9
The Intel iAPX 432

9.1 Introduction

In 1981, Intel introduced the first object-based microprocessor, the iAPX 432 [Intel 81, Rattner 81, Organick 83]. Like the IBM System/38, the Intel 432 implements many operating system functions in hardware and microcode, including process scheduling, interprocess communication, and storage allocation. The integration of such software operations in hardware is particularly impressive when considered with the Intel 432's VLSI implementation.

The Intel 432 design effort began in 1975 with an attempt to implement in silicon a system much like Carnegie-Mellon's Hydra operating system [Wulf 74a]. Three chips compose the Intel 432 processing elements. The central processing unit, called the General Data Processor (GDP), is implemented on two 64-pin VLSI chips. Together, the GDP chips contain over 160,000 components. The Interface Processor (IP), responsible for communication and data transfer between the Intel 432 and its I/O subsystems, is the third 64-pin chip. Design and layout of the chip set took more than 100 man-years.

The 432 is a multiprocessor system that can accommodate a total of six processors, each either a GDP or IP. The general structure of the 432 multiprocessor system is shown in Figure 9-1. All of the processors are connected to a single multiprocessor message bus through which they communicate with each other and with shared system memory. The IPs connect the multiprocessor system to Intel Multibus subsystems. Each Multibus is controlled by an associated processor, such as an

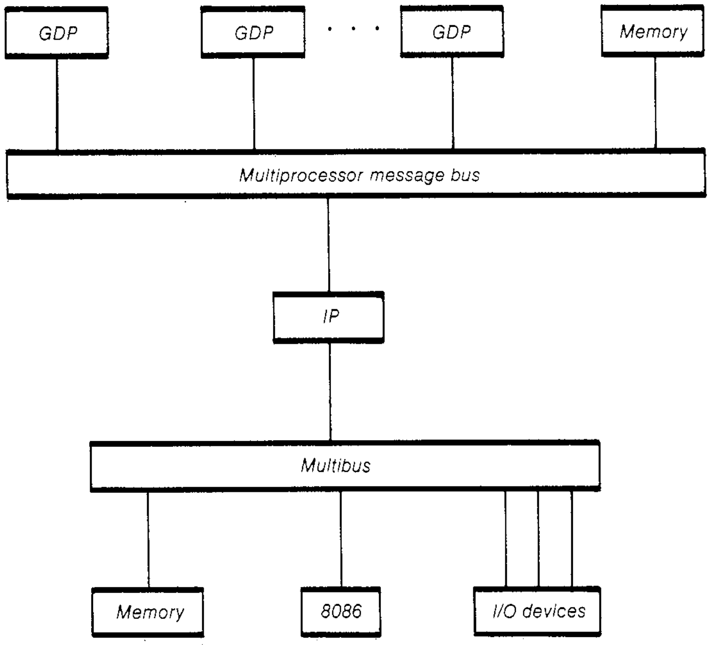

Figure 9-1: Intel iAPX 432 Structure

Intel 8086, that connects to local memory and some number of I/O devices. The IPs transfer data between Intel 432 memory and Multibus local memory; all I/O is actually performed by the associated processor.

The Intel 432 instruction set provides two types of instructions: scalar and object-oriented. The scalar instruction set consists of a small set of move and store operators, boolean arithmetic, binary and floating point arithmetic, and comparison operations. Scalar instructions operate on 8-bit characters, 16- and 32-bit signed and unsigned integers, and 32-, 64-, and 80-bit floating point numbers. The 432 has a stack architecture; instruction operands can be fetched from the stack and results can be pushed onto the stack. There are no general-purpose registers.

An object-oriented instruction set provides operations on abstract objects that are managed by a combination of hardware and software. The following sections examine many of those object types and the details of object addressing on the Intel 432. It should be noted that the Intel 432 architecture has evolved since its introduction; this chapter reflects the system as of revision 3 [Intel 82].

9.2 Segments and Objects

The concepts of object-based computing are deeply imbedded in the Intel 432. All system resources are represented as objects; for example, a *processor object* maintains the state of each GDP or IP in the system. Each processor object then contains a queue of *process objects,* which represent work to be scheduled and executed. All objects are addressed through capabilities which, on the Intel 432, are called access descriptors (ADs). (The vendor's terminology is used in this chapter for compatibility with Intel literature. The notation "AD" is used throughout for "capability.")

At the lowest level, objects are composed of memory segments, and a memory segment is the most fundamental object (called a *generic object* on the Intel 432). Each Intel 432 segment has two parts: a *data part* for scalars and an *access part* for ADs, as shown in Figure 9-2. Objects requiring both data and access descriptors can be stored in a single segment. Segments are addressed through ADs, as the figure illustrates. The data part grows upward (in the positive direction) from the boundary between the two parts, while the access part grows downward (in the negative direction) from the dividing line. The hardware ensures that only data operations are performed on the data part and that AD operations are performed on the access part.

Each part of a segment can be from 0 to 64K bytes in size. Data elements in the data part are addressed as byte displacements from the dividing line. ADs, which are 32-bits long, are addressed by integer indices from the dividing line. The access part can therefore contain up to 16K ADs. Both data elements and ADs are addressed as positive indices within the segment;

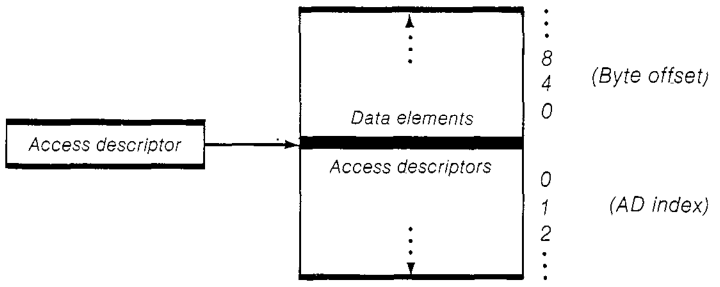

Figure 9-2: Intel 432 Segment

the hardware determines the part of the segment to access based on the type of the required operand.

In addition to basic storage segments, the Intel 432 hardware supports a number of system object types, listed in Table 9-1. The representation for an instance of a system object is maintained in a storage segment. Operating system type managers are responsible for creating new instances of system objects. A type manager creates and sets the type for an object through the CREATE TYPED OBJECT instruction. The operands for this instruction specify the object's type, the data part size, and the access part size. The instruction returns an AD for the new object, which the type manager uses to initialize the object appropriately.

For each system object type, the 432 architecture specifies the meaning of some of the data and/or access fields. These processor-defined fields are stored in the low-index portions of the two segment parts, adjacent to the boundary. A type manager is free to allocate additional data or access descriptor space in higher address parts of the two regions for object information needed by software.

