R.N.F.

Computer Systems

G.Bell, S.H.Fuller, and D.Siewiorek, Editors

Architecture of the IBM System/370

Richard P. Case and Andris Padegs
IBM Corporation

This paper discusses the design considerations for the architectural extensions that distinguish System/370 from System/360. It comments on some experiences with the original objectives for System/360 and on the efforts to achieve them, and it describes the reasons and objectives for extending the architecture. It covers virtual storage, program control, data-manipulation instructions, timing facilities, multiprocessing, debugging and monitoring, error handling, and input/output operations. A final section tabulates some of the important parameters of the various IBM machines which implement the architecture.

Key Words and Phrases: computer systems, architecture, instruction sets, virtual storage, error handling

CR Categories: 6.0, 6.21

Introduction

The years since the introduction of System/360 in 1964 have produced very substantial changes in most aspects of the design, manufacture, and use of information-processing systems. The hardware technology for realizing logic functions has evolved from semi-integrated circuit modules with single devices per chip to hundreds or thousands of circuits on a single silicon chip. The technology for high-speed storage has changed from magnetic cores to dense arrays of transistors on silicon chips. The growth in size and function of systems software has surprised even the practitioners. It is not surprising, therefore, to discover that extensions and refinements to the architecture[1] of System/360 were found to be necessary.

Authors' addresses: Richard P. Case, IBM Corporation, 1000 Westchester Ave., White Plains, N.Y. 10604. Andris Padegs, IBM Corporation, P.O.Box 390, Poughkeepsie, N.Y. 12602.

This paper reviews the motivation for extending the System/360 architecture and describes the design considerations associated with the extensions adopted for System/370.[2] It comments on some experiences with the original objectives and concepts of System/360. Finally, it summarizes the characteristics of IBM machines implementing the System/360 and the System/370 architectures [1,2,5,6,15,17].

Experience with System/360

At the time the major decisions were made on the System/370 architecture, a significant amount of experience was available with the initial implementations of System/360. The major conclusions from this experience were:

Compatibility: Compatibility really worked. It was in fact possible to transfer programs routinely from one model to another and expect them to produce the same results. Operational evidence was available that architecture and implementation could be separated; one need not imply the other.

Compatibility also helped reduce development expense. The original plan called for verifying each element of software on each model. Because of the growing confidence that programs which ran on one model would also run on other models, it was possible to significantly reduce the amount of cross-verification to be performed.

Implementation of a whole line of computers according to a common architecture did not take an undue amount of effort. It did, however, require unusual attention to detail and some new procedures, which are described in the Architecture Control Procedure section.

Performance Range: A greater performance range must be planned for. The original System/360 announcement included processors with a performance range of about 25 to 1. Six years later this had increased to about 200 to 1, and plans were being made for even further extensions.

Main Storage: It was obviously necessary to plan for main-storage sizes of more than 2^{24} bytes. The technological improvements in main storage which reduced the relative cost had happened at a rate greater than was expected. The result was that serious thought had to be given to the planned replacement of 24-bit addressing.

The extension of the address size proved to be more difficult than first thought. Our experience in this respect agrees with that of Bell and Strecker [4], who say: "There is only one mistake . . . that is difficult to recover from – not providing enough address bits . . . ."

[1] The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation.

[2] This paper is not the definitive reference work for the specification of the features and functions discussed. For the official, and maintained, description, refer to the *IBM System/370 Principles of Operation*, form GA22-7000, which is available through local IBM branch offices.

The basic addressing mechanism of System/360 had anticipated the eventual need and was well suited to the extension, since it depended on base registers that were already 32 bits wide. The interruption mechanism and the I/O control formats, however, did not have the required extensibility. (We knew in 1962 that this was the case, but the immediate cost and performance consequences outweighed the need to meet the eventual long-term requirements.) More importantly, the operating systems and compiler-produced application programs had used the extra bits in address words for control purposes and hence required extensive modifications.

System/370 does not extend the address size but incorporates some of the steps necessary to do it.

Operating Systems: Machine architecture must be developed in conjunction with changes and extensions to existing operating systems. Whereas the original System/360 architecture was developed to provide a good basis on which a completely new operating system could be built, extensions to that architecture have to consider the specific usages and capabilities of the available operating systems.

Architecture Control: The design and control of system architecture must be an ongoing function that can never be considered complete. We found ourselves well into the 1970s making changes in the architecture of System/360 to remove ambiguities and, in some cases, to adjust the function provided.

Objectives of System/370

Motivation: The motivation to extend the System/360 architecture for the new series of machines came from two main sources:

(1) The experience with the System/360 architecture in writing application programs, in designing and using operating systems, and in debugging and maintaining both software and hardware had identified a number of bottlenecks and limitations in the efficiency of system use and had pointed out areas where additional machine functions were desirable.

(2) The general lowering of the cost of technology for main storage and logic circuitry in relation to the overall system cost made it possible economically to include functions that did not appear justified in the original System/360 architecture.

Specific Objectives: The following were the specific objectives of the System/370 architecture:

(1) Improving the level of detail, precision, and predictability of the System/360 architecture. These improvements were made primarily in the areas of interruptions, system control, and the order of storage references. They were motivated largely by reliability and serviceability considerations.

(2) Adding new instructions to enhance the performance of frequent functions in application programs. A total of 17 new unprivileged instructions were introduced in the System/370 architecture.

(3) Extending the architecture to improve system reliability, availability, and serviceability. Extensions were included to assist diagnostics and recovery by software after a hardware failure (machine-check extensions), to assist in debugging software (program-event recording, monitoring, status storing), and to facilitate formation of multiprocessing systems with multiple CPUs sharing common main storage.

(4) Adding new facilities to enhance the performance and function of the operating system and to introduce uniform machine-implemented protocols in the system. Dynamic address translation, timing facilities, and a number of privileged instructions were the main extensions provided for this purpose.

Constraints on System/370

The System/370 architecture was developed subject to the following main constraints:

(1) Within the limitations described in the *System/370 Principles of Operation,* the architecture must be upward compatible with System/360 architecture as far as user programs are concerned; that is, user

Table I. Architectural extensions incorporated in System/370.

Facility	Instructions Unpriv.	Instructions Priv.
Virtual storage		
Dynamic address translation	-	2
Reference and change recording	-	1
Channel indirect data addressing	-	-
Program control and interruptions		
Control registers	-	2
Extended control	-	-
System-mask handling	-	2
PSW-key handling	-	2
Restart interruption	-	-
Extended masking	-	-
Data-manipulation instructions		
General instructions	7	-
Decimal instructions	1	-
Floating-point instructions	7	-
Byte-oriented operands	-	-
Timing facilities		
Time-of-day clock	1	1
Clock comparator	-	2
CPU timer	-	2
Multisystem operation		
Synchronization and serialization	-	-
Prefixing	-	2
Interprocessor signaling	-	2
Debugging and monitoring		
Program-event recording	-	-
Monitoring	1	-
Status storing	-	-
Machine-error handling		
Resets	-	-
Error reporting	-	-
Logout	-	-
Command retry	-	-
Storage validation	-	-
Machine identification	-	2
Input/Output		
Block multiplexing	-	-
Control	-	3
Data-rate improvement	-	-
	17	23

programs written for System/360 must run efficiently on System/370 models with no modification to these programs. These limitations are that the systems have the same or equivalent facilities and that the programs have no time dependence, use only model-independent functions defined in the Principles of Operation, and not use unassigned formats and operation codes. These limitations essentially mean that compatibility applies only to valid programs.

(2) It must be possible to run certain System/360 operating systems unmodified on System/370 models. Even though such operating systems could not fully benefit from the new functions available in System/370, and new support was planned, the ability to execute them was needed for the transition period.

(3) It must be possible to attach and operate most types of System/360 I/O devices on System/370.

(4) The System/370 architecture must preserve and extend the open-endedness and generality of design characteristic of the System/360 architecture.

Summary of Architectural Extensions

Table I lists the major categories of architectural extensions that have been added to the System/360[3] architecture to form the System/370[4] architecture, including those that were originally introduced on the System/360 Model 85. The extensions are grouped in terms of architectural facilities, which are mechanisms provided in the machine for performing a specific function. The table also lists the number of new instructions associated with the facility. Note that many of the new facilities have no new instructions associated with them. Table II lists all new instructions, which total 40.

Additionally, in a number of areas the System/360 architecture was made more specific and predictable within the freedom permitted by the original definition. The following are two examples:

(1) The result of a decimal-arithmetic operation is made predictable when an invalid sign code is encountered. This is a common error in source data, and the change permits correction and resumption of the operation.

(2) The priority of recognizing program-interruption conditions is specified to achieve repeatability and to make debugging easier.

[3] The System/360 Model 20 is not discussed in the referenced papers nor in this paper, as some of its architectural features are so specialized that it is not convenient to discuss them in the same context.

[4] This paper covers only those facilities that are described in *System/370 Principles of Operation.* It does not discuss certain extensions that were made available only on System/360 Models 44 and 67; nor does it describe the following special facilities that are available only on some models: virtual-machine assist (hardware assist for VM/370), extended control-program support (hardware assist for OS/VS1 and for VM/370), APL assist, OS/DOS compatibility, the assist for optical character recognition, emulators for other machines, as well as the System/370 extended facility and recovery extensions first made available on the IBM 3033 Processor Complex.

Compatibility with System/360

Methods of Achieving Compatibility: Major emphasis in the design of the System/370 architecture was placed on defining all changes and extensions so that a valid System/360 program, executed on a System/370 machine, would obtain the same results as specified in the *System/360 Principles of Operation.* [12] This compatibility was achieved by four devices:

Restriction: Narrowing System/370 to a more specific operation in areas where the System/360 definition allowed unpredictable results. This approach applied to the extensions in machine-check interruptions, as well as to a number of minor improvements.

Checking: Allowing new functions to be invoked only by a program that would have been considered invalid in System/360; that is, letting a program observe a change or extension to System/360 operation only when it uses an operation code or specifies a value for a bit in the program-status word or in an address that in System/360 is checked for validity and results in a program exception. This device was used for the large majority of extensions, including the byte-oriented-operand feature and virtually all new instructions.

This approach was used also to ensure that all subsequently introduced extensions, such as dynamic address translation and program-event recording, are compatible with the System/370 architecture as initially announced. An exception was that the unused positions in the 16 control registers introduced at the original System/370 announcement were not checked for zeros but instead were reserved for future extensions by an explicit warning in the Principles of Operation. This safeguard was chosen because only privileged programs can load and store control registers, because checking scattered bit positions in the 16 registers is expensive and time-consuming, and because even greater cost would have been required for a predictable ending of an invalid loading operation.

Mode Control: Defining mode-control and mask bits in control registers such that the reset state specifies an operation compatible with System/360. The external, channel, and machine-check masks, as well as a number of other controls, were defined this way.

Manual Switches: Introducing a manual switch for setting up a mode where the machine stops on encountering a deviation from System/360 operation. This approach was taken to handle CPU and channel diagnostic logouts. In System/360, the logout area starts with location 128 and, while no limit is set on its size, its extent is smaller than that on a comparable System/370 model. Since such a logout on a System/370 machine may overlay a program or data which assumes System/360 logout, stopping avoids continuation with invalid information. It was assumed that the stop-on-logout mode would be selected only for the rare situations when the machine is operated without the correct error-recovery program.

Incompatibilities: The extensions introduced for System/370 do not meet the compatibility objectives in the following five cases. In each case a program may exist that meets the System/360 validity requirements but does not obtain the same results on System/370. These incompatibilities, however, are confined to programs that are either executed in the supervisor mode or are components of an operating system, and they were deemed justified, considering both the alternative solutions and the likelihood and difficulty of operational problems. The five incompatibilities are reviewed here in some detail to emphasize the kind of careful attention that compatibility requires.

Use of USASCII-8 Bit for Control of EC Mode: System/360 anticipated the adoption of a proposal for a "Decimal ASCII" in punched cards and of a technique for expanding the seven-bit standard to eight bits. This data representation is referred to as USASCII-8 in the System/360 manuals. Both the card code and the particular expansion technique have since been rejected as a national standard.

System/360 provides for USASCII-8 by a mode under control of PSW bit 12. When bit 12 of the System/360 PSW is one, codes preferred for USASCII-8 are generated for decimal results. When PSW bit 12 is zero, the codes preferred for EBCDIC are generated.

In System/370, the USASCII-8 mode and the associated meaning of PSW bit 12 are removed, and all instructions whose execution in System/360 depends on the setting of PSW bit 12 are executed to yield the EBCDIC codes. PSW bit 12 is used instead to control the format of the PSW and of the information stored on an interruption.

This incompatibility affects only those System/360 programs that specify the USASCII-8 mode. Since the anticipated standard was never adopted, it is highly unlikely that any production programs ever used it. In fact, we are not aware of any instance of its use.

The alternative for System/370 was to assign a control-register bit for controlling the PSW format. Such a definition would not have permitted changing at the same time both the mode and the PSW contents which the mode controls, and it would have precluded program control of the PSW format on initial program loading.

Table II. New instructions incorporated in System/370.

Name	Mnemonic	Type		Op Code
ADD NORMALIZED (extended)	AXR	RR	Unpriv.	36
CLEAR I/O	CLRIO	S	Priv.	9D01
COMPARE AND SWAP	CS	RS	Unpriv.	BA
COMPARE DOUBLE AND SWAP	CDS	RS	Unpriv.	BB
COMPARE LOGICAL CHARACTERS UNDER MASK	CLM	RS	Unpriv.	BD
COMPARE LOGICAL LONG	CLCL	RR	Unpriv.	OF
HALT DEVICE	HDV	S	Priv.	9E01
INSERT CHARACTERS UNDER MASK	ICM	RS	Unpriv.	BF
INSERT PSW KEY	IPK	S	Priv.	B20B
LOAD CONTROL	LCTL	RS	Priv.	B7
LOAD REAL ADDRESS	LRA	RX	Priv.	B1
LOAD ROUNDED (extended to long)	LRDR	RR	Unpriv.	25
LOAD ROUNDED (long to short)	LRER	RR	Unpriv.	35
MONITOR CALL	MC	SI	Unpriv.	AF
MOVE LONG	MVCL	RR	Unpriv.	OE
MULTIPLY (extended)	MXR	RR	Unpriv.	26
MULTIPLY (long to extended)	MXDR	RR	Unpriv.	27
MULTIPLY (long to extended)	MXD	RX	Unpriv.	67
PURGE TLB	PTLB	S	Priv.	B20D
RESET REFERENCE BIT	RRB	S	Priv.	B213
SET CLOCK	SCK	S	Priv.	B204
SET CLOCK COMPARATOR	SCKC	S	Priv.	B206
SET CPU TIMER	SPT	S	Priv.	B208
SET PREFIX	SPX	S	Priv.	B210
SET PSW KEY FROM ADDRESS	SPKA	S	Priv.	B20A
SHIFT AND ROUND DECIMAL	SRP	SS	Unpriv.	FO
SIGNAL PROCESSOR	SIGP	RS	Priv.	AE
START I/O FAST RELEASE	SIOF	S	Priv.	9C01
STORE CHANNEL ID	STIDC	S	Priv.	B203
STORE CHARACTERS UNDER MASK	STCM	RS	Unpriv.	BE
STORE CLOCK	STCK	S	Unpriv.	B205
STORE CLOCK COMPARATOR	STCKC	S	Priv.	B207
STORE CONTROL	STCTL	RS	Priv.	B6
STORE CPU ADDRESS	STAP	S	Priv.	B212
STORE CPU ID	STIDP	S	Priv.	B202
STORE CPU TIMER	STPT	S	Priv.	B209
STORE PREFIX	STPX	S	Priv.	B211
STORE THEN AND SYSTEM MASK	STNSM	SI	Priv.	AC
STORE THEN OR SYSTEM MASK	STOSM	SI	Priv.	AD
SUBTRACT NORMALIZED (extended)	SXR	RR	Unpriv.	37

Clearing Storage on Power Off: In System/360, main storage originally was implemented with magnetic cores, and the architecture specifies that the storage preserve its contents when the power is turned off and on, provided that the CPU is in the stopped state. In System/370, with solid-logic technology, the power-on sequence normally clears storage to zeros. Incompatibility exists to the extent that a program that depends on information stored before power was turned off (in order to dump storage contents, for example) will not operate on System/370.

This change was mandated by the change from core to solid-logic technology, and it had minor impact on compatibility.

A "power warning" interruption is available as a feature on some models of System/370 which, in conjunction with equipment that monitors line voltage, signals when loss of power is imminent. The timing of the signal should be such that the operating system can transfer the contents of main-storage (or at least critical sections) to a permanent medium before the system stops operating. This usually requires some type of stored energy supply.

Operation Code for HALT DEVICE: The first eight bits of the operation code assigned to the new System/370 instruction HALT DEVICE are the same as those originally assigned to HALT I/O, the distinction between the two being specified by bit 15. In System/360, bit 15 is ignored, and HALT I/O is performed in both cases. Incompatibility exists to the extent that a HALT I/O instruction of a System/360 program is executed on a System/370 model as HALT DEVICE if bit 15 happens to be one.

This choice of the operation code was made to facilitate the attachment of the IBM 2880 Block Multiplexer Channel, which implements HALT DEVICE, to the Model 85 CPU, the design of which did not initially provide for this new instruction. The likelihood of a problem is minimal, because:

(1) Normally bit 15 is zero, since it is set to zero by IBM compilers and assemblers.

(2) In many cases the function performed by HALT DEVICE may be substituted for and may even be preferable to that performed by HALT I/O.

(3) The occurrence of the HALT I/O instruction is infrequent.

Command Retry: Most System/370 channels provide the command-retry facility, whereby the channel, in response to a signal from the device, re-executes a channel command. This re-execution is usually invoked when the device or control unit detects a malfunction. The following is a list of some of the effects of command retry:

(1) An immediate command specifying no chaining may result in condition code 0 being set rather than condition code 1.

(2) Multiple interruptions may be generated for a single channel-command word (CCW) with the program-controlled interruption flag.

(3) Since CCWs may be refetched, programs which dynamically modify CCWs may be affected.

(4) The residual count in the channel-status word reflects only the last execution of the command and does not necessarily reflect the maximum storage used in previous executions.

These potential difficulties were not deemed to be serious enough to warrant the hardware and software cost of placing command retry under mode control. No problem exists with the compatibility of I/O devices announced prior to System/370, as they do not signal for command retry.

Channel Prefetching: In System/360, on an output operation the channel may prefetch and buffer as many as 16 bytes; similarly, with data chaining specified, the channel may fetch the new CCW when up to 16 bytes remain to be transferred under control of the current CCW. In System/370, the restriction of 16 bytes is removed.

This incompatibility may affect programs that change data or command words during the execution of the operation. The change was needed for performance reasons and, as with command retry, was not deemed to warrant a mode control.

Extendability and Generality

The compatible evolution of the System/360 architecture into the System/370 architecture was made possible largely by judicious reservation in System/360 of unassigned formats and operation codes. The System/370 architecture maintains and extends the principle of frugal and controlled allocation of architecture resources, so that System/370 can be extended in the future to meet new requirements. The following are some examples where provision is made for future extension:

(1) Main-storage-address fields in the new PSW format, control registers, and the permanently allocated storage locations were assigned 32 bit positions, should they be needed for address expansion.

(2) The new EC-mode PSW format was defined to provide space for additional control bits.

(3) The control registers provide a general method of handling control information that is not contained in the PSW, and provide space for new facilities and for an expansion of the present facilities.

(4) The time-of-day clock format contains 12 unassigned low-order bit positions, which could be used for higher resolution.

(5) A new instruction format was introduced for instructions that need a single operand address. The unused eight-bit field in this format is made a part of the operation code, thus expanding the number of available operation codes by 255.

Architecture Control Procedure

Beginning with the development of System/360, and continuing to the present day, IBM has gradually adopted a process for the specification and control of architecture. This process has been largely successful in maintaining compatibility among many and varied machines developed in several laboratories around the world. The following are some important attributes of this process.

Specification: There is but one specification of the architecture. It tells IBM machine designers the functions the machine must provide, and it describes to IBM programmers how the machine operates. The same specification, called the Principles of Operation, [12] is made available outside IBM and is the only authoritative specification that describes the architecture.

The architecture specification covers all functions of the machine that are observable by a program. It either specifies the action the machine performs or states that the action is unpredictable. The latter applies to the detailed functions for which neither frequency of occurrence nor usefulness of results warrants identical action in all models or at all times. Normally the specification of unpredictable operation is a considered architectural choice, since the architecture specification must anticipate future implementations and the potential cost of providing specific results of marginal value. Occasionally, it is introduced into the definition because the specific detailed function is overlooked in the initial stages of the architecture resolution process or because the designs of the machines initially implementing the architecture mandate different operations.

All machine implementations are strictly monitored for compliance with the architecture specification. Affirmation of compliance with the architecture is a part of the internal IBM procedure for product-development control, and actual compliance is verified by formal and informal compliance audits and reviews of machine specifications. Deviations from the architecture must be corrected. In the rare cases when the cost to change the design or to retrofit installed machines is excessive in relation to the practical value of the compliance on that machine, deviations are permitted. Any deviation that is likely to affect the execution of a program is published in the IBM System Library manual for the machine.

Most machines have a few deviations, covering such aspects as the precise meaning of the test light on the operator-control panel, the indication of access exceptions for an unused part of an instruction, or the precise instant during execution of the WRITE DIRECT instruction when serialization is performed. A deviation by one implementation does not necessarily lead to a specification of unpredictability, as compliance with the definition may be essential for other applications, and the specific definition better conveys the intended structure, making the architecture simpler and easier to understand.

Development Procedure The architecture definition starts out with a proposal for extending or improving the function of the machine in a specific area. Extensions to the architecture normally are adopted as part of the development of a new machine or set of machines, and the process includes a number of steps:

(1) Preliminary Review: Depending on the scope of the extension, the cost and performance implications of new ideas may be evaluated in various studies and reviews among the architects and the machine and software designers. A number of iterations of such reviews and architecture definitions may take place.

(2) Resolution Meetings: After an architecture definition has been produced and reviewed by all interested areas, the adoption of the definition is placed as an item on the agenda of an architecture resolution meeting. These are periodic meetings where all interested and affected groups are represented by people with authority to commit their projects. Depending on the need, these meetings may take place monthly, weekly, or even more frequently. A proposal may be adopted or rejected at the resolution meeting, or concerns may be identified that require further study. A proposal that is adopted at an architecture resolution meeting becomes part of the architecture specification.

(3) Resolution Conferences: In order to set the direction for a new product line, stop debate on some issue, or resolve all loose ends, a resolution conference is called. Such conferences may take place a few times during the development of a product. They differ from regular resolution meetings in that participation is wider, higher level of management is involved, and more use is made of executive decision making.

(4) Interpretation: The architecture specification occasionally leaves out some aspect of the operation, or the wording may not be quite clear. Implementers are instructed to question the architecture on any doubtful point rather than make assumptions. Most questions are raised and answered by telephone, and the architect then periodically documents the questions and the answers for review by all implementers. These architecture interpretations supplement the original definition and are eventually integrated into the definition. Some questions demand further study or require action at one of the resolution meetings. Although the need for interpretation of the architecture normally diminishes after the initial implementation of the definition, some valid questions are raised and changes in the wording made years later. Continual maintenance and updating of the architecture specification are essential parts of the architecture control procedure.

Responsibility: Although the adoption of the architecture specification and compliance with it are based as much as possible on cost and performance analyses and on consensus among machine and software implementers, final authority for the architecture definition rests with the architecture group. Architecture is recognized within IBM as an autonomous function which analyzes the requirements of users and implementers and, in re-

sponse, produces the specification of how the machine must appear to the program. It is an ongoing operation, as the definition must be maintained and extended across product cycles.

One person, the chief architect, is responsible for the contents of the Principles of Operation. He must obtain the approval of the managers of each implementation before any change can officially be made, and he calls and chairs architecture resolution meetings. The architect's decisions at these meetings are binding unless and until successfully appealed to high authority.

These procedures, especially the parts that result in less authority or autonomy for implementing engineers, were not accepted lightly or without considerable debate and management leadership. Most of this methodology was developed by Fred Brooks during the early days of the System/360 development, and it has survived to the present. It succeeds in large part because of the high competence and personal professional dedication of the architecture group. They win most of the arguments by being right, not just because they have nominal authority. The process also works because the architecture group has considerable experience and sympathy with the problems of practicing engineers and programmers.

Architecture Extensions

This section describes the main features of the System/370 architecture extensions and provides some discussion of the motivation for them. It includes a brief summary of the architecture, the purpose of the function, the reasons for the architectural decisions, and some of the main alternatives considered.

Virtual Storage

Motivation: The single item that most distinguishes the architecture of System/370 from its predecessor, System/360, is the availability of a dynamic-address-translation facility, which allows programming systems to efficiently implement a group of functions which are collectively known as virtual storage. This system incorporates paging from a backing store as introduced in Atlas[13], and a second level of indirection, segmentation, as suggested by Dennis[9] and as further detailed by Arden, et al.[3].

The System/370 version of this facility was largely patterned after the System/360 Model 67[10]. Our experience with that machine and its operating system, TSS, had verified the value of many of the concepts and had given us actual usage data with which to judge design decisions for System/370.

The motivation for virtual storage and some of its value can be understood by considering several somewhat overlapping topics:

(1) Roll-in and roll-out
(2) Fragmentation of real main storage
(3) Application-program development
(4) Dynamic size adjustment
(5) Compatibility of large and small storage sizes
(6) Protection and sharing
(7) Virtual-data access
(8) Virtual-machine simulation

The following sections discuss each of these items.

Roll-In and Roll-Out: Prior to the introduction of virtual storage, each application program was assigned real main-storage locations at the time it was initiated. Thereafter, the program, as well as its data, might be swapped out of main storage while waiting for terminal or I/O service. When the program was subsequently returned to main storage, it was constrained to occupy the same real locations as it did previously, since relocation to a different set of locations was extremely inconvenient.[5]

This restriction of programs and data to the initially assigned real-storage locations leads to conflicts, such as when a program that is ready for execution is barred from entering main storage by another program residing at the assigned locations, even though contiguous unused space of sufficient size is available at some other address, and even though the CPU may not be fully occupied. The overall result is that system throughput is reduced and response time increased.

With virtual storage, any part of main storage is available for any application, regardless of the locations to which it had initially been assigned. By preventing conflicts for real-storage locations, the performance of the whole system may well be significantly improved.[6]

Fragmentation of Real Main Storage: If the various application programs are of differing size, the storage-allocation problem is even more difficult. Not only may a program be blocked from its initially assigned locations, but even in batch operations, which run applications to completion after they are initially loaded, only part of the main storage can be utilized at any one time. As jobs are completed at various times, the available storage can be assigned to new jobs only to the extent to which the waiting jobs can utilize the available contiguous spaces. As a result, relatively long-lived "holes" are formed in main storage which are individually too small for any job, but which collectively are larger than needed for some or all waiting jobs.

---

[5] It has been argued that this is not necessarily so. The basic System/360 architecture makes all problem-program main-storage references via a register. With appropriate programming conventions, an operating system might be built to allow the relocation of programs and data on arbitrary boundaries without dynamic-address-translation hardware. In practice, however, such a design would probably become too restrictive in the types of programs allowed, or too complex and too slow to be acceptable for a broad class of applications. It would be particularly inconvenient for programs that store base-register values for later use or for programs which do arithmetic on base-register values, as is often required for the use of SS-format instructions. Finally, because it would introduce new programming conventions, it is very unlikely that such relocation could be applied to existing programs.

[6] This benefit could also be obtained by a system with a simpler relocation mechanism than the one described here.

Virtual storage allows the efficient collection of fragments of main storage into one contiguous address space without moving or disturbing the programs in process. The result is a more efficient use of main storage and more throughput.

Application-Program Development: Prior to virtual storage, the size of the installed main storage constrained application-program development. Often the effective upper limit of an application program had to be much less than the installed storage size in order to provide for a resident supervisor and I/O package, and because partitions for other applications were needed to ensure a reasonable level of multiprogramming.

In many cases, a considerable programming effort was expended in planning overlays or phases in processing. This was true even when the application program was such that most of the code was seldom executed, it being present only for unusual or error situations. Furthermore, sometimes modifications to a program which once fitted its allocated partition would cause it to just exceed the available space. Fitting this program into its previous space was likely to require substantial rework for little return.

Virtual storage allows programs to run with an allocation of real main storage which is independent of the size of the application code. It allows many applications to be coded with little regard for absolute space limits. Space in real main storage is not assigned to seldomly executed parts of the program, and programs can continue to be properly executed even if they grow.

It is, of course, misleading to suggest that developers of large or frequently executed applications should remain ignorant of their main-storage requirements or addressing patterns. Poor design can require extensive paging and thus result in poor system performance.

Dynamic Size Adjustment: In many cases it has been observed that the dynamic allocation of storage to a program can be more effective than the best static allocation by a programmer. Thus, the effective size of an application may well be smaller under dynamic allocation than with preplanned overlays. This allows even more efficient use of main storage and may further increase system throughput. The functions of dynamic location assignment and dynamic size control interact with each other in a favorable way. The "working size" of the application changes with time, and the allocation capability allows more applications to be resident in a fixed memory space. Without dynamic size adjustment, contiguous storage was often reserved to meet the largest storage requirement for the application, part of the storage being unused for most of the execution time.

Compatibility of Large and Small Storage Sizes: The machine compatibility objectives of System/360 stated that valid programs on one model would also be valid programs on another model, provided (in part) that the second model was configured with at least as much main storage as the program required. On some models it was not possible to install a large enough main storage.

The advent of virtual storage makes this condition obsolete. Since the available virtual storage of all models is now equal, programs written to run under a virtual-storage operating system may be freely transferred to another model, provided that it meets the real storage-size requirements of the operating system. Performance, of course, is significantly degraded on a model that has much less main storage. The ability to run a program on any model, even if at a degraded performance, may prove particularly useful in emergency situations where critical processing must be done when the normal equipment is unavailable.

In addition, virtual storage allows, without reprogramming, an immediate increase in system performance when real main storage is enlarged. This may be important to installations with increasing workload where it is not desired to recode or restructure the application set.

Although usually the contrary is assumed, it is possible to consider systems in which the real main storage is larger than the virtual storage assigned to any one program. Several routines, multiprogrammed, then would reside to utilize the available main storage. Such a system would have the advantage that address constants in problem programs could be smaller. Only the supervisory program would need to have enough total addressability to access the entire main storage.

Protection and Segmentation: By appropriately managing the contents of the address-translation tables, an operating system may allow one problem program access to only a part of the total data in main storage, or, alternatively, may allow two or more programs to share the same data. This ability to share some but not the entire contents of main storage and to prevent all access to other contents is very useful in maintaining the integrity and security characteristics of an installation.

This method of protection is more flexible and selective than the System/360 key-controlled protection because even routines with key 0 are restrained from accidental access to data that is not assigned to them by translation-table entries. (It may be possible for these routines to modify the tables.) Furthermore, whereas the keys permit up to 15 different concurrently resident programs to be isolated from each other, translation tables permit individual access control for any number of programs. Operating systems may well use a combination of storage keys and translation-table contents for maximum flexibility and control.

Virtual Data Access: Normally I/O operations are used to transfer data between the data sets on an external storage device and the storage that can be directly addressed by the program. Virtual storage can be used to avoid these explicit I/O operations. This is accomplished by combining the mechanism used to manage virtual storage with that used for managing external files.

Programs which implement virtual storage include tables, related to the address-translation tables, that identify, for pages not currently in main storage, the location of the page on the external storage medium,

such as a disk. Analogous tables normally exist for external data files, which map data-set names to locations in external storage. With appropriate design of these tables and data formats, it is possible to "move" data between the virtual-storage area and the data-set area by modifying table entries, thus taking advantage of the paging mechanism to perform the I/O operation.

Such data access improves efficiency, as actual data movement into main storage occurs only when the application program refers to the data; on output, movement may be avoided when the data is already in the external device.

Viewed from another perspective, this approach provides a way of extending the size of the virtual storage to encompass all online data, with the restriction that any one program can have only part of the online data in its own virtual storage at any one time.

This technique was advantageously used in the TSS operating system on System/360 Model 67, where it was known as VIO.

Virtual-Machine Simulation: It has been found useful in many installations to use an operating system to simulate the existence of several machines on a single physical set of hardware. The IBM VM/370 operating system is one example. This technique allows an installation to multiprogram several different operating systems (or different versions of the same operating system) on a single physical machine. The dynamic-address-translation hardware allows such a simulator to be efficient enough to be used, in many cases, in production mode.

Dynamic-Address-Translation Mechanism: Address translation is achieved by treating the addresses supplied by and available to the CPU program as designating locations in virtual storage. The dynamic-address-translation mechanism translates these addresses to real addresses, which designate locations in real main storage.[7]

Translation Procedure: Translation is performed by the use of two stages of tables in main storage. The high-order bits of the virtual address are used to select an entry from the *segment table.* This entry contains the origin of a *page table,* which is indexed by the mid-order bits of the address. The low-order bits of the virtual address are concatenated with the real address contained in the page-table entry to form the real main-storage address. The origin of the segment table is designated by the contents of control register 1. The extent of virtual storage accessed through a segment-table entry and a page-table entry is referred to as a *segment* and a *page,* respectively.

Controls are provided in the PSW to turn dynamic address translation on and off and in control register 0 to specify the size of segments and pages. The instruction LOAD REAL ADDRESS allows a program to explicitly determine the current real address corresponding to any virtual address. This is needed in several routines that translate channel programs or allocate and manage real main storage.

The two-stage translation procedure was selected for several reasons:

(1) It provides a convenient way for segments to be shared among different programs, using differing virtual addresses, without requiring multiple page tables and multiple table changes when the pages are replaced.

(2) It results in less total storage taken by the tables by permitting the tables to be abbreviated when the total possible virtual storage is only sparsely allocated.

(3) It limits the size of the largest table to less than a page, thus facilitating the allocation of main storage to the tables.

(4) It provides a convenient way for a portion of the tables (the page tables) not to be resident in main storage at all times. The page tables themselves may be paged out, in which case the "invalid" bit in the segment-table entry causes an interruption on an attempt to use the page table for translation.

Translation-Lookaside Buffer: If translation tables in main storage were actually accessed for each storage reference, the number of storage references would be tripled, causing a totally unacceptable performance degradation. In order to avoid such degradation, all implementations in the System/370 line include a hardware facility called the *translation-lookaside buffer* (TLB). The TLB is a group of fast-access registers that contain the results of recent references to translation tables. The access time to information in these registers is a small fraction of the main-storage access time, and they intercept about 99% of all the references to tables in storage. The TLB makes the performance degradation associated with table references minimal.

The instruction PURGE TLB causes the TLB to be cleared of all entries. It provides a way of informing the translation mechanism that the software has changed the contents of the current translation tables in main storage and that the tables must be reaccessed rather than relying on their previous contents as reflected in the buffers.

Segment and Page Sizes: The architecture, as well as its machine implementations, provides for any combination of two different segment sizes (64K bytes and 1M bytes) and two different page sizes (2K bytes and 4K bytes).[8]

These parameters were provided to accommodate the range of expected main-storage sizes and disk characteristics. Small page sizes are needed for efficient use of the smaller main-storage sizes, while large pages are needed to reduce CPU and I/O time in main-storage to disk transfers. Large segment sizes allow convenient handling of large data and program files, while small

[7] The actual reference to main storage may occur only after a further translation known as "prefixing." This is described in the section on multiprocessing.

[8] In this paper K stands for $2^{10} = 1024$, and M stands for $2^{20} = 1{,}048{,}576$.

segment sizes provide for easier storage allocation to translation tables and for more segment names.

Each IBM operating system uses only one combination, the use being as follows:

Segment	Page	System
64K	2K	DOS/VS, OS/VS1
64K	4K	OS/VS2, VM/370
1M	2K	
1M	4K	TSS

Reference and Change Recording: When a reference is made to a page not currently in main storage, the operating system must decide which currently resident page is least likely to be used next and hence should be replaced. For the page that is to be replaced, it must decide if the copy in main storage has been modified and hence needs to be saved or if it can be overlaid because the copy in external storage is still current. Two bits of information about each 2K-byte real-storage block, as well as the instruction RESET REFERENCE BIT, are provided to assist these decision processes.

One bit, called the *reference bit,* is set by the machine to one whenever the block is referred to by the CPU or the channel. It is intended to provide the basis for selecting the page to be replaced. The other bit, called the *change bit,* is set to one whenever storing is performed into the block. This bit may be used by the software to determine if the copy of the page occupying the block must be transferred to external storage prior to reallocation of the the block. The page-replacement algorithm may also select unchanged pages in preference to changed pages in order to avoid this transfer.

Translation of Channel Programs: After considerable analysis, it was decided not to include the address-translation capability in the System/370 channels but rather to provide a mechanism to assist software in performing the function. Several considerations were important in reaching this conclusion:

(1) The allowable channel data rates were limited in many cases by the main-storage accesses necessary during data chaining or command chaining. On some implementations, the extra storage accesses implied by a translation capability would have reduced the maximum data rates to unacceptably low values.

(2) Since the channels operate asynchronously with the CPU, and often on behalf of different tasks, a full channel relocation capability would have implied different translation tables for the CPU and for each of the many subchannels. The resulting constraints on software paging and storage management were felt to be unnecessarily burdensome.

(3) The extra channel hardware cost, especially to retrofit some of the existing implementations, would have been significant.

(4) The performance penalty, of scanning each newly created channel program at START-I/O time and replacing the virtual addresses with real addresses, was reduced somewhat by the need to scan the same programs anyway to cause the allocated page frames to be fixed (removed from the set eligible for paging) even if the channel were to contain hardware to perform the virtual-to-real translation.

(5) The design of new access methods, such as VSAM and VTAM, was expected to eliminate the need for software to translate I/O data addresses in channel programs and hence cause this issue to disappear in the future.

The channel-indirect-data-addressing facility is provided to assist the operating system in the translation of channel programs. It permits a single channel-command word (CCW) to control transfer of data that spans several potentially noncontiguous pages in main storage. When a CCW specifies indirect data addressing, the data-address field of the CCW is not used directly to address data but rather contains the address of a list of indirect-data-address words. A new address word is obtained by the channel whenever a 2K-byte boundary is crossed in main storage. The address words, containing a 32-bit address field, provide also for the eventual extension of the storage address in conformity with the general System/370 objectives.

Program Control

PSW and Control Registers: In System/360, all CPU state information (other than the contents of general and floating-point registers) is arranged in the 64-bit program-status word (PSW), which provides a convenient way of introducing a new CPU state by an instruction or an interruption, as well as a way of saving the old state on an interruption. The new facilities introduced by System/370 expanded the amount of information relevant to the CPU state, as certain additional control information had to be specified that spans the execution of a sequence of instructions; and, on encountering exceptions, additional status information had to be provided to the program. Since no unused bit positions were available in the PSW, the requirements for the additional control and status information were met by modifying the PSW format, by introducing a set of sixteen 32-bit control registers, and by assigning locations in main storage for control and status purposes.

Additional space in the PSW is obtained by removing the 16-bit interruption code and the two-bit instruction-length code from the System/360 format and by replacing the six channel masks with a single I/O mask. Two new controls are placed in the PSW — one bit to turn program-event recording on and off and one bit to turn dynamic address translation on and off.

All additional control information is placed in the control registers. The control registers are, in effect, an extension to the PSW, except that their contents are not changed by the machine on an interruption. Two instructions, LOAD CONTROL and STORE CONTROL, are provided for loading and inspecting their contents. The control registers are addressed similarly

to the 16 general registers, and multiple contiguous registers may be handled by one instruction.

All information that describes the cause of an interruption is placed in specifically assigned main-storage locations. The information is arranged by interruption classes, with additional fields left unassigned for future expansion.

For I/O, a four-byte location is also assigned in main storage that contains an address that specifies the storage area for diagnostic channel logout. Additionally, a four-byte location is assigned in main storage where channel identification is placed on execution of the instruction STORE CHANNEL ID. These I/O related fields are in main storage rather than a control register since they must be accessed or modified by the channel. The channel is, in effect, a separate processor sharing main storage but having otherwise a limited ability to communicate with the CPU.

PSW bit 12 specifies the format of the PSW and the execution of interruptions. When PSW bit 12 is zero, the PSW has the System/360 format, and the CPU is said to operate in the basic-control (BC) mode; when bit 12 is one, the new PSW format and the extended-control (EC) mode are specified. It should be noted that the BC-EC mode distinction pertains only to information appearing in the PSW. Control registers, as well as the facilities associated with control registers (monitoring, machine-check controls, extended external masking, etc.), are operative in both modes, subject to the availability of PSW control bits. Program-event recording is defined to be off in the BC mode, as is implicitly invoked dynamic address translation, but the instruction LOAD REAL ADDRESS with the associated explicit use of the dynamic-address-translation facility is valid in the BC mode.

The following observations guided the architectural decisions:

(1) On an interruption, as well as on a programmed transfer of control (LOAD PSW), the machine must indivisibly replace a certain amount of control information, including the instruction address, protection key, problem-supervisor mode specification, and masks to disable further interruptions. For performance reasons, changing of other control information should be optional and can be explicitly performed by the supervisory program. This applies particularly to control information that pertains to system functions and that is changed infrequently (page size, controls for recovery from machine errors, etc.).

(2) Certain information in the BC-mode PSW is meaningful only for the determination of the cause of the interruption and is not used to control machine operation. Priority for PSW space should be given to control information. The interruption code and the instruction-length code, which for most interruptions is only a fraction of the total status information provided, can as well be placed with the rest of the status information in main storage.

One alternative for handling the additional control information was to expand the size of the PSW. Such an approach leads to the temptation to define a program status block for the control of the machine containing all information for a dispatchable program unit, including the values of general and floating-point registers, timer values for accounting purposes, etc. This in turn requires some assumptions for operating-system procedures, such as conventions for passing parameters in subroutine linkages. Thus, it leads to further extensions of the control block with information required by the operating system.

Such an approach would have increased the time for simple task switches, already too slow. Additionally, a number of considerations argued against incorporating operating-system structures in the machine architecture. A number of operating systems, with differing requirements, were anticipated for the System/370 line of machines, and no one set of formats and algorithms could satisfy them all. More importantly, the architectural extensions introduced a number of new concepts and facilities that had not yet been implemented in a total system design. As a result, the general design principle was adopted to include in the machine architecture only the essential primitives and elemental tools for performing the needed function.

System-Mask Handling: Normally, on System/360 machines, the OS/360 operating system operated either entirely enabled or entirely disabled for I/O and external interruptions; accordingly, enabling and disabling was accomplished by setting PSW bits 0-7 to a byte of ones or zeros. With the change in the PSW format and the introduction of dynamic address translation, program-event recording, and other potential extensions having control bits in PSW bit positions 0-7, setting all bits to the same value was no longer appropriate, and the operating system had to be modified to treat the system mask accordingly. This required the identification of all places in the program where the mask is changed, including interruptions and execution of LOAD PSW or SET SYSTEM MASK (SSM).

Because of the difficulty of finding all occurrences of SSM and because in the EC-mode PSW bits 0-7 normally are not replaced in their entirety, a mode was introduced where the execution of SSM is suppressed and instead causes a program interruption. The interruption signals where the original program needs to be modified.

The suppression of SSM is useful also for the conversion of the operating system from uniprocessor to multiprocessor operation. In a single-CPU system, the disabling of the CPU is a sufficient means for avoiding use of a serially reusable resource associated with I/O or external interruptions. When two or more CPUs share those routines, such disabling is not adequate, as the use of the resource by the other CPU also must be prohibited. Access to the serially reusable resource must be controlled by other means, and the interruption on encoun-

tering SSM aids the conversion by signaling where the semaphore instructions should be placed.

The two new instructions STORE THEN AND SYSTEM MASK and STORE THEN OR SYSTEM MASK provide the means for turning any bit in PSW bit positions 0-7 off or on. Furthermore, these instructions save the original value of the field in main storage so that a service routine making these changes could, on exit, restore the field to its original value. In System/360 the current value of the masks can not be determined without causing an interruption.

PSW-Key Handling: In the original design, most parts of the OS/360 operating system operated with a protection key of zero, thus having access to all parts of main storage. In the design of the OS/VS2 operating system, one step taken to catch programming errors was to use a nonzero protection key for the various components of the control program, thus protecting one component from inadvertent storing by another component.

Two instructions are provided for inspecting and setting the protection key in the PSW: INSERT PSW KEY (IPK) and SET PSW KEY FROM ADDRESS (SPKA). The first one places the protection key into a general register, and the latter replaces the key in the PSW with the four low-order bits of the operand address.

These instructions permit the key in the PSW to be set and subsequently restored when a component is entered with an unknown key and subsequently left, or when a routine must modify data having a different storage key. When a supervisor routine which normally uses a key of zero is called to perform a service that involves storing in a user region, SPKA is also useful for verifying that the requestor is authorized to perform the storing. In this case, the supervisor can use SPKA to set up the user's key for the duration of the operation.

Interruptions: System/370 expands the five System/360 interruption classes (machine check, supervisor call, program, external, and I/O) by introducing a new class — the restart interruption. This interruption occurs in response to the externally initiated restart signal and is intended for the manual debugging of the machine and for intervention by another CPU. In view of the intended purpose, no mask bit is provided for disallowing the interruption.

The control of interruptions is made more flexible by providing mask bits in control registers for each type of external condition, for each I/O channel, and for the different types of machine-check conditions. For any specific source, an interruption can occur only when both the corresponding mask in the control register and the class mask in the PSW allow it.

By means of the masks in the control registers, the supervisory program can disallow interruptions for some sources within a class, such as for machine-check recovery reports. They also allow the enabling for conditions of higher priority after an interruption for a lower-priority condition within the class has occurred, but before other interruptions from the lower-priority condition can be permitted. Thus, the program can simulate an interruption mechanism with a priority hierarchy.

Data-Manipulation Instructions

Well over a hundred instructions were considered for inclusion in System/370 architecture to improve the cost effectiveness of the machine for the applications and data structures that had evolved with the use of System/360 or that were anticipated for System/370.

Out of these, seven general instructions, one decimal instruction, and seven floating-point instructions were adopted for System/370. The floating-point instructions provide for arithmetic on the new extended-precision format, as well as for rounding from extended to long precision and from long to short precision.[9] The extended-precision format has a fraction of 28 hexadecimal digits, and the considerations associated with the design of the architecture are described by Padegs[16].

The following is a summary of the operation and design considerations for the general and decimal instructions.

Justification Methodology: The value of a new instruction can be expressed in terms of an increase in CPU performance and a reduction in the program size, the performance gain being a function of the gain per occurrence of the instruction and its frequency of use. On the other hand, each instruction has a machine implementation cost that can be expressed in terms of additional circuits and control-storage locations. A serious attempt was made to express the cost effectiveness for the more promising proposals in terms of specific value and cost numbers. However, the decision was ultimately based largely on judgment because of the following difficulties:

(1) The performance of a new instruction depends on the extent to which it is integrated in the machine. A specific estimate for an addition to the architecture can be made only when the basic design of the machine is already laid out, and such an estimate normally is made assuming the least perturbation of the design, yielding lower performance.

(2) An instruction is used depending on its performance, and its performance in a new machine design is a function of its frequency of use. A new instruction without a proven value is likely to be implemented at minimum cost and performance.

(3) When the function performed by a new instruction is a concatenation of functions performed by a sequence of more primitive instructions, the cost and performance considerations differ in large and small machines:

> The elimination of the instruction fetching time may yield some performance gain in a medium-speed machine but is likely to be insignificant in a very

[9] The extended-precision floating-point capability was also available on System/360 Models 85 and 195.

small serial machine or in a large machine that overlaps phases of execution.

In a large machine, frequent simple instructions may be performed in their entirety in hardware as part of the instruction decoding phase. If such a simple function is made a part of another more complex instruction, either the execution of the composite function is made slower by implementation in microcode, or additional cost in hardware is incurred.

(4) Some instructions, such as those for conversion between fixed- and floating-point formats, are used only in specialized environments, and an average number for their frequency of use is not meaningful. The potential usage of other instructions, such as those for setting and testing bits, is so pervasive that it is not possible to determine a meaningful usage frequency.

(5) For some instructions, such as those for moving bit strings or for operations on list structures, justification cannot be based on where the new instructions could be used in programs currently written but rather on what new applications or program structures the instructions would make attractive.

The final choice of the new instructions was strongly moderated by such somewhat subjective attributes as consistency of design, generality of function, and simplicity of use. It was made subject to the rule that a new instruction can be adopted only if it will appear in the object code compiled from a high-level language or if it will be used within a programming system in a significant way.

Movement and Comparison with Long Operands: The two instructions MOVE LONG (MVCL) and COMPARE LOGICAL LONG (CLCL) are enriched versions of the basic byte movement and comparison operations, respectively. They provide for operand sizes of up to 16,777,215 bytes, true length designation, padding, marking the byte of mismatch (for CLCL), and test for destructive overlap (for MVCL).

Many users had asked for "move" and "compare" instructions with long operands, and the padding function in MVCL is valuable for clearing storage to zeros, blanks, or any other code. The specific attributes of these instructions, however, were established largely to permit convenient byte-string manipulation in programs generated by the PL/I compiler. At the time a byte-string operation is compiled, the size and relation of the two operands is not known, the specific parameters being bound in the program only at execution time. Hence, the object code must provide for various special cases of overlap, length mismatch, etc. It was estimated that MVCL could eliminate as many as 1,000 bytes in the PL/I object-code library.

Because the processing of an operand of 16 million bytes would take much longer than the execution time of any other System/370 instruction, execution of MVCL and CLCL was made interruptible, thus avoiding the loss of real-time responsiveness due to the potentially long operands. If a condition is due to cause an interruption, the execution of the instruction is suspended, operand addresses and counts in the general registers are adjusted by the number of bytes processed, and the instruction address is left to point to the MVCL or CLCL instruction. When control is returned to the interrupted program, execution of the interrupted instruction is resumed. To the machine, the initial start and the resumption of execution are identical.

Handling of Bytes in Registers: The three instructions INSERT CHARACTERS UNDER MASK, STORE CHARACTERS UNDER MASK, and COMPARE LOGICAL CHARACTERS UNDER MASK are provided to increase the convenience of manipulating a variable number of bytes between general registers and storage. The instructions select the bytes in the designated register by means of a four-bit mask, with the bits corresponding to the four bytes. The storage operand contains the bytes in a contiguous field. Among other functions, the instructions permit loading and testing 24-bit addresses.

Conditional Swapping: The two instructions COMPARE AND SWAP (CS) and COMPARE DOUBLE AND SWAP (CDS) are intended for use by programs sharing common storage areas in either a multiprogramming or multiprocessing environment. They may be used to add or delete elements in chained lists or to identify the holder or requestor associated with a lock for a serially reusable resource. They are System/370 primitives which can be used to control access to critical regions in a manner similar to Dijkstra's semaphores.

These two instructions designate a storage operand and two register operands. They cause the storage operand to be compared with the first register operand: if they are equal, the storage operand is replaced with the second register operand; if not, the first register operand is replaced with the storage operand. The result is indicated by the condition code. When an equal comparison occurs, no access is permitted to the storage location between the fetching of that operand and its replacement. The two instructions are the same except that for CS the operand comprises one word and for CDS a doubleword.

The following is an example of a procedure using CS, whereby a program can modify the contents of a storage location even though the possibility exists that the program may be interrupted by another program that will update the location or that another CPU may simultaneously update the location.

First, the storage operand is loaded into a general register, which then contains the first register operand. Next, the updated value is made the second register operand. Then CS is executed. If condition code 0 is set, the update has been successful. If condition code 1 is set, the storage location has been found to contain a different value, the update has not been successful, and the first register operand has been replaced by the new current value of the storage operand. The program in

this case can repeat the procedure, bypassing the first step.

Decimal Shifting: The SHIFT AND ROUND DECIMAL instruction is provided for the convenience of decimal shifting, which is common in commercial applications and in the simulation of the decimal floating-point format. To permit "late binding" in the object code generated by a compiler, both left and right shift are included in one instruction. Rounding is accomplished by adding a specific digit specified in the instruction.

Byte-Oriented Operands: System/370 removes the original System/360 requirement that halfword, word, and doubleword operands in storage must be aligned on the natural boundary for the size of the operand. Instead, it permits the operands of virtually all nonprivileged instructions to start on any byte boundary.[10]

This change was made to allow direct processing of all fields obtained from external sources without knowledge of whether they are properly aligned. The primary motivation was to make it easier for users to determine record lengths and to allow compilers to provide a consistent alignment algorithm and therefore to permit convenient data exchange among programs written in different languages. The principal compiler problem occurs when sub-parts of data structures are passed as parameters to separately compiled procedures. In this situation the receiving program cannot assume the starting alignment position, and no universal padding convention can be established to shift the field to its natural boundary. In addition, the change may assist in processing records which are obtained from or destined for equipment not in the System/360-370 families.

The use of operands which are not aligned on natural boundaries will result in considerable performance penalties on some models, especially the faster ones. All machines, however, are designed with the guideline that the performance penalty should be less than the time required to move the operand to an aligned location and then move the result back.

Timing Facilities

Summary: The new timing facilities are introduced as a replacement for the System/360 location-80 interval timer. The 31-bit format of the interval timer provided for a resolution of 13 microseconds and a period of about 15.5 hours and did not meet some of the more demanding timing requirements. Furthermore, the need to share the single timer for the various timing needs introduced significant software overhead.

System/370 offers three new facilities for measuring time: a time-of-day clock, a clock comparator, and a CPU timer. These facilities jointly provide the time measurements which a program may need. System/370 continues to provide the interval timer at location 80 in main storage, which is included for compatibility with System/360. It meets no requirements not already met by the other three facilities.

The time-of-day (TOD) clock is a binary counter with a period of about 143 years and a resolution, depending on the model, that is on the order of one microsecond. The doubleword format allows for an extension of the resolution to one-quarter nanosecond. Operating in conjunction with the TOD clock, the clock comparator causes an interruption when the TOD clock has advanced to a value greater than that in the clock comparator. The CPU timer is also a binary counter, with a format the same as that of the TOD clock, except that it is considered to have a signed value. The contents of the timer are decremented, and an interruption occurs when the value is negative.

Three "setting" instructions are provided whereby the program can place a specific value in each of these timers, and three "storing" instructions allow for placing the current contents of the timers into main storage for subsequent inspection. The STORE CLOCK instruction is not privileged so that any program can have access to the TOD clock; the other five instructions are made privileged to ensure integrity of the timer values and to permit sharing the clock comparator and CPU timer among programs. Additionally, the SET CLOCK instruction is interlocked with the operation of a console switch, so that the program can alter the clock setting only when such alteration is allowed by the operator. This interlock ensures that the clock value does not get changed accidently because of an error in the operating system, which is helpful for recovering and debugging system operation.

To provide a compatible recording of time among systems, January 1, 1900, 0 am GMT is established as the standard time origin, or epoch, that is the calendar date and time to which a clock value of zero corresponds. This date permits retroactive assignment of TOD clock values to transactions. The enforcing of this convention is the responsibility of the operating system. Local time is calculated when needed by subtracting an offset from the TOD clock value. It is only this offset that needs to be changed for different time zones, daylight-savings time, etc.

Design Considerations: The interaction of several design considerations was involved in the final specification.

Timing Functions: The new timers are provided to meet four distinct timing functions. Two of these needs relate to real time:

> *The current real-time value,* which is needed for labeling events and transactions with the time of their occurrence (time-stamping) and for measuring elapsed real time. Time stamping is needed, for example, to record the time when an exceptional condition is detected or when a transaction request is received. Elapsed real-time measurements, obtained

[10] The byte-oriented-operand capability was also available on System/360 Models 85 and 195.

by taking the difference between two real-time values, are needed for such purposes as determining the duration of real-time processes and establishing charges for use of the system's storage media or terminals. This need is met by the TOD clock.

An interruption at a specific real-time instant, which is needed for the control of many real-time processes. Applications may include sampling a sensor, changing traffic light patterns for an approaching rush hour, or polling a terminal. This need is met by the clock comparator.

The time which accrues only when the CPU is actually executing a particular program is referred to as the process time for that program. The following two needs must be met in relation to process time:

The current process time value, which is needed for establishing elapsed process time for performance evaluation and accounting for the use of the CPU, and related functions.

An interruption at a specific process-time instant, which is needed for such functions as checking a program to protect against unending loops and rotating the use of the CPU among different programs, referred to as "time-slicing."

The system must maintain as many accumulators of process time as the number of independent programs that concurrently reside in the system. However, since the CPU executes only one process at a time, only one of these timers can be running at one time, and hence only one machine timer is needed. The CPU timer is provided to satisfy both needs associated with the process time.

Long TOD-Clock Period: In order to permit direct problem program access to the TOD-clock value and to avoid the need for special software procedures for handling of clock overflow, the period should span the lifetime of the environment using the format and algorithm for time measurement. As a minimum, it should cover a number of hardware and operating system generations. A period of 143 years provides this, even with a time origin set to the year 1900.

Unique TOD-Clock Values: The clock should provide nonrepetitive readings, so that the time-stamp labels provided by the clock can serve as unique serial numbers for the identification and cataloging of system objects. In view of this, the STORE CLOCK instruction is defined such that no two references to the TOD clock of a CPU, or to any of the TOD clocks of a shared-main-storage multiprocessing system, provide the same value. Either the clock has a high enough resolution to be updated between two such instructions, or references to the clock are specifically interlocked to ensure the uniqueness of readings.

Synchronization with External Signals: For the accuracy of the TOD clock's real-time indication to be comparable to its resolution, it must be possible for the program to set the clock to a specific value and then start its operation in response to an external signal. This function is particularly essential for synchronization of the clocks of two CPUs and is provided by the TOD-clock synchronization control, which is included in the multiprocessing feature. When the control bit is one and SET CLOCK is executed, the clock stops. It resumes incrementing only after a synchronizing signal from the other CPU arrives. This signal is generated by a carry into bit position 31 of a running clock, and is defined so that, with zeros in bit positions 32-63 of the stopped clock, the low-order words of the two clocks are subsequently incremented in synchronism. The high-order words of the clocks, approximately corresponding to counts of seconds, can be synchronized by the program.

