# 8

# The IBM System/38

## 8.1 Introduction

IBM's capability-based System/38 [Berstis 80a, Houdek 81, IBM 8a, IBM 82b], announced in 1978 and delivered in 1980, is an outgrowth of work that began in the late sixties and early seventies on IBM's future system (FS) project. Designers at the IBM Development Laboratory in Rochester, Minnesota incorporated ideas from FS, modified by their needs, to produce a system for the commercial marketplace. It is interesting that such an advanced, object-based architecture has been applied to a very traditional product space. Initially, only the COBOL and RPG III languages were provided. The system, which includes the CPF (Control Program Facility) operating system, is intended to support transaction processing and database applications constructed in commercial languages.

A major goal of the System/38 design is to maintain programmer independence from the system implementation [Dahlby 80]; IBM wished to retain maximum flexibility to modify System/38's implementation for future technologies while supporting previously written System/38 programs. The designers also wished to support a high level of integrity and security at the machine interface and to support commonly executed user and system functions efficiently, such as database searches and memory management [Hoffman 80]. To meet these goals, IBM chose a layered machine structure with a high-level programming interface. The layers of this design are shown in Figure 8-1.

At the lowest level is a hardware machine that directly exe-

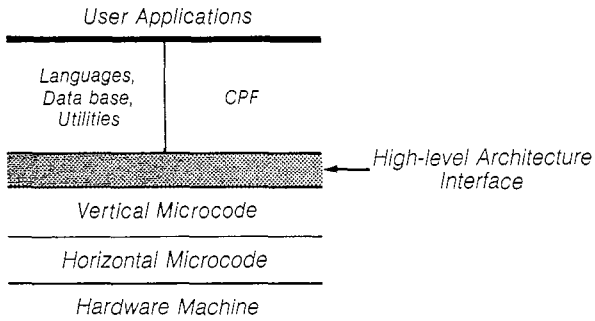



*Figure 8-1*: System/38 Implementation Layers

cutes 32-bit *horizontal* microcode. This horizontal microcode implements a more-or-less standard 32-bit register machine that executes *vertical* microcode.[1] The interface above the vertical microcode, called the high-level architecture interface in Figure 8-1, is the level described in this chapter; it supports the user-visible (or CPF-visible) System/38.

This high-level architecture interface is supported across implementations, while the structure of the underlying layers can change. For example, performance-critical functions, such as interprocess communication and memory allocation, are handled by the horizontal microcode. The system object and capability support is handled in part by both microcode layers. Different functions can be moved between microcode levels or into hardware in future versions, as performance experience is gained. In fact, this movement has already occurred on newer System/38 releases and models.

The CPF operating system and the vertical microcode are implemented in PL/S, a PL/I-like system programming language. There are approximately 900,000 lines of high-level PL/S code and an additional 400,000 lines of microcode support needed to implement CPF and its program products. The System/38 hardware includes a non-removable disk that holds this large store of microcode.

The System/38's high-level architecture interface is actually an *intermediate language* produced by all System/38 compilers. Before a program is executed, CPF translates this intermediate language into vertical microcode and calls to vertical microcode

[1]Although IBM calls this layer vertical microcode, it would generally not be considered microcode because it resembles a traditional IBM 370-like 32-bit instruction set and is programmed in a high-level language.

procedures. That is, the high-level interface is not directly executed. This translation process is described later.

IBM terminology is used throughout this chapter for compatibility with System/38 publications; it differs somewhat from that used in previous chapters. In particular, IBM uses the following terms: *space* for segment, *pointer* for capability, *authority* for rights, and *context* for directory. These synonyms will be presented again as each of the terms is introduced.

## 8.2 System Objects

System/38 instructions operate on two types of entities: *scalar data elements* and *system objects*. The scalar types are 16- and 32-bit signed binary, zoned and packed decimal, and character strings. The machine supports 14 types of system objects, described in Table 8-1. A set of type-specific instructions is provided for each system type.

| | |
|---|---|
| SPACE | byte-addressable storage segment |
| PROGRAM | procedure instructions and associated data |
| USER PROFILE | object containing information about user's resource limits and authority to access any system objects |
| CONTEXT | directory of object names and capabilities |
| QUEUE | message queue for interprocess communication |
| DATA SPACE | collection of identically-structured records |
| DATA SPACE INDEX | object used to provide logical ordering for data space entries |
| CURSOR | direct interface to entries in a data space, or indirect interface through a data space index |
| INDEX | accesses data sequences based on key values |
| PROCESS CONTROL SPACE | object containing state information for a process |
| ACCESS GROUP | set of objects grouped together for paging performance reasons |
| LOGICAL UNIT DESCRIPTION | object describing an I/O device |
| CONTROLLER DESCRIPTION | object describing the attributes of a device controller |
| NETWORK DESCRIPTION | object describing a communications port |

*Table 8-1:* System/38 System Object Types

Each system object consists of two parts: a functional portion and an optional space portion, as shown in Figure 8-2. The *functional portion* of an object is a segment containing object state (its representation); the data in the functional portion can be examined and modified only by microcode as a result of type-specific instructions. Thus, the functional portion is said to be *encapsulated* because it is not accessible to programs [Pinnow 80]. Optionally, a *space portion* can be associated with an object (IBM uses the word *space* to refer to a storage segment). The space portion is an attached segment for storing scalars and pointers that can be directly manipulated by user programs.