GENERIC SEGMENT	basic storage for data and access descriptors (capabilities)
DYNAMIC SEGMENT	storage segment created by a programmer-defined type manager
INSTRUCTION SEGMENT	segment containing executable code
PROCESS	basic unit of scheduling
PROCESSOR	432 GDP or IP
DOMAIN	module or package
CONTEXT	dynamic state for a procedure invocation
MESSAGE PORT	interprocess communication object
CARRIER	extension of a message used to queue it to a port
TYPE DEFINITION	object containing information about a specific object type
TYPE CONTROL	object permitting creation of specific object types
STORAGE RESOURCE	source of primary memory for object storage allocation
OBJECT TABLE	mapping table of object descriptors

Table 9-1: Intel 432 System Object Types

9.3 Object Addressing

As in previous capability-based systems, there are two components to the Intel 432 addressing structure. First, a single descriptor contains the physical mapping information for each object. These descriptors, on the Intel 432, are called *object descriptors*. Second, programs specify *access descriptors* to refer to objects that they wish to manipulate. All ADs for an object refer to that object indirectly through its single object descriptor. The following sections describe first object descriptors and then access descriptors.

9.3.1 Object Descriptors

For each Intel 432 object there is a single *object descriptor*. The object descriptor contains information about the physical location and state of the object. The purpose of the object descriptor is to locate this physical object information in a single place so that objects can be easily relocated or synchronized. Each object descriptor is 16 bytes long. There are several types of object descriptors, but the most common is a storage segment descriptor, shown in Figure 9-3. Table 9-2 describes the fields in the storage segment descriptor.

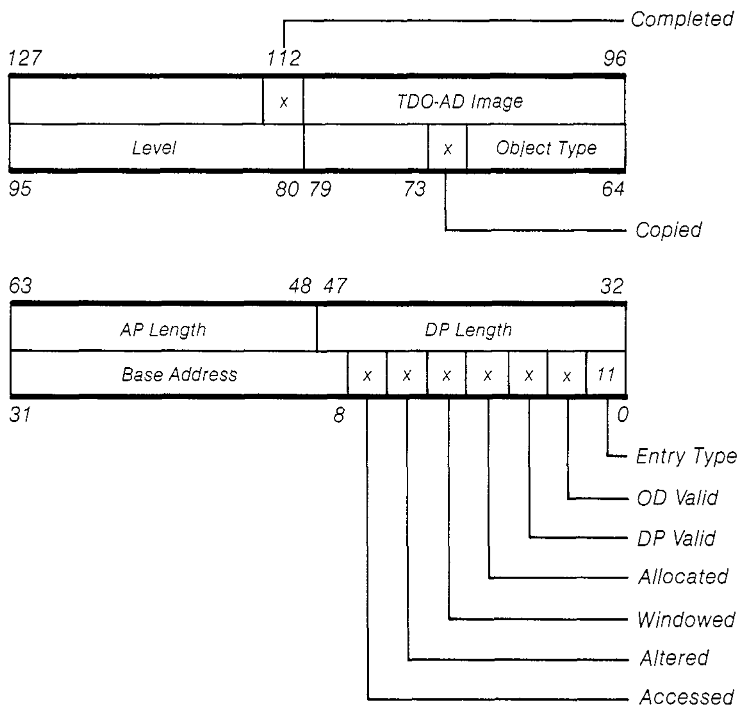

Figure 9-3: Intel 432 Storage Segments Descriptor

ENTRY TYPE	indicates that this is a storage descriptor
OD VALID	specifies whether the object descriptor can be used for addressing
DP VALID	indicates whether or not the object has a data part
ALLOCATED	specifies whether or not storage is allocated for this object
WINDOWED	indicates whether or not this object is being mapped by an IP
ALTERED	set to 1 whenever the object is written
ACCESSED	set to 1 whenever the object is accessed
BASE ADDRESS	primary memory address of the first byte of the segment's data part
DP LENGTH	length in bytes (minus one) of the segment's data part
AP LENGTH	length in bytes (minus one) of the segment's access part
OBJECT TYPE	type of the object, consisting of a 5-bit system type field (specifying system objects, shown in Table 9-1) and a 3-bit processor type field (specifying whether a GDP or IP owns the object)
COPIED	set to 1 whenever an AD referencing this object is copied
LEVEL	level of this object (generally the call depth at which it was allocated)
TDO-AD IMAGE	AD that defines the type manager that created this object
COMPLETED	used by software in object initialization

Table 9-2: Intel 432 Storage Segment Descriptor Fields

Each object descriptor is contained in an *object table.* The Intel 432 object table corresponds to the central capability table of previous systems. Unlike previous systems, however, there are many object tables in existence at any time. In general, every process executing in the 432 has an associated object table. Or, several processes can share a single object table. An object table therefore contains information about objects local to one or more processes.

In addition to the many process object tables, there is a *single* system-wide *Object Table Directory.* The Object Table Directory contains object descriptors that address each of the process object tables. Object tables are thus objects themselves and can be swapped out or relocated like other objects. The Object Table Directory, however, must always reside in primary memory. Each processor object contains the primary memory address of the Object Table Directory.

9.3.2 Access Descriptors

While each object has only one object descriptor, many *access descriptors* can be used to address the object. ADs are 32-bits long and specify addressing and access rights to an object. To execute an instruction that manipulates an object, the programmer specifies the location of an AD for the object. The AD is specified by its index in the access part of a segment. The collection of ADs accessible to a procedure define that procedure's execution environment: that is, the set of objects the procedure can address and manipulate.

An AD, illustrated in Figure 9-4, contains access rights to an object along with two 12-bit mapping indices. The read, write, and type rights fields are rights with respect to the addressed object. Type rights are type dependent and their encoding is different for each object type. Some type rights for system objects are defined by the architecture and evaluated by hardware instructions. The delete rights bit permits the possessor to delete the AD itself. An attempt to delete an AD with this bit set to zero causes a fault. The unchecked copy rights bit, indicating whether the object was allocated from a global or local storage pool, is used to avoid dangling references (described in Section 9.6).

Table 9-3 lists the instructions that operate on ADs. Note that ADs can be freely copied to the access part of any accessible segment. The INSPECT ACCESS DESCRIPTOR instruction copies the image of an AD to a segment's data part for examination. Of course, an AD image stored in a data part cannot be used as an AD.

Locating an Intel 432 object through an AD requires two steps. The AD, in addition to the rights bits, contains two indices: an index into the system-wide Object Table Directory

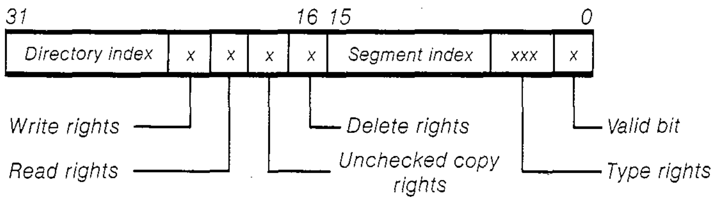

Figure 9-4: Intel 432 Access Descriptor

COPY ACCESS DESCRIPTOR	Copies an AD from one segment's access part to another.
NULL ACCESS DESCRIPTOR	Invalidates an AD.
INSPECT ACCESS DESCRIPTOR	Copies the information in an AD into a segment's data part.
INSPECT OBJECT	Copies the information from an AD and the object descriptor to which it refers into a segment's data part.
AMPLIFY RIGHTS	Amplifies the rights in an AD under control of a Type Control Object.
RESTRICT RIGHTS	Removes rights specified by an AD.
CREATE OBJECT	Creates a segment with specified data part and access part lengths, and returns an AD for the segment.
CREATE TYPED OBJECT	Creates a segment of the specified type with specified data part and access part lengths, and returns an AD for the segment.

