Appendix A for detecting program interaction and utilizing memory locks, and the method presented in Appendix B for locating save instructions to achieve restart protection are in part a result of the concepts explored for program correctness proofs.

8.5 Master Test Plan

The development and use of verification tools described in the preceding sections must be coordinated to minimize overall verification cost and maximize verification effectiveness. To this end a master test plan should be developed. The importance of doing so cannot be overemphasized, for the costs resulting from using duplicate or incompatible verification tools and from unanticipated reiterations during verification can far exceed the hardware or software costs associated with any particular computer architectural feature.

The following paragraphs describe the three aspects that the master test plan must cover; these are the levels at which verification activities should

be conducted, the allocation of verification tools to the tests conducted at each level, and the coordination of simulation tool development.

8.5.1 Verification Levels

The size and complexity of the onboard software make testing only the final, integrated program completely infeasible. Delaying verification until the complete program has been constructed would make it very difficult to isolate errors or anomalies. The calendar time and simulation facility time needed to perform all of the verification tests for every mission phase would both be excessive. Management would not have adequate visibility into the status and quality of the software being developed. And of course detecting errors early in software development simplifies their correction and greatly reduces total testing requirements. For these reasons, verification must be done at each stage in the development of the onboard program. The phasing should be such as to test small software units thoroughly; to assemble these units into large units that can be shown to be error-feee by testing the interfaces between the smaller units; to test the growing program for progressively longer mission phases and/or more complex subsystems; and, finally, to compare the performance of the complete, integrated program with the performance observed in all previous tests and with the mission requirements. Thus at least five distinct verification levels must be identified in the master test plan:

- Functional testing
- Unit testing
- Mission phase/subsystem testing
- Integrated testing
- Mission verification

For each of these verification levels, specific requirements must be satisfied before testing at that level can be considered complete. This does not mean that progressing from one level to another must await satisfaction of all requirements at the lower level; indeed, much testing may be done in parallel. However, the most cost-effective place in which to discover errors is the earliest possible level.

8.5.1.1 Functional Testing: Functional testing evaluates software performance at the equation level to determine if the software design itself can satisfy all possible mission requirements. Thus it is concerned with the algorithms to be coded, rather than the code itself. The following are examples of performance aspects that must be demonstrated correct at the functional testing level:

- Accuracy and stability
- Operator interfacing
- Redundancy requirements, including backup, reaction to failure, and reconfiguration
- Capability of being verified
- Subsystem loading requirements, including volumes and rates

Functional testing is initiated as early as possible, that is, as soon as a fairly complete programming specification is available. When additions or modifications to the programming specification are made, functional testing is repeated to verify their correctness and assess their overall software impact. The results of functional testing are used as references for comparison with results of testing at higher levels. Another important functional testing output is the identification of performance limits for the software design. Clearly if the basic design is adequate only within certain limits, the implementation of that design will very probably lie within the same performance envelope.

The principal tool employed in software testing is the engineering simulation. Because of the speed of the engineering simulation, a return to the functional testing level may often be advantageous in the diagnosis of software deficiencies at a subsequent testing level.

8.5.1.2 Unit Testing: This is the first level at which the actual coding is tested, rather than the algorithms or some intermediate representation of the ultimate onboard program. A unit may be a subroutine, a program module, or any other identifiable program element that performs a specific function and that has interfaces with other units, receiving particular inputs and transmitting particular outputs to them. In general, units should be chosen compatible with: the subsystem hardware configuration, when the hardware is used during the mission plan, and what the functional requirements are. Thus one unit might be the computations associated with the rendezvous radar preflight checkout functions, and another might be attitude control during reentry.

At the level of unit testing, verification is conducted in terms of requirements relating to the correctness of program units as self-contained packages. The primary effort is devoted to proving that the unit is a correct

translation of the specification for it. Some of the aspects of software behavior that must be demonstrated correct at this level are:

- Unit entry and exit
- Arithmetic accuracy
- Nominal logic paths
- Off-nominal and backup logic paths
- Ranges of variables
- Data format and sequence compatibility
- Execution time

Unit testing, initially accomplished with an interpretive computer simulation operating in the open-loop mode, can profitably employ automatic flowchart generators and program correctness proofs as well. Closed-loop ICS operation usually requires a larger portion of the total onboard software than a typical unit, so its use at the unit testing level is restricted. When adequate unit testing can be done only in closed loop and the unit itself is not self-sufficient, the missing units can be replaced by their counterparts developed for the engineering simulation and already demonstrated to be correct at the functional testing level.

The isolation afforded by unit testing from interaction with other segments of code is invaluable in localizing programming errors, timing irregularities, and other difficulties at the subroutine level. Because of the isolation of units and the concomitant isolation of the effects of software errors, errors detected at higher levels can often be more efficiently diagnosed at the unit level, using the outputs from the higher levels to drive the suspect unit in an open-loop mode to reproduce the failure.

8.5.1.3 Mission Phase/Subsystem Testing: This is the testing of many program units assembled into a relatively large group that performs a specific function, such as all software used during boost, or all display system software. For the Space Shuttle, the most likely grouping for testing at the mission phase/subsystem level is by functional breakdown, such as guidance and navigation, and, within the functional breakdown, by mission phase, such as the guidance and navigation portion of boost.

Mission phase/subsystem testing differs from unit testing in that it requires the grouping of many units, and from the subsequent integrated testing in that not all of the units are used, but only those required for a specific function or period of time. The mission phase/subsystem testing level assumes that there has been some previous unit testing, and is primarily directed

toward testing the interaction of the tested units. Requirements demonstrated correct at this level include:

- Input/output processing
- Interrupt processing
- Restart provisions
- Refreshing or reconfiguring the software
- Performance of the range of anticipated mission objectives
- Operator interfacing

Mission phase/subsystem testing uses both interpretive and hybrid simulations, and also can employ automatic equation generators and program correctness proofs. Most simulations at this level are performed in the closed-loop mode.

8.5.1.4 Integrated Testing: This is the first level at which the complete onboard program is assembled and tested. Conformance with the programming specification, previously checked for single units or functional groups of units at the earlier levels, is demonstrated at this level for every unit. At the level of full integration testing, the emphasis shifts to whether the program meets overall requirements and provides all needed capabilities. Integrated testing is concerned with interaction problems that become apparent only with the final assembling of the complete program. Examples of software aspects that must be demonstrated correct at this level include:

- Integration of the assembled or compiled program
- Overlay structure
- Formatting, sequencing, and timing of data transfers between functional groups

While integrated testing may be executed on either a hybrid or an interpretive simulation, the large number of simulations required combined with the longer execution time of the ICS prevent heavy reliance on interpretive simulation. Automatic flowcharters and program correctness proofs do not have very great suitability at this level, being chiefly useful for detecting and verifying unit interactions unobservable at the lower levels.

8.5.1.5 Mission Verification: While prior levels are primarily concerned with ferreting out all errors that might lie within the program, the emphasis changes at this final level to verifying program aspects for a single

mission; that is, instead of being concerned with a broad data range, verification testing is concerned with whether specific data and procedures will properly perform the mission for which they are to be used. This requires, among other things, that complete mission studies be conducted.

Examples of software aspects that must be demonstrated correct during mission verification are:

- Satisfaction of all specific mission requirements
- Correspondence between mission tests and the results of functional testing
- Consistent and complete documentation

One good technique of determining whether mission requirements are satisfied is to compare the results of the detailed simulations used at this level (an ICS/engineering simulation or a hybrid simulation employing a model of the flight computer) with the results of the earlier functional testing for which simpler simulations were used.

8.5.2 Verification Tool Allocation

The efficient utilization of the tools and facilities discussed above requires their allocation to appropriate verification levels so that fulfillment of all requirements at each level can be demonstrated. In the past, allocations have usually been made on a local basis, with little attempt to use general tools such as engineering simulations for more than one purpose; to develop such tools according to a common standard; or to take advantage at one level of information gained at an earlier level. The arguments in favor of spending the resources necessary to make an effective allocation are straightforward: the cost of developing the support tools and facilities could be lowered, some confusion in comparing results from different simulations could be eliminated, and the amount of effort required to set up specific tests could be reduced.

Several valid approaches exist for allocating tools and facilities to verification levels, and each must be considered for the Space Shuttle. One is to determine the types of errors expected to be discovered at a level, and to allocate the tools and facilities most likely to reveal errors of the types expected to be most prevalent. For example, the first time errors in unit interaction are likely to be discovered is at the mission phase/subsystem level; hence analytic tools that check data, file, and format compatibility should be heavily used at this level.

Another approach is to make the allocation that minimizes the disadvantages of each tool and facility. For example, an ICS/engineering simulation requires comparatively large amounts of large-scale general-purpose computer time and so cannot run a large number of test cases in a short time. Thus it would be relatively unsuitable for integrated testing. The speed advantage of a hybrid simulation over an ICS is at least an order of magnitude in present systems, and may be far greater as the capability of the flight system increases, indicating the hybrid simulation's suitability for performing a large number of tests. The relatively low level of detailed diagnostic information about program behavior that can be obtained from a hybrid simulation indicates that effective diagnosis of any anomalous results will require utilization of the ICS/engineering simulation. The composite cost of verification using the two complementary simulations should be greatly reduced over the cost of using one alone, since it would be expected that few tests would be anomalous.

A third approach to allocation is to utilize the specific tools or facilities at the levels where no other setup would demonstrate satisfaction of the requirements. For example, hybrid simulation can provide a more accurate representation of computing delays and other subsystem-timing-related aspects than is possible with the ICS and should be used for demonstrating such aspects.

8.5.3 Verification Tool Development and Comparisons

The final element in the master test plan concerns the development of verification tools, with particular emphasis on the capabilities and limitations of each simulation. In the course of most past software developments, several engineering, interpretive, and hybrid simulations were produced. In part this resulted from the nature of the software development process. Different organizations having dissimilar large general-purpose computers often independently developed their own engineering and interpretive computer simulations; different organizations testing their own hardware components similarly developed hybrid simulations, with each subsystem represented to an appropriate but different fidelity. One often-expressed motivation for duplication is that an error in one simulation is unlikely to exist in a second; therefore an onboard software error that would be masked by an error in one simulation would be likely to be revealed using a redundant simulation. This motivation does not justify developing redundant simulations for verification: the effort could be better spent in ascertaining and improving the fidelity of a single appropriate simulation. Whatever the practical reasons for developing multiple simulations, the verification process does not require more than one of each type.

However many simulations may be developed, it is important that meaningful comparison between them can be made. For each simulation, the master test plan should indicate the fidelity of its constituent components, its range of validity, the mission phases for which it is appropriate, and the differences that will exist between it and other simulations. For example, the relationship between simulation execution time and real time must be described to indicate both the applications for which the simulation is useful and the number of simulations that can be performed in a given period. This information should be prepared before simulation development and updated during and after development. Specific test procedures for each simulation should be described in the plan, along with the comparisons that must be made before the simulation can be accepted as a software verification tool. To the greatest extent possible the test procedures should include the content and formats of the outputs produced, particularly those which are not normal onboard computer outputs. The output formats and the input procedures and formats should be the same where similar simulated quantities are involved. The master test plan should indicate the availability of each simulation facility from the standpoints of its development schedule, its normal setup time, and the time required by its other users. And finally, although the master test plan should be produced at a very early stage in the Space Shuttle software development, certainly before any major simulations are constructed, it must be revised as often as necessary to reflect the observed behavior of the verification tools and facilities and the changing requirements imposed on the onboard software.

8.6 Summary of Conclusions

Minimizing the verification effort needed for the Space Shuttle onboard software has several aspects. The verification effort required is a function of the size and the complexity of the onboard software; both, in turn, are determined by the functional requirements. As discussed in Section 3, several alternatives are possible depending on the hardware configuration and the overall requirements imposed on the software. The first step, then, in minimizing verification costs is to choose those alternatives which reduce the onboard software's size and complexity. The next step, as discussed in Section 4, is to select the computer configuration and the software executive design which can easily satisfy the functional requirements and which also offer those characteristics, such as repeatability, that facilitate verification. The third step is to choose a particular computer architecture which has those features that simplify software development, such as floating-point arithmetic, single-instruction restart capability, and a flexible interrupt mechanism, for these features will also simplify software verification.

For a given set of functional requirements and a particular computer configuration and architecture, software verification costs can be reduced by improving the tools and facilities used and by preparing and following a master test plan for the efficient utilization of these tools and facilities. The most promising improvements that should be made in the simulation tools include increased automatic checking and more flexible diagnostic control for interpretive computer simulations, and more visibility into the internal software operation for the hybrid simulations. The interpretive computer simulation provides the greatest insight in the internal behavior of software, but a major deficiency in light of the expected size and complexity of the Space Shuttle software is the length of time needed to perform simulations. By increasing the information gained from each interpretive simulation through the use of the suggested improvements, the total number of simulations required can be reduced. By enriching hybrid simulation capabilities in the suggested areas, this tool can be a practical substitute for the interpretive simulation in many areas. In addition, verification tools such as automatic flowcharters and program correctness proof checkers should be developed to supplement the simulation tools. Finally, the tools themselves must be developed and used in accordance with a coordinated plan that will enable the detection of errors as early as possible, the easy diagnosis of errors, and the comprehensive testing of all significant software characteristics.

The further development of the concepts presented into a complete, dynamic, and useful master test plan document is the next step that must be taken to assure that the development and application of the verification tools are both done with the least cost and schedule impact. Even with the best of tools and plans, software verification will be a major constituent in the Space Shuttle software development process. To make certain that verification can be accomplished in an effective and timely manner, it is imperative that verification continue to receive early attention.

