A VLSI architecture for enhancing software reliability

Kanad Ghose  
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
http://lib.dr.iastate.edu/rtd/9345

This Dissertation is brought to you for free and open access by Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
Accessing the World’s Information since 1938
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
A VLSI architecture for enhancing software reliability

Ghose, Kanad, Ph.D.
Iowa State University, 1988

Copyright ©1988 by Ghose, Kanad. All rights reserved.
A VLSI architecture for enhancing software reliability

by

Kanad Ghose

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved: Members of the Committee:

Signature was redacted for privacy.
In Charge of Major Work
Signature was redacted for privacy.
For the Major Department
Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa

1988

Copyright © Kanad Ghose, 1988. All rights reserved.
TABLE OF CONTENTS

1. INTRODUCTION .............................................................................................................. 1
2. CAPABILITY-BASED ADDRESSING ................................................................................. 25
3. MAJOR DESIGN DECISIONS .......................................................................................... 38
4. THE CAPABILITY MECHANISM AND RELATED INSTRUCTIONS ................................. 55
5. OTHER ASPECTS OF THE ARCHITECTURE ................................................................. 139
6. ASSESSMENT AND CONCLUSIONS ............................................................................. 178
7. REFERENCES .................................................................................................................. 199
8. ACKNOWLEDGEMENTS ............................................................................................... 205
9. APPENDIX I: SELECTED MICRO-PROGRAMS ............................................................... 207
10. APPENDIX II: ABBREVIATIONS AND GLOSSARY ..................................................... 227
1. INTRODUCTION

The purpose of this dissertation is to propose an architecture which will be instrumental in enhancing the quality of software by incorporating mechanisms for efficiently detecting a wide variety of run-time errors and by providing efficient support for linguistic constructs and system functions that promote software reliability. The proposed architecture is thus one way of combating the so-called software crisis. In this chapter, we first describe the software crisis and identify the major factor behind this crisis. We then describe the linguistic constructs and system functions available for enhancing the quality of software and then proceed on to criticize the support provided, or the lack of support, for such constructs and functions in traditional and innovative architectures. In doing so, we point out some major shortcomings in the traditional and innovative architectures that limit or discourage the use of these linguistic constructs and systems functions. These shortcomings in existing architectures provide the motivations for designing the architecture proposed in this dissertation. We conclude this chapter by describing the objectives of this dissertation and its organization.

1.1 The Software Crisis

Despite advances in programming language design and programming methodologies, software quality assurance continues to be a critical issue [1], [30]. Software is error-prone and likely to be unreliable, thus requiring maintenance over the course of its useful lifetime. Maintenance, in this respect, refers to correcting problems in the original design and
specification of the software as bugs are discovered. It also subsumes the act of upgrading the software in order to meet evolving requirements. Today, a significant fraction, typically 75% to 80% [1], of the cost of a software system is for maintaining it. To compound this problem, the software components of a typical system have become increasingly sophisticated and complex during recent years. In clear contrast to this, the cost of hardware components and their failure rates have drastically fallen with the advent of VLSI technology. As a result, the cost of the software components of a typical computer system have dominated the cost of the hardware components. Currently, two-thirds to three-fourths of the cost of a computer system is for its software components [1], [30]. This phenomenon has been referred to as the software crisis [30]. The unreliability of software is therefore a major contributor to the software crisis.

This software crisis is undesirable for economic and other practical reasons. One obvious way of resolving the software crisis is to provide mechanisms that will enhance the reliability of software.

1.2 Techniques for Enhancing Software Reliability

In this section, we take a detailed look at the techniques employed in traditional systems for enhancing software reliability. We group these techniques and mechanisms into the following categories:

1. **Avoidance techniques**, representing mechanisms and strategies employed at program design time, for reducing errors in the program.
2. *Static analysis*, representing techniques and mechanisms applied to the program before it is run, for detecting some potential errors in the program.

3. *Dynamic techniques*, which are invoked during program execution for detecting and handling potential errors in the program. This category also includes the facilities for program tracing and high-level run-time error reporting.

In the following sections, we look at specific techniques/mechanisms in each of these three groups.

### 1.2.1 The avoidance techniques

The techniques for avoiding programming errors enforce a programming discipline, either (i) explicitly through appropriate linguistic constructs and/or other mechanisms or (ii) implicitly as good programming style. The enforcement of a programming discipline is expected to reduce programming errors.

The important linguistic constructs for enforcing a programming discipline in imperative style programming languages are as follows:

a) *Use of strong typing in the language.* In a strongly-typed language errors related to implicit type conversions (either static or run-time) are absent.

b) *Use of type abstraction.* Many programming languages, such as Pascal, Ada (U. S. Department of Defense [63]), Modula (Wirth [65]), incorporate facilities for constructing new types out of already-existing or already-defined types, thus providing a facility for type abstraction. The type abstraction facility lets a programmer define a type that is more appropriate for a particular application. Two common type constructors found
in all of the languages mentioned above include the subrange constructor and the enumerated type constructor. The subrange constructor lets the programmer define a type in which values are restricted to a contiguous subrange of values for an already-existing or already-defined type. As an example, in Pascal one can define the following type

```pascal
type int_subrange = 21..200;
```

This defines a new type 'int_subrange', whose corresponding values are a subrange of integer values, between 21 and 200, both inclusive. Typical usage of variables of this type would be as loop counters, array indices, etc. The enumerated type constructor provides a facility for enumerating all possible values corresponding to a type. Again, as a Pascal example, one can define the following:

```pascal
type enum_type = (a,b,cd,ef)
```

This defines a new type 'enum_type', whose corresponding variables are restricted to the values a, b, cd, and ef only. The use of subrange and enumerated types guarantee consistency in data values of these types by signaling occurrences of forbidden values. In some cases, enumerated types also improves program readability.

c) **Use of control abstraction.** Many modern programming languages employ control constructs with a higher abstraction level, that lead to well-structured programs. The control flow constructs of Pascal, (excluding to go to) serves as a particularly good example of this philosophy -- each control construct is of a sufficiently high-level and features one well-defined entry point and one well-defined exit point.

d) **Use of Abstract Data Types.** The mechanism of data abstraction, as found in languages like Ada [63], CLU (Liskov, Snyder and Atkinson [42]), Modula [65] and SIMULA
(Dahl and Nygaard [11]) basically restricts the way in which a data item or structure can be altered by a program. This is done by defining procedures and/or functions to serve as an interface between the data item/structure and the rest of the program. These interfacing procedures are the only means of manipulating the data item/structure from the rest of the program. Direct access to the underlying representation of the data structure/item is disallowed, thus encapsulating the data item/structure very effectively. The interfacing procedures thus define a high-level abstraction of the underlying implementation, and effectively makes the resulting abstract data item/structure a strongly-typed entity. The interfacing procedures are the only type-specific operations allowed on the abstract data type. Private procedures and types, not accessible from the rest of the program, can also be used in implementing the interface. Figure 1.1 shows the declaration of an abstract data type, 'Integer_Stack' in Ada. The 'package specification' part lists the interfacing procedures and functions (Push, Pop, Top, Initialize, Is_empty, Is_full) as well as a private type, Istack. The private type 'Istack' is not a part of the interface -- it is used only by the compiler. The actual implementation of the interfacing procedures and the stack are in the 'package body' part, as shown in Figure 1.1. In particular, the fact that the stack is implemented as an array is not visible to the users of this package. Inadvertent or deliberate errors due to direct modification of the underlying implementation are impossible. Axiomatic techniques, although somewhat limited in terms of applicability, also exist for verifying the correctness and completeness of the interfacing procedures [24], making the concept of data abstraction a very useful and powerful one. The so-called object-oriented languages like Smalltalk and CLU [41], [42] also support mechanisms for data abstraction in conjunction with the notion of
Package Integer_Stack is
  type Istack is private;
  procedure Push (S: in out Istack;
                  item: in integer);
  procedure Pop (S: in out Istack;
                 item: out integer);
  procedure Initialize (S: in out Istack);
  function Is_empty (S: in Istack) return Boolean;
  function Is_full (S: in Istack) return Boolean;
  function Top (S: in Istack) return integer;
private
  stksize: constant: integer := 100;
  type Istack is
    record
      top: 0..stksize;
      sarray: array (1..stksize) of integer;
    end record;
end;

Package body Integer_Stack is
  procedure Push (S: in out Istack;
                 item: in integer);
  begin
    S.top := S.top + 1;
    S.sarray(S.top) := item
  end Push;

  --- Complete definition
  ----of other procedures

  end Top;
end Integer_Stack;

Figure 1.1 The Abstract Data Type Integer_Stack
referential transparency.

e)  **Programming using modules.** Many programming languages provide facilities for breaking a large program into a number of smaller units called modules, which can be independently designed and tested. This requires a specification of the interface to each module that describes how the module is supposed to behave with respect to the other modules. Module-based programming has a number of advantages from the perspective of software reliability. First, modules can be independently designed and tested, facilitating team programming. Second, if the interface of a module is held constant, the module can be altered or replaced without affecting the rest of the program, thus allowing improved maintainability and code reusability. Third, modules represent a logical breakup of a large program into smaller tasks, and being smaller in complexity are less likely to be in error.

f)  **Small Protection Domains.** The concept of a 'protection domain' or a 'sphere of protection' has been well formulated and developed in the context of protection issues in an operating system [12], [37], [67]. Basically, a protection domain represents an abstraction of the set of objects accessible to a computation and the access rights allowed on these objects. A protection domain can thus be represented as a set of object/access-rights pair [37]. A computation can be divided into several phases, and a protection domain corresponding to each phase can be set up. In protection domains constructed following the 'need-to-know' principle [12] (also called the principle of least privileges [67]), the access mode allowed on every object in the domain are the minimum set of access modes that the computation absolutely needs to the object. As an example, if the computation needs to read data from an object A in a domain D,
where D is constructed following the need-to-know principle, the access mode needed
by the computation to the object is read-access only; there is no need to give the com-
putation any more access privilege (such as write-access, execute-access). Two advan-
tages result when protection domains constructed following the need-to-know principle
are small (i.e., each domain has a small number of objects in it). First, error propaga-
tion is limited. An error in one protection domain affects only the objects in that
domain, and only in the specified access mode. Second, a logical and finer granularity
of protection is achieved if a program unit, such as a procedure has a protection domain
associated with it. Traditionally, the use of protection domains had been restricted to
the sharing of computational objects across users/programs in a system. One can also
adapt the same concept and allow the protection of objects within various parts of a
program as well, and thus enhance software reliability. Linguistic constructs are avail-
able for specifying such protection domains [66], [67].

All of the above linguistic features explicitly enforce a programming discipline whose
purpose is to minimize errors in a program. An important additional source of software
unreliability stems from incorrect or vague specifications for the program itself. There are,
however, explicit mechanisms for formally specifying the requirements of a program [30],
[64] for combating this aspect of software unreliability.

There are also a variety of implicit techniques to help improve the readability, main-
tainability and error rates. These range from giving meaningful names to variables to the use
of hierarchy charts, comments and other forms of documentation. Such details can be found
in [30], [64].
1.2.2 Techniques based on static analysis

These techniques can be grouped into the following sub-classes:

a) Static Error detection, during compilation (or pre-processing). Errors detected are mostly of syntactical nature, as well as semantics-related. Errors in the second category are related to type violations, scope-rule violation and other inconsistencies between declaration and use.

b) Flow Analysis [64], which is a form of pre-processing a program in order to trace use-definition correspondences, reachability and similar properties.

c) Proof techniques, for establishing the correctness (or partial correctness) of a program analytically, usually through stated post-and pre-conditions.

All of these techniques are either limited in their scope of usefulness or not sufficiently general. Since our focus will be eventually be not on the static techniques, we will not discuss them any further in this thesis.

1.2.3 Dynamic techniques

The dynamic techniques for enhancing software reliability entail the use of one or more of the following facilities/mechanisms.

• Run-time error detection facilities. A large variety of programming errors occur only during the execution of a program and thus cannot be detected at compile time. Examples of such errors include reference to an uninitialized location, reference using an invalid pointer, array reference using an invalid index, inter-module parameter inconsistency, subrange violation and so on. A list of the most common programming errors
presented in [13], [45] indicate that most of such errors are run-time errors.

- **Program Debugging Mechanisms** Frequently, when errors in a program cannot be located by a visual inspection of the code or by mentally simulating the execution of the code, an established technique for tracing program bugs is to trace the execution of the program, either continuously or at distinct points or when certain specified events occur. The specified events are typically, execution of a successful conditional branch, flow of control to a previously 'marked' point in the code, execution of a procedure call and so on. User-programmable facilities for program debugging through tracing are thus very useful.

- **Exception handling mechanisms.** An exception designates an error condition in a program. These exceptions range from simple arithmetic overflow/underflows to programmed exceptions such as software interrupts and program defined conditions. Many modern programming languages like Ada [63], Modula [65], CLU (Liskov and Snyder [41]) (and not-so-modern one like PL/1) provide the programmer with the facility of writing dedicated routines for handling specified exceptions. With such a mechanism available, the programmer can specify appropriate recovery actions as part of the program, rather than rely on the generic, system-defined handlers. Chances of error recovery are thus improved when the programmer has the ability to specify the actions to be taken on an exception.

Some languages, like Ada [63] provide the facility for nested (or hierarchical) fault handling. If the handler for an exception is not provided in the unit where the exception occurs, a corresponding handler specified in the closest enclosing unit is automatically used. If a handler is not available in an enclosing unit, the system-specified default
handler is invoked. Nested exception handling facilities thus provide a more structured approach to recovery from exceptions.

- **Mechanisms for software fault-tolerance.** Software fault-tolerance refers to the realization of a fault-tolerant system through the use of redundant code, possible (but not necessarily) assisted by redundant hardware. The major objective of software fault-tolerance is to overcome residual design errors and other transient errors in the software [3]. Most mechanisms for achieving software fault-tolerance will also handle less-severe, transient hardware faults [64]. Typically, in implementing software fault-tolerance, when a program segment (usually, a block or a procedure) is considered to have failed in producing the intended results, the state of the computation is restored to the one existing just prior to the entry to the failed segment. An alternate, standby program segment, that is functionally equivalent to the failed segment is then executed.

Where programming errors are the most likely reasons for the fault, the alternate segment typically uses a different algorithm/approach from that used in the faulty segment.

To date, two broad techniques for realizing software fault tolerance, (as described above) are well-known. One is the concept of ‘N-version programming’ [3] and the other is the concept of ‘recovery blocks’, as proposed by Horning and Randell in [26]. A number of variations and extensions of both schemes also exist [3], [56], [59], [60], [64].
1.3 Implementation in Traditional Systems

In this section, we describe the implementation of the avoidance techniques and the dynamic techniques for enhancing software reliability in traditional systems and provide a critique of these implementations, where appropriate. A major basis of the criticism rests on the execution efficiency of these techniques, since the execution efficiency will decide to a large extent how acceptable these techniques would be to programmers. We will therefore not discuss the static approaches to enhancing software reliability, since they have no effect on runtime performance.

1.3.1 Architectural characteristics of traditional systems

Traditional systems embody the so-called von Neumann architecture or its derivatives [6], [46]. The functional characteristics of the von Neumann architecture are significantly different from the general requirements of the software systems it is intended to support, a discrepancy that has been referred to as the 'semantic gap' [46]. The wide semantic gap between the von Neumann architecture (vNA) and imperative style high-level languages (HLLs) and operating system functions can be attributed to the following factors:

- HLLs have the concept of typed data elements and the distinction between data and instructions, in contrast to the 'typeless' nature of memory words in the vNA.
- HLLs embody the concept of data structures like arrays, lists, records and sets. The only counterpart to these on the vNA is a primitive storage structure in the form of a one-dimensional array of memory words.
• HLLs employ a high degree of control abstraction that regulate the control flow in a program and restricts a statement's accessibility to data. Many aspects of control abstraction such as block-structuring and the procedure call mechanism have no parallel on the vNA.

• The concept of abstract data types is one that combines data and control abstraction to regulate the way in which data can be manipulated by user-defined procedures/functions. The vNA has no counterpart to this concept. The semantic gap between the vNA and the operating system, on the other hand, can be ascribed to the following:

• The functional requirements of an operating system related to process and resource management have no counterpart on the vNA.

• Support for exception handling, a central role played by an operating system, is minimal on the vNA.

• Protection, an intrinsic issue in operating systems, is a concept totally alien to the vNA.

These shortcomings of the von Neumann architecture have detrimental effects on the performance of the system [6], [46] and the performance of the techniques for enhancing software reliability that can be implemented on such an architecture.

1.3.2 Software reliability enhancement mechanisms in traditional systems

In this section, we will look at the implementation of the avoidance and dynamic techniques of enhancing software reliability in traditional systems.
The von Neumann architecture does not embody mechanisms for detecting run-time errors in an efficient way. The only way to detect these errors in traditional systems is to use compiler-generated code that is executed whenever the possibility of an error exists. Such run time checks are prohibitively expensive in terms of their detrimental effects on run-time performance. An example of the overhead for compiler-generated run-time check is provided in [46]. This example involves the execution of the array assignment:

\[ A(I, J) = B(I, J) + C(J, I) \]

in PL/1, executing on an IBM S/370. When the compiler option for checking array subscript violations is enabled, the execution time for this assignment statement goes up by a factor of three. Because of the extreme overhead of run-time checks, they are left out of production code, more-often-than-not, leading to the possibility of unreliable software.

The support for language constructs and programming methodologies that enhance software reliability by imposing a programming discipline is minimal on the von Neumann architecture. As a result, these mechanisms are supported in a very inefficient way in terms of software implementations or sometimes not supported at all. Several examples can be given here. Subrange and enumerated types accrue the same order of overhead for checks against run-time violations, as do the run-time checks for array subscript violations. Data abstraction, while easy to impose by the compiler when all the units of a program are compiled together, is very difficult to enforce when abstract types are used across independently compiled modules. For the same sort of reason, it is difficult to check the consistency of parameters across independently compiled modules in a easy way. Since the von Neumann architecture does not rigidly enforce the interface specifications of modules, it is possible for modules to interact in obscure ways (for example, by implicitly inheriting the environment of
another module) and develop strange bugs [45]. As a final example, the concept of small protection domain is extremely inefficient for software implementation on traditional systems, so much so in fact that such an implementation of this concept was never made!

Support for exception handling on the von Neumann architecture centers around the interrupt mechanism, which is under the purview of the operating system. The implementation of user-specified exception handling on the traditional systems is therefore very inefficient and clumsy, since a context switch is involved when an exception is signaled via the interrupt mechanism. The implementation of the mechanisms for realizing fault-tolerant software is also very awkward and inefficient, although some experimental software implementations do exist [2], [39], [59]. A multi-processor embodying the von Neumann architecture has been proposed in [34] for supporting recovery blocks.

Implementing debugging tools in traditional systems is also a major problem, since the von Neumann architecture lacks the necessary mechanisms for generating traps on certain events which are extremely useful in debugging, such as access to a specified location in a specified access mode. Although much still remains to be done about debugging tools for use in traditional systems, their performance is not of serious concern, since debugging will always be a slow, interactive process.

Efficient support for the mechanisms for enhancing software reliability have come in the way of architectural innovations. We discuss some notable innovations in the next section.
1.4 Innovative Architectures

Since the 60s, a number of fundamental techniques have been proposed or implemented to reduce the semantic gap with respect to imperative high-level languages and operating systems in order to remove the problems associated with a wide semantic gap. The architectures that have resulted are as follows.

*Stack-based architectures*, whose basic storage model is a pushdown stack. These architectures provide efficient support for block-structured programming languages, the sub-program mechanism and interrupt-handling (an interrupt is simply an unexpected procedure call!). The stack-based instruction set also makes code generation from a postfix intermediate code a trivial task. An excellent assessment of stack machines can be found in [14] and [48].

*Tagged architectures* [17], where storage words are made self-identifying using tags and thus serve as type-specific value holders. These type tags are usually consistent with the base types supported in programming languages and operating systems. Tagging enables the use of generic instructions (thereby making compilation simpler) and help in enforcing type-specific operations on the base types. The machine-enforced consistency checks do not affect the speed of operations drastically and, being done as a rule rather than as an exception, improves program reliability significantly [46].

*Descriptor-based architectures* [46], [48], which employ special words, called descriptors, to hold information used by the machine to access structures efficiently and correctly. The formation and the use of a descriptor is controlled by the system, thus ensuring the integrity of structures. Descriptor-based architectures typically employ tags to distinguish a
descriptor word from a non-descriptor [46].

**Content-addressed architectures**, in which the storage model is an associative memory. These architectures are mainly application-oriented towards real-time searching-dominated problems, rather than being of a general-purpose nature. Foster provides an extensive presentation of these architectures in [19].

**Capability-based architectures**, where the storage model is a network of objects that are accessed as described in the following. These objects represent information grouped together for protection on a per-object basis. The only way to access an object is to use a special pointer to the object, called a capability to the object [12]. A capability to an object incorporates the system-wide unique name of the object and information that designates the access modes allowed on the object using this capability. Capabilities are made tamper-proof and un-forgable and a correspondence between an object’s name and its physical address is maintained by the system. By controlling the granting of capabilities to processes, a fine-grain protection system can thus be built. Capability-based architectures thus represent a generalization of the segmented model of storage. The advantages of capability-based addressing are well-described in [16] and the book by Levy [40] describes most of well-known capability-based systems in detail.

**Linked allocation-based architectures** [23], in which the storage model is a linked-list of tagged cells or cell groups. This storage model can effectively support a large variety of dynamic data structures, including those in non-imperative style languages like Lisp and others. SYMBOL (Laliotis [36]) was the first large-scale computer to embody this model of storage, as a direct-execution architecture for an imperative language.
The issue of software reliability and its relationship to the architecture was not a primary concern in most of these techniques used for reducing the semantic gap. A notable exception to this is the capability-based architecture, for which protection against programming errors was one of the driving motivations. In systems employing these architectures, the complexity of the systems software is lower, which has a positive effect on the reliability of the systems programs.

A general characteristic of these architectures oriented towards (imperative style) programming languages is the enhanced semantics of most of the instructions in the resulting instruction set. These machines thus fall under the category of Complex Instruction Set Computers, CISCs (Colwell et al. [9]). Most of the innovative architectural concepts presented above did not appear in isolation. For example, tags, descriptors and stacks were successfully used in the long-standing line of Stack Machines from Burroughs [48]; Tags, descriptors and capability-based addressing are used in Intel's innovative iAPX 432 microprocessor (Intel Corporation [29]) and in IBM's S/38 (IBM Corporation [28]).

1.5 Software Reliability Enhancement on Innovative Architectures

Many of the mechanisms for enhancing software reliability have close counterparts in some of the innovative architectures that make the performance of their implementation relatively acceptable. We will look at such implementations in this section.

In the tagged architectures and descriptor-based architectures, much of the type checking is done by the hardware at run time. Thus, many of the mechanisms for detecting common run-time errors (such as reference to uninitialized locations, subrange violations and
parameter inconsistencies) are very efficiently detected. SYMBOL [36] and SWARD (Myers [46]), for instance, are two architectures in this class. Both detect all of these three common run-time errors efficiently with little or no overhead. Of these two, only SWARD requires an explicit instruction for checking subrange violations. The tagged and descriptor-based architectures are also good at providing efficient debugging and error-reporting tools. Since tags preserve the data type at the storage level, core dumps produced by tagged machines are very intelligible [6], [46].

Capability-based architectures are particularly efficient in implementing many of the other mechanisms for enhancing software reliability. The C.mmp/ Hydra system (Wulf [67]), the Cm* /StarOS system (Gehringer, Siewiorek and Segall [20]), the CAP (Wilkes and Needham [66]), the Plessey PP 250 (England [15]), the Intel iAPX 432 [29], the IBM S/38 [28] and the SWARD architecture [46] are all capability-based machines and have the necessary mechanisms for implementing small protection domains and abstract data types, however, with varying degrees of efficiency. Capability-based architectures are also capable of enforcing the specified interface across independently compiled modules detecting run-time errors involving invalid pointers. We will look at these aspects of the capability mechanism in more detail in the next chapter.

A number of architectures have also been proposed to support the concept of object-oriented programming [32], [59], [62]. These architectures are however limited in their ability to provide support for many of the mechanisms for enhancing software reliability other than those directly related to the object-oriented programming paradigm.

Although the implementations of the above-mentioned mechanisms for enhancing software reliability are considerably more efficient than their implementations on traditional
system, they are still unacceptably slow. For example, a switch from one protection domain to another via a protected procedure call on the C.mmp/Hydra takes 70 milliseconds on the average, discouraging its use by programmers [67]. The Intel iAPX 432, a comparatively recent design suffers from the same malady, although the only type of call supported on this machine is a protected procedure call. Existing capability-based architectures, although useful in realizing many of the goals of software reliability enhancement, suffer from this and several other problems. We will analyze the source of many of the problems in capability-based architectures in the next chapter in detail.

The other innovative architectures have little, if any, use in isolation in providing most or all of the mechanisms for enhancing software reliability. Special-purpose add-on hardware for realizing some of the mechanisms for enhancing software reliability, such as an unit that provides limited support for recovery blocks [39] and hardware for interactive real-time debugging [38] have also been developed. These special-purpose units are also limited in their ability to enhance the overall software reliability when used in isolation.