Format: For interpretation by people, a TOD clock format of such form as year-month-day-hour-minute-second-fraction is most convenient. Such a format, however, was rejected because of the difficulties it would present for arithmetic operations. The specific format was adopted because of the efficiency of binary encoding and by observing that the external formats may have to meet different operating-system or installation requirements and hence should be under software control.

Implementation: In spite of the need for the functions, inclusion of three 64-bit timing facilities would appear rich if each actually required a hardware register. It is possible, however, for a microprogrammed machine to implement the clock comparator and the CPU timer with a small counter and two doublewords of local storage. This is, in fact, the implementation used on most models. Further savings are permissible by implementing most high-order bytes of the TOD clock in local storage.

Multiprocessing

System/370 architecture includes a number of facilities that permit formation of a system where two or more CPUs share common main storage and are controlled by a single copy of the operating system. Such a system has a number of advantages:

(1) It offers higher processing power and throughput.

(2) It improves reliability by making an alternate CPU available and by increasing the redundancy of other system components.

(3) It permits more flexibility in sharing I/O and external storage devices.

(4) It provides a larger pool of main storage, channels, and I/O equipment for allocation of these resources in response to demands by various jobs.

This section reviews the facilities included in System/370 for multiprocessing.

A rudimentary form of some multiprocessing facilities was available also on System/360 Models 65 and 67, which offered a shared-main-storage multiprocessing capability. Prefixing on these models was provided using a manually settable prefix. A limited inter-

processor signaling capability was made available through the use of the channel-to-channel adapter. Instructions-stream synchronization and serialization were left mostly unspecified by the architecture; the action of the machine was determined by the implementation. In addition, configurations of modified Model 50 CPUs, designated the IBM 9020, were built as part of a special system for the Federal Aviation Administration.

It should be noted that although current implementations offer multiprocessing systems comprised of two CPUs, the architecture allows for a multiplicity of CPUs.

Synchronization and Serialization: In a uniprocessor, the execution of a single instruction, as well as of a disabled routine, can be considered instantaneous in that no other program can observe or change any intermediate result values, and all references to main storage can be considered to occur in the sequence specified by the program.[11] In a multiprocessing system, the results of all communication between CPUs through main storage are based on the actual storage accesses. When these accesses are observed by another processor, they may differ from the expected operation in the following ways:

(1) A single instruction may make a number of distinct accesses to main storage, and accesses associated with single instructions may be interleaved by CPUs.

(2) The accesses due to a single instruction and due to any two instructions are not necessarily performed in the specified order.

(3) Accesses within a field, such as for an instruction or an operand, may be made piecemeal.

(4) Multiple accesses may be made to a storage location for a single use of its contents.

Results become unpredictable, and the conventions of a uniprocessor communications protocol become inadequate when one CPU is changing the contents of a common storage location while the other is observing it, or when both CPUs are updating the contents of the location at the same time.

System/370 architecture includes a number of specific rules and extensions to make a multiprocessor communications protocol more flexible and efficient. Included are constraints on the concurrency, multiplicity, and order of storage accesses. Specific instructions are defined to serialize and synchronize events. A detailed discussion of those considerations is beyond the scope of this paper.

Prefixing: The control and status information associated with a CPU (PSWs, interruption codes, I/O control words, etc.) reside in fixed low-order locations of main storage. When storage is shared by multiple CPUs, each CPU must have a private control and status area. This is accomplished by providing in each CPU a prefix address, which specifies the storage block to which references with addresses 0 to 4,095 are relocated. In order for each processor to have access to all of the attached storage, and for one processor to access another's fixed addresses even if they are prefixed with a value of zero, reverse prefixing is employed; that is, references to the 4K-byte block identified by the prefix address cause access to block 0. Prefixing, as well as reverse prefixing, is applied after dynamic address translation, and it applies to all storage references by the CPU. Two instructions, SET PREFIX and STORE PREFIX, are associated with the facility.

Prefixing is not applied to storage references associated with I/O data transfers. This decision was made to avoid any logical affinity between a channel and a CPU, thus permiting any CPU to start an I/O operation on any channel in a multiprocessing configuration. It also avoids some additional cost for the relocation hardware in standalone channels and for keeping the prefix address in each subchannel.

Interprocessor Signaling: To fully utilize the potential advantages of a multi-CPU system, some explicit ability for programmed communication among the CPUs is necessary. Such communication is needed for initial startup of the operation, to dispatch jobs because of changes in priority or because of an imbalance of I/O equipment, to recover operations after software or hardware failure, and to diagnose a machine or program problem.

All program-initiated CPU-to-CPU communication is performed by means of the SIGNAL PROCESSOR (SIGP) instruction, which designates the addressed CPU and includes an order specifying an operation to be performed. The instruction can be addressed to the issuing CPU. The orders provide for the following types of functions:

Start; Stop. These two orders are the same as the corresponding operator-console functions.

Stop and Store Status. A sequence of operations is performed comprising the corresponding two operator-console functions.

Restart. A restart interruption is initiated at the addressed CPU, which can be used for initial startup or for dispatching a job.

External Call; Emergency Signal. These two signals cause the corresponding type of external interruption at the addressed CPU, each type of interruption being controlled by a separate mask. They can be used to establish a communications protocol of two priority levels, covering general and unusual conditions.

Sense. The signaling CPU is informed whether the addressed CPU is stopped, still has an external call pending, is in check-stop state, etc.

Reset. Four types of orders are provided for resetting the addressed CPU, permitting a choice in

[11] This statement is not strictly true with respect to channels which may access an area of storage concurrently with the CPU. The channel may see intermediate or out-of-sequence result values if the CPU changes the contents of the I/O data areas during channel operation.

whether channels must be reset and in whether some program-addressable registers must be initialized.

When a CPU enters the check-stop state or loses power, it implicitly generates a malfunction alert. This signal is broadcast to all other CPUs in the system and causes an external interruption in those CPUs that are enabled for it. This mechanism provides for an automatic error alert if and only if programmed communications are no longer possible; at any other time, signaling of all exceptional conditions is under explicit control of the program.

The address assigned to a CPU may be determined by issuing STORE CPU ADDRESS on that CPU. The CPU address may be used to associate with the CPU any facilities that are unique to it, such as an emulator or I/O devices accessible only by it.

Debugging and Monitoring

Two facilities are introduced in System/370 for selectively passing control to a supervisory program on the occurrence of specific events during program execution: *program-event recording* and *monitoring.* Additionally, the *status-storing* facility provides an operator control for recording program status.

Program-Event Recording: The program-event-recording (PER) facility extends and places under program control functions that previously have been available only to the console operator. It is a debugging tool that can be invoked without any preplanning in the design of the program.

The PER facility causes a program interruption on the occurrence of one or more of the following events:

(1) Successful execution of a branch instruction

(2) Alteration of the contents of designated general registers

(3) Fetching of an instruction from a designated main-storage area

(4) Alteration of the contents of a designated main-storage area

The information concerning a program event is provided by means of a program interruption, with the cause of the interruption being identified in the interruption code. The occurrence of the event does not affect the execution of the instruction, and the PER interruption is taken after the execution of the instruction responsible for the event. The supervisory program has control over the conditions that are considered events for recording purposes and specifies the registers and the storage area involved.

The PER facility does not affect CPU performance when it is completely disabled by means of the PSW mask; but on most models it reduces performance when the machine is instructed to search for some events. Its primary use is under conditions when the program is suspected of having a bug. In order to reduce the frequency of PER interruptions, the debugging procedure can select events hierarchically, the initial monitoring being only for instruction fetches or storage alteration occurring outside (or within) a designated area. Recording successful branches or base register alterations should be invoked only when the fault is localized to a particular routine.

Monitoring: The monitoring facility causes an interruption when the MONITOR CALL (MC) instruction is encountered. Each MC instruction identifies itself as belonging to one of 16 separately maskable classes and provides a 24-bit code. On a monitor-call interruption, both the class number and the code are stored to identify the condition.

The MC instruction takes very little execution time when the class is not enabled for interruption; it is useful for signaling critical points in a program, such as dispatching, procedure entries, queue access, and page faults. It is expected that potentially useful points will be identified as part of the design of the program, and that the instruction will be a permanent part of many routines. These instructions then could be used to assist in debugging the system, as well as to record frequency and path information for system performance analysis.

Status Storing: The status-storing facility consists of an operator control that causes the contents of the current PSW and of all addressable registers to be stored at preassigned locations in main storage. It provides a means of preserving the essential status information, upon the failure of a program, for subsequent dumping and analysis. This facility makes it possible for a standalone dump program to record the status of the failing program, without the dump program destroying the status that is to be saved.

Machine-Error Handling

System/370 implementations provide extensive checking for equipment malfunction and include a number of steps for automatic recovery by the machine. The architecture includes extensions that permit reporting of error conditions to assist maintenance and repair and to help with programmed recovery. It provides model-independent structure for the initial response and damage assessment and permits passing additional information for model-dependent analysis. This section reviews the architecture extensions and outlines the characteristics of the implementations that motivated the architecture extensions.

Model independence, or compatibility, in the context of machine-check handling has objectives and constraints somewhat different from those applying to the rest of the system. First, the architecture specifies machine actions in the case when the machine is failing, and hence absolute compliance cannot be guaranteed. Second, the architecture has to reflect the physical structure of the machine, and thus has to provide for some model dependence. As a result, the architectural definition permits a set of actions and alternatives, allowing the machine to choose among them and requiring that it indicate the action it has taken. For virtually all error

situations, the machine must, however, comply with certain basic rules.

One of the fundamental rules of both System/360 and System/370 architecture is the separation of programming and machine errors. Specifically, it must not be possible either inadvertently or by deliberate programmed action to cause an indication of machine malfunction. (This excludes the use of the instruction DIAGNOSE, which is intended for diagnostic and maintenance functions.) Any condition indicating that the operation of the equipment deviates from that normally expected is brought to the attention of the program either via a machine-check interruption or by turning on the corresponding equipment-error bit in the status-word stored by the channel or the SIGNAL PROCESSOR instruction. Conversely, all invalid program situations that are detected by the machine are reported by condition codes, status bits, and interruptions that are distinct from those used for machine errors. In order to ensure that the machine is in a known valid state at the initiation of processing, System/370 architecture defines and introduces a hierarchy of specific reset functions.

The machine-check architecture assumes a rather extensive recording and analysis program as a part of the operating-system facilities. In many cases it is possible to limit the bad effects of a malfunction to just one user, and it should usually be possible to perform an automatic restart so that newly submitted jobs can run. Some solid failures, of course, prevent any useful work from being done. In these cases information must be recorded to expedite diagnosis and repair of the fault.

Recovery Mechanisms: System/370 implementations provide some or all of the following five mechanisms to minimize the destructive effect of machine malfunctions and to ensure integrity of system operation.

Data-Error Detection: Most data and control paths in the CPU, in channels, and on the I/O interface include redundant bits to verify correct transmission and readout of information. The redundancy typically is one bit per byte, or 12.5%. The redundant bit is so chosen as to provide an odd parity for the nine-bit field, thus requiring that at least one bit always have a nonzero value. This organization is capable of detecting any single-bit error.

Data-Error Correction: Main storage for all models except Model 195 is organized into blocks of eight bytes, with eight redundant bits included with the block. The redundant bits form an error-correction code capable of correcting any single-bit error and detecting any double-bit error. When a single-bit error is detected on readout, the error is corrected in the storage array, correct parity is provided to the CPU, and an alert condition is generated. On double-bit errors, an error indication is generated. Error correction may be used also in other parts of the system. Checking and correction is accomplished typically in a fraction of a machine cycle.

CPU Retry: Recovery from transient errors can be accomplished by reexecuting the sequence of steps in which the error occurred. On some models such reexecution, or retry, is invoked automatically by the machine whenever an error is detected, and the steps typically cover the execution of one or a few instructions. CPU retry requires that the machine periodically establish points, referred to as checkpoints, with a known machine-state information. Whenever changes to the machine-state are subsequently made, the previous value for the changed attribute is recorded. In the case of an error, the machine-state is restored to that at the checkpoint, and reexecution is attempted. If the error persists, retry from the same check-point typically may be performed eight times. If the retry is successful, an alert condition is generated; if not, an error is indicated. The time for CPU state restoration and error analysis may be a millisecond or significantly more.

Unit Deletion: On some models, malfunctions of certain transparent units of the system can be circumvented by discontinuing the use of the unit while still continuing processing. Examples include the disabling of all or a part of the cache, translation lookaside buffer, or the high-speed multiplier. When such automatic reconfiguration has occurred, a special signal indicating degradation of operation is generated.

Command Retry: The command-retry facility, which permits recovery from errors detected by the I/O device, is described in the section "Incompatibilities."

Error Reporting: System/370 architecture groups machine errors by type and severity and provides model-independent means for their identification. All machine-check interruptions are subject to the control of PSW bit 13. Additionally, masks for specific conditions permit control over the causes that are to be reported.

Two major types of machine-check conditions are identified, *repressible* and *exigent*. The indication of repressible conditions can be delayed without affecting the integrity of CPU operation. They include recovery indications, alerts of degradation or imminent power loss, and indications of damage to timing or external facilities.

For exigent machine-check conditions, the execution of the current instruction or interruption cannot safely continue and normally is terminated. If the CPU is disabled for machine-check interruptions, the CPU enters the check-stop state. The machine, however, may choose to proceed with processing when the check-stop-control bit so permits. This option is desirable for some real-time applications.

When a machine-check interruption occurs, extensive model-independent information is provided describing the cause of the error. In addition to the machine-check old PSW and source identification, contents of control registers, general registers, floating-point registers, TOD clock, clock comparator, and CPU timer are stored, and, for storage errors, the address of the suspected location is provided. Such automatic saving

avoids the need for programmed storing, which may be impossible because of the error condition. Because of the check-point capability in models with CPU retry, the interruption resulting from an exigent machine-check condition may identify a point in the recovery cycle which is prior to the point of error. For this reason a number of bits are stored to describe the validity of the status information and the relation between the points of error and interruption. Finally, extensive model-dependent logout information may be provided at permanently-assigned locations in main storage or in an area designated by an address in a control register.

Storage Validation: Since the block size for error correction may be larger than the bus width of the system, only part of the checking block may be replaced in any one CPU cycle. In the case of an uncorrectable storage error, such replacement cannot force valid checking block code on the entire storage block, as no information is available as to which part of the block is invalid. Furthermore, on some models validation of storage contents can be performed only when the entire "cache line" is replaced, which may include a number of checking blocks.

To permit validating storage, that is, replacing storage contents with a valid checking-block code, the instructions MOVE and MOVE LONG are defined to force valid checking-block code on the destination operand when the operand designation meets certain size and alignment requirements.

Machine Identification: The instruction STORE CPU ID provides information that identifies the particular CPU executing the instruction by type, model number, version, and serial number. It also provides the length of the model-dependent status and error-logout fields for this model. The instruction STORE CHANNEL ID provides analogous information for the addressed channel. These instructions make it possible to invoke model-dependent recovery programs and help a general-purpose analysis routine to record essential information about the physical unit for diagnostic and repair purposes.

Input/Output

System/370 architecture adds several facilities and functions in the area of input/output (I/O) operations to improve channel utilization, to make the control of operations more efficient and flexible, and to increase the maximum data rate on the I/O (channel-to-control-unit) interface. This section discusses some of the more important additions.

Utilization of Channel Facilities: The System/360 architecture provided for two channel types, a selector channel capable of operating with relatively high data rates but with only one device at a time, and a byte-multiplexer channel capable of simultaneously operating many devices but at relatively low data rates. System/370 adds the block-multiplexer[12] channel with both high-data-rate and multiple-device capabilities [7].

The block-multiplexer channel is similar to a byte-multiplexer channel in that both have a number of subchannels, each associated with an I/O device or a group of I/O devices. The subchannel is the logical entity that controls an I/O operation and contains the addresses, count, and control bits associated with the operation. The channel provides the data paths and controls for communicating with the CPU, main storage, and I/O control units and for associating the proper subchannel with each communications sequence. The main difference between the block- and byte-multiplexer channels is in the level of multiplexing: whereas the byte-multiplexer channel can interleave the transfer of individual bytes for different subchannels, the block-multiplexer channel, being designed for high data rates, is limited to interleaving complete blocks of data.

The block-multiplexing capability is particularly advantageous when used in conjunction with rotational position sensing on rotating-storage devices, such as disks and drums. This feature allows the device to disconnect from the channel during the period of rotational delay, thereby releasing the channel for operation with other devices. When the addressed sector is approaching on the track, reconnection is attempted for the transfer of data. In case the channel is so busy that the connection cannot be established by the time the sector is reached, another attempt is made after a delay of one rotation time.

Rotational position sensing is available, for example, on the IBM 2305 fixed head file. The control unit for this file can appear to have 16 devices, each associated with its own subchannel and able to sustain an I/O operation.

In the absence of the block-multiplexing capability, efficient utilization of I/O facilities required separate START I/O instructions to specify the position of the arm on the disk and the subsequent reading or writing. On the block-multiplexer channel, these commands are chained, thus avoiding the interruption of the CPU at the completion of the positioning operation. The number of START I/O instructions is also reduced.

Control: Since the periods when the block-multiplexer channel is busy transferring blocks of data are asynchronous to CPU operation, a new interruption, the channel-available interruption, is provided to indicate when the channel is free to process a CPU instruction. The block-multiplexer channel generates this signal when the busy condition ceases to exist that had previously caused rejection of an I/O instruction.

The new HALT DEVICE instruction also is introduced largely because of the block-multiplexer channel. It is similar to the previously available HALT I/O except that, when the channel is busy, only the operation on the addressed subchannel is affected. HALT I/O termi-

[12] The IBM 2880 Block-Multiplexer Channel included most of the System/370 I/O architecture extensions and was available on System/360 Models 85 and 195.

nates the current burst operation on the channel, ignoring the device address.

The new CLEAR I/O instruction is provided to permit freeing the subchannel associated with the addressed device without such freeing being contingent on the completion of the current I/O operation at the device. This function is useful for situations involving machine errors or reconfiguration of I/O devices and control units.

Finally, an extension is provided to reduce the CPU time to start an I/O operation. When START I/O (SIO) is issued, the channel signals the device as part of SIO execution to ascertain the device's ability to execute the command. This involves a number of signal sequences and the associated propagation delays and logic delays in the channel and the control unit. According to the I/O interface specification [11], the portion of the total delay introduced by the circuitry in the control unit can be as high as 32 microseconds. Additional delays may be introduced by the channel. On a CPU that can perform a few million average instructions per second, the delay due to the communications with the device can be equivalent to a hundred or more instruction executions.

The new instruction START I/O FAST RELEASE (SIOF) allows the acceptance to be signaled and the CPU to be released as soon as the channel has fetched the channel address word from main storage. The channel subsequently initiates the operation at the device and verifies the validity of the command information. Any exceptions are signaled by means of an interruption. Normally such exceptions are infrequent, and thus, overall, little time is spent processing the interruptions.

Some channels do not currently implement the early release on SIOF and instead execute SIOF as SIO. Such implementations are compatible and permit early conversion of programs to the use of SIOF.

Data Rates: The original System/360 I/O interface specification was adequate for data rates up to about 1M bytes per second. In special cases for disk devices and for very short channel cable lengths, a rate up to 1.25M bytes per second could be supported. With the advent of storage technologies employing higher recording densities, it was necessary to increase this limit. A higher limit was desirable also for certain buffered devices. Changes to System/360 were made in both the width of the interface and in the interface signaling protocols.

The fully interlocked signaling protocol on the System/360 I/O interface allowed one channel cable connection to sustain data transfer at a very wide range of rates, with both the channel and device having complete control over the timing of each byte transfer. It did, however, require an electrical signal to be propagated between the channel and the control unit four times for each byte transferred.

The System/370 channels modify this signaling protocol, with two additional wires in the interface, to provide the same level of transfer interlocks at the expense of only two propagation times per byte transferred. It depends on the control unit if the new facility is used, so that control units implemented to operate with the System/360 protocols can be attached to System/370 channels.

The basic interface bus is one byte wide, comprising eight data bits and a parity bit. On some System/370 models the bus width can be extended optionally to two bytes, thus doubling its data transfer capacity.

As a result of these two additions, the System/370 I/O interface can sustain a data transfer rate of over 1.5M bytes per second in the one-byte version and over 3.0M bytes per second in the two-byte version. Concurrently with the data rate improvement, the allowable cable lengths have been increased.

Implementation

While this paper is concerned mainly with the logical structure of the system as seen by the programmer, some of the parameters of the realizations are important for practical and efficient use of the equipment and to understand the motivation behind some of the features. This section summarizes some attributes of the System/370 models. For convenience of comparison, it includes also the corresponding values for the models of System/360. Only the most recent characteristics are listed; some of the models were improved after initial announcement.

Central Processing Units

Variation in the cycle time and data-flow width of the central processing unit (CPU) and in the characteristics of its control storage is one important way of obtaining cost and performance differences in a compatible family of machines. Table III shows these factors for the various models of System/360, and Table IV for System/370.[13]

Depending on the CPU, a different amount of "work" is accomplished per CPU cycle. Hence these numbers cannot be used directly as a measure of relative speed. CPU data-flow width is given in bytes and indicates the largest field that can be handled in one cycle time. Instruction fetches and a limited set of operations may be handled by wider paths, as indicated by footnotes.

Control storage, which contains the microprogram, is described in terms of the following attributes: capacity (in K words, where $K = 2^{10} = 1024$), word size (in bits), and cycle time (in nanoseconds) as used by the processor. The type of storage is also indicated: read-write (RW) or read-only (RO).

[13] In this paper, capacities and widths are usually given in bytes. A byte is comprised of eight bits. Physical implementations include additional bits for error detection and correction. This redundancy in CPU data flow and in processor storage typically is one bit per byte.

Table III. System/360 CPU and control storage characteristics.

Model	CPU		Control Storage				TLB
	Cycle (nsec)	Width (Bytes)	Capacity (K Words)	Wd Size (Bits)	Type (RW/RO)	Cycle (nsec)	Entries
22	750	1	4	50+5	RO	750	none
25	900	1	8	16+2	RW	900	none
30	750	1	4	50+5	RO	750	none
40	625	2[a]	4	52+2	RO	625	none
44	250	4	none				none
50	500	4	2.75	85+3[b]	RO	500	none
65	200	8	2.75	87+4[c]	RO	200	none
67	200	8	2.75	87+4[c]	RO	200	8
75	195	8	none				none
85	80	8	2	105+3[d]	RO	80	none
			0.5	105+3[d]	RW	80	
91	60	8	none				none
195	54	8	none				none

[a] Certain registers and paths are 17 or 18 bits wide where a main-storage address is processed in one cycle.
[b] Extended to 90+3 for the 1410 emulator, or 92+3 for the 7070 emulator.
[c] Extended to 94+4 when any emulator is installed.
[d] Extended to 122+4 when any emulator is installed.

Table IV. System/370 CPU and control storage characteristics.

Model	CPU		Control Storage				TLB
	Cycle (nsec)	Width (Bytes)	Capacity (K Words)	Wd Size (Bits)	Type (RW/RO)	Cycle (nsec)	Entries
115	480	1	20-28	20+2	RW	480	8
115-2	480	2	12-20[a]	19+3	RW	480	16
125	480	2	12-20	19+3	RW	480	16
125-2	320	2	16-24	19+3	RW	320	16
135	275 - 1485[b]	2[c]	12-24	16+2	RW	275	8
135-3	275 - 1485[b]	2[c]	64	16+2	RW	275	8
138	275 - 1430[b]	2[c]	64	16+2	RW	275	8
145	203 - 315[b]	4[d]	8-16[e]	32+4	RW	203	8
145-3	180 - 270[b]	4[d]	32	32+4	RW	180	8
148	180 - 270[b]	4[d]	32	32+4	RW	180	8
155	115	4	6	69+3	RO	115	none
155-II	115	4	8	69+3	RO	115	128
158	115[f]	4	8	69+3	RW	115	128
158-3	115[f]	4	8	69+3	RW	115	128
165	80	8	2	105+3	RO	80	none
			2	105+3[g]	RW	80	
165-II	80	8	4	105+3[g]	RO	80	128
			1	105+3[g]	RW	80	
168	80	8	4	105+3[g]	RO	80	128
			1	105+3[g]	RW	80	
168-3	80	8	4	105+3[g]	RO	80	128
			2	105+3[g]	RW	80	
195	54	8	none				none
3031	115[f]	4	8	69+3	RW	115	128
3032	80	8	4	105+3	RW	80	128
3033	58	8	4	105+3	RW	58	128

[a] The 115-2 contains a separate I/O processing unit for some functions that were executed on the CPU in a 115; hence the smaller CPU control storage capacity.
[b] Variable, depending on the type of operation performed.
[c] A 4-byte wide path is used for instruction fetch and for data access for some instruction types.
[d] An 8-byte wide path is used for instruction fetch.
[e] This capacity is physically a part of the main-storage array. Increments above 8K words subtract from the 145 processor-storage capacities listed in Table VI.
[f] 57.5 nsec for the execution of some instructions.
[g] Extended to 122+4 when any emulator is installed.

A range in the capacity is given for those models where the amount installed depends on the selection of certain optional features. The word size is expressed in terms of two numbers. The number before the plus sign is the number of bits used for logic or control purposes. The number after the plus sign is the number of additional bits used for checking the parity of the control-storage contents.

As explained in the section on virtual storage, the dynamic-address-translation mechanism includes a

Table V. System/360 processor storage characteristics.

Model	Processor Storage			Cache			
	Size (K Bytes)	Width (Bytes)	Cycle[a] (nsec)	Size (K Bytes)	Cycle (nsec)	Line Width (Bytes)	Assoc.
22	24 - 32	1	1500	none			
25	16 - 48	2	1800	none			
30	16 - 64	1	1500	none			
40	32 - 256	2	2500	none			
44	32 - 256	4	1000	none			
50	128 - 256	4	2000	none			
65	256 - 1024	8x2	750	none			
67	256 - 1024	8x2	750	none			
75	256 - 1024	8x4	750	none			
85	512 - 4096	16x4	960	16-32	80-160	256x4[b]	16
91	2048 - 6144	8x16	780	none			
195	1024 - 4096	8x16	756	32	54-162	8x8	4

[a] All models use magnetic-core technology.
[b] Blocks of 64 bytes (16x4) are fetched from main storage only if referenced.

Table VI. System/370 processor storage characteristics.

Model	Processor Storage			Cache			
	Size (K Bytes)	Width (Bytes)	Cycle (nsec)	Size (K Bytes)	Cycle (nsec)	Line Width (Bytes)	Assoc.
115	64 - 192	2	480	none			
115-2	64 - 384	2	480	none			
125	96 - 256	2	480	none			
125-2	96 - 512	2	480	none			
135	96 - 512	4	935	none			
135-3	256 - 512	4	880 R 935 W	none			
138	512 - 1024	4	880 R 935 W	none			
145	160 - 2048	8	540 R 608 W	none			
145-3	192 - 1984	8	405 R 540 W	none			
148	1024 - 2048	8	405 R 540 W	none			
155	256 - 2048	8	2070 [a]	8	115-230	16	2
155-II	256 - 2048	8	2070 [a]	8	115-230	16	2
158	512 - 6144	16	920 R 1035 W	8	115-230	16	2
158-3	512 - 6144	16	920	16	115-230	16x2	4
165	512 - 3072	8x4	2000 [a]	8-16	80-160	8x4	4
165-II	512 - 3072	8x4	2000 [a]	8-16	80-160	8x4	4
168	1024 - 8192	8x4	320	8-16	80-160	8x4	4-8[b]
168-3	1024 - 8192	8x4	320	32	80-160	8x4	8
195	1024 - 4096	8x16	756	32	54-162	8x8	4
3031	2048 - 6144	8x4	920	32	115-230	8x4	8
3032	2048 - 6144	8x4	320	32	80-160	8x4	8
3033	4096 - 8192	8x8	290	64	58-116	8x8	16

[a] Magnetic core
[b] Depends on cache size used.

translation lookaside buffer (TLB) to improve performance. The number of entries in this buffer is indicated in the last column.

Processor Storage

Another set of key attributes that distinguish various implementations is the size and speed of processor storage. Table V shows the options available for the System/360 models, and Table VI describes System/370. The range of sizes shows the smallest and largest total capacity available on that model. Intermediate values are usually also offered. The width is expressed in terms of two numbers: (basic width) x (interleaving factor). The basic width is the width of the data path from the storage controller to the instruction processor or channels. The interleaving factor indicates the number of accesses to sequential locations that can be made in one cycle. Thus, the storage of the Model 168 is implemented in four sections, each eight bytes wide. Each section contains every fourth doubleword, and their clocks are offset by 1/4 of the storage cycle time, so that the total available transfer rate for sequential locations is 8 x 4 = 32 bytes per 320-nsec cycle. For the 3033 the effective transfer rate is limited to eight bytes per CPU cycle. The cycle time shown is the minimum time between successive references to the same location.

Some models employ a high-speed buffer, referred to as the cache [8,14], to reduce the average access time to processor storage. The cache contains copies of recently accessed data in processor storage, and its existence is not apparent to the program.[14] The tables list the total cache size in K bytes. The two-number notation for the cycle time indicates the minimum time between successive read accesses and the total cache access time. The line-width column gives the number of bytes in the cache which are considered as one unit for addressing and replacement purposes. The first element of the product notation is the minimum transfer unit from processor storage to cache; the second element is the number of such transfer units required to make a line. A CPU instruction which is waiting for data may proceed as soon as the first unit has been transferred.

Usually, a particular virtual address may be represented in the cache in a subset of the available cache locations. The column labeled "Assoc." shows the number of different locations in the cache that may contain a particular virtual address. The set of virtual addresses that share a group of cache locations is known as an equivalence class. The replacement algorithm (usually LRU or a close variant) is executed separately for each equivalence class.

Announcement and Shipment Dates

Table VII lists the year and month when the various models of System/360 and System/370 were announced and first shipped.

Table VII. Announcement and shipment dates.

Model	Announced	First Shipped
	System/360 dates	
22	71-4	71-7
25	68-1	68-10
30	64-4	65-5
40	64-4	65-4
44	65-8	66-7
50	64-4	65-8
65	65-4	65-11
67	65-8	66-6
75	65-4	66-1
85	68-11	69-8
91	66-1	67-11
195	69-8	71-4
	System/370 dates	
115	73-3	74-3
115-2	75-11	76-4
125	72-10	73-4
125-2	75-11	76-2
135	71-3	72-5
135-3	72-8	73-8
138	76-6	76-11
145	70-9	71-8
145-3	72-8	73-8
148	76-6	77-1
155	70-6	71-2
158	72-8	73-4
158-3	75-3	76-9
165	70-6	71-4
168	72-8	73-8
168-3	75-3	76-6
195	71-6	73-5
3031	77-10	
3032	77-10	
3033	77-3	

Received May 1977; revised September 1977

References

1. Amdahl, G.M., Blaauw, G.A., and Brooks, F.P. Architecture of the IBM System/360. *IBM J. of Res.Develop. 8,2* (April 1964), 87–101.
2. Amdahl, G.M. The structure of System/360; Part III — processing unit design considerations. *IBM Syst. J. 3*, 2 (1964), 144–164.
3. Arden, B.W., Galler, B.A., O'Brien, T.C., and Westervelt, F.H. Program and addressing structure in a time-sharing environment. *J.ACM 13*, 1 (1966), 1–16.
4. Bell, G., and Strecker, W. D. Computer structures: What have we learned from the PDP-11? *Proc. 3rd Annual Symp. on Computer Architecture*, Jan. 1976, pp. 1–14. (available from ACM, New York)
5. Blaauw, G.A., and Brooks, F.P. The structure of System/360; Part I — outline of the logical structure. *IBM Syst. J. 3*, 2 (1964), 119–135.
6. Blaauw, G.A. The structure of System/360; Part V — multisystem organization. *IBM Syst. J. 3*, 2 (1964), 181–195.
7. Brown, D.T., Eibsen, R.L., and Thorn, C.A. Channel and direct access device architecture. *IBM Syst. J. 11*, 3 (1972), 186–199.

[14] This means that the cache does not appear in System/370 architecture, and the operation of the machine is completely described without reference to the cache. Although the cache is not architected, the decision not to do so is a significant architectural conclusion. It means that, except for performance considerations, the program can ignore the existence of the cache. On the other hand, the designer of the machine must ensure that in no case can the existence of the cache affect the logical appearance of the machine.

8. Conti, C.J., Gibson, D.H., and Pitkowsky, S.H. Structural aspects of the System/360 Model 85; Part I — general organization. *IBM Syst. J. 7,* 1 (1968), 2–14.
9. Dennis, J.B. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12,* 4 (Oct. 1965), 589–602.
10. Gibson, C.T. Time-Sharing in the IBM System/360: Model 67. *Proc. AFIPS 1966 SJCC, Vol. 28,* (AFIPS Press, Montvale, N.J., pp. 61–78.
11. IBM Corp. IBM System/360 and System/370 I/O interface channel to control unit original equipment manufacturer's information. *GA22-6974, IBM corp.*
12. IBM Corp. IBM System/370 principles of operation. *GA22-7000, IBM corp.*
13. Kilburn, T., Edwards, D.B.G., Lanigan, M.J., and Sumner, F.H. One-Level storage system. *IRE Trans. on Electron. Comptrs. 11* (1962), 223–235.
14. Liptay, J.S. Structural aspects of the System/360 Model 85; Part II — the cache. *IBM Syst. J. 7,* 1 (1968), 15–21.
15. Padegs, A. The structure of System/360; Part IV — channel design considerations. *IBM Syst. J. 3,* 2 (1964), 165–180.
16. Padegs, A. Structural aspects of the System/360 Model 85; Part III — extension to floating-point architecture. *IBM Syst. J. 7,* 1 (1968), 22–29.
17. Stevens, W.Y. The structure of System/360; Part II — system Implementations. *IBM Syst. J. 3,* 2 (1964), 136–142.

Professional Activities

Calls for Papers: Important Dates

15 January 1978. Trends and Applications 1978, "Distributed Processing," NBS, Gaithersburg, Md., May 18, 1978. Sponsor: IEEE-CS Washington, D.C. Chapter. See November *Communications.* 3c of 1000-word abstract to prog. chm: Ashok Agrawala, Dept. of Computer Science, University of Maryland, College Park, MD 20742. Notifications by Feb. 15; camera-ready copy due April 1.

15 January 1978. Second Conference of the European Cooperation in Informatics, Venice, Italy, Oct. 10-12, 1978. Sponsor: ECI. See July *Communications.* 5 c. of paper (max. 6000 words) and abstract to prog. chm: Peter Lockemann, Institut für Informatik II, Universität Karlsruhe, Postfache 6380, D-7500, Karlsruhe 1, W. Germany. Notifications by May 31. Proceedings.

15 January 1978. Computer Science and Statistics: 11th Annual Symposium on the Interface, North Carolina State University, Raleigh, N.C., March 6-7, 1978. Sponsor: University of North Carolina in cooperation with ACM and ASA statistical computing section. See December *Communications.* Submit papers to: Thomas M. Gerig, Institute of Statistics, N.C. State University, Box 5457, Raleigh, NC 27607. Proceedings.

16 January 1978. Conference on Information Sciences and Systems, Johns Hopkins University, Baltimore, MD., March 29-31, 1978. See November *Communications.* Title and summary of "regular" or "short" paper to: 1978 CISS, Department of Electrical Engineering, Johns Hopkins University, Baltimore, MD 21218. Notifications by February 17. Proceedings.

18 January 1978. Ninth Southeastern Conference on Combinatorics, Graph Theory and Computing, Florida Atlantic University, Boca Raton, Fla., Jan., 30-Feb. 2, 1978. Sponsor: Florida Atlantic University. Abstract (10-20 single spaced lines) and paper to: Frederick Hoffman, Dept. of Mathematics, Florida Atlantic University, Boca Raton, FL 33431; after conference to: Ronald C. Mullin, Dept. of Combinatorics and Optimization, University of Waterloo, Waterloo, Ont., Canada N2L 3G1. Final text due May 1. Proceedings.

20 January 1978. AI Conference, Hamburg, West Germany, July 18-20, 1978. Sponsors: AISB, Gesellschaft für Informatik. For details of paper format contact prog. chm: Derek Sleeman, Dept. of Computer Studies, The University, Leeds LS2 9JT, United Kingdom.

31 January 1978. International Conference on Interactive Techniques in Computer Aided Design, Palazzo dei Congresso, Bologna, Italy, Sept. 21-23, 1978. Sponsors: ACM Italian Chapter, IEEE-CS, AICA Working Group on Design Automation, and other organizations, under patronage of University of Bologna. See December *Communications.* 5c. of paper (1000-5000 words) to prog. chm: Umberto Cugini, Instituto di Disegno di Macchine, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milan, Italy; or N. Amer. coord: Ira Cotton, Inst. Computer Science and Technology, NBS, Washington, DC 20234. Notifications by April 15. Proceedings.

31 January 1978. 17th Annual Technical Symposium, "Tools for Improved Computing in the 80's," NBS, Gaithersburg, Md., June 15, 1978. Sponsors: ACM Washington, D.C. Chapter, NBS. Submit papers to prog. chm: Bryce Elkins, Computer Sciences Corp., 400 Army-Navy Drive, Arlington, VA 22202.

1 February 1978. Simulation, Modeling, and Decision in Energy Systems, Montreal, Canada, June 1-2, 1978. Sponsors: IASTED. 3c. of abstract (200-250 words) to: M.B. Carver, Atomic Energy of Canada Ltd., Chalk River, Ontario KOJ 1JO, Canada. Final papers due May 15.

1 February 1978. Personal Computing Festival, in conjunction with 1978 National Computer Conference, Anaheim, Calif., June 6-8, 1978 (NCC is June 5-8). Sponsor: AFIPS. See November *Communications.* Submit letter of intent including abstract to: Jim C. Warren Jr., Star Route Box 111, Redwood City, CA 94062; 415 851-7664. Prospective session chairmen should send 2c. of abstract (250 words) including title and scope of proposed sessions. Notifications to session chairmen by Feb. 10; final deadline for authors March 15. Papers will be published in *Festival Digest '78,* to be available at NCC.

1 February 1978. First Annual Symposium on Small Systems, New York City, Aug. 2-3, 1978. Sponsor: ACM SIGMINI. See November *Communications.* 4c. of paper (max. 15 pages) to prog. chm: George W. Gorsline, Dept. of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061. Notifications by March 15, Proceedings.

1 February 1978. Symposium on Computer Films for Research in Physics and Chemistry, University of Colorado, Boulder, Colo., March 20-22, 1978. Sponsors: American Physical Society Div. of Electron and Atomic Physics, with cooperation of American Chemical Society Div. of Physical Chemistry and of Computer Chemistry. See January *Communications.* Short abstract and one-paragraph description of the associated film J.H. Eberly/D.G. Hummer, SCFRPAC, JILA, University of Colorado, Boulder, CO 80309.

15 February 1978. MECO 78, International Symposium on Measurement and Control, Athens, Greece, June 26-29, 1978. Sponsors: International Association of Science and Technology Development, Panhellenic Society of Mechanical and Electrical Engineers. 3c. of 200-250 word English abstract to: MECO 78, The Panhellenic Society of Mechanical and Electrical Engineers, 26 Bouboulinas Str., Athens 147, Greece. Notifications by April 1; paper deadline June 15.

20 February 1978. First Annual Conference on Modular Computers, Brookhaven National Laboratory, Upton, N.Y., September 27-29, 1978. Sponsors: Brookhaven National Laboratory, SUNY at Stony Brook. ACM SIGPLAN. See January *Communications.* 4c. of paper (3000-6000 words) to prog. chm: Richard Kieburtz, Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11970. Notifications by April 10. Proceedings.

24 February 1978. International Symposium on Operating Systems, IRIA, Paris, Oct. 2-5, 1978. Sponsors: Carnegie-Mellon University, Institute de Recherche d'Informatique et d'Automatique. 5c of paper plus title and short abstract in French or English to: D. Lanciaux, IRA, BP 105, 78105, Le Chesnay, France.

1 March 1987. 3rd USA-Japan Computer Conference, San Francisco, Calif., Oct. 10-12, 1978. Sponsors: AFIPS, Information Processing Society of Japan. Complete draft (max. 5000 words) and abstract (150 words) to: Edward J. McCluskey, Digital Systems Laboratory, Stanford University, Stanford, CA 94305; 415 497-1451. Proceedings.

1 March 1978. Fourth International Conference on Very Large Data Bases, Berlin, Germany, Sept. 13-15, 1978. Sponsors: ACM SIGMOD, SIGIR, SIGBDP, IEEE-CS, SMIS, IFIP. See December *Communications.* 5c. of full paper to U.S. prog. comm. chm: S. Bing Yao, Computer Applications and Information Systems, New York University, 40 West 4th St., New York, NY 10003; or European prog. comm. chm: Janis A. Bubenko Jr., Chalmers University of Technology, Dept. of Computer Sciences, Fack, S-402 20 Goteborg, Sweden. Notifications by May 18; final papers due June 15. Proceedings.

31 March 1978. Fourth Workshop on Computer Architecture and Non-Numeric Processing, Syracuse University Minnowbrook Conference Center, Blue Mountain Lake, N.Y. Sponsors: ACM SIGARCH, SIGIR, SIGMOD, in cooperation with IEEE-CS TCARCH, TCDBE, and Syracuse University. See January *Communications.* 4c. of paper (20 pages) and abstract (100 words) or extended abstract (min. 1000 words) to prog. chm: Lee A. Holaar, Dept. of Computer Science, University of Ilinois Urbana, IL 61820; 217 333-3162. Notifications by May 20. Proceedings.

15 April 1978. Third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif., Aug. 29-31, 1978. Sponsors: Lawrence Berkeley Laboratory, University of California, U.S. Dept. of Energy. Regular papers (max. 5000 words), short papers on abstracts (max. 1500 words); abstracts only for panel discussions to prog. co-chm: Steve Kimbleton, National Bureau of Standards, B-212 Technology Bldg., Washington, DC 20234. Final papers due May 1; Notifications by June 15.

1 July 1978. ACM 78, Sheraton Park Hotel, San Francisco, Calif., Dec. 4-6, 1978. Sponsor: ACM. See January *Communications.* 5c of paper and/or session proposal to prog. co-chm: Gerald L. Engel, Dept. of Mathematics and Computing Sciences, Old Dominion University, Norfolk, VA 23508; 804 489-6524; or Dennis M. Conti, Systems and Software Division, NBS, Washington, DC 20234; 301 921-3485. Proceedings.

1 October 1978. Second International Conference on Computational Methods in Nonlinear Mechanics, University of Texas, Austin, Tex., March 26-28, 1979. Sponsor: Texas Institute for Computational Mechanics. Submit abstracts (300-1000 words) to: TICOM Conference, TICOM ENL Bldg. 305, The University of Texas, Austin, TX 78712. Proceedings.

Calendar of Events

ACM's calendar policy is to list open computer science meetings that are held on a not-for-profit basis. Not included in the calendar are educational seminars institutes, and courses. Submittals should be substantiated with name of the sponsoring organization, fee schedule, and chairman's name and full address.

One telephone number contact for those interested in attending a meeting will be given when a number is specified for this purpose.

All requests for ACM sponsorship or cooperation should be addressed to Chairman, Conferences and Symposia Committee, Seymour J. Wolfson, 643 MacKenzie Hall, Wayne State University, Detroit, MI 48202, with a copy to Louis Fiora, Conference Coordinator, ACM Headquarters. For European events, a copy of the request should also be sent to the European Representative. Technical Meeting Request Forms for this purpose can be obtained from ACM Headquarters or from the European Regional Representative. Lead time should include 2 months (3 months if for Europe) for processing of the request, plus the necessary months (minimum 2) for any publicity to appear in *Communications.*

Events for which ACM or a subunit of ACM is a sponsor or collaborator are indicated by ■. Dates precede titles.

In this issue the calendar is given to May 1978. New Listings are shown first; they will appear next month as Previous Listings. A full listing is in the November 1977 Communications.

NEW LISTINGS

20-22 March 1978
Symposium on Computer Films for Research in Physics and Chemistry, University of Colorado, Boulder, Colo. Sponsors: American Physical Society Div. of Electron and Atomic Physics, with cooperation of American Chemical Society Div. of Physical Chemistry and of Computer Chemistry. Symp. chm: D.G. Hummer, JILA, University of Colorado, Boulder, CO 80309.

3-4 April 1978
Emerging Patterns in Automatic Imagery Pattern Recognition, NBS, Gaithersburg, Md. Sponsors: NBS, Electronic Industries Association. Contact: Russell Kirsch, A317 Administration Building, National Bureau of Standards, Washington, DC 20234; 301 921-2337.

21-22 April 1978
Eleventh Annual Small College Computing Symposium, St. Cloud State University, St. Cloud, Minn. Sponsor: St. Cloud State University. Symp. co-chm: Randal Kolb, Director, Academic Computer Services, St. Cloud State University, St. Cloud, MN 56301; 612 255-4103.

22-24 May 1978
7th ASIS Mid-Year Meeting, Rice University, Houston, Tex. Sponsor: American Society

(Calendar continued on p. 102)

The 8K buffer is shown in Figure 10.15.4. It contains 64 columns of 128 bytes each. Every buffer column is subdivided into four blocks. A block is 32 bytes and can contain 32 consecutive bytes from processor storage that are on a 32-byte boundary. The 8K buffer can contain a maximum of 256 different blocks of processor storage data (4 blocks per column times 64 columns). A valid trigger is associated with each buffer block and is set to indicate whether or not the block contains valid data. All valid triggers are set off during system reset or IPL.

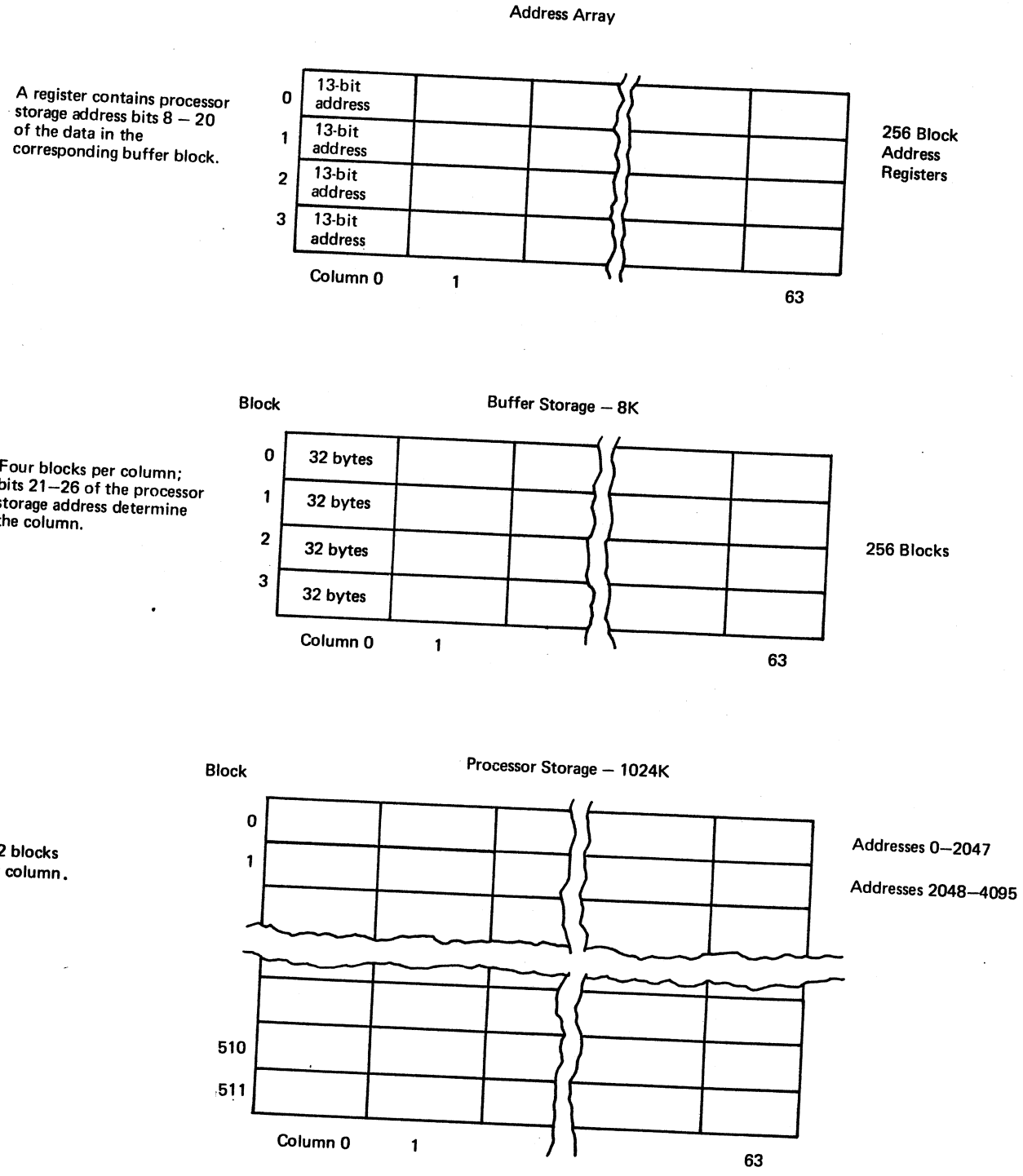


Figure 10.15.4. 8K buffer organization

G. M. Amdahl
G. A. Blaauw
F. P. Brooks, Jr.,

Architecture of the IBM System/360

Abstract: The architecture* of the newly announced IBM System/360 features four innovations:

1. An approach to storage which permits and exploits very large capacities, hierarchies of speeds, read-only storage for microprogram control, flexible storage protection, and simple program relocation.

2. An input/output system offering new degrees of concurrent operation, compatible channel operation, data rates approaching 5,000,000 characters/second, integrated design of hardware and software, a new low-cost, multiple-channel package sharing main-frame hardware, new provisions for device status information, and a standard channel interface between central processing unit and input/output devices.

3. A truly general-purpose machine organization offering new supervisory facilities, powerful logical processing operations, and a wide variety of data formats.

4. Strict upward and downward machine-language compatibility over a line of six models having a performance range factor of 50.

This paper discusses in detail the objectives of the design and the rationale for the main features of the architecture. Emphasis is given to the problems raised by the need for compatibility among central processing units of various size and by the conflicting demands of commercial, scientific, real-time, and logical information processing. A tabular summary of the architecture is shown in the Appendices.

Introduction

The design philosophies of the new general-purpose machine organization for the IBM System/360 are discussed in this paper.† In addition to showing the architecture* of the new family of data processing systems, we point out the various engineering problems encountered in attempts to make the system design compatible, at the program bit level, for large and small models. The compatibility was to extend not only to models of any size but also to their various applications—scientific, commercial, real-time, and so on.

The section that follows describes the objectives of the new system design, i.e., that it serve as a base for new technologies and applications, that it be general-purpose, efficient, and strictly program compatible in all models. The remainder of the paper is devoted to the design problems faced, the alternatives considered, and the decisions made for data format, data and instruction codes, storage assignments, and input/output controls.

Design objectives

The new architecture builds upon but differs from the designs that have gradually evolved since 1950. The evolution of the computer had included, besides major technological improvements, several important systems concepts and developments:

* The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation.

† Additional details concerning the architecture, engineering design, programming, and application of the IBM System/360 will appear in a series of articles in the *IBM Systems Journal*.

1. Adaptation to business data processing.

2. Growing importance of the total system, especially the input/output aspects.

3. Universal use of assembly programs, compilers, and other metaprograms.

4. Development of magnetic recording on tapes, drums, and disks.

5. Hundred-fold expansion of storage capacities.

6. Adaptation for real-time systems.

During this period most new computer models, from the point of view of their logical structure, were improved, enlarged, or technologically recast versions of the machines developed in the early 1950's. IBM products are not atypical; the evolution has gone from IBM 701 to 7094, 650 to 7074, from 702 to 7080, and from 1401 to 7010.

The system characteristics to be described here, however, are a new approach to logical structure and function, designed for the needs of the next decade as a coordinated set of data processing systems.

• *Advanced concepts*

It was recognized from the start that the design had to embody recent conceptual advances, and hence, if necessary, be incompatible with existing products. To this end, the following premises were considered:

1. Since computers develop into families, any proposed design would have to lend itself to growth and to successor machines.

2. Input/output (I/O) devices make systems specifically useful for given applications. A general method was needed for using I/O devices differing in data rate, access, and function.

3. The real value of an information system is properly measured by answers-per-month, not bits-per-microsecond. The former criterion required specific advances to increase throughput for a given internal speed, to shorten turnaround time for a given throughput, and to make the whole complex of machines and programming systems easier to use.

4. The functions of the central processing unit (CPU) proper are specific to its application only a minor fraction of the time. The functions required by the system for its own operation, e.g., compiling, input/output management, and the addressing of and within complex data structures, use a major share of time. These functions had to be made efficient, and need not be different in machines designed for different applications.

5. The input/output channel and the input/output control program had to be designed for each other.

6. Machine systems had to be capable of supervising themselves, without manual intervention, for both real-time and multiprogrammed, or time-shared, applications. To realize this capability requires: a comprehensive interruption system, tamper-proof storage protection, a protected supervisor program, supervisor-controlled program switching, supervisor control of all input/output (including unit assignment), nonstop operation (no HALT), easy program relocation, simple writing of read-only or unmodified programs, a timer, and interpretive consoles.

7. It must be possible and straightforward to assemble systems with redundant I/O, storages, and CPU's so that the system can operate when modules fail.

8. Storage capacities of more than the commonly available 32,000 words would be required.

9. Certain types of problems require floating-point word length of more than 36 bits.

10. As CPU's become increasingly reliable, built-in thorough checking against hardware malfunction is imperative for all systems, regardless of application.

11. Since the largest servicing problem is diagnosis of malfunction, built-in hardware fault-locating aids are essential to reduce down-times. Furthermore, identification of individual malfunctions and of individual invalidities in program syntax would have to be provided.

• *Open-ended design*

The new design had to provide a dependable base for a decade of customer planning and customer programming, and continuing laboratory developments, whether in technology, application and programming techniques, system configuration, or special requirements.

The various circuit, storage, and input/output technologies used in a system change at different times, causing corresponding changes in their *relative* speeds and costs. To take advantage of these changes, it is desirable that the design permit asynchronous operation of these components with respect to each other.

Changing application and programming techniques would require open-endedness in function. Current trends had to be extrapolated and their consequences anticipated. This anticipation could be achieved by direct provision, e.g., by increasing storage capacities and by using multiple-CPU systems, various new I/O devices, and time sharing. Anticipation might also take the form of generalization of function, as in code-independent scan and translation facilities, or it might consist of judiciously reserving spare bits, operation codes, and blocks of operation codes, for new modes, operations, or sets of operations.

Changing requirements for system configuration would demand not only such approaches as a standard interface between I/O devices and control unit, but also capabilities for a machine to directly sense, control, and respond to other equipment modules via paths outside the normal data routes. These capabilities permit the construction of supersystems that can be dynamically reconfigured under program control, to adapt more precisely to specialized functions or to give graceful degradation.

In many particular applications, some special (and often minor) modification enhances the utility of the system. These modifications (RPQ's), which may correct some shortsightedness of the original design, often embody operations not fully anticipated. In any event, a good general design would obviate certain modifications and accommodate others.

- *General-purpose function*

The machine design would have to provide individual system configurations for large and small, separate and mixed applications as found in commercial, scientific, real-time, data-reduction, communications, language, and logical data processing. The CPU design would have to be facile for each of these applications. Special facilities such as decimal or floating-point arithmetic might be required only for one or another application class and would be offered as options, but they would have to be integral, from the viewpoint of logical structure, with the design.

In particular, the general-purpose objective dictated that:

1. Logical power of great generality would have to be provided, so that all combinations of bits in data entities would be allowed and might be manipulated with operations whose power and utility depend upon the general nature of representations rather than upon any specific selection of them.

2. Operations would have to be code-independent except, of course, where code definition is essential to operation, as in arithmetic. In particular, all bit combinations should be acceptable as data; no combination should exert any control function when it appears in a data stream.

3. The individual bit would have to be separately manipulatable.

4. The general addressing system would have to be able to refer to small units of bits, preferably the unit used for characters.

Further, the implications of general-purpose CPU design for communications-oriented systems indicated a radical departure from current systems philosophy. The conventional CPU, for example, is augmented by an independent stored-program unit (such as the IBM 7750 or 7740) to handle all communications functions. Since the new CPU would easily perform such *logical* functions as code translation and message assembly, communications lines would be attached directly to the I/O channel via a control unit that would perform only character assembly and the electrical line-handling functions.

- *Efficient performance*

The basic measure of a good design is high performance in comparison to other designs having the same cost. This measure cannot be ignored in designing a compatible line. Hence each individual model and systems configuration in the line would have to be competitive with systems that are specialized in function, performance level or both. That this goal is feasible in spite of handicaps introduced by the compatibility requirement was due to the especially important cost savings that would be realized due to compatibility.

- *Intermodel compatibility*

The design had to yield a range of models with internal performance varying from approximately that of the IBM 1401 to well beyond that of the IBM 7030 (STRETCH). As already mentioned, all models would have to be strictly program compatible, upward and downward, at the program bit level.

The phrase "strictly program compatible" requires a more technically precise definition. Here it means that a valid program, whose logic will not depend implicitly upon time of execution and which runs upon configuration **A**, will also run on configuration **B** if the latter includes at least the required storage, at least the required I/O devices, and at least the required optional features. Invalid programs, i.e., those which violate the programming manual, are not constrained to yield the same results on all models. The manual identifies not only the results of all dependable operations, but also those results of exceptional and/or invalid operations that are not dependable. Programs dependent on execution-time will operate compatibly if the dependence is explicit, and, for example, if completion of an I/O operation or the timer are tested.