Every object in the system has several associated attributes. First is a *type* that identifies it as one of the 14 system object types listed in Table 8-1. (Objects can also have *subtypes* for further software classification.) Second is a symbolic *text name* chosen by the user to refer to the object. Last is a *unique identifier* (ID) that uniquely specifies an object for the life of the system. Object identifiers are never reused. When an object is created, the object ID is assigned by the system, while the text name and type are specified by the programmer.

Although the contents and format of the encapsulated data in an object are not programmer accessible, programmers must be able to specify initial object values or examine an object's state. The System/38 instruction set uses templates to convey initial information and communicate encapsulated data. A template is simply a data structure with defined fields used to transmit information at the instruction level. For example, the CREATE QUEUE instruction needs to specify some information about the maximum number of messages, the size of messages, the queueing discipline, and so on. This information is

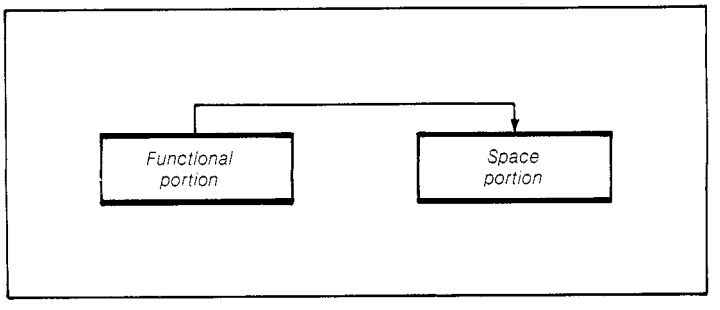



*Figure 8-2*: IBM System/38 System Object

conveyed by creating a template in a space and specifying a capability to that space as a parameter to the instruction. Later, an instruction can be executed to produce a template showing information about the queue. Although the architecture fixes the format of the template used to communicate information at the high-level interface, it does not dictate how that information is maintained once it is encapsulated within the object.

The only object not containing a functional part is a *space object.* A space object is a contiguous segment and is the only object that can be manipulated at the byte level by scalar instructions.

A system object, then, is an instance of an abstract data type. System/38 instructions exist to create, manipulate, examine, and delete each of the system object types. The machine provides an interface that hides the implementation of an object from the user. An object's state is stored in one or more segments; its attributes include a type that indicates what operations are allowed and an identifier that uniquely specifies the object. A base segment for each object contains pointers to any other segments composing the object, as well as type and ID information.

## 8.3 Object Addressing

Before examining object addressing in detail, it is necessary to describe memory management and segment addressing on the System/38. Object addressing, using capabilities, is based on lower-level segment addressing mechanisms.

### *8.3.1 Virtual Memory*

The IBM System/38 architecture supports a flat, single-level, 64-bit virtual address space. To the user at the high-level interface (either the operating system or application programmer), all addressable objects and segments are in directly accessible memory; there is no concept of secondary storage. The System/38 microcode is responsible for moving segments between primary and secondary storage to create this virtual memory environment.

The structure of a 64-bit virtual address is shown in Figure 8-3. The System/38 segment size is 64K bytes. Each segment is divided into 512-byte pages. The low-order 16 bits of the ad-

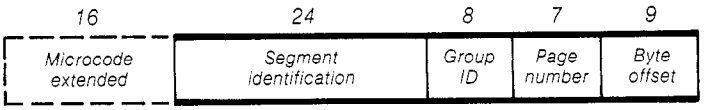



*Figure 8-3*: System/38 Virtual Address

dress thus provide the page number and byte offset for the pages of a segment. For larger objects, up to 256 segments can be grouped together into segment groups. The group ID field specifies which 64K-byte segment is being addressed within a 16M-byte segment group. The next 24 bits of the address provide a unique segment ID for the segment group.

The System/38 hardware only supports 48-bit physical addresses composed of these fields. However, when an object is created, the microcode extends the address to 64 bits by adding an additional 16-bit field.

The full 64-bit address is stored in a special header with the segment. When a 64-bit address is used to access a segment, the upper 16 bits of the address are compared with the upper 16 bits of address in the segment's header. If a mismatch occurs, the addressed object has been destroyed and the reference is not allowed. At any one time, then, there can only be $2^{24}$ or 16 million segment groups in existence.