APPENDIX A
TASK INTERACTION

For software that has a large number of tasks executable in many permutations and combinations, demonstrating that the tasks do not interact undesirably or erroneously is very difficult. The problem is particularly apparent for interrupt executives: two routines, each of which may be internally correct, can together cause a software failure to occur when one routine interrupts another at an inopportune point in its execution. This appendix presents techniques for detecting undesirable logical interactions between an interrupting routine and the routine(s) it interrupts. It shows that, from the standpoint of logical interaction, the question of allowing routine B to be executed in parallel with routine A is identical to the question of allowing routine B to interrupt routine A. Formal criteria already existing for determining whether two routines can be executed in parallel are applied to the problem of determining situations where an interrupt could cause a software failure. These criteria are also developed for determining situations in which potentially undesirable task interactions can occur between two tasks that could be executed in parallel, as in a polling executive. The results for the interrupt and polling cases are then compared. Finally, hardware and software methods are described for eliminating interference problems.

A-1. ANALYTIC DETECTION OF LOGICAL INTERFERENCE

In considering the memory locations accessed by a routine A, four disjoint sets can be distinguished:

$M_1(A)$ = the set of memory locations which are only read by A.

$M_2(A)$ = the set of memory locations which are first read by A. and later written into by A.

$M_3(A)$ = the set of memory locations which are only written into by A.

$M_4(A)$ = the set of memory locations which are first written into by A and later read from by A.

Now define

$$R(A) = M_1(A) \cup M_2(A) \cup M_4(A)$$
$$W(A) = M_2(A) \cup M_3(A) \cup M_4(A)$$
$$R_f(A) = M_1(A) \cup M_2(A)$$

Thus R(A) is the set of locations which are read from by A at some point, and W(A) is the set of locations which are read first by A. From a practical standpoint, it is important to add that if routine A is involved in identification activity, the inputs to A must be included in R(A) and the outputs from A must be included in W(A).

In deciding whether two routines can be executed in parallel, the situation in Figure A-1 is considered. A and B are two segments of code (routines) that tentatively have a serial ordering. P represents the remainder of the program. The basic observation on logical interaction is: if, during the execution of the routine, none of the memory locations which it reads are modified by another routine, then that routine is guaranteed free of undesirable logical interference. This observation is embodied in the following three conditions, which ensure that two routines A and B can be executed in parallel (as shown in Figure A-2):

$$R(B) \cap W(A) = \varphi$$
$$R(A) \cap W(B) = \varphi$$
$$W(A) \cap W(B) \cap R_f(P) = \varphi$$

The first condition states that B does not read any memory locations written into by A. The second condition stated that A does not read any memory locations written into by B. The third condition states that P does not first read from any locations which are written into by both A and B. If this last condition were not true, a race condition would exist: the execution of P would depend on whether A or B was the last to write into the elements of $W(A) \cap W(B)$.

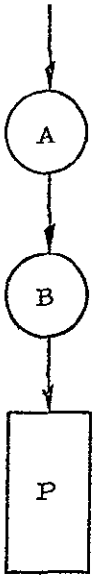


Figure A-1. Serial Task Execution

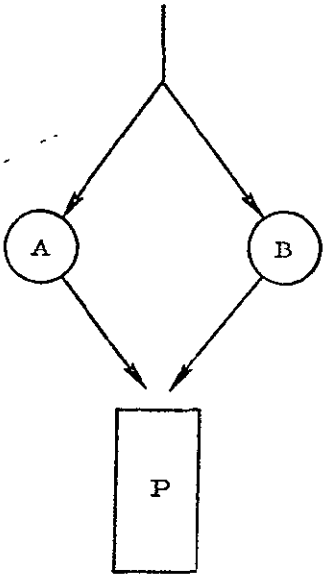


Figure A-2. Parallel Task Execution

Now consider the problem of interrupts. Two routines, A and B, executed in normal order can be indicated as in the first part of Figure A-3. If B interrupts A, the execution sequence can be indicated as in the second drawing. A_1 and A_2 indicate the two segments of A which are separated by the execution of B. α indicates the interrupt point. P indicates succeeding routines. Again absence of undesirable logical interaction can be guaranteed if the memory locations which a routine reads are undisturbed. The conditions for a particular interrupt situation are:

$$W(A_2) \cap R(B) = \varphi$$
$$R(A_2) \cap W(B) = \varphi$$
$$W(A_2) \cap W(B) \cap R_f(P) = \varphi$$

The interrupt point α may occur anywhere in A, and the instructions comprising A_2 will always be a subset of instructions comprising A, i.e., $A_2 \subseteq A$. Thus the general conditions are:

$$W(A) \cap R(B) = \varphi$$
$$R(A) \cap W(B) = \varphi$$
$$W(A) \cap W(B) \cap R_f(P) = \varphi$$

These conditions are identical to those which guarantee that A and B can be executed in parallel as shown in the third illustration of Figure A-3.

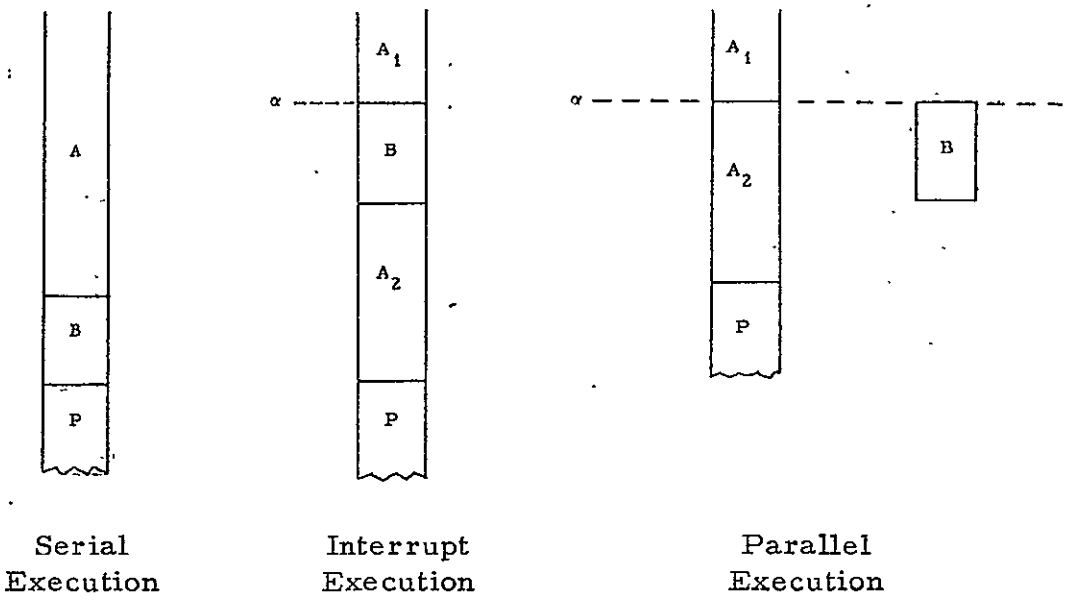


Figure A-3. Task Sequencing

It has been proposed that the verification problem posed by interrupts might be solved if all routines were segmented, with "interrupts" permitted only at the end of a segment rather than at the end of any instruction. It is certain that this suggestion was made on the implicit assumption that verification of interrupts would be performed by the ad hoc manual techniques which have been used in the past. The development of analytic interruptibility criteria brings into question the desirability of the segmentation idea. The obvious disadvantages of segmentation are:

- Degraded response time (since an interrupting routine must wait until a segment has been completed)
- The extra effort required to program routines in segments

It is felt that in some instances this last point could become a major burden on the programmer.

To further investigate the segmentation suggestion we will need the concept of execution commutivity. Two routines A and B will be termed commutative if the execution sequence A, B produces the same results as the execution sequence B, A. The conditions which ensure commutivity of two routines are slightly less restrictive than the conditions which ensure interruptibility. Only locations in the set R_f must remain unchanged. If a read from a location is preceded by a write into the same location, a prior write by another routine has no effect. The commutivity conditions take the form:

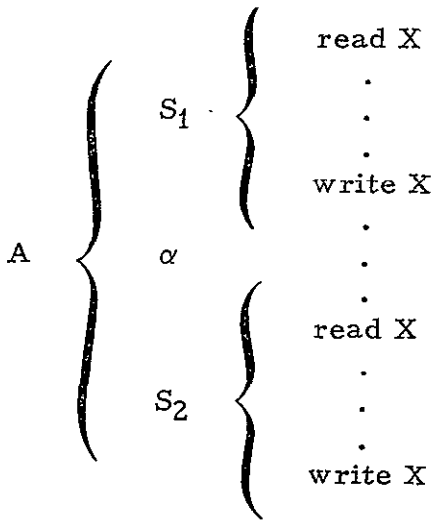
$$R_f(A) \cap W(B) = \varphi$$
$$R_f(B) \cap W(A) = \varphi$$
$$W(A) \cap W(B) \cap R_f(P) = \varphi$$

If the R's and W's are replaced by the corresponding M's in the conditions for commutivity and interruptibility and the redundant terms are eliminated, the terms which must be null are found to be:

Commutivity	Interruptibility
$M_1(A)\ M_2(B)$	$M_1(A)\ M_2(B)$
$M_1(A)\ M_3(B)$	$M_1(A)\ M_3(B)$
$M_1(A)\ M_4(B)$	$M_1(A)\ M_4(B)$
$M_2(A)\ M_1(B)$	$M_2(A)\ M_1(B)$
$M_2(A)\ M_2(B)$	$M_2(A)\ M_2(B)$
$M_2(A)\ M_3(B)$	$M_2(A)\ M_3(B)$
$M_2(A)\ M_4(B)$	$M_2(A)\ M_4(B)$
$M_3(A)\ M_1(B)$	$M_3(A)\ M_1(B)$
$M_3(A)\ M_2(B)$	$M_3(A)\ M_2(B)$
$M_3(A)\ M_3(B)\ R_f(P)$	$M_3(A)\ M_3(B)\ R_f(P)$
$M_3(A)\ M_4(B)\ R_f(P)$	$M_3(A)\ M_4(B)$
$M_4(A)\ M_1(B)$	$M_4(A)\ M_1(B)$
$M_4(A)\ M_2(B)$	$M_4(A)\ M_2(B)$
$M_4(A)\ M_3(B)\ R_f(P)$	$M_4(A)\ M_3(B)$
$M_4(A)\ M_4(B)\ R_f(P)$	$M_4(A)\ M_4(B)$

The two lists are almost identical. However, in three instances an extra $R_f(P)$ term makes the commutivity conditions slightly more complex. To analytically check interaction of two segmented routines A and B it is necessary to check the commutivity of each segment of A with each segment of B. Define C_1 as the amount of computation required to check interruptibility of two routines A and B. Define C_2 as the amount of computation required to check the commutivity of each of n segments of A with each of n segments of B. Define C_3 as the amount of computation required to check commutivity of A and B. We have already seen that C_1 is slightly less than C_3. A comparison between C_2 and C_3 is desired, so that C_1 and C_2 can be compared. To compare C_2 and C_3 we make the following argument:

With segmented commutivity, each set M_i would contain roughly $1/M$ as many elements as with unsegmented commutivity. Hence, with segmentation, checking the nullity of each double term (e. g., $M_3(A)\ M_1(B)$) would require roughly $1/n^2$ as much computation. However, since each routine is divided into n segments, there are n^2 commutivity combinations to check instead of one. This would imply that C_2 is roughly independent of n, i. e., $C_2 \approx C_3$. However, in general each set M_i will contain <u>more</u> than $1/n$ as many elements. A memory location may be accessed several times, but it will appear only in one of the sets M_i. If segmentation divides the accesses into two parts, the total number of elements is actually increased. To illustrate this the following example is presented.



X would fall in the category $M_2(A)$. If segmentation occurs at point α, then $X \in M_2(S_1)$ and $X \in M_2(S_2)$. Segmentation can also cause elements to change sets. Hence $C_2\ (n+1) \geq C_2(n)$, and in general the greater-than relation will hold, i.e., C_2 increases with increasing n. Consequently $C_1 < C_2$ and the segmentation idea is judged disadvantageous if interrupt verification is performed analytically.

A-2. PREVENTION OF LOGICAL INTERFERENCE

It has been shown how logical interactions between routines can be detected or shown to be nonexistent. The next question is: If a logical interaction is detected, what, if anything, can be done about it? The following discussion presents alternative methods for "curing" the problems once they are detected. Both hardware/software and software/software tradeoffs are discernible.

A-2.1 Discrimination by the Executive

If a software failure will occur when routine B interrupts routine A, one solution is to prevent B from ever interrupting A. For each routine a list can be prepared of all routines which are permitted to interrupt it (or which are not permitted to interrupt it, if this list is shorter). If the executive would interrupt routine A and begin routine B, it must first check routine A's list to see if the interrupt is allowable.

A-2.2 Memory Locks

To make any routine A secure from undesirable interrupts, the following requirements are sufficient:

1) Read locks are placed on all locations in the set W(A) at the beginning of A. After the last write statement in A for a given location, the lock on that location is lifted.

2) Write locks are placed on all locations in the set R(A) at the beginning of A. After the last read statement in A for a given location, the lock on that location is lifted.

3) Write locks are placed on all locations in the set W(A) at the beginning of A and lifted at the end of A.

The pointer lock is one method for implementing requirement 1 or 2 for a linear data structure.