Compatibility would ensure that the user's expanding needs be easily accommodated by any model. Compatibility would also ensure maximum utility of programming support prepared by the manufacturer, maximum sharing of programs generated by the user, ability to use small systems to back up large ones, and exceptional freedom in configuring systems for particular applications.

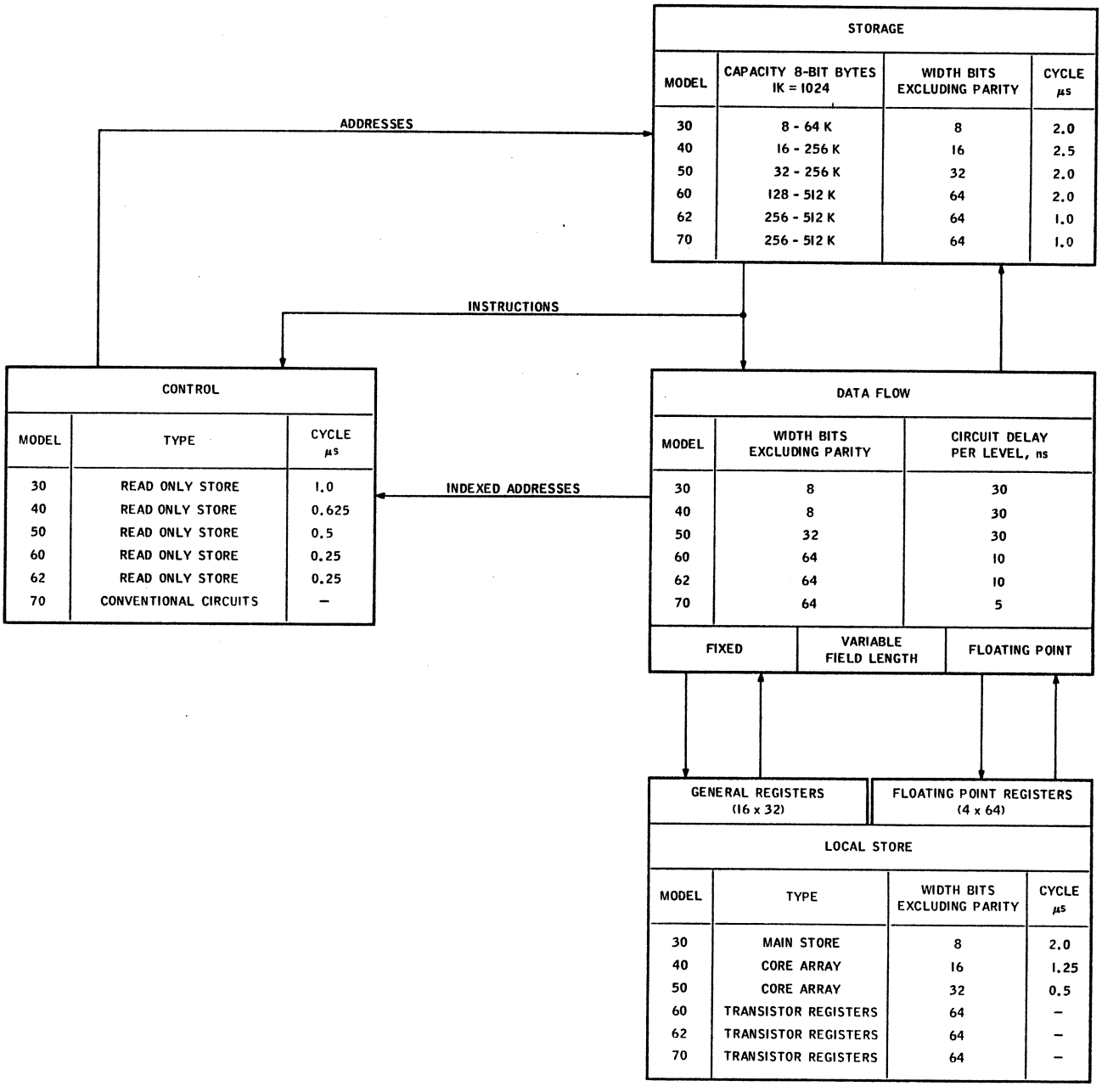
It required a new concept and mode of thought to make the compatibility objective even conceivable. In the last few years, many computer architects had realized, usually implicitly, that logical structure (as seen by the programmer) and physical structure (as seen by the engineer) are quite different. Thus each may see registers, counters, etc.,

that to the other are not at all real entities. This was not so in the computers of the 1950's. The *explicit* recognition of the duality of structure opened the way for the compatibility within System/360. The compatibility requirement dictated that the basic architecture had to embrace different technologies, different storage-circuit speed ratios, different data path widths, and different data-flow complexities. The basic machine structure and implementation at the various performance levels are shown in Fig. 1.

The design decisions

Certain decisions for the architectural design became mileposts, because they (a) established prominent characteristics of the System/360, (b) resolved problems concerning the compatibility objective, thus illuminating the essential differences between small models and large, or (c) resolved problems concerning the general-purpose objective, thus illuminating the essential differences among applications. The sections that follow discuss these de-

Figure 1 **Machine structure and implementation.**



cisions, the problems faced, the alternatives considered, and the reasons for the outcome.

- *Data format*

The decision on basic format (which affected character size, word size, instruction field, number of index registers, input-output implementation, instruction set layout, storage capacity, character code, etc.) was whether data length modules should go as 2^n or 3.2^n. Even though many matters of format were considered in the basic choice, we will for convenience treat the major components of the decision as if they were independent.

Character size, 6 vs 4/8. In character size, the fundamental problem is that decimal digits require 4 bits, the alphanumeric characters require 6 bits. Three obvious alternatives were considered — 6 bits for all, with 2 bits wasted on numeric data; 4 bits for digits, 8 for alphanumeric, with 2 bits wasted on alphanumeric; and 4 bits for digits, 6 for alphanumeric, which would require adoption of a 12-bit module as the minimum addressable element. The 7-bit character, which incorporated a binary recoding of decimal digit pairs, was also briefly examined.

The 4/6 approach was rejected because (a) it was desired to have the versatility and power of manipulating character streams and addressing individual characters, even in models where decimal arithmetic is not used, (b) limiting the alphabetic character to 6 bits seemed short-sighted, and (c) the engineering complexities of this approach might well cost more than the wasted bits in the character.

The straight-6 approach, used in the IBM 702–7080 and 1401–7010 families, as well as in other manufacturers' systems, had the advantages of familiar usage, existing I/O equipment, simple specification of field structure, and commensurability with a 48-bit floating-point word and a 24-bit instruction field.

The 4/8 approach, used in the IBM 650–7074 family and elsewhere, had greater coding efficiency, spare bits in the alphabetic set (allowing the set to grow), and commensurability with a 32/64-bit floating-point word and a 16-bit instruction field. Most important of these factors was coding efficiency, which arises from the fact that the use of numeric data in business records is more than twice as frequent as alphanumeric. This efficiency implies, for a given hardware investment, better use of core storage, faster tapes, and more capacious disks.

Floating-point word length, 48 vs 32/64. For large models addition time goes up slowly with word length, and multiplication time rises almost linearly. For small, serial models, addition time rises linearly and multiplication as the square of word length. Input/output time for data files rises linearly. Large machines more often require high precision; small machines more urgently require short operands. For this aspect of the basic format problem, then, definite conflicts arose because of compatibility.

Good data were unavailable on the distribution of required precision by the number of problems or running time. Indeed, accurate measures could not be acquired on such coarse parameters as frequency of double-precision operation on 36-bit and 48-bit machines. The question became whether to force all problems to the longer 48-bit word, or whether to provide 64 to take care of precision-sensitive problems adequately, and either 32 or 36 to give faster speed and better coding efficiency for the rest. The choice was made for the IBM System/360 to have both 64- and 32-bit length floating point. This choice offers the user the option of making the speed/space vs precision trade-off to best suit his requirements. The user of the large models is expected to employ 64-bit words most of the time. The user of the smaller models will find the 32-bit length advantageous in most of his work. All floating-point models have both lengths and operate identically.

Hexadecimal floating-point radix. With no conflicts in questions of large vs small machines, base 16 was selected for floating point. Studies by Sweeney[1] show that the frequency of pre-shift, overflow, and precision-loss post-shift on floating-point addition are substantially reduced by this choice. He has shown that, compared with base 2, the percentage frequency of occurrence of overflow is 5 versus 20, pre-shift is 43 versus 58, and precision-loss post-shift is 11 versus 18. Thus speed is noticeably enhanced. Also, simpler shifting paths, with fewer logic levels, will accomplish a higher proportion of all required pre-shifting in a single pass. For example, circuits shifting 0, 1, 2, 3, or 4 binary places cover 82% of the base 2 pre-shifts. Substantially simpler circuits shifting 0, 1, or 2 hexadecimal places cover 93% of all base 16 pre-shifts. This simplification yields higher speed for the large models and lower cost for the small ones.

The most substantial disadvantage of adopting base 16 is the shift in bit usage from exponent to fraction. Thus, for a given range and a given *minimum* precision, base 16 requires 2 fewer exponent bits and 3 more fraction bits than does base 2. Alternatively and equivalently, rounding and truncation effects are 8 times as large for a given fraction length. For the 64-bit length, this is no problem. For the 32-bit length, with its 24-bit fraction, the minimum precision is reduced to the equivalent of 21 bits. Because the 64-bit length was available for problems where the minimum precision cramped the user, the greater speed and simplicity of base 16 was chosen.

Significance arithmetic. Many schemes yielding an estimate of the significance of computed results have been proposed. One such scheme, a modified form of unnormalized arithmetic, was for a time incorporated in the design. The scheme was finally discarded when simulation runs showed this mode of operation to cost about one hexadecimal digit of actual significance developed, as compared with normalized operation. Furthermore, the

significance estimate yielded for a given problem varied substantially with the test data used.

Sign representations. For the fixed-point arithmetic system, which is binary, the two's complement representation for negative numbers was selected. The well-known virtues of this system are the unique representation of zero and the absence of recomplementation. These substantial advantages are augmented by several properties especially useful in address arithmetic, particularly in the large models, where address arithmetic has its own hardware. With two's complement notation, this indexing hardware requires no true/complement gates and thus works faster. In the smaller, serial models, the fact that high-order bits of address arithmetic can be elided without changing the low-order bits also permits a gain in speed. The same truncation property simplifies double-precision calculations. Furthermore, for table calculation, rounding or truncation to an integer changes all variables in the same direction, thus giving a more acceptable distribution than does an absolute-value-plus-sign representation.

The established commercial rounding convention made the use of complement notation awkward for decimal data; therefore, absolute-value-plus-sign is used here. In floating point, the engineering virtues of normalizing only high-order zeros, and of having all zeros represent the smallest possible number, decided the choice in favor of absolute-value-plus-sign.

Variable- versus fixed-length decimal fields. Since the fields of business records vary substantially in length, coding efficiency (and hence tape speed, file capacity, CPU speed, etc.) can be gained by operating directly on variable-length fields. This is easy for serial-by-byte machines, and the IBM 1401–7010 and 702–7080 families are among those so designed. A less flexible structure is more appropriate for a more parallel machine, and the IBM 650–7074 family is among those designed with fixed-word-length decimal arithmetic.

As one would expect, the storage efficiency advantage of the variable data format is diminished by the extra instruction information required for length specification. While the fixed format is preferable for the larger machines, the variable format was adopted because (a) the small commercial users are numerous and only recently trained in variable-format concepts, and (b) the large commercial system is usually I/O limited; hence the internal performance disadvantage of the variable format is more than compensated by the gain in effective tape rate.

Decimal accumulators versus storage-storage operation. A closely related question involving large/small models concerned the use of an accumulator as one of the operands on decimal arithmetic, versus the use of storage locations for all operands and results. This issue is pertinent even after a decision has been made for variable-length fields in storage; for example, it distinguishes IBM 702–7080 arithmetic from that of the IBM 1401–7010 family.

The large models readily afford registers or local stores and get a speed enhancement from using these as accumulators. For the small model, using core storage for logical registers, addition to an accumulator is no faster than addition to a programmer-specified location. Addition of two arbitrary operands and storage of the result becomes LOAD, ADD, STORE, however, and this operation is substantially slower for the small models than the MOVE, ADD sequence appropriate to storage-storage operation. Business arithmetic operations (as hand coded and especially as compiled from COBOL) often take this latter form and rarely occur in strings where intermediate results are profitably held in accumulators. In address arithmetic and floating-point arithmetic, quite the opposite is true.

Field specification: word-marks versus length. Variable-length fields can be specified in the data via delimiter characters or word-marks, or in the instruction via specification of field length or start-finish limits. For business *data*, the word-mark has some slight advantage in storage efficiency: one extra bit per 8-bit character would cost less than 4 extra length bits per 16-bit address. Furthermore, instructions, and hence addresses, usually occupy most core storage space in business computers. However, the word-mark approach implies the use of word-marks on instructions, too, and here the cost is without compensating function. The same is true of all fixed-field data, an important consideration in a general-purpose design. On balance, storage efficiency is about equal; the field specification was put in the instruction to allow all data combinations to be valid and to give easier and more direct programming, particularly since it provides convenient addressing of parts of fields. Length was chosen over limit specification to simplify program relocation and instruction modification.

ASCII vs BCD codes. The selection of the 8-bit character size in 1961 proved wise by 1963, when the American Standards Association adopted a 7-bit standard character code for information interchange (ASCII). This 7-bit code is now under final consideration by the International Standards Organization for adoption as an international standards recommendation. The question became "Why not adopt ASCII as the *only* internal code for System/360?"

The reasons against such exclusive adoption was the widespread use of the BCD code derived from and easily translated to the IBM card code. To facilitate use of both codes, the central processing units are designed with a high degree of code independence, with generalized code translation facilities, and with program-selectable BCD or ASCII modes for code-dependent instructions. Neverthe-

Figure 2a **Extended binary-coded-decimal (BCD) interchange code.**

Bit Positions → 01 / 00 / 23 / 4567	00	01	10	11
0000	NULL			
0001				
0010				
0011				
0100	PF	RES	BYP	PN
0101	HT	NL	LF	RS
0110	LC	BS	EOB	UC
0111	DEL	IDL	PRE	EOT
1000				
1001				
1010				
1011				
1100				
1101				
1110				
1111				

01 / 00 / 23 / 4567	00	01	10	11
0000	BLANK	&	-	
0001			/	
0010				
0011				
0100				
0101				
0110				
0111				
1000				
1001	.		,	"
1010	?	!		:
1011	.	$	,	#
1100	←	*	%	@
1101	(	)	‿	'
1110	+	;	–	=
1111	‡	¢	±	✓

10 / 00 / 23 / 4567	00	01	10	11
0000				
0001	a	j		
0010	b	k	s	
0011	c	l	t	
0100	d	m	u	
0101	e	n	v	
0110	f	o	w	
0111	g	p	x	
1000	h	q	y	
1001	i	r	z	
1010				
1011				
1100				
1101				
1110				
1111				

11 / 00 / 23 / 4567	00	01	10	11
0000	>	<	‡	0
0001	A	J		1
0010	B	K	S	2
0011	C	L	T	3
0100	D	M	U	4
0101	E	N	V	5
0110	F	O	W	6
0111	G	P	X	7
1000	H	Q	Y	8
1001	I	R	Z	9
1010				
1011				
1100				
1101				
1110				
1111				

Figure 2b **8-bit representation of the 7-bit American Standard Code for Information Interchange (ASCII).**

Bit Positions → 76 / 00 / X5 / 4321	00	01	10	11
0000	NULL	DC_0		
0001	SOM	DC_1		
0010	EOA	DC_2		
0011	EOM	DC_3		
0100	EQT	DC_4 STOP		
0101	WRU	ERR		
0110	RU	SYNC		
0111	BELL	LEM		
1000	BKSP	S_0		
1001	HT	S_1		
1010	LF	S_2		
1011	VT	S_3		
1100	FF	S_4		
1101	CR	S_5		
1110	SO	S_6		
1111	SI	S_7		

01 / 00 / X5 / 4321	00	01	10	11
0000	BLANK	0		
0001	!	1		
0010	"	2		
0011	#	3		
0100	$	4		
0101	%	5		
0110	&	6		
0111	'	7		
1000	(	8		
1001	)	9		
1010	*	:		
1011	+	;		
1100	,	<		
1101	-	=		
1110	.	>		
1111	/	?		

10 / 00 / X5 / 4321	00	01	10	11
0000			@	P
0001			A	Q
0010			B	R
0011			C	S
0100			D	T
0101			E	U
0110			F	V
0111			G	W
1000			H	X
1001			I	Y
1010			J	Z
1011			K	[
1100			L	\
1101			M	]
1110			N	↑
1111			O	←

11 / 00 / X5 / 4321	00	01	10	11
0000				P
0001			a	q
0010			b	r
0011			c	s
0100			d	t
0101			e	u
0110			f	v
0111			g	w
1000			h	x
1001			i	y
1010			j	z
1011			k	
1100			l	
1101			m	
1110			n	ESC
1111			o	DEL

less, a choice had to be made for the code-sensitive I/O devices and for the programming support, and the solution was to offer both codes, fully supported, as a user option. Systems with either option will, of course, easily read or write I/O media with the other code. The extended BCD interchange code and an 8-bit representation of the 7-bit ASCII are shown in Fig. 2.

Boundary alignment. A major compatibility problem concerned alignment of field boundaries. Different models were to have different widths of storage and data flow, and therefore each model had a different set of preferences. For the 8-bit wide model the characters might have been aligned on character boundaries, with no further constraints. In the 64-bit wide model it might have been preferred to have no fields split between different 64-bit double-words. The general rule adopted (Fig. 3) was that each fixed field must begin at a multiple of its field length, and variable-length decimal and character fields are unconstrained and are processed serially in all models. All models must insure that programmers will adhere to these rules. This policing is essential to prevent the use of technically invalid programs that might work beautifully on small models but not on large ones. Such an outcome would undermine compatibility. The general rule, which has very few and very minor exceptions, is that invalidities defined in the manual are detected in the hardware and cause an interruption. This type of interruption is distinct from an interruption caused by machine malfunctions.

- *Instruction decisions*

Pushdown stack vs addressed registers. Serious consideration was given to a design based on a pushdown accumulator or stack.[2] This plan was abandoned in favor of several registers, each explicitly addressed. Since the advantages of the pushdown organization are discussed in the literature,[3] it suffices here to enumerate the disadvantages which prompted the decision to use an addressed-register organization:

1. The performance advantage of a pushdown stack organization is derived principally from the presence of several fast registers, not from the way they are used or specified.

2. The fraction of "surfacings" of data in the stack which are "profitable," i.e., what was needed next, is about one-half in general use, because of the occurrence of repeated operands (both constants and common factors). This suggests the use of operations such as TOP and SWAP, which respectively copy submerged data to the active positions and assist in clearing submerged data when the information is no longer needed.

3. With TOP's and SWAP's counted, the substantial instruction density gained by the widespread use of implicit addresses is about equalled by that of the same instructions with explicit, but truncated, addresses which specify only the fast registers.

4. In any practical implementation, the depth of the stack has a limit. The register housekeeping eliminated by the pushdown organization reappears as management of a finite-depth stack and as specification of locations of submerged data for TOP's and SWAP's. Further, when part of a full stack must be dumped to make room for new data, it is the *bottom* part, not the active part, which should be dumped.

5. Subroutine transparency, i.e., the ability to use a subroutine recursively, is one of the apparent advantages of the stack. However, the *disadvantage* is that the transparency does not materialize unless additional independent stacks are introduced for addressing purposes.

6. Fitting variable-length fields into a fixed-width stack is awkward.

In the final analysis, the stack organization would have been about break-even for a system intended principally for scientific computing. Here the general-purpose objective weighed heavily in favor of the more flexible addressed-register organization.

Full vs truncated addresses. From the beginning, the major challenge of compatibility lay in storage addressing. It was clear that large models would require storage capacities in the millions of characters. Small (serial) models would require short addresses to conserve precious core space and instruction fetch time. Some help was given by the decision to use register addressing, which reduces address appearances in the instruction stream by a factor approaching 2.

An early decision had dictated that all addresses had to be indexable, and that a mechanism had to be provided for making all programs easily relocatable. The indexing technique had fully proved its worth in current systems.[4] This technique suggested that abundant address size could be attained through a full-sized index register, used as a base. This approach, coupled with a truncated address in the instruction, gives consequent gains in instruction density. The *base-register* approach was adopted, and then augmented, for some instructions, with a second level of indexing.

Now the question was: How much capacity was to be made directly addressable, and how much addressable only via base registers? Some early uses of base register techniques had been fairly unsuccessful, principally because of awkward transitions between direct and base addressing. It was decided to commit the system completely to a base-register technique; the direct part of the address, the *displacement*, was made so small (12 bits, or 4096 characters) that direct addressing is a practical programming technique only on very small models. This

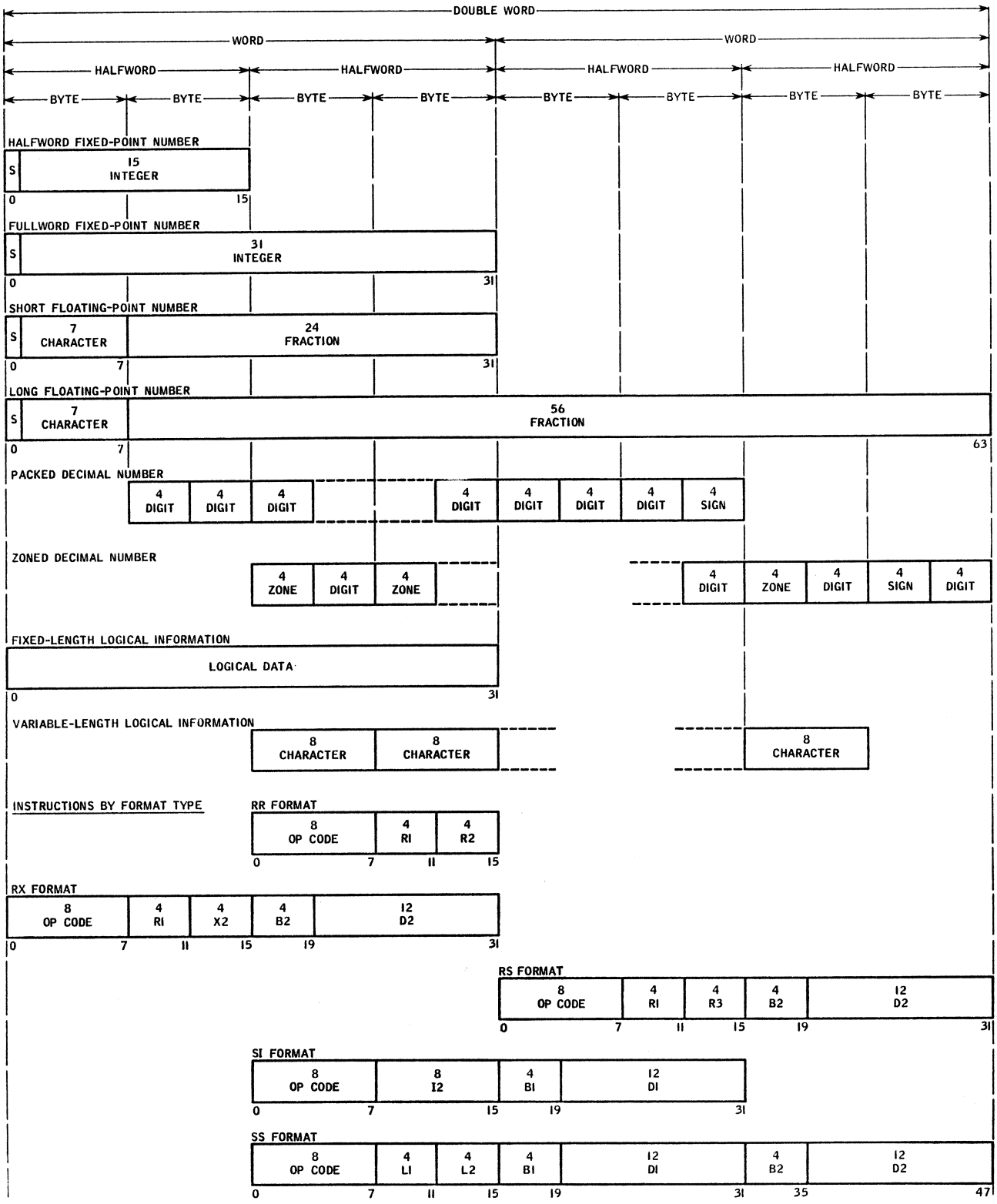


Figure 3 **Boundary alignment of formats.**

commitment implies that all programs are location-independent, except for constants used to load the base registers. Thus, all programs can easily be relocated. This commitment also implies that the programming support effectively and efficiently handles the mechanics of base-register use. The assembler automatically constructs and assigns base-plus-displacement addresses as it constructs the symbol table. The compilers not only do this, but also allocate base registers to give efficient programs.

Decimal vs binary addressing. It was decided to use binary rather than decimal addressing, because (a) assembly programs remove the user one level from the address, thus reducing the importance of familiar usage, (b) binary addressing is more efficient in the ratio 3.32/4.00, and (c) table exploitation is easier and more general because any datum can be made into or added to a binary address, yielding a valid address. This decision, however, represented some conflict with past approaches. Machines for purely business applications had often used decimal addressing (in the ancestral machine of the family). Most business computers now have binary addressing or have evolved to mixed-radix addressing.

Multiple accumulators. An extrapolation of technological trends indicated the probable availability of small, high-speed storage. Consequently, the design uses a substantial number of logically identifiable registers, which are physically realized in core storage, local high-speed storage, or transistors, according to the model. There are sixteen 32-bit general-purpose registers and four 64-bit floating-point registers in the logical design, with room for expansion to eight floating-point registers. Surprisingly enough, the multiple-register decision was not a large-small conflict. Each model has an appropriate (and different) mechanization of the same logical design.

Storage hierarchies. Technology promises to yield a continuing spectrum of storage systems whose speed varies inversely with capacity for equal cost-per-bit. Of equal significance, problem requirements naturally follow a matching pattern — small quantities of data are used with great frequency, medium quantities with medium frequency, and very large quantities with low frequency. These facts promise substantial performance/cost advantages if storage hierarchies can be effectively used.

It was decided to accept the engineering, architectural, and programming disciplines required for storage-hierarchy use. The engineer must accommodate in one system several storage technologies, with separate speeds, circuits, power requirements, busing needs, etc., all requiring asynchronous operation of all storage with respect to the CPU. The system programmer must contend with awkward boundaries within total storage capacity and must allocate usage. He must devise addressing for very large capacities, block transfers, and means of handling, indexing across and providing protection across gaps in the addressing sequence.

Separate vs universal accumulators. There are several advantages of having fixed- and floating-point arithmetic use the same logical (as opposed to physical) registers. There are some less obvious disadvantages which weighed in favor of separate accumulator sets. First, in a given register specification (4 bits, in our case) the use of separate sets permits more registers to be specified because of the information implications of the operation code. Second, in the large models instruction execution and the preparation of later instructions are done concurrently in separate units. To use a single register set would couple these closely, and reduce the asynchronous concurrency that can be attained. Historically, index registers have been separated from fixed-point registers, limiting analysis of register allocation to index quantities only. Integration of these facilities brings the full power of the fixed-point arithmetic operation set to bear upon indexing computations. The advantages of the integration appear throughout program execution (even compiler and assembly execution), whereas the register allocation burdens only compilation and assembly.

- *Input/output system*

The method of input/output control would have been a major compatibility problem were it not for the recognition of the distinction between logical and physical structures. Small machines use CPU hardware for I/O functions; large machines demand several independent channels, capable of operating concurrently with the CPU and with each other. Such large-machine channels often each contain more components than an entire small system.

Channel instructions. The logical design considers the channel as an independently operating entity. The CPU program starts the channel operation by specifying the beginning of a channel program and the unit to be used. The channel instructions, specialized for the I/O function, specify storage blocks to be read or written, unit operations, conditional and unconditional branches within the channel program, etc. When the channel program ends, the CPU program is interrupted, and complete channel and device status information are available.

An especially valuable feature is *command chaining*, the ability of successive channel instructions to give a sequence of different operations to the unit, such as SEARCH, READ, WRITE, READ FOR CHECK. This feature permits devices to be reinstructed in very short times, thus substantially enhancing effective speed.

Standard interface. The generalization of the communication between the central processing unit and an input/output device has yielded a channel which presents a standard interface to the device control unit. This interface was achieved by making the channel design transparent, passing not only data, but also control and status

information between storage and device. All functions peculiar to the device are placed in the control unit. The interface requires a total of 29 lines and is made independent of time through the use of interlocking signals.

Implementation. In small models, the flow of data and control information is time-shared between the CPU and the channel function. When a byte of data appears from an I/O device, the CPU is seized, dumped, used and restored. Although the maximum data rate handled is lower (and the interference with CPU computation higher) than with separate hardware, the function is identical.

Once the channel becomes a conceptual entity, using time-shared hardware, one may have a large number of channels at virtually no cost save the core storage space for the governing control words. This kind of *multiplex channel* embodies up to 256 conceptual channels, all of which may be concurrently operating, when the total data rate is within acceptable limits. The multiplexing constitutes a major advance for communications-based systems.

Conclusion

This paper has shown how the design features were chosen for the logical structure of the six models that comprise the IBM System/360. The rationale has been given for the adoption of the data formats, the instruction set, and the input/output controls. The main features of the new machine organization are its general-purpose utility for many types of data processing, the new approaches to large-capacity storage, and the machine-language compatibility among the six models.

The contributions discussed in this paper may be summarized as follows:

1. The relative independence of logical structure and physical realization permits efficient implementation at various levels of performance.

2. Tasks that are common to operating a system for most applications require a complement of instructions and system functions that may serve as a base for the addition of application-oriented functions.

3. The formats, instructions, register assignment, and over-all functions such as protection and interruption of a computer can be so defined that they apply to many levels of performance and that they permit diverse specialization for particular applications.

It is hoped that the discussions of these design features will shed some light on the present and future needs of data processing system organization.

Appendices

The design resulting from the decision process sketched above is tabulated in five appendices showing formats, data and instruction codes, storage assignments and interruption action. (*Appendices 1 through 5 appear on the following four pages.*)

Acknowledgments

The implementation of System/360 depends upon diverse developments by many colleagues. The most important of these developments were glass-encapsulated semi-integrated semiconductor components, printed circuit backpanels and interconnections, new memories, read-only storages and microprogram techniques, new I/O devices, and a new level and approach to software support.

The scope of the compatibility objective and of the whole System/360 undertaking was largely due to B. O. Evans, Data Systems Division Vice-President—Development.

References

1. D. W. Sweeney, "An Analysis of Floating-Point Addition and Shifting," to be published in the *IBM Systems Journal.*
2. See, for example, R. S. Barton, "A New Approach to the Functional Design of a Digital Computer," *Proc. WJCC* 19, 393–396 (1961).
3. F. P. Brooks, Jr., "Recent Developments in Computer Organization," *Advances in Electronics* 18, 45–64 (1963).
4. G. A. Blaauw, "Indexing," in *Planning a Computer System*, W. Buchholz, ed., McGraw-Hill Book Company Inc., 1962, pp. 150–178.

Received January 21, 1964

Appendix 1 **All operation codes are shown in the following table. The 8-bit codes are grouped by the main classes, such as fixed-point arithmetic, floating-point arithmetic and logical operations. The codes are furthermore grouped according to the five main instruction formats RR (register-register), RX (register-indexed storage location), RS (register-storage), SI (storage-immediate information) and SS (storage-storage).**

OPERATION CODES

FORMAT	RR	RR	RR	RR
CLASS	BRANCHING AND STATUS SWITCHING	FIXED-POINT FULLWORD, AND LOGICAL	FLOATING-POINT LONG	FLOATING-POINT SHORT
xxxx	0000xxxx	0001xxxx	0010xxxx	0011xxxx
0000		LOAD POSITIVE	LOAD POSITIVE	LOAD POSITIVE
0001		LOAD NEGATIVE	LOAD NEGATIVE	LOAD NEGATIVE
0010		LOAD AND TEST	LOAD AND TEST	LOAD AND TEST
0011		LOAD COMPLEMENT	LOAD COMPLEMENT	LOAD COMPLEMENT
0100	SET PROGRAM MASK	AND	HALVE	HALVE
0101	BRANCH AND LINK	COMPARE LOGICAL		
0110	BRANCH ON COUNT	OR		
0111	BRANCH/CONDITION	EXCLUSIVE OR		
1000	SET KEY	LOAD	LOAD	LOAD
1001	INSERT KEY	COMPARE	COMPARE	COMPARE
1010	SUPERVISOR CALL	ADD	ADD N	ADD N
1011		SUBTRACT	SUBTRACT N	SUBTRACT N
1100		MULTIPLY	MULTIPLY	MULTIPLY
1101		DIVIDE	DIVIDE	DIVIDE
1110		ADD LOGICAL	ADD U	ADD U
1111		SUBTRACT LOGICAL	SUBTRACT U	SUBTRACT U

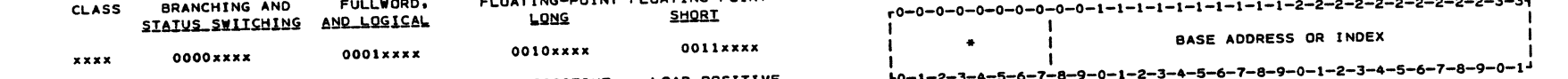
FORMAT	RX	RX	RX	RX
CLASS	FIXED-POINT HALFWORD AND BRANCHING	FIXED-POINT FULLWORD AND LOGICAL	FLOATING-POINT LONG	FLOATING-POINT SHORT
xxxx	0100xxxx	0101xxxx	0110xxxx	0111xxxx
0000	STORE	STORE	STORE	STORE
0001	LOAD ADDRESS			
0010	STORE CHARACTER			
0011	INSERT CHARACTER			
0100	EXECUTE	AND		
0101	BRANCH AND LINK	COMPARE LOGICAL		
0110	BRANCH ON COUNT	OR		
0111	BRANCH/CONDITION	EXCLUSIVE OR		
1000	LOAD	LOAD	LOAD	LOAD
1001	COMPARE	COMPARE	COMPARE	COMPARE
1010	ADD	ADD	ADD N	ADD N
1011	SUBTRACT	SUBTRACT	SUBTRACT N	SUBTRACT N
1100	MULTIPLY	MULTIPLY	MULTIPLY	MULTIPLY
1101		DIVIDE	DIVIDE	DIVIDE
1110	CONVERT-DECIMAL	ADD LOGICAL	ADD U	ADD U
1111	CONVERT-BINARY	SUBTRACT LOGICAL	SUBTRACT U	SUBTRACT U

Appendix 2 continued

from addressable registers and storage. The PSW is stored upon interruption. The Channel Command Word controls input/output operation and sequencing. The commands which may be given to the channel are listed as part of the table. The Channel Address Word is used to initiate input/output sequencing. The Channel Status Word indicates the channel status at the completion of an input/output operation or, when specified, during an I/O operation.

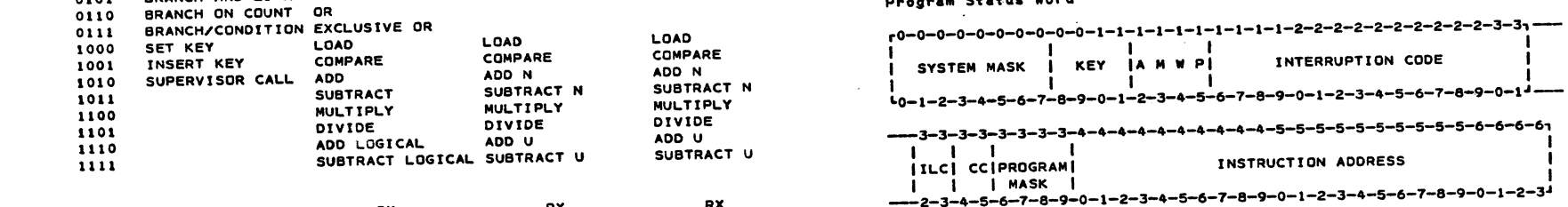
CONTROL WORD FORMATS

Base and Index Registers



0 - 7 Ignored
8 - 31 Base address or index

Program Status Word



0 - 7 System mask
- 0 Multiplexor channel mask
- 1 Selector channel 1 mask
- 2 Selector channel 2 mask
- 3 Selector channel 3 mask
- 4 Selector channel 4 mask
- 5 Selector channel 5 mask
- 6 Selector channel 6 mask
- 7 External mask

8 - 11 Protection key
12 ASCII mode (A)
13 Machine check mask (M)
14 Wait state (W)
15 Problem state (P)
16 - 31 Interruption code
32 - 33 Instruction length code (ILC)
34 - 35 Condition code (CC)
36 - 39 Program mask
- 36 Fixed-point overflow mask
- 37 Decimal overflow mask
- 38 Exponent underflow mask
- 39 Significance mask

40 - 63 Instruction address

Appendix 1 continued

FORMAT CLASS	RS,SI BRANCHING, STATUS SWITCHING AND SHIFTING	RS,SI FIXED-POINT, LOGICAL, AND INPUT/OUTPUT		
xxxx	1000xxxx	1001xxxx	1010xxxx	1011xxxx
0000	SET SYSTEM MASK	STORE MULTIPLE		
0001		TEST UNDER MASK		
0010	LOAD PSW	MOVE		
0011	DIAGNOSE			
0100	WRITE DIRECT	AND		
0101	READ DIRECT	COMPARE LOGICAL		
0110	BRANCH/HIGH	OR		
0111	BRANCH/LOW-EQUAL	EXCLUSIVE OR		
1000	SHIFT RIGHT SL	LOAD MULTIPLE		
1001	SHIFT LEFT SL			
1010	SHIFT RIGHT S			
1011	SHIFT LEFT S			
1100	SHIFT RIGHT DL	START I/O		
1101	SHIFT LEFT DL	TEST I/O		
1110	SHIFT RIGHT D	HALT I/O		
1111	SHIFT LEFT D	TEST CHANNEL		

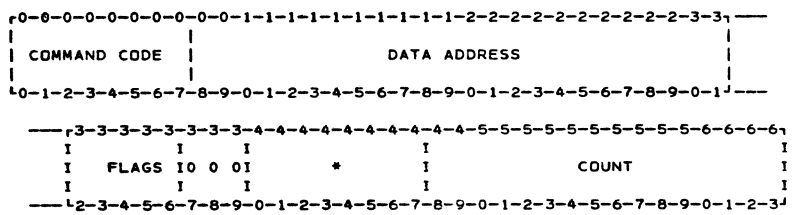
FORMAT CLASS		SS LOGICAL		SS DECIMAL
xxxx	1100xxxx	1101xxxx	1110xxxx	1111xxxx
0000				
0001		MOVE NUMERIC		MOVE W OFFSET
0010		MOVE		PACK
0011		MOVE ZONE		UNPACK
0100		AND		
0101		COMPARE LOGICAL		
0110		OR		
0111		EXCLUSIVE OR		
1000				ZERO AND ADD
1001				COMPARE
1010				ADD
1011				SUBTRACT
1100		TRANSLATE		MULTIPLY
1101		TRANSLATE AND TEST		DIVIDE
1110		EDIT		
1111		EDIT AND MARK		

Legend N = Normalized U = Unnormalized
SL = Single logical S = Single
DL = Double logical D = Double

Appendix 2 **The formats of all control words required for CPU and channel operation are shown in the following table. The base and index registers provide 24 bits of address and are specified by the B and X fields of instructions. The Program Status Word controls instruction sequencing and indicates the complete CPU status apart**

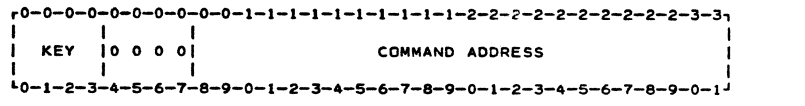
Appendix 2 continued

Channel Command Word



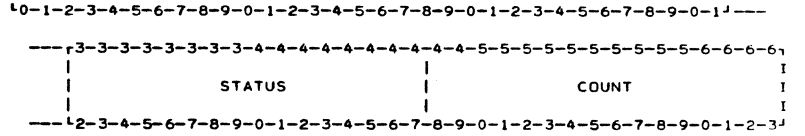
0 - 7 Command code
8 - 31 Data address
32 - 36 Command flags
32 Chain data flag
33 Chain command flag
34 Suppress length indication flag
35 Skip flag
36 Program-controlled interruption flag
37 - 39 Zero
40 - 47 Ignored
48 - 63 Count

Channel Address Word



0 - 3 Protection key
4 - 7 Zero
8 - 31 Command address

Channel Status Word



0 - 3 Protection key
4 - 7 Zero
8 - 31 Command address
32 - 47 Status
32 Attention
33 Status modifier
34 Control unit end
35 Busy
36 Channel end
37 Device end
38 Unit check
39 Unit exception
40 Program-controlled interruption

(*continued overleaf*)

Appendix 2 continued

```
   41    Incorrect length
   42    Program check
   43    Protection check
   44    Channel data check
   45    Channel control check
   46    Interface control check
   47    Chaining check
48 - 63 Count
```

Appendix 3 **All permanently assigned storage locations are shown in this table. These locations are addressed by the CPU and I/O channels during initial program loading, during interruptions and in order to update the timer. During initial program loading 24 bytes are read from a specified input device into locations 0 to 23. This information is subsequently used as CCW's to specify the locations of further input information and as a PSW to control CPU operation after the loading operation is completed. During an interruption the current PSW is stored in the "old" location and the PSW from the "new" location is obtained as the next PSW. The timer is counted down and provides an interrupt when zero is passed. All permanently assigned locations may also be addressed by the program.**

PERMANENT STORAGE ASSIGNMENT

	ADDRESS	LENGTH	PURPOSE
0	0000 0000	double word	Initial program loading PSW
8	0000 1000	double word	Initial program loading CCW1
16	0001 0000	double word	Initial program loading CCW2
24	0001 1000	double word	External old PSW
32	0010 0000	double word	Supervisor call old PSW
40	0010 1000	double word	Program old PSW
48	0011 0000	double word	Machine old PSW
56	0011 1000	double word	Input/output old PSW
64	0100 0000	double word	Channel status word
72	0100 1000	word	Channel address word
76	0100 1100	word	Unused
80	0101 0000	word	Timer
84	0101 0100	word	Unused
88	0101 1000	double word	External new PSW
96	0110 0000	double word	Supervisor call new PSW
104	0110 1000	double word	Program new PSW
112	0111 0000	double word	Machine new PSW
120	0111 1000	double word	Input/output new PSW
128	1000 0000		Diagnostic scan-out area*

---

* The size of the diagnostic scan-out area depends upon the particular model and I/O channels.

Appendix 4 continued

Legend

available	Unit and channel available
busy	Unit or channel busy
carry	A carry out of the sign position occurs
complete	Last result byte nonzero
CSW ready	Channel status word ready for test or interruption
CSW stored	Channel status word stored
equal	Operands compare equal
F	Fullword
g zero	Result is greater than zero
H	Halfword
halted	Data transmission stopped. Unit in halt-reset mode
high	First operand compares high
incomplete	Nonzero result byte; not last
L	Long precision
l zero	Result is less than zero
low	First operand compares low
mixed	Selected bits are both zero and one
not oper	Unit or channel not operational
not working	Unit or channel not working
not zero	Result is not all zero
one	Selected bits are one
overflow	Result overflows
S	Short precision
stopped	Data transmission stopped
working	Unit or channel working
zero	Result or selected bits are zero

Note

The condition code also may be changed by LOAD PSW, SET SYSTEM MASK, DIAGNOSE, and by an interruption.

Appendix 5 **All interruptions which may occur are shown in the following table. Indicated here are the code in the old PSW which identifies the source of the interruption, the mask bits which may be used to prevent an interruption, and the manner in which instruction execution is affected. The instruction to be performed next if the interruption had not occurred is indicated in the instruction address field of the old PSW. The length of the preceding instructions, if available, is shown in the instruction length code, ILC, as is further detailed in the table.**

INTERRUPTION ACTION

INTERRUPTION SOURCE IDENTIFICATION	INTERRUPTION CODE PSW BITS 16-31	MASK BITS SET	ILC	INSTRUCTION EXECUTION
Input/Output (old PSW 56, new PSW 120, priority 4)				
Multiplexor channel	00000000 aaaaaaaa	0	x	complete
Selector channel 1	00000001 aaaaaaaa	1	x	complete
Selector channel 2	00000010 aaaaaaaa	2	x	complete

Appendix 4 **All instructions which set the condition code (bits 32 and 33 of the PSW) are listed in the following table. All other instructions leave the condition code unchanged. The condition code determines the outcome of a** BRANCH ON CONDITION **instruction. The four-bit mask contained in this instruction specifies which code settings will cause the branch to be taken.**

CONDITION CODE SETTING

	0	1	2	3
Fixed-Point Arithmetic				
ADD H/F	zero	l zero	g zero	overflow
ADD LOGICAL	zero	not zero	zero,carry	carry
COMPARE H/F	equal	low	high	--
LOAD AND TEST	zero	l zero	g zero	--
LOAD COMPLEMENT	zero	l zero	g zero	overflow
LOAD NEGATIVE	zero	l zero	--	--
LOAD POSITIVE	zero	--	g zero	overflow
SHIFT LEFT DOUBLE	zero	l zero	g zero	overflow
SHIFT LEFT SINGLE	zero	l zero	g zero	overflow
SHIFT RIGHT DOUBLE	zero	l zero	g zero	--
SHIFT RIGHT SINGLE	zero	l zero	g zero	--
SUBTRACT H/F	zero	l zero	g zero	overflow
SUBTRACT LOGICAL	--	not zero	zero,carry	carry
Decimal Arithmetic				
ADD DECIMAL	zero	l zero	g zero	overflow
COMPARE DECIMAL	equal	low	high	--
SUBTRACT DECIMAL	zero	l zero	g zero	overflow
ZERO AND ADD	zero	l zero	g zero	overflow
Floating-Point Arithmetic				
ADD NORMALIZED S/L	zero	l zero	g zero	overflow
ADD UNNORMALIZED S/L	zero	l zero	g zero	overflow
COMPARE S/L	equal	low	high	--
LOAD AND TEST S/L	zero	l zero	g zero	--
LOAD COMPLEMENT S/L	zero	l zero	g zero	--
LOAD NEGATIVE S/L	zero	l zero	--	--
LOAD POSITIVE S/L	zero	--	g zero	--
SUBTRACT NORMALIZED S/L	zero	l zero	g zero	overflow
SUBTRACT UNNORMALIZED S/L	zero	l zero	g zero	overflow
Logical Operations				
AND	zero	not zero	--	--
COMPARE LOGICAL	equal	low	high	--
EDIT	zero	l zero	g zero	--
EDIT AND MARK	zero	l zero	g zero	--
EXCLUSIVE OR	zero	not zero	--	--
OR	zero	not zero	--	--
TEST UNDER MASK	zero	mixed	--	one
TRANSLATE AND TEST	zero	incomplete	complete	--
Input-Output Operations				
HALT I/O	not working	halted	stopped	not oper
START I/O	available	CSW stored	busy	not oper
TEST CHANNEL	not working	CSW ready	working	not oper
TEST I/O	available	CSW stored	working	not oper

Appendix 5 continued

Selector channel 3	00000011 aaaaaaaa	3	x		complete
Selector channel 4	00000100 aaaaaaaa	4	x		complete
Selector channel 5	00000101 aaaaaaaa	5	x		complete
Selector channel 6	00000110 aaaaaaaa	6	x		complete
Program (old PSW 40, new PSW 104, priority 2)					
Operation	00000000 00000001		1,2,3		suppress
Privileged operation	00000000 00000010		1,2		suppress
Execute	00000000 00000011		2		suppress
Protection	00000000 00000100		0,2,3		suppress/terminate
Addressing	00000000 00000101		0,1,2,3		suppress/terminate
Specification	00000000 00000110		1,2,3		suppress
Data	00000000 00000111		2,3		terminate
Fixed-point overflow	00000000 00001000	36	1,2		complete
Fixed-point divide	00000000 00001001		1,2		suppress/complete
Decimal overflow	00000000 00001010	37	3		complete
Decimal divide	00000000 00001011		3		suppress
Exponent overflow	00000000 00001100		1,2		terminate
Exponent underflow	00000000 00001101	38	1,2		complete
Significance	00000000 00001110	39	1,2		complete
Floating-point divide	00000000 00001111		1,2		suppress
Supervisor Call (old PSW 32, new PSW 96, priority 2)					
Instruction bits	00000000 rrrrrrrr		1		complete
External (old PSW 24, new PSW 88, priority 3)					
External signal 1	00000000 xxxxxxx1	7	x		complete
External signal 2	00000000 xxxxxx1x	7	x		complete
External signal 3	00000000 xxxxx1xx	7	x		complete
External signal 4	00000000 xxxx1xxx	7	x		complete
External signal 5	00000000 xxx1xxxx	7	x		complete
External signal 6	00000000 xx1xxxxx	7	x		complete
Interrupt key	00000000 x1xxxxxx	7	x		complete
Timer	00000000 1xxxxxxx	7	x		complete
Machine Check (old PSW 48, new PSW 112, priority 1)					
Machine malfunction	00000000 00000000	13	x		terminate

Legend

a Device address bits
r Bits of R1 and R2 field of SUPERVISOR CALL
x Unpredictable

INSTRUCTION LENGTH RECORDING

INSTRUCTION LENGTH CODE	PSW BITS 32-33	INSTRUCTION BITS 0-1	INSTRUCTION LENGTH	INSTRUCTION FORMAT
0	00		Not available	
1	01	00	One halfword	RR
2	10	01	Two halfwords	RX
2	10	10	Two halfwords	RS or SI
3	11	11	Three halfwords	SS

M. J. Flynn*
P. R. Low

The IBM System/360 Model 91: Some Remarks on System Development

Introduction

• *System objectives*

The primary goal of the System/360 Model 91 development program was to produce the highest performance capability that advanced design philosophy and extensions of System/360 circuit technology could achieve, within a balanced development schedule. "Performance," as used here, means general computer availability and high-speed execution of general problem programs.

A system, of course, is composed of many parts other than the processor, and the design was implemented to incorporate a number of existing models and modules of peripheral equipment. For example, the system was to be optimized for a 3/4-μsec memory, the same basic unit used in System/360 Models 65 through 75.

Another consideration in system development was manufacturability. The system was designed so that timing considerations, "de-skewing" and tuning of the clock did not become inordinately complex. Similarly, component design for any part which is used extensively was carefully matched to manufacturing requirements.

A major aspect of the Model 91 development effort was the checking and serviceability philosophy. It was recognized that all arithmetic operations and all data transfers should be fully checked. Some of this checking, such as parity checking to the byte level on data transfers, was made necessary by the compatibility requirement. Experience has shown, however, that the size and complexity of a large system demand an active concern for checking features on the execution of as many operations as possible. Similarly, with respect to serviceability, it was felt at the outset that display provisions for every storage element in the system should be provided. Such provisions, together with the System/360 "log out" feature which provides for a complete readout of the machine state when an error is incurred, served as the initial basis for serviceability design.

• *Technology objectives*

The obvious objective of the technology† group was to develop a high-speed, very efficient technology and place it in production within the time provided by the development schedule. It was recognized at the outset that if this goal were to be attained, tooling compatibility would have to be maintained between the Model 91 technology—or ASLT (Advanced Solid Logic Technology), as it has been named —and SLT (Solid Logic Technology).

One could find references in the literature to kilomegacycle transistors as early as 1962 and references reporting work on 2-nsec circuits in 1963. Ultra-fast circuits, however, do not necessarily make an ultra-fast computer. For example, 2 nsec is approximately the time required for the electromagnetic wave representing the state of a circuit to travel along 12 in. of printed circuit conduction path; thus, if the particular implementation of the ultra-fast circuits requires that, on the average, the circuits be spaced 12 in. apart, one will have a delay of not 2 nsec, but 4 nsec per circuit. Furthermore, circuit delay quoted in the literature sometimes fails to consider the effect of interactions among circuits connected to the input and output of the device under test. These so-called "loading effects" typically add from 2 to 4 nsec to the basic circuit delay. Thus, when all factors have been considered, one finds that the circuit delay in the systems environment may not be 2 nsec, but may in fact be much closer to 6 nsec. As a result of this confusion, the performance objectives for the ASLT technology were always stated in terms of delay "in the environment." It was estimated that a 5-nsec circuit measured as described above would be required if the

* Present address: Northwestern University, Evanston, Illinois.

† "Technology" as used here will be taken to mean those aspects of components, circuitry, physical structure, etc. which are considered separately from system architecture, programming, organization and gross-function algorithms. The latter have been identified as "systems."

machine were to meet its performance objectives. To achieve such an objective would require a balanced effort that would simultaneously develop circuits, devices, modules, and second-level packaging.

The basic circuit, of course, had to be as fast as possible. Equally important, however, were the rules by which the circuits could be interconnected, so that the system's designer was not unduly constrained during the logic design and card layout phase of the program. In high-speed computer design, predictable circuit delays can be as important as short delays. Thus, a rather extensive effort was undertaken to be certain that the various combinations of circuits and printed conduction paths would behave in a predictable manner. In certain sections of the machine, pulse-width preservation (and thus the ability to operate at a high repetition rate) was extremely important. Hence it was necessary to guarantee that the circuit rise and fall times would be as short as possible. It would have been unacceptable, for example, to have a circuit that had a 1 nsec delay but a 3–5 nsec rise or fall time.

The basic packaging density had to be significantly increased to assure that the performance of the circuit would not be wasted by excessively long printed circuit paths. At the module level, a packaging density increase of at least a factor of three over SLT was required. An increase in the circuit packaging density, however, would put additional loads on the cooling and power signal distribution systems; significant modifications both to those areas and to the printed circuit card and back panel supporting structure were therefore required.

Results

- *System*

After consideration of all requirements, the prime specification was established as the execution of an average of one instruction per machine cycle. The basic machine cycle was defined as that time required to decode a System/360 instruction, which for Model 91 is 60 nsec.

The concept of processing one instruction per cycle has now become the basis for the entire system and forms the interface between major areas of the system. Thus, the storage unit and its communication system must be able to provide the processor with data and instructions at a rate sufficient to permit execution of one instruction per cycle. Similarly, the instruction unit must be able to distribute, queue, and organize its resources at a rate consistent with this same (one instruction per cycle) target. Finally, the execution unit itself must be designed to provide an average execution rate of one instruction per cycle. Clearly, all the processing required for any one instruction cannot be completed in one machine cycle. Instead, as many as nine or ten instructions are in various stages of processing at the same time. The instruction unit coordinates the acquisition, distribution, and execution of the instructions by buffering both instructions and data. The design of this unit is described in the paper by D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, entitled "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling."

The interface from the instruction unit to the storage communications unit and thence to the storage unit itself, and the subsequent achievement of a regular supply of instructions and data from the storage unit are discussed in the paper, "The IBM System/360 Model 91: Storage System," by L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina and J. W. Smith. An interface also exists between the main storage unit and the hierarchy of storage and input/output equipment. The important point here is that, despite the delay that accrues when the desired words must be accessed, the flow of information is maintained at a rate of one instruction per cycle, even across this hierarchy of boundaries.

Even when the information is available however, its execution, especially in important floating-point programs, is quite difficult at the target rate of one instruction per cycle. Two major developments represented in the Model 91 accomplish this rate. The first development is an algorithm that permits the decomposition of apparently dependent instruction sequences into independent elements; execution may proceed regardless of the order of these elements. The algorithm is described by R. M. Tomasulo in the paper entitled "An Efficient Algorithm for Exploiting Multiple Arithmetic Units."

In addition to the Tomasulo algorithm, the actual execution of floating-point instructions must be considerably enhanced over prior art in order to meet the goal. The second development, described in the paper by S. F. Anderson, J. Earle, R. E. Goldschmidt and D. M. Powers, achieves this with a three-cycle multiplication using a highly efficient iterative design approach. Sharing the same iterative hardware as the multiply function is an iterative divide scheme based on the Newton-Raphson algorithm. This is the first known hardware implementation of such a divide scheme.

The Model 91 system has a substantial performance improvement compared with previous IBM systems. (Table 1 makes this comparison.) In particular, note that floating-point programs can be executed more than a hundred times faster than on a machine such as the IBM 7090. Programs with many branch instructions, however, suffer somewhat by comparison because of the delay that results from the access to storage.

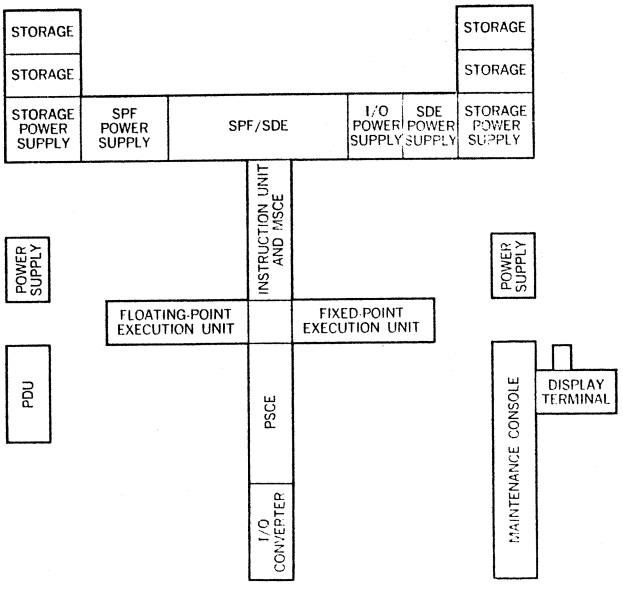
The system is also compatible with other models of System/360. This compatibility, however, is qualified by three specific exceptions:

1. Decimal option instructions are not provided in the hardware because the system was optimized for the scientific user, and thus it was felt that there would not be substantial interest in these instructions. However, programming simulations can be provided to effect the execution of these instructions.
2. Data- and address-dependent interrupts are not discretely noted, since instruction processing is concurrent; instead, these cause an "imprecise interrupt."
3. The quotient of a floating-point divide instruction may differ somewhat from that obtained from other models of the System/360 family. Although integrity of integers is preserved and the resulting precision is at least as good as that in other machines, the nature of the algorithm is such that somewhat different fractions may result.