Because the address space is so large, particularly with the 16-bit extension to the segment ID field, segment IDs are never reused. The system assigns a new segment ID at creation that is unique for the life of the system. If the object is deleted, references to the segment ID are not allowed. The system need not search for dangling references when an object is deleted. The segment ID, therefore, provides a mechanism for determining the unique ID for system objects. System objects are named with the unique ID of the first segment containing the functional portion of the object. The unique ID is the upper six bytes of the virtual address.

### *8.3.2 Pointers*

As in other capability systems, objects as well as scalar data elements are addressed through capabilities. System/38 capabilities are known as *pointers*. There are four types of pointers in the System/38:

- *system pointers* address the 14 system object types (listed in Table 8-1),
- *space pointers* address a specific byte within a space object (segment),
- *data pointers* address a specific byte within a space and also contain attribute information describing the type of element (e.g., character or decimal), and
- *instruction pointers* address branch targets within programs.

Each System/38 pointer is 16 bytes long. In order to access an object or an element within a segment, a program must specify a pointer that addresses the object or segment element. Pointers can contain different information at various times, including symbolic text names, authorization information (access rights), the object type, and the unique ID for system pointers or virtual address for data and space pointers. The information within a pointer can be modified, for example, from text name to unique ID, to allow for late binding of the pointer to the object.

Unlike the systems previously examined, which use C-lists for the storage of capabilities, System/38 pointers can be freely mixed in segments along with scalar data. To allow storing of capabilities with data in the same segment while still maintaining capability integrity, the System/38 implements a memory tagging scheme. Memory is byte addressable and words are 32 bits long. However, physical words of primary memory are actually 40 bits wide. Invisible to the programmer are a 1-bit tag field and a 7-bit error correcting code. Pointers must be aligned on 16-byte boundaries. When a pointer is stored in a segment by a valid pointer instruction, the hardware sets the associated tag bits for the four consecutive 32-bit words used to hold the pointer. Any instruction that requires a pointer operand checks that the pointer is aligned and that the four tag bits are set before using the element for addressing. Program data instructions can freely examine pointers. However, if a program instruction modifies any data in a pointer, the microcode turns off the tag bit in the associated word or words, invalidating the pointer.

Table 8-2 lists some of the instructions that operate on System/38 pointers. Note that a space object (a memory segment) is a system *object* that is addressed by a system *pointer*. A space pointer, on the other hand, is a capability that addresses a particular byte in a space object.

| | |
|---|---|
| ADD SPACE POINTER | adds a signed offset to the byte address in a space pointer |
| COMPARE POINTER FOR ADDRESSABILITY | compares two pointers to see if they address the same object, the same space, or the same space element |
| RESOLVE POINTER | searches a directory (see Section 8.3.3) for a named object and returns a pointer for that object |
| SET DATA POINTER | returns a data pointer for an element in a space |
| SET SPACE POINTER | returns a space pointer for an element in a space |
| SET SPACE POINTER FROM POINTER | if the source is a space or data pointer, creates a space pointer for the specified byte; or if the source is a system pointer, returns a space pointer for the associated space |
| SET SYSTEM POINTER FROM POINTER | if the source is a space or data pointer, returns a pointer for the system object containing the associated space; if the source is a system pointer, returns a system pointer for that same object |

*Table 8-2:* System/38 Pointer Instructions

### *8.3.3 Contexts*

Pointers are used to address objects; however, users refer to objects by symbolic text names. System objects called *contexts* implement directories for storing symbolic object names and pointers. When a new object is created, its symbolic name and an associated pointer are stored in a specified context. Table 8-3 lists the context instructions supported by the System/38.

The symbolic names stored in contexts are not necessarily unique, and a user can possess several contexts containing the same name but referring to different objects. This feature allows for testing and logical object substitution. A program that refers to an object by name can receive different objects depending on what context is used for name resolution. When a reference is made to a pointer containing an object name, the system examines the user's *Name Resolution List* (NRL). The NRL contains pointers to user contexts in the order that they should be searched. By changing the context ordering or manipulating entries, the user can change the objects on which the program operates.

| | |
|---|---|
| CREATE CONTEXT | creates a new context object and returns a system pointer to address it |
| DESTROY CONTEXT | deletes a context object |
| MATERIALIZE CONTEXT | returns name and pointer for one or more objects addressed by a context |
| RENAME OBJECT | changes the symbolic name for an object in a context |

*Table 8-3:* System/38 Context Instructions

### *8.3.4 Physical Address Mapping*

Because of the large size of a System/38 virtual address, standard address translation schemes involving indexing of segment/page tables with the segment/page number address field cannot be used. Instead, the System/38 hardware uses hashing with linked list collision resolution to find the primary memory address for a specified virtual address.

The basic units of physical and virtual storage are 512-byte pages. A translation scheme is used to locate a page in primary memory. The upper 39 bits of a 48-bit virtual address, encompassing the unique segment ID, specify a unique virtual page address for the page. A hashing function is applied to these bits to obtain an index into a data structure called the *Hash Index Table* (HIT), shown in Figure 8-4. The hashing function is an EXCLUSIVE-OR of low-order bits from the segment ID and group ID fields, and reverse-order bits from the page number field. This function provides uniform mapping from the sparse address space to the HIT [Houdek 80].