To see that these locking procedures are effective, recall the three conditions for interruptibility:

$$R(B) \cap W(A) = \varphi$$
$$R(A) \cap W(B) = \varphi$$
$$R_j(P) \cap W(A) \cap W(B) = \varphi$$

Requirement 1 exactly satisfies the first condition, since the read lock on W(A) halts any routine B only if it tries to read from W(A). Requirement 2 exactly satisfies the second, since the write lock on R(A) halts any routine B only if it tries to write into R(A). Requirement 3 oversatisfies the third condition. The write lock on W(A) will halt any routine B if it tries to write into W(A). This is equivalent to the condition $W(A) \cap W(B) = \varphi$, which implies that the third condition is true.

A simpler procedure is to place a total lock on R(A) and W(A) for the entire duration of routine A. The disadvantage is that other routines will sometimes be blocked unnecessarily.

A-2.3 Data Duplication Schemes (Buffering)

For the important special case where the routine to be interrupted is performed repeatedly, the common memory locations can be made proprietary by duplicating them. Figure A-4 shows the situation before correction. $M = W(A) \cap R(B)$, that is, M is the set of locations written into by routine A and read from by routine B. Consequently B may not interrupt A. The dashed arrows indicate the cyclic performance of A, with delay function Δ. B may also be cyclic, but this is not indicated. One solution is shown in Figure A-5. Here M_1 and M_2 are duplicate sets of memory locations. C is a routine or possibly a single, uninterruptible block transfer instruction

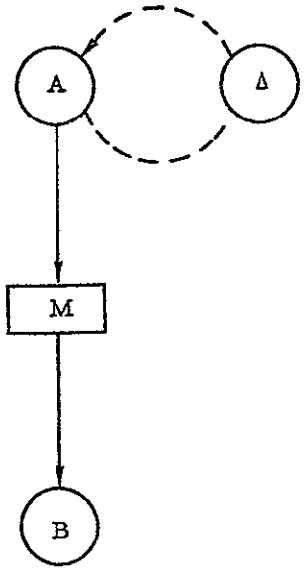


Figure A-4. Single Memory Region Accessing

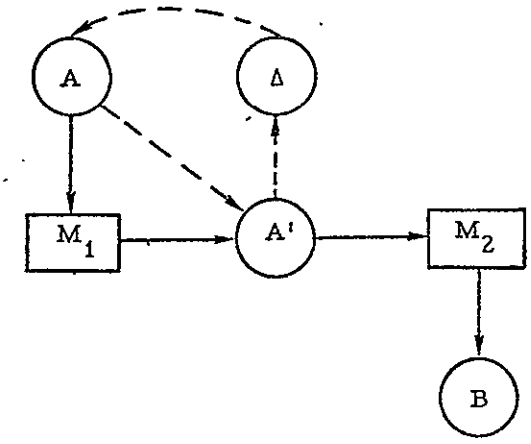


Figure A-5. Memory Buffering

that copies the contents of M_1 into M_2. C is always executed after execution of A. Since now $W(A) \cap R(B) = \omega$, B can be allowed to interrupt A after the first iteration of A, or at any time, if M_2 is initially primed with values. However, B cannot interrupt C, since $W(C) \cap R(B) = M_2$. Thus noninterruptibility of A has been traded for noninterruptibility of C. This is often a good trade since C will generally take much less time to execute than A. Consequently, this solution is useful in instances where B cannot wait until A is completed. The cost is the duplication of M and the cost of C.

It will be noted that C cannot be permitted to interrupt B. From a logical standpoint, there is no reason why C should be permitted to interrupt B. A' is allowed to begin its computations (using possibly volatile data) on schedule, and only the copying operation need be delayed until completion of B. Either a memory lock or executive discrimination can be used to ensure that A does not interrupt B. However, if conflicts are to be eliminated solely by making C a block transfer instruction, more elaborate buffering is required. The same procedure that was used to isolate A from B can be used to isolate C from B. The result is shown in Figure A-6. B copies M_2 into M_3 and is always executed immediately prior to B.

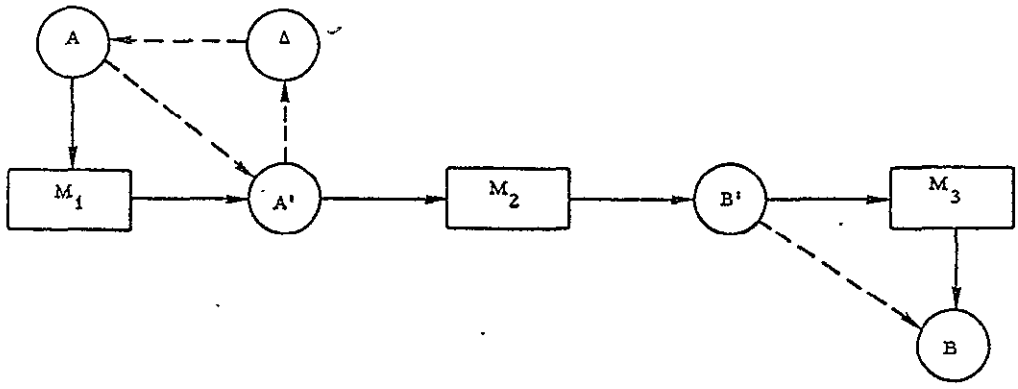


Figure A-6. Extended Buffering

An alternate solution is indicated in Figure A-7. Again M_1 and M_2 are identical. b is a single pointer bit.

$$b = 0 \Rightarrow \begin{cases} \text{A writes into } M_1 \\ \text{B reads from } M_2 \end{cases}$$

$$b = 1 \Rightarrow \begin{cases} \text{A writes into } M_2 \\ \text{B reads from } M_1 \end{cases}$$

D is a "routine" which inverts the bit b. The dashed arrow indicates program flow. Thus D is always performed immediately after A. Let u be a continuous variable indicating the iteration of A being performed. If A is not in execution or interrupted, u is an integer. If A is in execution, say working on the third iteration, then $2 < u < 3$, depending on how far through A the execution has progressed. This scheme guarantees that while A is in execution

$$R(B) \cap W(A^u) = \varphi$$

and, depending on the value of b,

$$R(B) \cap W(A^{[u]}) = M_1 \text{ or } M_2.$$

Hence B may now interrupt A at any point. Noninterruptibility of A has been traded for noninterruptibility of D, since $b = W(D) \cap R(B)$. Timewise,

this is an excellent trade, since D merely involves the changing of a single bit. The cost is the added complexity of A and B, the duplication of M, and the cost D and the bit b.

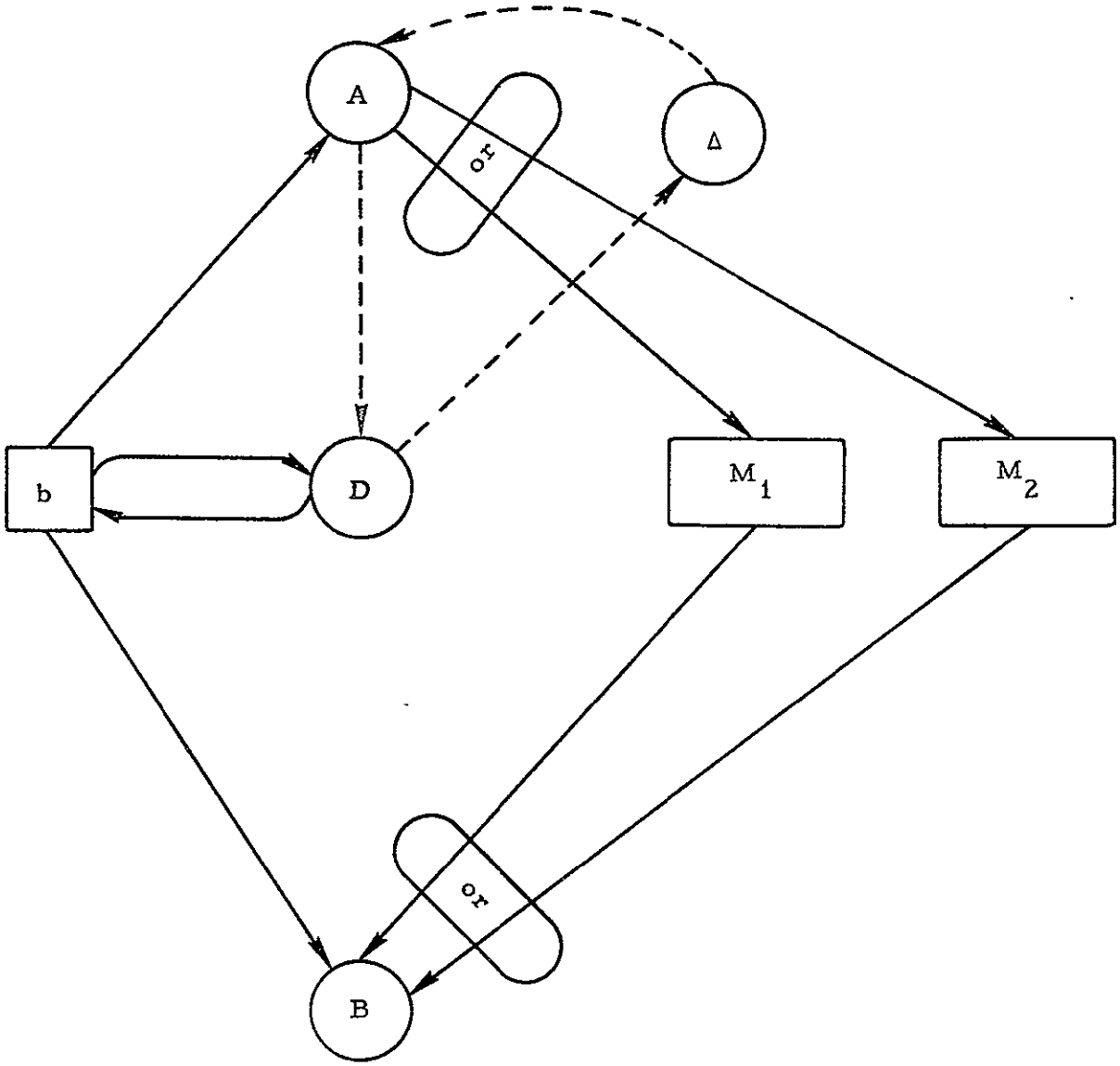


Figure A-7. Buffering with Pointer Bits

Again, D may not interrupt B. If this capability is necessary or desirable, the arrangement of Figure A-8 is required. Here B' copies the bit b_1 into the bit b_2 prior to execution of B. Now only A' and B' are not mutually interruptible, and each could be accomplished by no more than one or two instructions, representing a negligible delay.

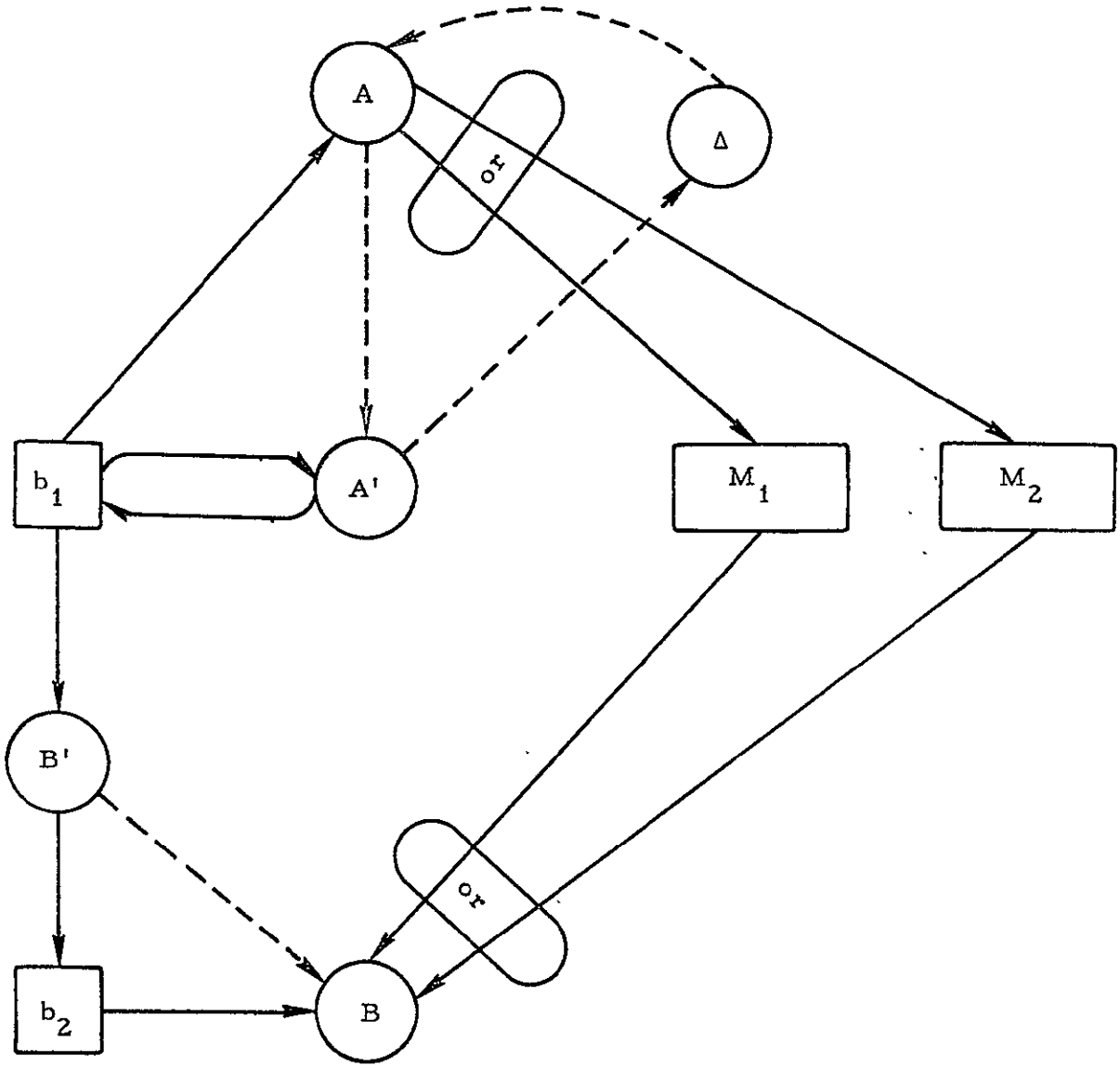


Figure A-8. Buffering with Pointer Bits (Elaboration)

APPENDIX B
A METHODOLOGY FOR RESTART PROTECTION

The Space Shuttle system is dependent on successful computer operation for mission success, if not for crew safety. While software alone can protect the system against some classes of failure, special-purpose hardware is needed for detection and diagnosis of others. Such a failure may be external to the data management system (e. g. , power loss) or internal (e. g. , parity failure). In either case, the data in the central registers of the system are suspect, and must be restored to consistent values before operation can be resumed. This appendix defines the requirements for success in a restart following a transient failure.