Table 1 Comparison of machine performance on various problem kernels.

	Relative Internal Machine Speed		
Kernel Type	*7090*	*System/360 Model 75*	*System/360 Model 91*
FORTRAN 1F (Branch instructions predominate)	1	6.6	14.5
Matrix Multiply (Floating-point instructions predominate)	1	6.85	102
Polynomial Evaluation (Floating-point instructions predominate)	1	7.6	95.7

Figure 1 Schematic of system configuration (I/O and peripheral storage not shown).



The Model 91 can assume a number of configurations, but these vary only in the size and location of storage and peripheral equipment. The processor itself is organized to accommodate the instruction handling, storage control and arithmetic functions that are common to all variations, and its appearance in the same form is a characteristic of every Model 91 system.

One possible configuration is shown in Fig. 1. The processor, or CPU, is assembled in the form of a cross. One arm of the cross contains the floating-point execution unit, and another contains the fixed-point execution unit. A third arm contains the instruction unit and the main storage control element (MSCE), and the fourth arm contains an I/O converter and the peripheral storage control element (PSCE). Assembled around the CPU are free-standing frames such as the power distribution unit (PDU) and two power frames which feed regulated dc power directly to the CPU itself. Also adjacent to the CPU are the maintenance console and its associated display unit. The main storage element is shown adjacent to the main storage control element and contains the power supplies used to operate storage. In addition, the storage frame contains a storage distribution element (SDE) and the storage protect feature (SPF).

The Model 91 processor can communicate with a great variety of I/O equipment and peripheral storage, via the PSCE and the I/O converter. Two types of channels may be used. A possible channel layout is shown in Fig. 2, in which two Selector channels are connected directly to the processor and the processor storage. Each of these Selector channels controls high-speed I/O equipment as indicated. In addition, a Multiplexor channel may also be provided to control communication with slow-speed I/O equipment and to provide additional Selector sub-channels for such devices as magnetic tape units.

As indicated in the initial remarks, much attention has been devoted to checking and serviceability. The resulting system has all its arithmetic operations and data transfers checked. Also, the contents of all register positions (more than 6000 in all) are displayed either on the maintenance CRT console or with indicator lights, and features such as log-out are implemented via the maintenance console.

Careful attention has been paid to the detection of failures; indeed, for some portions of the machine, failures can be detected, the error pattern stored, and the failure later diagnosed to the level of the failing card, all from the maintenance console. It is possible to enter information to any buffer register or any level of a queue or to inject an error (to check the parity circuitry) from the console.

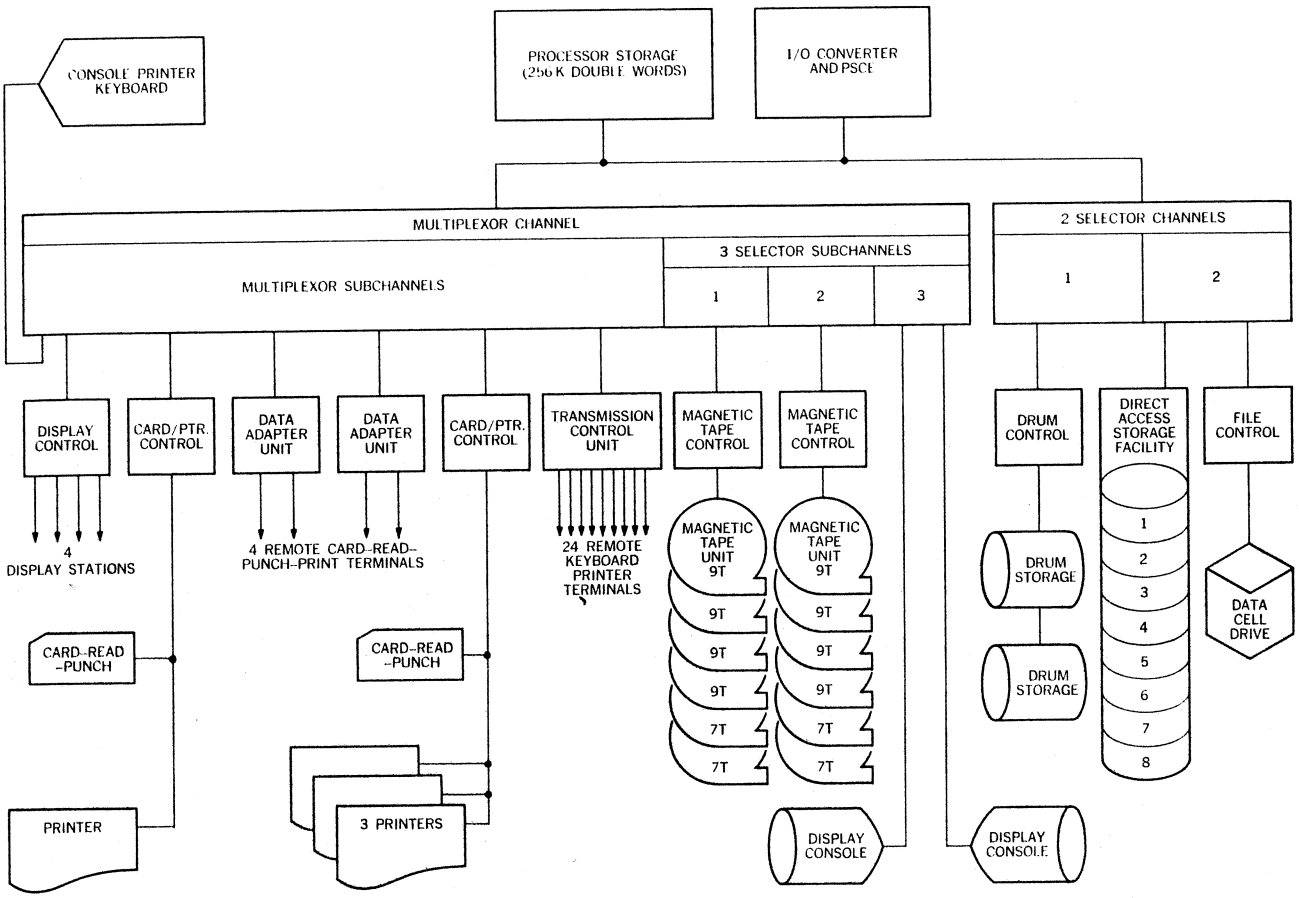


Figure 2 Example of I/O configuration.

Also, the timing of the machine may be varied so that only one instruction is being executed at a time, and the rate of processing may be slowed from a multiple-cycle rate to single-cycle (manual or stepped). Complete memory testing may be performed by searching or comparing all of storage with respect to a reference pattern. Memory contents may be displayed sixteen 72-bit words at a time. Re-try is also possible on a machine check interrupt; i.e., the program will be re-run from the last checkpoint.

Attention was given to timing and the attendant logical design to minimize the potential manufacturing problems. All the latches in the machine are of the fail-safe variety; that is, they are not sensitive to skewing between data and the control lines. (This particular method of timing is described in the paper by S. F. Anderson et al.)

- *Technology*

The results of the technology development program were quite successful in meeting the design objectives. Circuit delay (without wire) ranged from 1.6 to 2.5 nsec, depending upon the environment in which the circuit was placed. The output rise time varied from 1 to 1.5 nsec. The circuit fan power (that is, the number of inputs that a given circuit may have and the number of similar circuits that a given circuit may drive) was significantly higher than had been attained in any previous high-speed circuit family. The effective number of inputs for a given circuit was 16, and each of the standard logic blocks could drive 10 others. A technique was developed that allowed the systems designer to predict the circuit delay within 10-12%, and when prediction errors in that range did occur, the measured delay was in general smaller than predicted. A general comparison of the technology with high-speed SLT is shown in Table 2.

An important checkpoint was established early in the development phase of the engineering program. A standard, or test model, was jointly developed by the technology and systems development groups and was implemented using the ASLT technology. The logic implemented represented

Table 2 General comparison of high-speed SLT and ASLT.

	High-speed SLT	*ASLT*
Circuit speed	5.5 nsec	1.8 nsec
Circuits/back panel board	800 (max.)	5000 (max.)
Transmission line impedance	100 Ω	50 Ω and 90 Ω

one version of a floating-point adder that was under consideration by the systems group during development. The logic design and card layout were done by the systems group; the construction and evaluation were done by the technology group.

The logic path consisted of 23 levels of logic, packaged on ASLT cards. One hundred and forty-five circuits, representing the 23 logic stages, plus the additional fan-out and fan-in loading dictated by the systems design, were used. A total of 247.6 in. of printed circuit, wire, and cable interconnected the 23 levels of logic. For the systems group to meet their machine performance objectives, it was essential that the logic delay through this path, including the delay introduced by the printed circuit wire, be less than 115 nsec (or 5 nsec per logic level). The actual delay measured after the model was constructed was 103 nsec, representing 4.48 nsec per logic decision, measured in the system environment. It was also found that a 6 nsec pulse width could be used.

As a check, the delay was calculated using delay prediction techniques. The prediction program indicated that the delay through this network would be approximately 108 nsec and that the minimum pulse width that could be propagated through this network would be approximately 5.5 nsec.

Once the basic performance objectives of the development program had been achieved, the technology was released to the manufacturing plants concerned. At this writing, the modules, cards and boards are all in production. The technology has been improved somewhat during its manufacturing phase. This improvement derives primarily from refinements in transistor fabrication and from the more positive control of process variables provided by automatic manufacturing tools. The result is that the delay through the test model, using current production modules, has been reduced from 103 to just under 100 nsec.

A more detailed discussion of the various aspects of the technology development programs is contained in three papers in this issue. "ASLT Circuit Design and Engineering," by A. R. Strube, R. F. Sechler and J. R. Turnbull, describes the basic circuit design philosophy, the performance objectives, and the delay prediction techniques. Another paper, by J. Langdon and E. J. Van Derveer, describes the design and development of the high-speed switching transistor used in the circuit described by Strube et al. A final paper, by R. H. F. Lloyd, outlines the techniques which have extended SLT to the packaging of high-speed circuits. Particular attention is paid to the increase in density achieved at the module level.

Some considerations for the future

It has become evident that the target design goal of the execution of one instruction per cycle was a prudent one; targets much in excess of this figure would require exponentially more hardware. We infer from this that, in the future, order-of-magnitude improvements in computer performance due to systems organization alone will be unlikely, at least where emphasis is placed on simplex instruction processing (single-line instruction processing as against multi-processing). The converse of the last statement is, of course, that, if substantially greater performance improvements must be achieved by means of systems organization, probably the most fruitful areas would be based on the intimate connection of many independent processors.

Acknowledgments

Adequate acknowledgment for a development effort of this size is impossible. We make a sincere attempt below, knowing beforehand that it will be inadequate.

Primary direction was given by Mr. B. O. Evans.

Mr. L. E. Kanter provided the overall engineering management, and Mr. D. J. Galage was manager of the development group.

Drs. G. Amdahl and T. C. Chen worked tirelessly to insure both market and management acceptance for the system.

Although not discussed in detail in the papers mentioned above, an important feature of the Model 91 is its means for accommodating variable field-length instructions. Messrs. R. J. Litwiller and J. G. Adler were responsible for its development.

Mr. R. M. Meade provided early management guidance. Mr. C. J. Conti was an indispensable contributor to both the over-all system and the processor organization.

Messrs. W. Baskin, R. W. Emery and E. Morris provided the direction of the system technology, while Messrs. G. Halgas, R. Lehtonen and J. Laschenski developed the maintenance features.

Dr. E. M. Davis was responsible for the over-all direction of the technology program.

Mr. R. Rinne and Mr. R. Zurowski were responsible for and made important contributions to the printed circuit packaging phase of the effort.

Mr. D. D. Metzger had the responsibility for the module development effort and Mr. D. DeWitt was responsible for the semiconductor device development program.

Important contributions to the circuit designs and specifications were made by Mr. E. J. Rymaszewski.

Mr. R. A. Henle and Mr. J. L. Walsh provided early technical direction and established some of the basic principles on which the circuit design was based.

Received January 11, 1966.

S. F. Anderson
J. G. Earle
R. E. Goldschmidt
D. M. Powers

The IBM System/360 Model 91: Floating-Point Execution Unit

Abstract: The principal requirement for the Model 91 floating-point execution unit was that it be designed to support the instruction-issuing rate of the processor. The chosen solution was to develop separate, instruction-oriented algorithms for the add, multiply, and divide functions. Linked together by the floating-point instruction unit, the multiple execution units provide concurrent instruction execution at the burst rate of one instruction per cycle.

Introduction

The instruction unit of the IBM System/360 Model 91 is designed to issue instructions at a burst rate of one instruction per cycle, and the performance of floating-point execution must support this rate. However, conventional execution unit designs cannot support this level of performance. The Model 91 Floating-Point Execution Unit departs from convention and is instruction-oriented to provide fast, concurrent instruction execution.

The objectives of this paper are to describe the floating-point execution unit. Particular attention is given to the design of the instruction-oriented units to reveal the techniques which were employed to match the burst instruction rate of one instruction per cycle. These objectives can best be accomplished by dividing the paper into four sections—*General design considerations*, *Floating-point terminology*, *Floating-point add unit*, and *Floating-point multiply/divide unit*.

The first section explains how the desire for concurrent execution of instructions has led to the design of multiple execution units linked together by the floating-point instruction unit. Then the concept of instruction-oriented units is discussed, and its impact on the multiplicity of units is pointed out. It is shown that, with the instruction-oriented units as building blocks and the floating-point instruction unit as the "cement," an execution unit evolves which rises to the desired performance level.

The section on floating-point terminology briefly reviews the System/360 data formats and floating-point definitions.

The next two sections describe the design of the instruction-oriented units. The first of these is the floating-point add unit description which is divided into two sub-sections, *Algorithm* and *Implementation*. In the algorithm sub-section, the complete algorithm for execution of a floating add/subtract is considered with emphasis on the difficulties inherent in the implementation. Since the add unit is instruction-oriented, (i.e., only add-type instructions must be considered), it is possible to overcome the inherent difficulties by merging the several steps of the algorithm into three hardware areas. The implementation section describes these three areas, namely, characteristic comparison and pre-shifting, fraction adder, and post-normalization.

The last section describes the floating-point multiply/divide unit. This section describes the multiply algorithm and its implementation first, and then the divide algorithm and its implementation. The emphasis of the multiply algorithm sub-section is on recoding the multiplier and the usefulness of carry-save adders. In the implementation sub-section the emphasis is on the iterative hardware which is the heart of the multiply operation. An arrangement of carry-save adders is shown which, when pipelined by adding temporary storage platforms, has an iteration repetition rate of fifty Mc/sec. The divide algorithm is described next with emphasis on using multiplication, instead of subtraction, as the iterative operator. The discussion of divide implementation shows how the existing multiply hardware, plus a small amount of additional circuitry, is used to perform the divide operation.

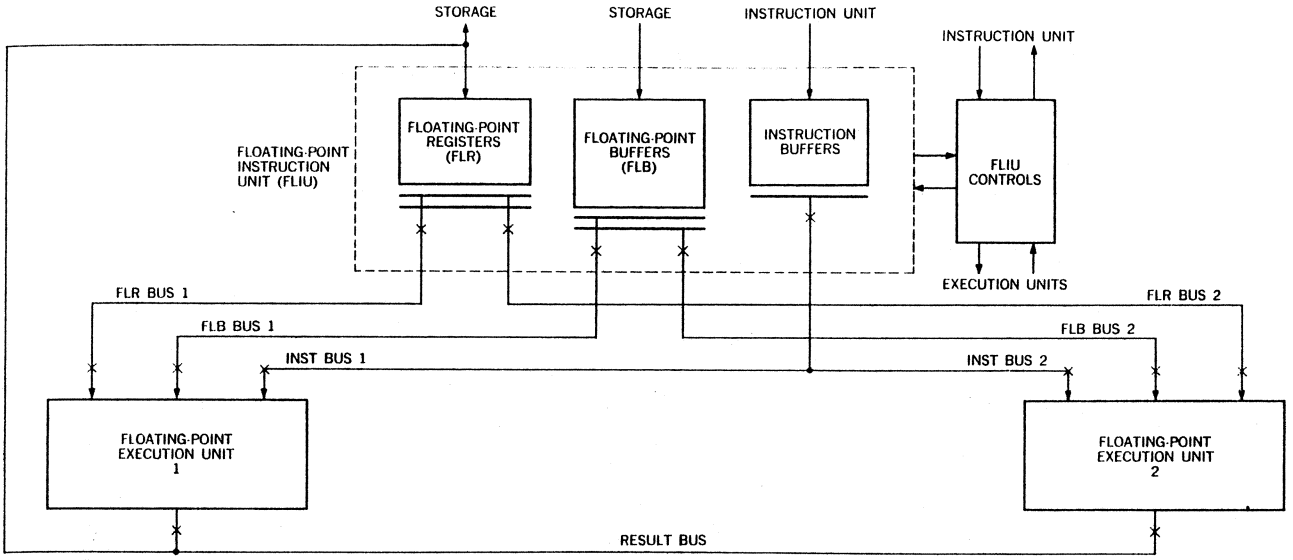


Figure 1 Floating-point execution unit capable of concurrent execution.

General design considerations

The programs considered "typical" by the user of high-performance computers are floating-point oriented. Therefore, the prime concern in designing the floating-point execution unit is to develop an overall organization which will match the performance of the instruction unit. However, the execution time of floating-point instructions is long compared with the issuing rate of these instructions by the instruction unit. The most obvious approach is to apply a faster technology and with special design techniques reduce the execution time for floating-point. But a study of many "typical" floating point programs revealed that the execution time per instruction would have to be 1 to 2 cycles in order to match the performance[3] of the instruction unit.* Conventional execution unit design, even with state-of-the-art algorithms, will not provide these execution times.

Another approach considered was to provide execution concurrency among instructions; this obviously would require two complete floating-point execution units.† An attendant requirement would be a floating-point instruction unit. This unit is necessary to sequence the operands from storage to the proper execution unit; it must buffer the instructions and assign each instruction to a non-busy execution unit. Also, since the execution time is not the same for all instructions the possibility now exists for out-of-sequence execution, and the floating-point instruction must insure that executing out of sequence does not produce incorrect results.* The organization for an execution unit capable of concurrent execution is shown in Fig. 1. Buffering and sequence control of all instructions, storage operands, and floating-point accumulators are the responsibility of the floating-point execution unit. Each of the execution units is capable of executing all floating-point instructions.

One might be led to believe that this organization is a suitable solution in itself. If multiply can be executed in seven cycles and two multiplies are executed simultaneously, then the effective execution time is 3.5 cycles. Similarly, for add the execution time would go from three cycles to 1.5 cycles. However, the operating delay of the floating-point instruction unit must be considered, and it is not always possible to execute concurrently because of the dependence among instructions. When these problems are considered the effective execution time is close to three cycles per instruction, which is not sufficient. A third execution unit would not help because the complexity of the floating-point instruction unit increases, and the amount of hardware becomes prohibitive.

The next solution to be considered was to improve the execution time of each instruction by employing faster algorithms in the design of each execution unit. Obviously this would increase the hardware, but since the circuit

* Even though the burst rate of the instruction unit is one instruction per cycle, it is not necessary to execute at the same rate.

† Since two complete execution units are necessary for concurrent execution, the cost-performance factor is important. Analysis showed that execution times of three cycles for add and seven cycles for multiply were reasonable expectations.

* Dependence among instructions must be controlled. If instruction $n + 1$ is dependent on the result of instruction n, instruction $n + 1$ must not be allowed to start until instruction n is completed.

Table 1 Floating-point instructions executed by floating-point execution unit.

Type	*Instruction*	*Condition code*	*Floating-point exceptions**	*Arithmetic unit*
RR-RX	Load (S/L)	NO		FLIU
RR	Load and Test (S/L)	YES		FLIU
RX	Store (S/L)	NO		FLIU
RR	Load Complement (S/L)	YES		ADD
RR	Load Positive (S/L)	YES		ADD
RR	Load Negative (S/L)	YES		ADD
RR-RX	Add Normalized (S/L)	YES	U, E, LS	ADD
RR-RX	Add Unnormalized (S/L)	YES	E, LS	ADD
RR-RX	Subtract Normalized (S/L)	YES	U, E, LS	ADD
RR-RX	Subtract Unnormalized (S/L)	YES	E, LS	ADD
RR-RX	Compare (S/L)	YES		ADD
RR	Halve (S/L)	NO		ADD
RR-RX	Multiply	NO	U, E	M/D
RR-RX	Divide	NO	U, E, FK	M/D

* *Exceptions:*
U—Exponent-underflow exception
E—Exponent-overflow exception
LS—Significance exception
FK—Floating Point Divide Exception

delay is a function not only of the circuit speed but also of the number of loads on the input net and the length of the interconnection wiring, more hardware may not make the unit faster.[5] These two factors—the desire for faster execution of each instruction and the size sensitivity of the circuit delay, have produced a concept which is unique to the organization of floating-point execution units, and which was adopted for the Model 91: the concept of using separate execution units for different instruction types. Faster execution of each instruction can be achieved if the conventional execution unit is separated into arithmetic units designed to execute a subset of the floating-point instructions instead of the entire set. This conclusion may not be obvious, but a unit designed exclusively for a class of similar instructions can execute those instructions faster than a unit designed to accommodate all floating-point instructions. The control sequences are shorter and less complex; the data flow path has fewer logic levels and requires less hardware because the designer has more freedom in combining serial operations to eliminate circuit levels; the circuit delay per level is faster because less hardware is required in the smaller, autonomous units. To implement the concept in the Model 91, the floating-point instruction set was separated into two subsets: add and multiply/divide. Table 1 shows a list of the instructions and identifies the unit in which each instruction is executed. With this separation, an add unit which executed all add class instructions in two cycles, and a multiply/divide unit which executed multiply in six cycles and divide in eighteen cycles, were designed.

The use of this concept somewhat changes the character of concurrent execution. It is possible to have concurrent execution with one execution unit—i.e., two arithmetic units, add and multiply/divide. The performance is not quite as good as that attainable using two execution units, but less hardware is required for the implementation. Therefore, more arithmetic units can be added to improve the performance. First, two add units and two multiply/divide units were considered. But the floating-point instruction unit can assign only one instruction per cycle. Therefore, since an add operation is two cycles long, two add units could be replaced by one add unit if a new add class instruction could be started every cycle. This would introduce still another example of concurrent execution: concurrent execution within an arithmetic unit.

Such concurrency within a unit is facilitated by the technique of pipelining. If a section of combinatorial logic, such as the logic to execute an add, could be designed with equal delay in all parallel paths through the logic, the rate at which new inputs could enter this section of logic would be independent of the total delay through the logic. However, delay is never equal; skew is always present and the interval between input signals must be greater than the total skew of the logic section. But temporary storage platforms can be inserted which will separate the section of combinatorial logic into smaller synchronous stages. Now the total skew has been divided into smaller pieces; only the skew between stages has to be considered. The interval between inputs has decreased and now depends on the skew between temporary storage platforms. Essentially the temporary storage platform is used to separate one complete job, such as an add, into several pieces; then

several jobs can be executed simultaneously. Thus, inputs can be applied at a predetermined rate and once the pipeline is full the outputs will match this rate.

The technique of pipelining does have practical limits, and these limits differ for each application. In general the rate at which new inputs can be applied is limited by the logic preceding the pipeline (e.g., add is limited to one instruction per cycle by the floating-point instruction unit) or by the rate at which outputs can be accepted. Also, both the rate of new inputs and the length of the pipeline are limited by dependencies among stages of the pipeline or between the output and successive inputs (e.g., the output of one add can become an input for the next).

The add unit requires two cycles for execution and is limited to one new input per cycle. Thus pipelining allows two instructions to be in execution concurrently, thereby increasing the efficiency with a small increase in hardware.

Further study of pipelining techniques would indicate that a three-cycle multiply and a twelve-cycle divide are possible. Here the technique of pipelining is used to speed up the iterative section of the multiply which is critical to multiply/divide execution. (This is discussed in detail in the section on the multiply/divide unit.)

The execution unit would consist at this point of a floating-point instruction unit, an add unit which could start an instruction every cycle, and a multiply/divide unit which would execute multiply in three cycles and divide in twelve cycles. However the performance still would not match the instruction unit. The execution times would be adequate but the units would spend considerable time waiting for operands. Therefore, instead of duplicating the arithmetic unit (which is expensive) extra input buffer registers have been added to collect the operands and necessary instruction control information. When both operands are available, the control information is processed and a request made to use an arithmetic unit. These registers are referred to as "reservation stations." They can be and are treated as independent units.

The final organization is shown in Fig. 2. It consists of three parts: the floating-point instruction unit; the floating-point add unit; and the floating-point multiply/divide unit. Another paper in this series[3] explains the floating-point instruction unit in detail. The problems involved and both the considered solutions and the implemented solutions are discussed. The floating-point add unit has three reservation stations and, as stated above, is treated as three separate add units, A1, A2 and A3. The floating-point multiply/divide unit has two reservation stations, M/D1 and M/D2. The last two sections of this paper describe the design of these two units in detail.

Floating-point terminology

The reader is assumed to be familiar with System/360 architecture and terminology.[2] However, the floating-point

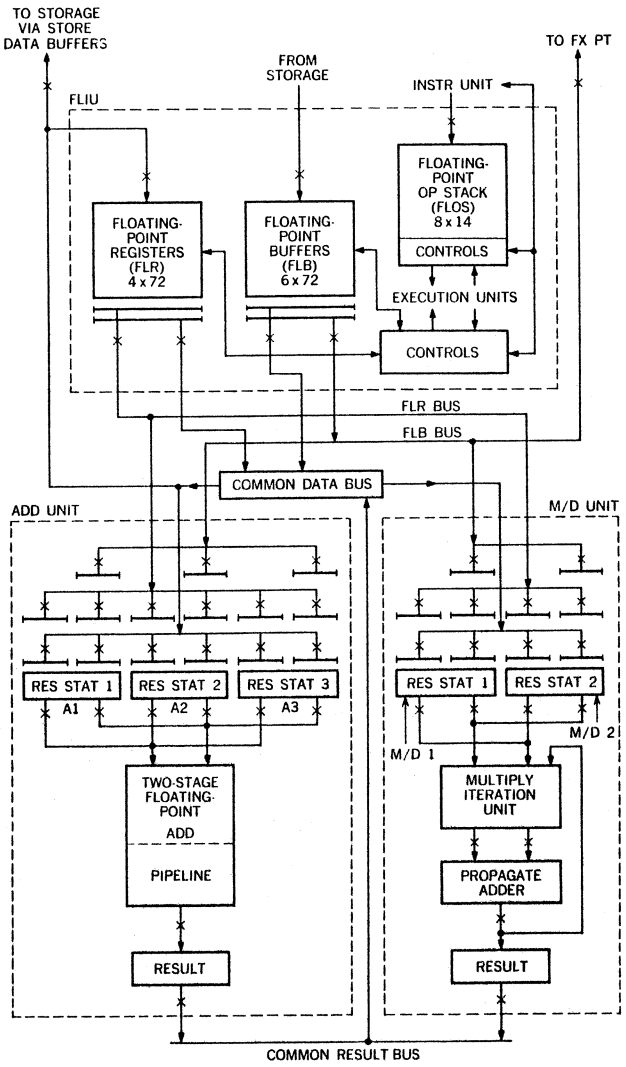


Figure 2 Overall organization of floating-point unit.

data format and terminology will be briefly reviewed here.

Floating-point data occupy a fixed-length format, which may be either a full-word short format or a double-word format:

Short Floating-Point Binary Format

Sign	Characteristic	Fraction
0	1 — — — — — — 7	8 — — — — — 31

Long Floating-Point Binary Format

Sign	Characteristic	Fraction
0	1 — — — — — 7	8 — — — — — — — 63

The first bit(s) in either format is (are) the sign bit(s). The subsequent seven bit positions are occupied by the charac-

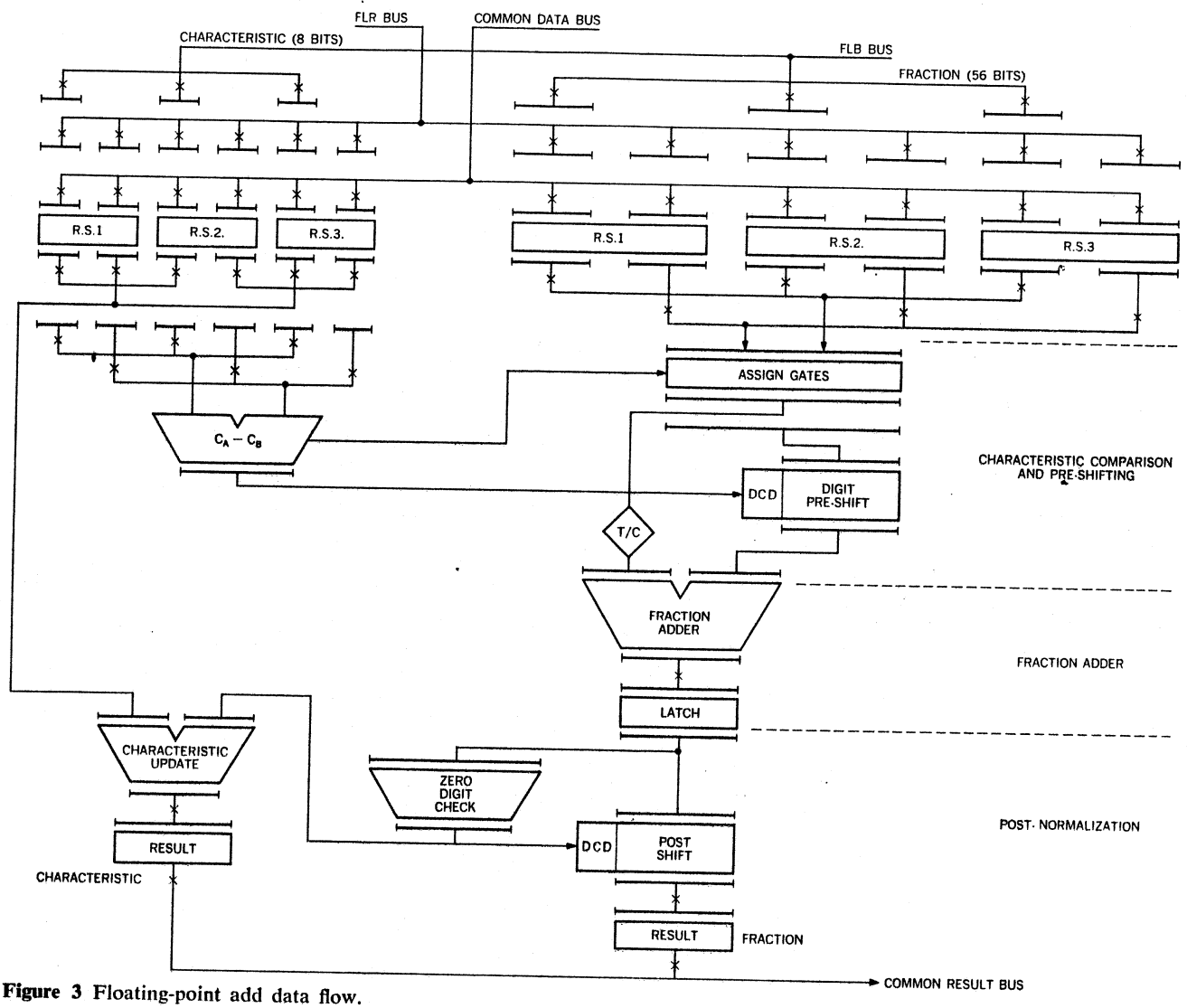


Figure 3 Floating-point add data flow.

teristic. The fraction consists of six hexadecimal digits for the short format or 14 hexadecimal digits for the long.

The radix point of the fraction is assumed to be immediately to the left of the high-order fraction digit. To provide the proper magnitude for the floating-point number, the fraction is considered to be multiplied by a power of 16. The characteristic portion, bits 1–7 of both floating-point formats, indicates this power. The characteristic is treated as an excess 64 number with a range from −64 through +63 corresponding to the binary expression of the values 0–127.

Both positive and negative quantities have a true fraction, the difference in sign being indicated by the sign bit. The number is positive or negative accordingly as the sign bit is zero or one.

A normalized floating-point number has a non-zero high-order hexadecimal fraction digit. To preserve maximum precision in subsequent operation, addition, subtraction, multiplication, and division are performed with normalized results. (Addition and subtraction may also be programmed to be performed with unnormalized results. The operands for any floating-point operation can be either normalized or unnormalized.)

Floating-point add unit

The challenge in the design of the add unit was to minimize the number of logical levels in the longest delay path. However, the sequence of operations necessary for the execution of a floating-point add impedes the design goal.

Consider the following operations:

(a) Since the radix point must be aligned before an add can proceed, the characteristics of the two operands must be compared and the difference between them established.
(b) This difference must be decoded into the shift amount, and the fraction with the smaller characteristic shifted right a sufficient number of positions to make the characteristics equal.
(c) Since subtraction is to be performed by forming the two's complement of one of the fractions and then adding the two fractions in the fraction adder, one of the fractions must pass through true/complement logic.
(d) The two operand fractions are added in a parallel adder. The carries must propagate from the low order end to the high order end.
(e) Because of subtraction, the output must provide for both the true sum and the complement sum, depending on the high-order carry.
(f) If the system architecture calls for left justification or normalized operation, the result out of the adder must be checked for high-order zeros and shifted left to remove these zeros.
(g) The characteristic must be reduced by the amount of left shift necessary to normalize the resultant fraction.
(h) The resultant operand must be stored in the proper accumulator.

The above sequence of operations implies a series of sequential execution stages, each of which is dependent on the output of the previous stage. The problem then, is to arrange, change and merge these operations to provide fast, efficient execution for a floating-point add.

None of the steps can be eliminated. Each step is required in order to execute add; but the steps can be merged so that the interface between them is eliminated,* and each step can be changed to provide only the necessary information to the next stage. For example, the long data format consists of 14 hexadecimal digits; therefore any difference between the two characteristics which is greater than 14 will result in an all zero fraction. This means that the characteristic difference adder need not generate a sum for the high-order three bits. Instead, if the difference is greater than 14, a shift of 15 is forced. As a result, the characteristic difference adder is faster and less expensive.

The add unit algorithm is separated into three parts: characteristic comparison and pre-shifting, fraction adder, and post-normalization (Fig. 3). The first section, the characteristic comparison and pre-shifting operation, merges the first three operations from the sequence given above; the second section—the fraction adder—merges the next two operations; the final section—post normalization—merges the final three operations. The hardware implementation of each of these three sections is discussed below.

* Levels are used to encode the output of one step, which is subsequently decoded in the next step. Merging the two steps will eliminate these levels.

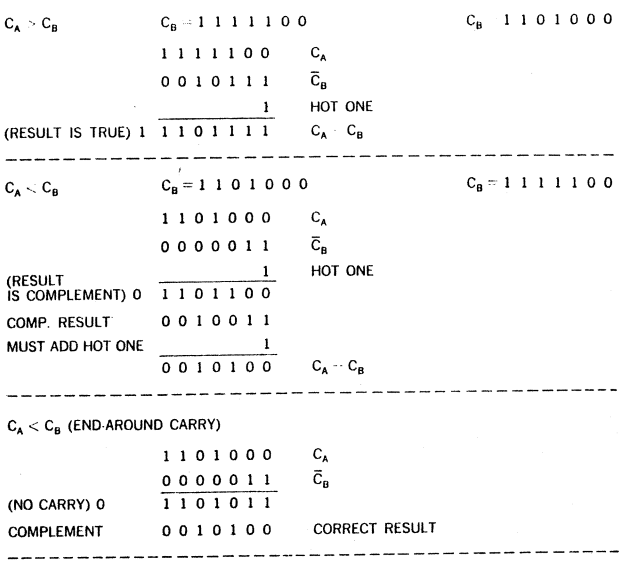


Figure 4 Examples of exponent arithmetic.

- *Implementation*

Characteristic comparison and pre-shifting

The first stage of execution for all two-operand instructions (floating-point add, subtract, and compare) is to compare the characteristics and establish the magnitude of the difference between them. The characteristic (C_B) of one operand is always subtracted from the characteristic (C_A) of the other operand ($C_A - C_B$). Characteristic B is always complemented as it is gated in at the reservation station.

If the output of the characteristic difference adder is the true sum or the complement of the true sum, the output can be decoded directly at the pre-shifter. But the adder always subtracts C_B from C_A and if $C_B > C_A$ the sum would be negative. Therefore, to eliminate the possibility of having to add a 1 in the low order position and complement when C_B is greater than C_A, an "end-around-carry" adder is used. This is shown by the example in Fig. 4.

The characteristic comparison can result in two states—$C_A \geq C_B$ or $C_B > C_A$. If $C_A \geq C_B$, there is a carry out of the high order position of the characteristic difference adder, and the carry is used to gate the fraction of operand B to the pre-shifter. The true sum output of the characteristic difference adder is the amount that the fraction must be shifted right to make the characteristics

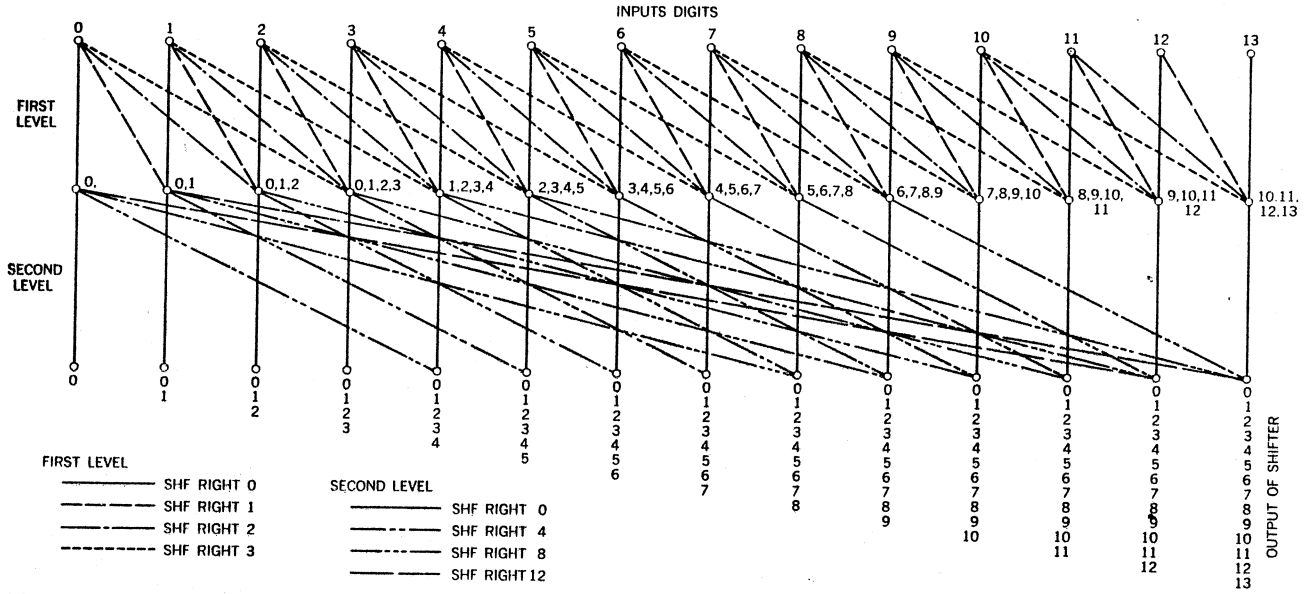


Figure 5 Digit pre-shifter.

equal. If $C_B > C_A$, there is no carry out of the high order position of the characteristic difference adder, and the absence of a carry is used to gate the fraction of operand A to the pre-shifter. In this case the complement of the sum output of the characteristic difference adder is the amount that the fraction must be shifted right to make the characteristics equal. In both cases the second operand fraction (the one with the larger characteristic) is gated to the true-complement input of the fraction adder.

The characteristic of the unshifted fraction becomes the resultant characteristic. It is gated to the characteristic-update adder, and after updating, if necessary, it is gated to the accumulator specified by the instruction.

The output of the characteristic difference adder is decoded by the pre-shifter and the proper fraction shifted right the necessary number of positions. The pre-shifter is a parallel digit-shifter which shifts each of the 14 digits right any amount from zero to fifteen. The decode of the shift amount is designed into each level, thereby eliminating serial logic levels for decoding.

The pre-shifter consists of two circuit levels. The first level shifts a digit right by 0, 1, 2 or 3 digit positions. The second level shifts a digit right by 0, 4, 8, or 12 digit positions. Thus, by the proper combination of these amounts any right digit shift between and including 0 and 15 can be executed. Figure 5 shows an example of the pre-shifter.

The un-shifted fraction is gated to the true/complement gates of the adder. Here the fraction is gated unchanged if the effective operation is ADD and complemented if the effective operation is SUBTRACT. The true/complement gating is overlapped with the pre-shifter on a time basis. The output of both the true/complement logic and the pre-shifter are the inputs to the fraction adder.

Fraction adder

Most of the time required for binary adders is carry propagation time. Two operands must be combined and the carries allowed to ripple from right (low order) to left (high order). The usual method of finding the sum is to combine the half sum* of bit n (higher order) with the carry from bit $n - 1$ ($S_n = A_n \forall B_n \forall C_n$).† The carry ($C_n$) into bit position n is also a three term expression which includes the carry into bit position $n - 1$

$$(C_n = A_{n-1} \cdot B_{n-1} \vee A_{n-1} \cdot C_{n-1} \vee B_{n-1} \cdot C_{n-1}).$$

If the carry term is rearranged to read

$$C_n = A_{n-1} \cdot B_{n-1} \vee (A_{n-1} \vee B_{n-1}) C_{n-1},$$

two new terms can be defined which separate the carry into two parts—generated carry, and propagated carry. The generated carry (G_{n-1}) is defined as $A_{n-1} \cdot B_{n-1}$, and the carry propagate function (often abbreviated to simply propagate or P_{n-1}) is defined as $A_{n-1} \vee B_{n-1}$. Now the

* The half sum is the exclusive OR of the two input bits, ($A_n \forall B_n$).
† The two operand fractions are designated as A, B and the bits as A_n, B_n, A_{n-1}, B_{n-1}, etc. C_n is the carry into bit position n, which is the carry out from bit $n - 1$.

carry expression can be rewritten as:[1,6]

$$C_n = G_{n-1} \vee P_{n-1}C_{n-1}$$

$$C_n = G_{n-1} \vee P_{n-1}G_{n-1} \vee P_{n-1}P_{n-2}C_{n-2}$$

$$C_n = G_{n-1} \vee P_{n-1}G_{n-1} \vee P_{n-1}P_{n-2}G_{n-2} \vee P_{n-1}P_{n-2}P_{n-3}C_{n-3}$$

$$\vdots$$

The expansion can continue as far as one desires and one could conceive of C_n being generated by one large OR block preceded by several AND blocks (in fact n AND blocks—one for each stage). But it is obvious that the limiting factor would be the circuit fan-in. Only a limited number of circuit stages can be connected together in this manner. This technique is defined as carry look-ahead, and by cascading different levels of look-ahead the technique can be made to fit the circuit fan-in, fan-out limitations.

For example, assume that four bits can be arranged in this manner, and that each four bits form a "group." The adder is now divided into groups and the carries and propagates can be arranged for carry look-ahead between groups just as they were for look-ahead between bits. It is possible to carry the concept even further and define a section as consisting of one or more groups. Now the adder has three levels of carry look-ahead: the bit level of look-ahead, the group level, and the section level.

The fraction adder of the floating-point add unit is a carry look-ahead adder. A group is made up of four bits (one digit) and two groups form a section. Since it must be capable of adding 56 bits, the fraction adder consists of seven sections and 14 groups. Each pair of input bits generate the three bit functions: half-sum ($A \veebar B$), bit carry generate ($A \cdot B$) and bit propagate ($A \vee B$). These functions are combined to form the group generate and propagate which in turn are combined to form the section generate and propagate. A typical group is shown in Fig. 6 and the group and section look-ahead are shown in Fig. 7.

The high-order sum consists of nine bits to include the end-around carry for subtraction and the overflow bit for addition. The end-around carry is needed for subtraction because the fraction which is complemented may not be the subtrahend. This is illustrated by the example given in the description of the characteristic comparison. If the effective sign of the instruction is minus (the exclusive OR of the sign of the two fractions and the instruction is the effective sign) the effective operation is subtract. Also, the high-order bit (ninth bit of the high order section) is set to a one, thus conditioning it for an end-around-carry. If there is no end-around-carry when the effective sign is minus the adder output is complemented.

Post-normalization

Normalization or post-shifting takes place when the intermediate arithmetic result out of the adder is changed to the final result. The output of the fraction adder is checked for high-order zero digits and the fraction is left-shifted until the high-order digit is non-zero.

The output of the fraction adder is gated to the zero-digit checker. The zero-digit checker is simply a large decoder, which detects the number of leading zero digits, and provides the shift amount to the post-shifter. Since this same amount must be subtracted from the characteristic, the zero-digit checker also must encode the shift amount for the characteristic update adder.

The implementation of the digit post-shifter is the same as the digit pre-shifter except for the fact that the post-shift is a left-shift. The first level of the post-shifter shifts each of the 14 digits left 0, 1, 2 or 3 and the second level shifts each digit 0, 4, 8, or 12. The output of the second level is gated into the add unit fraction result register, from which the resultant fraction is routed to the proper floating-point accumulator.

The characteristic update is executed in parallel with the fraction shift. The zero-digit checker provides the characteristic update adder with the two's complement of the amount by which the characteristic must be reduced. Since it is not possible to have a post-shift greater than 13, the high-order three bits of the characteristic can only be changed by carries which ripple from the low order four bits. The update adder makes use of this fact to reduce the necessary hardware and speed up the operation.

Floating-point multiply/divide unit

Multiply and divide are complicated operations. However, two of the original design goals were to select an algorithm for each operation such that (1) both operations could use common hardware, and (2) improvement in execution time could be achieved which would be comparable to that achieved in the floating-point add unit. Several algorithms exist for each instruction which make the first design goal attainable. Unfortunately, the best of the algorithms generally used for divide are not capable of providing an improvement in execution comparable to the improvement achievable by those used for multiply. The algorithm developed for divide in the Model 91 uses multiplication as the basic operator. Thus, common hardware is used, and comparable improvement in the execution time is achieved.

In order to give a clear, consistent treatment to both instructions, this section discusses the multiply algorithm and hardware implementation first. Then the divide algorithm is discussed separately. Finally, it is shown how divide utilizes the multiply execution hardware and the hardware which is unique to the execution of divide is described.

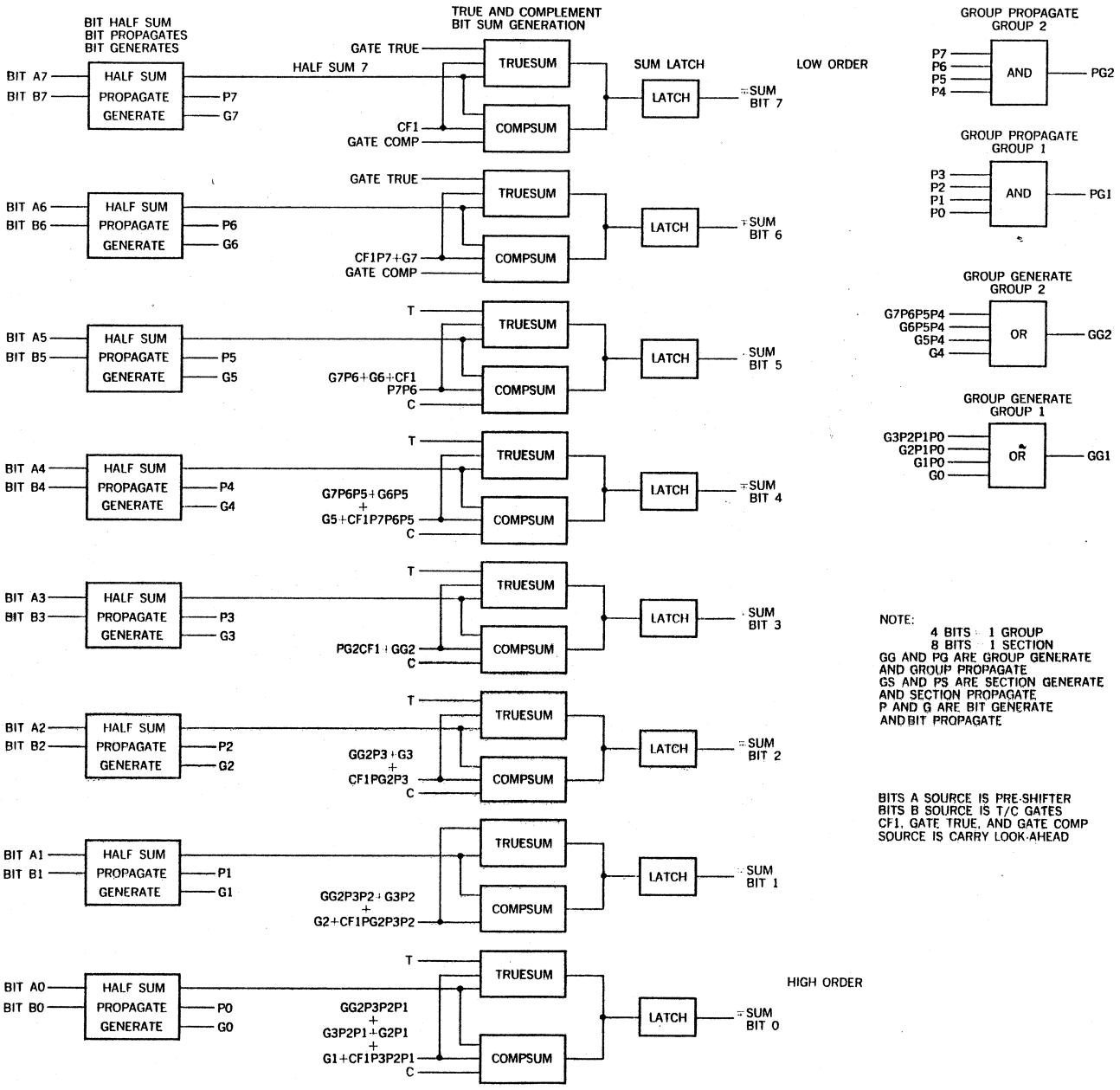


Figure 6 Fraction adder, section 1 (high-order).

- *Multiply algorithm*

Computers usually execute multiply by repetitive addition, and the time required is dependent on the number of additions required.[1,6] A zero bit in the multiplier results in adding a zero word to the partial product. Therefore, because shifting is a faster operation than add, the execution time can be decreased by shifting over a zero or a string of zeros. Any improvement in the multiply execution beyond this point is not obvious. However, certain properties of the binary number system combined with complementing to allow subtraction as well as addition can be used to reduce the number of necessary additions.

An integer in any number system may be written in the following form:

$$a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b^1 + a_0 b^0,$$

where

$0 \leq a \leq b - 1$, and b = base of the number system

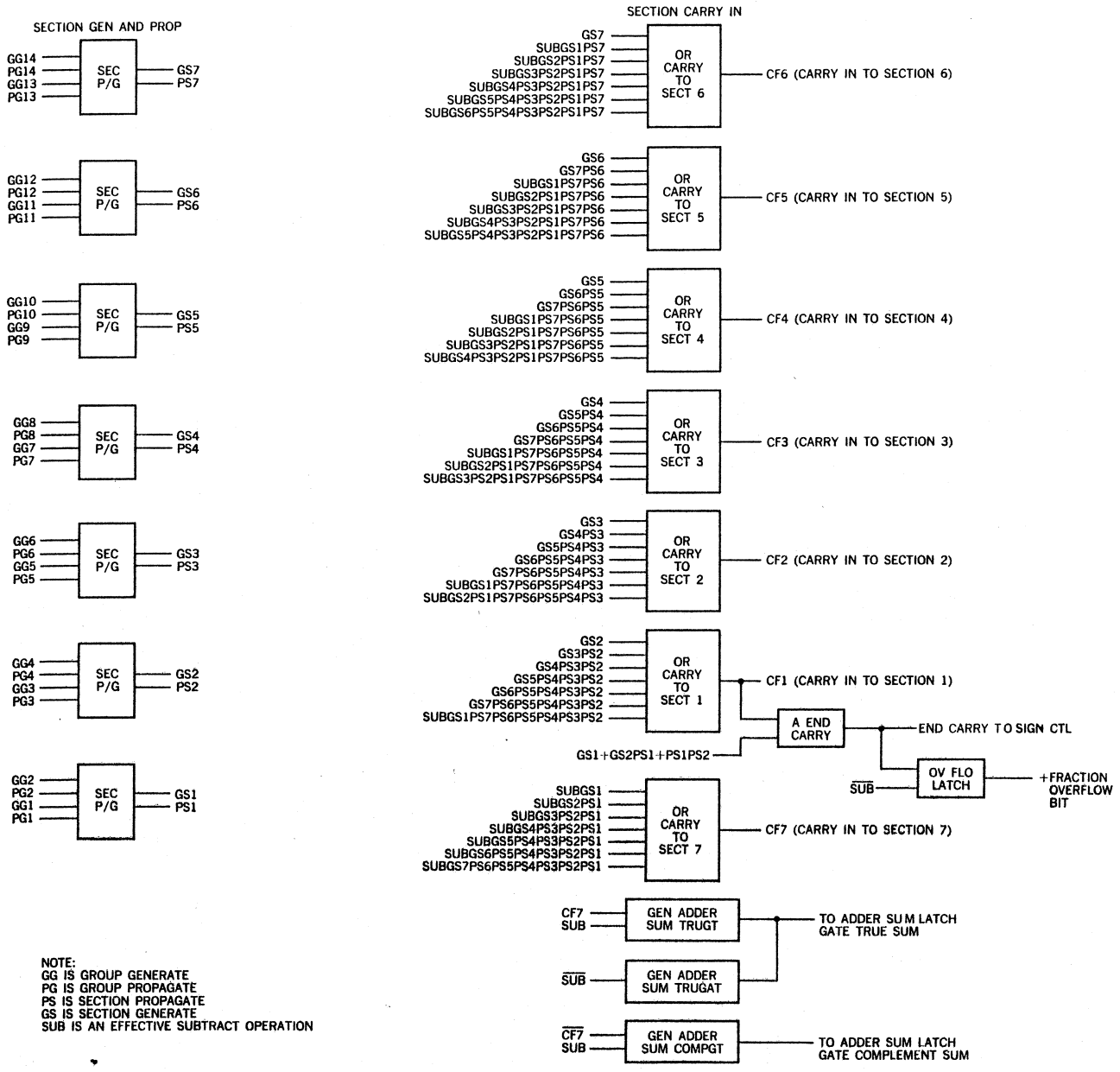


Figure 7 Fraction adder, carry look-ahead.

One of the properties of numbering systems which is particularly interesting in multiply is that an integer can be rewritten as shown below.

$$a_n b^n + a_{n-1} b^{n-1} + \cdots + a_k b^k + \cdots + a_{n-x} b^{n-x},$$

where

$$a_k = b - 1 \text{ for any } k.$$

In the binary number system a_k can take only the values 0 and 1. Thus, using the above property, a string of 1's can be skipped by subtracting at the start of the string and adding at the end of the string:

$$112_{10} = 2^6 + 2^5 + 2^4 = 2^7 - 2^4,$$

$$112_{10} = 1110000_2 = 10000000_2 - 10000_2.$$

Therefore, a string of 1's in the multiplier can be reduced from an addition for each 1 in the string to a subtraction for the first 1 in the string, shift the partial product one position for each 1 in the string, and an addition for the last 1 in the string.

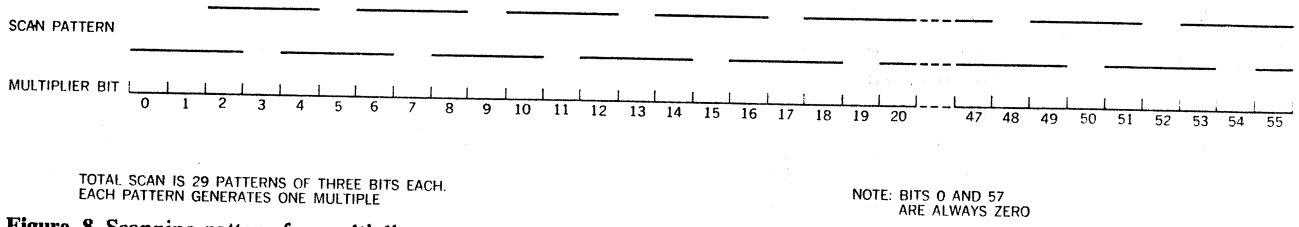


Figure 8 Scanning pattern for multiplier.

However, the method described above requires a variable shift and thus does not permit one to predict the exact number of cycles required to execute multiply. Furthermore, it does not permit the use of carry-save adders in the implementation. (Carry-save adders will be discussed later.)

A multiplier recoding-algorithm, which is based on the property described above, but which uses uniform shifts is used in the Model 91. The multiplier is divided into uniform groups of k bits each. These k bits are recoded to generate a multiple of the multiplicand, which is added to or subtracted from the partial product. The multiples are generated by shifting the position of the multiplicand in relation to the normal position at which it would enter the adder for a k equal to one. After adding the generated multiple to the partial product, the partial product is shifted k positions and the next group of k bits is considered.

The correct choice for k is important since an average of $1/2^k$ of the generated multiples will have a value of zero, and increasing k (over k equal to one) reduces the amount of operand reduction capability that is used inefficiently. However, if k is greater than two, carry propagate addition is necessary to generate the needed multiplicand multiples (shifting can only be used to generate multiples which are a power of two). In the context of a fast multiply, the carry-propagate adder increases the start-up time, which is undesirable. The Model 91 uses a k equal to two.