The HIT entry contains an index of an entry in the *Page Directory Table* (PDT). The PDT contains one entry for each page of primary memory. Each entry contains the virtual address of a corresponding primary memory page. That is, the index into the PDT is the page frame number for the virtual address described in the entry. Each entry also contains a link. The hardware checks the virtual address at the first entry pointed to by the HIT and follows the linked list until a virtual address match is found or the list ends. If a match is found, the index of that entry is used as the page frame number in the primary memory address. If no match is found, the page is not

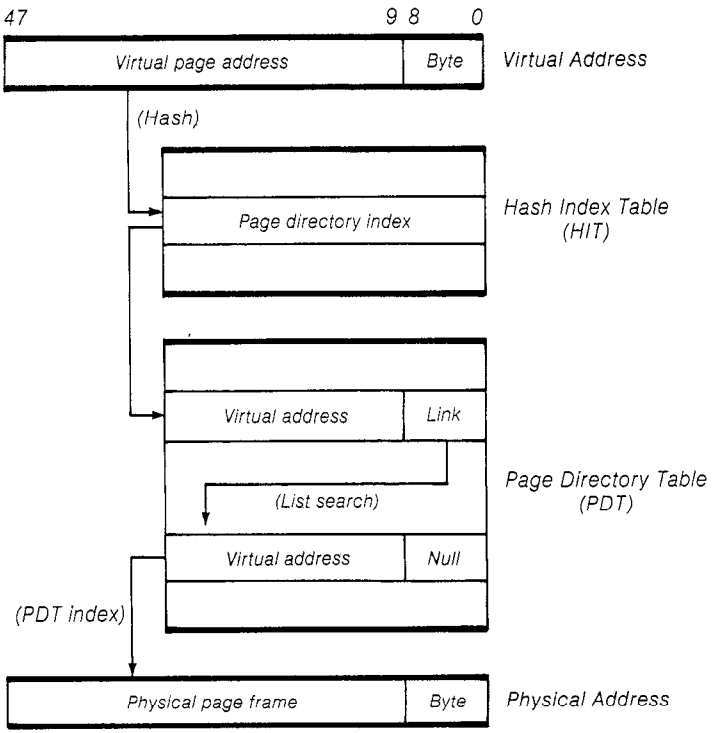



*Figure 8-4:* System/38 Virtual Address Translation

in primary memory and the hardware must load the page from secondary storage.

The performance of this search depends on the uniformity of the hashing function and the length of the lists in the Page Directory Table. In order to shorten the list lengths, the Hash Index Table is constructed to be twice the size of the Page Directory Table.

Two optimizations are used to avoid this two-level table search on every reference. First, the hardware contains a two-way associative translation buffer to cache recent address translations (the buffer size is different for different System/38 models, typically 2 x 64 or 2 x 128 entries). To check the translation buffer, the virtual page field is hashed to an offset that selects one entry in each half of the buffer. The two selected entries, which contain a virtual page address and translated primary memory page frame number, are checked for a match. If the virtual address matches, the page frame number is used to construct the primary memory address. If no match occurs,

the table search proceeds, eventually replacing one of the selected translation buffer entries with its data, based on a least recently used bit.

The second optimization is the use of *resolved address registers* in the hardware. These registers are used in the CPU to hold virtual page, physical page, and byte offset information while a page is being processed. As long as references are made to the addressed page (e.g., during the sequential search of elements of an array), the hardware need not search the translation buffer for consecutive accesses.

## 8.4 Profiles and Authority

The System/38 hardware provides a mechanism for ensuring privacy and separation of data and for sharing information between users. The basic unit of computation, from which protection stems, is the *process*. Each user process is defined by a *Process Control Space* object that contains its state. When a user logs onto the system, a new process is created; a *user profile* object is associated with that process based on the user's name. The user profile contains:

- the user's name,
- the user's password,
- any special authorization or privileges the user possesses,
- the maximum priority,
- the maximum storage usage,
- an initial program to run upon log-in (if any),
- a list of objects that the user owns, and
- a list of non-owned objects that the user is authorized to access, and the permitted authorities.

All authority to perform operations on objects is rooted in the user profile. When an object is created, it is created with an attribute stating whether the object is permanent or temporary. The profile associated with the process issuing the CREATE operation on a permanent object becomes the owner of the object. An owner has all rights to the object and can perform any operations, including deletion. Temporary objects receive no protection and have no owner. They are destroyed when the system is booted.

The owner of an object can grant various types of access to other user profiles in the system. There are a number of *authorities*, or access rights, that a process can have with respect to an object. The authorities define what object operations the

process can perform. The authorities also define what operations can be performed on pointers for the object. Object authorities are divided into three categories:

- *object control* authority gives the possessor control of the object's existence (for example, the right to delete or transfer ownership),
- *object management* authority permits the holder to change addressability (for example, to rename the object or grant authority to other profiles), and
- *operational management* authority includes basic access rights to the contents of the object, such as retrieve, insert, delete, and update entry privilege.