In order to identify the conditions for successful restart, we must first define "success." It is straightforward to pin down what is meant in the event that there are no external inputs to the program, but the problem is complicated when such outside sources as a clock, guidance sensors, or a human operator may alter the processing.

Definition: A restart is said to be successful if and only if each value calculated following recovery is identical with that which would have been calculated at the same time in the absence of the interruption.

To satisfy the condition above, all external data must be supplied appropriately despite the interruption, the program must preserve accuracy over the gap, and all data written by the program after the restart must be consistent. The hardware must accumulate or extrapolate mission data, since the program is not aware of the time lost; some software assistance may be provided in the restart routine if a noninterruptible clock is provided and appropriate code is executed. Programming must incorporate safeguards (notably in integration routines) against loss of accuracy. This appendix addresses only the problem of data consistency and protection through use of a save instruction.

The memory of the computer is divisible into volatile and nonvolatile regions. A volatile cell is one whose value may have been altered during the interruption; it is typified by the central registers. A nonvolatile cell is assumed to retain its value during the interruption, as would be the case for a magnetic core. The save instruction preserves the values in volatile memory by copying them into a nonvolatile area reserved for that function. Following interruption, the restart routine copies the saved values into volatile

memory and resumes operation at the save; any software extrapolation is performed within the restart routine, prior to copying the saved values.

For the restart to succeed, each subsequent write operation must provide a value accurate by the defined conditions. Clearly, if each datum read after recovery is accurate and the program runs correctly, then the restart is successful. Some special cases may tolerate reading an invalid datum, but these are so few and so hard to account for in analysis of the code that they will be neglected. Thus, reading of accurate data is assumed necessary as well as sufficient for successful restart. Any datum read by the program must come from one of three sources: external interface (assumed accurate by hardware or hardware/software provision); volatile memory (restored to a consistent set by the restart routine); or nonvolatile memory. It is only the last of those sources which may be in error. For a datum from nonvolatile memory to be inconsistent with the restored volatile memory, it must have been written since the last save instruction, the one to which volatile memory was restored.

To derive and prove the sufficiency condition, we consider the first invalid datum read following the restart. Its value must have been written into nonvolatile memory since the last save, i.e., since the save before the interruption. Only two cases may be constructed:

1) It was written after the save but before the interruption and not written since restart.

2) It was written after the restart.

But if it had been written after the restart, it would either have been written correctly or derived from only valid source data (since it was assumed to be the first invalid datum). Therefore:

A) It is sufficient for protection against restart failure that each read of a nonvolatile cell is separated from a subsequent write by a save instruction.

Under this condition, no cell may be the first invalid one read, since each sequence READ X ... WRITE X is interrupted by a save.

The above argument leads to a still stronger condition which may be shown to be necessary as well as sufficient. If the cell containing the first incorrect value had been written after the restart, it would have had only valid source data, hence would not have been incorrect. Consequently:

B) It is necessary and sufficient for protection against restart failure that in each sequence of code beginning with one save instruction and containing no more, each nonvolatile cell which is both read and written is written before it is first read.

The proof of necessity is based on a sequence violating the condition: SAVE ... READ X ... WRITE X, with no other save or write instruction. Ignoring the trivial cases in which the value written is always identical with that which was read, or in which a value is read but never used in subsequent code, we examine the impact of an interruption immediately following the WRITE X. After restart, the updated value written before interruption will be read instead of the value read on the previous pass. Hence, the WRITE X following restart will be incorrect and the restart will have been unsuccessful.

The complexity of condition B suggests lengthier restatement in line with the flowchart of Figure B-1. Two lists are established: one of the nonvolatile cells which are being written (T), the other of those being read (S). The lists are mutually exclusive, so that once a cell is entered into either S or T, it is not repeated in the other. The lists are erased by a save instruction in the logical flow. Thus in the sequence of code between consecutive saves, a cell read before it is written is entered into S and one written first is entered into T. The necessary-and-sufficient condition is that in such a sequence, no cell in S is written.

Figure B-2 illustrates the flow for the sufficient-only condition using the same terminology applied in Figure B-1. In both figures, progress along the logical path is traced by incrementing the instruction counter I. The additional logic required to treat branching and calls to subroutines is not shown, since it is dependent on the particular system design. Similarly, implementation-peculiar instructions, including implicit reads and writes, are not shown in the illustrations.

The choice between the necessary-and-sufficient and the sufficient-only conditions is based upon the requirements of the program. The following example is illustrative of the difference in effect which may be obtained.

Hypothesize a routine which initializes, then executes a loop. The initialization is a sequence of writes, while each pass through the loop reads, then rewrites the same variables. The necessary-and-sufficient condition requires only a save at the start of initialization, while the sufficient-only condition will require at least one save for each pass. Thus, a restart

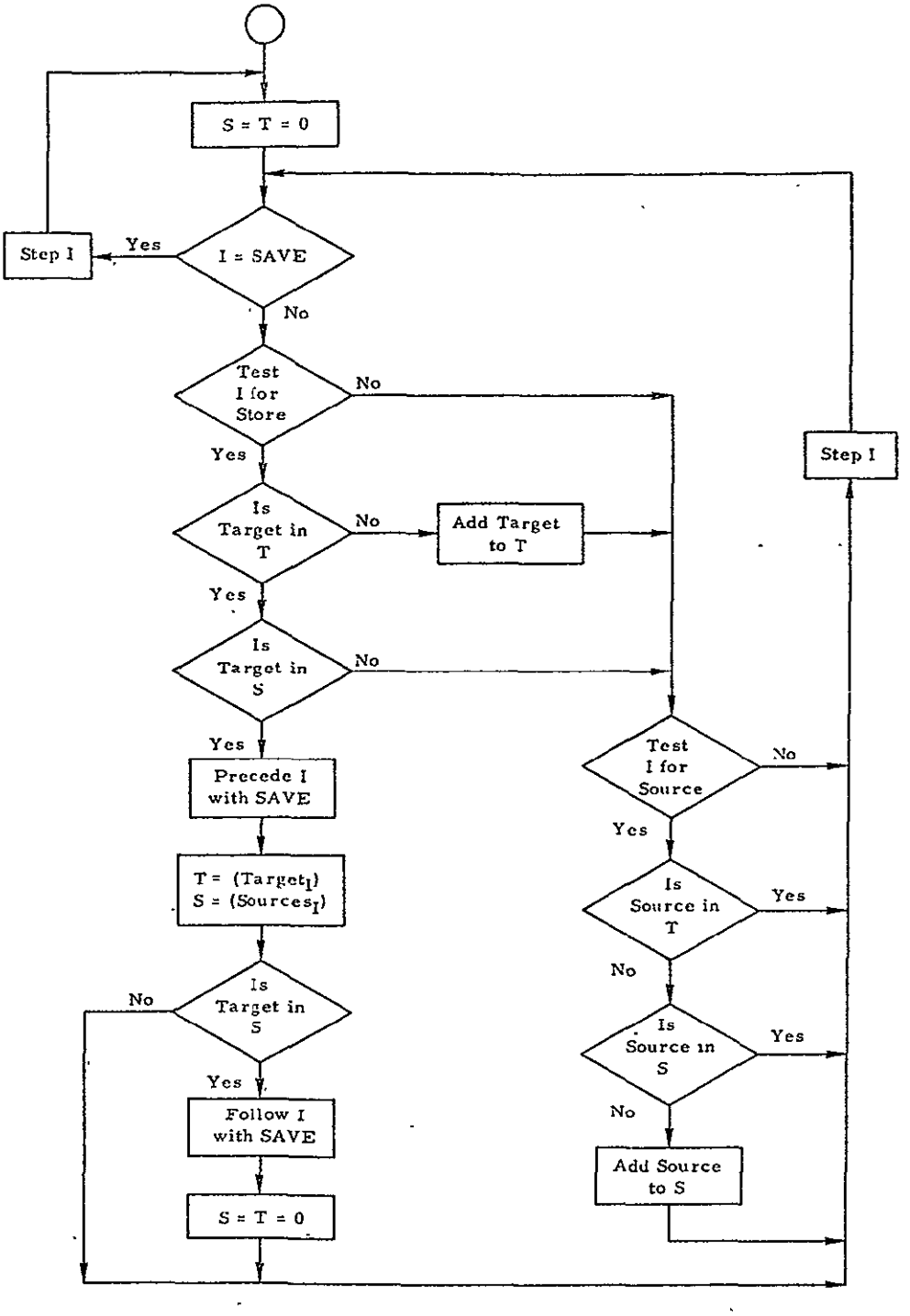


Figure B-1. Necessary-and-Sufficient Algorithm

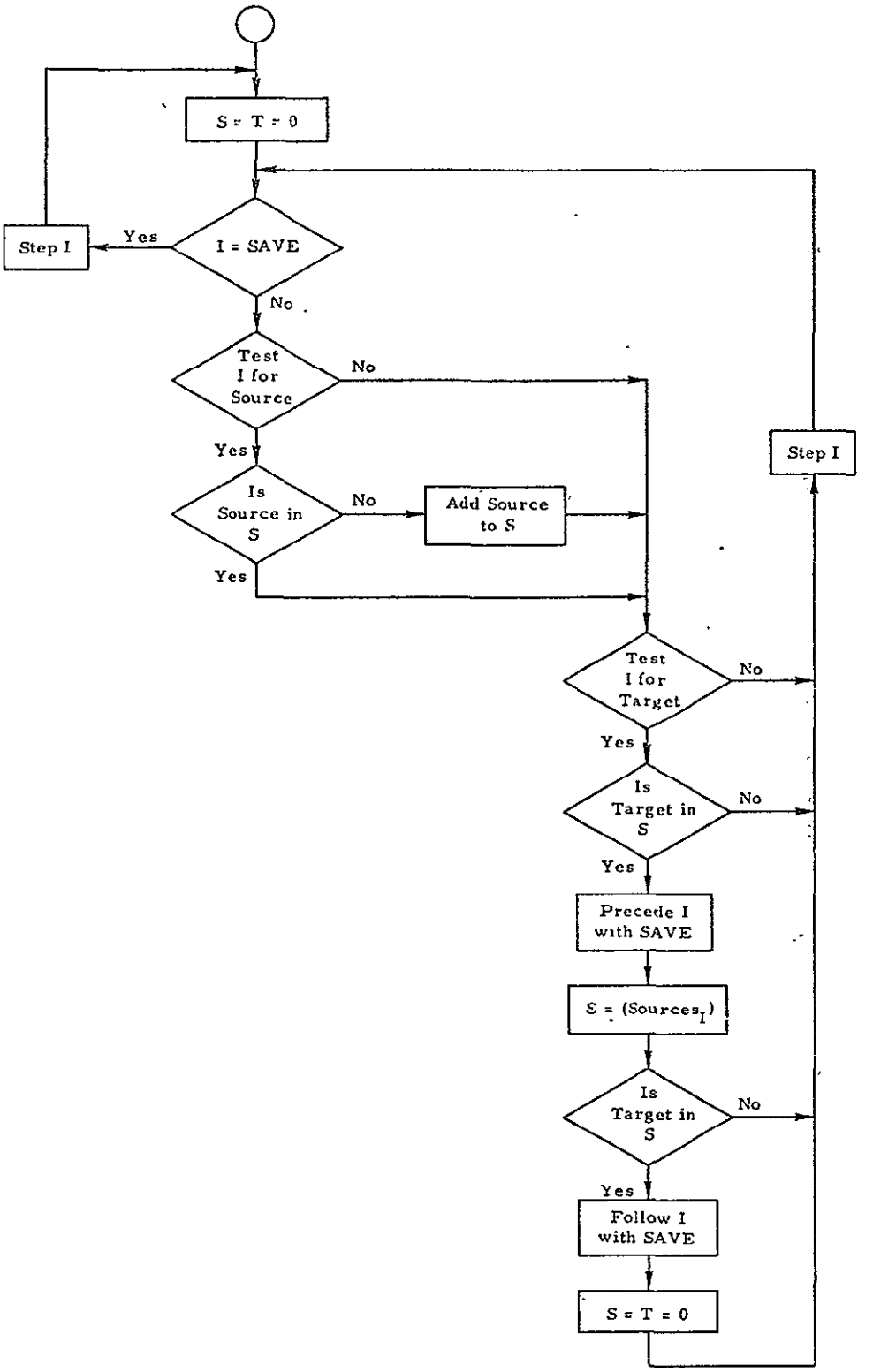


Figure B-2. Sufficient-Only Algorithm

during looping will recycle to the initialization in one case and to the last pass in the other. (Good programming practice will usually require only one save in the loop.)