To date only two architectures, the Intel iAPX 432 [29] and the SWARD system [46] have been motivated by the need for supporting mechanisms for software reliability enhancement. We will discuss specific details about these systems as occasion demands.

1.6 Implementation Issues

The architectures that have been proposed and/or implemented for enhancing software reliability along the lines stated earlier have an instruction set characterized by a large variety of instructions and instructions with considerably enhanced semantics. Consequently, these
architectures would be classified as complex instruction set compilers (CISCs) [9]. Prevailing and near-term technological limitations impose severe throughput-based performance limitations on the VLSI realization of CISCs for a number of reasons. First, the CISC instruction set is too unwieldy to be put on a single VLSI chip -- the CISC architecture is therefore realized, typically, in a distributed fashion over a number of VLSI chips. A performance penalty is then imposed in the form of interchip communication delays, which may be significant since off-chip delays dominate on-chip delays by at least an order of magnitude in all technologies (Nmos, Cmos, GaAs, etc.) [43], [51]. Second, in order to localize and accommodate as much of the processing function as one can on a single chip, serious compromises may be made in the organization of the architecture [49]. These compromises have a negative impact on performance. Last, but not the least, compromises made in the instruction set design (i.e., the architecture itself) have similar detrimental effect on the performance.

The advent of Reduced Instruction Set Computers, RISCs (Patterson [51]), during recent years have clearly demonstrated the tradeoff that exists between the instruction processing rate and the instruction set complexity under prevailing technological constraints. In the RISCs, only the more frequently-used instructions are implemented, using a fairly uniform encoding scheme. Consequently, the circuit complexity of the implementation is considerably lower than that for a CISC [18], and in the VLSI realizations of these architectures, considerable chip area can be devoted to features that result in marked enhancement in the throughput [25], [49]. These features include a large overlapping register file [49], multi-stage synchronous pipeline [49], [54] and an on-chip instruction cache [54]. Some studies have also shown the RISCs outperforming well-known CISCs in some instances [50]. The high throughput realizable by RISCs as opposed to the more global performance advantages
of the CISC have led to the raging RISC-CISC controversy [9], [52].

No matter how controversial the RISC philosophy is, it has clearly demonstrated the advantages of a simple instruction set in relation to its VLSI realization. In particular, the following points have been established:

- Large register files help in reducing the throughput considerably by reducing the off-CPU traffic (although this feature is not exclusive to RISCs).
- Less dense instruction encoding leads to simple, faster instruction interpretation.
- Uniform instruction execution times lead to simpler pipeline designs that provide considerably better throughput than asynchronous pipelines found in many CISCs.

The CISC approach and the RISC approach essentially represents two ends of the design spectrum for general purpose instruction sets. While a complex instruction set is likely to provide better ‘performance-in-the-large’ [46], a reduced instruction set is likely to provide a better instruction processing rate (performance-in-the-small).

1.7 Objectives of the Thesis

The objective of this dissertation is to propose an architecture that would provide efficient support for mechanisms for enhancing software reliability and yet be realizable in the same area-efficient manner as the less complex instruction sets. This latter feature would ensure that the VLSI implementations of the architecture would be free from the performance problems in similar realization of the more complex instruction sets. Specifically, the mechanisms for enhancing software reliability would include efficient support for:
1. Detecting run-time errors related to variables of enumerated and subrange types, parameter transfers across independently-compiled modules and accesses using pointers.

2. Enforcing data abstraction across and within program units.

3. Realizing small protection domains.

4. Enforcing module interface specifications.

5. Implementing user-specified, hierarchical exception handlers.

6. Implementing sophisticated event-driven debugging tools.

7. Realizing common operating systems functions related to process control, interprocess communication and storage management.

We will show later how many of these seemingly complex functions can be supported using a few complex instructions and other very simple instructions.

Towards the provision of architectural support for enhancing software reliability, our architecture will employ the following mechanisms:

- A very conservative tagging scheme.

- A highly-efficient capability mechanism that will enable the realization of data abstraction, small protection domains and highly explicit module interfaces. The resulting abstraction will enable the architecture to be 'object-based' [5], [53]. The capability mechanism will also enable the detection of dangling references.

- A register-oriented instruction set that would enable most data operation-related instructions to be executed in one (processor) cycle.

The instruction set design of the proposed architecture will take into account the technological limitations by partitioning the overall processing requirements into functions as ortho-
gonal as possible and realize such functions on independent VLSI chips serving as co-processors. Our approach to the minimization of the effect of the interchip communication delays will be to pipeline the operations of the execution unit and the capability unit and the use of a common instruction fetch strategy for the co-processors using a packet-switched bus.

An overview of an initial version of the architecture proposed in this thesis appears in [22]. In [21], details about the capability mechanism are provided.

1.8 Thesis Organization

Since capability-based addressing is a very central mechanism in our architecture, we examine the problems and issues in existing capability based architecture in some detail in Chapter 2. In Chapter 3, we present an overview of the proposed architecture, explaining the major design decisions on the way. The issues related to the organization of the architecture and its VLSI implementation are also described in Chapter 3. Chapter 4 presents the capability mechanism in the proposed architecture and related instructions. Chapter 5 deals with other aspects of the architecture and related mechanisms. In Chapter 6, we assess the proposed architecture by comparing it with similar architectures and describe our conclusions. Appendix I gives details about some of the instructions in the form of their corresponding micro-programs. Appendix II is a list of abbreviations and a glossary of terms used in this dissertation.
2. CAPABILITY-BASED ADDRESSING

In this chapter we look at capability-based systems and the implications of capability-based addressing in the context of software reliability. We also provide a critique of early and existing capability-based systems in this chapter in order to motivate the subsequent design for the capability mechanism to be incorporated in our architecture.

2.1 Capability-Based Addressing

In this section, we describe the concept of capability-based addressing and its advantages in realizing a secure computing system.

2.1.1 Protection mechanisms

In an environment where computational entities like code and data are shared among a number of users in possibly different access modes, a protection mechanism is necessary to ensure the integrity of the system. The protection mechanism ensures that the computational entities -- called objects -- are shared among users (or processes) in the prescribed access mode(s). The basic information structure that any such protection mechanism implements is called an access matrix [37]. The access matrix lists processes against objects, and an entry in this matrix, specified by a process-object pair, indicates the access modes allowed to the process on that object [37], [40].
A 'protection domain' [37] or a 'sphere of protection' [12] is an execution environment for a process that is defined by the objects accessible to processes executing in the domain, together with the access modes allowed to the objects in the domain [37]. In a generalized form, the access matrix lists objects against protection domain, with each entry describing the access mode allowed on an object by a process operating in that domain. In this form, the protection mechanism allows a process, operating in domain D, to access an object O in the access mode m if and only if

\[\{ O, A \} \in D \text{ and } m \in A.\]

Dynamic protection mechanism treats domains as objects -- this allows a process, given appropriate rights, to modify the contents of a domain or create altogether new domains [37]. Capability-based addressing [12], [16] represents one way of constructing such a dynamic protection scheme.

2.1.2 Capabilities

A capability is a special kind of pointer to an object that consists of (at least) the following two fields:

1) A field containing the name of the object.

2) A field ('rights field') describing the access mode allowed on the object using this capability.

A secure protection system can be built by ensuring the following:

1) A process P can access an object O in access mode m if and only if P has a capability to O, in which the rights field allows mode m.
2) Capabilities cannot be forged or altered, so that capabilities to other objects cannot be formed (by altering the name field) or higher-than-allowed access modes cannot be fabricated (by allowing the rights field, in effect amplifying the granted privileges).

In a system that uses capability-based addressing, the system, objects are implemented as variable-length segments. To access a word within an object, a process needs to specify the capability C to the object, as well as an offset ‘X’ to locate the desired word within the object. The underlying system performs the following functions to map this reference to the appropriate physical address:

1) The name in the capability is translated to the starting address of the segment representing the object named by the capability.

2) The offset X is added to the starting address to locate the desired word after ensuring that X is <= the size of the object.

3) The access is allowed to proceed if the requested access mode is consistent with the one in the rights field of the capability.

Exceptions are generated if violations occur at any point in the above sequence of resolving a capability reference to a physical location.

The underlying system (or a privileged process under its control) is responsible for the following essential functions:

i) It is responsible for generating capabilities for objects.

ii) It is responsible for ensuring the integrity of capabilities by protecting the words representing capabilities from modification by processes -- either by tagging such words or by storing capabilities in specially-protected segments called C-lists [12], [67].
iii) It maintains all necessary information for mapping the name in a capability to a physical address.

iv) It implements pre-specified policies to ensure that capabilities are formed and distributed in a controlled way.

A number of experimental and commercial capability-based systems have been built. Levy [40] provides a fairly detailed description of these systems. The systems include software implementations of capability-based addressing in the CAL-TSS system [40], and the MIT Timesharing system [40]. Most of the prior capability-based computer systems incorporate capability-based addressing into the architecture. Architectures in this class, in approximate chronological order, are as follows: the Plessey PP 250 system [15], the Magic Number Computer (abandoned) [40], the CAP systems [40], [66], the C.mmp/Hydra system [67], the Cm*/Star OS system [20], the IBM S/38 [28], SWARD [46], [47] and the iAPX 432 system [29], [40] (scrapped) from Intel.

2.1.3 Advantages of capability-based addressing

A number of potential advantages exist when capabilities are used to implement protection systems:

- Once a capability has been granted to a process and protected as described, the steps needed to check the validity of the access are fairly trivial compared to the checks that need to be performed for the other implementations of the access control matrix [37].

- If the name of an object in a capability is unique system-wide and independent of context, objects can be shared in a very general way without the restriction imposed by
other systems that do not insist on system-wide unique object names [12], [16]. Dennis has referred to this as 'generality of programming'.

- Capability-based addressing can be used to enforce data abstraction and thus implement abstract data types, as found in languages like Ada [63], CLU [42], Modula [65] and others. (Section 2.2.5 will describe relevant details.)

- By incorporating very simple mechanisms to restrict the access rights in the rights field of a capability, protection domains can be fine-tuned to the actual needs of a process. This fine-tuning process consists of granting only those capabilities that are absolutely necessary to a process and only in the minimum necessary access modes. Constructing protection domains in this manner is consistent with the 'principle of least privileges' [40] (also called the 'need-to-know' principle [66]), and results in a finer-grain protection system.

- With capability-based addressing, the traditional concepts of process and domain hierarchies are absent. This, coupled with the fact that capabilities are context independent, would permit sensitive programs (like operating systems) to be easily extended [40]. Other advantages of capability-based addressing can be found in [12] and [16].

2.2 Software Reliability and Capability-Based Addressing

The advantages of capability-based addressing were described in general terms in the last section. In this section, we describe the advantages of capability-based addressing from the perspective of software reliability.
2. 2. 1 Cleaner module interfaces

When capability-based addressing is used, a program module A can access only those objects of module B to which it has capabilities. Thus, the interfaces between the two modules are clearly defined — there are no hidden interfaces. Consequently, modularity is enforced, resulting in a well-defined interaction among the modules.

Many systems, such as [29] and [67], incorporate mechanisms to form capabilities to small contiguous zones within an object. Such capabilities are called refinement capabilities [29]. Given this ability, one can construct more exact parameter capabilities. For example, if only the third element of an array, (which is implemented as a single object) is to be passed to a module as a parameter, one just needs to let the called module access only the third element of the array — not the other elements. One can easily do this by forming a refinement capability to the third element of the array, and pass that capability as an argument to the called module (instead of passing a capability to the entire array). In a likewise fashion, capabilities corresponding to specific entry points in a module can be constructed. These approaches, again, will result in precisely defined module interfaces with their associated advantages.

2. 2. 2 Error localization

If protection domains are constructed following the principle of least-privileges, errors in a program/module will be confined fairly well in its protection domain. In fact, error propagation would be minimal; errors are also likely to be detected where they occur or close to the place of occurrence. The error localization property is simply a manifestation of the
fine-grain protection abilities allowed when small protection domains are used. A procedure call that entails a switch to the protection domain of the callee is called a protected procedure call.

2.2.3 Very general sharing

As mentioned earlier, context-independent names in capabilities permit objects to be shared in a very general way, in possible different access modes, across processes. The sharing objects can be made more secure, without the possibility of the well-known Trojan-Horse problem [46], by the use of refinement capabilities for parameters. A subservient, lower security class module will then have access only to the absolutely necessary parts of the higher security class calling module. Thus, in this context, refinement capabilities are just farther mechanisms for enforcing the principle of least privilege.

2.2.4 Secure parameter passing

A capability to an object is more than a mere pointer to the object -- it also has a rights-field that dictates the access mode allowed to the possessor of the capability. A calling module/subprogram can appropriately restrict the access codes in the rights field of parameter capabilities to make the capability consistent with the parameter type. Specifically, call-by-value parameters need to have a capability with just the 'read' access code in the rights field; call-by-result ('out' parameters in Ada, for example) should just have the write-access code in their capability; traditional call-by-reference parameters and 'in'-'out' parameters in Ada need to have both read and write access codes in the parameter capability; parameters corresponding to entry points need to have just 'execute' access in the corresponding parameter
2. 2. 5 Data abstraction

The concept of data abstraction, as found in languages like Ada [63], CLU [42] and others allows the encapsulation of a data item/structure with its associated operations. The encapsulation mechanism denies direct access to the underlying representation of the data and ensures that the only way to operate on the structure is via a call to one of its associated abstraction procedures and functions. Capability-based addressing has been successfully used in some earlier systems [20], [40], [46], [67] for enforcing data abstraction. It takes the form of a capability, severely restricted in rights, to the object corresponding to the underlying representation of the abstract type. Such a 'sealed' capability is then passed along to the users of the abstract type who are therefore denied the ability to directly manipulate the underlying representation. The users call one of the abstraction procedures/functions, specifying this sealed capability as a parameter. To gain access to the underlying representation, the called abstraction procedure/function, in turn, calls a privileged procedure, known as the type manager, to amplify the rights field in the capability to the representation object. The abstraction procedures/functions take care not to pass back the amplified capability back to the caller. Access to the underlying representation object is thus made in a very controlled way, under the purview of the type manager, thus enforcing the abstraction. Details of a specific implementation can be found in [42].
2.3 Other Issues in a Capability-Based System

In this section we will briefly discuss some other peripheral issues in the design of a capability mechanism.

It may be a natural requirement in capability-based systems to withdraw the access rights previously granted to a user. This has been referred to as the capability revocation problem. The solution adopted is to use an indirect capability. An indirect capability points to another capability, which in the most simple case, points to an object. (In this case, we will refer to the latter capability as a ‘direct capability’.) To grant a revocable access privilege to a user, one then simply needs to grant an indirect capability to the object. To revoke the access rights, one then simply needs to void the direct capability, so as to prevent any indirection through it.

Another issue in a capability-based system concerns the propagation of sensitive information from one domain to another. There may be a need to grant a capability to a user for use in a particular protection domain and at the same time ensure that information accessible using the granted capability is used within the specified domain only. This problem has been referred to as the ‘confinement problem’ [67], and to date a complete solution to this problem does not exist.

Another frequently-encountered problem in capability-based system is the ‘lost capability problem’, which is as follows. If capabilities are allowed to be removed in an uncontrolled manner, it is possible to have, at some point in time, an object that does not have any capability to it. (This situation has no effect on the security of the capability mechanism.) Garbage collection is usually employed to reclaim storage from such dangling objects.
Garbage collection can also be employed to reclaim capabilities to an object, that has been deleted by an appropriately authorized process, to prevent dangling references.

Finally, an implementation problem needs to be pointed out. The storage area occupied by a capability-addressed object cannot be simply reclaimed and re-allocated to another object. If this is done, the latter object may accidentally inherit capabilities left behind in the storage area by the earlier object and thus compromise the security of the system. The solution to this problem is obvious: reclaimed storage areas are 'cleaned' (to invalidate capabilities left behind by the previous user of the storage area) before allocation.

2. 4 Problems in Existing Capability-Based Systems

Existing capability-based systems suffer from a number of performance problems in the form of large capability translation time, large overhead for swapping objects across storage hierarchies, large domain switching time and large amplification/de-amplification times. We will try to identify the source of these problems in this section.

In many capability-based systems and architectures, like the Intel iAPX 432, the Plessey PP 250, the CAP and the Cm*StarOS system, the name of an object, as used in a capability, depends on the disposition of the object in the storage hierarchy. In particular, the name of an object, while it is resident in the main memory, is based on the indices of its corresponding entries in a root-based mapping-table, that maps its name to a physical address. A capability that uses an object name of this form is called a 'short-form' or 'in-form' capability [15], [66]. Since objects, as well as entries in the lower-level mapping tables are swappable,
the object name in an in-form capability is not unique. To identify an object uniquely in the system, a 'long-term' or an 'out-form' capability is used. This out-form capability uses a name of the object that is unique to it at least through its lifetime. (Many systems use the secondary storage address of the object as its unique name in this regard.) The major reason behind using these two forms of capabilities lie in the relative sizes of the object-name in these two forms. The name in the in-form capability is considerably shorter and simpler algorithms that do not entail any searching if all the mapping information is resident in memory, can be used to map this in-form capability to a physical address.

There are two major sources of inefficiency in systems that use capabilities in these two-forms [15], [20], [29], [66]:

- The translation of an in-form capability to a physical address takes a long time, due to the use of multi-level mapping tables. This may be compounded by the fact that some of these lower level tables are swappable.

- Whenever an object is swapped out, all in-form capabilities to it have to be converted to the out-form to prevent incorrect references to objects that will subsequently use mapping table entries vacated by the swapped out object. Likewise, all in-form capabilities within the swapped out object have to be converted to their out-form to prevent inconsistent references later.

A particularly notorious example of the large capability translation time occurs in the Intel iAPX 432, where seven levels of indirection through mapping tables are necessary to translate a capability to a physical address [40]. Systems like the IBM S/38, SWARD and the C.mmp/Hydra do not suffer from this problem since they use object names that are independent of the disposition of the object in the storage hierarchy.
All the existing capability-based systems that provide instructions or primitives for
domain switching are notoriously deficient in this regard [20], [29], [46], [67]. Domain
switching is a very expensive process in these systems -- so much so that it discourages its
use by programmers [67]. An examination of the results published in [8], [20] and [67] indi­
cates that a significant amount of the domain switching time is spent in creating the context
object (or environment) in the called domain or in 'cleaning' an already allocated context
object. A significant amount of time is also spent in transferring parameters to the called
domain [67]. For example, the Hydra system spends 30 mS., of the 70 mS. needed to per­
form a domain switch, to create a context for the called domain; most of the remaining 40
mS. are spent in passing and amplifying parameter capabilities [67]. As another example, the
Intel iAPX 432 spends 334 clock cycles, out the 982 cycles it needs to perform a domain
switch with just four parameters, to clean a pre-allocated context object [8]. The major over­
head for domain switching in existing capability-based systems are thus due to context object
preparation and parameter passing.

All the architectures/systems that have provided architectural support for implementing
abstract data types use capability-based addressing and most of them follow the technique
pioneered in the Hydra operating system based on the C.mmp [67]. In these systems, type-
managers, as well as users with sufficiently privileged capabilities can get at the underlying
implementation of an abstract data type. This generality requires the provision of generic
objects and their associated operations and is not required in most instances [40], [67]. Also,
the amplification of the supplied capability to get at the underlying implementation is a fairly
complicated process and "the full generality of the rights amplification process is not neces­
SARY for the vast majority of applications" [67]. The SWARD architecture provides a fairly
straightforward approach for implementing abstract data types by using a capability for each abstraction procedure. In SWARD, however, the same CALL instruction that provides inter-module calling facility is used for implementing calls to the abstraction procedures of an abstract data type. This is an overkill, since the CALL instruction in SWARD does several other things like parameter consistency checking, allocation and initialization of locals, and formals to actuals besides transferring control to the called procedure, after saving the current status. The implementation of abstract data types in SWARD is inefficient from another standpoint. For each instance of an abstract type, one is required to make a copy of the code representing the abstraction procedures for integration into the object corresponding to the representation. Unlike other implementations, it does not share a common copy of the abstraction codes among all representation objects.

As seen from the discussions in this chapter, capability-based addressing is a powerful tool for the realization of many of the mechanisms needed for enhancing software reliability. However, the capability mechanisms in existing systems need a substantial upgrading in their performance to make the usefulness of capability-based addressing worthwhile. In the next chapter, we discuss the major considerations involved in the design of the proposed architecture and provide an overview of its structure.
3. MAJOR DESIGN DECISIONS

In this chapter, we first examine the functional requirements of the instruction set of the proposed architecture, as dictated by the need to support mechanisms for enhancing software reliability. We then discuss the major issues involved in the implementation of the instruction set under prevailing and near-term technological constraints. The inevitable tradeoff involved between the semantic power of the instruction set and the performance of an implementation of this instruction set under technological constraints dictate some major design decisions. We will reveal such design decisions for the proposed architecture and try to justify them. Some of these design decisions will be hard to justify in this chapter — more concrete evidence in favor of these decisions will be presented in later chapters.

3.1 Functional Requirements of the Instruction Set

Primarily, the instruction set of the proposed machine is intended to efficiently support mechanisms that need run-time facilities for enhancing software reliability. Thus, the proposed architecture will not provide any kind of support for the static techniques for enhancing software reliability as discussed in Chapter 1. We believe that the compilers or preprocessors provide sufficient support for the static mechanisms and, in any case, the execution times for the static techniques are not critical.

In the light of the discussions made in Chapters 1 and 2, the architecture should incorporate the following:
• Facilities to detect most of the common run-time errors.

• Facilities to support and encourage module-based programming.

• Mechanisms to support and enforce data abstraction (of both system and user-defined types) across independently-compiled modules, where an abstract type defined in one module is used in another module. (In a single-module environment or where an abstract type is used only within the module where it is defined, the compiler can enforce the abstractions.)

• Mechanisms to enforce and encourage the use of small protection domains.

• Facilities to promote the sharing of data and code in a secure manner across users.

• Mechanisms to support hierarchical exception handling, as found in languages like Ada, CLU and Modula.

• Mechanisms to implement common operating system functions related to process control, interprocess communication and storage management.

• Mechanisms to aid the implementation of efficient debugging/tracing tools.

In the next section, we choose appropriate mechanisms towards the realization of these functional requirements.

3.2 Design Decisions Based on Functional Requirements

Based on the functional requirements laid out in the previous section, the following architectural characteristics are identified for the proposed machine which are independent of any implementation constraint:
3.2.1 Capability-based addressing

With regard to facilities for data abstraction, module-based programming, small protection domains and information sharing, capability-based addressing becomes a rather obvious choice for the addressing mechanism in the proposed architecture. As indicated in Chapter 2, capability based addressing will also provide a number of fringe benefits in the form of 'cleaner' module interfaces, secure parameter passing mechanisms and as a mechanism for detecting dangling references.

3.2.2 Tagging

To protect capabilities from casual manipulation and to prevent forgery of capabilities, a mechanism is needed to distinguish word representing capabilities from other types of words. A one-bit tag associated with each word is chosen for this purpose. This would also permit the storing of capability and non-capability items in the same segment. In particular, an integrated activation stack can be maintained -- there would be no need to have two parallel activation stacks -- one for capabilities and the other for non-capabilities. We will also employ tagging for realizing a number of other facilities -- in the implementation of hierarchical fault-handling (Chapter 5) and in the implementation of a mechanism to detect references to uninitialized locations (Chapter 4).

3.2.3 Support for run-time error detection

Based on the programming error statistics for PL/1 in [45], [46] and for Ada in [13], the more prominent run-time errors, in approximate order of frequency of occurrence, are as
follows:

a) Referencing an uninitialized location.

b) Reference using an invalid pointer.

c) Use of an invalid array subscript.

d) Assignment of an out-of-range value for an enumerated or subrange type.

e) Inconsistency across independently compiled modules in number and type of parameters.

f) Address space violations (excluding those similar to (e)).

We will incorporate necessary mechanisms in the architecture for detecting these run-time errors efficiently.

Capability-based addressing will be instrumental in detecting errors involving the use of invalid pointers, address space violations, and some forms of inter-module parameter inconsistencies. As stated earlier, tagging will be used to detect use of an uninitialized variable.

Arrays are the most widely used data structure in programming languages that are prone to run-time errors. The variable-usage statistics for some representative Pascal and C programs, as reported in [33], indicate that 15% to 40% of dynamic references are to arrays. The variable usage statistics, measured statically, in a large Ada program is shown in Figure 3.1. Although the ‘record’ type is the most commonly used structure, as shown in this figure, arrays are still the most commonly-used structure in Ada that are prone to run-time errors. (Unlike array subscripts, record field selectors can be checked at compile-time.) It is therefore especially important to check for array index violations at run-time, possibly on every array reference. Further, this check should have very little, or ideally, no impact on the
<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Usage (percentage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration and Subrange</td>
<td>23.8</td>
</tr>
<tr>
<td>Record</td>
<td>16.8</td>
</tr>
<tr>
<td>Array</td>
<td>11.9</td>
</tr>
<tr>
<td>Access (pointer)</td>
<td>11.9</td>
</tr>
<tr>
<td>Integer</td>
<td>11.9</td>
</tr>
<tr>
<td>Task</td>
<td>7.7</td>
</tr>
<tr>
<td>String</td>
<td>7.0</td>
</tr>
<tr>
<td>Private</td>
<td>5.6</td>
</tr>
<tr>
<td>Others</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Figure 3.1 Variable Usage Statistics (Static) in Large Ada Programs

run-time performance.