The authority information for each object is thus profile-based. Each user has a profile that indicates what objects are owned and what access is permitted to other objects. If a user wishes to allow access for an owned object to another user, the owner *grants* authority for the object to the sharer's profile. To execute a GRANT AUTHORITY instruction, a user must own an object or have object management rights. A user cannot grant an authority that the user does not possess.

Table 8-4 lists some of the profile/authority management instructions supported by the System/38. These instructions allow a properly authorized user to grant access privileges to other users, to examine what objects are authorized to him or her, and to see what authorizations have been given to other users for owned objects.

In addition to specific object authority granted to specific profiles, each object can have an associated *public authorization*. The object's owner grants public authority with the GRANT AUTHORITY instruction by omitting the profile parameter. The public authority is stored in the object's header and allows any user to access the object in the permitted modes. When an attempt is made to access an object, the public authority is checked first. If the access is not permitted by the object's public authority, the user's profile is then examined.

### *8.4.1 Authority/Pointer Resolution*

Thus far, the System/38 protection mechanism has been described from the perspective of the profile object. The profile provides a standard *Access Control List* mechanism. The owner of an object can explicitly permit other profiles to have access to that object and can later *revoke* that access.

| | |
|---|---|
| CREATE USER PROFILE | builds a new user profile (this operation is privileged) |
| DESTROY USER PROFILE | deletes a profile |
| GRANT AUTHORITY | grants specified authorities for an object to a specified user profile |
| MATERIALIZE AUTHORIZED OBJECTS | returns list of all owned objects or authorized objects |
| MATERIALIZE AUTHORIZED USERS | returns a list of owning or authorized users for a specified object |
| RETRACT AUTHORITY | revokes or modifies authority for an object from a specified user profile |
| TEST AUTHORITY | tests if specified authorities are granted to the current process for a specified object |
| TRANSFER OWNERSHIP | transfers ownership of an object to another profile |

*Table 8-4:* System/38 Authority Management Instructions

The ability to revoke object access is an important part of the System/38 design; this feature has not been provided in any of the previously examined systems. Revocation is, in fact, a difficult problem in capability systems and is generally expensive to implement in terms of addressing overhead. The IBM System/38 design allows an object's owner to decide whether revocation is needed for the object. The System/38 provides two pointer formats: one for which access can be revoked and another for which access cannot be revoked. An object's owner can decide which type of pointer to use for each object in each instance depending on the relative importance of revocation and addressing efficiency.

In order to access an object in the System/38 a process must possess a pointer for that object. Pointers can be stored in two formats: *unauthorized* and *authorized*. An unauthorized pointer contains an object's unique identifier but *does not* contain authorizations (i.e., access rights) to the object. When an unauthorized pointer is used to access an object, the hardware checks the profile of the executing process to verify that the requested operation is permitted. Without this check, revoca-

tion of authority would be impossible. An unauthorized pointer, then, cannot be used in the way that traditional capabilities can be used. Additional overhead is added to pointer usage because of the profile check.

In cases where revocation is not required or higher performance is needed, access rights can be stored in a pointer, creating an *authorized pointer*. An authorized pointer acts as a capability, and reference to an object with an authorized pointer does not require a profile lookup. The RESOLVE SYSTEM POINTER instruction is used to create authorized pointers. An authorized pointer can only be created by a user whose profile has object management authority for the object; the created pointer cannot have rights not available to the creating profile. Once constructed, an authorized pointer maintains authority to access an object for the life of that object. The pointer can be stored and passed to other processes. Because the profile check is avoided with authorized pointer usage, authority cannot be revoked later.

### 8.5 Programs/Procedures

IBM uses the term *program* to refer to what is typically called a procedure or subroutine. A System/38 program is an executable system object. A program object is created by a CREATE PROGRAM instruction, which specifies a template containing System/38 instructions and associated data structures. The CREATE PROGRAM instruction returns a system pointer allowing the program to be called.

As noted previously, the System/38 source language (i.e., the high-level architecture interface shown in Figure 8-1) is really an intermediate language produced by compilers. The effect of the CREATE PROGRAM instruction is to compile this intermediate language source into microcode that can be executed on the next lowest "level" of the machine. Source instructions, depending on their complexity, either compile directly into System/38 vertical micro-instructions or into micro-procedure calls. The compiled program is thus encapsulated in the program object, and the form of the micro-machine is hidden by the CREATE PROGRAM instruction. Once encapsulated, the format of a program object cannot be examined.

Thus, the System/38 high-level architecture is *never directly executed*. It is a specification for a language that all System/38 implementations support; however, that language is translated

into a proprietary vertical micro-language before execution. The format of the encapsulated program in this micro-language cannot be examined and can be different on different System/38 implementations.