One class of instructions with representatives in many languages presents problems in restart protection. These instructions carry implicit read or write operations, or both. A list-processing computer provides an implicit store into the stack with each load from core, and an implicit read from the stack with each write into memory. An instruction such as increment both reads and writes a single cell; we term such an operation reflexive. Other reflexive instructions are shifts, complement, absolute value, and sign. The reflexive instructions are established by the instruction set and must be considered separately for each machine. Some characteristics are treated here since they are applicable to a spectrum of computers.

A reflexive instruction operating on a cell of volatile memory presents no difficulty since restart protection is effected by the save. Some such instructions, notably absolute value and sign, may be repeated without altering the value of the cell, hence add no difficulty in the general case. Most reflexive operations read a cell, alter the value, then write into the same location in nonvolatile memory. In most applications, the necessary-and-sufficient condition would then require a save during the execution of the single instruction; the sufficient-only condition would always require it. The object of the reflexive instruction may have been written since the last save; in that case, the necessary-and-sufficient condition is satisfied without special attention. Otherwise, returning to the previous save on restart would cause the reflexive instruction to execute twice on the cell, leading to failure. Software methods -- somewhat cumbersome -- may obviate the problem through transfer to scratch pad, reflexive operation on the scratch location, save, then storage. The remaining alternatives are to incorporate the save within the reflexive instruction or accept the risk of interruption following the reflexive operation but prior to the next save.

Because a list processor contains many implicit read and write operations, we shall consider this configuration separately. The hardware is assumed to contain a nonvolatile list and a volatile pointer into the list to indicate the present state. A load operation adds a value to the list and increments the pointer; a store pushes the pointer below the previous location in transferring the word from the list. Any unary operation is reflexive, and one of higher order (e.g., binary or vector) is reflexive on one or more lower cells and displaces the pointer. Finally, we shall assume that there is an instruction COPYn which repeats the top n entries above the previous value of the pointer.

The top cell of the list is the most sensitive to reflexive operations. If it lies above the saved location of the pointer, on restart it will be written before it is read, hence is safe. Unfortunately, any write into the stack below the saved location is prone to restart failure. The general case has arithmetic operations and stores pushing the pointer down until it penetrates the saved location. A subsequent store is generally safe since most hardware would retrieve the value from the stack and move the pointer, but would leave the copied datum in the stack in its former position. Any other stack operation entails a write into a nonvolatile (list) location previously read. In most cases, the first such write below the saved pointer will not have been written since the save, requiring the addition of a save instruction.

The following rules safeguard the list in a simple implementation:

1) The only stack operation permitted below the saved pointer is a simple store.

2) An instruction which would violate rule 1 on execution must be preceded by a save and a COPYn, with n at least equal to the possible penetration of the instruction.

3) A sequence of store instructions penetrating the saved-pointer location must be followed by a save.

The particular instruction set of the computer dictates the implementation required, but the condition which must be met remains inflexible. For example, the HDC-701P computer for Minuteman III returns from a subroutine with an instruction that restores the saved data to those prior to the subroutine call. To preclude restart failure, no save instruction should be incorporated into a subroutine, and no need for a save can be tolerated in one. The coding required to remove the need for a save from the subroutine is extensive and costly, but without such coding there would exist a possible failure point in the program.

Implementation of the constraints may take many forms, and should be dictated by the application. If each entry point into a routine is a save instruction, that routine is isolated from the rest of the program; when more than one programmer is working on the code, such isolation is highly desirable. One implementation would call for the programmer to identify the separable routines to be coded and for the compiler or assembler to insert a save in one pass at each entry point and to add the necessary-and-sufficient saves in a second pass. As an alternative, the sufficient saves may be inserted

throughout the program by the assembler; that course is preferred except in a list processor, where stack operations should use the stronger condition. Figures B-1 and B-2, previously introduced, chart flow through an implementation for each condition. The flow might be incorporated in support programs for any level of the development: compiler, assembler, or post-coding analysis. Both the choice of level and the difficulty of incorporation are functions of the language, the host machine, and the pre-existing support tools. In some cases (e.g., languages with indexed instructions), fully automatic insertion may be impractical.

Ground systems may be less dependent than flight software on restart protection for life and ultimate success, but the financial penalty for non-restartable programs is large. The save instruction may be added to the present repertoire without increasing the size of the instruction set. It need only be added to the hardware, software, or firmware implementing the programming language for each jump instruction. It would then be implemented before storage by adding a GOTO (next instruction) where needed. A penalty is paid in the efficiency of the resulting program, since any implementation of the save costs time and core, but many applications would find it profitable overall. The feasibility of implementation, like the desirability of protecting against all restarts, must be evaluated for each system considered.

The constraints outlined above have been applied to HDC-701P Minuteman III coding now under way, with some difficulty arising from the instruction set employed. Had the requirements for restart protection been realized prior to hardware design, a compatible instruction set might have been generated, reducing both the cost and the difficulty of coding.

It should be noted that application of the necessary-and-sufficient algorithm provides analytic assurance of restart protection; no testing is required to verify safety in the event of restart. Similar work has been undertaken in the area of interruptibility by Bernstein[1] and others[2] but does not appear to have been reduced to practice yet. With the massive increase in the cost of software verification in recent projects, it is to be hoped that extension of analytic tools can reduce the verification burden.

---

1. A. J. Bernstein, "Analysis of Programs for Parallel Processing," IEEE Trans. on Electronic Computers, October 1966, pp. 757-763.

2. C. V. Ramamoorthy and M. J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," FJCC Proceedings, 1969, pp. 1-15.

APPENDIX C
PROVING PROGRAM CORRECTNESS

The purpose of a program correctness proof is to show in a rigorous mathematical fashion that a given program performs a desired function and halts. (The word "function" is used here in the mathematical sense.) A program is regarded as performing a mapping from the input variables to the output variables. An essential requirement for a correctness proof is a precise specification of the function to be computed. Without this specification the word "correctness" can have no meaning.

Current methods for showing that a program "works" are ad hoc and inherently inconclusive. The usual technique is to observe the program's performance on specially selected data, and to show that the program produces the correct answer in these cases. In general, this proves very little about the program's performance on other data, and all programmers can point to cases where mistakes were found after the programmer had convinced himself that the program was correct. A program correctness proof <u>proves</u> that a program will perform correctly on <u>all</u> data, even if, as is the usual case, there is an infinite or extremely large number of possible input data sets. The amount of effort required by a correctness proof appears to be a linear function of program size. As presently performed, program verification is a function of the number of possible execution paths, and the effort required to check all execution paths increases roughly exponentially with program size.

It is critically important in discussing correctness proofs to distinguish between the function computed and the algorithm (program) for computing that function. The correctness proof demonstrates that a given algorithm (program) computes a specified function. The word "correctness" must not be extrapolated beyond the domain to which it applies. A correctness proof does not show that the function being computed is the correct one for accomplishing some desired goal. A correctness proof can only be used to show that a program satisfies its specifications. The proof says nothing about the correctness of those specifications. For example, if one of the guidance equations is erroneous in the specifications for a missile flight program, a correctness proof will not detect the error.

C-1. THE SEMANTICS OF IDEALIZED PROGRAM STATEMENTS

At present, correctness proofs are formulated in terms of idealized algorithms consisting of three types of statements:

1) Assignment statements
2) Unconditional branch statements ("go to" statements)
3) Conditional branch statements

In discussing the semantics of these statements, it is more convenient to consider the flowchart form of an algorithm than the linear listing form. In proving that an algorithm is correct, we make assertions at convenient points in the flowchart. These assertions will specify relations which hold between program variables. The "meaning" of a statement is operationally defined as the change from the assertions which are true before the statement is executed into the assertions which are true after the statement is executed.

A sample assignment statement is shown in Figure C-1.* If S is the set of assertions true at point a, then

$$S \wedge (A = B + C)$$

is true at point b. A significant question is: What is the difference between an assignment operator and an equal sign? If a given program variable is assigned a value only once in a program, then there is no difference. If a variable is assigned a value more than once, it is convenient for proof purposes to regard each assignment as creating a new variable. For example, the sequence

$$A \leftarrow B$$

$$A \leftarrow A + I$$

is equivalent to the two equations

$$A_1 = B$$

$$A_2 = A_1 + I$$

Sequence subscripts will be used when a variable is assigned a value more than once. Assignment statements then become equations and their sequential ordering is rendered superfluous. If the variable is assigned values in a loop, a variable sequence subscript is used. This will be considered in more detail when loops are discussed.

A "go to" statement becomes merely a connecting line in a flowchart. For example, the program sequence

*Figures appear at the end of the appendix.

```
      Statement 1
      Statement 2
      Statement 3
L1:   Statement 4
      Statement 5
      Statement 6
      go to L1
```

becomes the flowchart of Figure C-2. The meaning of the statement "go to L1" is: if S is the set of assertions true before execution of the "go to" statement, then S is also true at the point L1.

Conditonal branch statements are assumed to be of binary type, as shown in Figure C-3. P is any predicate defined on the program variables; for example $(A + B > 0)$. If S is true at point a, $S \wedge P$ is true at point b and $S \wedge \overline{P}$ is true at point c.

C-2. METHODS FOR PROVING CORRECTNESS

In discussing the methods for proving correctness, it is desirable to distinguish three types of structures:

1) Straight-line segments
2) If P then X structures
3) Loops

Straight-line segments are treated by assigning sequence subscripts. Assignment statements then become equations. The assertions true at a given point are simply the collection of equations defined up to that point. Consider the sequence

$$I \leftarrow 1$$
$$A \leftarrow B + C$$
$$A \leftarrow A + I$$
$$E \leftarrow A$$
$$A \leftarrow I + 4$$

The flowchart with assertions attached is shown in Figure C-4. From the fixed set of equations it is easily shown by simple substitution that

$$I = 1$$
$$E = B + C + 1$$
$$A = 5$$

The number of assertions is seen to increase with each new assignment statement. However, substitution can be used to eliminate variables which have been written over, in the case A_1 and A_2.

"If P then X" structures represent loop-free branching, as shown in Figure C-5. The "X" may be thought of as any block of instructions, for example, an ALGOL begin-end block. P is any predicate over the program variables. $\underline{V}_0$ is the vector of variables defined at point a. The block X performs a mapping f from $\underline{V}_0$ to $\underline{V}_1$. If S is the set of assertions true at point a, then*

$$S \wedge (\underline{V}_1 = \underline{V}_0\,\overline{P} \vee (\underline{V}_1 = f(\underline{V}_0))P)$$

is true at point b. Thus, to maintain consistent sequence subscripts, the "do nothing" branch $\overline{P}$ is regarded as performing the operations $\underline{V}_1 \leftarrow \underline{V}_0$.

As a simple example, consider the program segment of Figure C-6. At point a the assertion is $X_1 = A$. At point B the assertions are:

$$X_1 = A$$

$$(X_2 = A) \wedge \overline{(X_1 < 0)} \vee (X_2 = -A) \wedge (X_1 < 0)$$

From these assertions it can be concluded that

$$X_2 = X = |A|$$

Loops can be classified according to the number of entry and exit points. We can further distinguish three classes of single-exit loops: test-first, test-middle, and test-last. The classification is according to where the exit test is in relation to the body of the loop. The test-first loop is shown in Figure C-7. The loop index k is added for purposes of analysis and proof. The dotted lines indicate it is artifactual. The assertions true at point a are a function of the loop index, and are written A(k). Similarly, since the variables in the predicate P are a function of k, the predicate is expressed as P(k). On exit from the loop it is known that $P(k) \wedge A(k-1)$ is true, and P(r) is false for $1 \leq r < k-1$.

*The usual precedence rule for logical operators is employed, i.e., negation, conjunction, disjunction, in order of strong to weak. Also, the conjunction symbol ($\wedge$) is occasionally replaced by concatenation of operands.

The test-last loop is shown in Figure C-8. On exit from this loop it is known that $A(k) \wedge P(k)$ is true and $P(r)$ is false for $1 \leq r < k$.

The test-middle loop is shown in Figure C-9. On the kth iteration of the loop, A(k) is the set of assertions true at point b. On exit, it is known that

$$A(k) \wedge B(k - 1) \wedge P(k)$$

is true. Since the section B is not performed until after the test, if exit occurs the variables assigned values in B have the values assigned on iteration k-1, but variables assigned values in section A have values assigned on iteration k. Variables used in the predicate may be a combination of values from section A and section B.

The single-entry, single-exit loop may then be modeled as in Figure C-10. S_0 is the set of assertions true on entry to the loop; S_e is the set of assertions true on exit from the loop. The loop thus can be regarded as a black box with a logical transfer function which converts S_0 into S_e.

If a loop has multiple exits, the set of assertions that are true when each exit is taken can be found just as for the case of a single exit. The black box model for such a loop is shown in Figure C-11. At point a, S_0 (the set of assertions true on entry to the loop) is true. At point b_i, $B_i(k)$ (the set of assertions true on taking the ith exit on the kth iteration of the loop) is true. From a black box standpoint, a loop with n exits is like an n-ary conditional branch. For such a loop, proof proceeds by determining the conditions $B_i(k)$. Once these are determined, the interior of the loop is no longer of interest. Its entire effect is expressed by the transformation for S_0 to the $B_i(k)$.