Table 9-3: Intel 432 Access Descriptor Instructions

and an index into an object table. This mapping is shown in Figure 9-5. The first index locates the object descriptor for an object table. The second index locates the object descriptor for the specified object in the selected table.

Each access to a byte in a segment potentially requires four references, one each to:

- the access descriptor in an access segment,
- the Object Table Directory,
- the object table, and
- the byte itself.

With the exception of the access to the AD, the two-level mapping overhead is comparable to the overhead required on any conventional virtual memory system. Of course, caches can be used to decrease this overhead substantially. The first implementation of the 432 has several small on-chip caches to remember recently used translations.

Since AD index fields are 12 bits, an object table can have a maximum of 4096 (2^{12}) object descriptors. In addition, there

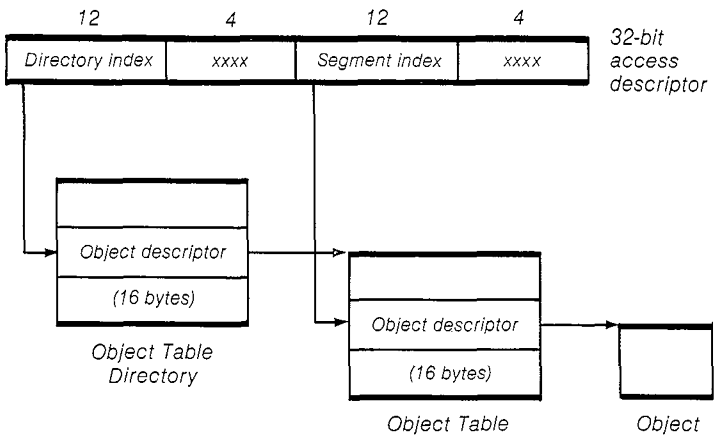

Figure 9-5: Intel 432 Address Translation

can be a maximum of 4096 object tables in the system at any time. Combined with the fact that a segment has a maximum size of 64K bytes, the total size of the address space is 2^{40} bytes. However, the maximum address space available to a procedure at any one time is 2^{32} bytes.

9.4 Program Execution

Several system objects exist to support the representation and execution of procedures on the Intel 432, including:

- the *domain object,* which defines a module, package, or set of related procedures,
- the *instruction object,* which defines a single executable procedure, and
- the *context object,* which provides the execution environment for an executing procedure.

These objects can be grouped into two classes—those that describe the *static* representation of procedures (the domain and instruction objects) and those that describe the *dynamic* execution of procedures (the context object). An instruction object corresponds to a Hydra procedure object, while the context object corresponds to a Hydra local name space object. At any time, there may be several context objects that represent different invocations of a single instruction object. The following sections describe these program objects in more detail.

9.4.1 Domains and Instruction Objects

A domain object, illustrated in Figure 9-6, contains ADs for the instruction objects and local objects used within a module. The Intel 432 architecture specifies the format of the first two ADs in a domain. These ADs address instruction objects that handle fault and trace conditions for all procedures in the domain. In the event of a fault or trace condition, the hardware automatically branches to the first instruction in the fault or trace object specified in the domain of the currently executing procedure. The remainder of a domain's access part contains ADs for procedures and objects needed by the domain; these ADs are defined by the software system (usually a compiler) creating the domain.

One of the objects typically addressed by a domain is a segment containing scalar constants used by the domain's procedures. Each instruction object, shown in Figure 9-6, contains the domain index of its scalar constants segment. This segment is needed because Intel 432 instructions do not have literal operand values embedded within the instruction stream. The instruction object also specifies the size of the context object to be produced as the result of the procedure call. The initial

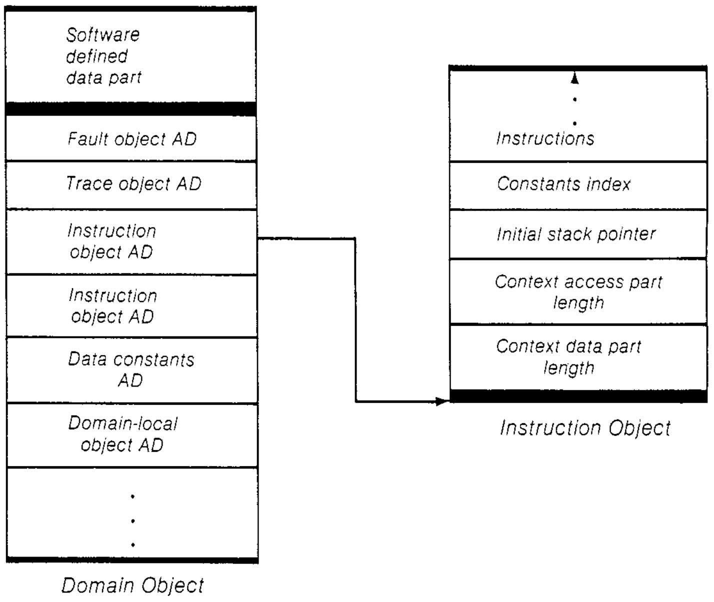

Figure 9-6: Intel 432 Domain and Instruction Objects

stack pointer index is the displacement to the start of the data stack in the context object. The use of these fields will become apparent in the following discussion of context objects.

Instruction objects contain only a data part. Because Intel 432 instructions are bit-addressable and can start on arbitrary bit boundaries, instructions are addressed as bit offsets into instruction objects. The first instruction in each instruction object begins at bit displacement 64, following the header of four 16-bit predefined fields. The maximum size of an instruction segment is 64K bits, or 8K bytes, due to the bit addressing. Although there is generally one instruction object for each procedure in the domain, procedures larger than 8K bytes require additional instruction objects. The BRANCH INTERSEGMENT instruction can be used to transfer control to another instruction object within the same domain.

9.4.2 Procedure Call and Context Objects

To transfer control to a procedure, a program executes a CALL instruction, causing the procedure to be invoked. On execution of a CALL instruction, the hardware constructs a new context object. The context object is the procedure invocation record and defines the dynamic addressing environment in which the procedure executes. All addressing of objects and scalars occurs through the context object, and the context is the root of all objects reachable by the procedure. The structure of the context object is illustrated in Figure 9-7.

Although somewhat complicated, it is important to examine the context object in more detail to understand the addressing environment of the Intel 432. The context object has both a data part and an access part. The data part contains pointers that describe the current instruction execution. The domain index locates the AD for the executing instruction object within the current domain; the instruction pointer contains the bit offset of the current instruction in that instruction object. At the high-address end of the context object's data part is the operand stack. This stack is used by instructions for computation and intermediate storage of scalar values. The current stack pointer is also stored in the data part.