### *8.5.1 The Instruction Stream*

The program template presented to the CREATE PROGRAM instruction consists of three parts:

- a program consisting of a sequence of instructions,
- an *Object Definition Table* (ODT), and
- user data.

Each instruction consists of a number of 2-byte fields including an operation code, an optional operation code extender, and one to four operands. The operands can specify literals, elements in space objects, pointers to system objects, and so on. The information about operand addressing and characteristics is stored in the Object Definition Table included in the template. The ODT is a dictionary that describes operands for the instruction stream.

Each instruction operand contains an index into the Object Definition Table. The ODT actually consists of two parts: a vector of fixed-length (4-byte) elements called the *Object Directory Vector* (ODV), and a vector of variable-length entries called the *ODT Entry String* (OES). An operand is either completely described by its 4-byte ODV entry, or the ODV entry has a partial description and a pointer into the OES, where the remaining description is found. Most commonly occurring cases are handled by the fixed-length ODV itself. Several ODV entries can point to the same OES entry. The ODT can contain information such as operand type (e.g., fixed-length decimal string), size, location, allocation (static or dynamic), initial value, and so on. Figure 8-5 shows an example of an instruction with three operands. The operands index ODT information defining their type and location.

Each instruction operand consists of one or more 2-byte fields. The first 2-byte field contains a 3-bit mode field and a 13-bit ODV index. The mode field indicates what type of addressing is required and what additional 2-byte fields (called secondary operands) follow in the instruction stream to describe the operand completely. For example, a string operand may require three 2-byte fields to describe a base, index, and length.

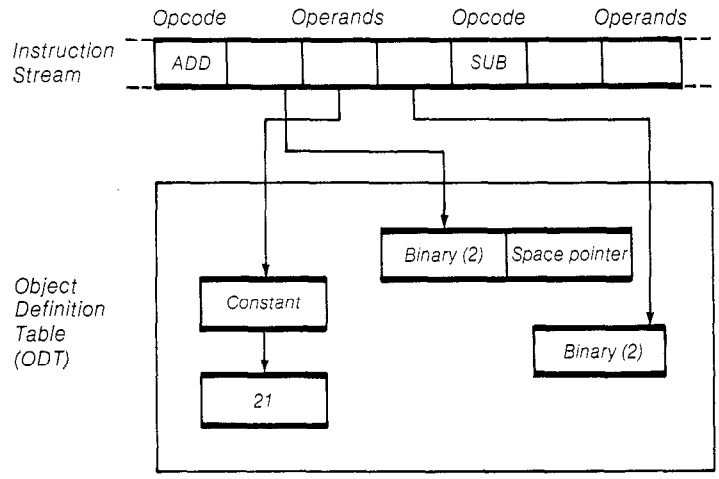



*Figure 8-5*: System/38 Example High-level Instruction

Since the ODT completely describes each operand, the scalar opcodes are generic. For example, there is only one ADD NUMERIC instruction that operates on all numeric data types. The machine interprets the ODT entry to decide how the operation should be performed and what conversions are required.

The *Object Mapping Table* (OMT) is the final data structure that is part of the encapsulated program (although not included in the initial template). It contains 6-byte mapping entries for each entry in the ODV that maps to a space.

### *8.5.2 Program Activation and Invocation*

A program, then, is a system object that represents a separately compiled unit of execution (typically known as a procedure). Programs are called by the CALL instruction. There are actually two parts to the initiation of a program on the System/38: activation and invocation.

Before a program can be invoked (called), it must be *activated*. Activation of the program causes static storage for the program to be allocated and initialized. Also, any global variables in program static storage are made addressable. A process data structure called the *Process Static Storage Area* (PSSA) contains an activation entry for each activated program in the process. The activation entry contains status information, a count of the number of invocations using the activation, the

size of static storage, and the storage itself. The first entry in the PSSA contains headers for the chain of activation entries and a free space chain.

*Invocation* occurs as the result of a transfer of control to the program. At invocation time, program automatic (that is, dynamic) storage is allocated and initialized in a process data structure called the *Process Automatic Storage Area* (PASA). Each invocation entry contains status information, a pointer to the previous invocation entry, a pointer to the program, and the automatic storage. After the invocation entry is allocated and initialized, control is transferred to the program at its entry point.

Activation can occur implicitly or explicitly. If invocation is requested of a program that has not been activated, activation is done automatically by the hardware.

### *8.5.3 Protected Procedures*

The IBM System/38 provides a mechanism for creating protected subsystems. As on previous systems, a protected subsystem mechanism must allow programs to execute in an amplified protection environment. That is, some programs must be able to access objects not available to their caller. Since the System/38 profile object defines a domain of protection, protected subsystems are provided through profile-based facilities called *profile adoption* and *profile propagation.*

The authority of each System/38 process is determined by its profile. When a process calls a program, that program generally gains access to the process's profile and, therefore, to the process' objects. However, it is possible to construct System/38 programs that can access additional objects not available to the caller. When a program is created, the program's owner can specify that the program retain access to the *owner's* profile, as well as its caller's profile. This feature, called profile adoption, allows a called program to access objects not available to the caller and can be used to construct a protected subsystem.