If a loop has more than one entry point, for purposes of analysis and proof it is possible to split it into several single-entry loops for which the entry is at the top. The technique will be illustrated for the example of Figure C-12. A test-last, single-exit loop is shown, but the technique can be applied to loops with multiple exits of any type. Entry 1 is at the top and entry 2 is somewhere in the body of the loop. A and B are the segments of the loop body above and below the second entry, respectively. For purposes of analysis and proof, the single loop can be split into the two loops of Figures C-13 and C-14. For entry 1, the loop of Figure C-12 is equivalent to the loop of Figure C-13. For entry 2, the loop of Figure C-12 is equivalent to the loop of Figure C-14. Thus for middle entries, the phase

of the loop is shifted to produce a top-entry loop. By successively splitting loops and shifting the phase, a multiple-entry loop can be converted into several single-entry loops with the entry at the top. These loops can then be treated by the techniques already developed.

From a logical standpoint, the criterion for correctness may be regarded as a theorem to be proved, with the statements of the program regarded as the given. Rules of inference, such as substitution of equals, induction, and properties of numbers, are used to show that the theorem is implied by the given.

C-3. EXAMPLES OF CORRECTNESS PROOFS

As a simple example, consider the following program segments:

```
        I ← 1
        A ← 0
L1:  if (I > N) then go to L2
        A ← A + B(I)
        I ← I + 1
        go to L1
L2:  halt
```

The variables I and A are assigned new values on each iteration of the loop. For purposes of analysis, these variables are given the variable subscript k. Proof statements are added to the program below and are shown in braces.

```
        I ← 1
        A ← 0
        {I_0 = 1, A_0 = 0}
L1:  if (I > N) then go to L2
        {P = (I_k-1 > N)}
```

$A \leftarrow A + B(I)$

$I \leftarrow I + 1$

$\{A_k = A_{k-1} + B(k)\}$

$\{I_k = I_{k-1} + 1\}$

go to L1

L2: halt

$$A = \sum_{r=1}^{N} B(r)$$

A nonrecursive definition of A_k is found by induction:

$$A_0 = 0$$

$$A_1 = 0 + B(1) = B(1)$$

$$A_2 = A_1 + B(2) = B(1) + B(2)$$

$$\vdots$$

$$A_k = \sum_{r=1}^{k} B(r)$$

Similarly for I,

$$I_0 = 1$$

$$I_1 = I_0 + 1$$

$$\vdots$$

$$I_k = k + 1$$

On exit, P(k) is true, i.e., $I_{k-1} > N$ or $k > N$.

But P(k-1) is false, i.e., $I_{k-2} \leq N$ or $k-1 \leq N$. If N is a nonnegative integer, $k - 1 = N$.

On exit,

$$A = A_{k-1} = \sum_{r=1}^{k-1} B(r)$$

or

$$A = \sum_{r=1}^{N} B(r)$$

One aspect which has not been mentioned is the halting problem. To show that a program will halt, it is necessary to show that for every loop in the program the exit test is satisfiable after some finite number of iterations. From the theory of recursive functions, it is known that the halting problem is in general undecidable; that is, there exist programs for which it is impossible to predict halting. For the vast majority of practical programs, however, showing that an exit test will eventually be satisfied is straightforward. In many cases loops are performed a fixed number of times, i. e, DO loops in FORTRAN. For loops with a potentially unbounded number of iterations, mathematical convergence theorems may be applicable. If the proof that an exit test is satisfiable is not apparent, it should be the responsibility of the programmer to show that an infinite loop cannot result. It should not be the burden of the verification analyst to show that an infinite loop can occur.

As a second example, a proof will be given of the logical correctness of a simple sorting routine. The flowchart is shown in Figure C-15. The proof will be presented in terms of the flowchart. However, it is also possible to present the same proof by inserting the proof statements at appropriate points in the program.

The program operates on a vector of N elements A(1), A(2), ..., A(N), rearranging the elements so that A(1) is the smallest and A(N) the largest, with values increasing monotonically from left to right. In rigorous form, the criteria for correctness are:

$$(\alpha \in \underline{A}_{in}) \Leftrightarrow (\alpha \in \underline{A}_{out})$$

$$A(k) \leq A(k+1) \text{ for } 1 \leq k \leq N-1$$

The first condition merely states that all the elements of the initial vector are present in the final vector. The second condition states that the values do not decrease in value from left to right.

For the proof, the inner loop is treated first. Indices are assigned to obtain the diagram of Figure C-16. For k = 0, it is true that

$$A(IMIN_k) \leq A(\ell) \text{ where } M \leq \ell \leq M + k$$

To show that this assertion is true for general k, we show that its validity for k implies validity for k + 1.

$$J_k = M + k$$

so
$$P_3(k) = \left\{A(M + k) < A(IMIN_{k-1})\right\}$$

and
$$P_3(k+1) = \left\{A(M + k + 1) < A(IMIN_k)\right\}$$

We assume that

$$A(IMIN_k) \leq A(\ell) \text{ for } M \leq \ell \leq M + k$$

The instructions between point a and point b in Figure C-15 form an <u>if</u> P <u>then</u> X structure, and so at point b it is true that

$$(IMIN_k = J_k)\, P_3 \vee (IMIN_k = IMIN_{k-1})\, \overline{P}_3$$

Consequently, on the $(k+1)^{st.}$ iteration,

$$IMIN_{k+1} = J_{k+1} = M + k + 1$$

if

$$A(M + k + 1) < A(IMIN_k)$$

and hence

$$A(IMIN_{k+1}) = A(M + k + 1)$$

If

$$A(IMIN_k) \leq A(M + k + 1)$$

then

$$IMIN_{k+1} = IMIN_k$$

and hence

$$A(IMIN_{k+1}) \leq A(M + k + 1)$$

Hence, on the $(k+1)^{st}$ iteration at point b,

$$A(IMIN_{k+1}) \leq A(M + k + 1)$$

and so

$$A(IMIN_{k+1}) \leq A(\ell) \text{ for } M \leq \ell \leq M + k + 1$$

Thus for the general k,

$$A(IMIN_k) \leq A(\ell) \text{ for } M \leq \ell \leq M + k$$

On exit, $J_k = N$ or $M + k = N$. So $k = N - M$, and hence

$$A(IMIN) \leq A(\ell)\ M \leq \ell \leq N$$

The inner loop, together with its initialization, is thus reduced to this single statement.

The entire sorting routine can now be represented as shown in Figure C-17. In closed form, $M_k = k + 1$. The last four statements in the body of the loop can be reduced to

$$A_k(IMIN_k) = A_{k-1}(k)$$

$$A_k(k) = A_{k-1}(IMIN_k)$$

After the first iteration,

$$A_1(s) \leq A_1(s+1) \text{ for } 1 \leq s \leq k \text{ for } k = 1$$

since

$$A(1) = A(IMIN_1) \leq A(\ell) \text{ for } 1 \leq \ell \leq N$$

and the swap of

$$A(MIN) \circlearrowleft A(M)$$

preserves the relation

$$(\alpha \epsilon \underline{A}_{in}) \Leftrightarrow (\alpha \epsilon \underline{A}_{k}) \text{ for } k = 1$$

Again we assume this is true for general k, and show it is true for k+1.

$$A_{k+1}(k+1) = A_k(IMIN_{k+1}) \leq A_k(\ell) \text{ for } k+1 \leq \ell \leq N$$

$$\therefore \quad A_{k+1}(k+1) \leq A_{k+1}(k+1+1)$$

and so

$$A_{k+1}(s) \leq A_{k+1}(s+1) \text{ for } 1 \leq s \leq k+1$$

Since

$$(\alpha \epsilon \underline{A}_{k}) \Leftrightarrow (\alpha \epsilon \underline{A}_{k+1})$$

then

$$(\alpha \epsilon \underline{A}_{in}) \Leftrightarrow (\alpha \epsilon \underline{A}_{k+1})$$

On exit,

$$M_k = N + 1$$

$$k + 1 = N = 1 \qquad \text{or} \qquad k = N$$

So

$$A_N(s) \leq A_N(s+1) \text{ for } 1 \leq s \leq N$$

and hence

$$\underline{A}_{out} = \underline{A}_N$$

$$(\alpha \epsilon \underline{A}_{in}) \Leftrightarrow (\alpha \epsilon \underline{A}_{out})$$

While the examples of this discussion are relatively simple, it is important to point out that the same principles will apply to programs of arbitrary size. London ("Proving Programs Correct: Some Techniques and Examples," BIT, Vol. 10, 1970) reports that programs of realistic size have been proven correct using a proof technique which is actually less systematic than the method presented here.