The technique used to scan the multiplier is shown in Fig. 8. Overlapping the high-order bit of one group and the low-order bit of the next group insures that the beginning and end of a string of 1's is detected once and only once. Table 2 shows which multiples are selected for all possible combinations of the two new bits and the overlapped bit.

Since the objective is fast multiply execution, six groups of multiplier bits are recoded at one time, and the resultant six multiples are added to the partial product. Five iterations are sufficient to assimilate the full 56 bits of the multiplier fraction. Figure 9 shows how the multiplier fraction is separated for each iteration and how each iteration is separated for the six generated multiples.

A tree of carry-save adders is used to reduce the generated multiples from six to two. A carry-save adder, which can be used whenever successive addition of several operands is necessary, requires less hardware, has less data skew and has less delay than a carry-propagate adder.[6] The individual carry-save adder takes three input operands and generates the resulting sum and carry. However, instead of connecting the carries to the next higher-order bits and allowing them to ripple, they are treated as independent outputs. In accordance with the customary rules for addition, the carries will be added to the next higher-order bits as separate inputs to the next carry-save adder down the tree.

Figure 10 illustrates a tree of carry-save adders which will reduce six input operands to two, thereby retiring 12 bits of the multiplier on each iteration. Note that the final output of the carry-save adder tree is two operands—sum and carry—which are shifted right 12 positions and loop back to become input operands. Thus, the partial product is accumulated as a partial sum and a partial carry. After the multiplier has been assimilated, these two operands, sum and carry, are added in a carry propagate adder to form the final product.

- *Implementation*

A block diagram of the data flow for the execution of a multiply is shown in Fig. 11. This data flow can be separated into two parts, the iterative hardware and the peripheral hardware (that hardware which is peripheral to the iterative hardware). The latter includes the input reservation stations, the pre-normalizer, the post-normalizer, the propagate adder, the result register, and the characteristic arithmetic. The peripheral hardware is described first, but since the iterative hardware is the heart of multiply execution, the major part of this section is devoted to a discussion of this hardware.

Input peripheral hardware

The input hardware includes the reservation stations, pre-normalizer, and the characteristic arithmetic. As was stated earlier, the multiply unit has two reservation stations and appears to the floating-point instruction unit for assign-

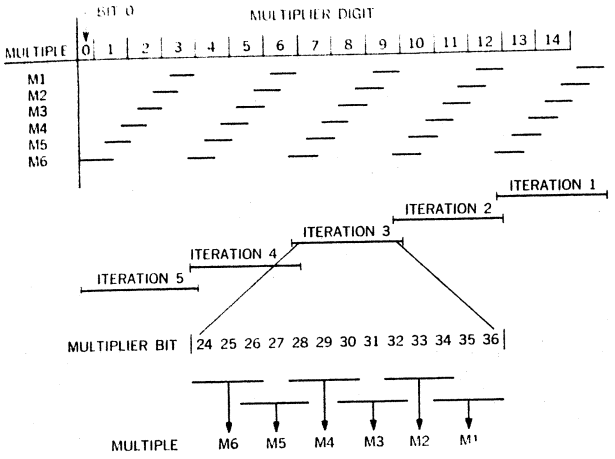


Figure 9 Iterations and multiple generation for multiply.

ment purposes as two distinct multiply units. If both units have been selected for a multiply operation, the first unit to receive both operands is given priority to begin execution. In the case where both units receive their second operand simultaneously, the unit which was selected by the floating-point execution unit first is given priority for execution.

The system architecture specifies that multiply is a normalized operation. Thus, if the input operands are unnormalized, they must be gated to the pre-normalizer, normalized, and then returned to the originating reservation station. In some cases, one additional machine cycle is added to the execution time for each unnormalized operand. However, normalization takes place as soon as the first operand enters the reservation station, provided there is not an operation in execution. Thus, normalizing can take place while the unit is waiting for the second operand.

The design of the zero digit detector and the left-shifter are similar to those described earlier for the add unit. If the zero digit detector, detects an all-zero fraction, the multiply is executed normally, but the outgate of the result to the floating-point accumulator is inhibited. Thus the required result, and all-zero-fraction, is stored.

The amount of left shifting necessary to normalize an operand is gated to the characteristic arithmetic logic, where the characteristic is updated for this shift. Characteristic arithmetic for multiply simply requires the two characteristics to be added but this operation can be overlapped with the execution of the multiply. Thus, the implementation is simple and straightforward.

It remains only to update the characteristic because of post-normalization. The post-shift can never be more than one digit because the input operands are normalized. Therefore, in order to eliminate logic levels at the end of multiply execution, two characteristics are generated: the normal resultant characteristic and the normal characteristic minus one. Subsequent to post-normalization the correct characteristic is outgated.

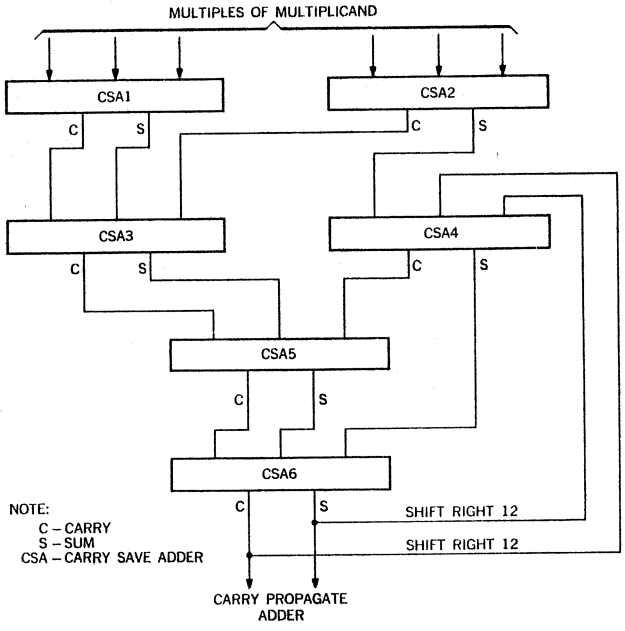
Output peripheral hardware

The output peripheral hardware includes the carry-propagate adder, the result register and the post-normalizer. Since the product is accumulated as two operands (sum and carry) the output of the iterative hardware is gated to a carry-propagate adder to form the final product. The design of the carry propagate adder is similar to the one used in the add unit with the exception that multiply does not require an end-around carry adder. A result register is created by latching the last level of the carry propagate adder. The output of the result register is gated to the common data bus via the post-normalizer. Detection of the need for post-normalization is done in parallel with the carry propagate adder and the result is gated to the common data bus, either shifted left one digit or unshifted.

Iterative hardware

The multiply execution area has conflicting design goals. The execution time must be short but the amount of hardware necessary for implementation has a practical upper limit. One could design a multiply unit which would take two cycles for execution. A large tree of twenty-eight carry-save adders could be interconnected so that the multiplicand and the multiplier would be the input to the tree and the output would be the product.[8] The performance of this multiply unit would be acceptable but

Figure 10 Carry-save adder tree.



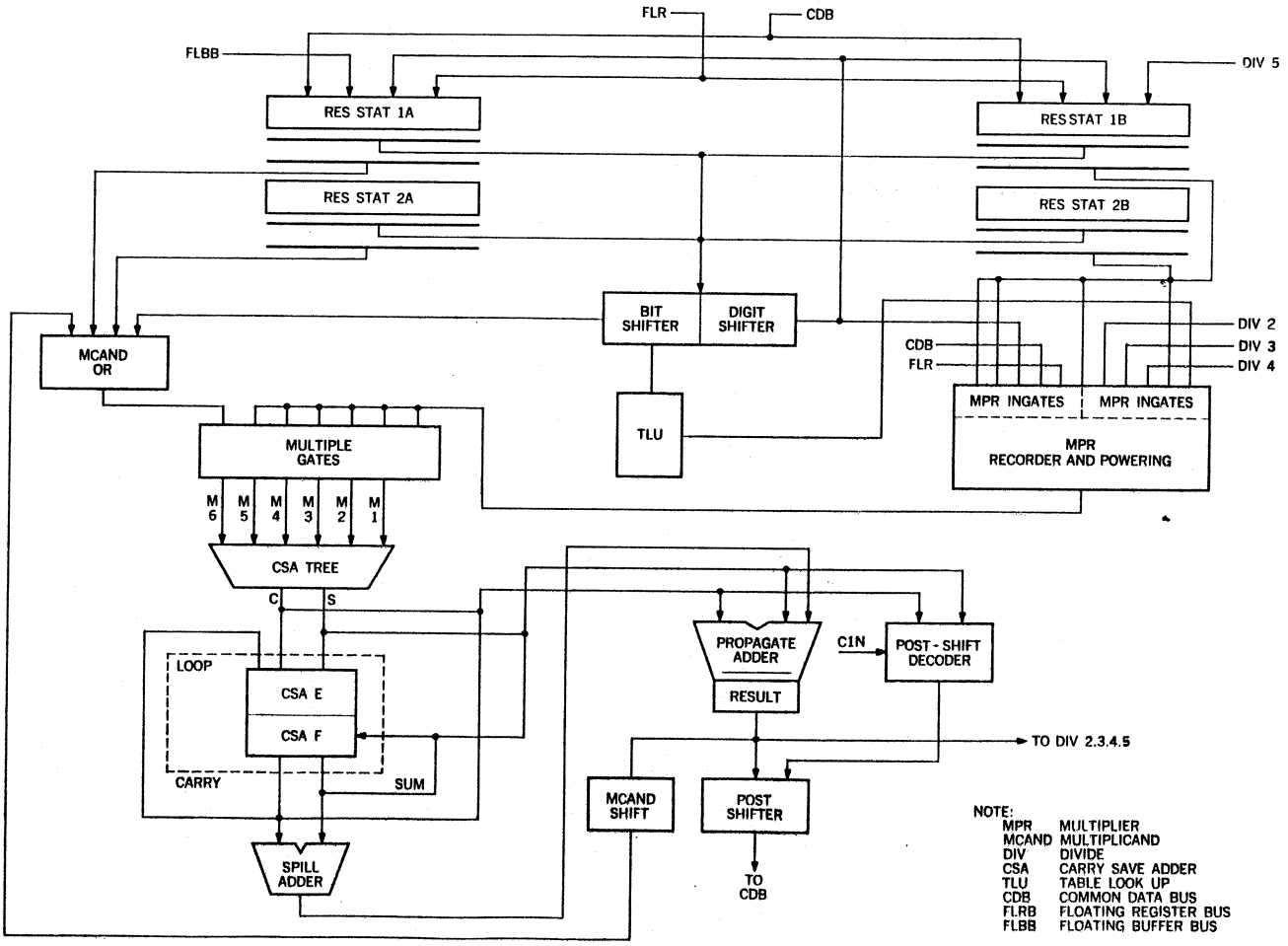


Figure 11 Floating-point multiply/divide data flow.

the amount of hardware necessary for implementation is much too high.

The adopted alternative approach was to select a subset of the carry-save adder tree such that one iteration through the tree retires 12 bits of the multiplier. This iteration is repeated until the full 56 bits of the multiplier have been exhausted. If each iteration is fast enough, the multiply execution time for this method approaches that for the large tree of carry-save adders. In fact, if each iteration can be 20 nanoseconds the second method can execute a multiply in three cycles, and the iterative hardware can be reduced to 20% of that required for the first method. Thus, with an iterative loop, the primary design problem is to design the carry-save adder tree so that the iteration period is minimized. The faster the repetition rate of the iterative hardware, the better the cost-performance ratio of the multiply area.

There are several ways to arrange the carry-save adders, and each method affects the iteration period differently. For example, if they are arranged as shown in Fig. 12, the feedback loop (the partial product) is from the output back to the input. In this case, the iteration period becomes the time required to make one complete pass through the tree. However, the adopted arrangement, shown in Fig. 13, allows the iteration period to approach the delay through the last carry-save adders (these two carry-save adders are accumulating the partial product). But the delay through the path leading to the last two carry-save adders (the multiplier recoding, multiple generation and the first four carry-save adders) is much longer than the delay through the adders. If, however, temporary storage platforms are inserted in the iterative loop the concept of pipelining, explained earlier, can be put to use here. Temporary storage platforms are inserted in the iterative hardware for deskewing so that the rate of inserting new inputs (twelve bits of the multiplier) and the

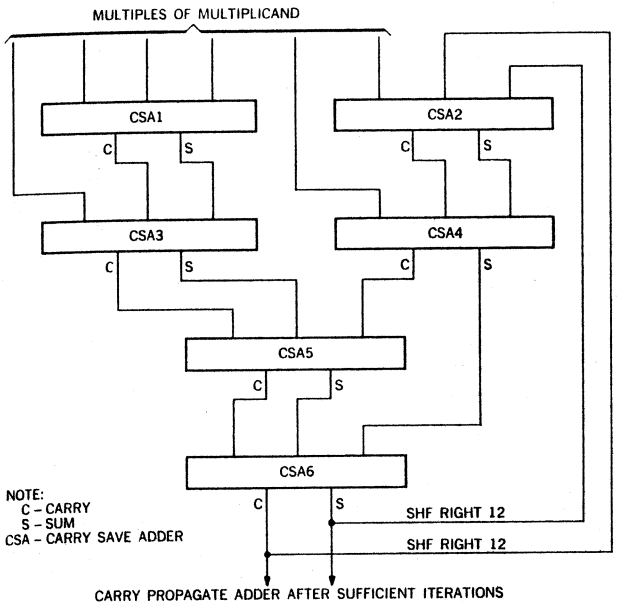


Figure 12 CSA tree with feedback loop from output.

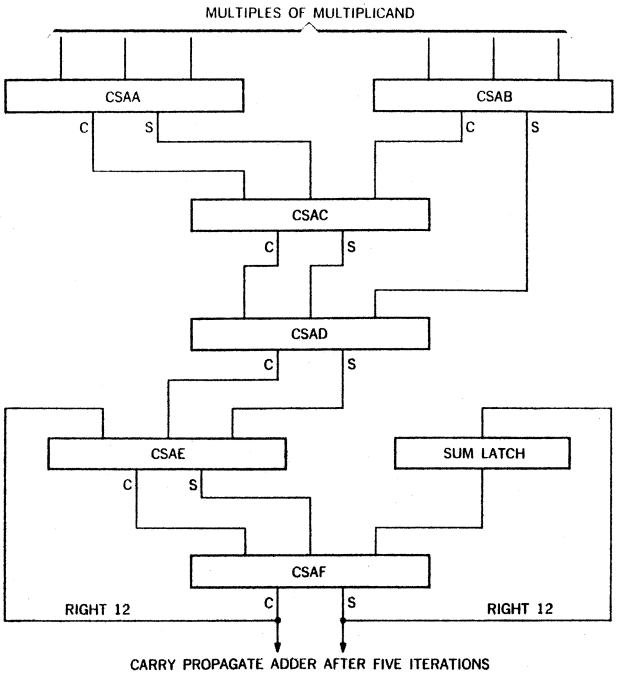


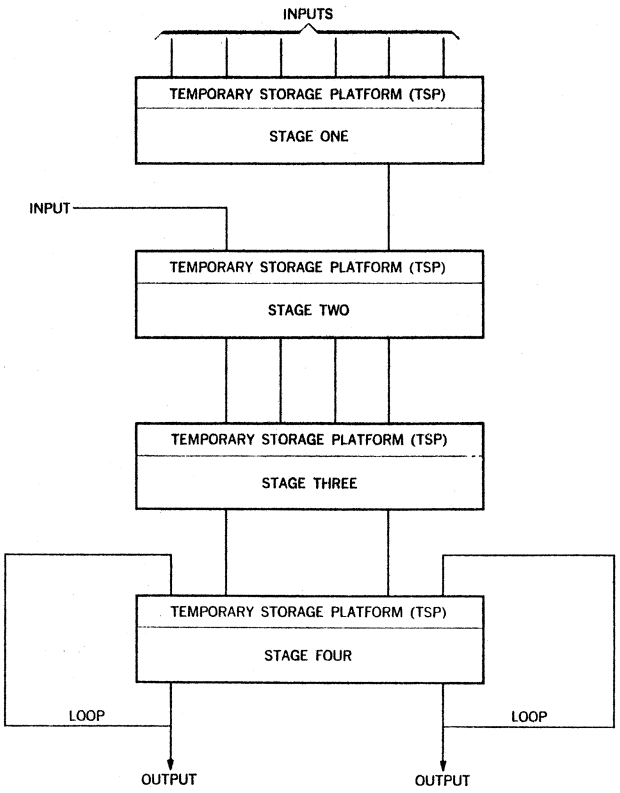
Figure 13 CSA tree with accumulating loop at output.

rate of accumulating the partial product may safely be made equal. Therefore, by pipelining the carry-save adder tree, the second arrangement can be used and the iteration period is equal to the delay through the last two carry-save adders.

In order to explain the pipelined tree, the path is abstracted in Fig. 14. Each block represents the logic associated with the stages of the pipeline and the first level of each block represents the temporary storage platform. The period of the clock is set by the logic delay of the accumulating loop. In the abstract design the logic delay of all paths between stages of the pipeline is assumed to be the same as the clock period.

Figure 15 is a timing diagram for the abstracted iterative hardware. At clock time zero, the first input, I_1, is gated into the temporary storage in stage one. At clock time one, I_1, after being operated on by the logic in stage one, is gated in at stage two and I_2 is gated in at stage one. This process continues until at clock time three, the original input, I_1, is entering stage four. During this clock time, the pipeline is filled, i.e., each stage of the pipeline now contains data in various forms of completion. At clock time four, the last input, I_5, enters stage one, and the partial product starts to accumulate at stage four. The next three clock times are used to drain the pipeline and accumulate the full partial product. Thus the total iterative loop time is that necessary to fill up the carry-save adder

Figure 14 Abstract drawing of "pipelined" iteration.



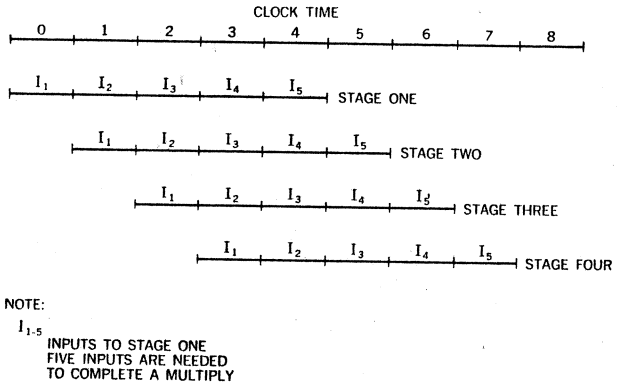


Figure 15 Timing diagram for abstracted iterative hardware.

tree plus five passes around the accumulating loop, or eight clock periods. If the feedback loop were from output to input, as shown in Fig. 12, the total iterative loop time would be twenty clock periods. Therefore the iterative loop time has been reduced by a factor of 2.5, with only a small increase in hardware. (This is described later.)

The actual implementation of the pipeline is not simple. First, the temporary storage platforms require extra hardware and add delay to the path. Second, the placement of the temporary storage platforms is important for two reasons: (1) The purpose of the temporary storage platform is to deskew the logic (difference between fast and slow logic paths) and the logic delay is not ideally distributed, and (2) the placement can affect the amount of hardware necessary for implementation.

The solution to the first problem led to a design in which the logic function was designed into the temporary platform; e.g., a latched carry-save adder or a latched multiple gate. The extra hardware is only that required for the feedback loop which latches the logic function; the added delay is eliminated because the logic function is designed into the temporary storage. The solution to the second problem was more complex. First, the clock used to control the temporary storage platform ingate was designed as a series clock. All of the pulses of an iteration are initiated by a single oscillator pulse and then delayed to drive the ingates of the successive pipeline stages. The clock delay between successive temporary storage ingates is equal to the long path circuit and wiring delay of the logic between these ingates. The time between iterations (the oscillator period) is still the delay of the accumulating loop, but the time between pipeline stages is not equal to the clock period. This allows the placement of temporary storage to vary without being dependent on the clock.

The relationship between the logic skew and clock period can be expressed as

Short path > [long path − clock period] + gate width,

where short path is the shortest logic delay between two temporary storage platforms; long path is the longest logic delay between two temporary storage platforms; and gate width is the time necessary to set and latch the temporary storage platform.

The temporary storage platforms were placed to minimize the hardware; then a careful data path analysis was made to determine the logic skew. The above relationship was next applied and the short paths "padded" with additional delay to satisfy the relationship. The result is shown in Fig. 16. The temporary storage platforms are at the multiplier recorder, the multiple gates, carry-save adder C and the accumulating loop, and carry-save adders E and F.

Since the design goal was to make the iteration period as short as possible, the design of the last two carry-save adders required a minimum number of levels and was constrained to account for the "short path around the loop." Carry-save adders E and F are each designed as a temporary storage platform and are orthogonal—i.e., are not ingated simultaneously. The first, carry-save adder E, is ingated on the first-half of the clock period and the second, carry-save adder F, is ingated on the second half of the clock period.

The low order thirteen bits of the multiplier are gated into the latched multiplier recoder at clock time zero and recoded to six control lines. Every clock period—20 nanoseconds—a new set of bits is gated into the multiplier recoder until the full word (56 bits) is exhausted. The next step in the pipeline is the latched multiple gates. Six multiples are generated by shifting the multiplicand, under control of the output from the multiplier recoder. These six multiples are reduced to four (two sums and two carries) by carry-save adders (CSA) A and B. Carry-save adder C takes three of these outputs and reduces them to two latched outputs. The sum from CSA-B is latched in parallel with CSA-C and combines with the two outputs from CSA-C to provide CSA-D with three inputs. At the output of CSA-D, the sum and carry are the result of multiplying twelve bits of the multiplier and the full multiplicand. The next two latched carry-save adders are used to accumulate the partial product. Each iteration adds the latest sum and carry from CSA-D to the previous results. After five iterations of the accumulating loop the output of CSA-F is the bit product in carry-save form. Now the sum and carry operands are gated to the carry propagate adder and the carries allowed to ripple to form the final product.

- *Divide algorithm*

Several division algorithms exist,[1,6] of varying complexity, cost and performance, which could be used to execute the divide instruction in the Model 91. But because of the relatively complex and iterative nature of divide

algorithms, the execution time is out of balance with other processor functions. Even the higher-performing conventional algorithms contain a shortcoming which requires that successive subtractions be separated by a performance-degrading decode interval.* The Model 91, however, utilizes a unique divide algorithm which is based on quadratic convergence.[7,8,9,10] A major advantage is that the number of required iterations is reduced (proportional to $\log_2$ of the fraction length), which reduces the number of data-control interactions. Another important advantage is that MULTIPLY is the basic iterative operator. This both reduces the cost, by exploiting existing hardware, and enhances the execution time, because in the Model 91 MULTIPLY is extremely fast.

The divisor and dividend are considered to be the denominator and numerator of a fraction. On each interation a factor, R_k, multiplies both numerator and denominator so that the resultant denominator converges quadratically toward one (1) and the resultant numerator converges quadratically toward the desired quotient.

$$\frac{N}{D} \times \frac{R}{R} \times \frac{R_1}{R_1} \times \frac{R_2}{R_2} \times \cdots \times \frac{R_n}{R_n}$$

$$\Rightarrow NRR_1 \cdots R_n = \text{Quotient},$$

where N = numerator = dividend,
D = denominator = divisor, and
$D\,R\,R_1\,R_2 \cdots R_n \Rightarrow 1$.

The selection of the factor R_k is the essential part of the procedure and is based on the following: The divisor can be expressed as

$$D = 1 - x,$$

where $x \leq 1/2$ since D is a bit-normalized, binary floating-point fraction of the form

0.1 xxx $\cdots$.

Now, if the factor R is set equal to $1 + x$ and the denominator is multiplied by R

$$D_1 = DR = (1 - x)(1 + x) = 1 - x^2,$$

where $x^2 \leq 1/4$, since $x \leq 1/2$.

The new denominator is guaranteed to have the form 0.11 xxxx $\cdots$. Likewise, selecting $R_1 = 1 + x^2$ will double the leading 1 on the next iteration to yield

$$D_2 = D_1R_1 = (1 - x^2)(1 + x^2) = 1 - x^4$$

$$= 0.1111\text{xxxx} \cdots,$$

where $x^4 \leq 1/16$ since $x \leq 1/2$.

* Conventional refers to previous division algorithms which use subtraction as the iterative operator. The faster algorithms generate more than one quotient bit in parallel through the use of pre-wired multiplies. However, the selection of the multiplies for the next iteration is dependent upon a decode of the partial remainder of the previous iteration.

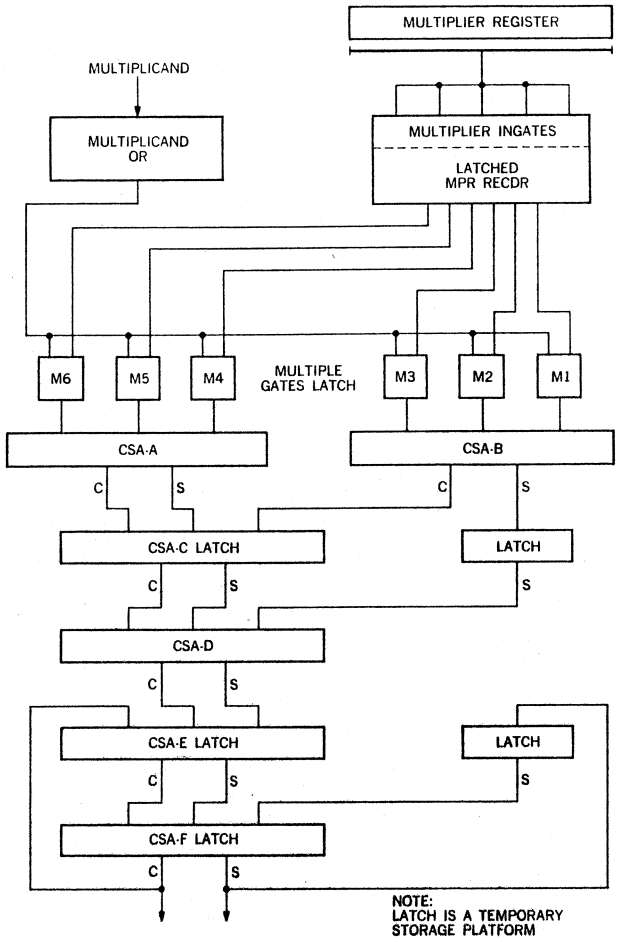


Figure 16 Multiply iterative loop showing temporary storage.

In general, if $x_k < 1/2^n$ then $x_{k+1} < 1/2^{2n}$. Thus, by continuing the multiplication until x_k is less than the least significant bit of the denominator (divisor fraction), the desired result, namely a denominator equivalent to one (0.1111111 $\cdots$ 111), is obtained.

It is important to note that the multiplier for each iteration is the two's complement of the denominator,

$$R_{k+1} = 2 - D_k = 2 - (1 - x_n) = 1 + x_n.$$

Thus the multiplier for iteration k is formed by taking the two's complement of the result of iteration $(k - 1)$. However, in this form the algorithm is still not fast enough. For a 56-bit fraction, eleven multiples are required with a two's complement inserted between six of the multiples:

$$Q = NRR_1R_2R_3R_4R_5$$

$$R_5 = 2 - D_4 \quad \text{and} \quad D_4 = DRR_1R_2R_3R_4.$$

Table 2 Multiplier recoder rules.

Input			Output multiple	Reason
n*	$(n+1)$	$(n+2)$		
0	0	0	0	No string
0	0	1	+2	End of string
0	1	0	+2	Beginning and end
0	1	1	+4	End of string
1	0	0	−4	Beginning of string
1	0	1	−2	Beginning and end
1	1	0	−2	Beginning of string
1	1	1	0	Center of string

* Bit n is the high-order position

But if the number of bits in the multiplier could be reduced, the time for each multiply would be decreased. If in order to obtain n bits of convergence the multiplier is truncated to n bits [$1 + x_T$ where $(x_T - x) < 2^{-n}$] it can be shown that the resultant denominator is equivalent to

$$(1 + x_T)(1 - x) = 1 - x^2 + |T|,$$

where $0 < T$ (which is due to truncation) $< 2^{-n}$.

Because the additional term T is always positive, the resultant denominator can now have two forms:

$$D_k = \begin{cases} 0.11111 \cdots \text{xxxxx} \cdots \\ 1.00000 \cdots \text{xxxxx} \cdots \end{cases}.$$

The denominator can converge toward unity from above or below, but it will converge, so no additional problems are encountered.

Therefore, the number of bits in the multiplier can be reduced to the string bits (all 0 or all 1) and the number of bits of convergence desired. The string bits, since they are all 0 or all 1, can be skipped in the multiply. Thus the multiply time has been improved considerably and so, consequently, has the divide time. To improve the initial minimum string length, thus reducing the number of iterations, the first multiplier, R, is generated by a table-lookup which inspects the first seven bits of the divisor. The first multiply guarantees a result which has seven similar bits to the right of the binary point ($1 \pm x$ has the form $\bar{a}.aaaaaaa \cdots$ etc.).

The following sequence outlines the operations which result in the execution of a divide.

1. Bit normalize the divisor and shift the dividend accordingly.
2. Determine the first multiplier, R, by a table-lookup.
3. Multiply D by R forming D_1.
4. Multiply N by R forming N_1.
5. Truncate D_k and complement to form R_k.
6. Multiply D_k by R_k forming D_{k+1}.
7. Multiply N_k by R_k forming N_{k+1}.
8. Iterate on 5, 6 and 7 until $D_{k+n} \Rightarrow 1$ and then $N_{k+n} =$ Quotient.

• *Divide implementation*

Each iteration of divide execution consists of three operations as shown above. The problem in implementation is to accomplish these three operations utilizing the multiply hardware described previously and accomplish them in the minimum amount of time. But there are three points which create difficulty. First, the multiplier is a variable length operand, the length being different on each iteration. The first multiplier, determined by table-lookup, is ten bits and yields a minimum string length of seven; the second multiplier is fourteen bits; the third multiplier is twenty-eight bits, etc. In other words, the minimum string length can be doubled on each iteration after the first. Second, the result of one iteration is the multiplicand for the next iteration. Since the output of the multiply iterative hardware is two operands—carry and sum—the carry propagate adder must be included in the divide loop. Third, two multiplies are required in each iteration—one determines what to do on the next iteration (multiplier × denominator) and one converges the numerator towards the quotient (multiplier × numerator).

When all three of these points are considered simultaneously they present a dilemma. Since two multiplies are necessary it is desirable to overlap the two and save time, but any multiply for which the multiplier is greater than twelve bits requires that the carry-save adder loop be used. Also, the fact that the carry propagate adder must be included in the loop lengthens the time for each iteration. Several design iterations were required before arriving at the correct solution.

First consider the entries in Table 2 and note that the leading string of 1's or 0's in the multiplier can be skipped since they result in a zero multiple out of the multiplier recoder. Also, if the input of the multiplier recoder is complemented the sign of the output changes but the magnitude remains the same. Thus, this property can be used to produce $\mp x_n$ at the output of the recoder.

Next consider a multiplier (complement of truncated denominator) such as the following:

$$\begin{array}{llll} 1. & 0000 & 0000 & 000\left[\begin{array}{llll} 0 & 00xx & xxxx & xxx1 \\ 1 & 11xx & xxxx & xxx1 \end{array}\right] \\ 0. & 1111 & 1111 & 111 \end{array}$$

If *all* positions were recoded, a bit of value 1 would be recoded from the high-order end and a set of bits of value $\mp x_k$ from the right end ($1 \pm x_k$). However, if only the

Table 3 Formats of the denominators and their multipliers.

Digit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
D	0	1xxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0000	0000
R	[01	xxxx	xxxx	xx0]	Determined by table lookup of denominator												
$D \times R = D_1$	0	1111	111x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
	1	0000	000x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
R_1	1	0000	[000$\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}$11	1]	Determined by complementing denominator										
	0	1111	[111$\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}$11	1]											
$D_1 \times R_1 = D_2$	0	1111	1111	1111	11xx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
	1	0000	0000	0000	00xx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
R_2	1	0000	0000	0000	00$\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}$1]										
	0	1111	1111	111[1	11$\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}$1]										
$D_2 \times R_2 = D_3$	0	1111	1111	1111	1111	1111	111x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
	1	0000	0000	0000	0000	0000	000x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
R_3	1	0000	0000	0000	0000	0000	[000$\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	1]							
	0	1111	1111	1111	1111	1111	[111$\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	1]							
$D_3 \times R_3 = D_4$	0	1111	1111	1111	1111	1111	1111	1111	1111	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
	1	0000	0000	0000	0000	0000	0000	0000	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
R_4	1	0000	0000	0000	0000	0000	0000	0000	0[000	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	11]
	0	1111	1111	1111	1111	1111	1111	1111	1[111	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	11]
D_5 (not formed)	0	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111
	1	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Short precision divide result is $N_4 = NRR_1R_2R_3$
Long precision divide result is $N_5 = N_4R_4$

portion in brackets is gated to the recoder the output will have value $\mp 2^{12}x_k$.* The bits in the bracket are chosen such that the left-most three bits are identical. Thus, multiple six (refer to Fig. 9) is not used because a zero multiple is always recoded, and the product $\mp D_kx_k$ or $\mp N_kx_k$ is accomplished by the five operands gated to multiple gates one through five. If the unshifted multiplicand is gated simultaneously into the sixth multiple gate the sum of all six operands is $D_k + (\mp D_kx_k)$ or $D_k(1 \mp x_k)$, which is the desired result. The result which is generated by adding the carry and sum out of the carry-save adder tree (refer to Fig. 11) is the following:

$$D_{k+1} = \begin{cases} 0.\ 1111\ 1111\ 1111\ 1111\ 1111\ 111x\ xxxx \rightarrow \\ 1.\ 0000\ 0000\ 0000\ 0000\ 0000\ 000x\ xxxx \rightarrow \end{cases}$$

Thus, without using the carry-save adder loop the leading string has been increased by nine bits.

Table 3 presents the format of the multipliers and their denominators. Notice that the first multiplier is ten bits and the second is seven. These are fixed and cannot be changed without making the table-lookup decoder larger. Thus the third multiplier is the first one capable of using more than nine bits. But if a multiplier of more than nine bits is used, the carry-save adder loop must be included in the divide loop. Since this is undesirable (concurrency among multiplies is discussed below) the multiplier for the third and fourth iterations is chosen to be nine bits, thereby increasing the string length by nine each time. Thus, D_4 has 32 leading 1's or 0's. Now if D_4 is multiplied by multiplier four, R_4, the result will be 64 leading 1's or 0's, which is equivalent to unity within the desired accuracy. Therefore, since it is not necessary to calculate multiplier five, R_5, this multiply is not done and since only the numerator is going to be multiplied by multiplier four, the carry-save adder loop is used to speed up this last operation. (This is discussed more fully below.)

The second difficulty, which was that the carry propagate adder must be included in the path, was used to solve the third difficulty. Consider Fig. 17, which is the divide loop. To begin the execution of a divide the divisor is multiplied by the first multiplier (R), and the first denominator (D_1) is generated at the output of the CSA tree. These two outputs are added in the carry propagate adder; the output loops back to the input and becomes the new multiplicand; the truncated and complemented output forms the new multiplier. Note that the complete loop contains two temporary storage platforms—one at CSA-C and one at

* The multiplicand is shifted right twelve positions to compensate for the 2^{12} factor.

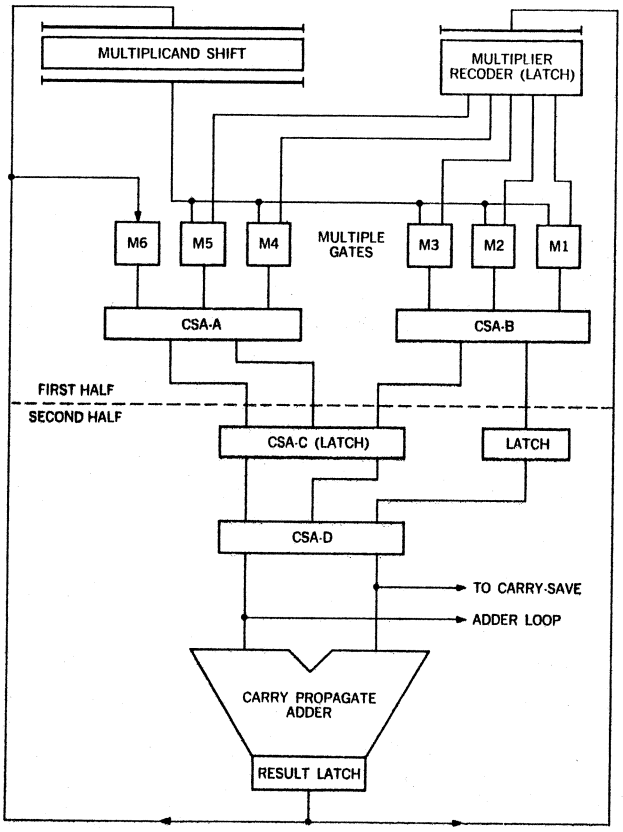


Figure 17 Divide loop.

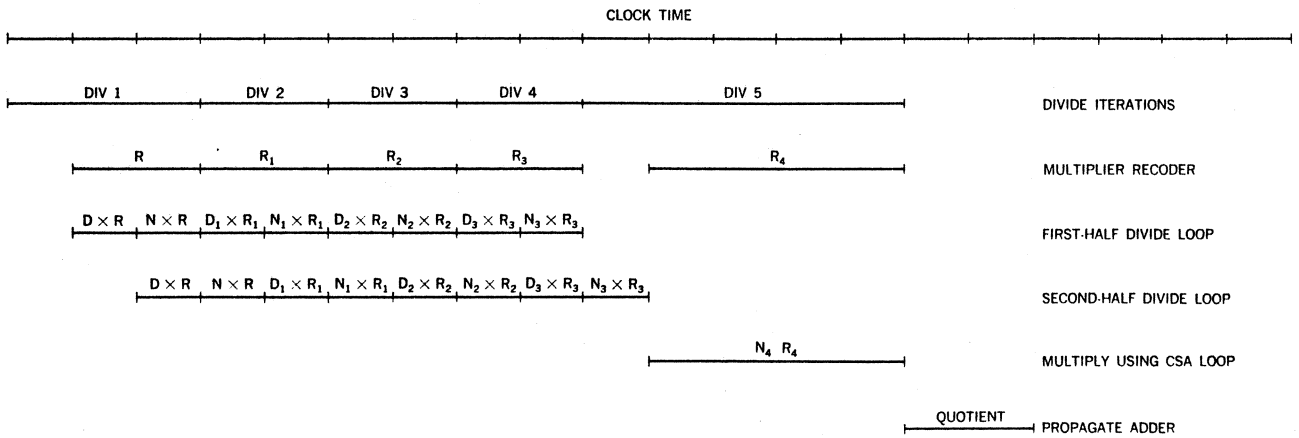
the output of the propagate adder, the result latch. Thus as soon as $R \times D$ is gated into CSA-C, the next multiply, $R \times N$, can be started. Now $R \times D$ advances to the result latch and loops back to start the next multiply $R_1 \times D_1$. At this time $R \times N$, which is latched in CSA-C, advances through the adder to the result latch. So the two multiplies follow each other around the divide loop. The first determines what the second should be multiplied by to converge eventually to the quotient.

This chain continues until multiplier four has been calculated. Since denominator five is equivalent to one, the multiply is not done. The 32-bit multiplier is gated into the reservation station and then gated to the multiplier twelve bits at a time as shown in Table 3. The result of this multiply, N_4R_4, is the final quotient. The diagram in in Fig. 18 shows the concurrency in the divide loop. The multiplier recoder latch is changed each time a denominator multiply is completed. Notice that two multiplies are always in execution, one in the first half of the divide loop (from input to CSA-C) and one in the second half of the divide loop (from CSA-C to the result latch).

Conclusions

The prime effort during the design of the floating-point execution unit was to develop an organization which would achieve a balance between instruction execution and preparation. Early in the design phase it appeared that an organization which would achieve this result would have a poor cost-performance ratio.

Figure 18 Timing diagram showing concurrency in divide loop.



Concurrency, obviously, had to be the key to high performance, but the connotation of concurrency in computers is parallel execution of different instructions. Thus the early organizations exhibited more than one execution unit and a high cost. In the final organization, concurrency is the key to the high performance, but this organization exhibits several levels of concurrency:

1. Concurrent execution among instruction classes.
2. Concurrent execution among instructions in the same class (add unit).
3. Concurrent execution within an instruction (multiply iterative hardware and divide loop).

The concepts of instruction-oriented units and reservation stations were used to keep the performance level sufficiently high but reduce the cost. These two concepts yield the same performance as several units without the cost of several units. The instruction-oriented units allow the design to be hand-tailored for faster execution and permit the use of a unique algorithm to execute divide.

Acknowledgments

The design of a computer unit such as this—containing nearly as many logical decisions as IBM's previous largest central processor—requires a great deal of decision making. The authors gratefully acknowledge the logical and engineering design contributions made by the following individuals: Mr. W. D. Silkman for the floating-point instruction unit; Messrs. J. J. DeMacedo, J. G. Gasparini, L. Grosman, R. C. Letteney and R. M. Wade for the multiply/divide unit; Messrs. M. Litwak, K. J. Pockett and K. G. Tan for the add unit; and Mr. E. C. Layden for the processor clock.

Acknowledgment is also made for the early planning efforts of Mr. R. J. Litwiller.

References

1. W. Buchholtz et al., *Planning a Computer System*, McGraw-Hill Publishing Co., New York, 1962.
2. G. M. Amdahl, G. A. Blaauw and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal* 8, 87 (1964).
3. D. W. Anderson, et al., "Model 91 Machine Philosophy and Instruction Handling," *IBM Journal* 11, 8 (1967) (this issue).
4. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal* 11, 25 (1967) (this issue).
5. R. F. Sechler, A. K. Strube and J. R. Turnbull, "ASLT Circuit Design," *IBM Journal* 11, 74 (1967) (this issue).
6. O. L. MacSorley, "High Speed Arithmetic in Binary Computers," *Proc. IRE* 49, 67, (1961).
7. R. E. Goldschmidt, "Applications of Division by Convergence," Master's Thesis, MIT, June 1964.
8. C. S. Wallace, "A Suggestion for a Fast Multiplier," *Trans. IEEE*, EC-13, 14–17 (1964).
9. M. V. Wilkes et al., *Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley Publishing Co., Cambridge, Mass., 1951.
10. T. C. Chen, "Fast Division Scheme," private communication, November 4, 1963.

Received November 1, 1965.

L. J. Boland
G. D. Granito
A. U. Marcotte
B. U. Messina
J. W. Smith

The IBM System/360 Model 91: Storage System

Abstract: This paper discusses the design concepts employed in the development of the IBM System/360 Model 91 storage system. Particular attention is paid to the exploitation of System/360 capabilities in the areas of large storage capacity, concurrent operation, and flexibility, as they apply to the highly overlapped Model 91 system.

An interleaved set of main storage modules is used with the Model 91 to help mask the difference between machine cycle time and storage access time. The set is connected to the central processor, peripheral storage control element and maintenance console by three time shared busses—one for addresses, one for data-in, and one for data-out. The main storage control element (MSCE) controls these busses to maximize the storage access rate. To achieve minimum access time, requests are normally sent directly to the storage modules. The proper module is selected by the MSCE, the address gated in, and the storage cycle started. If the module is busy from a previous request, the request is stored in a request stack for a later attempt. If the request is accepted, it is stored in an accept stack. This stack controls the data-out gating of the storage modules, and notifies the CPU of the destination of returning data. It also furnishes module busy information which controls the recycling of rejected requests.

An important feature is the ability of the MSCE to logically sequence store/fetch requests, by interlocking the rejected requests with the current request without any degradation of minimum access time. Additionally, each address sent to the MSCE is compared with the addresses of waiting and in-process requests. This allows serial fetching of two adjacent single words of a double-word storage cycle. Fetches following stores to the same location can be executed without waiting for a fetch storage cycle.

Peripheral storage is provided in the system for both block transfers of data and individual word fetches and stores. All requests to peripheral storage are sent via the peripheral storage control element.

The MSCE is synchronized with the CPU and uses the same machine cycle. Ideally, a request can be honored each machine cycle, but the actual rate is determined by storage module conflicts. The storage system performance is measured in access rate and access time. The MSCE has been simulated to measure the effects of storage speeds, degree of interleaving, and changes in MSCE controls.

Introduction

In the development of a highly concurrent processing system, there are two principal considerations:

- The maintenance of a high rate of information flow through the processor requires a storage control system capable of transferring large volumes of data with a minimum of interference.
- Throughput requirements dictate that the input/output handling and buffering capability be equally efficient.

The versatility of the Model 91 can be partly ascribed to a highly overlapped and flexible storage control system which satisfies these criteria. This system is based on a hierarchic concept of storage, with implications of a wide performance range which can vary according to application.

This paper is intended as a description of that system. It is divided into four major sections, to correspond to the points of view from which the design may be considered. The first section discusses the hierarchic concept and overall design objectives. Section two describes the main storage control element (MSCE) which serves as receptor of processor references and controller of all high-performance main storage references. It identifies the requirements imposed on the MSCE from the points of view of the processor and the peripheral storage control element and explains how these requirements were met. (A 4-way, interleaved, 750-nanosecond main storage is assumed for the description.) Section three deals with the peripheral storage control element (PSCE), which controls the flow of data among the peripheral elements of the system. The fourth section explains the characteristic interaction of the elements of the hierarchy, as exemplified by the main and peripheral storage control elements.

General design considerations

• *The hierarchic concept*

The hierarchic storage is characterized by its multilevel structure, consisting of a number of separate but interconnected components of varying sizes and speeds. Successful operation requires a versatile control scheme and depends upon the ability of the operating system and controls to move freely from one level to another in the hierarchy.

The Model 91 storage system has three principal media:

1) High performance main storage: a storage of intermediate size and capacity. Variations in this capacity and in speed and amount of interleaving provide a measured performance characteristic which lends itself to application "tailoring."
2) Extended main storage: This medium extends the capacity of core storage to accommodate the total addressing potential of System/360.
3) File storage and input/output

Each of these elements is monitored and controlled by its own storage control function (Fig. 1). Within the definition of a continuous addressing spectrum (pipeline*), it is possible to establish boundaries within which each control function can operate.

To fully exploit the hierarchic concept, the interconnection scheme must be able to move large quantities of data in several directions and at varying rates. As an example of this requirement, consider the *K-K'* configuration of the Model 91. The *K* component, a high performance main storage, can develop a 172 megabyte/sec rate. The *K'* component, extended main storage, can also develop a 172 megabyte/sec rate, even though it is relatively less accessible to the processor. (Although interference factors can decrease the actual rates, interference is minimized because there is a choice in the configuration of 32 storage units from which to select.) In addition the scheme must accommodate the concurrency of operation required by the processor (potentially 133 megabytes/sec), a storage channel, or data mover (capable of transfers from any point to any point at a rate up to 64 megabytes/sec), and files and I/O (5–10 megabytes/sec combined rate).

To meet these requirements the control system would ideally be able to create as much data transfer potential as there is storage potential. In the Model 91 *K-K'* there exists a potential data demand of 270 megabytes/sec and a potential storage availability of 344 megabytes/sec. (The demand is equivalent to a 72-bit word every 30 nanoseconds required to fully satisfy all users of storage.) Other configurations can extend the potential availability to more than 1,000 megabytes/sec.

* "Pipeline," in this case, means a continuous stream of operations or instructions, capable of being executed concurrently without neglecting any serial dependence among successive operations.

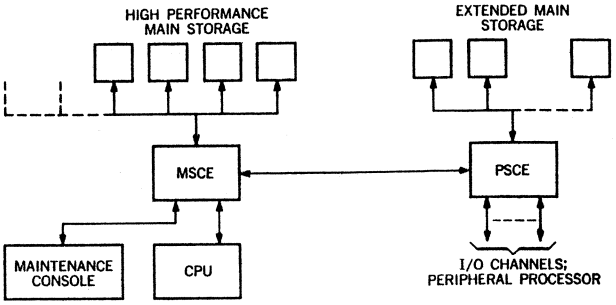
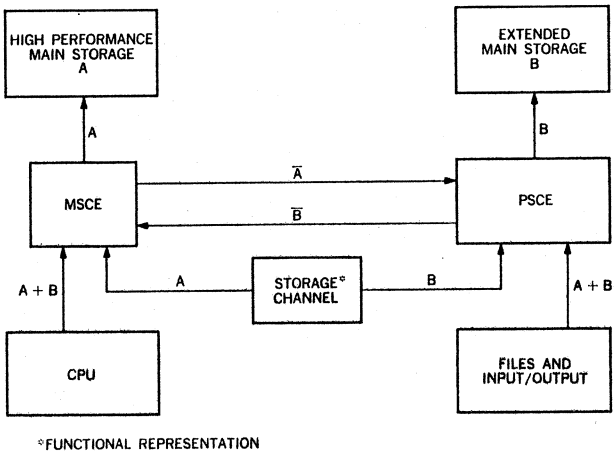


Figure 1 Block diagram of storage system.

Figure 2 Model 91 address flow.



In practice, however, there are constraints. For example, the machine cycle (60 nsec) generally determines the maximum pipeline transfer rate, and the word length cannot be substantially greater than 72 bits because of cable and skew problems. Consequently, one must look to concurrency of transfer to achieve high efficiency. The organization of the Model 91 storage control system is therefore characterized, within the hierarchic concept, by this multiple transfer potential. The separate, bounded control functions mentioned above provide each type of storage with its own transfer path (Fig. 2). Control of high performance main storage resides in the main storage control element (MSCE), while extended main storage and input/output are controlled by the peripheral storage control element (PSCE).

The paragraphs below summarize the specific design objectives which were developed for these units, and subsequent sections describe the operation of each.

• *Control system design objectives*

The key to success in a highly parallel pipeline processor is the ability to react quickly when the pipeline is diverted. Diversion occurs in the form of branching in general, and data-dependent branching in particular. Effective storage

reaction implies that a new flow be initiated in minimum time. Consequently, storage access time must be made as short as possible. Diversion of the pipeline in the concurrent system also implies that many accesses will be initiated without being used. It follows, therefore, that the storage control system must be able to provide many more transfers than can actually be used by a particular problem.

The design objective of the Model 91 was the achievement of a wide performance range which could vary with system application. This imposed on the storage system the following general requirements:

- Control of highly interleaved storages of different speeds.
- Overlap of a multiplicity of high speed I/O devices.
- Development of a very high performance storage channel.
- Minimum disturbance to the processor to achieve a wide performance range.
- Ability to accommodate advanced storage and I/O devices.
- Flexibility to react quickly to various application needs.

To develop these objectives, simulation methods were used to test proposed designs for the MSCE. The effects of interleaving, storage speed, and buffering were observed to determine their impact on processor performance. The impact of various memory cycles on overlapped I/O channels, under each of several design conditions, was also observed by simulation, to determine the form of the PSCE.

With the refinement provided by simulation, the general requirements were defined in terms of the following objectives for the final storage system design:

1) An overall design relatively insensitive to storage speed.
2) Minimization of access time to the processor while maintaining high data rates.
3) Control of large numbers of small storage arrays.
4) I/O control for optimization of overlap operations.
5) Elimination of multiple busses to the individual storage units.
6) Buffering of mismatch between fast I/O and slow storage.
7) Storage protection for highly interleaved variable sets.

By combining proven techniques with novel concepts the design of the MSCE and the PSCE has met these objectives very well. The sections which follow describe that design (and its operation) in detail.

Main storage control element

- *High performance storage principles*

From a CPU viewpoint, the ideal storage system would be one large storage unit with a cycle time equal to the basic machine cycle. The CPU can then issue storage requests on any, or every, cycle. Since this is impossible with the fast cycle of the Model 91, the technique of interleaving is used. Consider a main storage system composed of several (4 to 16) self-contained storage units, which are capable of simultaneous operation. Contiguous addresses are interleaved among the units in a sequential manner. For example, the sequential address string N, $N+1$, $N+2$, $N+3$ would be stored in four different units. The storage system can service a string of sequential requests by starting, or selecting, a storage unit every cycle until all are busy.

Interleaving also improves the servicing of a string of random addresses, since the large number of units reduces the probability that an address will go to a busy unit. Thus the access rate of the storage system is a function of the number of interleaved units, i.e., of the interleaving factor. In practice, the interleaving factor is a binary number, which permits storage address allocation to be determined be decoding the low order bits of the address (for example, the three low order bits for interleaving by 8, or 2^3).

Since the CPU issues storage requests at a one per cycle (or slower) rate, the use of a common set of busses on a time-shared basis is suggested. That is, on every cycle a new address can be transmitted to all storage units over an address bus. The same is true for data words on the busses to and from storage. Since bus cycles can be wasted because of storage conflicts, the control of the busses affects the maximum data rate.

Another performance criterion for the storage system is the access time, which is the time which elapses between the issuing of an address by the CPU and the return of data to the proper sink register. The minimum access time is the sum of the storage unit read time and the cable and logic delays in the MSCE. The average, or probabilistic, access time (which includes the effect of storage conflicts) is limited by the interleaving factor and the storage cycle time. It also depends upon the organization of the MSCE, which must make some provision for conflicts.

- *Design requirements*

Since the ultimate performance of the storage system is limited by the storage units themselves, obvious requirements are that the MSCE must minimize its share of the access time and optimize the data rates by properly controlling the time-shared busses. Certain logical requirements are imposed upon the MSCE by the design of other elements of the Model 91, particularly the instruction unit and the PSCE.

The input to the MSCE from the instruction unit is a storage request, which consists of an address, a return or sink address to route the returning data, control bits to define the operation more precisely, and data for store operations. The MSCE must act on the request by selecting the proper storage unit and furnishing it with an address

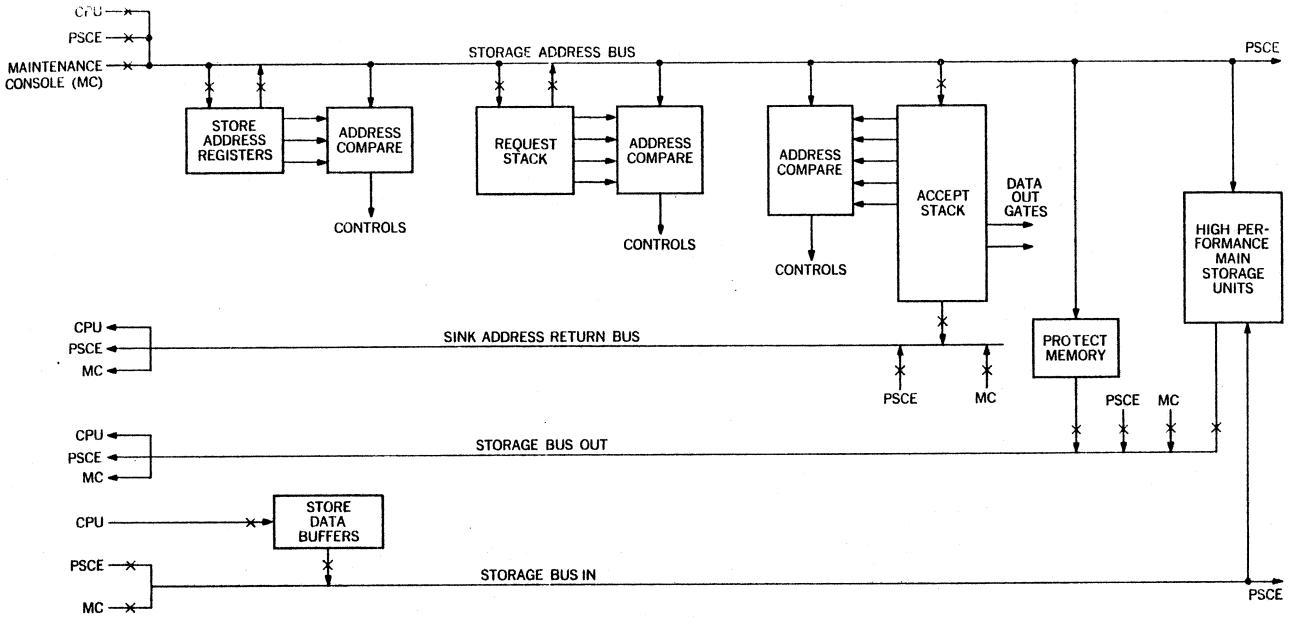


Figure 3 Block diagram of MSCE organization.

and data for stores. In case of a conflict, the MSCE must hold the request for later recycling, generally without stopping the instruction unit. Several requests can be in the MSCE and in storage at the same time, and these need not necessarily be handled in sequence. The various returning data words are correlated with their sinks by the MSCE, which sends the sink address to the CPU a cycle before the data.

In general, requests do not require servicing in sequence, but can instead be serviced in an order which will optimize bus utilization. There is a requirement, however, that the MSCE be able to correctly sequence several stores, or stores and fetches, to the same address.

The storage units considered in the design of the Model 91 have word sizes of 72 bits, including parity, and most units of the CPU are designed to use this word size. Fixed-point and single-precision operations, however, require 36-bit words. Since addresses sent to the MSCE and to the storage units define 72-bit words, two storage cycles could be used in accessing the two halves of a storage word. To avoid this performance degradation, the requirement known as Multi-Access was placed upon the MSCE. This feature allows a memory data register to be read out as many times as desired without recycling the storage unit.

The PSCE carries different requirements because it is connected to the storage and input/output channels. The storage channel objectives include the ability for the PSCE to make requests to the MSCE in bursts, at a one per cycle rate. In addition, requests by the input/output channels via the PSCE could not tolerate uncontrolled delays in the MSCE caused by storage or bus conflicts, because overruns would result. Thus it was decided to give the PSCE the ability to monitor the busy status of main storage, and to reserve main storage units. Given this ability, the PSCE can control its requests to the MSCE so that they are guaranteed acceptance. This will be discussed more fully in later sections of this paper.

