Chapter 51

Architecture of the IBM System/370[1]

Richard P. Case / Andris Padegs

Summary This paper discusses the design considerations for the architectural extensions that distinguish System/370 from System/360. It comments on 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.

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[2] of System/360 were found to be necessary.

This paper reviews the motivation for extending the System/360 architecture and describes the design considerations associated with the extensions adopted for System/370.[3] 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 [Amdahl, Blaauw, and Brooks, 1964; Amdahl, 1964; Blaauw and Brooks, 1964; Blaauw, 1964; Padegs, 1964; and Stevens, 1964].

[1]*Comm. ACM*, vol. 21, no. 1, January 1978, pp. 73–96.

[2]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 date flow and controls, the logical design, and the physical implementation.

[3]This chapter 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 Operations*, form GA22-7000, which is available through local IBM branch offices.

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 [1976], who say: "There is only one mistake . . . that is difficult to recover from—not providing enough address bits. . . ."

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.

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 *IBM 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 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 1 lists the major categories of architectural extension that have been added to the System/360[1] architecture to form the System/370[2] 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 2 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.

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 *IBM System/360 Principles of Operation.* 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

[1]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.

[2]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.

Table 1 Architectural Extensions Incorporated in System/370

	Instructions	
Facility	*Unpriv.*	*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

Table 2 New Instructions Incorporated in System/370

Name	*Mnemonic*	*Type*		*Op code*
ADD NORMALIZED (extended)	AXR	RR	Unpriv.	36
CLEAR I/O	CLRIO	S	Priv.	9DO1
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.	9EO1
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

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 empahsize the kind of careful attention that compatibility requires.

Use of USASCII-8 Bit for Control of EC Model. 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.

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, 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 resoltuion 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, the 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 response, 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 sytem incorporates paging from a backing store as introduced in Atlas [Kilburn et al., 1962], and a second level of indirection, segmentation, as suggested by Dennis [1965] and as further detailed by Arden et al. [1966].

The System/370 version of this facility was largely patterned after the System/360 Model 67[Gibson, 1966]. 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.[1]

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.[2]

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

[1]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.

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

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.

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.[1]

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.

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

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).[1]

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 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 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

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

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 0, 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 encountering 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.[1] 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 [1968].

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 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

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

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.[1]

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

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

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 related 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 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 times 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 interprocessor 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.[1] 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 addresses 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.

[1]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.

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 permitting 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 indicates 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 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 implicity 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 statusword 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[1] channel with both high-data-rate and multiple-device capabilities [Brown, Gibson, and Thorn, 1972].

[1]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.

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 terminates 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 [IBM, 1978a], 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 chapter 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 3 shows these factors for the various models of System/360, and Table 4 for System/370.[1]

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

[1]In this chapter, capacities and widths are usually given in bytes. A byte is composed 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 3 System/360 CPU and Control Storage Characteristics

	CPU		*Control storage*				*TLB*
Model	*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.

$K = 2^{10} = 1024$), 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).

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 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 5 shows the options available for the System/360 models, and Table 6 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) × (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 ¼ of the storage cycle time, so that the total available transfer rate for sequential locations is $8 \times 4 = 32$ 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 [Conti, Gibson, and Pitkowsky, 1968; Liptay, 1968], 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.[1] The tables list the

[1]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.

Table 4 System/370 CPU and Control Storage Characteristics

	CPU		Control storage				TLB
Model	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 6.

[f]57.5 nsec for the execution of some instructions.

[g]Extended to 122 + 4 when any emulator is installed.

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

Table 5 System/360 Processor Storage Characteristics

	Processor storage			*Cache*			
Model	*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	8 × 2	750	none			
67	256-1024	8 × 2	750	none			
75	256-1024	8 × 4	750	none			
85	512-4096	16 × 4	960	16-32	80-160	256 × 4[b]	16
91	2048-6144	8 × 16	780	none			
195	1024-4096	8 × 16	756	32	54-162	8 × 8	4

[a]All models use magnetic-core technology. [b]Blocks of 64 bytes (16 × 4) are fetched from main storage only if referenced.

Table 6 System/370 Processor Storage Characteristics

	Processor storage			*Cache*			
Model	*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	16 × 2	4
165	512-3072	8 × 4	2000[a]	8-16	80-160	8 × 4	4
165-II	512-3072	8 × 4	2000[a]	8-16	80-160	8 × 4	4
168	1024-8192	8 × 4	320	8-16	80-160	8 × 4	4-8[b]
168-3	1024-8192	8 × 4	320	32	80-160	8 × 4	8
195	1024-4096	8 × 16	756	32	54-162	8 × 8	4
3031	2048-6144	8 × 4	920	32	115-230	8 × 4	8
3032	2048-6144	8 × 4	320	32	80-160	8 × 4	8
3033	4096-8192	8 × 8	290	64	58-116	8 × 8	16

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

Table 7 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	68-1	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	76-6	77-2
138	76-6	76-11
145	70-9	71-8
145-3	76-6	77-4
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	

a close variant) is executed separately for each equivalence class.

Announcement and Shipment Dates

Table 7 lists the year and month when the various models of System/360 and System/370 were announced and first shipped.

References

Amdahl, Blaauw, and Brooks [1964]; Amdahl [1964]; Arden et al. [1966]; Bell and Strecker [1976]; Blaauw and Brooks [1964]; Blaauw [1964]; Brown, Gibson, and Thorn [1972]; Conti, Gibson, and Pitkowsky [1968]; Dennis [1965]; Gibson [1966]; IBM [1978a]; IBM [1978b]; Kilburn et al. [1962]; Liptay [1968]; Padegs [1964]; Padegs [1968]; Stevens [1964].