Modern programs tend to make fairly heavy use of the subrange and enumerated types. This is clearly corroborated by the statistics reported for Ada in Figure 3.1. It is thus important to provide facilities for detecting run-time errors related to these types.

In our architecture, we will employ a uniform mechanism for detecting errors related to array subscript violations, subrange and enumerated types. Our philosophy would be to consider an array subscript as a special instance of a subrange type. (Indeed, in well written programs, this would be explicitly stated by declaring array indices as subrange types.) We will also implement enumerated types by mapping them onto an appropriate subrange type. We will provide mechanisms in the architecture to detect run-time errors related to subrange types in a very efficient way. This mechanism will also detect out-of-range array indices and enumerated types.
3.2.4 Support for debugging tools

Towards the efficient implementation of debugging tools, we will provide the following support in our architecture:

- A mechanism to detect references to specific locations, with the ability to ‘mark’ these locations without the need to insert special ‘marking’ instructions and code re-compilation. Further, this mechanism should enable us to specify the access mode for the reference being detected.

- A mechanism to indicate the branch trace (address of instruction before and after branching) and call trace (trap on procedure calls).

- A mechanism to enable the processor to single-step through a program, displaying pre-specified registers and locations after every step.

Although, one is tempted to include architectural features for generating high level dumps (such as richly-tagged words or descriptions), we will leave them out of the proposed architecture for at least two reasons. First, dumping can be a slow process and one can tolerate delays, so that software-implemented dumpers are a viable alternative. Second, the architectural features for implementing high-level dumps are fairly expensive in terms of performance penalty and area requirements in a VLSI implementation of the architecture.

3.2.5 Support for exception handling

Program specified exception handling is a major feature of many programming languages, especially the modern ones like CLU, Ada and Modula. Further, most of these languages have the ability for multi-level exception handling, as discussed in Chapter 2. Our
architecture will provide appropriate facilities for exception handling in this manner by:

- recognizing a number of common exception conditions (predefined exceptions).
- incorporating mechanisms to locate the address of the nearest user-specified exception handling routine in the dynamic call chain.

In Chapter 5, we will describe the support for exception handling in our architecture.

3.3 Technological Constraints

Ideally, one would expect the design of an instruction set to be independent of the manner in which the instruction set is implemented. Reality, however, is otherwise. The semantic content of an instruction (or, loosely, how much processing it does) has a fairly direct relationship to the amount of hardware/firmware needed to implement it. Recent experiments [49], [54] centered around the Reduced Instruction Set Computers have also indicated that complex instruction sets, with a large number of instructions or higher level of instruction semantics, have a negative effect on the throughput-based performance of the system. Simpler instruction sets fare better than complex instruction sets on many traditional benchmarks. This apparently paradoxical behavior can, however, be explained, when one considers the technological constrains imposed on the implementation of an instruction set.

Current and near-term technology places a number of restrictions on the realization of a processor in VLSI technology. These were indicated in Chapter 1. We will repeat them here for convenience:
• There is a limit imposed on the chip area (and, correspondingly, on the number of active devices on the chip), in order to achieve meaningful chip yields [18].

• There is a limit on the number of input-output pins out of a chip package [18]. Consequently, there is a bandwidth limit imposed on the traffic in or out from the chip.

• Off-chip traffic delays are normally six to ten times higher than delays within the chip itself, in most currently-used VLSI technologies [43].

• In many VLSI technologies, device densities on the chip are severely restricted by available technology and limits imposed by heat-dissipation abilities [18].

The RISC research effort, discussed in Chapter 1, indicates the positive attributes of a simpler instruction set. The VLSI implementation of simple instructions make more effective use of the limited chip area under the prevailing technological constraints. Architectures with simple instructions sets also provide higher chip yields and require lower design and testing times [18], [49].

3.4 Design Decisions Based on Technological Constraints

The discussions on the last section enable us to reach further design decisions based on the technological constraints.

3.4.1 Instruction set complexity

Our formulation of the instruction set requirements in Section 4.2, and the functional descriptions of many of these mechanisms in Chapters 1 and 2, indicate that fairly complex
instructions would be inevitable for some of the operations related to abstraction and domain switching, interprocess communication, storage management, hierarchical exception-handling and debugging. It is, however, noteworthy, that one expects these operations to be rather infrequent compared to the 'normal' data manipulation operations. While we have very little choice in reducing the complexity of these higher order instructions, we do have a choice in the complexity of the data manipulation instructions. Making the data manipulation instructions complex (such as storage-to-storage operations, instructions for accessing arrays, etc.) would increase the overall complexity of the instruction set for the worse. On the other hand, keeping the data manipulation instructions simple can permit their implementation to be as area-efficient as the RISCs, and thus permit a better overall throughput, since data manipulation instructions will dominate all other types of instructions. In view of this we will opt for simple data manipulation instructions. This choice will be further justified when the multi-chip implementation of the architecture becomes inevitable, as explained later in Section 3.4.4. The instruction set of the proposed architecture will be hybrid in terms of instruction complexity and semantics: a small number of comparatively infrequently-executed complex instructions and a modest number of frequently-executed, simple data manipulation instructions.

3.4.2 Operand references in instructions

Even if data manipulation instructions are simple, a choice exists as to the sort of operand references these instructions would have. We can have storage-to-storage addressing, in which all operands are in memory locations, or register-based instructions, in which most operations use register-resident operands, or a combination of these two. Register-based
operands recognize the storage hierarchy, storage-to-storage instructions do not. To exploit storage hierarchy, storage-to-storage architectures employ caches to capture and hold the locality within the CPU. A reference to a storage location then takes about the same time as accessing a register. Even then, a VLSI machine with a register-oriented instruction set is likely to be superior in performance to a VLSI machine employing storage-to-storage addressing and a cache for the following reasons. When caches are used, it remains to be decided which one of these two addressing modes is the better approach. Several factors influence this decision:

1. **Tracking data locality:** Caches track the locality of reference dynamically, and are apparently better than registers, which track the locality based on (static) compile-time information. However, recent compiler technology, especially, those used in MIPS (Przybylski et al. [54]) and the IBM 801 (Radin [55]), employ register allocation strategies that enable the reference locality to be tracked almost as well as caches.

2. **Area requirements:** For the same word capacity, registers will require less circuitry and area than the cache. And since new compiler technology enable registers to track the locality of reference as well as caches, registers are desirable in view of their lower area requirements.

3. **Address specification overhead:** An issue in the choice between register addressing and storage-to-storage addressing concerns the number of bits needed in the instructions to address operands. Again, registers fare better, since fewer bits are needed to address the relatively smaller number of registers. We will, as a result of these advantages of register-based instructions, employ a register-oriented instruction set. Even for the few complex instructions, in most instances, operands would be register-based. (A fairly
recent example of a bad choice in choosing storage-storage addressing appears in the Intel iAPX 432. Colwell showed that by adding just eight registers to the 432 architecture, instruction throughput could be increased by 33% [8].

In view of these advantages of a register-oriented architecture over a storage-to-storage architecture, we will employ a register-oriented instruction set for the proposed machine.

3.4.3 Memory organization

In a capability-addressed space, objects are variable in length and thus normally implemented as segments of variable length [40]. We will not deviate from this choice. However, to permit efficient use of storage, we will implement segments on top of a paging scheme and incorporate memory management facilities into the architecture. To enable the page size to be tuned to system requirements, the architecture will incorporate facilities to choose one of a number of pre-specified page sizes.

3.4.4 Single chip vs. multi-chip implementation

A closer look at the various functional requirements formulated for the architecture reveals that the overall requirements are far too complex to fit into a single chip under current technological constraints. The obvious choice, at this point, is therefore to opt for a multi-chip implementation of the architecture. This permits partitioning the functional units of the architecture over a number of VLSI chips. However, once such a partition is made, interchip propagation delays are introduced which have a negative effect on throughput. Thus, the partitioning should be done carefully, in order to keep the effect of interchip traffic delays to a reasonable minimum. There are two approaches for doing this:
a) Partition the functional requirements of the original system into a number of fairly orthogonal sub-functions, and implement each sub-function on a single-chip. The orthogonality of these functions would ensure that the interaction among the chips would be fairly low.

b) Where orthogonal partitioning is not possible, in order to mask the effect of inter-chip delays, inter-chip communication can be pipelined [43].

Figure 3.2 shows a partitioning of the proposed architecture that employs both of these strategies. As shown, the system is implemented as four distinct chips, with the following roles:

1. Execution Unit (EU) -- this is the main processing element in the system and is responsible for controlling all other units. The EU itself implements all scalar data manipulation operations, using a register-based instruction set. It also implements some of the more complex instructions in conjunction with one or more of the other units in the systems.

2. Capability Unit (CU) -- this chip is responsible for implementing all capability-related operations, including the memory management functions related to a capability addressed space. It incorporates a set of registers for holding capabilities referred to by instructions, status/control registers, and a translation look-aside buffer, to hold recently translated capabilities. The capability unit also incorporates appropriate logic for generating universal identifiers (uids) for capabilities, for translating a capability-based reference to a physical address, and for other capability related operations. The operations of the execution unit and the capability unit are pipelined to mask out the communication delay between these two chips -- while an instruction is in execution in the
NOTE

1. Memory and I/O Processor are not shown.

2. Dotted lines represent multiplexed connections from the Execution Unit and the Capability Unit.

3. The Execution Unit and the Capability Unit are internally pipelined.

4. The Execution Unit and the Capability Unit together form an extended pipeline.

Figure 3. 2 Functional Units in the Proposed Architecture
EU, the capability reference for a subsequent instruction in translated by the CU.

3. A floating point co-processor, that implements floating point operations at the request of the execution unit. It incorporates a set of floating point registers, status registers, a floating point arithmetic unit and appropriate interface logic. We will not describe this chip further in this thesis -- its design and interfacing will be fairly conventional and identical to those used in many contemporary microprocessor system.

4. A Garbage Collecting Unit (GCU) (co-processor), whose responsibilities include the garbage collection of inaccessible objects and the 'cleaning' of deallocated page frames. The necessity of cleaning de-allocated page frames will be explained in Section 4. 5. 2 of the next chapter.

The justification for breaking up the functional requirements of the architecture over these functional units (or co-processors) will become evident in subsequent discussions.

3.5 Other Considerations

This section presents a list of some organizational features we will assume to be existent in a hypothetical implementation of the proposed architecture. The assumptions are realistic and represent an effective way of organizing the architecture. One can thus look upon these organizational features as design decisions related to an implementation of the architecture. Assuming these organizational features enables us to estimate the performance of the proposed architecture in terms of instruction timings from the algorithms implemented by these instructions, as given in Appendix I.
Finally, it should be noted that these organizational features broadly define an implementation of the proposed architecture and this implementation is clearly one of the many possible ones. We also do not claim that the implementation suggested by these assumptions is optimal, rather it represents a balanced and efficient trade-off between functionality and technological limitations.

3. 5. 1 Control and data path implementation

We will also assume that there are enough data paths within the Execution Unit and the Capability Unit, and enough clock phases to allow two register transfers and two uses of the shifter in one clock cycle. This is a very realistic assumption, consistent with technological limitations. (The RISC II microprocessor, for instance, uses a four-phase clock and three internal data busses to do just this [33].)

There are a number of possible alternatives for implementing the control circuitry of the proposed architecture. Our choice in this respect is to use microprogrammed control. We will assume a highly horizontal micro-instruction format so that many of the simpler data-manipulating instructions can be implemented using one micro-instruction word, and can execute in one machine cycle.

We will also make one more assumption regarding the microprogrammed control unit: we will assume the micro-sequencer to be pipelined, so that the execution of a micro-instruction is overlapped with the fetch of the next micro-instruction. This is a very reasonable assumption, that adds very little complexity to the microsequencer circuitry.
3.5.2 Pipelining the execution unit

Pipelining the Execution Unit's operation with the operation of the Capability Unit, as briefly mentioned in Section 3.4.4, is an obvious way of masking the communication delay between these two chips. Further pipelining of the (macro) instruction execution sequence is also possible within each unit. A wide range of possibilities also exist in the implementation of this pipeline [27], involving stage timings (synchronous/asynchronous), interstage and inter-instruction conflict handling, branch-handling and so on. We will refrain from making any specific assumptions in this regard and simply assume that pipelining provides overlapping operation of the Execution Unit and the Capability Unit, as described above.

3.5.3 Instruction issue logic

There will be some instructions that involve processing within more than one co-processor. As an example, the instruction for domain switching (as presented in the next chapter) will require both the Execution Unit and the Capability Unit to execute appropriate actions to effect the domain switching. To handle instructions like this, we will assume a common instruction fetch logic for each co-processor. Instruction fetches (triggered by the Execution Unit) retrieve instructions that are stored in buffers in each co-processor and independently decoded into appropriate actions. As a continuation of the last example, the instruction for domain switching will be fetched into all co-processors, but only the Execution Unit and the Capability Unit will decode these instructions and execute the necessary operations. So far as the other co-processors are concerned, this instruction is effectively a 'no-operate' instruction. This assumption is not crucial for the performance and circuit-
complexity estimates, but does indicate a possible solution to the reduction of traffic between the Execution Unit and the other co-processors.

3.5.4 Interrupt handling policy

In an architecture supporting complex instructions, it is both necessary and desirable to have the ability to temporarily suspend an instruction in the middle of its execution to service system-level interrupts [25]. To enable the instruction to continue after the service of the interrupt, two possibilities exist:

1. Undo the state change made by the partially-executed instruction and restart the instruction. Or,

2. Save the processor state at the point of interruption and continue the execution of the interrupted instruction from the point where it was interrupted.

Doing either of these would guarantee that an instruction is 'logically indivisible'. The second approach is preferable for complex instructions, because undoing the state change made by a partially-executed complex instruction may not be an easy thing to do. We have chosen the second approach for ensuring instruction indivisibility in our system, since some complex instructions do exist in our system. To do this, we allow instructions to be interruptable at the microinstruction level. This, of course, entails the saving of the registers used by the microprogram, as well as the various status registers.

Starting with the next chapter, we proceed to describe the architecture of the proposed system through a description of its instruction set.
4. THE CAPABILITY MECHANISM AND RELATED INSTRUCTIONS

In this chapter, we look at the architectural characteristics of the proposed machine that are centered around its capability mechanism. To this end we present an overview of the architecture that is a pre-requisite for all the sections to follow. We then proceed to describe the capability mechanism and related instructions. We estimate the execution time of these instructions on the basis of the micro-architectural characteristics assumed in Chapter 3. The instruction set-level performance evaluation will make use of statistics of program and system behavior collected by others to establish a basis for comparing the proposed architecture with similar machines.

4.1 An Overview of the Architecture

In this section, at the risk of some repetition with earlier chapters, we present an overview of the proposed architecture. This overview focuses mainly on the user-programmable aspects of the architecture -- organizational details will be kept to the bare minimum. This overview will describe the essential architectural attributes of the two major components in the system -- the Execution Unit and the Capability Unit.
4.1.1 Word format

The primary mechanism in the architecture for distinguishing between words that represent capabilities (or parts of a capability) and words that do not is tagging [17], [46]. Our motivation for using tagging in this respect was explained earlier in Section 3.2.2.

The logical word format in the architecture consists of a 34-bit physical word, with two primary tags followed by a 32-bit field for holding information. The two primary tags associated with each word are as follows:

1) A 1-bit tag, called the *word tag*, to indicate whether this word is part of a capability or part (or all) of a non-capability item (instruction or data).

2) A 1-bit tag (called the *initialization tag*) to indicate if the contents of the remaining 32-bit part of the word has been initialized or not. This tag bit is used by the hardware to detect an attempt to access an uninitialized storage location. When physical storage is allocated, this bit is reset (by the garbage collector unit, which is also responsible for such cleanup chores). If the 32-bit information field of a word is written, this bit is set (to indicate that the word has been initialized).

Capabilities occupy two consecutive physical words in the system and employ secondary tag bits to indicate the capability type, to designate the first word of a capability and so on. The format of a capability will be described in detail later, in Section 4.3.

4.1.2 Programmable registers

The programmable registers (registers usable directly by programs) within the EU and CU can be grouped into the following categories:
1) **Data registers**, that can hold non-capability items. These registers are within the EU, and serve as the source and destination of operands for the arithmetic/logic unit within the EU. Data from object representations in the main memory can be loaded into these registers using a LOAD instruction. Data from these registers can be stored in the main memory representations of objects using the STORE instruction. These registers are 33-bits wide and hold the 32-bit data item and the 1-bit initialization tag.

2) **Capability registers**, 66-bits long, holding capabilities (each two word long) and the initialization tags of the two words that represent the capabilities. These registers are within the CU.

3) **Offset registers**, similar in structure to the data registers, to hold offset values to locate words within objects, in conjunction with the capability registers. These registers are within the CU and there is an offset register for every capability register.

A data LOAD (or STORE) instruction specifies a target object and a location within the object using a capability register-offset register pair and a sink (or source) data register. Capabilities can be loaded to or stored from capability registers using the LOAD-CAP and STORE-CAP instructions, respectively. Offset-registers can be loaded from or stored to the main memory representations of objects using the LOAD-OFFSET and STORE-OFFSET instructions, respectively. These instructions will be described later in Section 5.1.

Hardware associated with the bus buffer registers within the EU and CU generate an exception when an attempt is made to load these registers with the contents of words whose initialization tags are reset. In the CU, this hardware also detects any attempt to load two consecutive words that do not represent a capability. Likewise, the hardware associated with the bus buffer in the EU generates an exception when an attempt is made to load a data
register with one of the two words that represent a capability. Special circuits associated with the arithmetic unit within the capability unit generates an exception if the contents of an offset register involved in an arithmetic operation becomes negative. The use of tags enable the hardware circuits for these checks to be logically very simple, possibly comprising of a few gates.

4.1. 3 Data register organization

There are 24 data registers, D0 through D23, and their usage is restricted as follows:

• These registers hold locals, globals, parameters and the top elements of the display to the activation stack. They can also hold temporary variables.

• When a protected procedure call entailing a domain switch is made, the initialization tags associated with all but registers D0 through D7 and registers D22, D23 are cleared. This, in effect, invalidates the contents of registers D8 through D21. The contents of the registers D0 through D7 and D22, D23 remain unaffected — these registers are therefore used for inter-domain parameter transfers and conveying information about parameters.

The number of data registers (viz., 24) was not chosen on an ad-hoc basis. The following statistics of program usage dictated this choice:

a) More than 97% of the called procedures have 6 or less parameters [33], [61].

b) Up-level addressing for non-locals rarely exceed 3-levels (so that 3 registers in the display will be sufficient in most instances).

c) 95% to 97% of procedures have 8 or fewer local variables [49], [61].
d) In well-structured programs, 5% or less of the dynamic references are to non-locals [68]. Thus, few registers to hold globals would not cause any serious performance problems.

These statistics were observed for languages like Pascal [33], [49], C [33], [49] and SAL [61]. Measurements made for a large Ada systems program [68] tend to agree with these statistics. In the very rare event that these data registers are not enough to hold the globals, locals and parameters, the additional items may be stored on the memory-resident activation stack or as memory-resident objects.

Although, we were tempted to use overlapped register windows to speed up procedure calls and returns, we refrained from doing so. Our decision was based on the availability of new compiler technology for register optimization [51], [55] that enables extremely fast calls and returns without the use of overlapping register windows. The performance measurements reported for the windowless, 16-register MIPS machine [54] corroborates this decision.

Some of these data registers will also have special functions when the subrange violation mechanism is invoked. (See Section 5. 5. 2.)

4. 1. 4 Capability and offset register organization

The set of capabilities specific to a domain can be categorized as follows:

- Capabilities to domain-local objects.
- Capabilities to other domains always callable from the domain.
- Capabilities passed in as parameters by the caller.
Despite the implementations of at least seven well-known capability-based architectures [15], [20], [28], [29], [46], [66], [67], no statistics exist regarding the usage of capabilities in these systems. Our choice for the number of capability registers is therefore based on the way these registers would be used in typical programs. A reasonable assumption to make in this regard would be to consider all inter-package (or inter-module) calls to be protected procedures calls, and then rely on statistics below to determine the number of capability registers.

- Colwell's static analysis of module connectivity in three large Ada programs [8] show that 38% of procedures (and functions) in a package do not call procedures outside the package, 22% call only one other package, 18% call two other packages, 10% call three other packages and less than 12% call four or more other packages. This statistics thus suggests that 4 capability registers for callable environments would be sufficient in 88% of the cases.

- Parameters are passed from one domain to another via procedure calls -- a call to a procedure in another package. The statistics collected for parameter usage, as mentioned in Section 4.1.3 will thus hold. In the worst case, if all parameters are capabilities, six capability registers for parameters would be sufficient in 97% of the cases.

- The package usage in a large Ada program as reported in [68] suggests that packages have four variables on the average.

Based on these (relatively sparse) statistics, we will choose 16 capability registers for the proposed architecture, C0 through C15 and 16 corresponding offset registers, F0 through F15. Eight of these capability registers, C0 through C7 and their corresponding offset registers, F0 through F7, would be used for inter-domain (capability) parameter transfer and the
others will be used to hold domain specific capabilities. On a domain switch, the initialization tag of capability registers C8 through C15 and offset registers F8 through F15 would be reset, in effect clearing them. In the rare event that the number of capability and offset registers are not sufficient to hold all capability-related information for a domain, a memory-resident object can be used to hold the additional items.

4.1.5 Object types

The proposed architecture will support the following hardware-recognized object types:

- Generic objects, that can hold capabilities, non-capabilities, (data/instruction) or both.
- Code objects, that can hold only executable code.
- Structured code objects (SCOs), corresponding to objects containing a collection of procedures implementing an abstract data type or objects containing code for modules with predefined entry points.
- Port objects, corresponding to the abstraction of an inter-process communication ports.
- Virtual storage objects (VSOs), that contains mapping information about a large contiguous virtual address space.
- System-type objects, that can be used only by the operating system for implementing customized object types. A typical use for an object of this type is to implement the 'context object' for a domain -- which contains the activation records of procedures executing in the domain.

The underlying architecture ensures that only instructions appropriate to an object's type can operate on it. The architecture is thus object-based.
Instructions, described later in Section 4.5, exist for creating generic objects. The context object is automatically created by the system on a domain switch, as described in Section 4.8. Objects of other types are created from generic objects in a manner described later in Section 4.7.4. Instructions are also provided for destroying all but the context object. The context object is automatically destroyed on return from a protected procedure call.

4.1.6 Control registers

There are a number of special-purpose registers within all units. These control registers, which are not accessible to the compiler, are used only by the underlying microprogram.

The major control registers within the capability unit are as follows:

- A capability-register, offset-register pair, PC, PF, serving the role of the traditional program counter.

- A capability offset register pair, CC and CF, serving as a pointer to the executing context object and as the top of stack pointer of the activation records in this context.

- A capability register TC holding the capability to the object (possibly of 'system' type) corresponding to an abstraction of the current process (or task), an offset register TF, for accessing locations within this process object and a register that gives the priority of the currently executing process.

- A CU-status register, designating the operational status of the CU and codes to designate fault-conditions (exception type).

- A capability register, CCS and an offset register, CCF, pointing to a 'system' object, containing the stack of capabilities to active context objects and the pointer to the top of
this stack, respectively. We will explain the use of these registers when we discuss the
domain switching mechanism in Section 4. 8.

- A capability register CH, and an offset register, FH, pointing to a 'handler stack' and
the top of this stack, respectively. We will explain the concept and use of a handler
stack in Section 5. 7.

- Three 33-bit registers, S0 through S2, for transferring arguments to the operating system
during exceptions or during explicit system calls.

- A collection of capability and offset registers used by the microprogram during instruc-
tion interpretation.

- Instruction pre-fetch registers and inter-stage latches, to accommodate the pipelining.

The major control registers within the Execution Unit (EU) are as follows:

- A status register, indicating the operational status of the EU and exception codes.

- Temporary registers used for instruction interpretation.

- Instruction pre-fetch buffers and inter-stage latches for implementing the pipelined
organization.