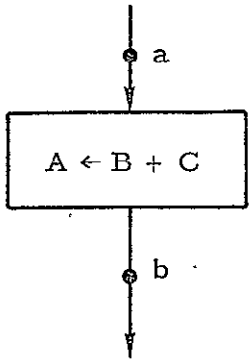


Figure C-1. Assignment Statement

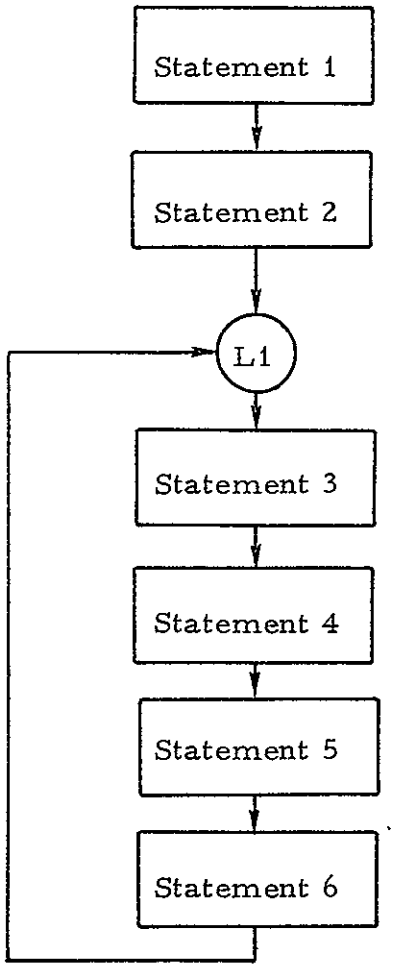


Figure C-2. "Go To" Statements

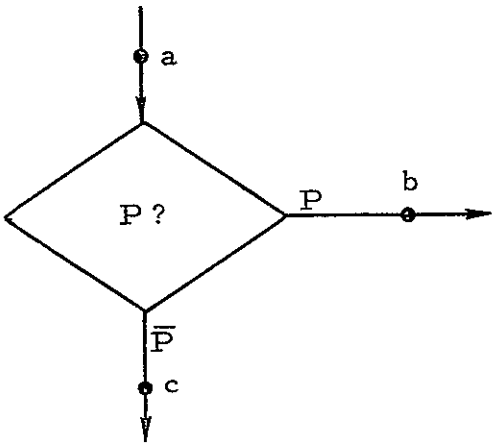


Figure C-3. Conditional Branch Statement

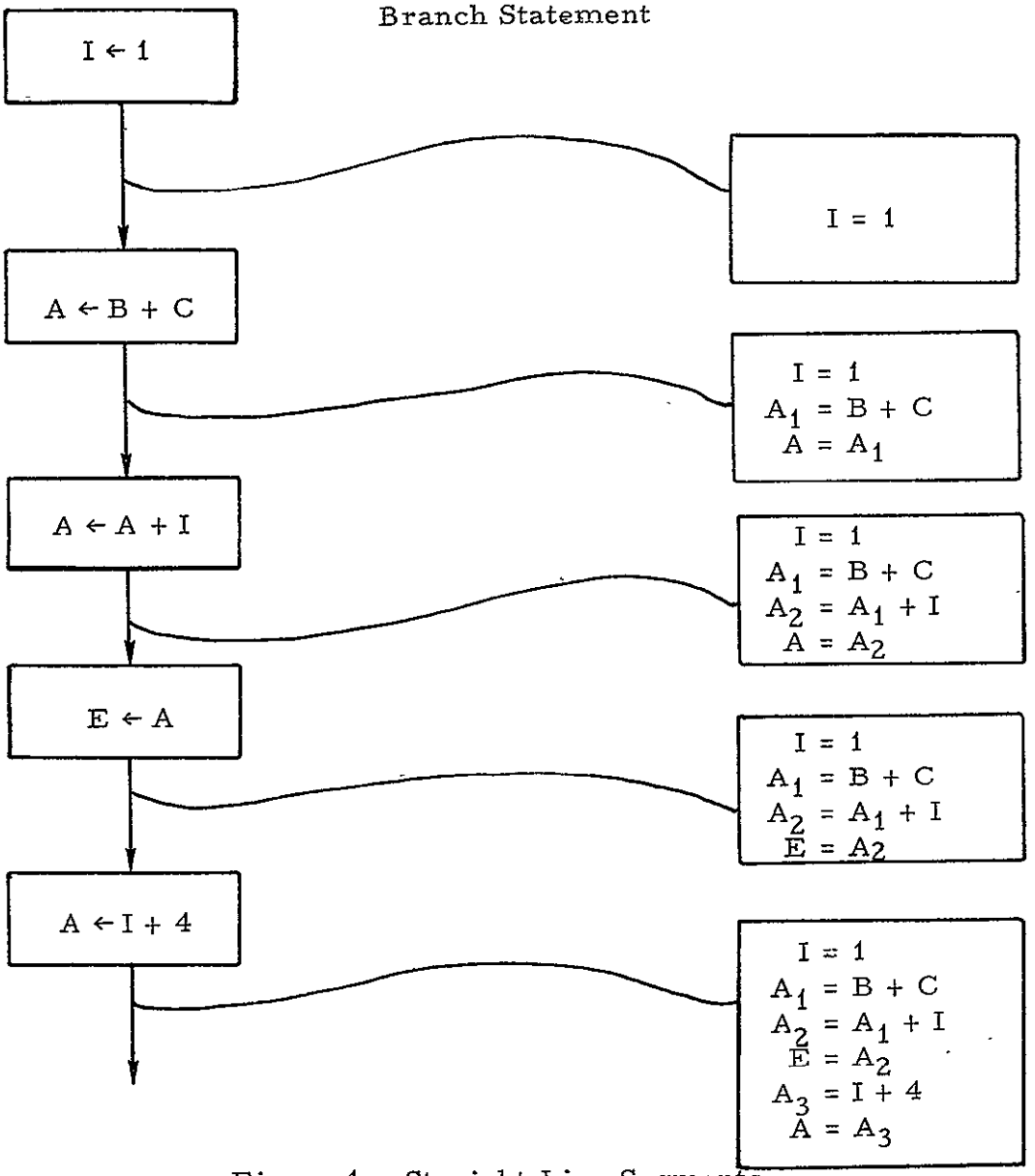


Figure 4. Straight-Line Segments

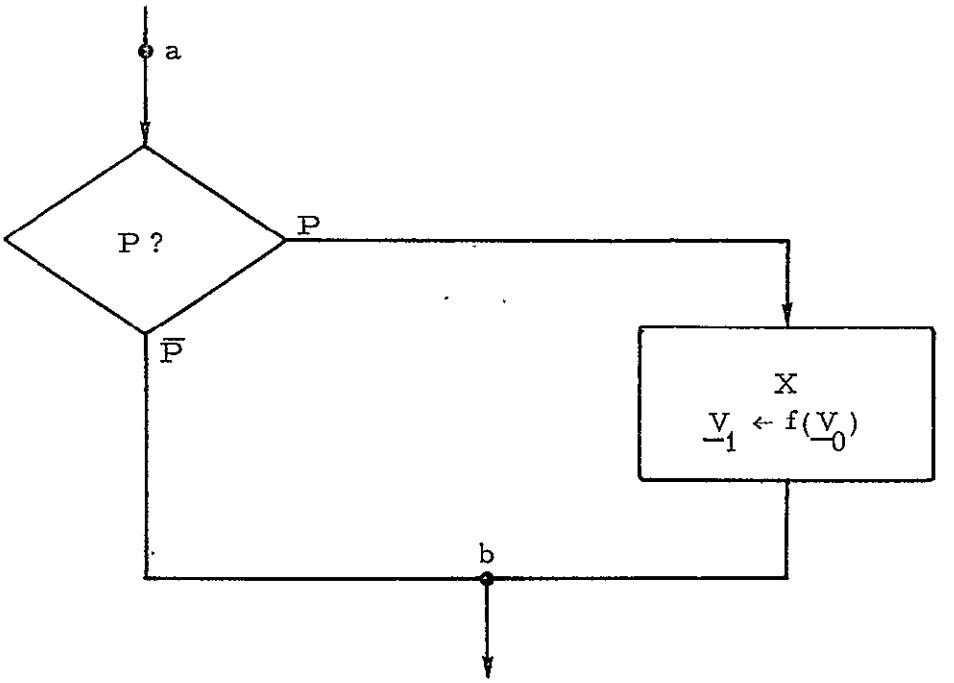


Figure C-5. "If P then X" Structures

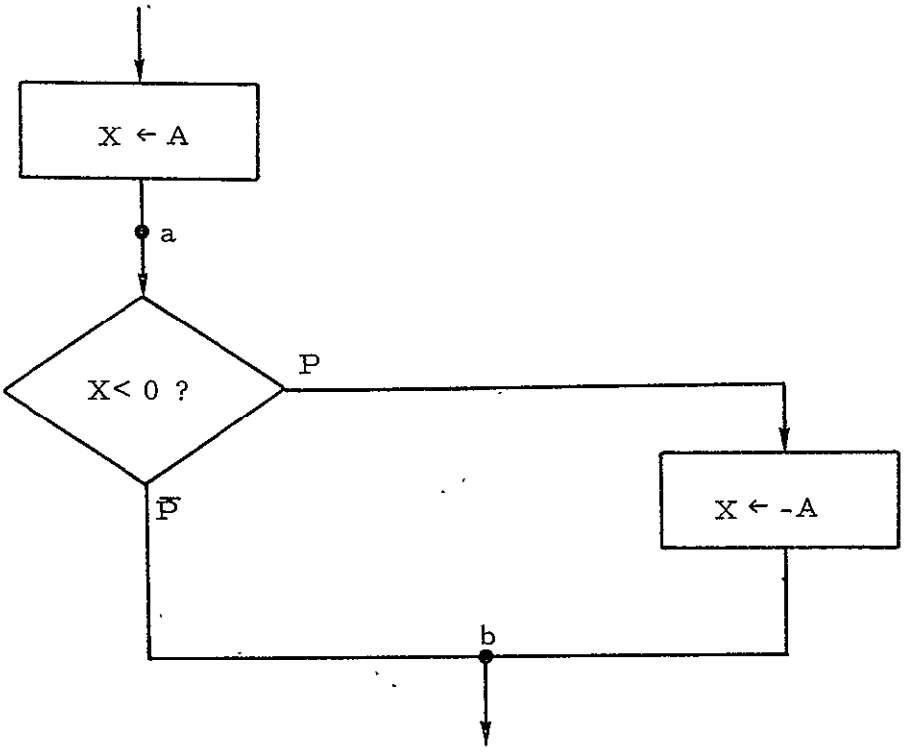


Figure C-6. Example of "If P then X"