The context object's access part contains ADs that define the addressing environment for the procedure. Included are ADs for the *current domain*, which was specified by the CALL instruction, and the AD for the *local constants segment*, which was specified in the called instruction object. The *global con-*

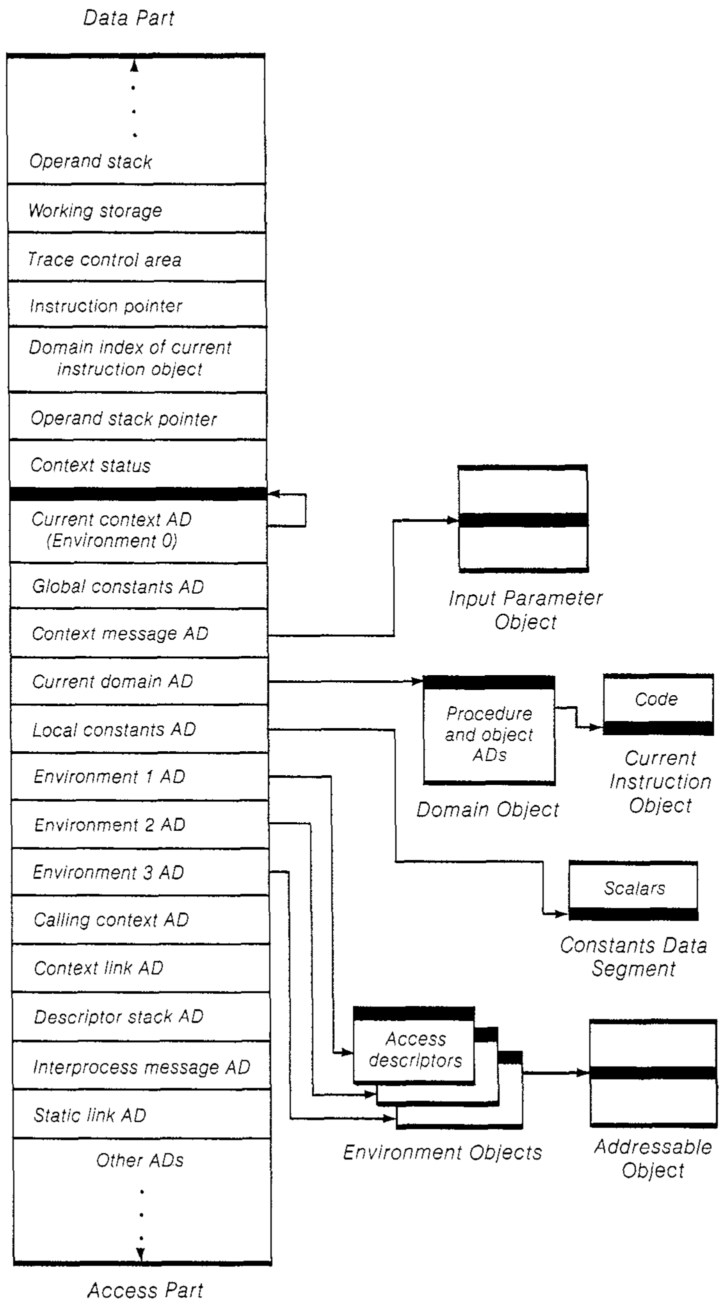

Figure 9-7: Intel 432 Context Object Representation

stants AD allows addressing of a process-wide data segment; the procedure explicitly loads this AD, if needed, through the COPY PROCESS GLOBALS instruction. The *calling context* AD addresses the caller so that a RETURN can be executed.

Interprocess communication is provided by instructions that send messages to and receive messages from port objects. Execution of a RECEIVE MESSAGE instruction causes the AD for the received message to be copied to the *interprocess message* AD in the context object's access part. In this way, the program has immediate addressability to the message. The *static link* AD, which follows the interprocess message AD in the context, is provided to support languages that use static lexical scoping.

9.4.3 Instruction Operand Addressing

The important context object ADs from the addressing point of view are those named *current context* and *environments 1, 2,* and *3* in Figure 9-7. As previously stated, an instruction must specify the location of an AD in order to manipulate any object. If the instruction manipulates one or more data elements, it must provide ADs for the segments containing those elements. In general, then, every instruction operand specifies one or more ADs that provide addressability to that operand.

At any moment during a procedure's execution, ADs specified by instructions must be located in one of four *environment objects*. Environment object 0 is the context object itself. Instructions can specify any of the ADs within the context object's access part; for example, to refer to the domain or the constants data segment. The three remaining environments, environments 1 through 3, are defined dynamically by the procedure. A procedure loads an AD for any object into the environment slots in the context object to make ADs in that object addressable. The ENTER ENVIRONMENT instructions are provided for this purpose.

Therefore, to address an AD, an instruction specifies one of the four environment objects and an index to an AD in the object's access part. Environment 0 is the context access part itself, which is self-addressed through the current context AD in the context object. Environments 1 through 3 are addressed through the three environment ADs in the context object. An instruction reference to an AD in one of the four environments is called an *access selector*. Figure 9-8 shows the three access selector formats. The low-order two bits in each selector spec-

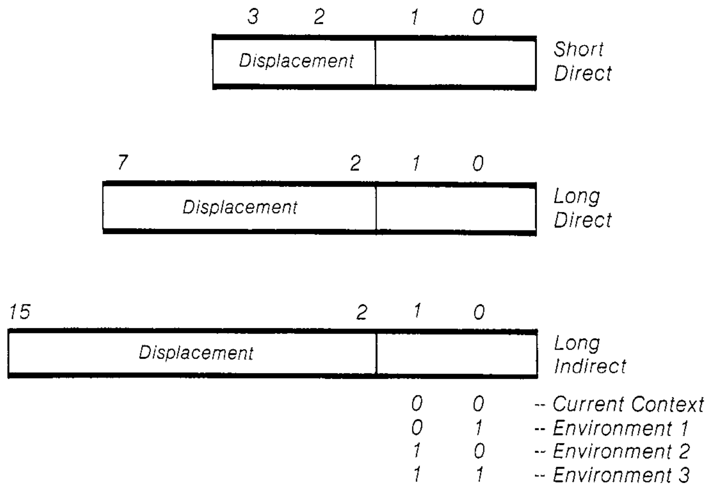

Figure 9-8: Intel 432 Access Selector Formats

ify the environment object; the three formats allow for 2-, 6-, or 14-bit displacements to an AD in the selected environment.

The four environment segments thus provide efficient addressing of ADs. An instruction can specify an immediate 4- or 8-bit access selector describing the location of an AD for an operand. Or, it can specify the location of a 16-bit access selector located in memory or on the stack. The short direct format efficiently addresses any of the first four ADs in any of the four environments. This includes the ADs for the global constants, context message (calling parameters), and current domain within the current context. All of the processor-defined ADs within the context object's access part can be addressed using an 8-bit access selector.