A description of control registers within the other co-processors will be provided later,
as needed. Figure 4. 1 summarizes the major registers within the Execution Unit and the
Capability Unit.
<table>
<thead>
<tr>
<th>REGISTER(S)</th>
<th>FUNCTION</th>
<th>SIZE (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 - D7</td>
<td>Data Parameters in Protected Call/Data</td>
<td>32</td>
</tr>
<tr>
<td>D8 - D15</td>
<td>Data</td>
<td>32</td>
</tr>
<tr>
<td>D16 - D20</td>
<td>Subrange Comparands/Data</td>
<td>32</td>
</tr>
<tr>
<td>D22 - D23</td>
<td>Parameter Template/Data</td>
<td>32</td>
</tr>
<tr>
<td>C0 - C7</td>
<td>Capability Parameters in Protected Call/Capabilities</td>
<td>64</td>
</tr>
<tr>
<td>C8 - C15</td>
<td>Capabilities</td>
<td>64</td>
</tr>
<tr>
<td>F0 - F15</td>
<td>Offset Register Corresponding to C0 - C15</td>
<td>20</td>
</tr>
<tr>
<td>PC, PF</td>
<td>&quot;Program Counter&quot; (Capability/Offset Register Pair)</td>
<td>64/20</td>
</tr>
<tr>
<td>CC, CF</td>
<td>Context Object Capability / Top-of-Stack Offset</td>
<td>64/20</td>
</tr>
<tr>
<td>TC, TF</td>
<td>Task Object Capability/Offset</td>
<td>64/20</td>
</tr>
<tr>
<td>TP</td>
<td>Task Priority</td>
<td>12</td>
</tr>
<tr>
<td>S0 - S3</td>
<td>System Call Arguments</td>
<td>32</td>
</tr>
<tr>
<td>CCS, CCF</td>
<td>Capability to Context Stack Object and Top-of-Stack Offset</td>
<td>64/20</td>
</tr>
<tr>
<td>CH, CF</td>
<td>Capability to Handler Stack and Top-of-Stack Offset</td>
<td>64/20</td>
</tr>
<tr>
<td>CUSR</td>
<td>Capability Unit Status</td>
<td>32</td>
</tr>
<tr>
<td>EUSR</td>
<td>Execution Unit Status</td>
<td>64</td>
</tr>
<tr>
<td>T0 - T9</td>
<td>Used by Microprogram</td>
<td>32</td>
</tr>
</tbody>
</table>

NOTE Register Sizes Exclude Tag Bits.

Figure 4.1 Registers in the Proposed Architecture
4. 2 Active and Passive Objects

We will classify objects in the system into two groups, depending on how they are used over time:

- **Active objects**: these are objects that have been created or used recently. The notion of ‘recently’ in this context will be explained later, in Section 4. 4. Active objects can exist in the main memory or in the secondary storage.

- **Passive objects**: these are objects that have not been used recently. Passive objects should be distinguished from inaccessible objects. The latter type of objects are effectively garbage since no one holds a capability to an inaccessible object. In contrast, processors or users can still hold capabilities to passive objects. Accessible objects that have not been used recently are (implicitly) passivated by the system under circumstances described on Section 4. 4. Active objects can be passivated explicitly using an instruction designed for that purpose. On being passivated, objects are archived in secondary storage. One way to view such passive objects is to consider them as objects stored in the filing system, for long-term storage.

A single capability format is used to address both passive and active objects. However, the interpretation of object names in the corresponding capabilities are different. We will refer to capabilities to active and passive objects as *active capabilities* and *passive capabilities*, respectively. Our notion of active and passive objects are identical to those employed in the Hydra system [67] and in the Intel iAPX 432 system [29]. The conditions under which passivation occurs implicitly in our system are, however, quite different from these and those of any other existing system.
4.3 Capability Format

Figure 4.2 depicts the common format of capabilities in our system. Each capability occupies two consecutive words. Bit 33 in both words is set to zero to identify these words as parts of a capability. Bit 32 (in both words) is the initialization flag (Section 4.1.1). Bit 31 in these two words are set to 0 and 1 respectively, to indicate the order on which two consecutive words come together to form a capability. We will refer to these two bits as the sequence tags, for obvious reasons. The hardware uses and sets these two sequence tag bits for loading and storing words forming a capability in a consistent way. In effect, bit 31 in both words prevents the interpretation of two consecutive words, one from each of the two neighboring capabilities, as a valid capability.

Bits 30 through 28 in the first word of a capability is a 3-bit field designating the type of the capability and/or the type of the object to which this capability points. It is customary to omit this information within the capability itself since a duplicate exists in the corresponding entry in the capability mapping table. We, however, include the type information as a part of the capability for a variety of reasons:

a) To facilitate instruction-operand compatibility checks without the need to access the capability mapping table (CMT).

b) To enable passive capability usage to be trapped, without the need to access the CMT.

c) To facilitate debugging.

The interpretation of the 3-bit type code field in a capability is as shown in Figure 4.2. The type code assignment for a passive capability and a capability to a refinement object are
Word Tag (0 = Capability)  
Initialization Tag (1 = Initialized) 
Sequence Tags 
  Representation Access Flag ("raf")  
  Type Seal Flag ("tsf")  
  Indirect/Direct Capability Indicator

<table>
<thead>
<tr>
<th>Type Codes</th>
<th>Access Codes For All Objects Excepting SCOs</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 - Passive Capability</td>
<td>Bit 0 - &quot;Read&quot; Privilege</td>
</tr>
<tr>
<td>001 - Generic Object</td>
<td>Bit 1 - &quot;Write&quot; Privilege</td>
</tr>
<tr>
<td>010 - Code Object</td>
<td>Bit 2 - &quot;Execute&quot; Privilege</td>
</tr>
<tr>
<td>011 - Structured Code Object (SCO)</td>
<td>Bit 3 - &quot;Destroy&quot; Privilege</td>
</tr>
<tr>
<td>100 - Port Object</td>
<td>Bit 4 - &quot;Lock&quot; Privilege</td>
</tr>
<tr>
<td>101 - Virtual Storage Object (VSO)</td>
<td>Access Code Field For SCOs Name</td>
</tr>
<tr>
<td>110 - System Object</td>
<td>One of 32 &quot;Exported&quot; Procedures By Number</td>
</tr>
<tr>
<td>111 - Refinement Object</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. 2 Capability Format
000 and 111, respectively. These two assignments facilitate the hardware implementation of circuitry that provides special treatment for these two types. The type code assignment for the other types are not critical, and are as shown in Figure 4.2.

Bits 27 through 23 in the first word of a capability is a 5-bit field designating the access mode allowed using a capability. For a capability to a structured code object (Section 4.8.2), the interpretation of this field is different. Figure 4.2 also shows the interpretation of the access code field for all other objects. In these objects, bits 24 and 23 in the access code field deserves special mention. If bit 24 is set, the possessor of the capability has the right to destroy the corresponding object and perform all other operations associated with this right. Normally, only the owner of the object would have this privilege. Bit 23 controls the locking of the object addressed by this capability. We will refer to this as the 'lock' privilege. If the lock privilege is missing, the corresponding object cannot be locked (using the LOCK instruction, described in Chapter 5).

Bit 22 in the first word of a capability is the \textit{representation access flag} (\textquote{raf}) and is fundamental to the implementation of user defined abstract types. When this flag is set, all instructions relevant to the type of the corresponding object and consistent with the privileges in the access code field can be performed on the object. When the raf flag is reset, all operations on the object are prohibited. The raf bit is thus useful in controlling access to the underlying representation of an abstract type, and thus requires that it be set or reset under tightly controlled situations. Section 4.9 will describe the implementation of abstract types further.

Bit 21 is called the \textit{type seal flag} (\textquote{tsf}) and is used for converting a generic object to a typed object, as discussed later in Section 4.7.4. Bit 20 in the first word of a capability
indicates whether a capability is an indirect capability or not. The interpretation of this bit is valid if and only if the capability points to a generic object. For capabilities to all other types of objects, this bit is set to indicate that capability is a direct one. This bit is used to solve the revocation problem (Section 2.3) in our system.

Bits 19 through 0 in the first word of a capability and bits 31 through 0 in the second word of the capability designate the least significant 20 bits and the most significant 31 bits of the 51-bit unique identifier (uid) for the corresponding object. The way we generate this uid has serious ramifications on the performance of capability translation and on the complexity of the cache of translated capabilities (capability TLB). We discuss the process of uid generation in the next section.

4.4 Object Name (uid) Generation

In Section 2.4, we looked at the shortcomings of existing and earlier capability-based systems with regard to capability translation and swapping. The manner of generating uids for active capabilities in our system overcomes these problems and ensures fast capability-translation and the absence of in-form/out-form conversions during swaps.

4.4.1 Object usage pattern in capability-based systems

We are interested in the pattern in which objects are used over time in a capability-based system, since our uid generation scheme makes effective use of it. So far, only one detailed study was undertaken to observe and report such patterns in object usage. This involved the Hydra/C.mmp system [67]. Although, the architectural characteristics of
Hydra/C.mmp and our system are different, we expect the following object-usage pattern in Hydra to be valid for our system, as well.

Measurements made on the Hydra/C.mmp system during actual and simulated multi-user sessions revealed the following pattern of object usage:

- 98% of the objects are created, used and destroyed without ever being rendered passive.
- The number of active objects in the system is fairly constant and is typically a few thousand at any time.
- Most of the objects are fairly small in size. An average-sized object contains around sixteen capabilities and 130 bytes of data or instructions.

This last fact is really not relevant to the scheme for uid generation that is described in the next section. It is merely included here for the sake of completeness and for reference later.

4.4.2 Generating the uid in an active capability

The contents of a monotonically increasing counter, with a large number of bits, has been employed as a capability uid in some early software and hardware implementation of capability systems [12], [40]. The use of a counter with a large of bits ensure that object names are not repeated over a sufficiently large span of time. Our approach to generating the uid for active capabilities is similar to this. However, unlike these existing schemes, we make a special interpretation of the least significant bits of the counter, as described below.

From the object usage pattern reported in Section 4.4.1, we can expect the number of active objects in our system to be fairly small and reasonably constant in population. We
will assume that for most of the time this number does not exceed $S$, where $S$ is a power of 2, say $2^N$. We will very conservatively assume that $S = 32K$ (so that $n = 15$). A capability mapping table (CMT) with $32K$ entries will thus be needed to map active capabilities to their corresponding physical address spaces. Since current technology provides us with very cheap memory, we can keep all of this CMT permanently in the main store, as a single-level table.

We employ a 51-bit counter, that is monotonically incremented to generated 51-bit uids for active capabilities in the following manner. The least significant 15-bits of this counter are used as an index into the CMT. If this index locates an empty slot, the contents of the 51-bit counter are added to a pool of available uids, for subsequent use as an uid in an active capability. The CMT slot so located is marked as occupied, and this slot will be used later to hold the mapping information corresponding to the uid generated. If the last 15 bits of the counter fails to locate an empty slot in the CMT, the counter is incremented and process is repeated, till an empty slot is found.

The scheme for generating uids for active capabilities, as described above, is implemented by circuitry within the Capability Unit chip. The starting address of the CMT is predefined to this hardware. For fast capability generation, this circuitry operates independently in the background and maintains a pool of available uids. To prevent this circuitry from accessing the off-chip CMT frequently, a bit-map is maintained in the memory to record the status of the 32-K CMT entries (available or occupied). The uid generating circuitry reads one memory word at a time from the bit-map area to obtain the status of 32 consecutive entries in the CMT. (Recall that a memory word in our system can hold 32 bits of information, excluding the tags.) When an object is deleted or passivated, the corresponding entry in the CMT is freed and the aforesaid bit map is updated in that order, so as to prevent incon-
Our scheme for generating uids for active capabilities is similar to the approach taken in the implementation of SWARD [47] and process id generations in some versions of Unix. It guarantees fast capability translation, a very frequent task. Our scheme, however, is different from the one in SWARD or Unix in some important respects:

- The delay in the less frequent incident of generating an uid is avoided by maintaining a pool of available uids, that are either reclaimed (from destroyed or passivated) objects or generated in the background.
- At any time, the least 15-bits of an uid are associated with only one active object. This fact is successfully used to come up with very area-efficient designs for the capability translation buffer (Section 4.6.2) and a cache used for tracing accesses to objects or parts within objects (Section 5.6.3).

There are two possible, though very unlikely, events that are of concern in the context of uid generation. These are addressed next.

Even at 5 to 10 times the object generation rate reported in [58] or [67], it is highly unlikely that the 51-bit uid counter will overflow. (At typical object creation rates, several years will elapse before the counter overflows!) When the counter overflows, in our system, a trap is generated to invoke software routine that would re-assign uids and reset the counter.

Another possibility, which is also very unlikely in view of the typical object usage patterns, is the situation when the CMT is full. When this happens, our strategy would be to passivate objects that have not been used recently. Passivation would necessitate the assignment of a passive form capability to each object being passivated, and the establishment of
appropriate entries in the CMT for passive capabilities. We will assume that a software routine is typically invoked by the operating system to detect if the active CMT is full or not. There is the possibility of another hazard when the CMT is full -- the uid generating counter must be stopped, otherwise it will be incremented continuously (since no empty CMT slot will be found) and overflow.

4.4.3 Detecting use of invalid capabilities

When an object is destroyed or passivated, its entry in the active CMT is de-allocated. It is quite possible that active capabilities to the deleted or passivated object are still held within other objects in the system, and these can be potentially used, albeit erroneously, in an attempt to access the deleted or passivated object. This is a form of dangling reference and should be detected by the system. When the CMT entry for the deleted object is allocated to another object, capabilities to the deleted (or passivated) object can be used to access this newly created object, since the last 15-bits in both capabilities are identical. (Recall from the last section that the last 15-bits of a capability are used to locate its corresponding entry in the CMT.) To prevent such illegal accesses, we store the most significant 36 bits of the 51-bit uid for an object, as part of the CMT entry for the object. On locating a CMT entry using the last 15-bits of the capability issued during an access, we check the most significant 36-bits of the issued capability against the 36 most significant bits of the uid to which the CMT entry belongs. If these are equal, the capability mapping process continues, otherwise a trap, designating the use of an invalid capability, occurs. Notice that the use of a monotonically increasing counter for generating the uids in active capabilities ensures that the most significant 36-bits in each uid is unique. In systems that do not use such all-time unique
uids, all capabilities to a deleted or passivated object must be searched for and invalidated in order to detect dangling references.

4. 5 An Overview of Object Creation and Deletion

In this section, we briefly outline the mechanism for creating and deleting objects in our system. This overview is necessary to facilitate discussions about the capability translation process in Section 4. 6. Details about object creation and deletion will be provided later, in Section 4. 7.

4. 5. 1 Storage management for capability-based systems

Existing capability-based systems treat the problem of managing object spaces no differently from that for managing a segmented object space. There are, however, some fundamental differences between these two problems and the failure of traditional capability-based systems to address these differences leads to serious performance degradation, as explained below.

A capability-addressed object space differs from the traditional segmented address space in several respects:

1) In capability-based systems, a large number of objects (typically 98%, Section 4. 4. 1) are created and destroyed dynamically, unlike segmented systems, where very few segments, if at all any, are created dynamically. Moreover, the objects created are predominantly small in size.
2) In a segmented system, the swap areas for most segments are allocated statically, when
the load module is created. In traditional capability-based systems, such swap areas for
objects are allocated dynamically to accommodate the dynamic nature of object creation
and deletion.

3) Unlike pure segmented systems, the available physical storage areas in capability-based
systems should be cleaned before allocation to prevent the accidental inheritance of
capabilities.

Thus, in traditional capability-based systems, object creation at run-time is an expensive
process. It entails calls (and corresponding context switches) to the software allocators for
primary and secondary storage. On top of this, allocated physical storage must be cleared
before actual usage. Domain switching is a fairly common event in capability-based systems
and entails the creation of a context object (or something equivalent, such as a local name
space [67]) dynamically on every context switch. Figure 4. 3 shows how the overhead for
dynamic object creation affects the domain switching time in systems for which published
results are available. (Most of the other capability-based systems assume the context object to
be pre-allocated and available in a cleaned state. For these systems, the overhead for dynam­
ically ‘re-creating’ a fresh supply of cleaned context objects, when the supply of pre-allocated
context objects becomes exhausted, is presumably very high. Published performance figures
for these system make no mention of this overhead!) For the Hydra/C.mmp system, 43% of
the total domain switching time involves the creation of a context object. For the iAPX 432,
the context object is assumed pre-allocated. Even then, 34% of the domain switching time is
needed to clear five capability slots and the data segment. The ability to create (and delete)
objects with low time overheads is thus of paramount importance in capability-based systems.
4.5.2 Fast object creation and deletion

In this section, we present our approach for creating the predominantly small objects in a capability-based system at a fast rate. We will also describe how larger objects are created using the same basic technique. Our approach for object creation requires the concept and use of a virtual storage object, VSO. A VSO is one of the object types recognized by the hardware and it essentially defines a large contiguous virtual address space. The VSO contains a page map that defines this virtual space by mapping logical page numbers in this virtual space to page frames or corresponding swap areas on the disk. This page table is indexed by the virtual page number. An entry in this page table has the format shown in Figure 4.4, and occupies two consecutive words in the main memory. There are four flags in each such entry, followed by two fields designating physical addresses.

The various fields in a page map table (PMT) entry are as follows:

- A presence bit (bit-31, 1st word), indicating if the corresponding page is in main memory (bit 31 = 1) or in the swap storage (bit 31 = 0).

<table>
<thead>
<tr>
<th>Architecture/System</th>
<th>Context Object Creation/Setup Time</th>
<th>Domain Switching Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.mmp/Hydra</td>
<td>30 mS.</td>
<td>70 mS.</td>
</tr>
<tr>
<td>Intel iAPX 432</td>
<td>334 cycles</td>
<td>982 cycles</td>
</tr>
</tbody>
</table>

Figure 4.3 Context Object Creation/Setup Overhead
Figure 4.4 The Format of an Entry in the Page Mapping Table

- An 'initial fault' bit (bit 30, 1st word) that is initially set when a virtual page is allocated. It is reset on the very first page fault to this virtual page.

- A 28-bit field (bits 27 through bit 0, 1st word), that designates a page frame number if the presence bit is on (bit 31 = 1).

- A 32 bit field (bits 31 through 0, 2nd word), designating a disk block address. This block is the swap area for the corresponding page.

We will explain how these fields are used during object creation and capability translation later.

When a VSO is created (through a software call to the allocator), the swap areas for the virtual pages in the virtual space it defines are also allocated on the swap disk. The swap
addresses are used to initialize the second address field in the corresponding page map table entry and the initial fault flag is set, while all the other flags are reset. A list, indicating the available virtual pages, which we will refer to as the AVSL, is also initialized. The VSO contains this list, the PMT and some other information. The pages forming the VSO are resident in the main memory during the lifetime of the VSO.

Small generic objects, less than a page in size, are created by executing the CREATE instruction. For such objects, the initial fault flag described above does not play any role. The CREATE instruction specifies the following:

1. A capability register, Cj, containing a capability to a VSO.
2. A size for the object to be created, contained in the offset register Fj corresponding to Cj.
3. A capability register, Ck, to hold the capability returned by CREATE to the Created object.

The invocation of the CREATE instruction results in the following:

- A free logical page in the AVSL is located.
- A clean page frame, from a pool of clean frames maintained by the system is allocated for the object, by inserting the frame number in the PMT entry for the logical page located in the earlier step.
- The initial fault bit in the PMT entry is cleared, the presence bit is set.
- An uid is assigned to the object and a Capability Mapping Table (CMT) entry is set up for the object.
To enable the CREATE instruction find a free logical page, a list of available contiguous virtual spaces (AVSL) is maintained within the VSO. This list is sorted by the starting virtual address of each contiguous sub-space, and entries in this list are of the form (starting address of virtual sub-space, size of sub-space). Since most of the objects are small, the page size may be appropriately chosen so that in most instances objects occupy one page only. Given this choice of page size, CREATE can employ a rotating first-fit algorithm to effectively pick up an appropriate virtual sub-space without any external fragmentation and any need to compact the AVSL.

The scenario described above for the CREATE instruction is ideal. It is possible for the following two events to occur:

a) A free logical page is not found for allocation to the object.

b) A clean page frame is not available (for physically allocating the object).

In either of these two cases, CREATE generates exceptions that are appropriately served by the system. The first problem can be avoided by using a number of VSOs or a large-enough VSO. The second situation can be avoided in a similar fashion -- by maintaining an adequate supply of cleaned page frames. Notice, however, that when page frames are de-allocated, they have to be cleaned before re-allocation. This cleaning is done concurrently with normal processing in our system by relegating the cleaning responsibilities to the Garbage Collection Unit (GCU).

Given an adequate number of page frames and a large VSO to start with, most of the predominantly small objects can be successfully created by CREATE without running into these non-ideal situations. This is due to the highly dynamic nature of object creation and deletion in capability-based systems. Experience on past capability-based systems (Section 4.
4. 1 and [67]) does indicate that one can indeed choose a page size to accommodate most of the objects created. In the proposed system, page size is selectable from one of 1K or 4K words to enable some experimentation in this respect.

The DELETE instruction is responsible for destroying an object formed by CREATE. It does so by deleting the corresponding CMT entry, releasing the virtual page occupied by the object (by updating the AVSL in the corresponding VSO). It also resets flags in the PMT entries for the page released by DELETE, and signals the GCU to clean the page frame that was allocated to the object being deleted. (The GCU will look at the ‘dirty bit’ associated with a page frame to decide whether it needs cleaning or not.)

Small objects are thus created with very low time overhead in our system for the following reasons:

1) Page mapping tables for objects carved out of the virtual space defined by a VSO are defined and allocated once and for all when the VSO is created (using a system call). Hence, when an object is created using CREATE, the page map for the object are found to be already defined.

2) Swap areas for the pages of the virtual space defined by the VSO are likewise allocated when the VSO is created. There is no need to allocate them on a CREATE.

3) Page frames are cleaned by the Garbage Collector Unit (GCU) in the background, concurrently with normal processing.

Object deletion, similarly, is also very fast in our system. The only real deallocation requires released page frames to be added to a list of frames to be cleaned by the GCU, and this is done by the microcode for DELETE.
The technique described above can be easily adapted to create objects larger than a page through a call to the operating system. It will be worthwhile to create objects more than a page long through system calls, since external fragmentation of the AVSL is then possible, requiring compaction and possible changes in allocation. These later operations would require a substantial amount of microcode to implement, so that it is not practicable to have an instruction for creating objects more than a page in length. Large objects created in this fashion will also require a system call for deletion for the same reasons. (When DELETE is used to delete a software-created object, it will generate a system call automatically.)

Page frames are allocated to larger objects formed through systems calls only on initial faults to corresponding pages of the object. Recall that each PMT entry had an initial fault flag for this purpose. On the very first access to the object (which is just one page long), the initial fault invokes a microcoded routine that allocates a cleaned page frame, writes its address (frame number) in the corresponding PMT entry, resets the initial fault flag (to indicate that the very first fault has already taken place) and sets the presence flag in this entry. By allocating page frames to larger objects on a piecemeal basis, only when they are needed, we make very conservative use of the global pool of cleaned pages. This gives a better opportunity for the successful creation of smaller objects using CREATE.

The VSOs are allocated on a per-process basis. In principle, it is possible to create objects from the same VSO using both CREATE and system calls. The VSO serves as a convenient structure for keeping track of the storage allocation to a process.
4. 6 Capability Mapping

In this section, we discuss how mapping tables to translate capabilities to physical addresses are set up, and how such tables are used to translate capabilities. In the following discussions, we will assume that an appropriate page size is chosen and the CREATE instruction, as described in the last section, is used to create generic objects. Larger objects, created using system calls are handled in the same way as smaller objects that are created using CREATE, so far as mapping table set ups are concerned. Typed objects will be created by converting generic objects to the required type, and would entail some small changes in the corresponding CMT entry. This will be discussed later in Section 4. 7. 4.

4. 6. 1 The capability mapping table

The Capability Mapping Table, CMT, is used in conjunction with page mapping tables (PMTs) to implement active objects in terms of variable number of pages. The format of entries in a PMT was already discussed in Section 4. 4. 2. In this section, we discuss the nature of entries in the CMT.

Each entry in the CMT occupies four consecutive words and contain the information necessary to locate the PMT for the segment representing the objects, as well as miscellaneous information. if we overlook the primary tags associated with words forming the CMT, then each entry can be viewed as a 128-bit entity with a number of fields. For all but the refinement (and the passive) object types, the format of a CMT entry is as shown in Figure 4. 5. As depicted in Figure 4. 5, the following fields are present:
### Description of Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Allocation Flag (1=Allocated, 0=Free)</td>
</tr>
<tr>
<td>F2</td>
<td>Bits Used in Garbage Collection and Program-Controlled Locking</td>
</tr>
<tr>
<td>F3</td>
<td>Object Type</td>
</tr>
<tr>
<td>F4</td>
<td>Object Size</td>
</tr>
<tr>
<td>F5a</td>
<td>Bits 50 through 47 in Object Identifier</td>
</tr>
<tr>
<td>F5b</td>
<td>Bits 46 through 0 in Object Identifier</td>
</tr>
<tr>
<td>F6</td>
<td>Physical Address of Page Mapping Table for Object</td>
</tr>
<tr>
<td>F7</td>
<td>'Create Code' (See Text)</td>
</tr>
<tr>
<td>F8</td>
<td>Offset for the GMT Entry of the VSO from which the Object was Created</td>
</tr>
<tr>
<td>F9</td>
<td>Refinement Flag (1=Refinement of Object Exists, 0=No existing Refinements)</td>
</tr>
<tr>
<td>F10</td>
<td>Offset for the CMT Entry for the First Refinement, if any</td>
</tr>
</tbody>
</table>

### NOTES

1. Word Tags and Initialization Tags are not shown.
2. Numbers in parentheses refer to field widths in bits.
3. The format of an entry for a refinement object is different from the one depicted above. (See Section 4.10.1.)

Figure 4.5 The Format of an Entry in the CMT
F1) (bit 127) An allocation flag, indicating if this CMT entry is free or allocated (to map a capability).