The third requesting unit is the maintenance console, which stores and fetches from manual keys, and also initiates the logging in storage of machine status for diagnostic purposes. Since a high data rate is not important, it was judged sufficient to allow the console one request in process at any time.

- *MSCE Organization*

A main storage control element, designed to meet the above requirements, is diagrammed in Fig. 3. It consists of the following functional areas:

- Store address registers (SAR's), which hold addresses of stores pending availability of store data.
- Store data buffers, which hold store data words from all areas of the processor pending availability of the proper storage unit.
- The request stack, a set of four registers which holds rejected requests from the processor pending availability of the storage unit, and thus buffers the processor from storage conflicts.
- The accept stack, a set of registers which holds information on accepted requests in process.
- The storage address bus (SAB), which transmits addresses to all storage units and to the PSCE.

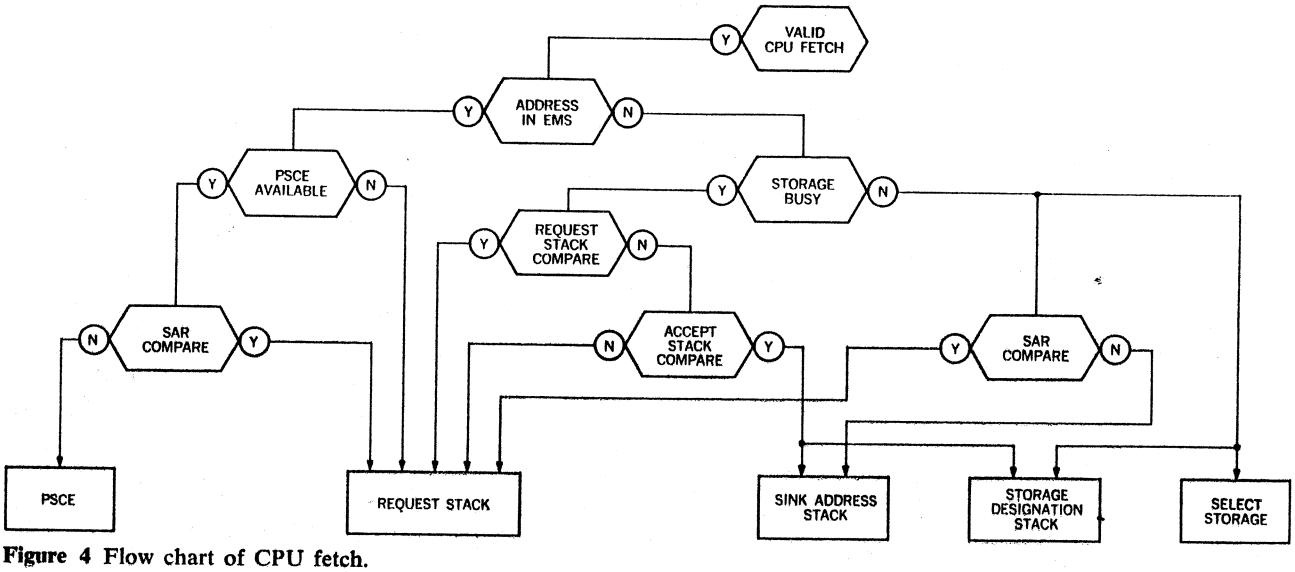


Figure 4 Flow chart of CPU fetch.

- The sink return bus, which transmits the sink address fetch to all sink registers one cycle before the fetch data.
- The storage-bus-out (SBO), which transmits 64 bits and parity of data from storage units, the PSCE, protect storage and the maintenance console keys to all data sinks.
- The storage-bus-in (SBI), which transmits 64 bits and parity of data from processor-filled data buffers, the PSCE, and the maintenance console to storage.
- Protect storage, which stores the keys required for the System/360 Protection Feature.
- Controls

If there are no conflicts, the MSCE can accept a request each cycle from one of its sources—the processor, PSCE, or maintenance console. During each cycle the MSCE controls determine which source will be allowed a request, and gates are conditioned to put the address on the address bus. This bus is also used to load the SAR's, which hold store addresses until the data word is generated by the processor. The main feature of the address bus is its direct path to storage, with no intervening buffers, to minimize access time.

The general organization can best be understood by considering a simple fetch request. During the cycle in which a request is gated on the address bus, the address bus controls determine its disposition. A successful request is sent to the proper storage unit, or to the PSCE if extended main storage is requested. A main storage request is also gated into the top position of the accept stack, the push-through stack which holds pertinent information about all requests in process.

Any rejected processor request is stored in a position of the request stack for later recycling. Requests are not taken from the PSCE or maintenance console unless they can be guaranteed acceptance, and thus never reside in the request stack.

Each address on the address bus is compared with addresses in the SAR's, the request stack and the accept stack. Comparison with an SAR forces rejection of the request and it is stored in the request stack, since its acceptance would cause an out-of-sequence fetch. Comparison with an address in the accept stack implies that the desired word is being fetched by a previous request, and can be obtained again without selecting a storage unit or waiting for it to "go not busy." This is the Multi-Access feature discussed in a previous section. As implemented, it applies to a fetch following either a fetch or a store.

Comparison with an address in the request stack causes the request to be tagged for a future Multi-Access operation, and to be gated into another position of the request stack. The presence in the stack of an outstanding request for the particular address causes the second request to be rejected, keeping the two in the proper sequence. The flow chart in Fig. 4 summarizes the handling of the fetch request by the address bus logic. If the request is accepted, the MSCE generates a select pulse to start the proper storage unit. The selected unit latches the address, which is on the bus common to all units, and starts its cycle.

While the storage unit is cycling, the request is moving down the accept stack, one position each machine cycle. The stack contains the bit code designating the selected unit for n-2 positions, the word address for five positions,

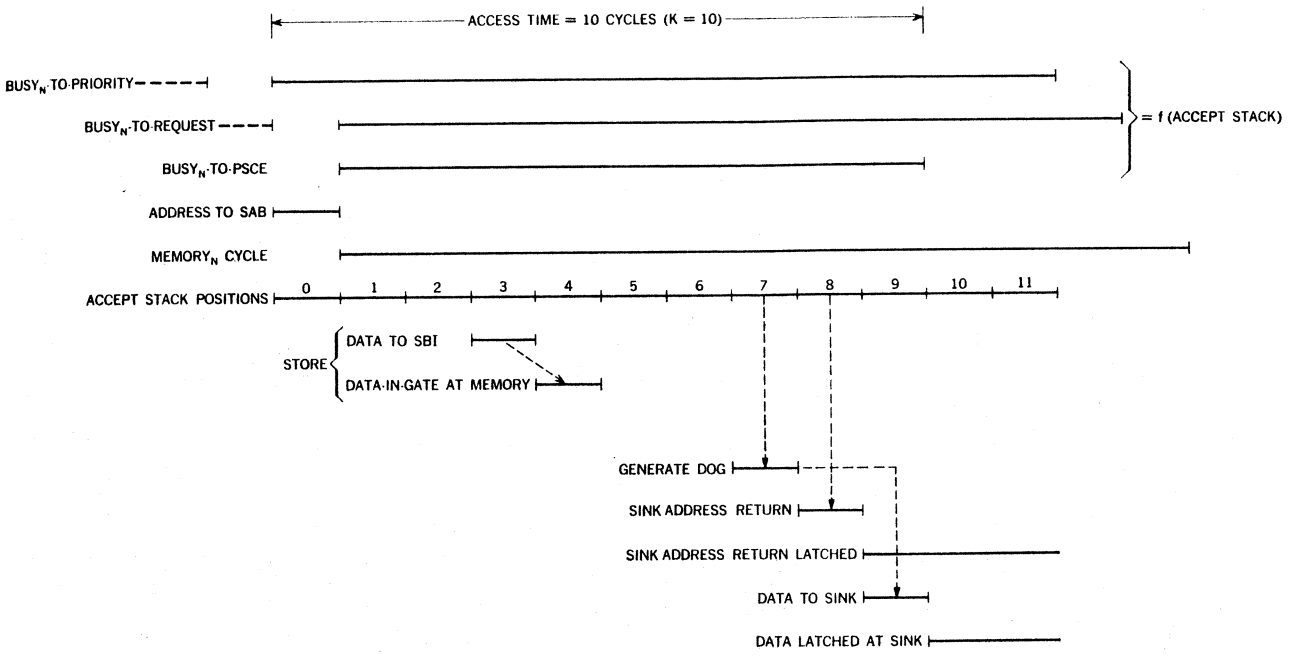


Figure 5 Timing diagram for CPU fetch.

and the sink address of the fetch for k-2 positions, where n is the number of machine cycles per storage cycle, and k is the smallest number of machine cycles required for access. Storage-busy information is obtained from the designation field of the stack, since a unit is busy only if its code is in any stack position. The same field is used to generate the data out gate (DOG), by decoding the k-3 position, and sending the decoded DOG lines to the proper data lines from storage. The DOGs are generated by the MSCE rather than by the individual storage units, to allow for Multi-Access operations. The full address is kept in five positions for comparisons for Multi-Access, as previously described.

The sink address is delayed in k-2 positions, the last position being used to determine the sink for which the fetch was made. The sink address is decoded, and the correct sink register is conditioned one cycle before the fetch data appear on the SBO. If there was no memory conflict, the data word is gated into the conditioned sink register k cycles after initiation of the request. Figure 5 is a timing chart for a simple fetch, where $k = 10$, the case for a 750-nanosecond memory unit. For completeness, store timing is also shown. As shown in the flow chart, the request could be rejected and gated into the request stack for one of the following reasons: (1) The requested storage unit was busy; (2) the request was to the PSCE, and the PSCE inhibit line (queue full) was on; or (3) the address compared with an address in a SAR or the request stack. Priority logic controls the re-cycling of rejected requests in a manner that optimizes the use of the address bus, protects against improper sequences, and guarantees acceptance of the request.

- *Accept stack*

The accept stack deserves a more detailed explanation since it generates many of the necessary control functions. The relation between memory cycle time, access time and depth of the stack has been given above. Figure 6 diagrams a stack with proper depths for a 750-nanosecond unit, with overall access to the processor of ten cycles.

The problem that led to the adoption of the accept stack was the requirement for three kinds of busy information from each main storage unit. This was complicated by the fact that multi-accesses to a unit could effectively make it busy for varying periods. The accept stack solved this problem, with several by-products, as shown by the following list of its functions:

1) Stores the coded designation of each busy storage unit, from which busy information is derived.
2) Delays the sink addresses, and correlates them with their respective data words.
3) Generates data out gates to gate fetched data words to the SBO at the proper time.
4) Stores the main storage address for five cycles, to compare with requests on the bus and identify Multi-Access cases.
5) Aids in maintenance, by effectively allowing single cycling of main storage fetches, and by correlating various errors with the requests causing them.

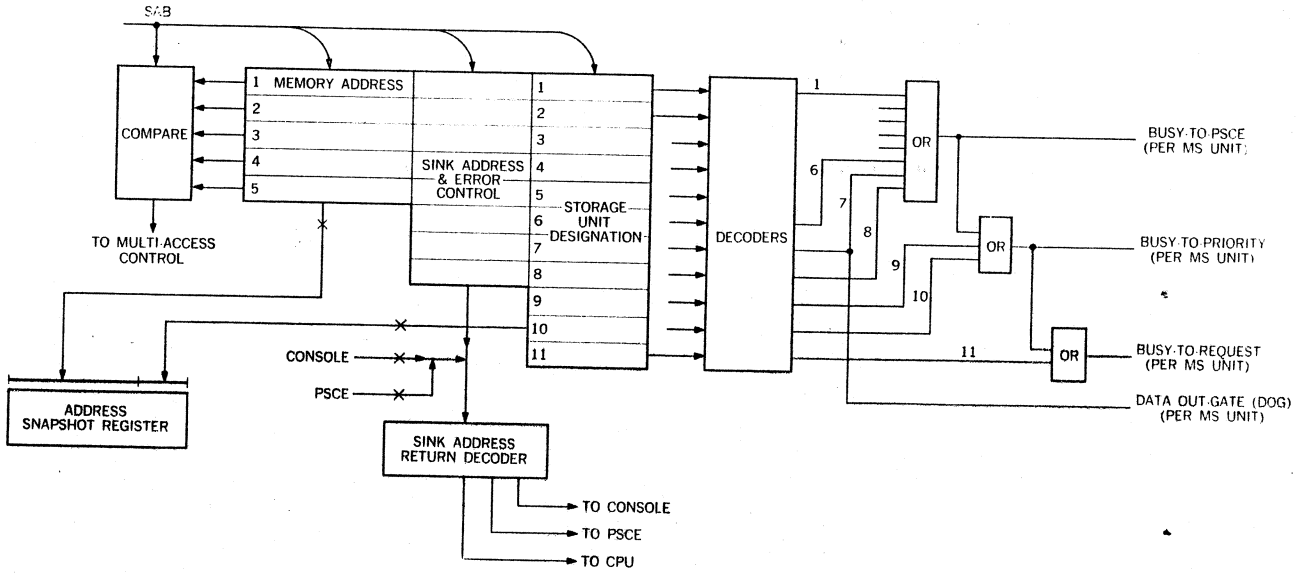


Figure 6 Block diagram of accept stack organization (750-nsec storage unit).

In Fig. 6, the stack itself is seen to consist of three major fields with varying depths. The deepest field, storage unit designation, is used to generate busy information and data out gates. Each position has a decoder, the output of which identifies a storage unit as busy to any request placed on the SAB. In cases of Multi-Access, the same code can appear in more than one position. The three types of busy information, busy-to-select, busy-to-priority and busy-to-PSCE are generated by examining eleven, ten and eight positions respectively as shown. They represent varying degrees of "look-ahead" on busy status. Their use will be explained more fully in the section on controls.

Position 7 is decoded to generate data-out gates. This means that the seven-cycle delay through the seven positions plus the communication time to the storage unit equals the internal access time of the unit.

The "sink and control" field is used mainly to delay the sink address sufficiently to correlate it with returning data. A total of eight cycles in the push-through stack, plus communication time to the sink registers, equals the proper delay to select the sinks one cycle before the return of data. This field is used also to carry error information, which is inserted in the proper position depending upon the source of the error. For example, if an address parity error is detected at storage an error bit is inserted in position three, since the request causing the error has been shifted down to that position.

The address field is used to compare for Multi-Access conditions. Because of circuit loading limitations on the address bus, the depth is limited to five positions. However, machine simulation runs were made on a variety of problems with different depths, and fortunately no improvement was noted with increased depth. The address field is used also, in conjunction with the aforementioned error bits, as a maintenance aid. If an error bit is detected in position 5, indicating that an error was associated with the particular request, the address field is gated into a special "snapshot" register and saved for later use in diagnostics.

- *Controls*

The controls in the MSCE consist explicitly of two functions. First, a decision must be made as to which addressing source should be gated to the address bus. Second, given some address on the address bus, a decision must be made as to which storage unit should be selected, if any. Within each control function there are, of course, many other subtle decisions required to effect logical sequencing. Implicit control of SBO, SBI, and sink address returns is a function of the push-down codes in the accept stack.

For the first decision, priority, the general order of service is:

1) PSCE to (main) storage.
2) Maintenance console to storage.
3) Request stack to (main) storage for Multi-Access.
4) SAR to storage.
5) Request stack to storage.
6) Processor to storage.

A new priority decision is made every cycle, resulting in a time-multiplexed pipeline of priority—address bus—fetch/store.

When no requesting source of priority higher than the processor requires the SAB, processor fetch requests are gated to the SAB (usually on the cycle following the address generation) without a prior test of storage availability. With a sufficient interleave factor and a short storage cycle the requests seldom encounter busy units. Thus the requirement of minimum access time can be attained. Although store requests are held in SAR's where a storage-busy test could be performed, the SAR's are gated to the SAB without the test. Thus the SAR's are unloaded as soon as possible to make them available again to the instruction unit, which considers a store operation completed once it loads a SAR.

Simulation has demonstrated that recycling of requests from the request stack after a fixed-time wait results in secondary rejections which reduce bus efficiency and complicate the control of real-time PSCE requests. Hence, initially rejected processor addresses are recycled only once to the optimized address bus, according to storage-available and first-in, first-out discipline.

Because PSCE requests for main storage require a very high data rate, the address bus efficiency for the PSCE must also be high. Hence, PSCE requests to main storage are granted priority within the PSCE itself as a function of impending availability of specific storage units.

To optimize the overlap of storage units, in priority and on the address bus, their imminent availability (i.e., non-busy status) is as valuable as their actual availability. Hence, three levels of busy status are decoded for each storage unit:

1) Busy-to-PSCE signal, which turns off four cycles before actual time-out of the storage unit.
2) Busy-to-priority signal, which turns off two cycles before actual time-out of the storage unit.
3) Busy-to-select signal, which turns off one cycle before actual time-out of the storage unit.

These signals allow the MSCE to "look-ahead" in order:

- To respond to a PSCE reservation for a storage unit, acknowledging the availability of the unit. Four cycle look-ahead covers the communication delay between the MSCE and PSCE and allows the PSCE to execute a priority cycle (busy-to-PSCE).

- To execute priority for the maintenance console or request stack, each of which needs to know when a specific storage unit is to be available during an address bus cycle (busy-to-priority).

- To allow the generation of a select signal for a busy storage unit during address bus time, if the unit is to be available on the following cycle (busy-to-select).

The second control function is concerned with the disposition of the contents of the address bus on the cycle following the associated priority cycle. When the address bus is valid, the decoded main storage unit or the PSCE is selected if available. If the unit to be selected is not available, the request is routed into the request stack where it resides until the desired storage unit becomes available. When a main storage unit is selected, certain fields are routed into the push-through accept stack for use later in controlling data out gate (DOG) generation, sink address returns, SBI, SBO, storage busy decode, and Multi-Access compares. Special interlocks in the form of address comparators (address bus vs. pending SAR's and pending requests in the request stack) order stores to the same address and recognize and re-order out-of-sequence store/fetch requests to the same address. These interlocks also link these same store/fetch or fetch/fetch requests to the same address for Multi-Access.

If extended storage is decoded, the address bus is gated into a buffer in the PSCE, from which point the address is decoded further to select the appropriate storage unit according to the discipline of the PSCE. If this buffer is not available the request is gated into the request stack in the MSCE until the buffer becomes available. Note that because of the PSCE-MSCE single buffer interface, the PSCE can expand capacity, increase speed, etc., without affecting MSCE control design.

- *Storage protection*

The storage protection feature in the Model 91 performs the same function as in other members of the System/360 family, which is the protection from unauthorized fetches and stores. All attached storage is considered to be in blocks of 2048 bytes, and a 4-bit key is kept in a protect storage for each block. Each request initiates the read-out of the proper address key, which is compared with a key furnished by the requesting source. A mismatch effectively cancels the operation.

Since a protect operation can be required on every MSCE cycle, this suggests either an interleaved set of protect storage units, or one unit with a 60-nsec cycle. If an interleaved set is used, each unit must be of sufficient size to store all keys required by the storage system. Furthermore, the access time of the unit must be fast enough to cancel stores when mismatches are detected, a requirement which becomes more difficult as the attachment of faster units is considered. These factors, as well as the requirement for adaptability to various storage units, led to the adoption of a single high-speed protect storage. The 60-nsec cycle requirement is met by implementing the protect storage in extra-high-performance logic.

- *Variations of storage*

Interfaces between the MSCE and other units have been designed to allow variations in interleaving factor, capacity, and storage speed. Simulation of a random addressing source has demonstrated the relative improvement in

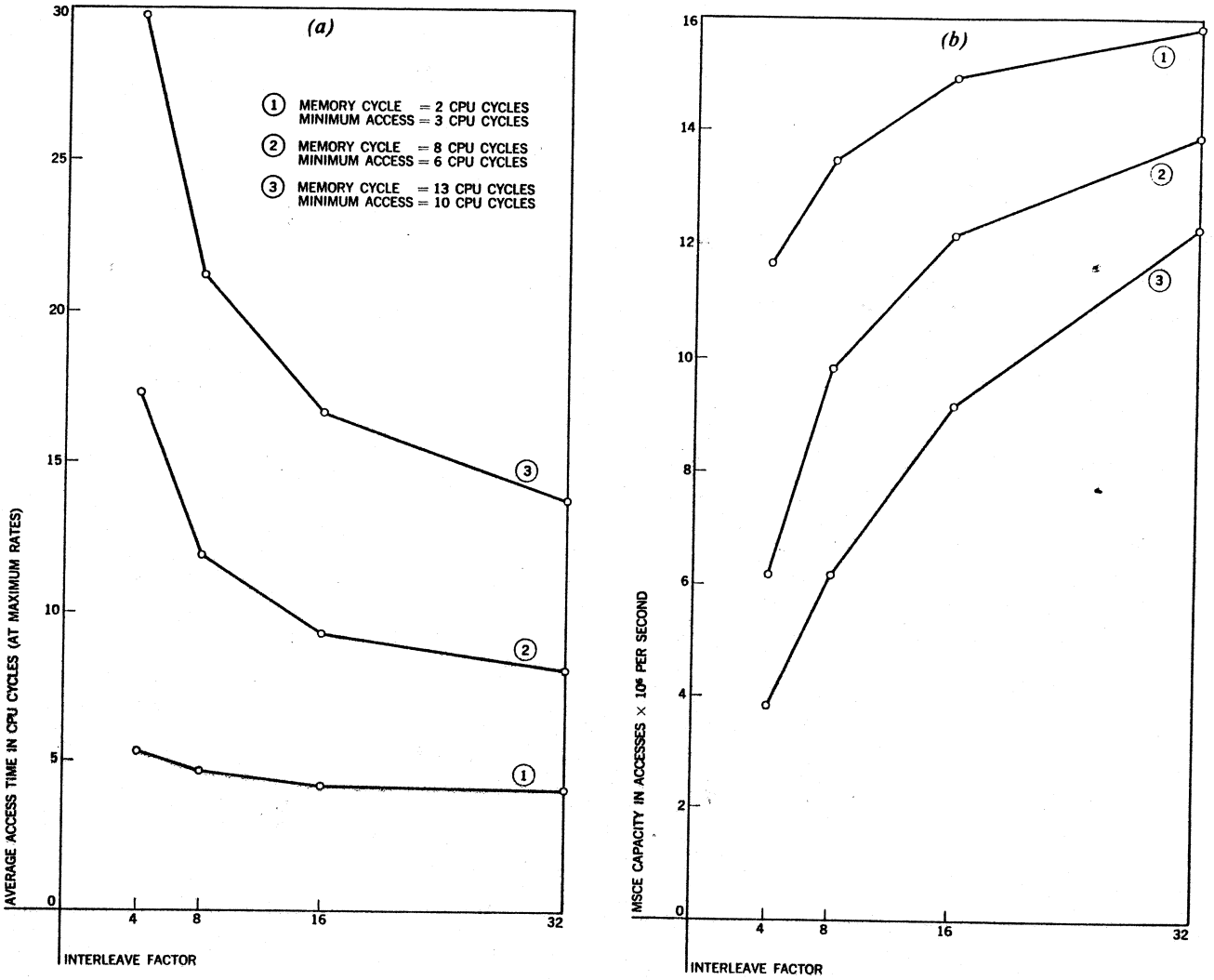


Figure 7 Simulation results. (a) Average access time vs. interleave factor; (b) MSCE capacity vs. interleave factor.

average data rate that is encountered as interleaving and speed are improved (see Fig. 7). The same simulation indicates the average access time on a given storage unit as a function of interleave factor. MSCE design accommodates the flexibility in storage configuration by modular circuit replacement, yielding interleave factors of 4, 8, or 16. Storage cycle time can range up to 750 nsec by varying the number of vertical push-down positions in the accept stack.

Note that the data rate and the average access time improve almost linearly with improvement in storage speed, whereas the interleave factor yields an exponentially diminishing improvement (see Fig. 7). The theoretical improvement due to interleaving can be achieved only by diligent attention to optimal physical distribution of logic and the connection of the interleaved units. Total logic and cable delay from a requesting source to storage to the data sink has been normalized to two cycles. Interleaving of the operation of storage units greater than 16 ways would exceed the two-normalized logic and cable delay cycles. The result is an increase in access time because the physical expansion of logic and cabling is not linear. The effect of interleaving, then, needs to be considered more than casually.

Peripheral storage control element (PSCE)

- *Design requirements*

The governing requirements for the PSCE were twofold. The first requirement was compatibility with other high performance models of System/360. The second requirement was the maximization of storage utilization. The

first requirement helped define the minimum number, types, and speed of I/O units that must communicate through the PSCE. This did not, however, define the maximum capability. The second requirement pointed out a need for something more elaborate than a switch to handle the congestion that could develop from an attempt to transfer data to and from extended main storage or high performance main storage.

Bench marks

The requirements were such that a set of bench marks was required to prove the superiority of any particular concept. Three bench marks were defined and used during the initial design phase for cost/performance evaluations.

Bench mark definition was difficult since little was known about all the likely applications or I/O configurations that would develop for the Model 91. Much was known of special applications and problems but little was known about the use of extended main storage. Before the bench marks were defined, an attempt was made to answer the following questions:

- How much I/O would a typical Model 91 configuration contain?
- What are the maximum data rates of the I/O units?
- With the availability of high performance I/O, what is the requirement for overlapping I/O?
- What is the chaining requirement for high performance I/O?
- What aggregate I/O rates would the PSCE be expected to handle?
- What is the storage range that the PSCE would be expected to handle?
- What is a typical size for storage configuration?
- Should storage be interleaved? If yes, what should the interleave factor be?
- What are typical random rates for processor activity?
- Should the Model 91 be able to share storage with peripheral processors?
- What is the minimum acceptable storage channel rate?
- What is the maximum transfer rate expected for the peripheral processor?

The questions were not simply resolved. It was difficult to put limits on any condition because the best performance was desired in all areas. It was possible, however, to develop a small number of reasonable alternatives, and the following bench marks were defined for comparing alternate PSCE designs:

I. Storage: 4 or 8 way interleave, 8 μsec cycle
   I/O*: 2 — 1.25 megabyte/sec devices
   1 — .150 megabyte/sec device
   Peripheral processor (PPE): 6.66 megabyte/sec
   Storage channel (SC)*: maximum rate

II. Storage: 4 or 8 way interleave, 8 μsec cycle
   I/O*: 2 — 1.25 megabyte/sec devices
   1 — .150 megabyte/sec device
   1 — 90 kilobytes/sec device
   PPE*: —6.66 megabyte/sec
   Central processor (CPU)†: maximum rate

III. Extended Main Storage: 4 or 8 way interleave
   I/O*: 1 — 1.25 megabyte/sec device
   1 — .150 megabyte/sec device
   1 — 90 kilobyte/sec device
   SC*: maximum rate

These bench marks were used to help select the design approach with the most potential. They were not used as ultimate objectives. Once a design concept was selected, simulation was used to help evaluate cost/performance.

Speed matching and other problems

The speed matching problems were as varied as the combination of interfaces and speed variations possible at each interface of the PSCE. The variations in the interfaces are due in part to the different storage technologies, storage hierarchies, bussing needs and circuit requirements that may exist for various system configurations. The storage hierarchies present many unique engineering problems in the area of the boundary detection which is used for interleaving, bus assignment, and storage protection.

One of the most difficult speed matching problems that the PSCE had to contend with was that of allowing high speed I/O to operate into a storage unit with a cycle time greater than the cycle time of the requesting I/O unit. This same mismatch exists for the processor and the storage channel (SC) but, since these units can wait indefinitely for service, they are not subject to overrun as are the I/O units. The traditional approach to the I/O problem has been to allow I/O units to have priority over any other element in a system. If the priority approach did not solve the problem, then it was usually necessary to bypass any bus in the path to storage and create an independent path for I/O. This approach had led to the development of "multi-tailed" storages.

Standard solutions were found to be inadequate because their implementation would only partially solve the I/O problem and still do nothing to improve the processor or SC rates. In the past, if the I/O rate into storage was such that it could not tolerate a conflict, the I/O would block the processor until the risk of overrun was past.

It was decided that any new solution would be acceptable only if it met the following requirements:

- CPU, peripheral processor and SC to have access to storage without the use of multi-input storage units.
- More than one high speed I/O channel to be able to use storage in an overlapped mode.

* Sequential addressing.
† Random addressing.

Table 1 PSCE bench mark evaluation.

Case	Bench mark	No. of Storage Units @ 8 μ sec	No. of buffers	SC[a] store or fetch	SC rate	CPU fetch rate	PPE[b] fetch rate	Storage utilization
1	I	8	12	All fetch	2.35	...	0.53	69%
2	I	8	12	All store	4.0	...	0.6	90%
3	I	8	12	50% fetch, 50% store	2.96	...	0.56	80%
4	I	8	16	All fetch	3.64	...	0.6	82%
5	I	4	12	All fetch	0.96	...	0.3	100%
6	II	8	12	...	...	2.35	0.6	70%
7	II	4	12	...	...	.5	0.42	88%
8	III	8	12	All fetch	5.7	...	...	86%
9	III	8	12	All store	6.7	...	...	100%
10	III	8	12	50% fetch, 50% store	6.2	...	...	93%
11	III	4	12	All fetch	2.7	...	...	100%

NOTE: All rates in megabytes/sec.
[a] Storage channel
[b] Peripheral processor (PPE)

- A processor request for a busy storage unit should not inhibit its ability to make other requests.
- Storage interleaving must be possible in order to improve the accessibility of requested data.
- No unit must resort to blocking storages in order to guarantee their availability at a later time.
- Due to the built-in overhead of a storage control unit, it must be able to handle requests in a pipeline fashion, a pipeline technique being one which allows concurrent execution of multiple operations while taking into consideration the serial dependence of the operations.
- The design must lend itself to growth and be able to adjust to different storage hierarchies.
- The design must be balanced to maximize the use of storage to all users.
- The design must be able to adjust to different storage configurations for proper boundary detection.

- *New concepts*

It was decided that the most promising concept for meeting the basic requirements was a bus organized around a buffer stack, or queue. The use of buffers was certainly not new but the manner in which they were to be used provided the flexibility and performance that was desired. The queue developed has the following operating characteristics:

- A variable number of queue positions are dynamically reserved for I/O inputs. The number reserved depends on the speed and number of I/O units in operation.
- The queue is used to store outstanding requests made by all users.
- Input requests to the queue are on a first come, first served basis except for simultaneous requests, which are handled in a fixed priority order.
- Output from the queue to storage is based on a three-level decision. The first decision level checks for available storage. The next decision level determines the unit that will have priority out of those requesting an available storage. The last decision level selects the first request for the unit getting priority.
- Output from the queue destined for channels is handled on a request basis. All other output (peripheral processor, CPU, and storage channel) is handled when no higher requests are outstanding.
- The queue can overlap all input and output operations. That is, at any point in time it can handle data returns from storage, two input priority requests, an output priority storage selection, and the return of a word to a channel and to the CPU.
- Outstanding requests in the queue may be handled out of sequence.
- Queue positions must be available for use by the processor and the storage channel when not reserved by channels.

- *Simulation*

The operating characteristics listed above were selected and developed for the PSCE only after a study of data obtained by simulating different bus designs. Simulation, based on the bench marks previously described, pointed to bottlenecks that would have caused an unbalanced bus under certain I/O configurations. The simulation results shown in Table 1 give an indication of probable storage utilization with a PSCE design of 12 queues working into an 8-μsec storage.

- *Queue design*

The decision to use a bus design with a shared buffer stack (queue) was made after studies indicated that all of the

proposed bus designs contained several buffers. However, the designs differed in the way the buffers were used and distributed among the individual control sections of the bus. It was found that, for approximately the same cost as designs with distributed buffers, it was possible to build a bus with a shared stack. It was necessary to build the registers of the central stack so that they could be used by all control sections of the bus. This required that they be more elaborate than would be the case if they had been designed to the specific requirements of one application. The increased complexity is more than balanced by the improved data rates that are possible for the processor and storage channel. The storage channel rates as a function of available queues and storage access time are shown in Table 2. (A more detailed explanation of these rates is given below.) It is obvious by looking at the table that the best storage channel rate is obtained by using all available queues. It was decided that 8 registers should be used in the queue because it was found that assuming a 3/4-μsec storage cycle and 4 channels with 1.25 megabyte devices, at least 6 registers were needed to handle the simultaneous operation of 6 or more channels.

Table 2 Storage channel best case transfer rates.

Number of queues available to storage channel	*Transfer rate, megabytes/sec* [b]*Access = 11 cycles*	[b]*Access = 10 cycles*
1[a]	7.74	8.35
2	15.7	16.7
3	23.5	25.0
4	31.4	33.3
5	39.2	41.7
6	47.1	50.0
7	54.7	58.3
8	62.5	66.6
0	SC Locked Out	

[a] For single word boundary, use this entry only.
[b] Access = Storage access time measured at PSCE tailgate for 3/4 μsec storage.

- *PSCE Organization*

The design of the PSCE merges several functions into one integrated unit. This organization consists of four major areas: queue and busses, queue priority, common channel controls, and storage channel (Fig. 8).

Queue and busses

The effectiveness of the queue depends in great measure upon its accessibility. The busses that communicate with the queue were planned with a goal of allowing simultaneous execution whenever feasible. These busses can be grouped in four main categories:

Unit entry busses There is a unit entry bus shared by all requesting units which provides access to all queues. In addition, two unit entry address busses are provided, one for the CPU and another shared by the other requesting units. The independent processor address path was provided because of the high incidence of fetch requests as compared to store requests and, since access to extended main storage depends in part upon access to the queue, the separate address path improves the overall access time.

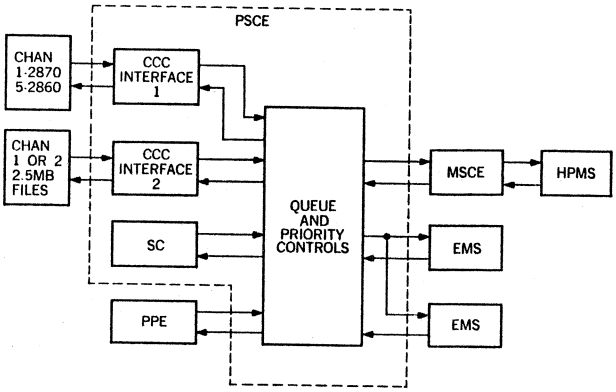


Figure 8 Block diagram of PSCE organization.

Unit return busses Two unit return busses are provided, one for the processor and another for all other requesting units. These busses are fed from each queue and return data to the requesting units. The separate CPU bus is included to minimize access time. In general, when a fetch is made, a queue register is held as a sink for the returning data. Data returns from storage are placed in the queue and can be sent immediately to the requesting unit or held until requested by I/O channels. The separate processor return bus allowed improvements in this procedure since, with the separate bus, the processor will not encounter conflicts with higher-priority unit returns. A further improvement was realized by utilizing the processor return bus in such a way that data returning from extended main storage could be sent directly to the processor without passing through the queue. This approach allows queues being used for fetches to be made available immediately after the fetch request is sent to storage. This means that fewer queues are required to handle any given processor request rate and more queues are available to other units.

Storage request busses Two storage request busses are provided, one for extended main storage and another for high performance main storage via the MSCE. These busses provide independent select paths to the two groups of storage and, since traffic to both groups can be high, these paths eliminate unnecessary inter-group conflicts which tend to increase access time.

Storage return busses Three independent storage busses

are provided in order to simultaneously handle up to three different storage access times. One is for returns from the MSCE and the others provide for up to two different storage speeds.

Queue priority

Queue priority can best be described in two parts, input priority and output priority.

Input priority In general, input priority is concerned with entry to the queue. The key functions performed are:

- Maintain dynamic queue availability status.
- Reserve queues for I/O channels upon request from the common channel control section of the PSCE.
- Assign unreserved queues to incoming requests based upon unit priority and queue availability.
- Assign reserved queues to I/O channel requests. A pre-priority function among individual I/O channels is performed by the channel controls which present a single request to input priority.
- Route returning storage data to the proper queues and update queue status as a result of these returns.

Output priority Output priority continually monitors the status of the queue in order to determine actions to be taken on the storage request busses and unit return busses. The following decision mechanism is simultaneously applied for each storage request bus:

- Compare available storage status with all requests in the queue in order to determine which request should access storage.
- If more then one request finds an available storage unit then select the highest priority unit among those requesting.
- If there is more than one outstanding request from a selecting unit, storage accesses are made on a first-in, first-out basis.

For the unit return busses the following actions are taken:

- Route memory returns to the central processor.
- Inform other units of returns available in the queue and return them on a first-in, first-out basis when requested.

Storage channel

The function of the storage channel is to provide fast data transfer from storage to storage, overlapped with other system activity. The storage channel operates as an independent unit with respect to the queue and is not treated as "just another channel." Communication delays encountered in conventionally independent I/O channels have been eliminated by integrating the storage channel with the PSCE.

Of the units that require access to storage, the storage channel was given lowest priority. I/O channels require higher priority because of their overrun nature; the peripheral processor, which may also have I/O channels operating, also requires higher priority; since CPU accesses imply an immediate need for storage and storage channel accesses imply a future or less immediate requirement, the CPU was also given higher priority. This decision was based on the fact that delayed processor access delays the system immediately while delayed storage channel access *may* delay the system in the future.

I/O activity and especially CPU activity will cause conflicts with storage channel activity. Every storage conflict will delay the storage channel and high interleaving of storage will help only to reduce the probability of such conflicts. However, if the storage channel can circumvent a current conflict and attempt its next access, the probability of a second storage conflict is considerably reduced. This, of course is the central philosophy of the PSCE, i.e., to permit units to access storage out of sequence in order to better utilize the high interleave and thereby increase both the overall rate of each unit and the effective use of storage.

The storage channel can initiate a fetch request every cycle and does so as long as space is available in the queue. Each queue register used becomes a sink for the data from storage. If this data were returned to the storage channel, it would eventually be sent back to the queue for storing in memory. It is not desirable to permit this round trip of data from queue to channel and back, because it would add handling time and buffering requirements to the storage channel. Instead, the data remain in the queue and a mechanism is provided to bring the store address to the queue.

The fact that fetches are made out of sequence from the queue implies that these store addresses would have to be supplied out of sequence in order to complete the data transfer as quickly as possible. Generation of out-of-sequence store addresses entails extensive address buffering and sequence controls, and a different mechanism, called re-address, is used. When the storage channel sends a fetch request to the queue, the data field of the queue register is "empty." In addition to the normal entry of the fetch address, the store address for that word is generated and placed in the "empty" data field. Whenever the fetch address is sent from the queue to storage, the store address is moved from the data field to the address field where it waits for the data to return from storage. As soon as the data return the store can be made, thereby allowing storage channel stores to be made out of sequence as well as fetches.

Since queue availability is a decisive factor in storage channel transfer rates, any factor which tends to increase queue availability also tends to increase this rate. As storage access time decreases, queues become available more quickly because the total fetch-store time is decreased. Table 2 shows the effect of the access time of the 3/4-μsec storage unit on storage channel transfer rates.

As system activity into storage increases, storage conflicts also increase and the number of queues available becomes more important in order to provide some minimum storage channel rate. Table 3 illustrates this fact by showing estimated transfer rates of the storage channel as a function of queues available and of storage conflicts, assuming a ten cycle access time. It should be noted that since a simple algorithm was used, these rates are only representative, but they do serve to illustrate the point.

Common channel control

The common channel control (CCC) uses one interface to provide for the attachment of up to five IBM 2860 Selector Channels plus one IBM 2870 Multiplexor Channel to the Model 91. The Selector Channels communicate directly with the CCC which in turn communicates with the input and output control sections of the PSCE. The CCC also has the potential for attachment of two very high performance channels (2.5 megabyte/sec rate) through a second interface. This interface is designed to minimize the communication time required to service a channel request. It is expected that the time required to service a channel will be reduced from 1 μsec for the standard interface to approximately 0.2 μsec for this second interface.

The CCC will accept a modified Selector Channel interface. The change in the interface was made to provide buffer control for channels that control devices with rates $\leq$1.25 megabytes. The changes allow words to be returned to different channels in a sequence different from that in which the requests were generated. This permits an aggregate channel rate which is higher than is normally possible over the standard Selector Channel interface. In addition, the changes permit the CCC to pre-fetch data for 1.25 megabyte/sec devices. Pre-fetching permits the overlapping of storage access time with channel service time. This overlapping allows the CCC to control the operation of four IBM 2860 Selector Channels with 1.25 megabyte/sec devices (no data chaining) into a 3/4 μsec. memory without the risk of overrun. Without pre-fetching it would only be possible to run two Selector Channels with the same restrictions stated above.

- *PSCE serviceability*

A feature included in the design of the PSCE permits it to be operated in a mode that does not depend on the availability of I/O equipment, storage or processors. The PSCE queue and controls can be exercised in a loop to repeat particular patterns. A separate maintenance panel is provided for the special PSCE maintenance features. The panel also permits maintenance on the PSCE to be overlapped with maintenance of other parts of the CPU. Serviceability of the PSCE is enhanced by these features:

- Variable effective queue size permits failures to be isolated.

Table 3 Storage channel transfer rates (assuming 750-nsec extended main storage and 750-nsec high performance main storage.

No. of queues available to storage channel	*Transfer rate, megabytes/sec*		
	Case I	*Case II*	*Case III*
1[a]	8.3	6.0	4.8
2	16.6	11.9	19.5
3	24.9	17.5	13.7
4	33.3	23.0	17.0
5	41.6	28.4	21.7
6	49.8	33.6	25.4
7	58.3	38.6	29.0
8	66.4	43.5	32.4

CASE I: Probability of EMS busy = 0
Probability of HPMS busy = 0
CASE II: Probability of one of EMS or HPMS busy = 1/2
Probability of both EMS and HPMS busy = 0
CASE III: Probability of both EMS and HPMS busy = 1/2
Probability of both EMS and HPMS not busy = 1/2.

[a] Use this entry for addresses on single-word boundaries.

- Queue contents on stores can be saved until storage advance time. This allows correlation of storage-detected errors with the contents of the queue register that generated the storage select.
- A register in the queue can "freeze" its contents on error. This feature allows all data associated with a request to be retained for future display or log out.

PSCE-MSCE interaction

It should be noted that the queue buffers are general purpose whereas the processor request stack in the MSCE has no provision for data bits, uses the processor data buffers, and is tailored to processor requests. Consequently, the PSCE buffering capabilities (8 queue positions) are utilized by both PSCE-to-main storage requests and by processor-to-extended storage requests. PSCE-to-main storage requests appear at the MSCE from the queue only after the specific storage unit has been reserved and becomes available for selection. Thus, two-way communication, on a main storage interleave-factor basis (4, 8, or 16), exists between the MSCE and the PSCE for PSCE-to-main storage requests. In other words, the PSCE monitors the state of each main storage unit.

Processor-to-extended storage requests can appear at the PSCE at any time (provided queue positions are available), independently of the immediate availability of the desired extended storage. Again, the PSCE buffering capability is utilized by stacking processor requests in the queue until service time. In effect, then, the processor requests monitor the state of the queue, rather than the

state of extended storage units. Processor-to-extended storage requests are (1) transmitted across the interface, (2) buffered in the queue, (3) transmitted to priority controls, and (4) permitted to select extended main storage. Conversely, PSCE-to-main storage requests (1) are buffered in the queue (2) enter priority (3) are transmitted across the interface, and (4) select main storage. Both types of requests use the queue as a buffer and enter priority only after the requesting address enters the queue.

The MSCE and the PSCE are synchronized to receive fetched data from each other on any cycle, although the control technique is different in each control element. The PSCE has unique data paths for main storage data returns and extended storage returns to the queue. The MSCE accepts data from extended storage by orthogonally multiplexing the main storage and extended storage.

Acknowledgments

The design of the storage system was a group effort with many contributors. In particular Messrs. M. C. Dales, S. A. Calta, H. A. Carlson, A. Gomez, L. W. Kaumeyer, J. Kloepping, and J. V. Mizzi contributed in the early phases of design. Also, the instruction unit design group gave valuable suggestions and criticisms. The simulations mentioned were developed by Mr. P. S. Cheng and Mr. J. R. Johnson.

Received November 1, 1965.

D. W. Anderson
F. J. Sparacio
R. M. Tomasulo

The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling

Abstract: The System/360 Model 91 central processing unit provides internal computational performance one to two orders of magnitude greater than that of the IBM 7090 Data Processing System through a combination of advancements in machine organization, circuit design, and hardware packaging. The circuits employed will switch at speeds of less than 3 nsec, and the circuit environment is such that delay is approximately 5 nsec per circuit level. Organizationally, primary emphasis is placed on (1) alleviating the disparity between storage time and circuit speed, and (2) the development of high speed floating-point arithmetic algorithms.

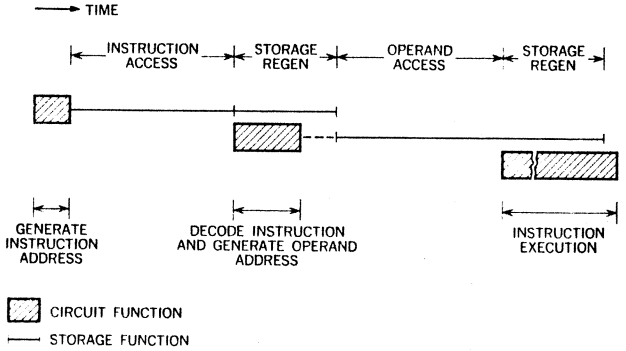
This paper deals mainly with item (1) of the organization. A design is described which improves the ratio of storage bandwidth and access time to cycle time through the use of storage interleaving and CPU buffer registers. It is shown that history recording (the retention of complete instruction loops in the CPU) reduces the need to exercise storage, and that sophisticated employment of buffering techniques has reduced the effective access time. The system is organized so that execution hardware is separated from the instruction unit; the resulting smaller, semiautonomous "packages" improve intra-area communication.

Introduction

This paper presents the organizational philosophy utilized in IBM's highest performance computer, the System/360[1] Model 91. The first section of the paper deals with the development of the assembly-line processing approach adopted for the Model 91. The organizational techniques of storage interleaving, buffering, and arithmetic execution concurrency required to support the approach are discussed. The final topic of this section deals with design refinements which have been added to the basic organization. Special attention is given to minimizing the time lost due to conditional branches, and the basic interrupt problem is covered.

The second section is comprised of a treatment of the instruction unit of the Model 91. It is in this unit that the basic control is exercised which leads to attainment of the performance objectives. The first topic is the fetching of instructions from storage. Branching and interrupting are discussed next. Special handling of branching, such that storage accessing by instructions is sometimes eliminated, is also treated. The final section discusses the interlocks required among instructions as they are issued to the execution units, the initiation of operand fetches from storage, status switching operations, and I/O handling.

Figure 1 Typical instruction function time sequence.



CPU organization

The objective of the Model 91 is to attain a performance greater by one to two orders of magnitude than that of the IBM 7090. Technology (that is, circuitry and hardware) advances* alone provide only a fourfold performance increase, so it is necessary to turn to organizational techniques for the remaining improvement. The appropriate

* Circuits employed are from the IBM ASLT family and provide an in-environment switching time in the 5 nsec range.

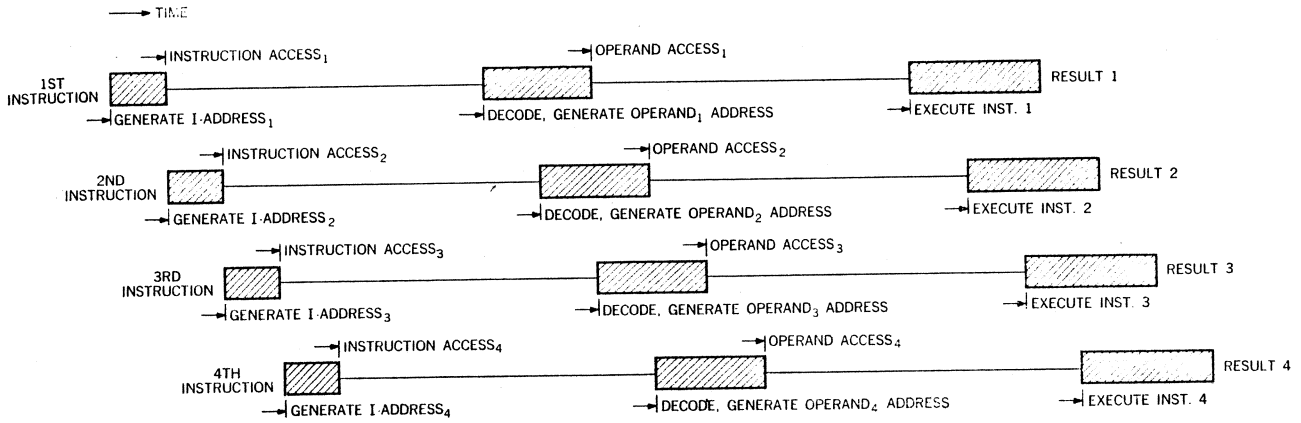


Figure 2 Illustration of concurrency among successive instructions.

selection of existing techniques and the development of new organizational approaches were the objectives of the Model 91 CPU design.

The primary organizational objective for a high performance CPU is concurrency—the parallel execution of different instructions. A consideration of the sequence of functions involved in handling a typical processor instruction makes the need for this approach evident. This sequence—instruction fetching, instruction decoding, operand address generating, operand fetching, and instruction execution—is illustrated in Fig. 1. Clearly, a primary goal of the organization must be to avoid the conventional concatenation of the illustrated functions for successive instructions. Parallelism accomplishes this, and, short of simultaneously performing identical tasks for adjacent instructions, it is desired to "overlay" the separate instruction functions to the greatest possible degree. Doing this requires separation of the CPU into loosely coupled sets of hardware, much like an assembly line, so that each hardware set, similar to its assembly line station counterpart, performs a single specific task. It then becomes possible to enter instructions into the hardware sets at shortly spaced time intervals. Then, following the delay caused by the initial filling of the line, the execution results will begin emerging at a rate of one for each time interval. Figure 2 illustrates the objective of the technique.

Defining the time interval (basic CPU clock rate) around which the hardware sets will be designed requires the resolution of a number of conflicting requirements. At first glance it might appear that the shorter the time interval (i.e., the time allocated to successive assembly line stations), the faster the execution rate will be for a series of instructions. Upon investigation, however, several parameters become apparent which frustrate this seemingly simple pattern for high performance design. The parameters of most importance are:

1. An assembly-line station platform (hardware "trigger") is necessary within each time interval, and it generally adds a circuit level to the time interval. The platform "overhead" can add appreciably to the total execution time of any one instruction since a shorter interval implies more stations for any pre-specified function. A longer instruction time is significant when sequential instructions are logically dependent. That is, instruction n cannot proceed until instruction $n + 1$ is completed. The dependency factor, therefore, indicates that the execution time of any individual instruction should not be penalized unnecessarily by overhead time delay.

2. The amount of control hardware—and control complexity—required to handle architectural and machine organization interlocks increases enormously as the number of assembly line stations is increased. This can lead to a situation for which the control paths determining the gating between stations contain more circuit levels than the data paths being controlled.

Parameters of less importance which influence the determination of the basic clock rate include:

1. The number of levels needed to implement certain basic data paths, e.g., address adders, instruction decoders, etc.

2. Effective storage access time, especially when this time is relatively short. Unless the station-to-station time interval of the CPU is a sub-multiple of storage access time the synchronization of storage and CPU functions will involve overhead time.

Judgment, rather than algorithms, gave the method by which the relative weights of the above parameters were evaluated to determine the basic station-to-station time

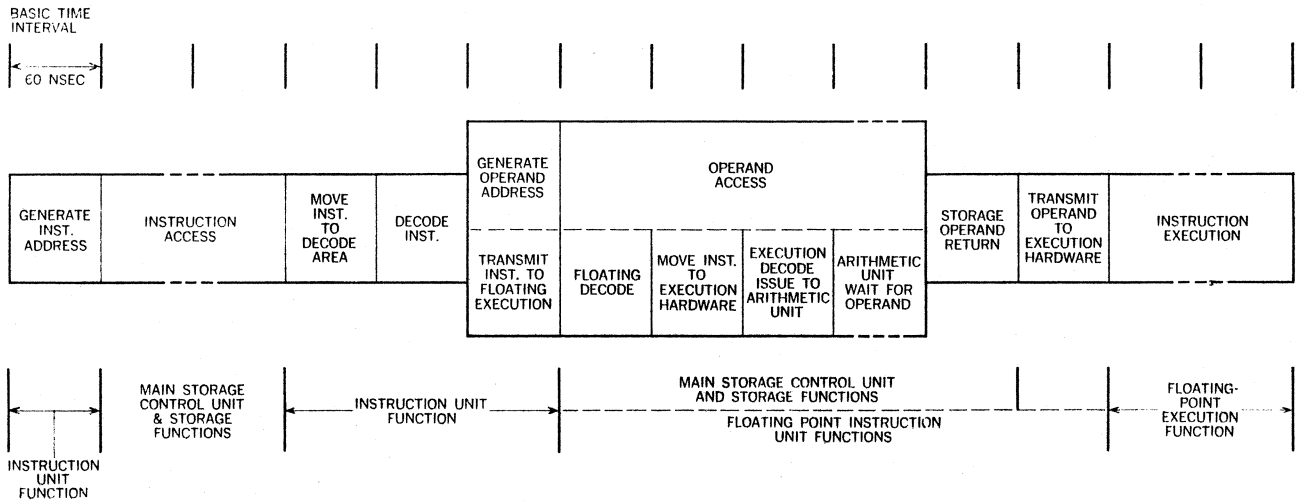


Figure 3 CPU "assembly-line stations required to accommodate a typical floating-point storage-to-register instruction.

interval.* The interval selected led to a splitting of the instruction handling functions as illustrated in Fig. 3.†

It can be seen in Fig. 3 that the basic time interval accommodates the assembly line handling of most of the basic hardware functions. However, the storage and many execution operations require a number of basic intervals. In order to exploit the assembly line processing approach despite these time disparities, the organizational techniques of storage interleaving,[2] arithmetic execution concurrency, and buffering are utilized.

Storage interleaving increases the storage bandwidth by enabling multiple accesses to proceed concurrently, which in turn enhances the assembly line handling of the storage function. Briefly, interleaving involves the splitting of storage into independent modules (each containing address decoding, core driving, data read-out sense hardware, and a data register) and arranging the address structure so that adjacent words—or small groups of adjacent words—reside in different modules. Figure 4 illustrates the technique.

The depth of interleaving required to support a desired concurrency level is a function of the storage cycle time, the CPU storage request rate, and the desired effective access time. The effective access time is defined as the sum of the actual storage access time, the average time spent waiting for an available storage, and the communication time between the processor and storage.*

Execution concurrency is facilitated first by the division of this function into separate units for fixed-point execution and floating-point execution. This permits instructions of the two classes to be executed in parallel; in fact, as long as no cross-unit dependencies exist, the execution does not necessarily follow the sequence in which the instructions are programmed.

Within the fixed-point unit, processing proceeds serially, one instruction at a time. However, many of the operations

* Effective access times ranging from 180–600 nsec are anticipated, although the design of the Model 91 is optimized around 360 nsec. Interleaving 400 nsec/cycle storage modules to a depth of 16 satisfies the 360 nsec effective access design point.

Figure 4 Arrangement of addresses in n storage modules of m words per module.

ADDRESS 0	1		$n-1$
n	$n+1$		$2n-1$
$2n$	$2n+1$		$3n-1$
$(m-2)n$	$(m-2)n+1$		$(m-1)n-1$
$(m-1)n$	$(m-1)n+1$		$mn-1$

* The design objective calls for a 60 nsec basic machine clock interval. The judgment exercised in this selection was tempered by a careful analysis of the number of circuit levels, fan in, fan out, and wiring lengths required to perform some of the basic data path and control functions. The analysis indicated that 11 or 12 circuit levels of 5–6 nsec delay per level were required for the worst-case situations.

† Figure 3 also illustrates that the hardware sets are grouped into larger units—instruction unit, main storage control element, fixed-point execution unit, floating-point execution unit. The grouping is primarily caused by packaging restrictions, but a secondary objective is to provide separately designable entities having minimum interfacing. The total hardware required to implement the required CPU functions demands three physical frames, each having dimensions 66″ L × 15″ D × 78″ H. The units are allocated to the frames in such a way as to minimize the effects of interframe transmission delays.

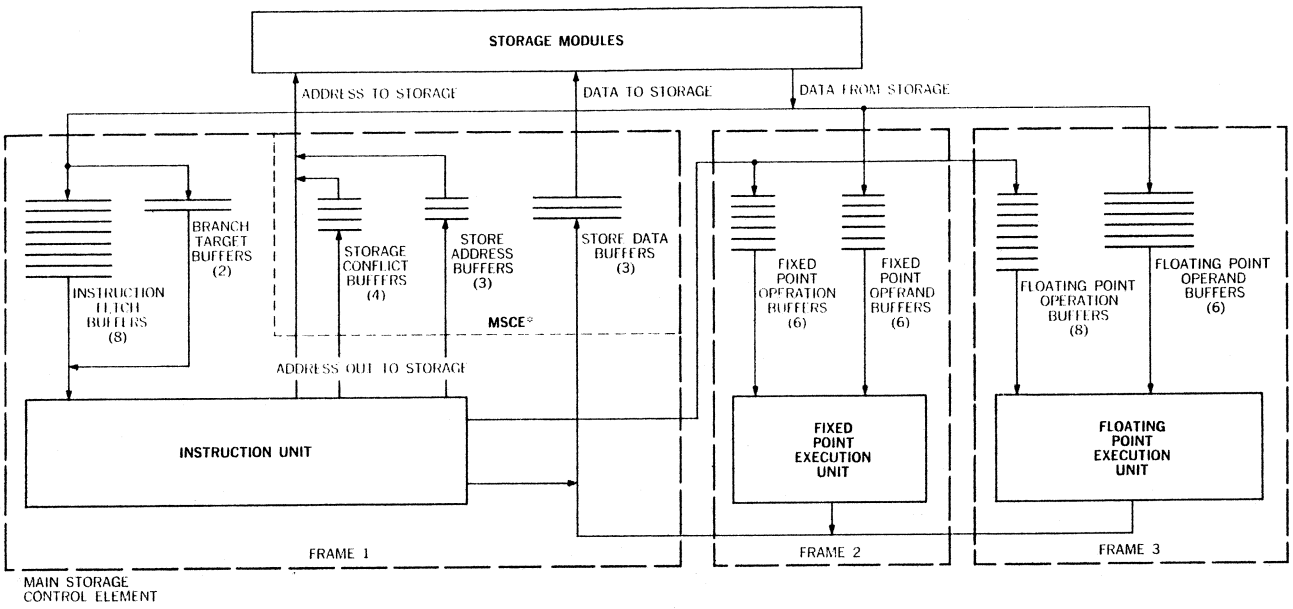


Figure 5 Buffer allocation and function separation.

require only one basic time interval to execute, and special emphasis is placed on the storage-to-storage instructions to speed up their execution. These instructions (storage-to-storage) enable the Model 91 to achieve a performance rate of up to 7 times that of the System/360 Model 75 for the "translate-and-test" instruction. A number of new concepts and sequences[3] were developed to achieve this performance for normally storage access-dependent instructions.