Although the general calling mechanism allows a called program access to its caller's profile, a calling process can also restrict this ability. When a program is created, the program's owner can specify whether its profile should be *propagated* to programs on calls. Thus, a program can also see that its owner's profile is protected from access by programs further down the call chain.

### 8.6 Special Privileges

It is worth noting that there are some special privileges in the System/38 authorization system. In addition to object-based authorities stored in a user profile, there may be other permitted authorities that are not connected with any particular object. For example, the ability to create user profiles, diagnose the hardware, or create objects representing physical I/O devices can be controlled by authorizations in a user profile. Also, the ability to dump and load objects to removable storage is protected, as well as the ability to execute operations to modify or service system hardware attributes. Finally, some objects, such as user profiles and device descriptions, receive special protection and can only be addressed through a special machine context (directory).

### 8.7 Discussion

The IBM System/38 is a complex architecture constructed from several levels of hardware, microcode, and software. Because of its commercial orientation and the fact that it is available from IBM, the System/38 is probably destined to become, at least in the immediate future, the most pervasive object architecture.

The most interesting feature of the System/38, from the viewpoint of capability systems, is its use of tagging. Capabilities and data can be freely mixed in segments with no loss of integrity. The ability to mix data and capabilities generally permits more natural data structuring than the C-list approach. A single tag bit associated with each 32-bit word indicates whether or not the word is *part* of a capability. This tag bit is hidden from the programmer and accessible only to the microcode. To be used for addressing, a pointer must be aligned on a 16-byte boundary and have all four tag bits set. The alignment requirement prohibits the user from specifying four consecutive words with tags set that lie within two contiguous capabilities.

The integrity of a capability system must be ensured on secondary storage as well as in primary memory, and the pointer tags must be saved on secondary storage. On the System/38, each disk page is 520 bytes long and stores a 512-byte data page and an 8-byte header. The 8-byte header for each block contains the virtual address for the page, an indication of whether or not the page contains any pointers, and if so, which 16-byte quadword contains the first pointer in the page. Each

page can contain, at most, 32 pointers; therefore, only 32 bits are required to specify whether each quadword contains a pointer. If a page contains pointers, the tag bits are stored within some unused bytes in the first 16-byte pointer on the page. When a page is written to disk, the hardware automatically writes the disk block header. When a page is read into primary memory, the header is automatically removed and the tags are reconstructed.

The System/38 architecture provides a large single-level address space. The details of memory management, I/O, and so on are hidden from the programmer. There is no need for a traditional file system. All objects can be declared permanent when created, can be stored for long periods of time, and can be addressed at any time as if they were in primary memory. Addressing is independent of the object's memory residency characteristics. One problem with schemes that remove the abstraction of secondary storage is in transaction systems or reliable data base operations. In some instances, the programmer may wish to ensure that the latest copy of a segment or object is checkpointed onto long-term storage. The one-level memory scheme has removed the ability to express the thought of writing the segment to disk. To solve this, CPF allows an object attribute that states how frequently data is to be backed up for a particular object.

The System 38 permits revocation by adding an access control list mechanism to the capability addressing mechanism. Two types of pointers, authorized and unauthorized, can be used depending on whether or not revocation is required. Authorized pointers are traditional capabilities because they contain access rights and can be freely copied. Passing an authorized pointer passes both the addressing rights and privileges. The ability to resolve a pointer to load the access rights is controlled by an authorized pointer authorization. Only suitably privileged profiles can create an authorized pointer.

In contrast, an unauthorized pointer is not a capability in the traditional sense. The same unauthorized pointer can permit different types of access when used by different processes. This is because the authorization rights are fetched from the process's profile when a reference is made. This extra step in pointer address evaluation permits explicit control over authority and combines the advantages of standard capability systems and access control lists. The user can specify (and determine at any time) what other profiles are allowed access to the user's objects. If only unauthorized pointers have been distrib-

uted, access can be revoked by removing authorization from other profiles.

Unauthorized pointers permit revocation but add complexity to the handling of pointers. For example, to pass a pointer to another process, the possessor of the pointer must be aware of whether that pointer is authorized or unauthorized. Unauthorized pointers, unlike capabilities, are not context independent. If the pointer is unauthorized, passing it to another process will not permit object access unless permission has been granted to the other process's profile. Also, unauthorized pointers cannot easily be used to build and share data structures. For example, if a user wishes to build a tree structure of segments and pass the tree or subtrees to other processes, the authorization scheme requires that authorization for each segment be granted separately to each profile involved.

The structuring of System/38 authorizations permits close control of pointers. Given the division of authority into object control, management, and access, it is possible for one user to be able to affect the propagation of addresses but not be able to access object data. Another user may be able to read and write but not propagate pointers.