F2) (bits 126 through 124) A 3-bit field jointly used by the garbage collector and program-invoked mechanisms for locking the object. (Refer to the micro-routine for the LOCK instruction in Appendix I to see how this field is used.)

F3) (bits 123 through 121) A three-bit field designating the type of the object. (Type code assignments were discussed earlier in Section 4.3.)

F4) (bits 120 through 101) A 20-bit field indicating the size of the object. Objects can thus be up to $2^{20}$ words or 1M words. (Entities larger than $2^{20}$ words would be implemented as multiple objects.)

F5a) (bits 99 through 96) A 4-bit field containing bits 50 through 47 in the uid of the corresponding capability. (Bit 100 is not used.)

F5b) (bits 95 through 64) A 32-bit field containing bits 46 through 15 in the uid of the corresponding capability. Thus, fields F5a and F5b together hold the most significant 36 bits in the uid.

F6) (bits 63 through 32) A 32-bit field designating the physical address of the PMT entry for the first logical page (page 0) of the object. Recall from our discussions in Section 4.5.2 that this PMT resides in the VSO from which the object was created, and that PMT entries for consecutive logical pages of the object occupy physically consecutive storage locations.

F7) (bit 31) A flag that indicates if the corresponding object was formed using the CREATE instruction (flag = 0) or allocated as a result of a systems call (flag = 1).
F8) (bits 30 through 16) This 15-bit field locates the CMT entry of the VSO from which the object was carved out. The information in this field is used to return the virtual space occupied by the object to the space defined by the corresponding VSO when the object is deleted. (Refer to Section 4.5.2 for clarifications.)

F9) (bit 15) A flag that indicates if any 'refinement' was formed from the object corresponding to this CMT entry.

F10) (bits 14 through 0) When F9 is 1, this 15-bit field designates an offset that locates the CMT entry for the very first refinement formed from this object. If F9 contains a zero, this field designates a null pointer. The use of this field will be explained later in Section 4.10.1.

The order of allocating fields F0 through F10 in the CMT entry for an object is not random. Recall that the CMT entry is a found-word entity that is fetched one-word-at-a-time into the capability unit. The fields in a CMT entry are allocated in an order to permit overlapping within the capability translation process.

An obvious improvement for mapping references to objects, which are at most a page long, would be to store the frame number in which it resides in its CMT entry instead of storing the base address of its PMT entry. In an actual implementation of the system, we will assume that this is done. However, in all of the subsequent discussions, will assume the entry format described earlier for the sake of clarity.

The CMT is always resident in main memory and occupies $2^{15} \times 4$ or 128K words. Although, this seems to overtax main memory occupancy, we prefer to keep the CMT in the main memory for these reasons:
1) It permits direct access to the CMT entries (for all active objects) and enables faster capability translation.

2) Memory cost is certainly not an issue today, and one can pay for large memories in order to gain performance.

3) The main memory area occupied by the CMT is a very small fraction of the $2^{32}$-word physical address space.

The system maintains a bit-map for recording the usage information for objects. There is a bit corresponding to every CMT entry in this bit map. On accessing an object, its corresponding bit is set. The operating system periodically resets this bit to keep the reference information current. This information is used by the system during passivation.

We will discuss the nature of CMT entities for refinement objects later in Section 4.10.

1. Passive objects do not have any entries in the CMT. Their entries are found in the Passive Capability Mapping Table (PCMT).

4.6.2 Capability TLB

In this section, we describe the structure and operation of the capability TLB (Translation Look-aside Buffer). The capability TLB is a fully-associative cache memory, maintained within the capability unit, and holds recently translated addresses of logical pages within an object. This TLB is used to avoid the repeated translation of a recent capability reference to a logical page within an object. The uid generation scheme described in Section 4.4.2 ensures a very simplified structure for this TLB. This will be evident from discussions that follow. The discussion applies to capability references to all but refinement objects. (Section
4. 10 will describe the use of the TLB in mapping references to refinement objects.)

Aspects of TLB organization and operation for capability references to refined objects will be discussed later in Section 4. 10. 2.

Figure 4. 6 depicts the structure of the TLB and some associated circuitry. As shown, a capability reference specifies a capability register and an offset register. Since object sizes are limited to $2^{20}$ words, offset registers are 20-bits wide. Assuming a page size of $2^P$ words, a 20-bit offset can be considered as being composed of a (20-p) bit (virtual) page number (in the most significant 20-p bits) and a p-bit line number. An entry in the TLB has the following fields:

- A bit that indicates the validity of the entry.
- The most-significant 36-bits in the uid of the capability.
- The line limit within the virtual page being referenced.
- The frame number to which the logical page maps.

The TLB is queried with the last 15-bits in the uid of the capability and the (20-p) bit page number as key. A TLB miss prompts the capability translation process. A TLB hit retrieves a valid entry into the TLB output register. Once the TLB entry is retrieved, the following comparisons are concurrently done by the hardware:

a) The most significant 36-bits in the uid of the issued capability are compared with the corresponding bits in the retrieved TLB entry. An inequality implies the use of an invalid capability and sets an exception flag.

b) The access codes and the object type information in the capability being used are compared with the object type information and the access mode for the operation requested
Figure 4.6 The Capability TLB and Associated Circuits
by the Execution Unit. (On decoding an instruction, the Execution Unit forwards the access mode requested and the appropriate object type needed for executing the instruction to the Capability Unit.) Any incompatibility in the types and access codes between the requested operation and that in the issued capability implies an error and sets appropriate exception flags.

c) The line number in the offset register is compared with the line limit in the retrieved entry. If the latter is lower, an addressing violation has occurred and appropriate exception flags are set.

If any exception flag was set during (a), (b) or (c) above, an interrupt is generated by the Capability Unit. When this happens, the Execution Unit suspends the current task and switches context to the appropriate exception handler, based on the exception flag(s) set in the Capability Unit. If the result of comparisons (a), (b) and (c) did not set any exception flags, the required physical address is formed by concatenating the frame number (from the retrieved TLB entry) with the line number in the offset register, and the appropriate memory operation is invoked. Concurrently with a successful address translation as described above, the hardware associated with the TLB updates the replacement information for the TLB entry involved and the page frame referenced.

From the nature of the TLB operation described above, the impact of the uid generation scheme (described in Section 4. 4. 2) on the structural complexity of the TLB should be obvious. Notice that instead of using the entire uid (51 bits) as part of the TLB key, we are using just 15-bits of the uid. This saves us considerable area on the chip: since at least 12 to 15 transistors are used to implement every bit of an associate memory [12], [35] versus 6 transistors for an ordinary cell, we are saving \( N \times 36 \times 6 \) to \( N \times 36 \times 9 \) transistors with an N-entry
TLB. Notice however that in using only the last 15-bits of the uid as a key, it becomes necessary to compare the most significant 36-bits later. This does not imply any extra time overhead, since some other comparisons (at least comparison (c) described above) are needed once a TLB entry is retrieved.

The associatively-searched fields of the TLB are the 15-bit least significant uid bit field and the (20-p)-bit logical page number. This is the key used to probe the TLB for resolving a capability reference. We will assume, very conservatively, that 12 transistors are needed to implement one bit of an associative memory cell, 6 transistors to implement a RAM bit and that 10% of the total transistors are used for the various drive and control circuitry associated with the TLB, the total number of transistors needed by a N-entry TLB is: (15 + (20 - p)) * 12 * N (associative part) + (36 + p + (32 - p)) * 6 * N (non-associative part-RAM) + 10% of sum of above two (control, drivers), giving a total of 1.1 * (828 - 12p) * N transistors. The choice of N and the replacement policy are the two major factors that decide the performance of the TLB. Only a detailed simulation using realistic program traces would indicate an acceptable value for N and what would be an acceptable replacement policy. Since such a simulation is well beyond the scope of this thesis, we will make a choice for N and the replacement algorithm based on published results and experiences of others. As reported in [4], contemporary 32-bit micros use 16 to 32-entry TLBs to achieve a very high hit ratio. Estimates made for the Intel iAPX 432, as reported in [8] also indicates that a 16-entry TLB will achieve hit ratios in excess of 95% for many popular benchmarks. We will assume a value of 32 for N for our system.

The least recently used (LRU) replacement algorithm can be implemented fairly easily in hardware [27] and trace-driven simulations indicate that it performs very well in most
instances [27]. In view of this, we will recommend a hardware-implemented LRU replacement policy for the capability TLB.

Using \( N=32 \) and \( p=10 \), the latter corresponding to the smallest programmable page size (1K words), the number of transistors needed by the capability TLB is \( 1.1 \times (828 - 12 \times 10) \times 32 \), or, approximately 25K. If all 51-bits of the uid in a capability was used as part of the key to the TLB, another 9K transistors would have been needed. The uid generation scheme described earlier permits us to use only the last 15 bits in the uid of a capability as the key and results in saving about 30% transistors (and corresponding chip area). If dynamic cells are used to implement the non-associative part of the TLB, this saving becomes even more dramatic.

A number of privileged instructions are associated with the TLB. These are as follows:

- **PURGE-ALL**: A parameterless instruction that invalidates the contents of the TLB.
- **PURGE-FRAME**: Invalidates a TLB entry for the specified page frame. The frame number is specified in any one of the offset registers. This instruction is invoked by the swapping routine when a page is swapped out.

We will describe the operation of the TLB in mapping references to refined objects in Section 4.10.2.

### 4.6.3 Capability mapping

Having described the TLB operation, we are now in a position to discuss the steps involved in translating a capability reference, which also involves the setting up of a TLB entry for the reference. Again, our discussion would apply to all object types, excluding the
refined objects (which we discuss in Section 4.10).

Figure 4.7 depicts an overview of the algorithm employed in translating a capability reference to a physical address. The microprogram corresponding to this basic algorithm is given in Appendix I. This microprogram takes into account the fact that a CMT entry occupies four consecutive words and overlaps parts of the capability translation operation with the memory fetches for the second, third and fourth words forming the CMT entry. This micro-routine is executed by the Capability Unit microcontroller on a TLB miss. Each step in the capability translation microprogram shown in Appendix I takes one cycle to execute (under the assumptions made about the microarchitecture in Chapter 3), so that in the best case, a capability translation takes 12 cycles. We have assumed that a memory operation takes two cycles, which is fairly representative of current RAM technology. Exceptions generated within the execution of this routine save the microprogram status and the microinstruction status to allow resumption of the instruction that prompted the capability translation, if possible.

4.7 Details of Object Creation and Deletion

In this section, we elaborate on the material presented in Section 4.5 earlier and provide more insight into the object creation and deletion processes.

Recall from Section 4.5 that small generic objects up to a page in size are created on-the-fly using the CREATE instruction. The CREATE instruction simply carves out the necessary virtual space from a larger virtual space defined by a VSO. Larger generic objects and VSOs are created in the more traditional way, using system calls. All objects are deleted
1) Access the CMT using the last 15 bits in the uid of the issued capability and fetch the CMT entry into working registers.

2) Set appropriate exception flags and raise an interrupt under any of the following conditions:
   - If the access codes allowed in the capability do not allow the requested operation.
   - If the type of the object (and capability) is not consistent with the operation requested.
   - If the CMT entry fetched is invalid.
   - If the most significant 36 uid bits (field F5) in the CMT entry does not match the corresponding bits in the issued capability.
   - If the specified offset is higher than the size of the object.
   (On an interrupt generated under any of these conditions, the offending operation is suspended and the operating system takes appropriate corrective actions.)

3) Add the page number specified in the offset to the PMT base address (field F6) and use the resulting address to access the PMT entry for the required page.

4) If the specified page is not in main memory, set the appropriate exception code, copy the 'initial fault' flag from the PMT entry retrieved and raise an exception to invoke the page fault handler. (If the 'initial fault' flag is on, the operating system invokes the 'initial fault' handler, which will allocate a clean page frame and reset the 'initial fault' flag in the PMT entry. If the 'initial fault' flag is reset when the exception is raised, the normal swap routine is invoked to swap in the missing page.)

5) Concatenate the line number in the offset with the frame number and start the required memory operation. Concurrently, establish a TLB entry for the page after selecting a victim entry (using the LRU hardware).

Figure 4.7 Overview of the Capability Translation Sequence

using the DELETE instruction. Sections 4.7.1 through 4.7.3 describe the VSO, the CREATE and DELETE instructions. All objects in our system are first created as generic objects and then converted to an object of the appropriate type using the CONVERT or BIND-TYPE instruction. These instructions are described in Section 4.7.4.
4.7.1 The virtual storage object (VSO)

The Virtual Storage Object is an object that is created through a system call and has the format depicted in Figure 4.8. The VSO is essentially a data structure that contains the following information:

- The status (allocated or free) of logical pages within a large contiguous virtual space. This status information is given in the form of a bit-map, called the Available Virtual Space List (AVSL), with one bit for each logical page in the virtual space. We will, hereafter, refer to this virtual space as the VSO-space.

- A page-mapping table, PMT, that is used to map every logical page in the VSO-space to a physical storage location. The format of entries in this PMT was described earlier, in Section 4.5.2.

- Miscellaneous information, in the header region of the VSO, as shown. These include a word that gives the size of the AVSL in words, an offset that gives the starting physical address of the first word of the AVSL, and a word that serves as a rotating bit pointer within the AVSL. The use of these entities will be described later.

The VSO itself is created using a system call and occupies at most one physical page. For convenience in the implementation of the CREATE instruction, we will restrict the size of the VSO-space to multiples of 32. Since each memory word can contain 32-bits of information, the bit-map AVSL will then require an integral number of words. If $S$ is the size of the AVSL in words, the total number of words needed to implement the VSO is $= 3$ (for the header) $+ 64 \times S$ (for the PMT, with 2 words per PMT entry) $+ S$ (for the AVSL) $= 65 \times S + 3$. Restricting the size of the VSO to a physical page (whose least size is 512 words),
Figure 4.8 The Virtual Storage Object

allows a maximum value of 7 for S. This means that the maximum size of the VSO-space is 32 * 7 pages or 224 pages, which is acceptable.

The bit-map nature of the AVSL provides a compact representation of the status of logical pages in the VSO-space. Bit i in this bit vector designates the status of logical page
number $i$ in the VSO-space: bit $i$ is 0 or 1, depending on whether the $i$-th logical page is free (available) or allocated, respectively. As indicated earlier, the bit-vector forming the AVSL may be spread out over a number of consecutive words, each containing 32 consecutive elements of the vector. In such a case, the $i$-th bit in the AVSL is located at bit position $(i \mod 32)$ within the $(i \div 32)$-th word of the AVSL. The bit map representation of the AVSL enables the status of 32 consecutive logical pages in the VSO-space to be obtained with a single memory reference and permits very compact and efficient allocation and de-allocation algorithms for implementing the virtual space of objects up to a page long.

4. 7. 2 The allocation and de-allocation algorithms

The allocation algorithm carves out a single logical page from the VSO-space to implement the predominantly small objects which are up to a page long. The de-allocation algorithm is employed to return a logical page to the VSO-space. A simple combination of shifts and logical operations permit the allocation and de-allocation algorithms to be implemented very efficiently, in microcode. The allocation algorithm employs a rotating first-fit allocation policy for the following reasons:

1) Rotating first-fit is easier to implement compared to other allocation policies.

2) As observed by Knuth (see [27], for instance), the simple first-fit algorithm is an acceptable strategy in many instances.

3) Normally, first-fit would create smaller holes at the top of the free list, and allocation would thus take longer on the average. This is not a problem in our system for two reasons. First, since most objects created are at most a page in length, all 'holes' are usable. Second, the use of a rotating first-fit strategy prevents localization of allocated
virtual pages.

The following simple example illustrates the basic idea behind the allocation and de-allocation algorithms: Assume, for the sake of the example, that the AVSL is just 8-bits long and has a current value of 11100101. This value of the AVSL-vector indicates that logical pages 3, 4 and 6 in the VSO-space are free, while logical pages 0, 1, 2, 5 and 7 are allocated. We will also assume, for simplicity, that a simple first-fit allocation strategy is employed. To allocate a page, we first form a Template of the bit pattern 10000000 (i.e., \(10^7\)). To determine if the very first page in the VSO-space is available, we check if the very first bit in the AVSL is zero. This is simply accomplished by testing if AVSL AND Template equals Template. (AND is the bit-wise logical 'and' operation.) In this case, this test fails. To continue our search, we first rotate the Template right by 1-bit position, getting 01000000 and perform the test again. We continue doing this till the test succeeds or till we have scanned all bits in the AVSL, whichever occurs earlier. For this example, the test described above succeeds after 3 rotations of the Template (resulting in a value of 00010000 for the Template). This indicates that bit 3 in the AVSL is zero, and the corresponding page can be allocated. If page 3 is allocated to reflect the allocation, we complement the Template (setting it to 11101111) and 'or' this Template to the AVSL (setting the latter to 1110101).

Deallocation, again, is a simple process -- if logical page 'j' is freed, we form a template with 0 in the j-th bit position and a 1 in all others. We then 'and' this template to the AVSL to set bit j in the AVSL to 0.

In our system, the AVSL occupies multiple words and a rotating first-fit algorithm is used. Figure 4.9 shows the allocation algorithm employed in our system, while Figure 4.10 depicts the algorithm employed to deallocate logical pages to the VSO. In the best case,
AND, OR, NOT are bit-wise operations */

```plaintext
count := 1; /* counts total # of shifts */
bitcount := rot_position;
b := bitcount mod 32;
w := bitcount div 32;
/* this determines starting word and bit position for search */
found := false;
Template := 0\[31]1;
shift 'Template' left by (31-b) bits;
Template := NOT Template;
/* Complements 'Template' */
/* - this has the effect of: Template := 0\[31-b] */

L: Fetch word at offset 'w' in AVSL into T0;
while (count <= 32 * size) and not found do
  if T0 AND Template = Template
    then found := true /* free page found */
    else /* continue searching */
      begin
        count := count + 1;
        bitcount := bitcount + 1;
        Rotate 'Template' right by 1-bit;
        if bitcount mod 32 = 0
          then /* get next word and continue */
            begin
              if w = size
                then /* next word is to be obtained by */
                  begin /* wrapping around... */
                    w := 1;
                    bitcount := 0
                  end
                else /* w <= size */
                  w := w + 1; /* next word is physically adjacent */
            go to L
          end /* if bitcount mod 32 = 0... */
        end /* else, while */
    end /* /* continue on next page... */
```

Figure 4. 9 The Allocation Algorithm for CREATE
if not found then exit, raising exception (unable to allocate);
if bitcount = 32 * size
then rot_position := 0
else rot_position := bitcount + 1;
/* updates rotating pointer */
write 'rot_position' back to VSO header;
T0 := T0 OR (NOT Template);
write T0 to location at offset 'w' in AVSL;
/* updates AVSL to reflect new allocation */
PMT_base := bitcount + 2 + VSO_base;
PMT_size := object_size;

Figure 4.9 (continued)

the microcoded version of the allocation algorithm will execute in 17 cycles (7 cycles for the
prelude to the loop, 3 cycles in the loop and another 7 cycles in the loop postlude). The
microcoded version of the de-allocation algorithm will always execute in 7 cycles.

4.7.3 The CREATE and DELETE instructions

The CREATE instruction is used for creating generic objects less than a page in size.
The CREATE instruction has the form:

    CREATE Cj, Ck

where Cj holds a capability to a VSO from which the object being created is to be carved
out; the capability in Cj has both the 'read' and 'write' access privileges. Fj is the offset
register associated with Cj, and contains the size of the object to be created in words. Ck is
a capability register which will eventually hold a fully privileged capability to the newly
/* AND, NOT are bit-wise operations */
/* Request to deallocate the k-th logical page of the VSO-space */

w := k div 32;
b := k mod 32;

/* Obtain word offset and bit position */
Fetch word at offset 'w' in AVSL to T0;
Template := 0^31 1; /* 31 zeros, followed by a 1 */
shift left Template (31-b) bits;
Template := NOT Template ;
/* Complements 'Template' */
T0 := T0 AND Template;
write contents of T0 to location at offset 'w' in AVSL;

Figure 4. 10 The De-allocation Algorithm for DELETE

created object.

The algorithm employed by the CREATE instruction is given in Appendix I. This algorithm gives sufficient detail to enable us to make an estimate of the best case timing. Assuming again that the micro-architecture permits two logical shifts or arithmetic/logic operations or two register transfers in one micro-instruction cycle, and two cycles for a memory operation, the best case execution time for a CREATE instruction is 50 cycles. As observed earlier in Section 4.4.1, dynamically created objects are predominantly small. This fact, coupled with the use of the rotating first fit allocation strategy is expected to result
in an average execution time (for the CREATE instruction) that is very close to the best case estimate made above. Only a detailed simulation using realistic benchmark programs will reveal how successful the CREATE instruction really is. Unfortunately, such a simulation is well beyond the scope of this thesis.

The DELETE instruction is used for deleting objects that are formed using CREATE, as well as objects formed through system calls. The DELETE instruction deletes all types of objects, excluding refinement objects. (Refinement objects are destroyed using a DESTROY-REFINEMENT instruction, described later.) The DELETE instruction specifies a capability with the 'destroy' access privilege, and is of the form:

\[
\text{DELETE C}_j
\]

where \(C_j\) is a capability register holding a capability to the object being destroyed. Details of the DELETE instruction are given in Appendix I.

The time needed to execute the DELETE instruction is constant and is equal to 36 cycles. As seen from the above timing estimates, small objects can be created and deleted very efficiently using the CREATE and DELETE instructions. Note also that the use of these instructions removes the necessity for making any explicit system calls (and corresponding context switches) at run time, and thus improves performance considerably.

4.7.4 Creating typed objects

Normal users, operating in the user-mode are permitted to create generic, code, and structured code objects. Only the operating system (in the privileged mode) is allowed to create the other object types. Context objects are implicitly created when the protected procedure call instruction, PCALL, is executed. (Refinement objects can be created in either
The sequence for creating typed objects (excluding refinement objects) is the following:

1) Create a generic object using either the CREATE instruction or a system call.

2) Initialize this object, as needed (using a sequence of LOAD and STORE, for example).

3) Associate a type with the object by converting it to an object of the required type. This is done, depending on the type of the object needed (and on the operating mode) by using the BIND-TYPE instruction (in the user mode) and the CONVERT instruction (in the privileged mode).

The BIND-TYPE and the CONVERT instruction have the following form:

BIND-TYPE Cj

and

CONVERT Cj

where Cj contains a capability to a generic object, with the type seal flag (tsf) (bit 22 in the first word of the capability) cleared (refer to Section 4.3). The corresponding offset register, Fj, contains a type code that designates the type to which the object is to be converted.

Appendix I gives details of these instructions. Once a generic object is associated with a type using these instructions, the type seal flag in the corresponding capability is set in order to prevent either accidental or malicious conversion to another type. Notice that even if one intends to use an object as a generic type, BIND-TYPE or CONVERT is executed, specific the generic type as a parameter to prevent accidental or malicious conversion to a different type later on. Notice also from the details given in Appendix I that the CONVERT instruction subsumes the BIND-TYPE instruction. CONVERT can transform generics to all types (excepting the refinement types!), whereas BIND-TYPE transforms generics to only types
allowed in the user mode.

4. 8 The Protected Procedure Call Mechanism

The protected procedure call mechanism allows a call to be made to a procedure, in conjunction with a switch to the called procedure domain. The instruction PCALL is provided for this purpose. A complementary instruction, PRET, allows the return of control to the caller, in the caller’s protection domain. Capability registers C0 through C7, offset registers F0 through F7 and data registers D0 through D7 are used for inter-domain argument transfers during the execution of PCALL and PRET. Registers D22 and D23 are used to convey information about inter-domain parameters or additional parameters. Three types of objects are primarily involved in the execution of PCALL and PRET. These are the context object and the structured code object and a special object of system type, called the context stack object (CSO).

4. 8. 1 The context object

The activation stack of subprogram invocations within a protection domain is implemented within the space defined by the context object associated with that domain. The control register CC (refer to Section 4. 1. 6) contains a capability to the context object for the current protection domain; the corresponding offset register CF points to the top of the activation stack within the current context object. A context object for the called domain is automatically created by microcode (in fact, using the microcode for CREATE) when a protected procedure call is made, with the PCALL instruction.
To save capabilities (from registers) on the activation stack within the current context object, the instruction PUSHC is used. The instruction PUSH is used to save data or offset registers in the activation stack. The form of these two instructions are:

PUSHC Cj

and

PUSH R

where Cj is a user-programmable capability register and R is a user-programmable data or offset register.

As mentioned earlier, the size of the microcode-created context object is restricted to a page. It is possible, therefore, that a PUSHC or PUSH may generate an offset higher than the size of the context, thus generating an illegal address. This situation is identical to an overflow of the activation stack in traditional systems. An exception will be generated by the address translation mechanism in our architecture when such an overflow occurs. One way of dealing with this exception is to let the system software create a larger context object, copy everything from the existing context object to the software-created context object, reset CC to hold a capability to this new context object, set CF as the new activation stack pointer, and resume execution. (The microcode-created context object, which 'overflowed' is, of course, destroyed to de-allocate the storage it occupied.) This approach enables a program to continue executing after it generated a context object overflow. In any case, this overflow is very unlikely to happen for the following reasons:

1) The size of the microcode-created context object is fairly large, at least 1K words.