9.4.4 Context Allocation

On an earlier version of the Intel 432 architecture, each CALL instruction caused dynamic allocation of the memory segment in which the new context object was constructed. Because this allocation was time-consuming, the latest version of the Intel 432 supports preallocation of contexts on a per-process basis. The operating system allocates a linked list of fixed-sized context object segments to each process. The contexts are linked through the *context link* field in each context object.

When a call occurs, the processor reads the context link field to find the AD for the next context object to use. The length of this object is compared with the length fields stored in the called instruction object. If the instruction object requires a context object larger than the preallocated size, a fault will occur. The operating system can then allocate a context of the needed size. Or, if the context link is null, indicating that the preallocated contexts have been consumed, a fault will allow the operating system to perform additional allocations. Otherwise, the hardware quickly constructs the new context object from the linked segment.

9.4.5 Parameter Passing

Parameter passing on the Intel 432 is associated with the preallocation of contexts and is handled by software. In addition to the default context object size, associated with each process is a default data part size and access part size of a *parameter segment* to be passed between contexts on procedure calls. However, instead of allocating a separate parameter segment, an area of the data part and access part of each context object is reserved for parameter passing. When the operating system constructs the linked list of contexts, it places in the *context message* field of each context, an AD for a *refinement* of the *previous* context object. This refinement provides addressability to the parameter data and access fields as if they were a single contiguous segment.

Figure 9-9 illustrates how a procedure accesses parameters passed by its caller. The calling procedure places its data and access parameters in the predefined parameter fields of its context object. The operating system had previously created a refinement object descriptor for these parameter spaces and placed an AD for the refinement in the next context object. When the call occurs, the called context can access its parameters through its context message AD.

9.5 Abstraction Support

The principal goal of the Intel 432 is support for object-based programming. As previously described, the Intel 432 provides a set of basic system object types. Each of the system object types is controlled by a type manager that is implemented partially in hardware—through a set of type-specific instructions—and partially in operating system software.

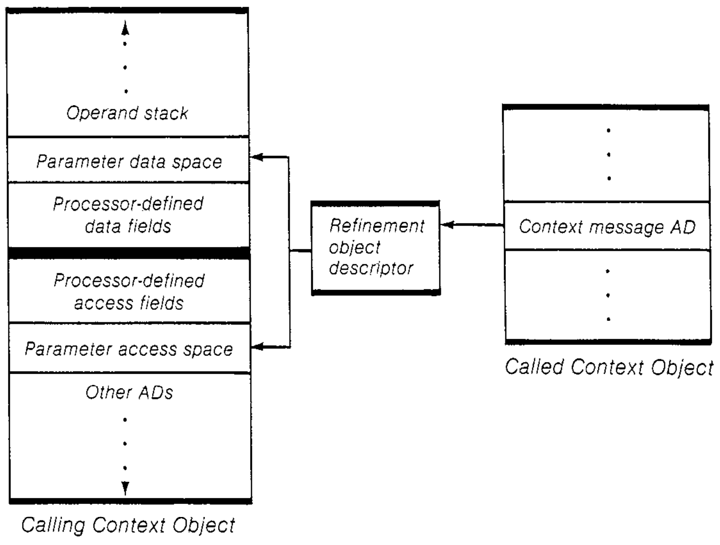

Figure 9-9: Intel 432 Parameter Passing

To extend the set of basic types, the Intel 432 provides mechanisms for the creation of programmer-defined types and programmer-defined type managers. Since all objects are accessed through a high-level language, the programmer uses the same interface when dealing with system objects and with programmer-defined objects. A programmer is free to create new types and type managers, adding to the set of available abstractions.

There are three system object types involved in type manager support:

- the domain, which defines the procedures and objects local to a single module of the type manager,
- the Type Control Object (TCO), which is used in creation of system and programmer-defined objects, and
- the Type Definition Object (TDO), which defines a particular type manager.

This section describes the use of these objects for the creation and manipulation of system and programmer-defined objects.

9.5.1 Domains and Refinements

A *domain* object defines a collection of procedures and associated objects accessible to those procedures. By using the 432 refinement mechanism, a programmer can create a protected procedure environment with a domain object. That is, a programmer can construct a set of callable procedures that will have access to objects not available to their callers.

Figure 9-10 shows a domain that consists of a collection of procedure ADs and object ADs. To construct a protected subsystem, the creator of the domain divides the domain into two sections: a public section and a private section. The public section consists of ADs for procedures that will be callable by users of the domain. The private section consists of ADs for procedures and objects that will be available only to called procedures executing within the domain.

Through the CREATE REFINEMENT instruction, the domain's owner constructs a refinement of the domain that addresses only the public section—the section that will be visible to users of the domain. The CREATE REFINEMENT instruction returns an AD for this refinement. The AD for the domain refinement can be made available to other programmers, who can use this AD to call any of the public procedures. However, a possessor of this refinement AD has access only to the domain's public part.

This use of domain refinement creates a protected subsystem because of the action of the CALL instruction. When a CALL instruction is executed, the hardware places an AD for the called domain in the new context object, where it is accessible to the called procedure. The hardware *always* loads an AD for

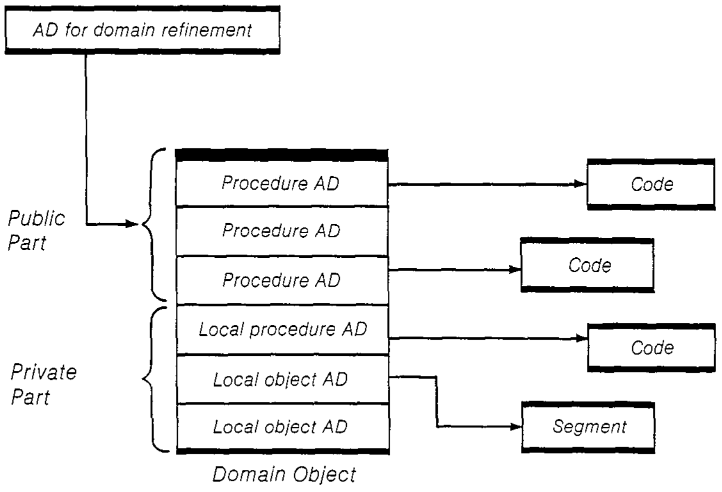

Figure 9-10: Intel 432 Domain Refinement

the *complete* domain, even if the CALL was made through a refinement. Therefore, a procedure invoked through a refinement of a domain will have access to all of the ADs in its domain through its context object. Once executing, the procedure can manipulate private data objects or call private domain procedures.

9.5.2 Creation of Typed Objects

The Intel 432 supports two kinds of object types: system types and programmer-defined types. The system types were listed previously in Table 9-1; instances of system types are identified by the 8-bit system type field in their object table object descriptors. Two of the system types are *generic object,* which is a basic segment object with no special attributes, and *dynamic object,* which is an object controlled by a programmer-defined type manager.