The floating-point unit is given particular emphasis to provide additional concurrency. Multiple arithmetic execution units, employing fast algorithms for the multiply and divide operations and carry look-ahead adders, are utilized.[4] An internal bus has been designed[5] to link the multiple floating-point execution units. The bus control correctly sequences dependent "strings" of instructions, but permits those which are independent to be executed out of order.

The organizational techniques described above provide balance between the number of instructions that can be prepared for arithmetic execution and those that can actually be executed in a given period, thereby preventing the arithmetic execution function from creating a "bottleneck" in the assembly line process.

Buffering of various types plays a major role in the Model 91 organization. Some types are required to implement the assembly line concept, while others are, in light of the performance objectives, architecturally imposed. In all cases the buffers provide queueing which smooths the total instruction flow by allowing the initiating assembly line stations to proceed despite unpredictable delays down the line. Instruction fetch, operand fetch, operand store, operation, and address buffering are utilized among the major CPU units as illustrated in Fig. 5.*

Instruction fetch buffering provides return data "sinks" for previously initiated instruction storage requests. This prefetching hides the instruction access time for straight-line (no branching) programs, thereby providing a steady flow of instructions to the decoding hardware. The buffering is expanded beyond this need to provide the capacity to hold program loops of meaningful size. Upon encountering a loop which fits, the buffer locks onto the loop and subsequent branching requires less time, since it is to the buffers rather than to storage. The discussion of branching given later in this paper gives a detailed treatment of the loop action.

Operand fetch buffers effectively provide a queue into which storage can "dump" operands and from which execution units can obtain operands. The queue allows the isolation of operand fetching from operand usage for the storage-to-register and storage-to-storage instruction types. The required depth† of the queue is a function of the number of basic time intervals required for storage

* Eight 64-bit double words comprise the array of instruction buffers. Six 32-bit operand buffers are provided in the fixed-point execution unit, while six 64-bit buffers reside in the floating-point execution unit. Three 64-bit store operand buffers along with three store address and four conflict address buffers are provided in the main storage control element. Also, there are six fixed-point and eight floating-point operand buffers.

† To show precise algorithms defining these and other buffering requirements is impractical, since different program environments have different needs. The factors considered in selecting specific numbers are cited instead.

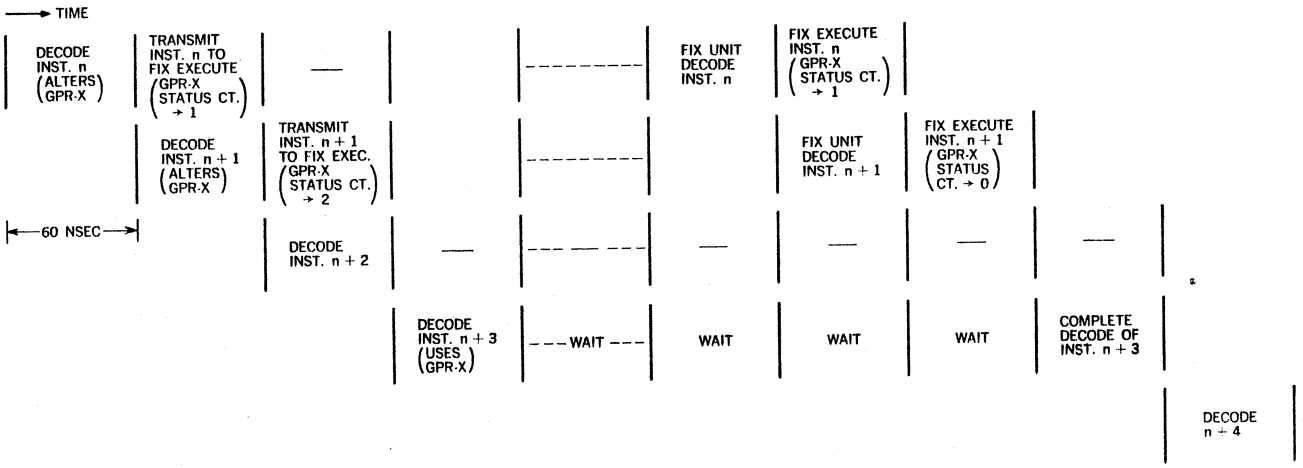


Figure 6 GPR address interlock.

accessing, the instruction "mix" of the operating program, and the relative time and frequency of execution bottlenecks. Operand store buffering provides the same function as fetch buffering, except that the roles of storage and execution are reversed. The number of store buffers required is a function of the average waiting time encountered when the desired storage module is busy and the time required for the storage, when available, to utilize the operand.

Operation buffers in the fixed-point and floating-point execution units allow the instruction unit to proceed with its decoding and storage-initiating functions while the execution units wait for storage operands or execution hardware. The depth of the operation buffering is related to the amount of operand buffering provided and the "mix" of register-to-register and storage-to-register instruction types.

Address buffering is used to queue addresses to busy storage modules and to contain store addresses during the interval between decoding and execution of store instructions. The instruction unit is thereby allowed to proceed to subsequent instructions despite storage conflicts or the encountering of store operations. These buffers have comparators associated with them to establish logical precedence when conflicting program references arise. The number of necessary store address buffers is a function of the average delay between decode and execution, while the depth of the queue caused by storage conflicts is related to the probable length of time a request will be held up by a busy storage module.[6]

- *Concurrency limitations*

The assembly line processing approach, using the techniques of storage interleaving, arithmetic concurrency, and buffering, provides a solid high-performance base. The orientation is toward smooth-flowing instruction streams for which the assembly line can be kept full. That is, as long as station n need only communicate with station $n + 1$ of the line, highest performance is achieved. For example, floating-point problems which fit this criterion can be executed internally on the Model 91 at up to 100 times the internal speed of the 7090.[7]

There are, however, cases where simple communication between adjacent assembly line stations is inadequate, e.g., list processing applications, branching, and interrupts. The storage access time and the execution time are necessarily sequential between adjacent instructions. The organization cannot completely circumvent component delay in such instances, and the internal performance gain diminishes to about one order of magnitude greater than that of the 7090.

The list processing application is exemplified by sequentialism in addressing, which produces a major interlock situation in the Model 91. The architecturally specified usage of the general purpose registers (GPR's) for both address quantities and fixed-point data, coupled with the assembly line delay between address generation and fixed-point execution, leads to the performance slowdown. Figure 6 illustrates the interlock and the resulting delay. Instructions n and $n + 1$ set up the interlock on GPR X since they will alter the contents of X. The decode of $n + 3$ finds that the contents of X are to be used as an address parameter, and since the proper contents are not available $n + 3$ must wait until $n + 1$ is executed. The interlock technique involves assigning the decode area a status count for each GPR. A zero status count indicates availability. As fixed-point instructions pass through the decode, they increment the appropriate counter(s). A decode requiring an unavailable (non-zero status count) GPR cannot be completed. As the fixed-point execution unit

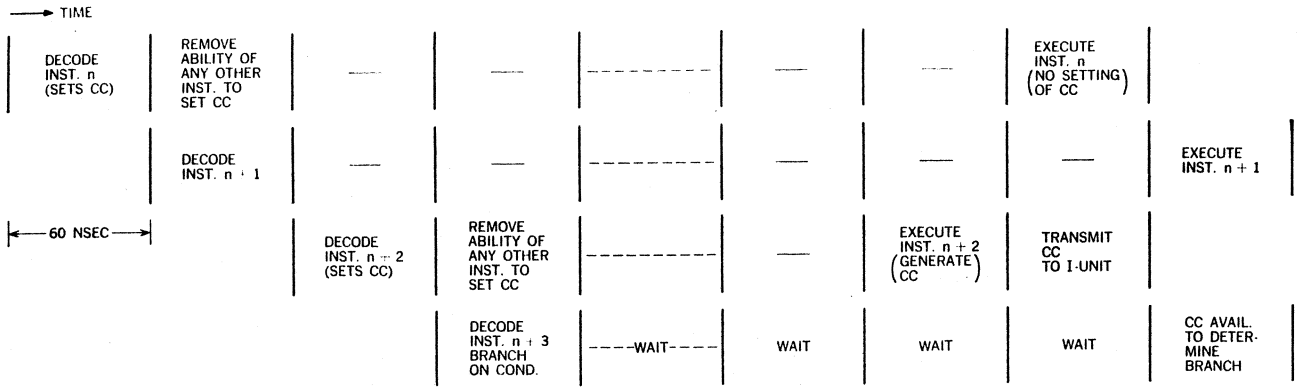


Figure 7 Condition code interlock.

completes instructions it decrements the appropriate counter(s), thus eventually freeing the register.

Branching leads to another sequential situation, since a disruption in the instruction supply is created. (Techniques employed to minimize or circumvent the storage access delay involved in obtaining the new instructions are discussed under *Instruction supplying* in the following section of this paper.) Conditional branching poses an additional delay in that the branch decision depends on the outcome of arithmetic operations in the execution units. The Model 91 has a relatively lower performance in cases for which a large percentage of conditional branch instructions lead to the branch being taken. The discontinuity is minimized, when the branch is not taken, through special handling of the condition code (CC) and the conditional branch instruction (BC). The condition code is a two-bit indicator, set according to the outcome of a variety of instructions, and can subsequently be interrogated for branching through the BC instruction. Since the code is to represent the outcome of the last decoded CC-affecting instruction, and since execution can be out of sequence, interlocks must be established to ensure this. This is accomplished, as illustrated in Fig. 7, by tagging each instruction at decode time if it is to set the CC. Simultaneously, a signal is communicated throughout the CPU to remove all tags from previously decoded but not executed instructions. Allowing only the execution of the tagged instruction to alter the code insures that the correct CC will be set. The decode hardware monitors the CPU for outstanding tags; only when none exists is the condition code considered valid for interrogation.

The organization assumes that, for a conditional branch, the CC will not be valid when the "branch-on-condition" (BC) is decoded (a most likely situation, considering that most arithmetic and logical operations set the code). Rather than wait for a valid CC, fetches are initiated for two instruction double-words as a hedge against a successful branch. Following this, it is assumed that the branch will fail, and a "conditional mode" is established. In conditional mode, shown in Fig. 8, instructions are decoded and conditionally forwarded to the execution units, and concomitant operand fetches are initiated. The execution units are inhibited from completing conditional instructions. When a valid condition code appears, the appropriate branching action is detected and activates or cancels the conditional instructions. Should the no-branch guess prove correct, a substantial head start is provided by activating the conditionally issued and initiated operand fetches for a number of instructions. If the branch is successful, the previously fetched target words are activated and provide work while the instruction fetching is diverted to the new stream. (Additional optimizing techniques are covered under the discussion of branching in a subsequent section of this paper.)

Interrupts, as architecturally constrained, are a major bottleneck to performance in the assembly line organization. Strict adherence to a specification which states that an interrupt on instruction n should logically precede and inhibit any action from being taken on instruction $n + 1$ leaves two alternatives. The first would be to force sequentialism between instructions which may lead to an interrupt. In view of the variety of interrupt possibilities defined, this course would totally thwart high performance and is necessarily discarded. The second is to set aside sufficient information to permit recovery from any interrupt which might arise. In view of the pipeline and execution concurrency which allows the Model 91 to advance many instructions beyond n prior to its execution, and to

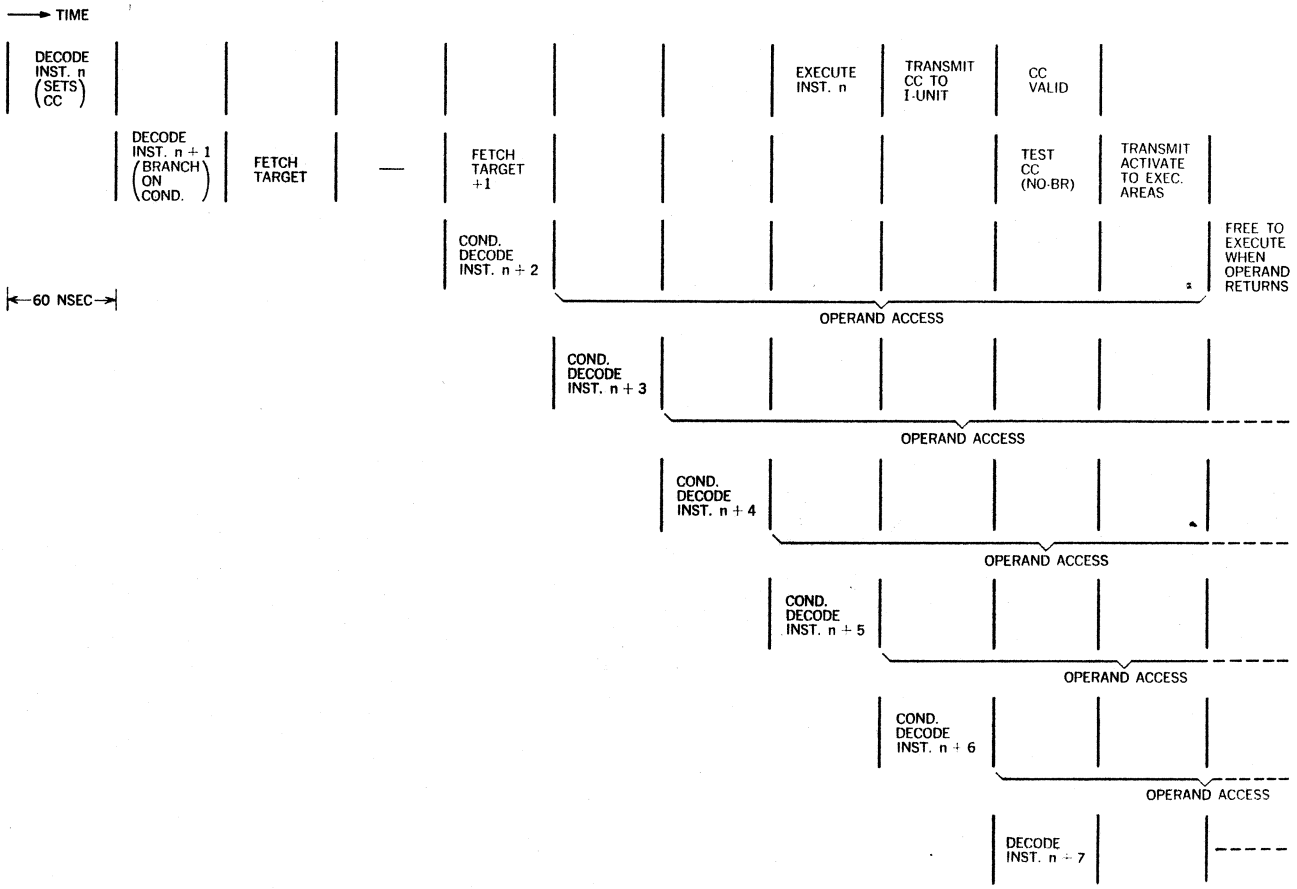


Figure 8 Conditional instruction issuing: the branch-on-condition philosophy.

execute independent instructions out of sequence ($n + m$ before n), the recovery problem becomes extremely complex and costly. Taking this approach would entail hardware additions to the extent that it would severely degrade the performance one is seeking to enhance. The impracticality of both alternatives by which the interrupt specifications could be met made it mandatory that the specifications themselves be altered. The architecture was compromised by removing the above-mentioned "precedence" and "inhibit" requirements. The specification change led to what is termed the "imprecise interrupt" philosophy of the Model 91 and reduced the interrupt bottleneck to an instruction supply discontinuity. The imprecise interrupt, and the manner in which the instruction discontinuity is minimized, are covered in the next section of the paper.

The bottlenecks discussed above gave rise to the major interlocks among the separate CPU areas. Within each of the areas, however, additional considerations hold. These are discussed as appropriate in the next section or in following papers.

Instruction unit

The central control functions for the Model 91 CPU are performed in the instruction unit. The objective here is to discuss these functions in terms of how they are performed and to include the reasons for selecting the present design. However, before proceeding with this discussion it will be useful to examine some over-all design considerations and decisions which directly affect the instruction unit functions. In approaching the design of the instruction unit, many program situations were examined, and it was found that while many short instruction sequences are nicely ordered, the trend is toward frequent branching. Such things as performing short work loops, taking new action based on data results, and calling subroutines are the bases upon which programs are built and, in many instances, these factors play a larger role in the use of available time than does execution. Consequently, emphasis on branch sequencing is required. A second finding was that, even with sophisticated execution algorithms, very

few programs can cause answers actually to flow from the assembly line at an average rate in excess of one every two cycles. Inherent inter-instruction dependencies, storage and other hardware conflicts, and the frequency of operations requiring multi-cycle execution all combine to prevent it.

Consideration of branching and execution times indicates that, for overall balance, the instruction unit should be able to surge ahead of the execution units by issuing instructions at a faster-than-execution rate. Then, when a branch is encountered, a significant part of the instruction unit slowdown will be overlapped with execution catch-up. With this objective in mind it becomes necessary to consider what constitutes a fast issue rate and what "trade-offs" would be required to achieve it. It is easily shown that issuing at a rate in excess of one instruction per cycle leads to a rapid expansion of hardware and complexity. (Variable-length instructions, adjacent instruction interdependencies, and storage requirements are prime factors involved.) A one-cycle maximum rate is thereby established, but it too presents difficulties. The assembly line process requires that both instruction fetching and instruction issuing proceed concurrently in order to hide storage delays. It is found through program analysis that slightly more than two instructions will be obtained per 64-bit instruction fetch* and that approximately 80% of all instructions require an operand reference to storage. From this it is concluded that issuing the average instruction entails approximately 1.25 storage accesses: 0.45 (instruction fetches) + 0.80 (operand fetches). This figure, with the one-per-cycle issue rate goal, clearly indicates a need for either two address paths to storage and associated return capabilities, or for multiple words returned per fetch. In considering these options, the initial tendency is to separate instruction and operand storage access paths. However, multiple paths to storage give rise to substantial hardware additions and lead to severe control problems, particularly in establishing storage priorities and interlocks due to address dependencies. With a one-at-a-time approach these can be established on each new address as it appears, whereas simultaneous requests involve doing considerably more testing in a shorter time interval. Multiple address paths to storage were considered impractical because of the unfavorable compromise between hardware and performance.

The multiple-words-returned-per-fetch option was considered in conjunction with instruction fetching since the instruction stream is comprised of sequential words. To prevent excessive storage "busying" this approach requires multiple word readout at the storage unit along with a wider data return path. Also, the interleaving factor is altered from *sequential* to *multi-sequential*, i.e., rather than having sequential double words in different storage modules, groups of sequential words reside in the same module. The interlock problems created by this technique are modest, the change in interleaving technique has little performance effect,* and storage can be (is, in some cases) organized to read out multiple words, all of which make this approach feasible. However, packaging density (more hardware required for wide data paths), storage organization constraints, and scheduling were such that this approach was also discarded. As a consequence, the single-port storage bus, which allows sequential accessing of double words, was adopted. This fact, in conjunction with the 1.25 storage accesses required per instruction, leads to a lowering of the average maximum issue rate to 0.8 instructions per machine cycle. The instruction unit achieves the issue rate through an organization which allows concurrency by separating the instruction supplying from the instruction issuing function.

- *Instruction supplying*

Instruction supplying includes the provision of an instruction stream which will support the desired issue rate in a sequential (non-branch) environment, and the ability to switch readily to a new instruction stream when required because of branching or interrupts.

Sequential instruction fetching

Provision of a sequential string of instructions has two fundamental aspects, an initiation or start-up transient, and a steady-state function. The initial transient entails filling the assembly line ahead of the decode station with instructions. In hardware terms, this means initiating sufficient instruction fetches so that, following a wait of one access time, a continuous flow of instruction words will return from storage. Three double-word fetches are the minimum required to fill the assembly line, since approximately two instructions are contained within a double word, and the design point access time is six machine cycles. The actual design exceeds the minimum for several reasons, the first being that during start-up no operand requests are being generated (there are no instructions), and consequently the single address port to storage is totally available for instruction fetching. Second, the start-up delay provides otherwise idle time during which to

* Storage-to-storage (SS) instructions are not considered here. They can be viewed as macro-operations and are treated as such by the hardware. The macro-operations are equivalent to basic instructions, and the number of micro-instructions involved in performing an SS function indicates that many instruction fetches would be required to perform the same function using other System/360 instructions.

* This is more intuitive than analytical. Certainly for strictly random addressing, the interleave technique is irrelevant. However, in real applications, programs are generally localized with (1) the instructions sequential and (2) branches jumping tens or hundreds rather than thousands of words. Data is more random because, even though it is often ordered in arrays, quite frequently many arrays are utilized concurrently. Also, various data constants are used which tend to randomize the total use. A proper analysis must consider all these factors and so becomes complex. In any event, as long as the interleave factor remains fixed the interference appears little affected by small changes in the interleaving pattern.

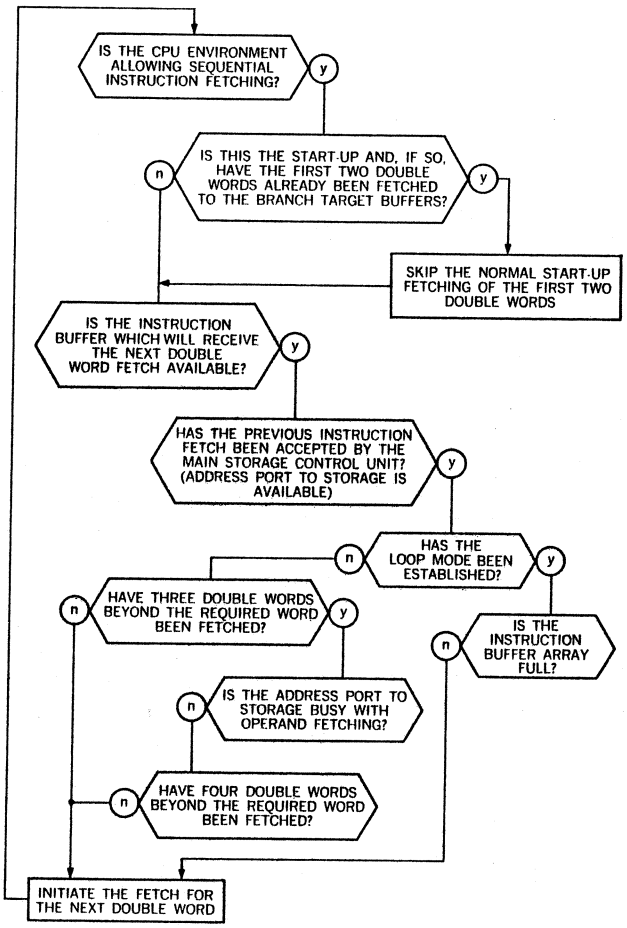


Figure 9 Flow chart of the sequential instruction-supply function.

initiate more fetches, and the eight double words of instruction buffering provide space into which the words can return. A third point is that, should storage requiring more than six cycles of access time be utilized, more fetching-ahead will be required. Finally, establishing an excess queue of instructions during the transient time will allow temporary maintenance of a full assembly line without any further instruction fetching. The significance of this action is that it allows the issuing of a short burst of instructions at a one-per-cycle rate. This follows from the fact that the single, normally shared storage address port becomes exclusively available to the issue function. A start-up fetching burst of five double instruction words was the design point which resulted when all of these factors had been considered.*

Steady-state instruction supplying serves the function of maintaining a full assembly line by initiating instruction fetches at appropriate intervals. The address port to storage is multiplexed between instruction fetches and operand fetches, with instructions receiving priority in conflict situations. An additional optimization technique allows the instruction fetching to re-advance to the start-up level of five double words ahead if storage address time "slots" become available. A flow chart of the basic instruction fetch control algorithm is shown in Fig. 9,* while Fig. 10 is a schematic of the data paths provided for the total instruction supplying function. Some of the decision blocks contained in the flow chart result from the effects of branch instructions; their function will be clarified in the subsequent discussion of branching. There are two fundamental reasons for checking buffer availability in the algorithm. First, the instruction buffer array is a modulo-eight map of storage that is interleaved by sixteen. Second, fetches can return out of order because storage may be busy or of varying performance. For example, when a branch is encountered, point one above implies that the target may overlay a fetch which has not yet returned from storage. In view of the second point, it is necessary to ensure that the unreturned fetch is ignored, as it would be possible for a new fetch to return ahead of it. Proper sequencing is accomplished by "tagging" the buffers assigned to outstanding fetches, and preventing the initiation of a new fetch to a buffer so tagged.

Branch Handling

Branching adds to the complexity of the instruction supplying function because attempts are made to minimize discontinuities caused by the branching and the consequent adverse effects on the issue rate. The discontinuities result because for each branch the supply of instructions is disrupted for a time roughly equivalent to the greater of the storage access period (start-up transient previously mentioned), or the internal testing and "housekeeping" time required to make and carry out the branch decision. This time can severely limit the total CPU performance in short program loops. It has a somewhat less pronounced effect in longer loops because the branch time becomes a smaller percentage of the total problem loop time and, more important, the instruction unit has greater opportunity to run ahead of the execution units (see Fig. 11). This last makes more time available in which to overlap the branch time with execution catch-up.

The detrimental performance effect which stems from short loops led to a dual branch philosophy. The first aspect deals with branches which are either forward into the instruction stream,† beyond the prefetched instructions, or if backward from the branch instruction, greater

* The one disadvantage to over-fetching instructions is that the extra fetches may lead to storage conflicts, delaying the subsequently initiated operand fetches. This is a second-order effect, however, first because it is desirable for the instruction fetches to win conflicts unless these fetches are rendered unnecessary by an intervening branch instruction, and secondly because the sixteen-deep interleaving of storage significantly lowers the probability of the conflict situation.

* In this flow chart, unlabeled exits from decision blocks imply that a "wait" state will exist until the required condition has been satisfied.

† In the actual program the branch instruction would precede the target for this case.

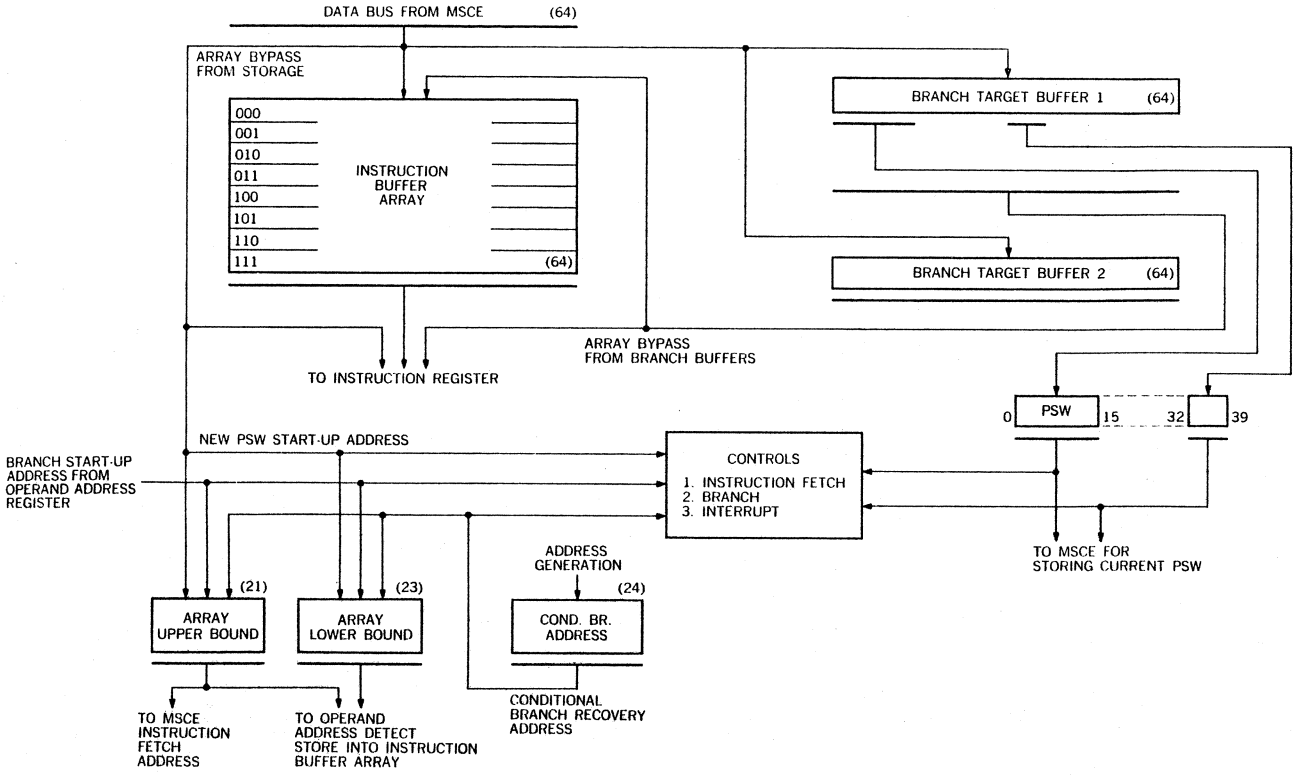
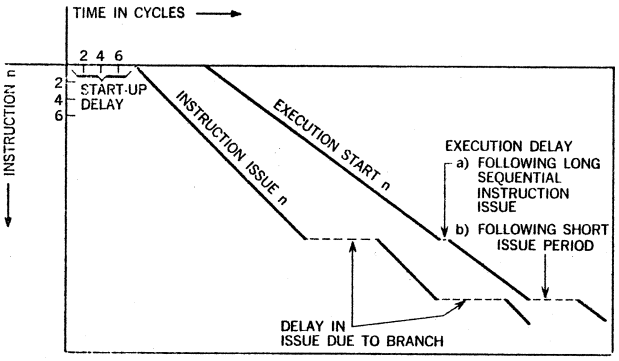


Figure 10 Data paths for the basic instruction supply.

than eight double-words back. In these situations the branch storage-delay is unavoidable. As a hedge against such a branch being taken, the branch sequencing (Fig. 12) initiates fetches for the first two double words down the target path. Two branch buffers are provided (Fig. 10—the instruction supply data flow) to receive these words, in order that the instruction buffer array will be unaffected if the result is a no branch decision. The branch housekeeping and decision making are carried on in parallel with the access time of the target fetches. If a branch decision is reached before the access has been completed, additional optimizing hardware routes the target fetch around the buffer and directly to the instruction register, from which it will be decoded. Minor disadvantages of the technique are that the "hedge" fetching results in a delay of the no-branch decision and may lead to storage conflicts. Consequently, a small amount of time is lost for a branch which "falls through."

The second aspect of the branch philosophy treats the case for which the target is backward within eight double words of the branch instruction. A separation of eight double words or less defines a "short" loop—this number being chosen as a hardware/performance compromise. Part of the housekeeping required in the branch sequencing is a "back eight" test. If this test is satisfied the instruction unit enters what is termed "loop mode." Two beneficial results derive from loop mode. First, the complete loop is fetched into the instruction buffer array, after which instruction fetching ceases. Consequently, the address port to storage is totally available for operand fetching and a one instruction per cycle issue rate is possible. The second advantage gained by loop mode is a reduction by a factor of two to three in the time required to sequence the loop-establishing branch instruction. (For example, the "branch on index" instruction normally requires eight

Figure 11 Schematic representation of execution delays caused by (branch) discontinuities in the instruction issuing rate, for the case in which the issuing rate is faster than the execution rate.



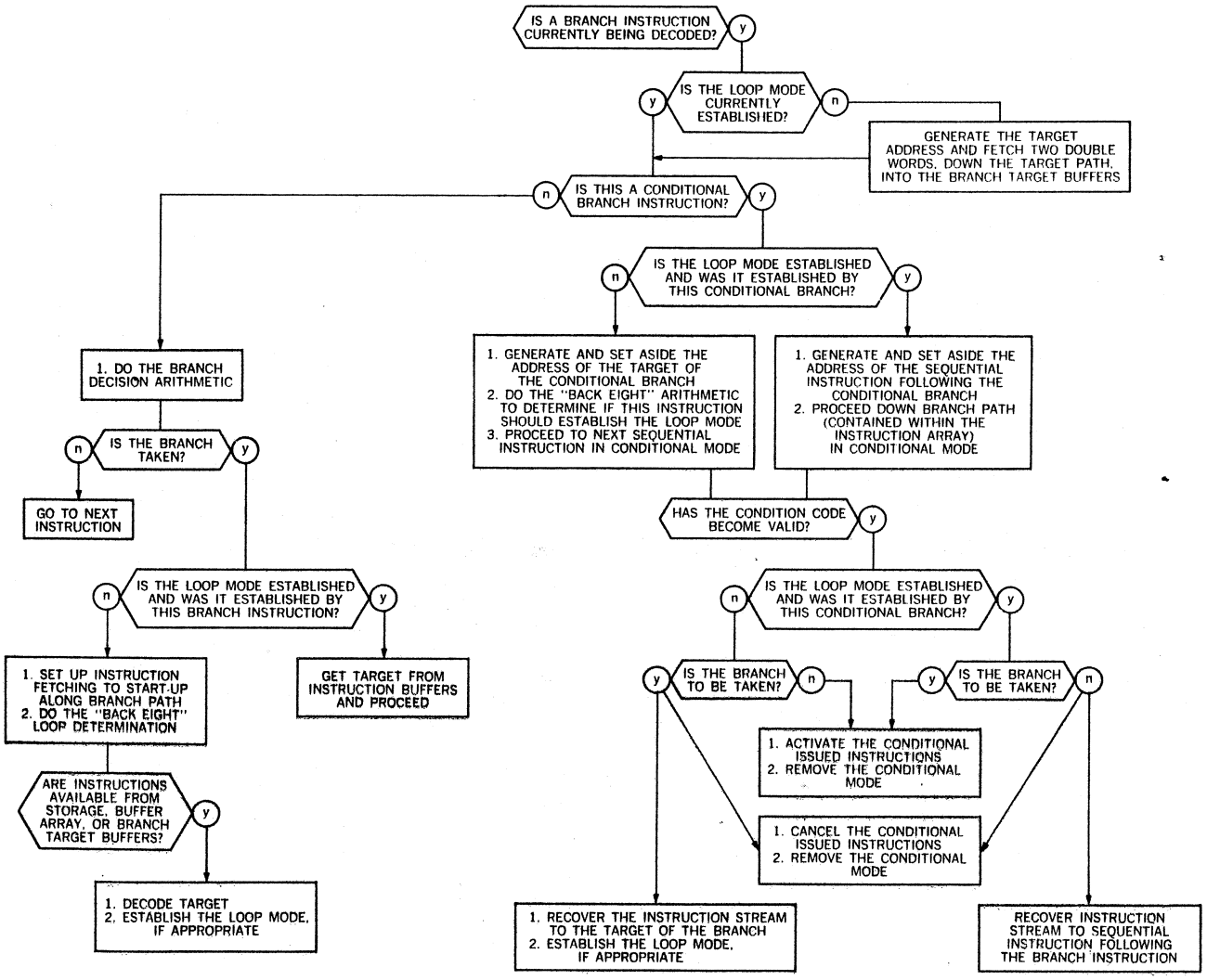


Figure 12 Flow chart of the branching sequence.

cycles for a successful branch, while in loop mode three cycles are sufficient.) In many significant programs it is estimated that the CPU will be in loop mode up to 30% of the time.

Loop mode may be established by all branch instructions except "branch and link." It was judged highly improbable that this instruction would be used to establish the type of short repetitious program loops to which loop mode is oriented. A conditional branch instruction, because it is data dependent and therefore less predictable in its outcome than other branch instructions, requires special consideration in setting up loop mode. Initial planning was to prevent looping with this instruction, but consultation with programmers has indicated that loops are frequently closed conditionally, since this allows a convenient means for loop breaking when exception conditions arise.

Furthermore, in these situations the most likely outcome is often known and can be utilized to bias the branch decision whichever way is desirable. For such reasons, the "back eight" test is made during the sequencing of a conditional branch instruction, and the status is saved through conditional mode. Should it subsequently be determined that the branch is to be taken, and the "saved" status indicates "back eight," loop mode is established. Thereafter the role of conditional mode is reversed, i.e., when the conditional branch is next encountered, it will be assumed that the branch will be taken. The conditionally issued instructions are from the target path rather than from the no-branch path as is the case when not in loop mode. A cancel requires recovery from the branch guess. Figure 12 is a flow chart of this action. In retrospect, the conditional philosophy and its effects on loop mode, although significant to the performance of the CPU and conceptually simple, were found to require numerous interlocks through-

out the CPU. The complications of conditional mode, coupled with the fact that it is primarily aimed at circumventing storage access delays, indicate that a careful re-examination of its usefulness will be called for as the access time decreases.

Interrupts

Interrupts, like branching, are another disruption to a smooth instruction supply. In the interrupt situation the instruction discontinuity is worsened because, following the recognition of the interrupt, two sequential storage access delays are encountered prior to receiving the next instruction.* Fortunately, and this is unlike branches, interrupts are relatively infrequent. In defining the interrupt function it was decided that the architectural "imprecise" compromise mentioned in the previous section would be invoked only where necessary to achieve the required performance. In terms of the assembly line concept, this means that interrupts associated with an instruction which can be uncovered during the instruction unit decode time interval will conform with the specifications. Consequently, only interrupts which result from address, storage, and execution functions are imprecise.

One advantage of this dual treatment is that System/360 compatibility is retained to a useful degree. For example, a programming strategy sometimes employed to call special subroutines involves using a selected invalid instruction code. The ensuing interrupt provides a convenient subroutine entry technique. Retaining the compatible interrupt philosophy through the decoding time interval in the Model 91 allows it to operate programs employing such techniques. The manifestation of this approach is illustrated in the flow chart of Fig. 13. In accordance with System/360 specifications, no further decoding is allowed once either a precise or an imprecise interrupt has been signalled. With the assembly line organization, it is highly probable that at the time of the interrupt there will be instructions still in the pipeline which should be executed prior to changing the CPU status to that of the interrupt routine. However, it is also desirable to minimize the effect of the interrupt on the instruction supply, so the new status word is fetched to the existing branch target buffer in parallel with the execution completion. After the return from storage of the new status word, if execution is still incomplete, further optimizing allows the fetching of instructions for the interrupt routine. Before proceeding, it becomes necessary to consider an implication resulting

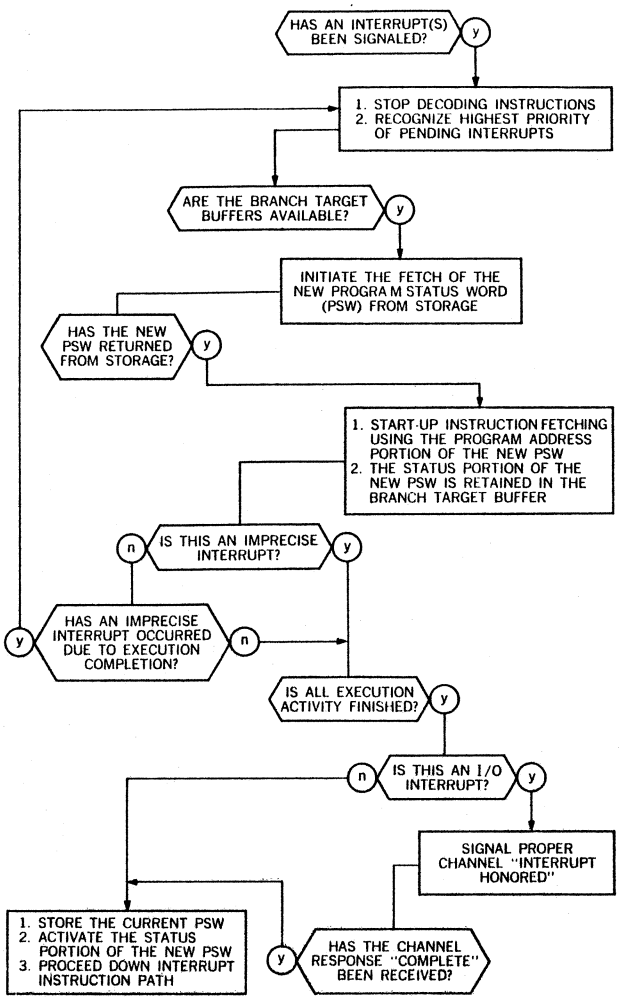


Figure 13 Flow chart of the interrupt sequence.

from the dual interrupt philosophy. Should a precise interrupt have initiated the action, it is possible that the execution "cleanup" will lead to an imprecise condition. In this event, and in view of the desire to maintain compatibility for precise cases, the logically preceding imprecise signal should cancel all previous precise action. The flow chart (Fig. 13) illustrates this cancel-recovery action. Should no cancel action occur (the more likely situation), the completion of all execution functions results, with one exception, in the release of the new status word and instruction supply. The I/O interrupts require special consideration because of certain peculiarities in the channel hardware (the System 360/Model 60–75 channel hardware is used). Because of them, the CPU-channel communication cannot be carried out in parallel with the execution completion. However, the relative infrequency of I/O interrupts renders negligible the degradation caused by this.

* This arises from the architectural technique of indirectly entering the interrupt subroutines. In System/360 the interrupts are divided into classes. Each class is assigned a different, fixed low storage address which contains the status to which the CPU shall be set should an interrupt of the associated class occur. Part of this status is a new program address. Consequently, interrupting requires obtaining a new supply of instructions from storage indirectly, through the new status word.

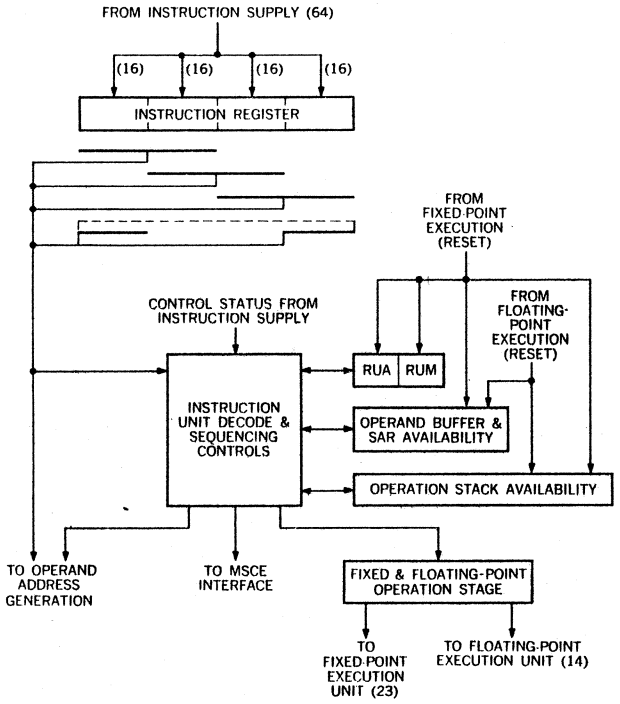


Figure 14 Data flow for instruction decoding and instruction issuing.

• *Instruction issuing*

The instruction-issuing hardware initiates and controls orderly concurrency in the assembly line process leading to instruction execution. It accomplishes this by scanning each instruction, in the order presented by the program, and clearing all necessary interlocks before releasing the instruction. In addition, should a storage reference be required by the operation, the issuing mechanism performs the necessary address calculations, initiates the storage action, and establishes the routing by which the operand and operation will ultimately be merged for execution. In addition, certain essential inter-instruction dependencies are maintained while the issue functions proceed concurrently.

In terms of the assembly line of Fig. 3, the moving of instructions to the decode area, the decode, and the operand address generation comprise the issue stations. The moving of instructions to the decode area entails the taking of 64-bit double-words, as provided by the instruction supply, and extracting from them the proper instruction half-words, one instruction at a time. The instruction register is the area through which this is accomplished (Fig. 14). The register efficiently handles variable-length instructions and provides a stable platform from which to decode. All available space in this 64-bit register is kept full of instructions yet to be decoded, provided only that the required new instruction information has returned from storage. The decoder scans across the instruction register, starting at any half-word (16-bit) boundary, with new instructions refilling any space vacated by instruction issuing. The register is treated conceptually as a cylinder; i.e., the end of the register is concatenated with the beginning, since the decode scan must accommodate instructions which cross double-word boundaries.

The decoding station is the time interval during which instruction scanning and interlock clearing take place. Instruction-independent functions (interval timer update, wait state, certain interrupts and manual intervention) are subject to entry interlocks during this interval. Instruction-associated functions also have interlocks which check for such things as the validity of the scanned portion of the instruction register, whether or not the instruction starts on a half-word boundary, whether the instruction is a valid operation, whether an address is to be generated for the instruction (and if so, whether the address adder is available), and where the instruction is to be executed. In conjunction with this last point, should the fixed- or floating-point execution units be involved, availability of operation buffering is checked. Inter-instruction dependencies are the final class of interlocks which can occur during the decoding interval. These arise because of decision predictions which, if proven wrong, require that decoding cease immediately so that recovery can be initiated with a minimum of backup facilities.

Such occurrences as the discovery of a branch wrong guess or a store instruction which may alter the prefetched instruction stream generate these inter-instruction interlocks. Figure 15 illustrates the interlock function. The placement of a store instruction in the instruction stream, in particular, warrants further discussion because it presents a serious time problem in the instruction unit. The dilemma stems both from the concurrency philosophy and from the architectural specification that a store operation may alter the subsequent instruction. Recall that, through the pipeline concept, decoding can occur on successive cycles, with one instruction being decoded at the same time the address for the previous instruction is being generated. Therefore, for a decode which follows a store instruction, a test between the instruction counter and the storage address is required to detect whether or not the subsequent decode is affected by the store. Unless rather extensive recovery hardware is used, the decode, if affected, must be suppressed. However, the assembly line basic time interval is too short to both complete the detection and block the decode. The simplest solution would require a null decode time following each store issue. However, the frequency of store instructions is high enough that the performance degradation would be objectionable. The compromise solution which was adopted reduces the number of decoding delays by utilizing a truncated-address compare. The time requirements

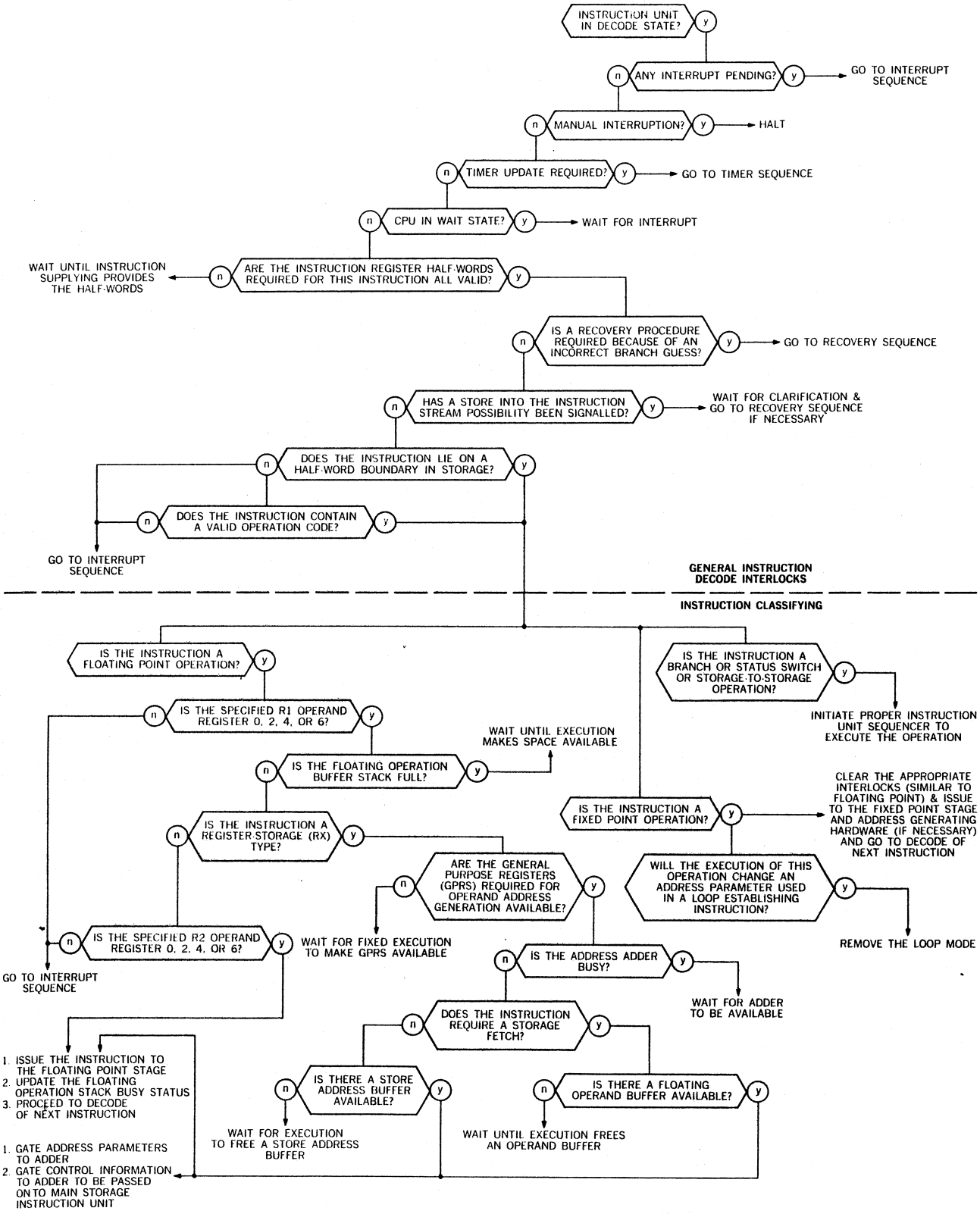


Figure 15 Decision sequence for instruction decoding and instruction issuing.

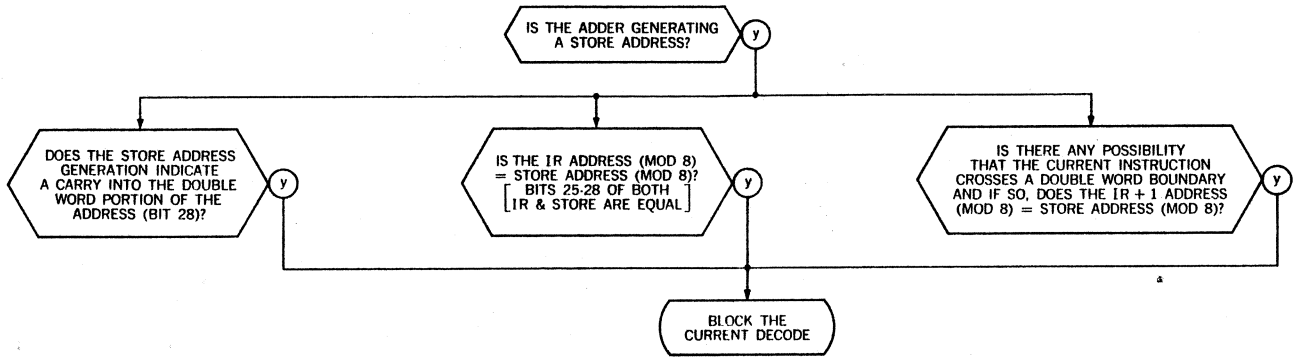


Figure 16 Decode interlock (established following the issue of a store instruction).

prohibit anything more than a compare of the low-order six bits of the storage address currently being generated, using the algorithm illustrated in Fig. 16.

The algorithm attaches relatively little significance to the low-order three adder bits (dealing with byte, half-word and full-word addresses) since the primary performance concern is with stores of double-words. It is seen, for example, that for the full-word case the probability of a carry into the double-word address is approximately 1/4, while for double-word handling it is negligible. The double-word address three-bit compare will occur with 1/8 probability while the word boundary crossover term has a probability of 1/16. (Probability that instruction can cross boundary, 1/2, $\times$ probability that the crossover is into the store-affected-word, 1/8). The two cases thus have the probabilities:

Full word $1/4 + 1/8 + 1/16 = 7/16$, and

Double-word $1/8 + 1/16 = 3/16$.

These figures indicate the likelihood of a decode time-interval delay following the issue of a store instruction. When such a decode delay is encountered, the following cycle is used to complete the test, that is, to check the total address to determine whether an instruction word has in fact been altered. To this effect, the generated storage address is compared with the upper and lower bounds of the instruction array (Fig. 16). A between-the-bounds indication results in a decode halt, a re-fetch of the affected instruction double-word, then resumption of normal processing. This second portion of the interlock is only slightly less critical in timing than the first. Figure 17 illustrates the re-fetch timing sequence. One difficulty with the store interlock is that in blocking the decode, it must inhibit action over a significant portion of the instruction unit. This implies both heavy loading and lengthy wire, each of which seriously hampers circuit performance. It was therefore important that the unit be as small as possible and that the layout of the hardware constantly consider the interlock.

For each instruction, following the clearing of all interlocks, the decode decision determines whether to issue the instruction to an execution unit and initiate address generation, or to retain the instruction for sequencing within the instruction unit. The issuing to an execution unit and the operand fetching for storage-to-register (RX) instructions constitutes a controlled splitting operation; sufficient information is forwarded along both paths to effect a proper execution unit merge. For example, buffer assignment is carried in both paths so that the main storage control element will return the operand to the buffer which will be accessed by the execution unit when it prepares to execute the instruction. With this technique the execution units are isolated from storage and can be designed to treat all operations as involving only registers.

A final decoding function is mentioned here, to exemplify the sort of design considerations and hardware additions that are caused by performance-optimizing techniques. The branch sequencing is optimized so that no address generation is required when a branch which established the loop mode is re-encountered. This is done by saving the location, within the instruction array, of the target. It is possible, even if unlikely, that one of the instructions contained in a loop may alter the parameter originally used to generate the target address which is now being assumed. This possibility, although rare, does require hardware to detect the occurrence and terminate the loop mode. This hardware includes two 4-bit registers, required to preserve the addresses of the general purpose registers (X and B) utilized in the target address generation, and comparators which check these addresses against the sink address (R1) of the fixed-point instructions. Detection of a compare and termination of loop mode are necessary during the decoding interval to ensure that subsequent branch sequencing will be correct.

The address-generating time interval provides for the combining of proper address parameters and for the forwarding of the associated operation (fetch or store) control to the main storage control element through an interface register. A major concern, associated with the address parameters, was to decide where the physical location of the general purpose registers should be. This concern arises since the fixed-point execution unit, as well as the instruction unit, makes demands on the GPR's, while the packaging split will cause the registers to be relatively far from one of the units. It was decided to place them in the execution unit since, first, execution tends to change the registers while address generation merely examines their contents, and secondly, it was desired that a fixed-point execution unit be able to iteratively use any particular register on successive time intervals. In order to circumvent the resulting time delay (long wire separation) between the general purpose registers and the address adder, each register is fed via "hot" lines to the instruction unit. The gating of a particular GPR to the adder can thereby be implemented locally within the instruction unit, and no transmission delay is incurred unless the register contents have just been changed.

Placing the GPR's outside the instruction unit creates a delay of two basic time intervals before a change initiated by the instruction unit is reflected at the address parameter inputs from the GPRs. This delay is particularly evident when it is realized that the address generated immediately following such a GPR change generally requires the contents of the affected register as a parameter. For example, branch on index, branch on count, branch and link, and load address are instruction unit operations which change the contents of a GPR. Further, in loop situations the target of the branch frequently uses the changed register as an index quantity in its address. Performance demands led to the incorporating of controls which recognize the above situation and effect a by-pass

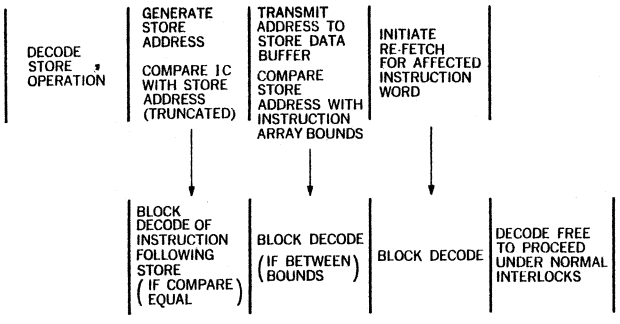


Figure 17 Effect of the decode interlock on pre-fetched instructions.