(Recall that this is the minimum value of the programmable page size.)
2) We had mentioned earlier that calls to exported procedures of a package be protected (i.e., be made with a PCALL). Languages that support the concept of data abstraction in the form of packages tend to use fewer levels of calls within a package [7] -- this is corroborated by the usage reported in [68] for large systems programs in Ada. It is therefore unlikely, that a context object, created by microcode on every PCALL, will overflow.

The instructions for restoring capability and data/offset registers from the activation stack within a context object to registers have, respectively, the forms:

\[ \text{PULLC } C_j \]

and

\[ \text{PULL } R \]

where \( C_j \) is a programmable capability register, and \( R \) is a programmable data or offset register. The PULL or the PULLC instruction may generate an activation stack underflow when it generates a negative offset. The algorithms employed by PUSHC and PULLC, although simple, are given in Appendix I to illustrate how the hardware mechanism for detecting an offset register underflow is invoked. The algorithms corresponding to PUSH and PULL are very similar.

4. 8. 2 The context stack object (CSO)

When a protected procedure call is made, a new context object is allocated for the called domain and the context object to the calling domain must be saved, for restoration on return from the call. To facilitate the restoration of a context object, a special system-created object, called the context stack object, is used. This object contains a stack of capabilities to
active context objects (and their associated top-of-stack pointers). This stack is set up in the following manner. Whenever a protected procedure call is made, as part of the call, after saving all the status of the caller in the callers context object, the CC and CF registers are pushed onto the context stack object. On return from a protected procedure call, the callers context object can be restored by retrieving the saved values of CC and CF from the context stack object. Figure 4.11 depicts the role of the CSO in saving the information for restoring active contexts.

4.8.3 The structured code object (SCO)

The SCO is a system recognized object that is essentially a collection of procedures with up to 32 pre-specified entry points. Typically, the SCO is used to implement a package (abstract data type) and the pre-specified entry points defined up to 32 exported procedures for the package. The SCO can, of course, contain locally called procedures.

Figure 4.12 depicts the format of a structured code object. As shown in this figure, the first 32 words called offset vectors of a SCO contain offsets to the actual entry points for the corresponding exported procedures. Since an object can be at most $2^{20}$ words long, these offsets are all 20-bits in length. If a SCO has fewer than 32 pre-specified entry points (exported procedures), the unused offset vectors are filled with all ones. Capabilities are provided for each one of the exported procedures in a SCO. These are referred to as SCO-capabilities and specify, using a 5-bit number in the access code field, a position in the offset vector (and thus, indirectly, an exported procedure). All SCO-capabilities to procedures within the same SCO use a common uid corresponding to that SCO. Our scheme for representing SCO capabilities is quite different from any other system. In SWARD, for
NOTE
1. Capabilities are shown as one-word pointers for the sake of clarity.
2. The sequence of protected procedure calls shown entails domain switches in the order: A → B → C → D.
3. Refer to Figure 4.1 for register acronyms.

Figure 4.11 Objects Involved in a Protected Procedure Call
Figure 4.12 The Structured Code Object (SCO)
example, the offset within an object is made part of the capability, and capabilities to each of the entry points are formed by appropriate alteration of the offset component [46]. This, however, imposes some restrictions on the way offsets can be manipulated; it also makes the capability wider, with a corresponding negative impact on capability access times. In C.mmp [67] and Cm* [20], a capability with a different uid is formed for each of the entry points, resulting in some duplication in the mapping information and corresponding overhead for mapping table management. In contrast, our system uses a single uid-based capability to uniquely name all of up to thirty-two prespecified entry points for a SCO. The uniqueness is imparted by the 5-bit offset vector position in the access code fields of the capability. Our approach, therefore does not have any of the problems associated with the two schemes described above.

The technique for forming SCO capabilities in our system makes use of the representation access flag in a capability (bit 22, in the first word of a capability) through a FORM-SCO-CAPABILITY instruction. The sequence for forming a SCO and the SCO capabilities are as follows:

1) Create a generic object (through CREATE or a system call).

2) Insert all the code corresponding to the SCO into this object, form the offset vectors appropriately in the first 32 words of this object.

3) Impart a type to the SCO by using the BIND-TYPE or CONVERT instruction. This returns a capability to the SCO object with the type seal flag (bit 21, 1st word in capability) set and with the representation access flag (raf, bit 22) set.

4) To form SCO capabilities to specific procedures, make a copy of the capability returned in step (3) and use the FORM-SCO-CAPABILITY instruction, specifying the 5-bit code
offset. This, in turn, returns a SCO capability to the specified procedure.

The FORM-SCO-CAPABILITY instruction has the form

\[ \text{FORM-SCO-CAPABILITY } C_j \]

where \( C_j \) is a register containing a capability to the SCO with the raf bit set, and the last 5-bits in the corresponding offset register \( F_j \) designates the offset value to be inserted into the 5-bit access code field. The instruction forms a SCO-capability in \( C_j \); with the raf bit reset. The FORM-SCO-CAPABILITY instruction generates an exception if an attempt is made to use a SCO capability with the raf bit reset. Thus, for SCO capabilities, the raf bit serves an analogous purpose as a type seal flag -- once a SCO capability to a specified entry point is formed, this capability cannot be used to form a capability to another entry point. Details of this instruction are not given, since it is very similar to the CONVERT or BIND-TYPE instruction. Only the owner of a SCO possesses and retains a capability to the SCO with the raf bit set. This bit is used by the DELETE instruction to check the authority of the issuer of DELETE. This also enables the owner to form SCO capabilities as needed, and then distribute them.

4. 8. 4 The PCALL instruction

When a protected procedure call is made, the only information passed to the called procedure are the (actual) parameters of the call. Before effecting transfer to the called procedure is made, one must guarantee that the environment of the caller (which is in the calling domains context object) is not accessible to the callee. This is ensured by creating a new context object for the called domain and by denying access to the callers context object by removing it from the control register CC. We saw in the last section how the context stack
object served as a perfect place for saving the capability to the callers context object.

The sequence for making a protected procedure call using the PCALL instruction is as follows:

1) Push registers to be saved into the (callers) context object (using PUSHC, PUSH).
2) Put outgoing parameters into the parameter registers (C0 through C7 for capabilities, F0 through F7 for offsets, D0 through D7 for data and D22, D23 for information about parameter type or additional data). Clear unused parameter registers using the CLRC or CLR instruction, as appropriate.
3) Execute the PCALL instruction.

The PCALL instruction will perform the actions necessary to save status information about the caller and isolation of the callers environment.

The PCALL instruction has the form:

PCALL Cj

where Cj is a capability register that holds a capability to the called procedure. This capability is a SCO-capability and specifies (by the number in its access code field) an exported procedure in this SCO. The PCALL instruction does the following:

- Saves the PC, PF and status registers within the context object of the caller.
- Uses the control register CCS (and the associated offset register, CCF) to push CC and CF onto the CSO.
- Clears all programmable capability and data registers, excepting the ones designated for parameter transfer.
• Creates a new context object and puts its capability in CC. Sets CF to 0.
• Transfers control to the specified routine.

A protected procedure call should ensure that no information should be passed from the
domain of the caller to the domain of the callee, excepting the explicitly passed parameters.
The instruction PCALL does this by clearing all but the parameter registers. Notice also that
the instruction PCALL does not set up the domain-specific capabilities for the called domain
into the capability registers. This is explicitly done by the called procedure, as needed.

Appendix I gives details of the PCALL instruction. In the best case, when all capability
references needed by PCALL are cached and a context object is created without any
faults, the execution time for a PCALL instruction is 84 cycles. One expects the average
time for PCALL to be very close to this best case time, for the same reasons that were given
to substantiate a similar claim for the CREATE instruction.

The CLRC and CLR instructions alluded to in the above description of a protected pro­
cedure call in the system both specify a literal as operand. This literal is a bit-vector mask
that indicates the parameter registers to be invalidated (i.e., cleared).

4. 8. 5 The PRET instruction

The PRET instruction is used to effect return from a protected procedure call to the cal­
ing procedure and set the prelude for the restoration of the caller’s protection domain. The
PRET instruction is used at the end a return sequence, which is as follows:
1) Put return arguments into the parameter registers. Clear the unused parameter registers
using the CLRC or CLR instruction.
2) Execute the instruction PRET.

The PRET instruction does the following:

- Destroys the context object of the callee.
- Clears all but the parameter registers.
- Restores CC and CF from the context stack object.
- Restores PC, PF and the other status registers and invokes the fetch microroutine.

Details of the PRET instruction are given in Appendix I. In the best case, PRET takes 66 cycles to execute. This corresponds to the case when the references to the context stack object and the earlier context object is cached. It is more likely, however, for the capability reference to the context object of the caller not to be cached. In that case, the execution time for PRET is 77 cycles.

4.9 The Abstraction Mechanism

The mechanism for enforcing data abstraction in our architecture relies on the ability to allow or deny direct access to the underlying representation using a capability. This is done through careful control of the representation access flag in a capability. A critical choice in the design of an abstraction mechanism concerns the times when access to the underlying implementation of an abstract type is enabled and disabled. A secondary, but no less critical, choice concerns the degree of access allowed to the procedures implementing the abstraction. In Chapter 3, we looked at two contrasting examples of the abstraction mechanisms in existing systems that illustrate these design choices. We will summarize them here for conveni-
In the C.mmp/Hydra system, capabilities to representation objects are amplified to the required privilege mode at the time of a protected procedure call as part of the local name space creation process [67]. All representation capabilities passed in as parameters, regardless of whether they are used or not, are amplified, and remain amplified in their respective slots within the local name space during the lifetime of the callee. In contrast, in the Cm*/StarOS system [20], capabilities are amplified (and de-amplified) as and when needed, by the type manager for the user-defined abstract types. In C.mmp/Hydra, an amplified capability has the minimum privileges needed by the procedures implementing abstraction. In contrast, capability amplification in StarOS on Cm* provides the type manager with a fully-privileged capability, which is thus not consistent with the principle of least privilege.

Our abstraction mechanism uses the best features from each of these two extreme examples. In our system, capabilities are amplified (and de-amplified) as and when needed and amplification grants only the minimum necessary privileges to the procedures implementing the abstraction. Our approach is thus similar to the one taken in the Intel iAPX 432 [29], however, it is considerably more simpler but no less powerful.

4.9.1 Implementation of user-defined abstract types

In our system, two types of objects are involved in the implementation of user defined abstract types (packages). These are the SCO, which is a collection of exported and private procedures implementing the abstraction, and an object (or a number of objects) corresponding to the representation (or implementation) of the abstract type.
In the simplest implementation of an abstract type, the object corresponding to the underlying representation is a generic object. In response to a user's request, a procedure for creating an abstract type, which is one of the exported (externally callable) procedures of a SCO, creates a representation object and returns a restricted capability to that object to the user (caller). The representation access flag in this capability is turned off, so that the user is denied direct access to the representation object. The exact sequence of creating a representation object is as follows:

1) Create a generic object (through a CREATE, if the object is within a page in size, through a system call, otherwise).

2) Execute the instruction:

   ASSOCIATE-SCO Cj, Dk

   where Cj contains the capability to the generic object created in step 1 above, its corresponding offset register specifies an offset (in this case zero), and the last five bits in the data register Dk specifies the access privileges needed in amplified capability to the representation object. What this instruction does is to store within the generic object, starting at the address specified, the following:

   a) a 'nulled' capability to the executing SCO in which all the access privileges are absent, and the sequence bit in the second word forming the capability is cleared. (Clearing the sequence bit in the second word of a capability makes it inaccessible at the instruction set level.)

   b) a word whose last 5-bits indicate the minimum access privilege needed to the generic object on amplification.
This step ensures that the generic object created in step 1 is correctly associated with the SCO whose code is executing this sequence.

3) Initialize the rest of the generic object as needed.

4) Use the DEAMPLIFY instruction to turn off the representation access flag and remove all access privileges in the capability to the representation object.

5) Return the de-amplified capability to the user.

Several explanations are due here. The objective behind storing a nulled SCO capability within the representation object in step 3 above has one purpose. It associates the representation object with a particular type definition object, and thus establishes the necessary type binding. The access code stored in the 3rd word of the representation object indicates what access modes are needed ('needed rights') in the amplified version of a capability to a representation object in order to implement the abstraction. We now proceed to describe instructions that amplify and de-amplify capabilities to representation objects in our system.

When an exported procedure is called using the PCALL instruction (or even a simple CALL instruction), specifying the unamplified capability to the representation object as a parameter, one of the first things it does is to gain the necessary minimum access privilege to the underlying representation object. It does so by executing the AMPLIFY instruction. The AMPLIFY instruction has the form:

AMPLIFY CRj

where CRj is a register containing an unamplified capability to the representation object (as supplied by the caller). The corresponding offset register, FRj, contains a value (in this simple case, zero) that specifies the address of the location within the representation object where the nulled SCO capability is stored. The AMPLIFY instruction compares the uid of this
nulled capability against uid in the capability of the SCO object containing the AMPLIFY instruction. (This latter capability is found in the 'program counter' capability register, PC.) If the uids match, the representation capability in CRj is amplified to the required access mode by copying the needed access code bits (stored within the word following the SCO capability within the representation object) to the access code field and setting the representation access flag to one.

The DEAMPLIFY instruction is used to seal off access to a representation object before returning control to the caller (or before non-local call is made by the called procedure, in turn, after a representation capability is amplified). The algorithm executed by the DEAMPLIFY instruction is similar to the one described for the AMPLIFY instruction, and is given in Appendix I. Notice that de-amplification becomes superfluous if the caller retains a copy of the unamplified representation object capability and the called procedure does not call (or pass along the amplified capability to any non-local procedure) any other procedure after amplifying the representation capability. It is also noteworthy that the rights needed after amplification is specified by the creator of the representation object. (The creator is simply one of the procedures exported by the SCO.) This implies that the caller places some reliance on the called procedures that implement the abstraction in regard to the use of the underlying representation. In particular, the caller trusts the called procedures to use only the minimum necessary access privileges in the amplified capability, and use the object in a consistent way. Such a reliance is natural and all existing capability mechanisms for implementing abstraction place a similar trust in the procedures implementing abstraction, in some form or other.
Figure 4. 13 depicts the objects involved in the implementation of abstract types as described above. The time required to execute the AMPLIFY and DEAMPLIFY instructions are 21 and 19 cycles, respectively. This assumes that the capability to the representation is not cached in the TLB. If a TLB hit occurs, AMPLIFY and DEAMPLIFY takes only 10 and 8 cycles, respectively.

4.9.2 More general implementation of abstract types

The implementation of abstract types as described in the last section is not sufficiently general for several reasons. First, it uses only generic objects for the representation. Second, it associates a common minimum set of access privileges with the amplified capability to enable its use by all procedures implementing the abstraction, and thus may violate the principle of least privilege. Third, as described, it does not allow multi-level abstraction, where more than one-level of abstraction is applied to the underlying representation. Finally, since a SCO can contain a maximum of 32 exported procedures, the scheme, as such, cannot implement abstract types with more than 32 associated exported procedures. In this section we present very general solutions towards the implementation of abstract types that overcome all of these deficiencies.

Objects other than generic types could be used to implement the underlying representation in the following manner. A generic object G can be used to provide a top-level abstraction, serving as an object through which one can indirect to the typed representation object. Fully-privileged capabilities to the underlying representation objects (generic types or non-generic types) are stored in the generic object G. Users possess an unamplified capability to the generic object G. Called abstraction procedures gain access to the underlying
"Nulled" Capability to SCO

Needed Rights

Data Corresponding to Underlying Representation of the Abstract Type

Typical Code Sequence for Accessing the Underlying Representation:

```
AMPLIFY (C_k, k)
DEAMPLIFY (C_t, k)
```

NOTE
1. Users of the Abstract Type Hold the Following Capabilities
   - A 'Sealed' Capability to the Generic Object Implementing the Underlying Representation
   - For Every Abstraction Procedure Callable by the User, a Capability to the Procedure in the Form of a SCO Capability Specifying the Procedure by Number.
2. C_k is the Sealed Capability to the Generic Representation Object; C_t is the Corresponding Amplified Capability.

Figure 4. 13 The Implementation of Simple Abstract Types
representation objects A, B and C by amplifying the capability to G in order to gain 'read' access to G and reading off the capabilities to A, B and C. Before returning to the caller, the called procedure invalidates the capabilities to A, B and C (using the INVALIDATE instruction) and de-amplifies the amplified capability to G. As explained in the last section, these invalidation and de-amplification may be superfluous in many instances.

The RESTRICT-PRIVILEGE instruction can be used to ensure that each abstraction procedure uses the amplified representation capability only in the minimum privilege mode appropriate to that procedure. The RESTRICT-PRIVILEGE instruction has two arguments -- one is a capability register holding a capability to a non-SCO object and the other is the corresponding offset register. The last 5-bits in this offset register contains a mask with 0s in the bit position corresponding to the privileges to be removed from access code field in the capability. The effect of the RESTRICT-PRIVILEGE instruction is to merely 'and' the mask bitwise on to the access code field or the capability (bits 27 through 23 in the 1st word forming a capability), thus reducing its access privileges.

When an abstract type has more than 32 associated procedures, these procedures are spread out over a number of SCOs, each with less than 32 abstraction procedures. The representation object will thus have more than one associated SCO. A representation object will also have more than one associated SCO if the abstraction is implemented over several levels. The solution corresponding to these situations, assuming a generic object is used for the representation, is as shown in Figure 4. 14. For each associated SCO, three words (triads) holding a nulled SCO capability and the needed access privileges are stored within the generic object. AMPLIFY instructions within a SCO simply specifies the right offset to locate the corresponding (nulled) SCO capability and needed rights within the representation object.
Figure 4.14 Implementation of a Very General Abstract Type

NOTE

Users of Abstract Type Hold Sealed Capability to Object G and Capabilities to Callable Procedures in SCO #1 and SCO #2.
When the representation object is not generic, multiple SCOs can be handled by storing the triads in the top-level generic object, along with capabilities to the typed representation objects.

4.9.3 Some remarks on the safety of the abstraction mechanism

The purpose of the abstraction mechanism is to deny users of abstract types direct access to the underlying implementation of the type. Our scheme does this in a direct fashion through the use of the hardware-sensed representation access flag in a capability. When the representation access flag (raf) in a capability is cleared, the holder of the capability is forbidden from accessing the corresponding object directly in any fashion. Turning off the raf bit is thus tantamount to temporarily marking the capability as invalid. The raf-bit in an unamplified capability is turned on only under tightly controlled circumstances by the AMPLIFY instruction, within the code of procedures implementing the abstraction. Only procedures within the associated SCO can execute AMPLIFY successfully. One apparent way of breaching the abstraction mechanism is as follows:

1) An user holding an unamplified representation capability sets up procedures within a fake SCO to use an amplified capability to the representation.

2) The user somehow stores a nulled capability to the fake SCO within the representation object, followed by a word designating the needed access code, using one of the legitimate exported procedures, and somehow learns at what offset this nulled capability is stored.

3) A procedure within the fake SCO then uses an AMPLIFY instruction, specifying the offset of the nulled, fake-SCO capability, and thereby executes AMPLIFY successfully.
While this sequence is sound in principle, it relies on something impossible -- the possession of a nulled capability that can be unsuspectingly passed to one of the legitimate exported procedures. The user cannot create a nulled SCO capability to the fake SCO, read it off, and then use it as suggested above, since nulled SCOs cannot be read and saved in registers or objects at the (macro) instruction set level. The only way to store the nulled, fake-SCO capability within the representation object would be to execute the ASSOCIATE-SCO instruction, which requires the execution of ASSOCIATE-SCO to possess an appropriately privileged, amplified capability to the generic object!

Another possible breach of the abstraction mechanism may result when the exported procedures use 'subcontractors', passing to them an amplified representation capability. If the subcontractors are not trustworthy, they may somehow pass the amplified capability back to the original caller, who then comes in possession of an amplified capability to the representation object. This is not to be likely, since the procedures implementing the abstraction does rely on the trust it places in the subcontractors, in the same way the caller relies on the trustworthiness of the procedures implementing the abstraction.

4.10 Refinement Objects

The refinement object is a special type of object that acts as a window to a contiguous subspace within the address space of an already created parent object. The parent object can either be a generic, a code object or even another refinement object. No storage allocation needs to be made for a refinement object, since the storage occupied by the refined object is subsumed by the storage occupied by the parent. An example of the need for having a
refinement object concerns the passing of only some physically consecutive elements of an array to a procedure as parameters. In this case, the callee needs to have access only to the physically consecutive elements passed in as parameters, and not to the entire array (which may be implemented in its entirety by a generic object). The caller can thus create a refinement object comprising of only the physically consecutive parameter elements, and pass along a capability to this refinement (with appropriate access privileges) to the callee. This example shows the major use of refinement objects: they are useful in the implementation of the need-to-know principle, since in effect refinements restrict the span of accessibility within an object. In this section we discuss the formation, use and deletion of refinement objects. Unlike the earlier sections, most of the mechanisms in our architecture related to refinement objects will be described at a comparatively higher level, since the similar details have already been given elsewhere in this chapter.

4.10.1 Creating a refinement object

Refinement objects are formed using the instruction CREATE-REFINEMENT which has the form:

\[
\text{CREATE-REFINEMENT } C_j, C_k
\]

where \( C_j \) contains a capability to the parent object (of type 'generic' or 'code' or 'refinement'), with at least the 'destroy' privilege and \( CR \) is the capability register in which a capability to the refinement object would be returned on successful completion of this instruction. The offset registers \( F_j \) and \( F_k \), corresponding to \( C_j \) and \( C_k \), respectively, specifies the starting (virtual) address and the size of the refinement in the virtual space of the parent. The capability to the newly-create refinement in \( C_k \) has access privileges identical to that in \( C_j \),
and is typed with the type corresponding to the earliest ancestor of the refinement object. The CMT entry of the refinement object created, however, designates the object formed by CREATE-REFINEMENT as ‘refinement’. The reason for doing this is to enable the instruction-issue logic to check the validity of the requested operation against the true type of the refinement object. (A refinement object is ultimately created, perhaps via intermediate refinements, from an object of type 'code' or 'generic'. The true type of a refinement object is thus same as the type of its foremost ancestor.) Before elaborating the effect of a CREATE-REFINEMENT instruction, we describe the relationship among the CMT entries of refinement objects and their parents.

Consider an object A of type ‘generic’ or ‘code’, and two refinement objects R1 and R2 created in that order from it. Consider also objects R3 and R4, created in that order, as refinements of R1, and objects R5, R6, R7, created in that order, as refinements of R2. The relationship among these objects are depicted in the family tree of Figure 4.15 (a). In these set of objects, A is the earliest ancestor of all other objects, A is the parent of R1 and R2, R1 is the parent of R3 and R4 and so on. Also, R1 is the first child of A, R3 is the first child of R1 and R5 is the first child of R2.

When an object is deleted (i.e., destroyed), all of its refinements (and their descendants) should be destroyed. To enable this to be done efficiently, it is useful to have pointers to the CMT entries of all ‘children’ (refinements) of an object in the CMT entry of the parent. Instead of storing pointers to all of the children in the CMT entry for the parent, we store a pointer to the very first child. Then, starting at the CMT entry of the first child, we set up a chain of pointers to the siblings, in succession. To facilitate the type and storage mapping information for refinement objects to be obtainable readily, the CMT entry for every
126

(a) Ancestral Relationships

(b) Relationships Among Corresponding Entries in the Capability Mapping Table (CMT)

NOTE Only the relevant fields of a CMT entry are shown

Figure 4.15 Refinement Objects and their Inter-relationships
refinement object points to the CMT entry for the earliest ancestor. We will now describe
the format of CMT entries for refinement objects, given the need to accommodate pointers as
described above.

The format of CMT entries for objects other than refinements were described earlier in
Section 4.6.1. For refinement objects, the interpretation of certain fields are different.
These fields, and their interpretation are as follows:

- **Field F4:** This field is now split into two fields, F4a (16-bits wide) and F4b (4-bits
  wide). The contents of F4a designate the size of the refinement object (which can thus
  have up to $2^{16}$ or 64K words). The significance of F4b is described next, in connec­
  tion with field F6.

- **Field F6:** This field is now split into two fields F6a (16 bits) and F6b (15 bits). The
  last bit of the 32-bit wide field F6 is not used. The contents of F4b, concatenated with
  the contents of F6a designate a 20-bit virtual address in the address space of the earliest
  ancestor object corresponding to this refinement. This is the address at which the
  refinement starts. The contents of field F6b locates the CMT entry of the earliest ances­
  tor object.

- **Field F8:** Contains a pointer to the next sibling of this refinement if F7 has the value 1.
  If F7 is zero, this field serves as a null pointer.

- **Field F10:** Contains a pointer to the first child of this refinement, if the contents of F9
  is one. If F9 is zero, this field serves as a null pointer.

Figure 4.15 (b) depicts the use of pointers in the CMT entries of refinement objects
shown in Figure 4.15 (a) to indicate family relationships.
The format of the CMT entry for the earliest ancestor object is as described in Section 4.6.1. Field F9 is set to 1 to indicate that a refinement of this object exists. F10 contains a pointer to the CMT entry of the first child (refinement) created from this object.