Typed objects of any kind are created through the CREATE TYPED OBJECT instruction. Execution of the CREATE TYPED OBJECT instruction requires possession of the AD for a *type control object* (TCO). A TCO permits its possessor to create and manipulate objects of a specific type. The data part of a TCO is illustrated in Figure 9-11.

Creation of a system object (with the exception of generic objects) requires possession of a TCO whose object type field

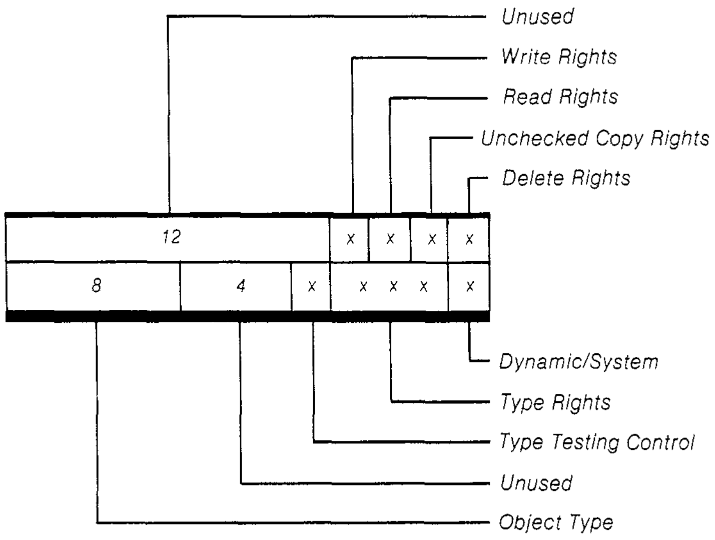

Figure 9-11: Intel 432 Type Control Object Data Part

contains the 8-bit type value of the system type to be created. In addition, the dynamic/system bit (bit 0) of the TCO must indicate that the TCO is for a system object. TCOs for the creation of specific system object types are constructed by the operating system and given to the operating system type managers for those types. The type manager for a system object is privileged only in its possession of the TCO for its type.

Possession of a TCO for a specific type also allows the type manager to execute an AMPLIFY RIGHTS instruction for objects of its type. In this way, the type manager can return restricted ADs to its clients. These restricted ADs cannot be used to access the objects to which they refer. When a client returns an AD to a type manager as a parameter, however, the type manager can use its TCO to amplify the rights in the AD. Given an AD for an object and an AD for a TCO with matching type, the AMPLIFY RIGHTS instruction ORs the rights bits specified in the TCO with the rights in the object AD, creating an AD with additional privileges. If the TCO and AD types do not match, the AMPLIFY RIGHTS instruction will cause a fault.

9.5.3 Programmer-Defined Types

To build a private type management system, a programmer obtains a *type definition object* (TDO) from the operating system. A TDO has no processor-defined fields, although its access part will typically be used to hold ADs for the domains that implement the type manager. The basic function of the TDO is to be the representative "type" for its objects. That is, while the type of a system object is specified by an 8-bit system object type field, the type of a dynamic object is specified by an AD for a TDO. All objects created by a specific type manager have an image of the AD for the type manager's TDO stored in their object table object descriptor (shown as TDO-AD in Figure 9-3).

Once a type manager has obtained a TDO, it then obtains a TCO from the operating system for its type. This TCO will be for a dynamic object, as specified in its system type field and in the dynamic/system field. A TCO for a dynamic object contains an additional field—a single AD in its access part. This is the AD for the defining TDO. When the type manager executes a CREATE TYPED OBJECT instruction to allocate a segment for the object's representation, it specifies its TCO and the size of the segment to allocate. The hardware copies the TDO access descriptor from the TCO into the object descriptor for the new segment, thereby "typing" the segment. Figure 9-12 shows this addressing structure; the object descriptor for the new dynamic object contains the physical storage information for the object and the AD for the TDO.

The programmer-defined type manager, like the system object type manager, protects its objects using restriction and amplification. When a client requests the creation of a new object, the type manager creates the object using the CREATE TYPED OBJECT instruction. The type manager then initializes the object appropriately and uses the RESTRICT RIGHTS instruction to produce an AD to be returned to the client. This AD does not allow direct access to the object. When the client later specifies this AD as a parameter, the type manager amplifies rights in the AD to regain access to the object's representation. Once again, the key to amplification is the possession of a TCO. The type manager executes an AMPLIFY RIGHTS instruction specifying its private TCO and the AD for the object. If the TCO and the object descriptor for the object both contain the same TDO AD, the instruction will amplify the rights in the object AD.

It is not necessary for programs to maintain ADs for all possible type managers. Given an AD for an object, a program can execute the RETRIEVE TYPE DEFINITION instruction; this instruction returns the AD for the TDO associated with the object. With the TDO AD, the program can access the AD for the domain implementing the type manager and can call type management procedures available through that domain. The domain AD stored in the TDO will typically be a refinement of the type manager's domain.

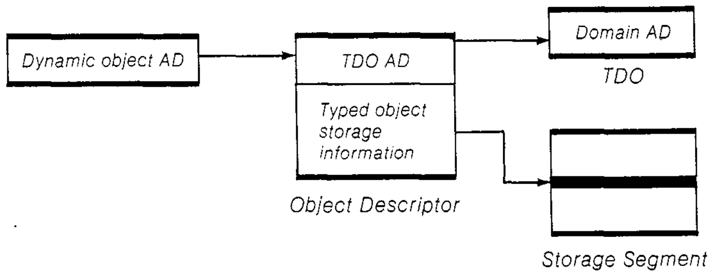

Figure 9-12: Intel 432 Dynamic Object Addressing

9.6 Storage Resources

Previous sections have described the creation of storage segments through the CREATE OBJECT and CREATE TYPED OBJECT instructions; however, they have not described the mechanism by which primary memory is allocated. The abstraction of primary storage is encapsulated in Intel 432 *storage resource objects*. A storage resource object (SRO) is a system object from which memory is allocated. Every memory allocation instruction specifies, either explicitly or by default, an SRO from which its primary memory is taken.

Figure 9-13 illustrates the structure of an SRO and its associated objects. The representation of an SRO consists principally of the AD for a *physical storage object* that describes a pool of available primary memory, and an AD for an object table. Each storage specifier in the physical storage object contains the primary memory address and size of a single contiguous block of free system memory. Initially, each physical storage object has one storage specifier for a single large block. As storage is dynamically allocated and deallocated from an SRO, its memory becomes fragmented and new storage specifiers must be created to address the discontiguous pieces.