The large size of the System/38 address space simplifies many problems. Segment identifiers are large enough that they are never reused. This allows use of the segment ID as a unique name for an object. Since the ID is never reused, there is no problem with dangling references. An attempt to access a deleted object simply causes an exception. Large IDs also simplify the implementation of the one-level memory system. There is no separation of long-term unique ID and address. The unique ID is the virtual address used to access a specific object, segment, or byte. There is no need for separate inform and outform capabilities or for transforming capabilities in memory when a segment is removed from memory.

Although the System/38 instruction stream and Object Definition Table are never used for direct execution, this interface has some interesting features. The ODT provides a form of tagging somewhat different from the tagged architectures examined earlier. Each data element is tagged; however, the tag is part of the operand, not part of the element. This allows for several different views of the same data element; different instructions can treat the same word as different data types. Operation codes can be generic, and conversion, truncation, etc. can be performed based on type information in the ODT. The information stored in the ODT and I-stream may not be ex-

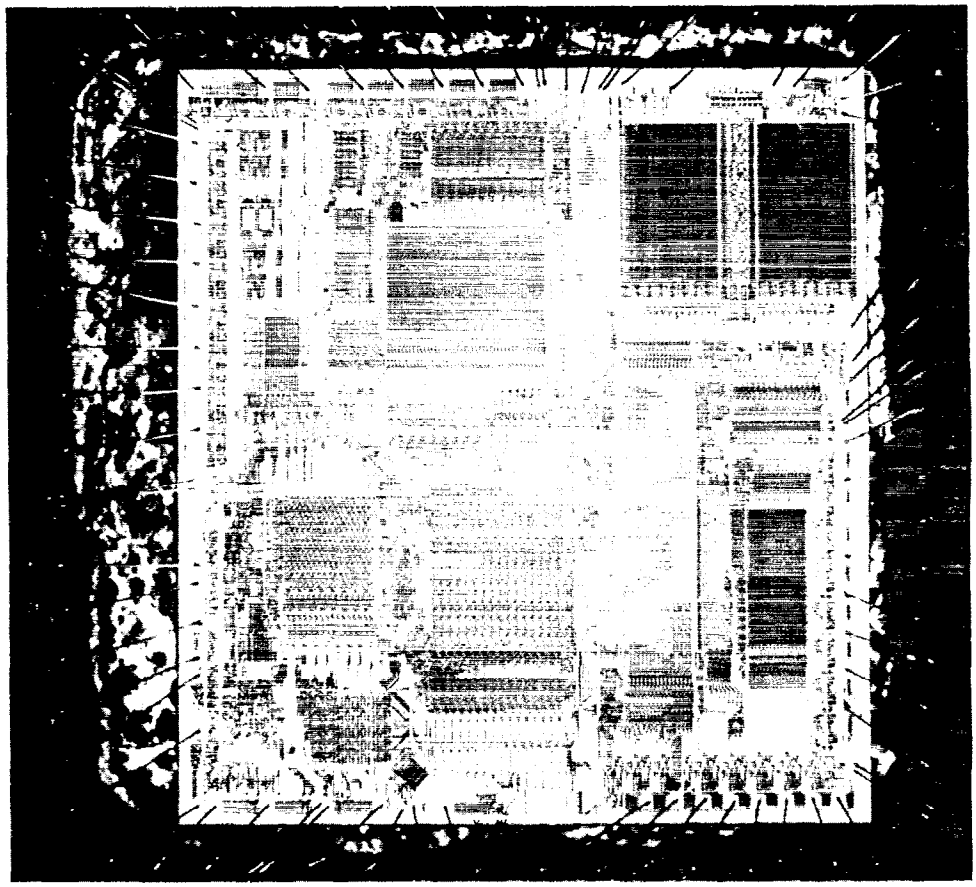
tremely compact, but the program in this form need not be retained after a program object is created.

Finally, IBM has used the object programming approach to allow isolated construction of components of a very complex system. The object approach is intended to hide from the programmer the implementation details of the System/38 hardware, so that future System/38 implementations can take advantage of advances in technology without affecting existing programs. Although this has been a goal of other architectures, the System/38 has used the object approach to place the user/system boundary at an unusually high level, hiding many details of the machine. For example, the System/38 high-level architecture has no registers, although the vertical microcode is free to use registers or to use different numbers of registers in different implementations.

The initial System/38 product, with its limitation to commercial languages, does not stress the architecture. It will be interesting in future years to see if IBM approaches other markets with this object-based machine structure.

## 8.8 For Further Reading

Detailed information about the System/38 high-level architecture can be found in two IBM manuals [IBM 80a, IBM 82]. IBM has also packaged a collection of 30 short technical papers, mostly dealing with hardware and implementation issues, into a document called *IBM System/38 Technical Developments* [IBM 80b]. Several papers describing the addressing and protection features of System/38 have also been published in technical literature [Berstis 80a, Houdek 81, Soltis 79, Soltis 81].



The Intel iAXP 432 computer. (Courtesy Intel Corporation.)