A fairly high-level description of the algorithm corresponding to the CREATE-REFINEMENT instruction is given in Appendix I. This algorithm clearly indicates that CREATE-REFINEMENT may require the traversal of sibling links, and its execution time is partly determined by the number of already existing siblings.

4.10.2 Mapping capability references to refinement objects

The description of the CREATE-REFINEMENT instruction given in Appendix I indicates that the CMT entry for a refinement object does not contain any field for holding the location of the page mapping table for the refinement. Instead, the page mapping table is located by going to the CMT entry for the root object, using the pointer in field F6b. (One could have stored the location of the PMT for the refinement as part of its CMT entry. The width of the CMT does not allow an extra field for this purpose.) Translating a capability reference to a refinement object thus requires the access of two CMT entries -- one for the refinement object itself and the other for the root object. The algorithm employed for translating a capability reference to a refinement object is fairly straightforward. It first locates the CMT entry for the root object, and computes the virtual address of the specified location in the root's virtual space. It then translates this offset, following the normal capability translation sequence. This algorithm is executed in microcode when the TLB fails to translate the reference (i.e., when a TLB miss occurs). The time taken to translate a reference to a refinement object is higher than the time taken to translate references to all other
types of objects for two reasons. First, it involves the access of two CMT entries (instead of one in the normal case). Second, but not the least, it takes more time to set up a TLB entry for a refinement object. This last claim will become obvious from our discussions in the next few paragraphs.

The process of translating a reference to a refinement object can be speeded up considerably by caching information, for mapping references to the most recent physical page accessed, in the TLB. A TLB entry for this purpose turns out to be somewhat wider than the corresponding entries for non-refinement objects, as we will see very shortly. For objects that are not refinements, a logical page within the address space of an object maps exactly onto a physical page frame. A TLB entry, as described in Section 4.6.2, suffices for mapping references to such objects. A refinement object can start anywhere in the virtual address space of the root object. Thus, a logical page within a refinement object can overlap two consecutive logical pages of its root object, and may thus map onto two different page frames. This situation is depicted in Figure 4.16. With the various offsets as shown in this figure, the physical line address corresponding to an offset 'k' in page 'q' of the refinement object is \((S_1 + k)\) in the lower frame (if \(k \leq t\)) or \((k - t - 1)\) in the upper frame (if \(t < k \leq L\)). If \(k > L\), the offset is illegal. (If 'p' is the number of bits in a line address, then \(2^p = \text{the page size} = S_1 + t\).) We will make use of these relationships between the virtual and physical addresses for a refinement object when designing the corresponding TLB entry format.

Whenever a TLB miss for a reference to a refinement object occurs, we will store an entry corresponding to the physical pages of the root object that contains the logical page being referenced. This entry will have the following fields:
Logical Page 'q' of the Refinement

Logical Page 'p' of Root Object

Logical Page 'p+1' of Root Object

L = Number of Lines in a Page (Except for the Last Logical Page of the Refinement)

k = Number of the Line Referenced in the q-th Logical Page of the Refinement

Figure 4. 16 Offsets Involved in Mapping a Reference to a Refinement

T0) The logical page number (in the refinement object's virtual space).

T1) The lower order 15-bits in the uid of a capability to the refinement object.

T2) The most significant 36-bits in the uid referred above.

T3) An offset indicating the last line number in the logical page that maps to the lower frame. (This offset is shown as 't' in Figure 4. 16.)
T4) An offset corresponding to the last line number on the logical page, shown as 'L' in Figure 4. For all but the last logical page, \( L = 2^P \).

T5) The number of the lower frame.

T6) The number of the upper frame.

Of these fields, T0 and T1 are associatively searched. The purpose of T2 and T3 are exactly as described in Section 4. 6. 2. The extra fields needed for the TLB entry for a refinement object are thus T4 and T5 (or T6). (Recall that a field was needed for the frame number in the TLB entry for non-refinement objects. Thus, either field T5 or T6, but not both, can be deemed as extra.) In terms of actual hardware requirements, the width of each entry is increased by the widths of fields T4 and T5 (or T6), plus one extra bit that distinguishes between TLB entries for refinement objects and entries for non-refinement objects. With a minimum page size of 512 (\( = 2^9 \)) words, and a 32-bit physical address, the width of the field T5 (or T6) is (32 - 9) bits, or 23-bits. The width of field T4 is the same as the number of bits in a virtual address within an object. Since we limited the size of an object to \( 2^{20} \) words, the width of field T4 is 20 bits. Thus, in order to map a reference to a refinement object using the TLB, we have to enhance the width of each entry by (23 + 20 + 1) bits or 44-bits from 94-bits (as discussed in Section 4. 6. 2). This implies an increase of approximately 50% in the width of each TLB entry and an increase in approximately 35% in terms of transistor requirements. Thus, in order to make possible the translation of capability references to refinement objects using the TLB, we are forced to increase the area of the TLB significantly. We will also see, in the next paragraph, that TLB-based translation of references to refinement objects, is expensive from another perspective -- they take twice the time needed to map a reference to a non-refinement object using the TLB.
To map a capability reference to a refinement object specifying an offset ‘k’ within logical page ‘q’ of the refinement, the TLB is queried with the least-significant 15-bits in the uid of the capability and ‘q’ as key. A miss on this query prompts the microcoded translation sequence described earlier. On a TLB hit, the matching entry is retrieved and the following TLB-based translation sequence proceeds:

1) The following three checks are performed concurrently, and appropriate exceptions are generated:
   a) The contents of field T2 are compared with the most significant 36 uid bits for equality.
   b) The access modes required by the operation that generated the reference are checked against the access privilege specified in the capability.
   c) The offset ‘k’ is compared against the contents of field T4 (= L, in Figure 4.16). If k > L, the specified offset is out of range.

2) Concurrently, the following two steps are performed:
   a) If k ≤ t (field T3 contains the value of ‘t’), \(2^p - t + k\) is evaluated, and concatenated with the contents of field T5 to generated the physical address. In this case, the offset locates an address in the lower frame.
   b) If k > t, \(k - t - 1\) is evaluated and concatenated with the contents of field F6 to generate the physical address. The specified offset, in this case, locates an address in the upper frame.

From the sequence described above, it should be obvious that translating a reference to a refinement object using the TLB is a much slower process compared to the translation of capability based references to non-refinement objects. Specifically, the TLB based translation
sequence described above will take at least 3 cycles of a four phase clock (versus just 1 cycle
needed for references to non-refinement objects).

What we have described above is just one way to use the TLB to map a reference to a
refinement object. Several other possibilities exist in this regard. We discuss some of these
in the next two paragraphs.

One possibility would be to store in the TLB the mapping information only for the
frame that was referenced (versus storing mapping information for the entire logical page,
which may be spread out over two frames). In other words, there could be two different
entries for the same logical page in the TLB -- one for the lower part (mapped to the 'lower'
frame) and another for the upper part (mapped to the 'upper' frame). This would save us the
trouble of storing both the upper and lower frame numbers in the same TLB entry, thus
decreasing the width of the entry. At the same time, we need to associate a flag with each
entry to identify it as corresponding to the lower half or upper half. Since the TLB is
queried with the logical page number and the last 15 uid bits, there could be two responders
to such a query -- one being the entry for the lower half and the other being the entry for the
upper half. The TLB circuit logic and the TLB-based translation algorithm will have to be
appropriately modified to handle such multiple responders.

Another approach to the design of a TLB-based translation logic for refinement object
would be to design a completely separate TLB for just the refinement objects. This TLB can
have far fewer entries than the 'main' TLB, which will translate references to all other object
types.

In any event, when refinement objects are being used, it is useful and efficient to save
the overhead of repeated, complete translations of references to refinement objects by using a
TLB. Since the complete translation of a reference to a refinement object may take considerably longer time than translating a similar reference to other object types, the TLB-based translation scheme is likely to be useful even if there are very few references to refinement objects.

4.10.3 Deleting refinement objects

Since a refinement object can itself be the parent of other refinement objects, deleting a refinement object is not a straightforward process. The process of deleting a refinement object is also complicated by the fact that CMT entries for related refinement objects are chained in a manner described in Section 4.10.1. It is because of these complications that we do not provide an instruction for deleting refinement objects in our system. In the proposed system, refinement objects are deleted using a system call, specifying a capability to the refinement with at least the ‘destroy’ privilege. When such a system call is made, the service routine performs the following:

- Marks the CMT entries of the object being deleted and for all refinements created out of it as void. (This entails traversal of the ‘first child’ link and the sibling links.)
- TLB entries of all objects deleted in the earlier step are purged.
- If the refinement object being destroyed has siblings, the link from the predecessor is updated to reflect the removal of the CMT entry for this object. This step entails some searching, since the CMT entry for a refinement object does not have a pointer to the CMT entry of its parent.
• If the refinement object being deleted was the only refinement created from its parent, field F9 in its parents CMT entry is set to zero.

Two comments on the performance of the service routine that deletes the refinement object are due here. First, it takes a variable amount of time to execute, due to searching and link traversing a potentially variable number of CMT entries. Second, since one does not expect too many levels of refinement, the time spent in searching is not appreciable. It is noteworthy, that no de-allocation of physical storage associated with a refinement object occurs during the process of deleting a refinement object. The storage associated with a refinement object is de-allocated at the time of deleting the corresponding root object.

4.11 Rights Revocation

In our system, the basic mechanism for revoking previously granted access rights involves the use of indirect capabilities. We first describe the mechanism for revoking access rights to a generic object.

To grant a revokable access right to a generic object, we first form an indirect capability to it by executing the instruction:

\[
\text{FORM-INDIRECT } C_j, C_k
\]

where \( C_j \) contains a capability to the generic object, and \( C_k \) will hold the indirect capability formed by this instruction. The effect of executing this instruction is as follows:

• An uid (and a corresponding CMT entry) is assigned for the indirect capability, and the uid is stored in the appropriate field in \( C_k \).
• A pointer to the CMT entry for the generic object is set up in the CMT entry allocated in the previous step. The type field in the CMT entry is set to mark this as an entry for an indirect capability.

• The access code and other flags in the capability in Cj are copied to the corresponding fields in Ck.

• The 'indirect capability' flag in the capability in Ck is turned on.

The above sequence results in the formation of an indirect capability, with the same access privileges in the capability to the generic object.

An indirect capability formed as described in the last paragraph can be used to access the generic object to which it (indirectly) points by using the instructions, LOAD-INDIRECT and STORE-INDIRECT. These instructions automatically perform the necessary indirection though the CMT entry for the indirect object. (These instructions, of course, requires the indirect capability to have 'read' and 'write' access privileges, respectively.) To revoke access rights to the generic object, one needs to execute the instruction:

```
REVOKE Cj
```

where Cj contains an indirect capability to the generic object. The effect of this instruction is to simply deallocate the CMT entry for the indirect object. It is assumed here that the owner of the generic object retains a copy of the indirect capability to make revocation possible in this manner. Finally, it is worth mentioning here that the hardware associated with the bus buffer register disallows the LOAD or STORE instruction with an indirect capability.

The mechanism for revoking access rights to non-generic (as well as generic) objects require the owner of the capability to simply invalidate all capabilities to the object, including
the ones in its possession, by renaming the object. This is done by using the RENAME instruction, which has the following form:

```
RENAME Cj
```

where Cj specifies a capability with at least the 'destroy' privilege. The effect of executing this instruction is as follows:

- A new uid (name) and the corresponding CMT entry is assigned to the object.
- All information in the old CMT entry of the object are copied into the new entry, excluding the Fields F5a and F5b (most significant 36 bits of the uid).
- The old CMT entry is de-allocated, making all the capabilities to the object useless.
- The new uid is copied into the corresponding field in the capability in Cj.

Thus, the effect of RENAME is to form a new capability to the object and make all earlier capabilities useless, in effect revoking all previously granted rights. Notice that only the executor of RENAME can still access the object using the new capability in Cj. Obviously, for this scheme to work, the owner must not grant out capabilities to the object with the 'destroy' privilege.

### 4.12 Concluding Remarks

In this chapter we took a detailed look at the user-mode instructions centered around the capability mechanism in our architecture. We looked at mechanisms for creating, using and deleting objects and for mapping capability references efficiently to physical addresses. We also looked at the capability mechanisms and relevant instructions related to the implementa-
tation of protected procedure calls, abstract types, refinement objects and revocable access rights. In the next chapter, we present other interesting instructions in the proposed architecture.
5. OTHER ASPECTS OF THE ARCHITECTURE

In the last chapter, we described the capability mechanism in the proposed architecture and some related instructions. In this chapter, we present instructions provided for data movement, instructions for data manipulation within the Execution Unit, instructions for tracing and debugging, instructions for supporting Operating Systems functions and miscellaneous other instructions.

5.1 Data Movement Instructions

As mentioned in Chapter 4, data manipulation is done entirely within the Execution Unit (EU), using register-based instructions. The registers D0 through D23 within the EU serve as the source and destination of operands and results, respectively, for all data manipulation instructions. Operands are fetched into one of these registers using the LOAD instruction from memory resident generic objects. A STORE instruction is used to store the contents of one of registers D0 through D23 into generic objects. Instructions also exist for moving data among various registers. We discuss such data movement instructions among registers and memory resident objects and among registers.

5.1.1 The LOAD and STORE instructions

The LOAD instruction is of the form

\[ \text{LOAD} \ D_j, \ C_k \]
where Dj is the register to be loaded, Ck is a capability register containing a valid capability to an object of type generic, code, or SCO with at least the 'read' access privilege and the representation access flag enabled. The corresponding offset register Fk specifies a location within the generic object (whose capability is in Ck), whose contents are to be loaded into Dj. On successful execution, a LOAD instruction also turns on the initialization flag associated with the specified data register. The LOAD instruction can generate the following exceptions:

- Exceptions involved in translating the capability reference (Ck, Fk), as described in the last chapter.

- An 'attempt to load capability into data register' exception, when parts of a capability are fetched into the EU (instead of a non-capability item). Circuitry associated with the memory bus buffer within the EU would be used to detect such a violation and suppresses the transfer of the fetched word into the target data register.

- An 'uninitialized data access' exception, if the word fetched does not have the initialization tag set (Section 4. 1. 1). This check would be performed by the circuitry associated with the memory buffer in the EU.

- An 'illegal attempt to use a indirect capability' exception if the indirection flag in the capability is on.

The LOAD instruction takes 3 cycles to execute under favorable conditions, 1 cycle for translating the capability reference, and 2 cycles for the actual memory access. The effective time for fetching a data into a data register using the LOAD instruction is actually lower than these 3 cycles due to the pipelined nature of operations of the Capability Unit and the Execution Unit and pipelining within each of these units, in turn (Sections 3. 4. 4 and 3. 5. 2).
Precise estimation of the effective execution time of the LOAD instruction is not possible unless we discuss an implementation of the proposed architecture in detail, and is thus beyond the scope of this thesis.

The STORE instruction has the form:

\[
\text{STORE } D_j, C_k
\]

where \( D_j \) is the data register whose contents are to be stored into an object of type generic, code or SCO, whose capability is in capability register \( C_k \), at a location specified in the corresponding offset register \( F_k \). The capability in \( C_k \) must be a valid one and must have at least the 'write' access privilege and the representation access flag enabled. The STORE instruction, on successful execution, stores the contents of \( D_j \) in the specified word, sets the initialization tag bit of the word to 1 and marks the word tag (Section 4.1.1) to indicate the word as a non-capability item. Notice that a STORE instruction does not check the word tag associated with the target location before storing the contents of the specified data register.

Since a generic object can hold both capability and non-capability (i.e., data) items, it is thus possible to overwrite one of the two words forming a capability (that is stored within the target generic object) using the STORE instruction. This does not pose a threat to the capability mechanism, since the word tags are changed: the original capability is not altered to a different capability -- it is destroyed, since the word tag associated with the target word marks it as a non-capability item. The solution to this problem is, of course, to specify the correct offset, so that capabilities are not overwritten. A better solution would be to store capabilities and non-capabilities in distinct objects, for example in a C-list object and a generic object, respectively. A STORE instruction generates the following exceptions:
• Exceptions associated with capability translation, as described in Chapter 4.

• An 'attempt to store uninitialized register' exception, when the initialization tag associated with Dj is reset. Again, circuitry associated with the memory buffer within the EU would be used to detect such an attempt.

• An 'illegal attempt to use indirect capability' exception if the capability specified in an indirect one.

The STORE instruction takes 3 cycles to execute under favorable conditions, for reasons similar to those for a LOAD instruction. Pipelining reduces the effective execution time of a STORE instructions to a much lower figure.

From the description given for the LOAD and STORE instruction, it should be obvious that both the Capability Unit and the Execution Unit actively participate in the implementation of these instructions.

5.1.2 Instructions for loading capability and offset registers

Although we have alluded briefly to the instructions for loading and storing capabilities in Chapters 3 and 4, we did not described them elaborately. In this section, we discuss details of these instructions.

The LOAD-RELATIVE instruction is used to load capability registers C0 through C15 with a capability from within a code object or a SCO. This instruction is of the form:

\textbf{LOAD-RELATIVE} Fj, Ck

where Fj is an offset register and Ck is a capability register. The effect of this instruction is to load Ck with a capability from within the currently executing code object or SCO (whose
capability is in PC), stored at an offset specified in Fj. The capability in PC must have the 'read' privilege. (It must certainly be valid and have the 'execute' privilege, too!) Exceptions generated by this instruction are as follows:

- All exceptions associated with capability translation.
- An 'attempt to load non-capability' exception if the location specified does not contain two capability words in the right sequence or if the initialization tags in one or both words forming the capability are reset. Recall from our discussions in Section 4.1.1, that a valid capability consists of two consecutive words with the word tags as zero, and sequence tags as 0 and 1, respectively. Circuitry associated with the capability assembly buffer within the capability unit would be used to detect such a violation.
- An 'illegal attempt to use indirect capability exception', if the capability in Ck is indirect.

The LOAD-RELATIVE instruction takes 5 cycles (1 for capability translation, 4 for memory fetches) to execute, and is implemented entirely within the Capability Unit. Like the LOAD (and STORE) instructions, pipelining results in a lower effective execution time for this instruction. A typical use of this instruction would be to load capability registers on a cold-start.

The LOAD-CAP instruction is of the form:

\[
\text{LOAD-CAP Cj, Ck}
\]
capability register Ck and its corresponding offset register, Fk specify an object (of type generic, C-list, code or SCO or refinements of these types) and a location within this object from which a capability is to be loaded into the capability register Cj. The offset register Fj corresponding to Cj is unaffected by this instruction. This instruction is similar to LOAD-
RELATIVE in all other respects.

The instruction STORE-CAP is used to store the contents of a capability register into an object (of type generic, C-list, code, SCO or refinements of these types). This instruction is of the form:

\[ \text{STORE-CAP } C_j, C_k \]

where \( C_k \) and its corresponding offset register \( F_k \) specifies the target object and the location within the target object where the capability in \( C_j \) is to be stored. The capability in \( C_k \) must be valid and have at least the 'write' access privilege and the representation access flag set. STORE-CAP is analogous to STORE and generates similar, but appropriate, exceptions. It takes 5 cycles to execute under favorable conditions (1 for capability translation, 4 for memory stores).

The instructions LOAD-OFFSET and STORE-OFFSET are used to load and store the contents of the offset registers, \( F_0 \) through \( F_{15} \) from/into objects of generic type. These instructions are similar to the LOAD and STORE instructions described in Section 5.1.1 with the following exception. Since offset registers are 20-bits wide and each memory word can contain 32-bits of information, a LOAD-OFFSET loads only the lower 20-bits in the source word into the target offset register. Likewise, STORE-OFFSET stores its contents into the lower 20-bits of a target word, padding the remaining 12 bits with zeros.

5.1.3 Instructions for moving data between registers

The MOVE instruction is used to copy the contents of one data register to another. It is of the form:

\[ \text{MOVE } D_j, D_k \]
The effect of the instruction is to copy the contents and initialization tags of Dj to Dk. The MOVECAP instruction is used to copy the contents of one capability register to another, while MOVE-OFFSET is used to copy the contents of one offset register to another. The syntax and semantics of these instructions are similar to those of MOVE. All of these instructions take 1 cycle to execute.

The instruction MOVEDO is used to transfer the contents of a data register to an offset register. During the move the upper 12-bits in the 32-bit data register content are not copied. Only the lower 20-bits in the data register are moved to the 20-bit offset register. The corresponding capability register is unaffected. A similar instruction MOVEOD is provided for transfers in the other direction. Both of these instructions take 2 cycles for execution.

The instruction MVI is used to move a 10-bit literal into a data register, and is of the form:

\[
\text{MVI Dj, literal.}
\]

A similar instruction, MVOI, exists for moving literals into a specified offset register. These instructions execute in 1 cycle each. (Instructions also exist for inserting and extracting bytes from data registers.)

5.2 Instructions for Arithmetical and Logical Operations

Our architecture will support the usual repertoire of arithmetic and logical instructions using a register-oriented format. Most of these instructions (excepting the MULTIPLY instruction) execute in one cycle. Logical and arithmetic shift instructions, based on a barrel shifter are also provided. The barrel shifter enables bit shift or rotate instructions to perform
variable shifts or rotates in one cycle. The underlying micro-architecture is assumed to allow the shifter or the arithmetic unit to be used twice in each cycle.

An example of an arithmetic instruction is

\[ \text{ADD } D_j, D_k \]

whose effect is to add the contents of \( D_j \) and \( D_k \) and transfer the result to \( D_j \). An exception is generated if the initialization flags of either \( D_j \) or \( D_k \) are reset before the operation. This instruction may also generate an exception if the subrange check mechanism, to be described later, is enabled.

Similar instructions also exist for operands in the offset registers. However, these instructions are restricted to logical shifts, addition and subtraction.

5.3 Control Flow Related Instructions

A variety of instructions are provided in our architecture for controlling program sequencing. These are as follows:

1) Conditional branching and related instructions:
   a) Instructions for comparing the contents of two data registers. These instructions are used to test for the following relationships: equality, inequality, less-than, less-than or equal, greater-than and greater-than or equal. Appropriate condition code bits in the PSW are set on execution of these instructions.
   b) A conditional branching instruction, which is of the form:

\[ \text{C-BRANCH condition, } F_k \]
where 'condition' is an eight-bit literal that specifies the condition for branching. (This literal simply serves as a mask that specifies bits in the PSW to be sensed by the C-BRANCH instruction.) If the specified condition(s) is (are) found to be valid, the effect of C-BRANCH is to jump to the location specified by the offset register, Fk, within the code object or SCO that contains this instruction. If the specified condition(s) is (are) not valid, the next instruction is executed. C-BRANCH is thus an instruction that may result in a 'local' branch to a location within the currently executing local or code object.

2) Instructions for unconditional branching: Instructions corresponding to an unconditional local jump and an indirect jump through a local location to another local address are provided in our architecture. A non-local unconditional jump instruction, L-JUMP is also provided. This instruction has the form: L-JUMP Cj where Cj is a capability register containing a valid capability to a code object or SCO, with at least the 'execute' access privilege. The effect of this instruction is to jump to the location specified by the offset register, Fj, and within the object specified by Cj.

3) Procedure call and return instructions: A CALL instruction is provided to enable a call to be made to a local subprogram, and a RETURN instruction is provided to effect return from such a locally-called subprogram. The CALL instruction has the form:

    CALL Fj

where Fj is an offset register that provides the offset of the locally-called subprogram. The effect of CALL is to save the status registers of the CU and EU and the return address (in PF), on the activation stack within the current context object and transfer control to the specified address. Based on the assumptions made about the size of the
status registers (three words in all), in Section 4.8, the execution time of CALL is nine cycles. The instruction, RETURN, effects return of control from a locally called subprogram by restoring the contents of the status and the PF registers. The execution time of RETURN is also nine cycles.

Non-local procedure calls (i.e. calls to subprograms within a different object) are all made using the protected procedure call mechanism described in Section 4.8. The instructions PCALL and PRET, described in Section 4.8 enable protected calls and returns to and from a different SCO. We disallow non-local calls to code objects, since their entry points are not clearly defined (unlike the SCOs), thus reducing the likelihood of error. This restriction is not a serious one at all and it improves software reliability by allowing interactions with non-local objects containing programs only through pre-specified interfaces. The only way of using a code object is to use the L-JUMP instruction from another code object or a SCO that are all compiled together, so that the compiler can verify the validity of the entry and exit points. This ensures that the only way of control flow across independently compiled units is through the rigidly-defined SCO interface.

4) Other control flow instructions: These include a HALT instruction to stop program sequencing, a software interrupt instruction (SWI) and a system call instruction SYSCALL. The effect of SWI is to save the EU and CU status registers, the registers PC and PF on the activation stack within the context object and wait until an interrupt occurs. The effect of SYSCALL is to make a call to the Operating System, specifying arguments systems arguments registers, S0 through S2.
5. 4 Instructions for Supporting Operating System Functions

Many of the instructions centered around the capability mechanism, and presented in Chapter 4 can be used to support many operating system (OS) functions in an efficient way. Additional instructions are provided in the architecture to support the following OS functions:

a) Synchronization of access to shared objects,
b) Interprocess communication,
c) Instructions related to special or sensitive operations involving the capability mechanism.