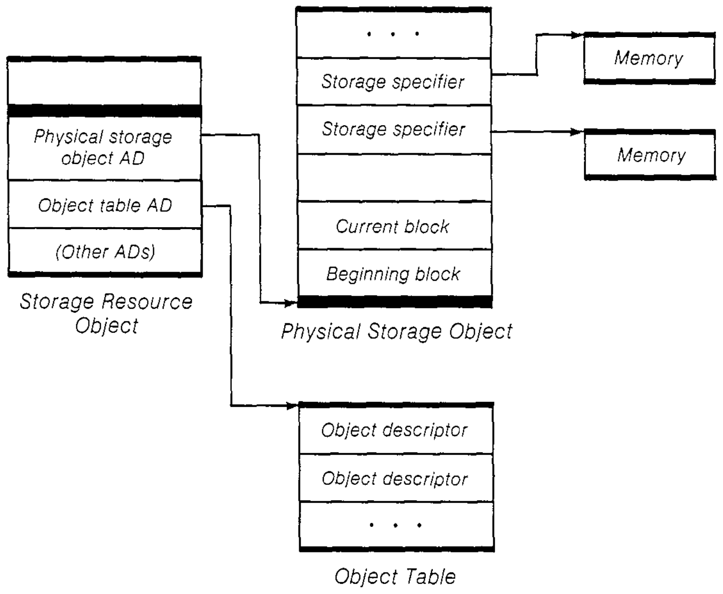

Figure 9-13: Intel 432 Storage Resource Object

When a program executes a CREATE OBJECT instruction, it specifies an SRO from which the storage is to be taken. The hardware allocates primary memory on a rotating first-fit basis from the SRO's storage specifiers. After allocating the memory, the hardware allocates an object descriptor for the new object in the SRO's object table; an AD is returned that addresses the object through that object descriptor.

The SRO in Figure 9-13 is known as a *global heap SRO* and is used to allocate relatively long-lived objects. Storage allocated from a global SRO can be returned at any time. The SRO's object table contains a descriptor that is the head of a list of unused object descriptors in the table. This list is used both for locating an empty table slot when an object is created and for returning an object descriptor when an object is destroyed. Returned storage is either combined with an adjacent free block in the SRO, or a new storage specifier is constructed to address it.

Global heap SROs provide great flexibility for dynamic storage allocation. The disadvantage of global heaps, however, is that they require garbage collection for deallocation of storage. Although the overhead of garbage collection is acceptable for long-lived objects, it is prohibitive for short-lived objects. In particular, most objects created during the lifetime of a procedure could be more efficiently deallocated when the procedure terminates. For this reason, the Intel 432 provides a second type of storage resource called a *local stack SRO*. A local stack SRO supports efficient allocation and deallocation of short-lived storage during the lifetime of a procedure.

A local stack SRO is not a separate object, but is associated with a process object. Each process object contains a local stack SRO, which consists of an AD for an object table and an AD for a physical storage object. This physical storage object is similar to that shown in Figure 9-13; however, it contains a *single* storage specifier for a *single* storage block. This storage block and the associated object table are used in a stack-like (LIFO) fashion for allocation of short-lived local storage. The local object table does not use a free list; instead, object descriptors are allocated consecutively.

During a procedure invocation, each short-lived object is allocated from a local stack SRO; each new object receives the next contiguous object descriptor and the next contiguous section of the storage block. When the procedure returns, all of the objects and object descriptors for short-term objects created by the procedure can be deallocated. This deallocation is simple when compared with global heap deallocation because both the object table and storage block are managed as stacks. All of the short-term objects and descriptors allocated during a procedure call can be quickly deallocated by returning the object table and physical storage objects to their pre-call states.

Local stack SROs are therefore more efficient for allocation and deallocation than global heap SROs, although they cannot accommodate objects of different lifetimes. The more difficult problem presented by local stack SROs is the control of ADs for local objects. Objects allocated from global heap SROs are only deallocated by a garbage collector. The garbage collector ensures that no ADs remain for an object before its storage and object descriptor are deallocated. If an object with an existing AD were deallocated, the AD would become a dangling reference. For example, suppose that AD X addresses object Y through object descriptor Z. If object Y and object descriptor Z are deallocated while X still exists, AD X will be a dangling reference. Eventually, object descriptor Z will be reused to address a newly created object, and AD X could be used erroneously to access that object.

This problem is compounded in the case of local stack SROs by the rate at which object descriptors are reused. An object descriptor deallocated by a procedure return will very likely be reused by the next procedure call. Therefore, the Intel 432 must be able to ensure that when a procedure returns, no ADs remain for short-term objects allocated during that call. To prevent such dangling references, the Intel 432 controls the propagation of ADs. The hardware prevents the storing of an AD into a segment whose lifetime is longer than the lifetime of the object addressed by that AD.

The lifetime of an object is determined by the *level number* stored in its object descriptor. Each process has a current level number; the level number is first initialized when the process is created and is incremented by one at each procedure call. When an object is created, the current level number is stored in its object descriptor. An attempt to copy an AD for an object created at level N into a segment created at level N-1 or lower will cause a fault. When an object allocated from a local stack SRO is destroyed on procedure return, the system can guarantee that no ADs for that object remain; that is, all of the storage into which the AD could have been copied must have been destroyed when the object was destroyed.

Any object that is to be passed to other processes or stored in a more global segment must be allocated from a global heap

SRO instead of the default local stack SRO. The architecture ensures that only correct copying of ADs takes place. The unchecked copy rights bit in Intel 432 ADs provides an optimization for the required level check. The unchecked copy flag indicates whether the object was allocated from a level-0 global heap. If so the level check can be avoided; otherwise, the level numbers in the object descriptors must be checked.

9.7 Instructions

The Intel 432 has a repertoire of about 225 instructions that operate on characters, integers, floating point numbers, and system objects. There are no general registers. Each context has a private operand stack that can be used for storing scalar temporaries. Scalar operands for instructions can be located either on the stack or in memory, and memory-to-memory operations are allowed.

One of the unique features of the Intel 432 is its instruction encoding. Instructions are bit-variable in length and can start on any bit boundary. The instruction pointer thus contains the bit offset into the current instruction segment, which can be up to 8K bytes in size. An instruction consists of up to four fields, as shown in Figure 9-14. The fields themselves are also variable-length and highly encoded.

The 4- to 6-bit *class* field specifies the number of operands and their sizes. For example, the class may indicate that an instruction requires three 32-bit operands or two 16-bit operands. Next, the 0- to 4-bit *format* field specifies whether each of the operands is (1) to be found on the stack or (2) to be specified explicitly by a reference in the references field, and (3) if specified explicitly, which reference corresponds to which operand. The *references* field specifies where the (one to three) operands are to be found. A stack operand requires no reference field entry, and a single reference may refer to two operands, as specified by the format field. For example, an operand that is both a source and destination requires only one reference field to define its location. Finally, the 0- to 5-bit *opcode* specifies the operation to perform.