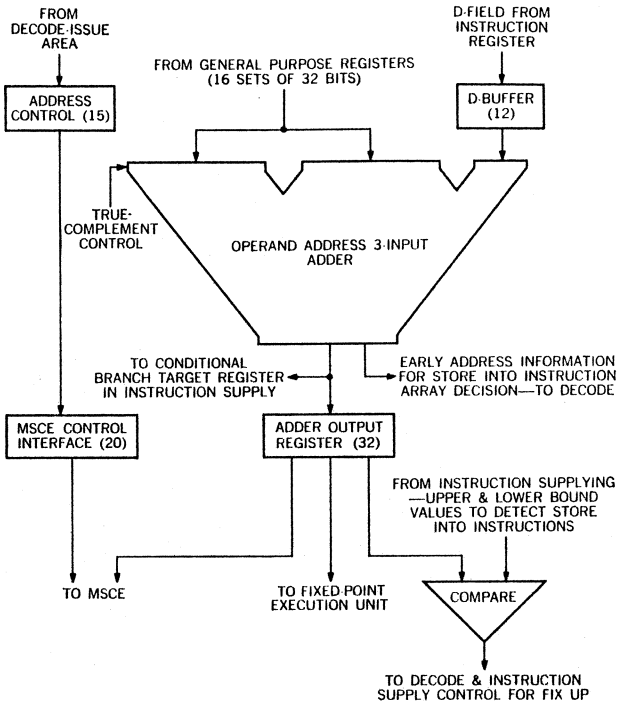


Figure 18 Data flow for address generation.

of the GPR. This entails substituting the content of the adder output register (which contains the new GPR data) for the content of the affected GPR. One performance cycle was saved by this technique.

In addition to address generation, the address adder serves to accomplish branch decision arithmetic, loop mode testing, and instruction counter value generation for various situations. In order to perform all of these functions, it was required that the adder have two 32-bit inputs and one input of 12 bits. One of the 32-bit inputs is complementable and a variety of fixed, single-bit inputs is provided for miscellaneous sequences. The data path is illustrated in Fig. 18.

Status switching and input/output

The philosophy associated with status switching instructions is primarily one of design expediency. Basic existing hardware paths are exercised wherever possible, and an attempt is made to adhere to the architectural interrupt specifications. When status switching instructions are encountered in conditional mode the instruction unit is halted and no action is taken until the condition is cleared.

The supervisor call (SVC) instruction is treated by the interrupt hardware as a precise interrupt. The same new status word pre-fetch philosophy is utilized in the load program status word (LPSW) operation.

One difficulty encountered in conjunction with the start-up fetching of instructions following a status switch (or interrupt) is that a new storage protect key* is likely to obtain. Consequently, a period exists during which two protect keys are active, the first for previously delayed, still outstanding accesses associated with the current execution clean-up, and the second for the fetching of instructions. This situation is handled by sending both keys to the main storage control element and attaching proper control information to the instruction fetches.

The set program mask (SPM) implementation has a minor optimization: Whenever the new mask equals the current mask, the instruction completes immediately. Otherwise an execution clean-up is effected before setting the new mask to make certain that outstanding operations are executed in the proper mask environment.

I/O instructions, and I/O interrupts, require a wait for channel communications. The independent channel and CPU paths to storage demand that the CPU be finished setting up the I/O controls in storage before the channel can be notified to proceed. Once notified, the channel must interrogate the instruction-addressed device prior to setting the condition code in the CPU. This is accomplished by lower-speed circuitry and involves units some distance away; consequently, I/O initiation times are of the order of 5–10 microseconds.

* The storage protect key is contained in the program status word (PSW). It is a tag which accompanies all storage requests, and from it the storage can determine when a protect violation occurs.

Acknowledgments

The authors wish to thank Mr. R. J. Litwiller for his interest, suggestions and design effort, and Messrs. J. G. Adler, R. N. Gustafson, P. N. Prentice and C. Zeitler, Jr. for their contributions to the design of the instruction unit.

References

1. G. M. Amdahl, G. A. Blaauw and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal* 8, 87 (1964).
2. W. Buchholz et al., *Planning a Computer System*, McGraw-Hill Publishing Co., Inc., New York, 1962.
3. R. J. Litwiller and J. G. Adler, private communication.
4. S. F. Anderson et al., "The IBM System/360 Model 91 Floating Point Execution Unit," *IBM Journal* 11, 34 (1967) (this issue).
5. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal* 11, 25 (1967) (this issue).
6. L. J. Boland, et al., "IBM System/360 Model 91 Storage System," *IBM Journal* 11, 54 (1967) (this issue).
7. M. J. Flynn and P. R. Low, "The IBM System/360 Model 91: Some Remarks on System Development" *IBM Journal* 11, 2 (1967) (this issue).

Received September 21, 1965.

R. M. Tomasulo

An Efficient Algorithm for Exploiting Multiple Arithmetic Units

Abstract: This paper describes the methods employed in the floating-point area of the System/360 Model 91 to exploit the existence of multiple execution units. Basic to these techniques is a simple common data busing and register tagging scheme which permits simultaneous execution of independent instructions while preserving the essential precedences inherent in the instruction stream. The common data bus improves performance by efficiently utilizing the execution units without requiring specially optimized code. Instead, the hardware, by 'looking ahead' about eight instructions, automatically optimizes the program execution on a local basis.

The application of these techniques is not limited to floating-point arithmetic or System/360 architecture. It may be used in almost any computer having multiple execution units and one or more 'accumulators.' Both of the execution units, as well as the associated storage buffers, multiple accumulators and input/output buses, are extensively checked.

Introduction

After storage access time has been satisfactorily reduced through the use of buffering and overlap techniques, even after the instruction unit has been pipelined to operate at a rate approaching one instruction per cycle,[1] there remains the need to optimize the actual performance of arithmetic operations, especially floating-point. Two familar problems confront the designer in his attempt to balance execution with issuing. First, individual operations are not fast enough* to allow simple serial execution. Second, it is difficult to achieve the fastest execution times in a universal execution unit. In other words, circuitry designed to do both multiply and add will do neither as fast as two units each limited to one kind of instruction.

The first step toward surmounting these obstacles has been presented,[2] i.e., the division of the execution function into two independent parts, a fixed-point execution area and a floating-point execution area. While this relieves the physical constraint and makes concurrent execution possible, there is another consideration. In order to secure a performance increase the program must contain an intimate mixture of fixed-point and floating-point instructions. Obviously, it is not always feasible for the programmer to arrange this and, indeed, many of the programs of greatest interest to the user consist almost wholly of floating-point instructions. The subject of this paper, then, is the method used to achieve concurrent execution of floating-point instructions in the IBM System/360 Model 91. Obviously, one begins with multiple execution units, in this case an adder and a multiplier/divider.[1]

It might appear that achieving the concurrent operation of these two units does not differ substantially from the attainment of fixed-floating overlap. However, in the latter case the architecture limits each of the instruction classes to its own set of accumulators and this guarantees independence.* In the former case there is only one set of accumulators, which implies program-specified sequences of dependent operations. Now it is no longer simply a matter of classifying each instruction as fixed-point or floating-point, a classification which is independent of previous instructions. Rather, it is a question of determining each instruction's relationship with all previous, incompleted instructions. Simply stated, the objective must be to preserve essential precedences while allowing the greatest possible overlap of independent operations.

This objective is achieved in the Model 91 through a scheme called the common data bus (CDB). It makes possible maximum concurrency with minimal effort (usually none) by the programmer or, more importantly, by the compiler. At the same time, the hardware required is small and logically simple. The CDB can function with any number of accumulators and any number of execution units. In short, it provides a hardware algorithm for the automatic, efficient exploitation of multiple execution units.

* During the planning phase, floating-point multiply was taken to be six cycles, divide as eighteen cycles and add as two cycles. A subsequent paper[2] explains how times of 3, 12, and 2 were actually achieved. This permitted the use of only one, instead of two, multipliers and one adder, pipelined to start an add cycle.

* Such dependencies as exist are handled by the store-fetch sequencing of the storage bus and the condition code control described in the following paper.[2]

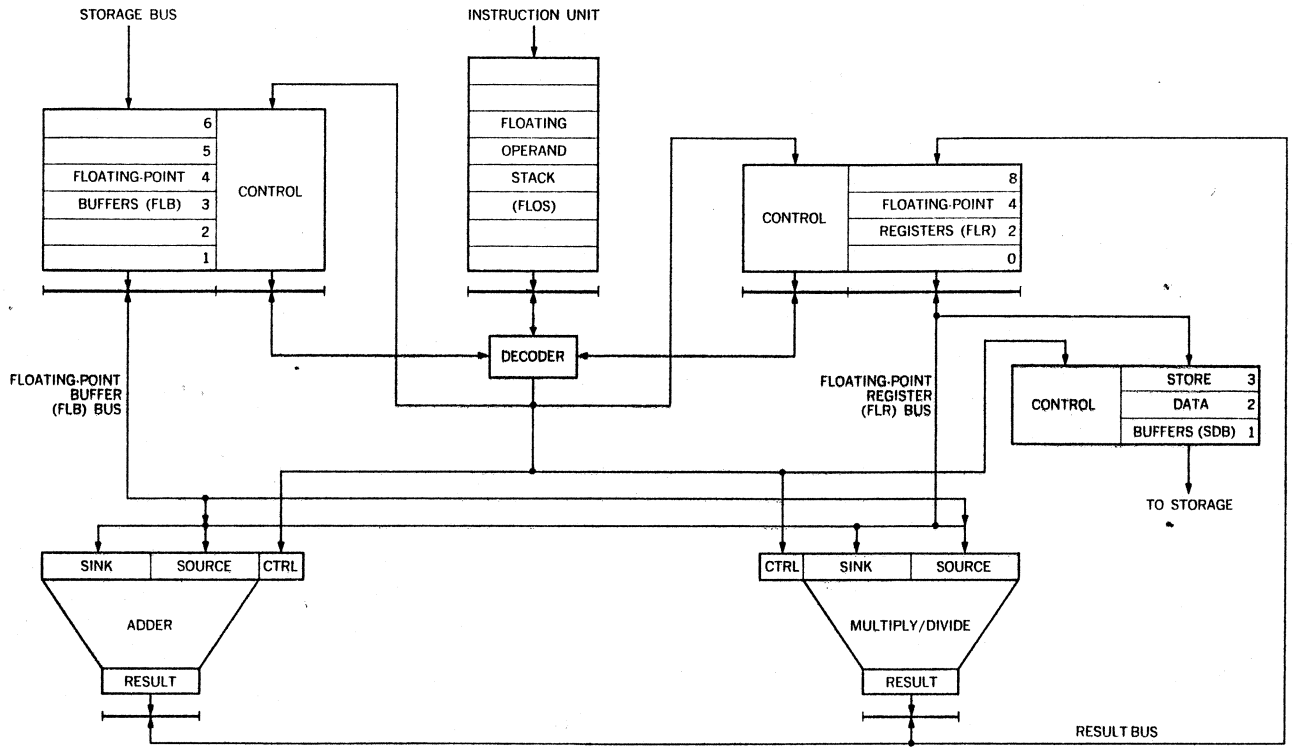


Figure 1 Data registers and transfer paths without CDB.

The next section of this paper will discuss the physical framework of registers, data paths and execution circuitry which is implied by the architecture and the overall CPU structure presented in a previous paper.[1] Within this framework one can subsequently discuss the problem of precedence, some possible solutions, and the selected solution, the CDB. In conclusion will be a summary of the results obtained.

Definitions and data paths

While the reader is assumed to be familiar with System/360 architecture and mnemonics, the terminology as modified by the context of the Model 91 organization will be reviewed here. The instruction unit, in preparing instructions for the floating-point operation stack (FLOS), maps both storage-to-register and register-to-register instructions into a pseudo-register-to-register format. In this format R1 is always one of the four floating-point registers (FLR) defined by the architecture. It is usually the *sink* of the instruction, i.e., it is the FLR whose contents are set equal to the result of the operation. Store operations are the sole exception* wherein R1 specifies the *source* of the operand to be placed in storage. A word in storage is really the sink of a store. (R1 and R2 refer to fields as defined by System/360 architecture.)

In the pseudo-register-to-register format "seen" by the FLOS the R2 field can have three different meanings. It can be an FLR as in a normal register-to-register instruction. If the program contains a storage-to-register instruction, the R2 field designates the floating-point buffer (FLB) assigned by the instruction unit to receive the storage operand. Finally, R2 can designate a store data buffer (SDB) assigned by the instruction unit to store instructions. In the first two cases R2 is the *source* of an operand; in the last case it is a sink. Thus, the instruction unit maps all of storage into the 6 floating-point buffers and 3 store data buffers so that the FLOS sees only pseudo-register-to-register operations.

The distinction between source and sink will become quite important during the discussion of precedence and should be fixed firmly in mind. All of the instructions (except store and compare) have the following form:

R1	op	R2 ⟶ R1
Register		Register Register
		or
		buffer
source		source sink

* Compares do not, of course, alter the contents of R1.

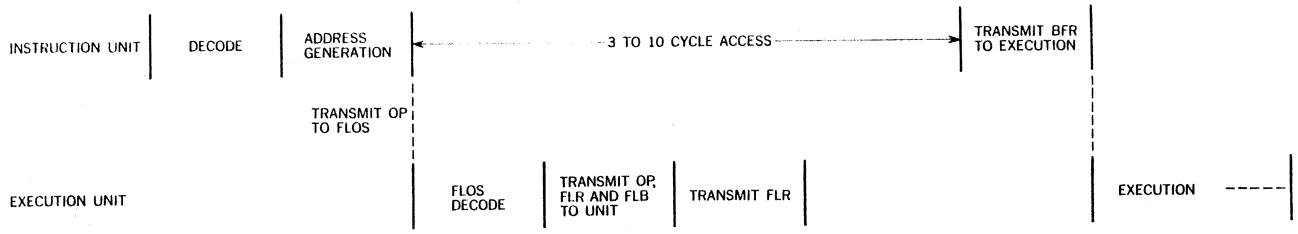


Figure 2 Timing relationship between instruction unit and FLOS decode for the processing of one instruction.

For example, the instruction AD0, 2 means "place the double-precision sum of registers 0 and 2 in register 0," i.e., R0 + R2 → R0. Note that R1 is really both a source and a sink.* Nevertheless, it will be called the sink and R2 the source in all subsequent discussion.

This definition of operations and the machine organization taken together imply a set of data registers with transfer paths among them. These are shown in Fig. 1. The major sets of registers (FLR's, FLB's, FLOS and SDB's) have already been discussed, both above and in a preceding paper.[1] Two additional registers, one sink and one source, are shown feeding each execution circuit. Initially these registers were considered to be the internal working registers required by the execution circuits and put to multiple use in a way to be described below. Later, their function was generalized under the reservation station concept and they were dissociated from their "working" function.

In actually designing a machine the data paths evolve as the design progresses. Here, however, a complete, first-pass data path will be shown to facilitate discussion. To illustrate the operation let us consider, in turn, four kinds of instructions—load of a register from storage, storage-to-register arithmetic, register-to-register arithmetic, and store. Let us first see how each can be accomplished *in vacuo*; then what difficulties arise when each is embedded in the context of a program. For simplicity double-precision (64-bit operands) will be used throughout.

Figure 2 shows the timing relationship between the instruction unit's handling of an instruction and its processing by the FLOS decode. When the FLOS decodes a load, the buffer which will receive the operand has not yet been loaded from storage.† Rather than holding the decode until the operand arrives, the FLOS sets control bits associated with the buffer which cause its content to be transmitted to the adder when it "goes full." The adder receives control information which causes it to send data to floating-point register R1, when its source register is set full by the buffer.

If the instruction is a storage-to-register arithmetic function, the storage operand is handled as in load (control bits cause it to be forwarded to the proper unit) but the floating-point register, along with the operation, is sent by the decoder to the appropriate unit. After receiving the buffer the unit will execute the operation and send the result to register R1.

In register-to-register arithmetic instructions two floating point registers are transmitted on successive cycles to the appropriate execution unit.

Stores are handled like storage-to-register arithmetic functions, except that the content of the floating-point register is sent to a store data buffer rather than to an execution unit.

Thus far, the handling of one instruction at a time has proven rather straightforward. Now consider the following "program":

Example 1

LD F0 FLB1 LOAD register F0 from buffer 1

MD F0 FLB2 MULTIPLY register F0 by buffer 2

The load can be handled as before, but what about the multiply? Certainly F0 and FLB2 cannot be sent to the multiplier as in the case of the isolated multiply, since FLB1 has not yet been set into F0.* This sequence illustrates the cardinal precedence principle: No floating-point register may participate in an operation if it is the sink of another, incompleted instruction. That is, a register cannot be used until its contents reflect the result of the most recent operation to use that register as its sink.

The design presented thus far has not incorporated any mechanism for dealing with this situation. Three functions must be required of any such mechanism:

(1) It must recognize the existence of a dependency.

* This economy of specification compounds the difficulties of achieving concurrency while preserving precedence, as will be seen later.

† A FULL/EMPTY control bit indicates this. The bit is set FULL by the Main Storage Control Element and EMPTY when the buffer is used. LOAD uses the adder in order to minimize the buffer outgates and the FLR ingates.

* Note that the program calls for the product of FLB1 and FLB2 to be placed in F0. This hints at the CDB concept.

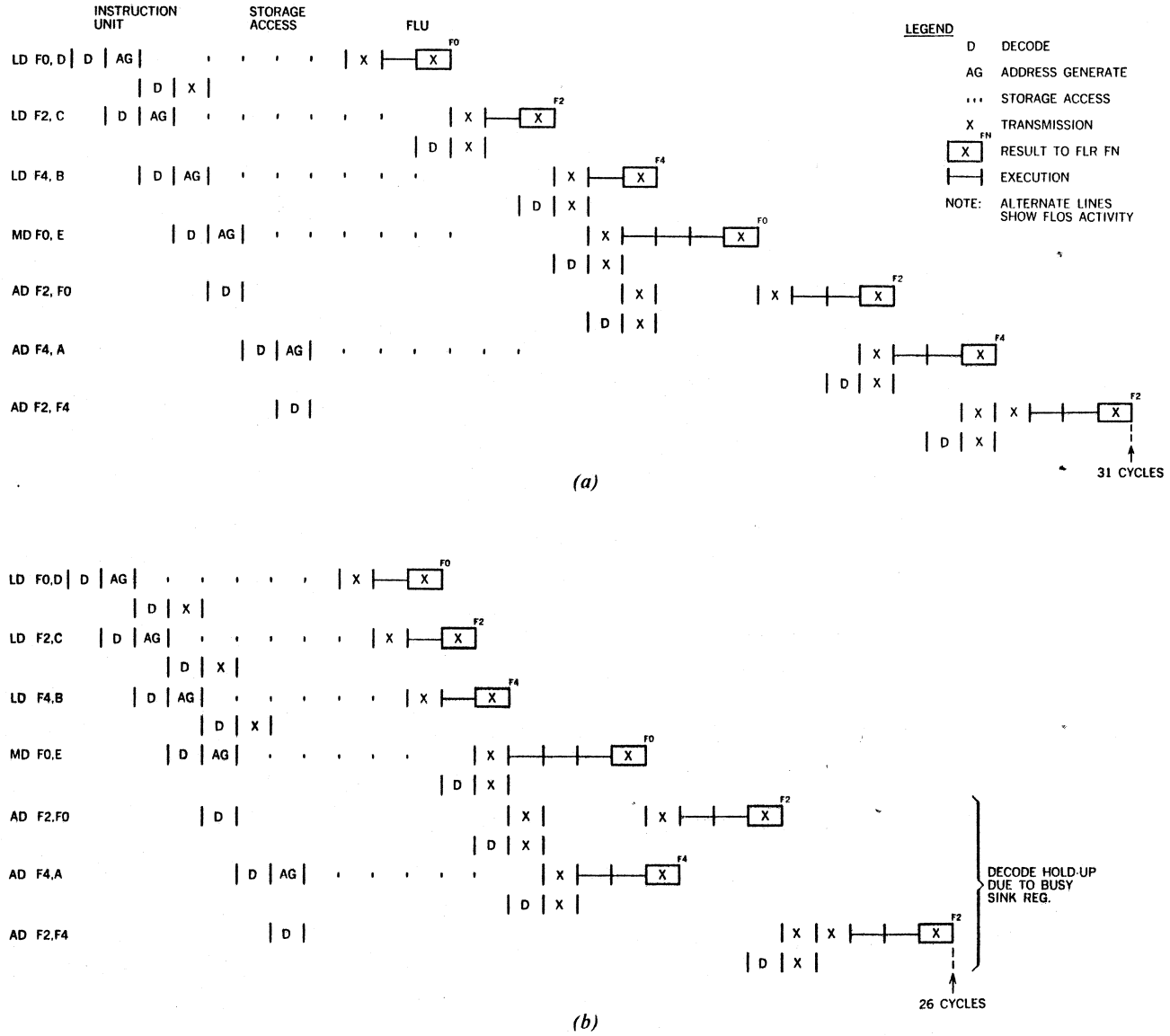


Figure 3 Timing for the instruction sequence required to perform the function A + B + C + D * E : (a) without reservation stations, (b) with reservation stations included in the register set.

(2) It must cause the correct sequencing of the dependent instructions.

(3) It must distinguish between the given sequence and such sequences as

```
LD   F0,  FLB1
MD   F2,  FLB2
```

Here it must allow the independent MD to proceed regardless of the disposition of the LD.

The first two requirements are necessary to preserve the logical integrity of the program; the third is necessary to meet the performance goal. The next section will present several alternatives for accomplishing these objectives.

Preservation of precedence

Perhaps the simplest scheme for preserving precedence is as follows. A "busy" bit is associated with each of the four floating-point registers. This bit is set when the FLOS decode issues an instruction designating the register as a sink; it is reset when the executing unit returns the result to the register. No instruction can be issued by the FLOS if the busy bit of its sink is on. If the source of a register-to-register instruction has its busy bit on, the FLOS sets

control bits associated with the source register. When a result is entered into the register, these control bits cause the register to be sent via the FLR bus to the unit waiting for it as a source.

This scheme easily meets the first two requirements. The third is met with the help of the programmer; he must use different registers to achieve overlap. For example, the expression A + B + C + D * E can be programmed as follows:

Example 2

```
LD   F0,  D      F0 = D
LD   F2,  C      F2 = C
LD   F4,  B      F4 = B
MD   F0,  E      F0 = D * E
AD   F2,  F0     F2 = C + D * E
AD   F4,  A      F4 = A + B
AD   F2,  F4     F2 = A + B + C + D * E
```

The busy bit scheme should allow the second add and the multiply to be executed simultaneously (really, in any order) since they use different sinks. Unfortunately, the timing chart of Fig. 3a shows not only that the expected overlap does not occur but also that many cycles are lost to transmission time. The overlap fails to materialize because the first add uses the result of the multiply, and the adder must wait for that result. Cycles are lost to control because so many of the instructions use the adder. The FLOS cannot decode an instruction unless a unit is available to execute it. When an assigned unit finishes execution, it takes one cycle to transmit the fact to the FLOS so that it can decode a waiting instruction. Similarly, when the FLOS is held up because of a busy sink register, it cannot begin to decode until the result has been entered into the register.

One solution that could be considered is the addition of one or more adders. If this were done and some programs timed, however, it would become apparent that the execution circuitry would be in use only a small part of the time. Most of the lost time would occur while the adder waited for operands which are the result of previous instructions. What is required is a device to collect operands (and control information) and then engage the execution circuitry when all conditions are satisfied. But this is precisely the function of the sink and source registers in Fig. 1. Therefore, the better solution is to associate more than one set of registers (control, sink, source) with each execution unit. Each such set is called a *reservation station.** Now instruction issuing depends on the availability of the appropriate kind of reservation station. In the Model 91 there are three add and two multiply/divide reservation stations. For simplicity they are treated as if they were actual units. Thus, in the future, we will speak of Adder 1 (A1), Adder 2 (A2), etc., and M/D 1 and M/D 2.

Figure 3b shows the effect of the addition of reservation stations on the problem running time: five cycles have been eliminated. Note that the second AD now overlaps the MD and actually executes before the first AD. While the speed increase is gratifying and the busy bit method easy to implement, there remains a dependence on the programmer. Note that the expression could have been coded this way:

Example 3a

```
LD   F0,  E
MD   F0,  D
AD   F0,  C
AD   F0,  B
AD   F0,  A
```

Now overlap is impossible and the program will run six cycles longer despite having two fewer instructions. Suppose however, that this program is part of a loop, as below:

Example 3b

```
LOOP 1   LD   F0,  Ei
         MD   F0,  Di
         AD   F0,  Ci
         AD   F0,  Bi
         AD   F0,  Ai
         STD  F0,  Fi
         BXH i, -1, 0, LOOP 1 (decrease i by 1,
         branch if i > 0)
LOOP 2   LD   F0,  Ei
         LD   F2,  Ei + 1
         MD   F0,  Di
         MD   F2,  Di + 1
         AD   F0,  Ci
         AD   F2,  Ci + 1
         AD   F0,  Bi
         AD   F2,  Bi + 1
         AD   F0,  Ai
         AD   F2,  Ai + 1
         STD  F0,  Fi
         STD  F2,  Fi + 1
         BXH i, -2, 0, LOOP 2
```

Iteration $n + 1$ of LOOP 1 will appear to the FLOS to depend on iteration n, since the instructions in both iterations have the same sink. But it is clear that the two iterations are, in fact, independent. This example illustrates a second way in which two instruction sequences can be independent. The first way, of course, is for the two strings to have different sink registers. The second way is for the second string to begin with a load. By its definition a

* The fetch and store buffers can be considered as specialized, one-operand reservation stations. Previous systems, such as the IBM 7030, have in effect employed one "reservation station" ahead of each execution unit. The extension to several reservation stations adds to the effectiveness of the execution hardware.

load launches a new, independent string because it instructs the computer to destroy the previous contents of the specified register. Unfortunately, the busy bit scheme does not recognize this possibility. If overlap is to be achieved with this scheme, the programmer must write LOOP 2. (This technique is called *doubling* or *unravelling*. It requires twice as much storage but it runs faster by enabling two iterations to be executed simultaneously.)

Attempts were made to improve the busy bit scheme so as to handle this case. The most tempting approach is the expansion of the bit into a counter. This would appear to allow more than one instruction with a given sink to be issued. As each is issued, the FLOS increments the counter; as each is executed the counter is decremented. However, major difficulty is caused by the fact that storage operands do not return in sequence. This can cause the result of instruction $n + 1$ to be placed in a register before that of n. When n completes, it erroneously destroys the register contents.

Some of the other proposals considered would, if implemented, have been of such logical complexity as to jeopardize the achievement of a fast cycle.

The Common Data Bus

The preceding sections were intended to portray the difficulties of achieving concurrency among floating-point instructions and to show some of the steps in the evolution of a design to overcome them. It is clear, in retrospect, that the previous algorithms failed for lack of a way to uniquely identify each instruction and to use this information to sequence execution and set results into the floating-point registers. As far as action by the FLOS is concerned, the only thing unique to a particular instruction is the unit which will execute it. This, then, must form the basis of the common data bus (CDB).

Figure 4 shows the data paths required for operation of the CDB.* When Fig. 4 is compared with Fig. 1 the following changes, in addition to the reservation stations, are evident: Another output port has been added to the buffers. This port has been combined with the results from the adder and multiplier/divider; the combination is the CDB. The CDB now goes not only to the registers but also to the sink and source registers of all reservation stations, including the store data buffers but excluding the floating-point buffers. This data path will enable loads to be executed without the adder and will make the result of any operation available to all units without first going through a floating-point register.

Note that the CDB is fed by all units that can alter a register and that it feeds all units which can have a register as an operand. The control part of the CDB enumerates the units which feed the CDB. Thus the floating-point buffers 1 through 6 are assigned the numbers 1 through 6; the three adders (actually reservation stations) are numbered 10 through 12; the two multiplier/dividers are 8 and 9. Since there are eleven contributors to the CDB, a four-bit binary number suffices to enumerate them. This number is called a *tag*. A tag is associated with each of the four floating-point registers (in addition to the busy bit*), with both the source and sink registers of each of the five reservation stations and with each of the three Store Data Buffers. Thus a total of 17 four-bit tag registers has been added, as shown in Fig. 4.

Tags also appear in another context. A tag is generated by the CDB priority controls to identify the unit whose result will next appear on the CDB. Its use will be made clear shortly.

Operation of this complex is as follows. In decoding each instruction the FLOS checks the busy bit of each of the specified floating-point registers. If that bit is zero, the content of the register(s) may be sent to the selected unit via the FLR bus, just as before. Upon issuing the instruction, which requires only that a unit be available to execute it, the FLOS not only sets the busy bit of the sink register but also sets its tag to the designation of the selected unit. The source register control bits remain unchanged. As an example, take the instruction, AD F0, FLB1. After issuing this instruction to Adder 1 the control bits of F0 would be:

BB	TAG
1	1010 (A1)

So far the only change from previous methods is the setting of the tag. The significant difference occurs when the FLOS finds the busy bit on at decode time. Previously, this caused a suspension of decoding until the bit went off. Now the FLOS will issue the instruction and update the tag. In so doing it will not transmit the register contents to the selected unit but it will transmit the "old" tag. For example, suppose the previous AD was followed by a second AD. At the end of the decode of this second AD, F0's control bits would be:

BB	TAG
1	1011 (A2)

One cycle later the sink tag of the A2 reservation station would be 1010, i.e., the same as A1, the unit whose result will be required by A2.

Let us look ahead temporarily to the execution of the first AD. Some time after the start of execution but before the end,† A1 will request the CDB. Since the CDB is fed by many sources, its time-sharing is controlled by a central

* The FLB and FLR busses are retained for performance reasons. Everything could be done by a slight extension of the CDB but time would be lost due to conflicts over the common facility.

* The busy bit is no longer necessary since its function can be performed by use of an unassigned tag number. However, it is convenient to retain it.

† Since the required lead time is two cycles, the request is made at the start of execution for an add-type instruction.

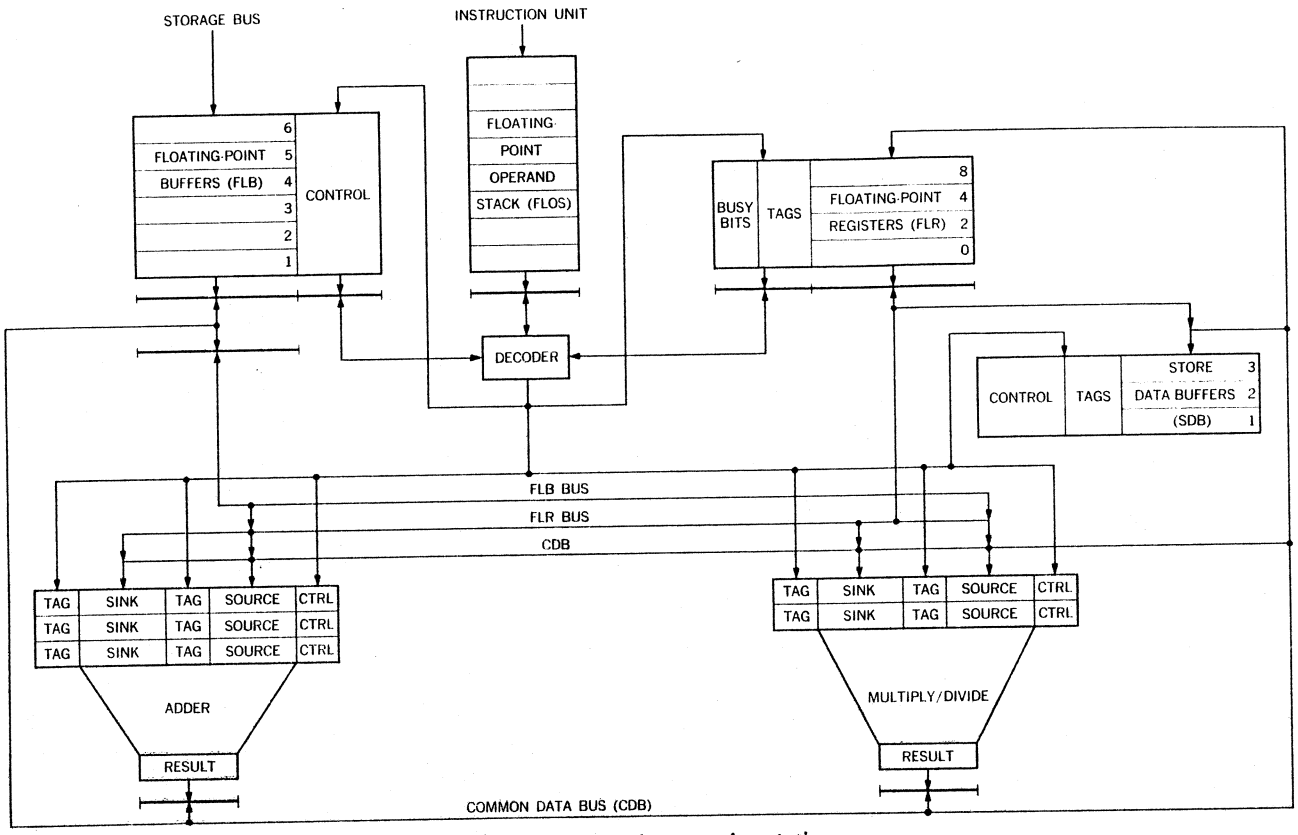


Figure 4 Data registers and transfer paths, including CDB and reservation stations.

priority circuit. If the CDB is free, the priority control signals the requesting adder, A1, to outgate its result *and* it broadcasts the tag of the requestor (1010 in this case) to all reservation stations. Each active reservation station (selected but awaiting a register operand) compares its sink and source tags to the CDB tag. If they match, the reservation station ingates the data from the CDB. In a similar manner, the CDB tag is compared with the tag of each "*busy* floating-point register. All busy registers with matching tags ingate from the CDB and reset their busy bits.

Two steps toward the goal of preserving precedence have been accomplished by the foregoing. First, the second AD cannot start until the first AD finishes because it cannot receive both its operands until the result of the first AD appears on the CDB. Secondly, the result of the first AD cannot change register F0 once the second AD is issued, since the tag in F0 will not match A1. These are precisely the desired effects.

Before proceeding with more detailed considerations let us recapitulate the essence of the method. The floating-point register tag identifies the last unit whose result is destined for the register. When an instruction is issued that requires a busy register the tag is sent to the selected unit in place of the register contents. The unit continuously compares this tag with that generated by the CDB priority control. When a match is detected, the unit ingates from the CDB. The unit begins executing as soon as it has both operands. It may receive one or both operands from either the CDB or the FLR bus; the source operand for storage-to-register instructions is transmitted via the FLB bus.

As each instruction is issued the existing tag(s) is (are) transmitted to the selected unit and then the sink tag is updated. By passing tags around in this fashion, all operations having the same sink are correctly sequenced while other operations are allowed to proceed independently. Finally, the floating-point register tag controls the changing of the register itself, thereby ensuring that only the most recent instruction will change the register. This has the interesting consequence that a loop of the following kind:

Example 5

```
LOOP  LD    F0,  Ai
      AD    F0,  Bi
      STD   F0,  Ci   STORE
      BXH    i,  −1,  0,  LOOP
```

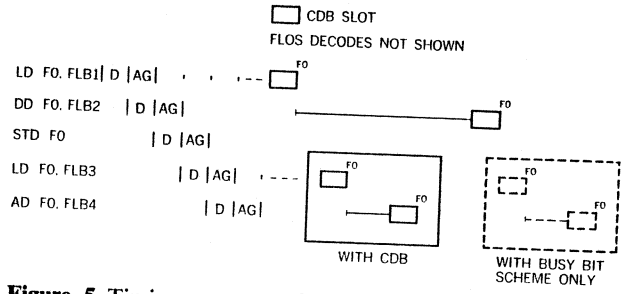


Figure 5 Timing sequence for Example 6, showing effect of CDB.

may execute indefinitely without any change in the contents of F0. Under normal conditions only the final iteration will place its result in F0.

As mentioned previously, there are two ways of starting an independent instruction string. The first is to specify a different sink register and the second is to load a register. The CDB handles the former in essentially the same way as the busy bit scheme. The load, which had been a difficult problem previously, is now very simple. Regardless of the register tag or busy bit, a load turns the busy bit on and sets the tag equal to the floating-point buffer which the instruction unit had assigned to the load. This causes subsequent instructions to sequence on the buffer rather than on whatever unit may have identified the register as its sink prior to the load. The buffer controls are set to request the CDB when the storage operand arrives. The following example and Fig. 5 show this clearly.

Example 6

```
LD      F0,  FLB1
DD      F0,  FLB2      DIVIDE
STD     F0,  A
LD      F0,  FLB3
AD      F0,  FLB4
```

Note that the add finishes before the divide. The dashed line portion of Fig. 5 shows what would happen if the busy bit scheme alone were used. Figure 6 displays the sequences followed under the two schemes. This figure graphically illustrates the bottleneck caused by using a single sink register with a busy bit scheme. Because all data must pass through this register, the program is reduced to strictly sequential execution, steps 1 through 7. With the CDB, on the other hand, the sink register hardly appears and the program is broken into two independent, concurrent sequences. This facility of the CDB obviates the need for loop doubling.

The CDB makes it possible to execute some instructions in, effectively, no time at all. In the above example the store took place during the CDB cycle following the divide. In a similar fashion a register-to-register load of a busy register is accomplished by moving the tag of the source floating-point register to the tag of the sink floating-point register. For example, in the sequence

```
AD      F0,  FLB1
LDR     F2,  F0      move F0 to F2
```

the tag of F0 will be 1010 (A1) at the time the LDR is decoded. The decoder simply sets F2's tag to 1010. Now, when the result of the AD appears on the CDB both F0 and F2 will ingate since the CDB tag of 1010 will match the tag of each register. Thus, no unit or extra time was required for the execution of the LDR.

A number of details have been omitted from this discussion in order to clarify the concept, but really only two are of operational significance. First, every unit must request the CDB two cycles before it finishes execution. (These two cycles are required for propagation of the request to the CDB controls, the establishment of priority among competing units, and propagation of a "select" signal to the chosen unit.) This limits the execution time of any instruction to a two-cycle minimum. (Of course, the faster the execution the less the need for, or gain from, concurrency.) It also adds one* cycle to the access time for loads. Because of buffering and overlap, this does not usually cause an increase in problem running time.

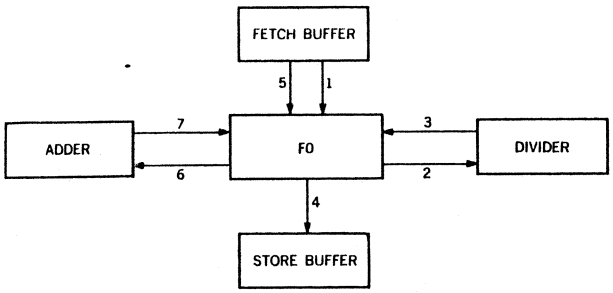
The second point is concerned with mixed precision. Because the architectural definition causes the low-order part of an FLR to be preserved during single-precision operation, an error can occur in the following kind of program:

```
LD      F0,  FLB1
AD      F0,  FLB2
AE      F0,  FLB3
```

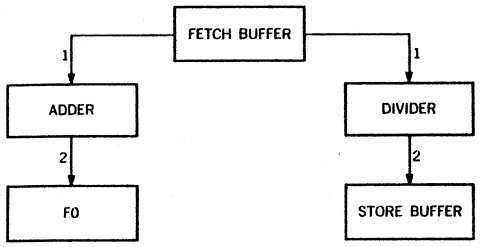
Since only the last instruction, which is single-precision, will change F0, the low order result of the double-precision AD will be lost. This is handled by associating a bit with each register to indicate whether a particular register is the sink of an outstanding single- or double-precision instruction. If this bit does not match the "length" of the instruction being decoded, the decode is suspended until the busy bit goes off. While this stratagem† solves the logic problem, it does so at the expense of performance. Unfortunately, no way has been found to avoid this. Note, however, that all-single- or all-double-precision programs run at the maximum possible speed. It is only the interface between single- and double-precision to the *same* sink register that suffers delay.

* It does not add two cycles since storage gives one cycle prenotification of the arrival of data.

† Further complications arise from the fact that single-precision multiply produces a double-precision product. This is handled separately but with the same time penalty as above.



(a)



(b)

Figure 6 Functional sequence for Example 6 (a) with busy bit controls only, (b) with CDB.

Conclusions

Two concepts of some significance to the design of high-performance computers have been presented. The first, reservation stations, is simply an expeditious method of buffering, in an environment where the transmission time between units is of consequence. Because of the disparity between storage access and circuit speeds and because of dependencies between successive operations, it is observed (given multiple execution units) that each unit spends much of its time waiting for operands. In effect, the reservation stations do the waiting for operands while the execution circuitry is free to be engaged by whichever reservation station fills first.

The second, and more important, innovation, the CDB, utilizes the reservation stations and a simple tagging scheme to preserve precedence while encouraging concurrency. In conjunction with the various kinds of buffering in the CPU, the CDB helps render the Model 91 less sensitive to programming. It should be evident, however, that the programmer still exercises substantial control over how much concurrency will occur. The two different programs for doing A + B + C + D * E illustrate this clearly.

It might appear that the CDB adds one cycle to the execution time of each operation, but in fact it does not. In practice only 30 nsec of the 60-nsec CDB interval are required to perform all of the CDB functions. The remaining time could, in this case, be used by the execution unit to achieve a shorter effective cycle. For example, if an add requires 120 nsec, then add plus the CDB time required is 150 nsec. Therefore, as far as the add is concerned, the machine cycle could be 50 nsec. Besides, even without the CDB, a similar amount of time would be required to transmit results both to the floating-point registers and back as an input to the unit generating the result.

The following program, a typical partial differential equation inner loop, illustrates the possible performance increase.

```
LOOP    MD      F0,   Ai
        AD      F0,   Bi
        LD      F2,   Ci
        SDR     F2,   F0
        MDR     F2,   F6
        AD2     F2,   Ci
        STD     F2,   Ci
        BXH     i,  -1,  0,  LOOP
```

Without the CDB one iteration of the loop would use 17 cycles, allowing 4 per MD, 3 per AD and nothing for LD or STD. With the CDB one iteration requires 11 cycles. For this kind of code the CDB improves performance by about one-third.

Acknowledgments

The author wishes to acknowledge the contributions of Messrs. D. W. Anderson and D. M. Powers, who extended the original concept, and Mr. W. D. Silkman, who implemented all of the central control logic discussed in the paper.

References

1. D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, "The System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal* 11, 8 (1967) (this issue).
2. S. F. Anderson, J. Earle, R. E. Goldschmidt and D. M. Powers, "The System/360 Model 91 Floating-Point Execution Unit," *IBM Journal* 11, 34 (1967) (this issue).

Received September 16, 1965.

The central processing complex of the System/360 Model 195 is made up of seven stand-alone units: a CPU, three CPU power supply units, a power distribution unit, a coolant distribution unit, and a system console (Figure 2). (A motor-generator set must be ordered separately and may be located in a remote area.)

CENTRAL PROCESSING UNIT

Functionally, the central processing unit consists of these major logical elements: instruction processor, fixed-point/variable-field-length (VFL)/decimal execution element, floating-point execution element, high-speed buffer storage, storage control unit, and processor storage (Figure 3). The instruction processor and the two execution elements make up the central processing element (CPE), also called the processor.

The instruction processor is the major coordinating element in the Model 195. For each instruction, it determines what must be done and issues the operation to the proper execution unit. Branching, status switching, and I/O instructions are handled by the instruction processor; other instructions are issued by the instruction processor to other processor elements for completion.

The fixed-point/VFL/decimal execution element contains the general registers, which are used also by the instruction processor. Functionally, this element operates as an independent stored-program computer; it has its own instruction stream, its own execution circuitry, and a set of operand buffers.

The floating-point execution element also operates as an independent computer. Although most of the floating-point instructions require more than one cycle of execution time, this element is capable of sustaining an execution rate of up to one instruction per cycle.

The storage control unit handles all fetching and storing of data for the CPE. It is designed to minimize conflicting requests for storage and to make the most efficient use of storage.

The high-speed buffer storage provides the principal means of reducing average access time to main storage. The most current blocks (a block is eight doublewords) of storage are maintained in the buffer storage, the operation of which is not obvious to the programmer. The first processor fetch to a block, for a storage address within that block, accesses the addressed location and initiates a transfer of the block into the buffer storage.

Subsequent accesses to that block can then be made directly from the buffer storage. Processor stores are made to both the buffer (if appropriate) and to processor storage. I/O fetches and stores are made to processor storage only.

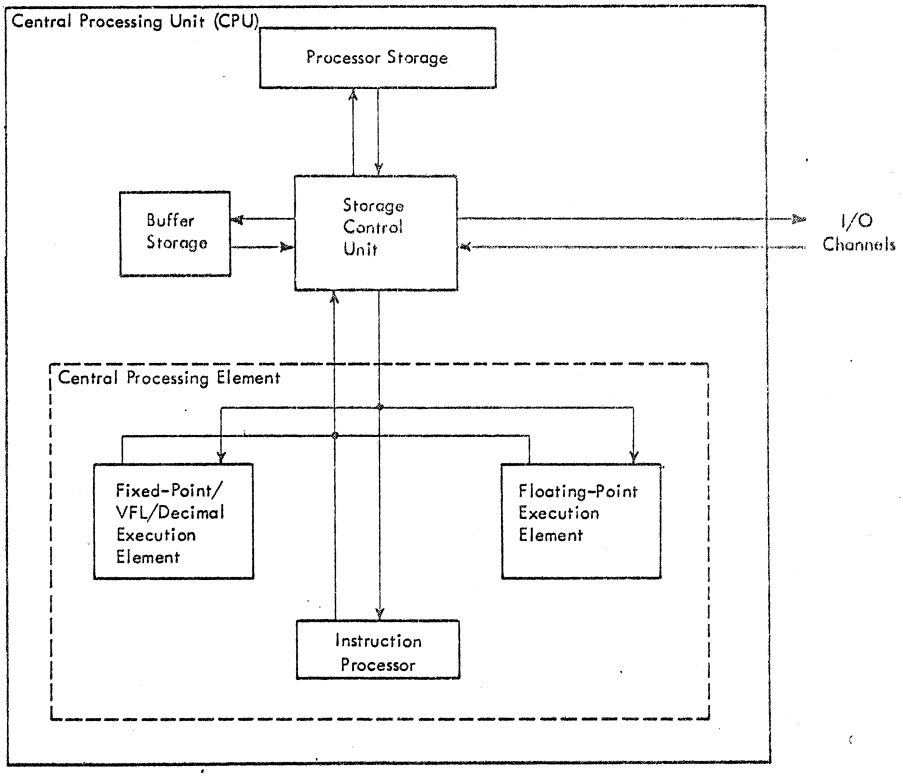


Figure 3. Model 195 Logical Elements

An I/O store to a location also currently held in the buffer storage invalidates the appropriate block in the buffer storage.

PROCESSOR STORAGE

Up to 4,194,304 bytes of processor storage are available with an individual access of eight bytes (a doubleword). Interleaving of processor storage is provided so that the addresses of successive doublewords are in successive, functionally independent units. Processor access to storage is performed in combination with the high-performance buffer storage. The effect is that average access time approaches the access time of the buffer storage. Transfers between the buffer storage and processor storage are made in blocks of eight doublewords. I/O accesses (eight bytes) to processor storage do not involve buffer storage except where necessary for control.

Function Performed	*Time (Nanoseconds)*
Access time to buffer storage	162
Access time to processor storage if not in buffer storage	810
Access time to processor storage for I/O channels	648
Cycle time for buffer storage, successive read or successive write cycles	54
Cycle time for processor storage	756

INSTRUCTION PROCESSOR

The primary functions of the instruction processor are fetching and buffering instructions from storage, fetching required operands, issuing instructions to the appropriate execution elements, handling interrupts, and executing all branching, status-switching, and I/O instructions.

The instruction processor has an instruction stack of eight doublewords, a set of three instruction-control registers, a set of temporary instruction buffer registers totaling two doublewords, a decoder, and a three-input adder for the generation of effective addresses (Figure 4). The instruction processor uses the general registers in the fixed-point/VFL/decimal execution element.

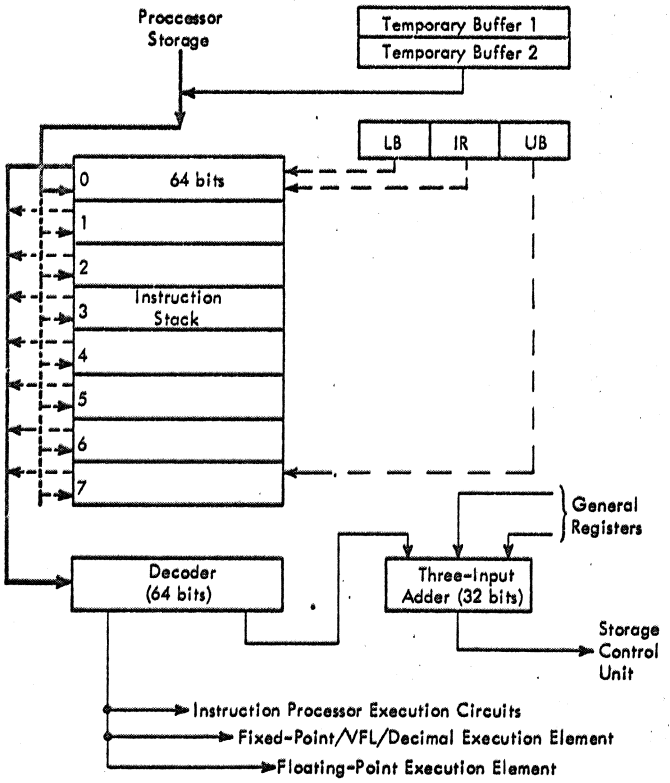


Figure 4. Instruction Processor

Instruction Fetching

Instructions fetched from storage are stored in the instruction stack of the instruction processor. An instruction stack is used for two principal reasons:

1. To minimize storage access time for instruction fetching.
2. To reduce the number of instruction fetches required while the program is executing a tight loop.

The instruction stack normally contains the current instruction doubleword, and seven doublewords of either history (instructions already decoded) or instructions to be executed. Keeping a number of doublewords ahead enables the fetching mechanism to fit instruction fetches into slack periods between data fetches and stores. The doublewords of history in the stack minimize refetching of instructions when a loop backward that can be contained in the instruction stack is detected.

The fetching mechanism operates differently under each of three conditions: initialization, normal operation, and recognition of a discontinuity. It is governed by three control registers: the instruction register (IR), the upper-bound (UB) register, and the lower-bound (LB) register. The IR points to the instruction being decoded, the UB register to the most recent doubleword brought into the stack, and the LB register to the earliest doubleword in the stack.

Initialization

Initially, the instruction stack is empty. When instruction fetching is initiated, the main-storage address of the first doubleword of instructions is set into the UB and LB registers, and part of the address of the first instruction is set into the IR. The UB register, which controls the actual fetching of doublewords of instructions, brings the first doubleword into the appropriate position of the instruction stack. At the same time, the first doubleword is brought into the decoder.

As each instruction doubleword is fetched during initialization, the UB register is incremented (a doubleword being brought into the stack for each increment) until any of three conditions occurs:

1. The address in the UB register is seven doublewords higher than that of the IR (Figure 4). Doubleword instruction fetches are made whenever it does not delay data fetching or storing.
2. A branch instruction is decoded that sets conditional mode (see "Execution of Branching Instructions").
3. A discontinuity is recognized (see "Discontinuities").

Normal Operation

During normal operation, the instruction fetching mechanism continually attempts to fetch a doubleword (Figure 4). Fetching will not take place if any of the three conditions just described is present.

When incrementing the UB register would cause the three low-order bits of that register to match the three low-order bits of the LB register, both the UB and LB registers are incremented. This incrementing of both registers causes the earliest (oldest) doubleword in the stack to be replaced with the latest doubleword just fetched. The LB and UB registers then point to a doubleword positioned one doubleword higher in the stack. This relative positioning of the LB and UB pointers (instruction stack addressing) remains constant during normal operation.

Discontinuities

A branch operation, interrupt, or store into the instruction stream may cause a disruption in fetching. (Branching operations and interrupts are discussed separately. See "Execution of Branching Instructions" and "Handling Interrupts.")

If the store instruction results in the alteration of the contents of a doubleword in the stack, the instruction fetching mechanism treats that doubleword slot as empty and fetches the altered doubleword from storage.

Because the Model 195 can execute several instructions at one time, the instruction STORE * + 4 presents a special problem. This problem is solved by making a check of the effective address of each store operation to determine whether the operation affects the instruction following the store; if the next instruction might be affected, measures are taken to preserve the logical consistency of the program.

Instruction Issuing

In addition to fetching and buffering instructions, the instruction processor fetches the required operands and issues instructions to the appropriate execution elements.

During each machine cycle, the instruction processor checks for interlocks. If there are none, the instruction selected by the instruction register is decoded. After an instruction has been decoded, the IR is incremented by the number of halfwords of the instruction just decoded, and the next instruction is then decoded. Decoding is the first of three possible stages in the issuing of the instruction.

Stage 1

During decoding, the following are determined:

1. The type of operation to be performed.
2. Whether the operation stack for the appropriate execution element can accept the operation.
3. If a storage operand is required, whether a buffer register in the appropriate execution element is available to receive the operand; or, if a store operation is specified, whether a store address register is available in the storage control unit.
4. If an effective address is required, whether the three-input adder and general registers used in generating the effective address are available.

When the results of these checks indicate that the instruction can be processed, the decoding control determines whether the instruction processor is operating in conditional mode (see "Execution of Branching Instructions"); if it is, the operation is tagged as conditional, indicating to the execution element that it is not to decode or execute the operation until signaled to do so. The operation is then issued for processing to the appropriate execution element (usually during stage 2), along with information about which buffer registers in the execution element, if any are needed, have been assigned by the instruction processor for use in the operation.

Stage 2

If address generation is required, the pertinent operand addresses are routed to the three-input adder. (Another instruction can now be processed at stage 1.) If the instruction is a store, a quick check is made of the effective address and, if this check indicates a possible store into the already fetched instruction stream, processing of the instruction at stage 1 is stopped until the processor determines whether the store is actually into the instruction stream. If it is, the processing at stage 1 remains stopped until the processor has issued a fetch to storage for the updated value of the instruction doubleword affected.

Fetches and stores can be made to operands that are not on proper boundaries; however, performance is degraded. Operands should be located on proper boundaries.

Stage 3

At this stage, the effective address of the storage operand is passed to the storage control unit. If a fetch operation is specified, the address of the buffer register to which the operand is to be issued is also specified. (During stage 3, another instruction can be processed at stage 1 and another at stage 2.)

Execution of Branching Instructions

The instruction processor executes all branching instructions. The actions taken by the instruction processor as a result of decoding a branch instruction are determined by

the type of branch instruction to be processed, the availability of circuitry for processing, and the following:

1. Whether the instruction processor is in conditional mode (see "Conditional Mode").
2. Whether the instruction processor is in loop mode (see "Loop Mode").
3. If loop mode is established, whether the current instruction is that which defined the current loop.
4. Whether the current instruction is the target of an 'execute' instruction currently being processed.

When a branch is taken, the target address of the branch normally is set into the instruction register, and the UB and LB registers and instruction stack are adjusted as required.

When a conditional branch is encountered and loop mode is not set, the instruction processor operates as though either direction could be taken. It continues to process the instructions in the instruction stack as long as conditions permit, while issuing operations to the fixed-point and floating-point execution elements on a conditional basis. These conditional operations will not be executed until after the condition code is set.

The instruction processor also makes use of two temporary buffers. Into these buffers it fetches the branch-target doubleword and the succeeding doubleword. Therefore, regardless of the outcome of the branch operation, the instruction processor will have a lead in the correct direction.

Conditional Mode

Conditional mode is established when the instruction processor executes a 'branch on condition' instruction for which the condition code is not yet determined.

When conditional mode is set, no additional instruction fetches are made beyond the first two doublewords at the target address of the branch. The instruction processor continues to decode instructions, generate addresses, and issue operations to the fixed-point and floating-point execution elements. The operations issued, however, are tagged as conditional and cannot be decoded or executed until the condition code is set and the instruction processor sends a signal to the execution element.

The instruction processor continues to decode instructions conditionally until any of the following occurs:

1. The condition code is set.
2. No instructions are available in the instruction stack.
3. The operation stack of the fixed-point or floating-point execution element is full, and the currently decoded instruction needs the filled execution element.
4. An instruction to be executed within the instruction processor is decoded, or a variable-field-length instruction is decoded. (However, a no-operation instruction or an unconditional branch may be executed during conditional mode.)

Loop Mode

Whenever a branch backward is taken to a target fewer than eight doublewords back from the current address in the instruction register, loop mode is entered and the instruction stack is reinitialized to contain the pertinent eight doublewords. The loop is then locked into the instruction stack and, as a result, can be executed repetitively without refetching the instructions. Thus, conflicts between instruction fetching and data fetching are eliminated, and branches can be executed faster.

During loop mode reinitialization, when no data fetches or stores are to be made, an instruction doubleword is fetched every cycle until the instruction stack is full. If data fetches or stores are to be made, instruction doubleword fetches take second priority.

When loop mode is entered, the branch target address is placed in one special register, and the address of the branch instruction is placed in a second special register. Subsequently, when a branch instruction is decoded during loop mode, that instruction address is compared with the address (in the second special register) of the branch instruction that initiated loop mode; if they are the same, the branch is made to the target address in the first special register. Because no time is taken to re-form the address specified in the branch instruction, one cycle is saved.

If a conditional branch instruction is processed when loop mode is already set, it is assumed that the branch will be taken; therefore, during loop mode no temporary fetches (down the no-branch path) are made for conditional branches.

Loop mode is turned off when any of the following occurs:

1. A branch out of the instruction stack is taken.
2. The instruction processor starts to decode the 32nd halfword in the instruction stack.
3. The target of the quick loop is the same as the target of the outermost loop, and the branch closing the quick loop is not taken. (If two nested loops fit in the instruction stack, the innermost loop is called the quick loop.)
4. The base register or index register of the quick-loop branch is altered.

Programming Notes: Because of item 2, a loop with 29-31 halfwords should be aligned on a doubleword boundary. If the loop has fewer than 29 halfwords, the loop is executed in loop mode regardless of boundary alignment; if it has more than 31 halfwords, it is not executed in loop mode.

Because of item 3, if the nested loops both have the same target address, loop mode will be destroyed every time an exit is made from the quick loop. To prevent loop mode from being destroyed, a no-operation instruction may be used as a dummy branch target for the outer loop.