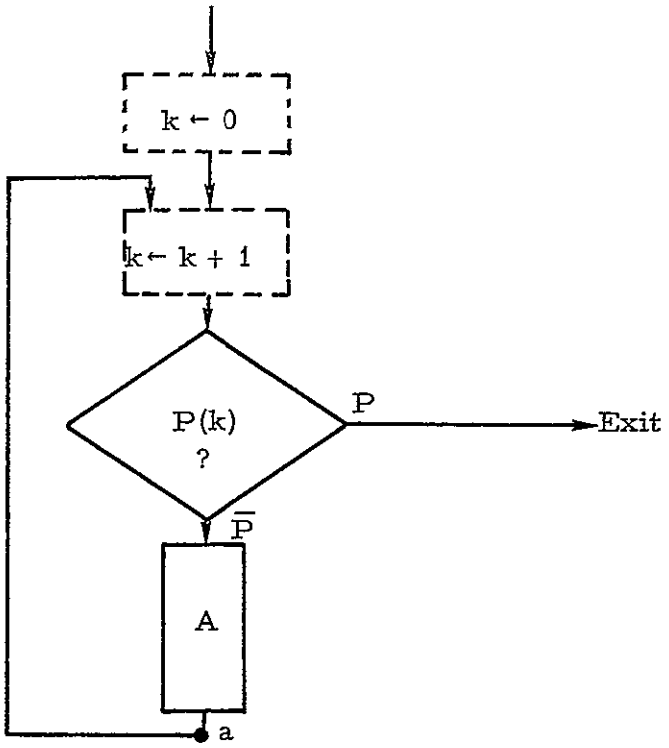


Figure C-7. Test-First Loop

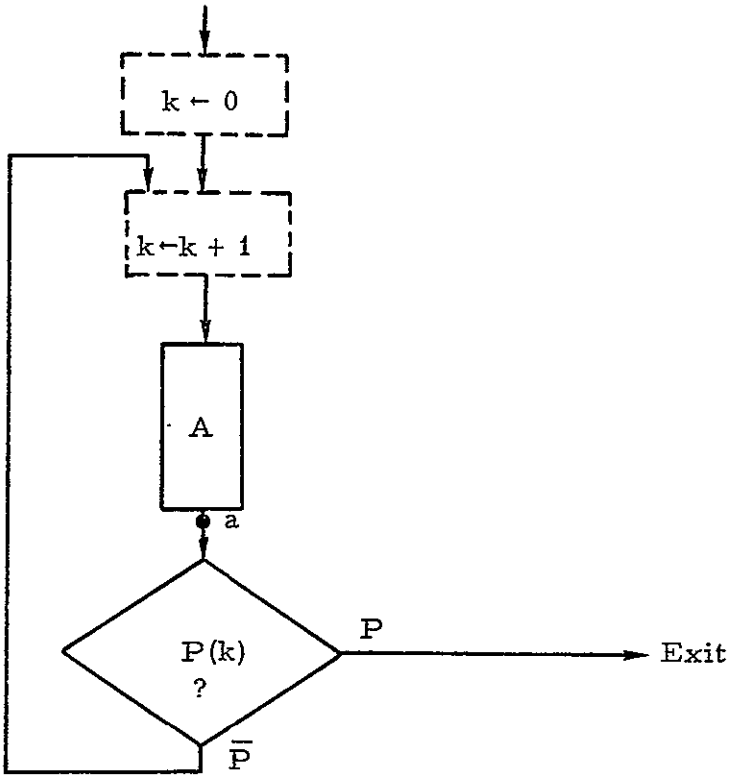


Figure C-8. Test-Last Loop

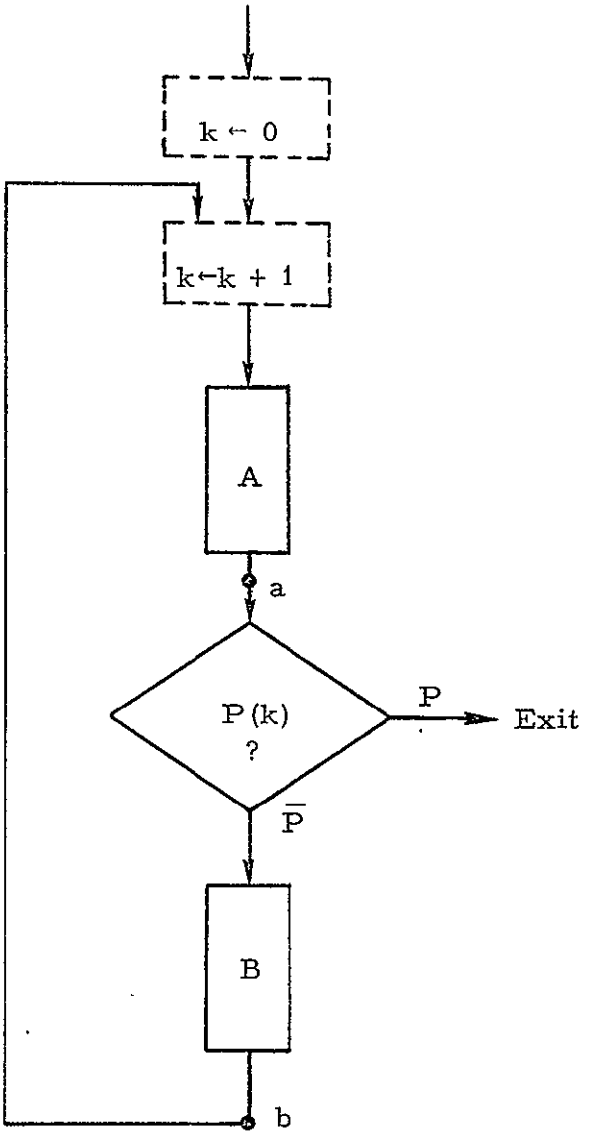


Figure C-9. Test-Middle Loop

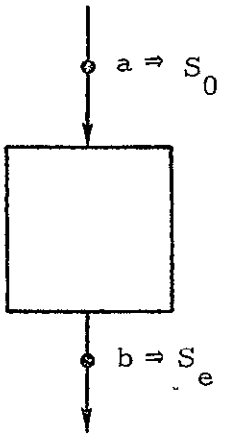


Figure C-10. Single-Entry, Single-Exit Loop

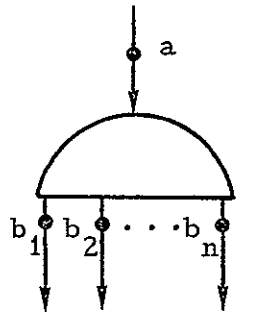


Figure C-11. Multiple-Exit Loop

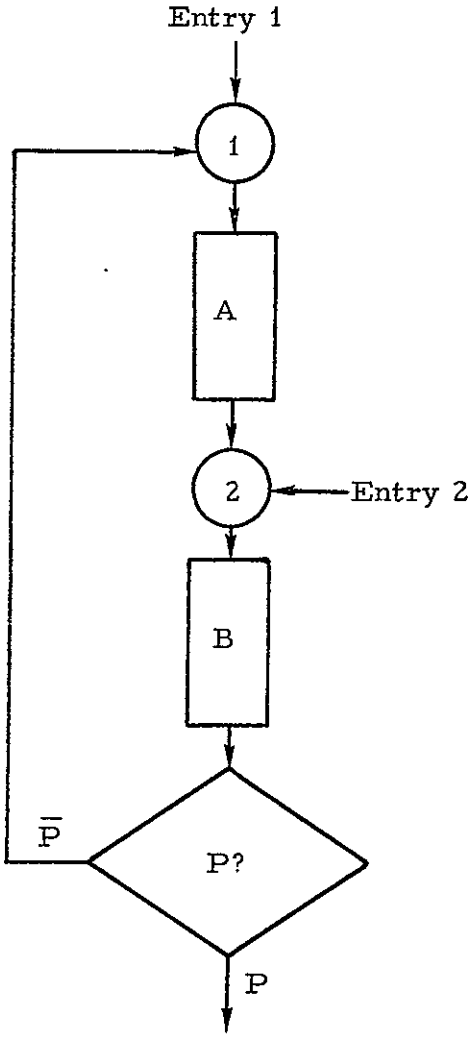


Figure C-12. Composite Loop Example

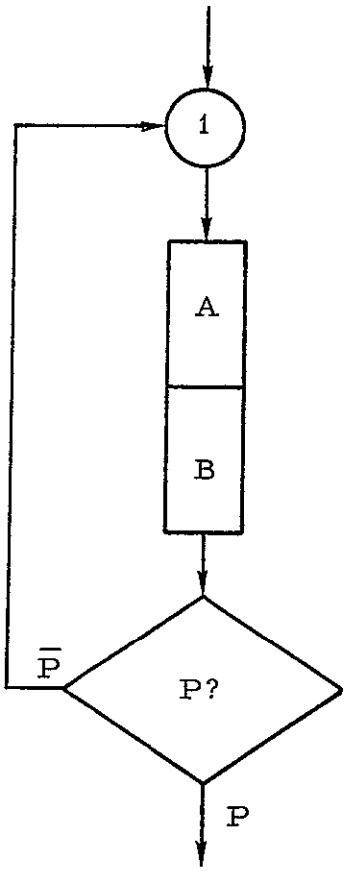


Figure C-13. Entry 1 Equivalent

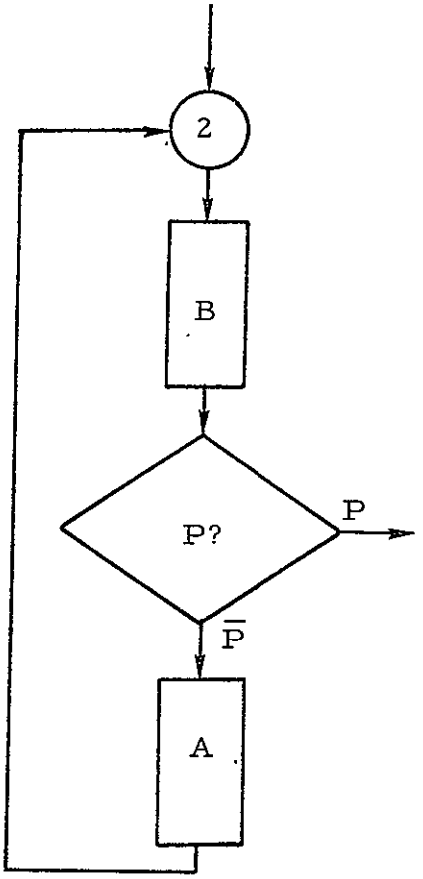


Figure C-14. Entry 2 Equivalent

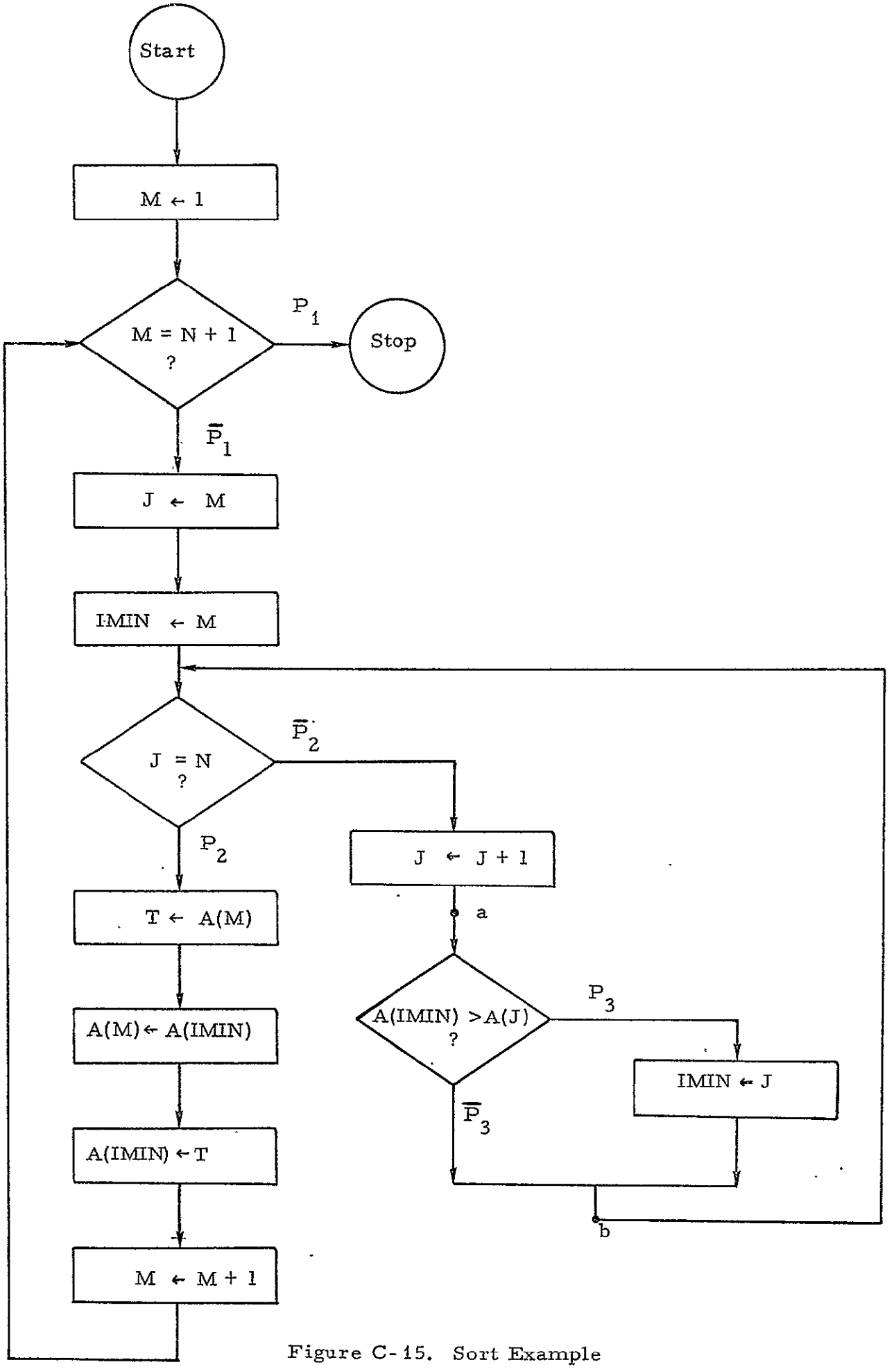


Figure C-15. Sort Example

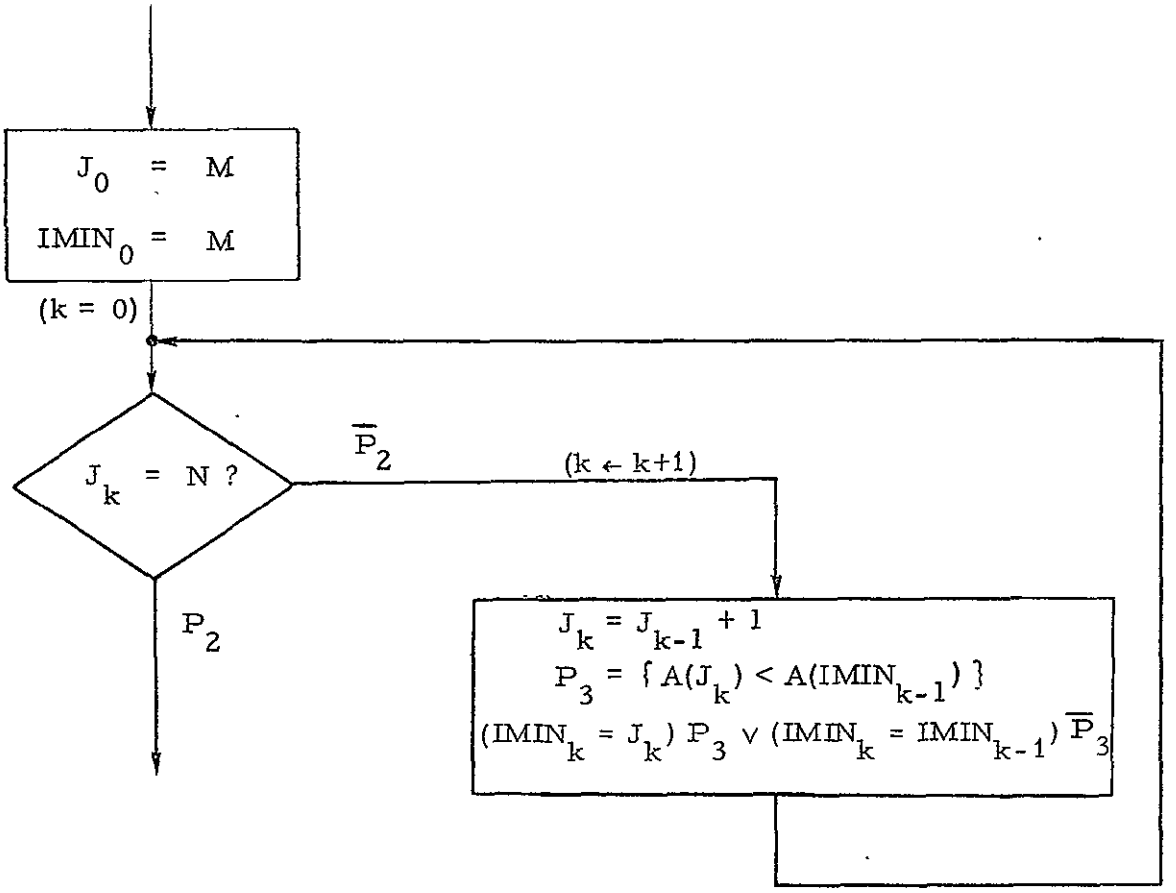


Figure C-16. Sort Example Inner Loop

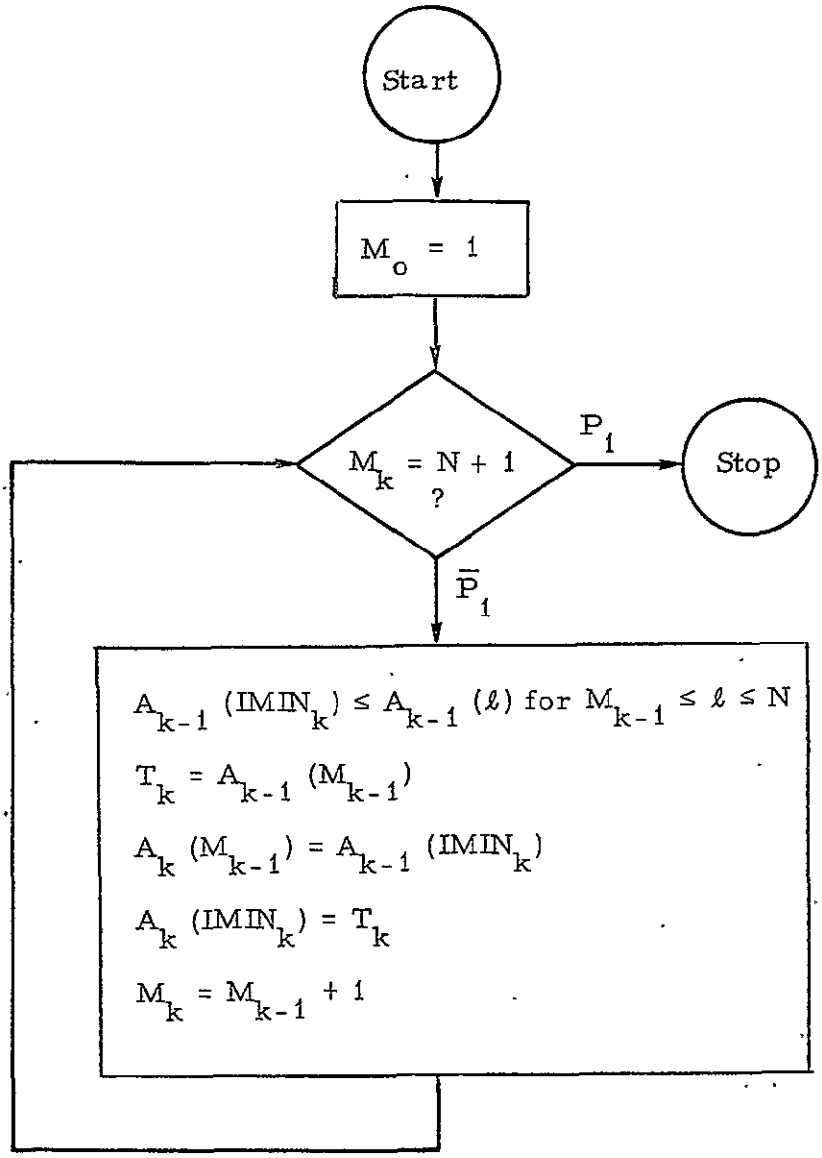


Figure C-17. Sort Example Outer Loop

255 w fifth street, san pedro, california 90731 • (213) 831-0611
1075 camino del rio south, san diego, california 92110 • (714) 291-4240
6231 leesburg pike, falls church, virginia 22044 • (703) 534-7087