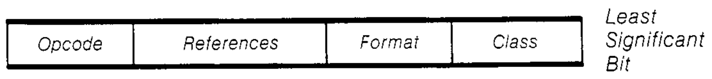

Figure 9-14: Intel 432 Instruction Format

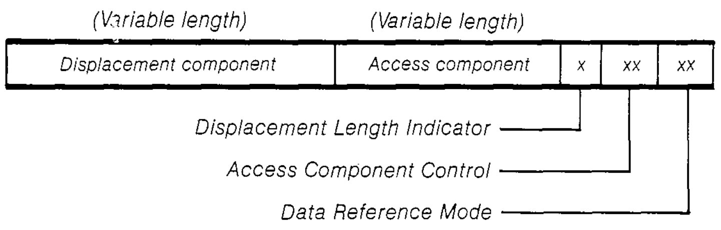

Figure 9-15: Intel 432 Reference Format

The references field is the most important with respect to object addressing and requires the most complex encoding. The size of the references field depends on the number of operand references and the addressing mode for each. An instruction operand can be either a *scalar operand* (e.g., integer, character, floating point) or an *object-level operand* (e.g., process, domain, message port). If the instruction operand requires a scalar, the reference specifies its location. If the instruction operand requires an object-level operator, the reference specifies an AD for the object.

The general format of a single reference is shown in Figure 9-15. The length and format of the variable-length access and displacement components are determined by the leading control fields. For example, in the case of a scalar operand, the instruction must specify two components needed to locate the scalar:

- the location of an AD for the object containing the scalar, and
- the displacement of the scalar within the object's data part.

The access component field locates the AD for the object; it is a 4- or 8-bit field whose format was shown in Figure 9-8. The displacement component, in the simplest addressing mode, is a 7- or 16-bit integer displacement.

Several addressing modes are allowed that provide for indirect specification of the access and displacement components; that is, the access and displacement specifications for the reference can be found in memory. For example, in the case of an indirectly specified displacement, the displacement field of the reference must itself contain an access and displacement part. Such general addressing modes provide for flexibility but can require many memory accesses in order to manipulate a single data element. Thus, a reference to an element of a dynamically allocated one-dimensional array might indicate:

- an access selector for the segment containing the array,
- the displacement of the array in the segment,
- an access selector for a possibly different segment containing the array index, and
- a displacement of the index in this second segment.

Many options are provided for each part of the specification and, in general, commonly occurring options can be efficiently encoded. Stack operands save the most instruction space because they require no reference field bits. Space can also be saved in the reference field if operands are located at the start of a segment because this requires no offset. There is a large variance in instruction size—a three-operand instruction can take from 10 to more than 300 bits, depending on where the operands are to be found.

9.8 Discussion

The Intel iAPX 432 is certainly one of the most sophisticated architectures in existence. By using the object-oriented approach throughout the development effort, the Intel 432 designers have produced an extremely uniform and tightly-integrated hardware/software system. This uniformity of hardware and software systems is due to the use of a consistent philosophy. Everything in the Intel 432 is an object. All objects have associated types that specify the operations that can be performed on those objects. Some objects have hardware-defined operations while others do not. However, from a language viewpoint, all objects are accessed in the same way.

All objects, whether hardware-supported or not, are controlled by type manager modules. Programmers can freely add new types to the system by creating new type managers. The mechanisms of domain refinement and type definition object provide a way for type managers to exhibit privilege over their objects and the environments in which their procedures execute. A type manager can restrict and later amplify privileges in ADs for its objects by using a privately held type control object. By permitting client access to type management procedures through a refinement, an executing type management procedure can obtain access to a richer environment than its caller.

There are no special privileges in the Intel 432 system. The mechanisms used by programmer-defined type managers are identical to those used by operating system type managers.

In addition, the concept of programmer-defined type is an integral part of the addressing system, in that each object descriptor has space for a TDO access descriptor. Few previous systems have allocated sufficient space to integrate programmer-defined objects so tightly into the hardware architecture.

The designers of the Intel 432 have closely adhered to the concept of separate procedure address spaces, as presented in the Dennis and Van Horn model. Each procedure invocation causes the construction of a new context object that defines the procedure's addressing environment. This is true even of calls to procedures within the same domain, for which both procedures will have access to a similar set of objects.

Although an initial implementation of the Intel 432 had separate data segments and capability segments, the current version supports segments with both a data part and a capability part, as on STAROS. The object descriptor addresses the barrier between the two parts and contains the size of each part. Refinements are provided that allow the construction of what appears to be a single two-part segment from contiguous subsets of the two parts of a segment. Two-part segments do not allow the flexibility provided by tagging; however, they effectively reduce the number of needed segments by a factor of two. This affects performance by reducing the number of segment allocations required to create a new object.

Another performance enhancement has resulted from the preallocation of context objects. When a procedure call occurs, the hardware simply follows the context link to the next waiting context object. That object has already been prepared with a refinement of the parameter space in the calling context. In addition, the use of local stack SROs for allocating short-lived objects reduces the need for garbage collection. These changes to the CALL instruction have reduced its execution time from 300 microseconds on early prototypes to under 100 microseconds on the current version of the Intel 432.

Capabilities on the Intel 432 are 32 bits in size. Of this, 24 bits form the actual ID or address part of the capability. Thus, there are a maximum of 2^{24} objects at any time. Segments have a maximum (data part) size of 64K, which is relatively small when compounded by the lack of cross-segment addressing. That is, due to the structure of Intel 432 addresses, it is not possible to transparently cross a segment boundary by incrementing the address. Therefore, the compiler must produce special code for objects whose data parts cannot be held in a single segment. This is true also of procedures that are larger than 8K bytes, although this is probably a rare occurrence.

The instantaneous address space of the Intel 432 is 2^{32} bytes, based on the use of the four environment ADs stored in the context object. These environment ADs act somewhat like capability registers, and, in fact, the Intel 432 GDP has special internal registers to hold their values. At any time, ADs in use by a procedure must be stored in one of the four environment objects. To access objects located indirectly through the environments, the procedure must explicitly traverse the structure, loading ADs for each level in the tree.

The Intel iAPX 432 is an ambitious system in terms of both architecture and implementation. It is particularly impressive when considered in relation to the other available single-chip processors. But it is fair to say that the Intel 432 has not been a commercial success. Although there were over 100 Intel 432 systems in the hands of universities and customers by 1983, this is a small number by microprocessor standards. The commercial problems of the Intel 432 are probably due in part to premature (and somewhat overzealous) marketing of the product before its implementation and software were ready. The initial version of the Intel 432 had performance problems, which have been corrected to some extent by later versions of the architecture. Still, whether or not the Intel 432 succeeds as a product, it has opened a new era of microprocessor design.

9.9 For Further Reading

The book by [Organick 83] presents the most comprehensive description of the Intel 432. It describes the major components of the Intel 432 system—the Ada compiler, the iMAX operating system, and the iAPX 432 hardware architecture—and provides Ada programming examples as well. In the published literature, the paper by Pollack, Kahn, and Wilkinson describes the philosophy behind the Intel 432 object filing system [Pollack 81], and the paper by Cox, Korwin, Lai, and Pollack discusses the Intel 432 interprocess communication facility used for both message passing and process scheduling [Cox 83]. Storage management on the Intel 432 is discussed in [Pollack 82]. The Architecture Reference Manual [Intel 81, Intel 82] contains detailed descriptions of the Intel 432 architecture.