Similar architectures, such as SWARD [46] and the iAPX 432 [29], provide instructions for process control (creating processes, delaying a process, blocking a process and the like). The absence of process control instructions in our architecture is noteworthy. The instructions provided in our system for interprocess communication enable all process-related operations to be implemented very efficiently.

The 'system' type object can be used very effectively to implement special types of objects used by the OS. One use of the system object type would be to implement a process object -- the abstraction of a process. Process-related functions can then be very simply implemented using the instructions provided for synchronization and interprocess communication, as described in Section 5. 4. 4.
5. 4. 1 The LOCK and UNLOCK instructions

The LOCK and UNLOCK instructions are used for controlling access to shared objects. The typical code sequence of using these instructions is as follows:

LOCK (shared object)

---code to use shared object

UNLOCK (shared object)

The purpose of the LOCK instruction is to lock an object for exclusive use in the following manner, by using the lock bits associated with the object's CMT entry. If the object is already found to be locked, the lock bit pattern is set to indicate that another process is waiting to access the object. The process executing LOCK is then suspended by calling the OS with appropriate parameters. If the object was not found to be locked, it is locked for use by the process executing LOCK. The semantics of LOCK is thus similar to the P-operation on a binary semaphore initialized to one.

The UNLOCK instruction is used to relinquish exclusive access to a shared object. It marks the lock bits associated with the shared object as unlocked if no other process was waiting on a LOCK. If UNLOCK discovers, on examining the lock bits that some other process was waiting for exclusive access to the shared object, a system call is made, with appropriate parameters, to revive one of the waiting processes. UNLOCK is thus similar to a V-operation on a binary semaphore which is used for mutually-exclusive access to a critical section.
The LOCK and UNLOCK instructions have the form:

\[
\text{LOCK } C_j \\
\text{UNLOCK } C_j
\]

where \( C_j \) is a capability register containing valid capability to the shared object with at least the 'read' or 'write' and 'lock' privilege. Details of the LOCK and UNLOCK instructions are given in Appendix 1. Both LOCK and UNLOCK instructions take fourteen cycles each.

5. 4. 2 The SEND and RECEIVE instructions

In our system, each process sets up one or more port objects to serve as its mailbox for interprocess communication via messages. The SEND instruction is used to write a message to a port and the RECEIVE instruction is used to read a message from the port. Although, such instructions could be implemented using LOCK and UNLOCK instructions earlier, our decision was to incorporate the message queueing discipline as part of the send and receive operations to enable efficient interprocess communication. The Intel iAPX 432 [29] provides similar, but more complex (and correspondingly inefficient!) SEND and RECEIVE instructions.

In our system, we distinguish among two message sizes. Short messages are less than sixteen-bits, while long messages occupy seventeen or more bits, and can be arbitrarily long. Figure 5. 1 shows the general format of a message when it is stored on the queue associated with a port. Each element holding the message consists of six logically consecutive words, with the first five words containing the actual message. The sixth word contains the process priority of the SENDing process and an offset serving as a link to the next element in the
<table>
<thead>
<tr>
<th>Message Type and Size</th>
<th>16-bit Literal Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capability to Generic Object Containing Rest of Message</td>
<td></td>
</tr>
<tr>
<td>Capability to Sending Process Object</td>
<td></td>
</tr>
<tr>
<td>Process Priority</td>
<td>Link to Next Queue Element</td>
</tr>
</tbody>
</table>

NOTE The Slot for the Generic Object Capability Depicted Above Remains Unused for Short Messages up to 2 Bytes

Figure 5. 1 The Format of a Queued Message

queue. (Since an object is at most $2^{20}$ words long, this offset is twenty-bits wide.) Short messages are sent as the sixteen-bit literal in the first word, while for longer messages, the rest of the message is put into a generic object and a capability to this object (with at least the 'read' privilege is put into the next two words).

Each port object consists of a five-word header describing various attributes about that object, followed by an area for implementing the message queue, as shown in Figure 5. 2.
The (successful) execution of a SEND instruction would result in writing a message to this queue following a prespecified queuing discipline, and the return of some information indicating the successful queuing of the message to the execution of SEND. The process executing the SEND can specify, in its message type field, that it be revived when the enqueued message is read by a RECEIVE instruction. A synchronous or blocking send-receive operation pair (similar to the rendezvous of tasks in Ada [63]) can be easily implemented by sending a message, as described above, to a port and then suspending the execution using a system call instruction, specifying the desire to suspend. The receiving process will have to suspend itself on executing a receive if no message was sent to the port. Non-blocking or asynchronous send and receive operations can also be implemented by using the SEND and RECEIVE instructions, if no system call to suspend the execution is made after a SEND or RECEIVE. In this case, neither the sender nor the receiver are blocked (suspended) after a SEND or RECEIVE. Descriptions of the SEND and RECEIVE instructions will provide greater insight into the implementation of blocking and non-blocking communication primitives.

The result of the execution of a SEND or RECEIVE instruction is indicated by two condition code bits -- the negative flag and the positive flag. The following overview of the SEND and RECEIVE instructions indicates how these flags are affected.

The process executing the SEND can specify, in the message type field of its message, that it be revived when the enqueued message is read by a RECEIVE instruction. The execution of both the SEND and RECEIVE instructions affect the negative and positive condition code flags to indicate the results of executing these instructions. We now provide an overview of the SEND and RECEIVE instructions.
<table>
<thead>
<tr>
<th>Maximum Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Queue Size</td>
</tr>
<tr>
<td><strong>Queueing Policy</strong></td>
</tr>
<tr>
<td>Link to First Element in the Free List</td>
</tr>
<tr>
<td>Unused</td>
</tr>
<tr>
<td>Unused</td>
</tr>
</tbody>
</table>

Space for Elements of the Free List and the Message Queue

$W = \text{Flag Designating the Presence of 'Receiver' Process(es) Awaiting the Arrival of Message(es)}$

**Figure 5.2 The Port Object**
The SEND instruction is of the form:

\[ \text{SEND } C_j, C_k \]

where \( C_j \) specifies a capability to a generic object containing the rest of a long message, if the message is long, and the corresponding offset register \( F_j \) specifies the message type and size, followed by a sixteen bit literal message. \( C_k \) designates a capability register containing a valid capability to the port object to which the message is to be sent. The capability in \( C_k \) must have at least the 'write' privilege. (For short messages, the contents of \( C_j \) are immaterial and \( C_j \) remains unaffected.) The effect of executing this instruction is as follows:

- The validity of the capability to the port object is checked.
- If the queue in the port is full, the negative flag is set and all other condition flags are reset and the execution of SEND terminates.
- If the queue is not full, the message is enqueued according to the specified queuing policy (FIFO or process-priority based).
- The positive flag is set and the other condition flags are reset.
- If any process was waiting for a message to arrive to the port, make a system call to revive one such waiting process.

The RECEIVE instruction has the form:

\[ \text{RECEIVE } C_j, C_k \]

where \( C_k \) specifies a valid capability to the port object with at least the 'read' privilege. \( C_j \) will hold a capability to the refinement object containing the rest of a long message (if the message received is long), and the corresponding offset register \( F_j \) will hold the message type and size, followed by a sixteen bit literal message when RECEIVE completes. If the execu-
tion of RECEIVE intends to suspend itself, if the port has no messages to be read in its queue, $F_j$ is set to zero. This enables RECEIVE to set a flag within the port object that will be sensed by a later-executed SEND to revive a process waiting for the arrival of a message. The effect of executing the RECEIVE instruction is as follows:

- The validity of the capability in $C_k$ is checked.
- If the queue in the port is empty, the negative flag is set and the positive flag is reset. If $F_j$ is zero (indicating that the execution of RECEIVE will suspend itself), a flag within the first word of the port object is set to indicate that at least one process (viz. the executor) is waiting for the arrival of a message. The execution of RECEIVE is then terminated.
- If the queue is not empty, the message at the head of the queue is read (into $F_j$ and $C_j$ if the message is long, into $F_j$ only if the message is short).
- The positive flag is set and the negative flag is reset.
- If the message type read into $F_j$ specifies that the process that sent the message be awakened when the message is read, a system call is made to fulfill this request.

Neither the SEND or RECEIVE instructions are privileged. Before presenting the details of the SEND or RECEIVE instruction, some comments are necessary concerning the structure of the port object shown in Figure 5.2. The first two words in a port object contain the maximum and actual queue sizes, respectively. The third word contains three fields as follows:

(i) A one bit field that indicates the queuing priority: this can be defined at the time of creating the port object as either first-in, first-out (FIFO) or process-priority based.
(ii) A one bit field that indicates if any process was waiting for a message to arrive. We will refer to this field as the *waiting status* flag.

(iii) A twenty bit field in the lower twenty bit positions of the word that indicates the offset for the first element in an available list of queue elements (the free list) in the virtual space of the port object.

The next two words (words four and five) in a port object indicate the offsets for the first and last elements of the message queue (in the lower twenty bit positions in each word). The remaining space in the port object is devoted for the elements in the free-list and the message queue, which is implemented as a linked-list of elements.

Details of the SEND and RECEIVE instructions are given in Appendix I. The best-case timings (in cycles) for each step are also shown in these figures. The best-case situation assumes that capability references can be translated in one cycle, i.e., TLB entries exist for the references. When a FIFO queuing policy is used, the time required to execute a SEND instruction, that successfully queues the message being sent, is fifty-one cycles. This corresponds to the execution of steps one through ten, twelve and thirteen of the algorithm for SEND. For a process priority based queuing policy, assuming that the message is queued at the middle of an N-element queue (corresponding to an average-case situation), SEND takes \((56 + (N/2) * 7)\) cycles to execute. (This corresponds to the execution of steps one through nine; one execution each of steps 11(i),(ii),(iii),(iv),(vi),(vii); N/2 executions of steps 11(iii),(iv),(v); steps twelve and thirteen.) The worst-case time for executing a SEND, when a process-priority based queuing policy is used is \((56 + (N*7))\) cycles. Notice that in the estimations made for the execution time of SEND, we have assumed that a system call to revive a waiting process is not made at the end of SEND. This corresponds to a best-case
situation for the entire SEND instruction. (If a system call is made at the end of SEND, the
time to execute and service this call is effectively added to the execution time of SEND.)

The execution time of a RECEIVE instruction in the best case (when capability refer-
ences made by RECEIVE can be translated in one cycle and when there is no sending
process to be revived) is thirty-five cycles. (This corresponds to the time to execute steps
one through five, seven through twelve, fourteen through sixteen.) The above timing applies
to a long message. For a short message, step eleven is not executed and the best-case timing
is twenty-nine cycles.

Both the SEND and RECEIVE instructions are sufficiently general and allow both
blocking and non-blocking 'send' and 'receive' primitives to be implemented. (A blocking
send primitive suspends the sender until the sent message is read by a receiver; a blocking
receive delays the receiver until a message arrives.) A blocking send is implemented easily
by sending a message (using SEND) whose type field specifies that the sender be revived
when the message is read, followed by a system call instruction SYSCALL to suspend the
sender if SEND sets the positive flag:

```
SEND Cj, Ck
/* message type specifies revival of sender */
/* when message is read */
BRA positive SLEEP
SLEEP:
SYSCALL /* suspend sender */
```

A blocking receive is simply implemented by executing a RECEIVE with a zero in Fj
and executing a SYSCALL to suspend the receiver if the negative flag is set. Synchronous
communication, like the task rendezvous in Ada [63], with a blocking send and a blocking
receive can thus be easily and efficiently implemented.

5. 4. 3 Implementing process-related operations

As mentioned earlier, the Operating System can implement 'process' objects using the system object type. The SEND and RECEIVE instructions, in conjunction with the other instructions, can be used to implement the common process related operations efficiently, making unnecessary the need for special process-related instructions.

The process control operations involving the scheduling of a runnable process, blocking a process until a specified event occurs, and delaying a process by a specified time unit, involve the migration of the process from one queue to another and the eventual execution of a process. This migration can be easily accomplished by SEND-ing the process object from one port to another. The various queues and their corresponding queuing discipline can be implemented as ports. Scheduling a runnable process on the processor can be accomplished similarly: the processor can be disguised as a port, and the service routine for the port (i.e., the routine that executes the RECEIVE for the port) can set up the various registers in the EU and CPU and invoke hardware mechanisms for running the task so loaded. When some of the ports do not use a FIFO strategy, which may well be the rule rather than the exception, a routine is needed to compute process priorities appropriately, to enable the use of the built-in priority-based queuing policy for the ports.

The primitives corresponding to the act of creating and destroying a process can be implemented in software, using the CREATE and DESTROY instructions for creating and destroying a process object, and SEND/RECEIVE for activating the process. (Since process objects are likely to be within a page in size, CREATE can be used successfully, along with
privileged instruction, CONVERT (Section 4. 7. 4) to create process objects dynamically.)

5. 4. 4 Other OS-related instructions

Although, the concept of privileged instructions is not consistent with the more general concept of controlled sharing using capabilities, some privileged instructions are needed to exercise sensitive control over the capability mechanism itself. We have already discussed one such privileged instruction (CONVERT) in the last chapter. In this section, we introduce some other privileged instructions.

The argument-less privileged instruction, DISABLE-CAP and ENABLE-CAP are provided to disable and enable the capability mechanism to enable trusted OS routines to bypass the capability mechanism. The instructions DISABLE-INT and ENABLE-INT are provided for disabling and enabling the interrupt mechanism. A privileged instruction, LOCK-FRAME is also provided to lock a page frame and prevent it from being swapped. A complementary instruction UNLOCK-FRAME is provided to unlock a locked page frame.

Two privileged instructions, SET-CONTROL and READ-CONTROL are provided to write and read, respectively from the control registers. The SET-CONTROL instruction has the form:

\[
\text{SET-CONTROL Dj, Ck}
\]

where Dj specifies a number identifying a control register, and Ck or its corresponding offset register Fk, as appropriate, specifies the data to be written to the control register. The format of READ-CONTROL is similar. The SET-CONTROL instruction is obviously useless when the system is initially started ('powered up'). Hardware booting mechanisms are provided for loading the control registers.
5. 5 Support for Run-Time Error Detection

We mentioned in Chapters 1 and 2 the need to detect the following common run-time errors efficiently: access to an uninitialized variable, a subrange variable or an array subscript going out of range and parameter inconsistencies across independently compiled units. In this section, we will describe how these errors are detected in our system.

5. 5. 1 Detecting access to an uninitialized variable

In our architecture, access to uninitialized locations are detected using the initialization flag associated with every word. Recall from our earlier discussions, that the GCU (garbage collection unit) clears the word tag and the initialization flag associated with every word in a deallocated frame before a deallocated page frame can be re-allocated. On every write to a physical memory word, the initialization flag is set (to mark the word as initialized). The fetch circuitry automatically detects (among other things) any attempt to fetch a word with its initialization flag reset, and raises an exception on such an event. This mechanism thus detects every attempt to access an uninitialized memory location.

The only apparent performance overhead for the mechanism involves the effort (and time) needed to clear the initialization flags in a freshly deallocated frame. The GCU clears both the word tag and the initialization flag of each word at the same time. Thus, there is no extra time overhead for clearing the initialization flags. (We have to clear the word tag anyway to prevent the accidental inheritance of capabilities!)
5.5.2 Detecting out-of-range subrange variables

In our architecture, out-of-range values can be assigned to subrange variables (such as subrange types and array subscripts) under the following circumstances:

(a) By writing into a memory location corresponding to a subrange variable using a STORE or STORE-OFFSET instruction,

(b) As a result of moving data into a data register that is supposed to hold the value of a subrange variable. The MOVE, MOVEOD, and LOAD instruction can do this directly by moving the result of an arithmetic operation into a data register,

(c) As a result of moving data into an offset register under circumstances similar to (b) above.

The error corresponding to case (c) can be easily detected in our architecture by the capability mechanism described in the last chapter. Recall that offset registers can hold only positive values and that any attempt to generate or store a negative value in an offset register is automatically detected by the CU hardware (which raises a negative offset exception). The real concern in this case is to detect an out-of-range positive value in an offset register. Our choice here is to detect such a value when it is used, rather than detect it when it is formed. The capability mapping logic has the ability to do this.

Detecting out-of-range assignments to subrange variables in cases (a) and (b) are substantially more difficult and require the use of some additional hardware in the EU. The circuitry required to detect such an error would essentially be a comparator that monitors the bus to the destination register memory location (corresponding to a subrange) during the execution of a data moving instruction or an arithmetic instruction. Doing so would enable
subrange violations to be detected without any additional time overhead and concurrently with the execution of these instructions. Two questions arise immediately: how do we enable this check selectively, and how do we specify the comparands (subrange limits)?

The information necessary for enabling or disabling the subrange check mechanism and for locating appropriate comparands can be easily supplied as part of the opcode of every instruction. The comparands (one pair for every subrange type) can be located in some data register pairs. We will restrict the number of simultaneously testable subrange types to three, so that three comparand register pairs would be sufficient for storing the subrange limits. We will designate these register pairs as D16-D17, D18-D19, and D20-D21 to the hardware. A two bit code, appended to the opcode of every instruction can specify the type of subrange check needed, as follows:

- 00- subrange check with comparands in D16-D17
- 01- subrange check with comparands in D18-D19
- 11- subrange check with comparands in D20-D21

(The scheme suggested above dictates the structure of the underlying machine to some extent.) The instruction decode logic will have to identify the destination bus appropriate to the instruction. For arithmetic instructions an additional problem arises. Since the result of an arithmetic instruction is not completed until towards the end of a cycle, the comparator within the ALU cannot possibly be used to check the subrange violation without stretching the execution time of these instructions. (Recall that most arithmetic instructions take a single cycle to execute.) The remedy is obvious- we need an extra comparator within the EU to monitor subrange violations. This comparator may be well worth the investment, considering the advantages gained. A natural question that arises at this point is, "if the mechanism for
detecting subrange violations is so very easy and simple why wasn’t it incorporated in earlier architectures?” The answer to this is two-fold. First, the need to efficiently detect subrange violations was not appreciated by earlier architects. Their design was rooted in architectures that came into being when the notion of subrange variables were absent in programming languages. (To be fair, SWARD [46] does have an instruction RANGE-CHECK that compares an operand with two comparands and raises an exception if the operand’s value is beyond the range specified by the comparands. This is an instruction and has to be explicitly invoked every time a possibility of subrange violation exists, and thus has a corresponding performance penalty.) The second reason of not incorporating the mechanism described earlier in existing systems has to do with the efficiency of encoding. Clearly, the width of the opcode field and the complexity of the decoding process is significantly increased. The first issue is of little concern to us since we propose to have a fairly straightforward encoding scheme to enable faster decoding, and also, the space in a memory word is sufficient to encode no more than one instruction, with some space left over, but not sufficient for a second instruction. (From our earlier description of instructions, a two-register operand format is suitable for all but the load and store instructions.) Thus, adding the two-bit subrange check specification in each instruction simply uses some left-over storage—it does not reduce the instruction fetch rate to any degree. The second issue, the complexity of the decoding process, is not a significant one, since the decoding functions for the two-bit subrange check specification are fairly (but not totally) disjoint from the decoding functions for the rest of the opcode.

The role of the compiler in using the mechanism described above is an important one. The compiler will have to keep track of the comparand register allocations and then generate
subrange check specifications appropriately for the instructions. For instructions that cannot
generate subrange violations, the two-bit subrange check specification field is ignored, and to
play it safe, the compiler can generate ‘00’ for the specification for such instructions.

The subrange checking mechanism described above is useful for detecting violations of
the following types at run time with no penalty on execution time:

• out-of-range assignment to subrange types;

• one-of-range assignments to enumerated types (if they are implemented by mapping
enumerated types to a corresponding subrange type);

• formation of out-of-range array subrange subscripts. Up to three subscripts can be
  checked at a time.

The ability to detect array subscript violations without any additional performance
penalty is a very useful one, since typically, the array happens to be the most frequently-used
data structure and 25% of the operands are array elements [49].

5.5.3 Detecting intermodule parameter inconsistencies

The sixty-four-bit template register, TR (which is really the data register pair D30, D31)
is used by the CHECK-PAR instruction to check the consistency among actual and formal
scalar parameters across independently compiled units. The calling procedure uses the move
(or MVI) instructions to write information above the number of active parameters and the
attribute of each parameter. This information is written to the TR register in the following
format:
• Bits 63 through 60: number of parameters.

• Bits 59 through 0: attributes of up to 10 actual parameters, in a 6-bit field for each parameter.

The six-bit field describing each parameter has the following interpretation:

• Bits 5 and 4: parameter-passing convention (00=call-by-value, 01=call-by-reference, 10=call-by-copy, 11=call-by-value-result).

• Bits 3 through 0: four-bits describing the parameter type (integer, boolean, real, character, capability, etc.).

We will refer to the sixty-four-bit entity in TR as the actual parameter template. In the called unit, the first instruction executed is the CHECK-PAR instruction, which has the form:

\[ \text{CHECK-PAR } F_j \]

where the offset register, Fj, locates the first of two consecutive words within the executing SCO that corresponds to a template for the expected (actual) parameters. (The contents of the corresponding capability register, Cj is immaterial to this instruction.) The effect of CHECK-PAR is to generate an exception if the actual parameter template (in TR) does not match the formal parameter template (located by Cj). Some comments are due here regarding the parameter-checking mechanism. The mechanism described above uses CHECK-PAR to verify the consistency of up to 10 parameters which is sufficient in most of the cases. (Typically, up to 97% of procedure calls involve 5 parameter or less [61].) When the number of parameters exceed 10, templates for the additional parameters can be themselves passed as a single argument to the callee and the callee can use the COMPARE and C-BRANCH instruction to explicitly check the other parameters. Notice that the use of the CHECK-PAR
instruction in the called unit is optional, although its use is highly recommended. In fact, the compiler should generate code to check for parameter consistency, in the manner indicated above, as a routine matter. It is also worth noting that the use of CHECK-PAR eliminates a sequence of COMPARE and BRANCH by a single instruction. It is included in the instruction set merely as a matter of convenience.

5. 5. 4 Detecting some other run-time errors

When entire objects are passed as parameters, the access code in the capability passed as a parameter helps in detecting inconsistencies among actuals and formals in the operational mode. For example, if a formal parameter needs write access, the capability to the actual object must have write access. If the capability to the object does not have 'write' access, we have an inconsistency in the operational mode which will be detected during any attempt to write to the object within the called routine.

The capability mechanism also enables the detection of run-time errors involving dangling capabilities, as discussed in Section 4. 4. 3.

5. 6 Support for Program Debugging and Tracing

This section presents the functions and/or instructions provided in our architecture to support program debugging and program tracing.
5. 6. 1 Detecting read access to a word

An instruction, NULL, is used to clear the initialization flag associated with a word. This instruction has the form:

\[ \text{NULL } C_j \]

where \( C_j \) contains a capability to the object containing the word, with at least the 'destroy' privilege, and \( F_j \) contains an offset that locates the word within the object. The NULL instruction can be used to detect any attempt to read the location specified by NULL. The NULL instruction can also be used to void the contents of a word and prevent any usage of the voided word.

5. 6. 2 Detecting accesses to words and objects

It is useful to have mechanisms for detecting access to specified locations in various possible access modes such as read, write, and execute (instruction fetch) [57]. It is also useful to have mechanisms for detecting access to any part of a specified object, in some specified access mode. These mechanisms are very elegantly implemented within the CU in our system by using an associative memory very similar to the capability TLB. We will refer to this associative memory as the trap address cache, TAC, for reasons that will be obvious from the ensuing discussions.

An entry within the TAC has the following fields:

- An associatively searched field comprising of the least significant fifteen-bits in the UID of a capability.
• An offset-field (logical page number and line number, twenty bits in all).

• A field containing the most significant thirty-six bits in the uid of a capability.

• A specified access mode field (five bits) that uses a bit vector to encode the various access modes possible.

• A two-bit trap region field whose interpretation is as follows:

  00 - detect reference to any part of the object in the specified mode.

  01 - detect reference to specified page in specified access mode.

  10 - detect reference to specified location in specified access mode.

  11 - unused.

To detect an access to any portion of an object in any of the specified access modes, a corresponding entry is set up in the TAC using the following instruction:

\[ \text{SET-TRAP } C_j, D_k \]

where \( C_j \) and the corresponding offset register specifies the object and the location within the object and the data register specifies the following:

a) A location within the TAC where the entry is to be set up (the TAC is addressed as a RAM during loading),

b) The trap access modes (five bits) and the trap region size (if the trap region size code is 00, the contents of \( F_j \) are immaterial).

On any access to object, if the TAC is not empty, it is queried with the least significant fifteen-bits in the UID of the capability being accessed. On a successful match, the matching TAC entry is retrieved and a trace exception is generated if all of the following conditions
are valid:

(i) The most significant 36 bits in the UID of the capability match the corresponding bits in the retrieved entry,

(ii) If the offset falls in within the specified trap region (page or lie, depending on the region size specification in the TAC entry) or if the region size specification is ‘00’.

(iii) If the access mode requested by the capability reference is one of the access modes to be trapped.

It is apparent from the previous descriptions relating to the TAC that its structure and operation is very similar to that of the TLB. The TAC, like the TLB is also implemented within the CU. It would therefore make the implementation simpler when the comparators used by the TLB are also shared by the TAC. When tracing via the TAC is enabled, the TAC gets a preference over the TLB in using these circuitry (since an exception is to be raised before the access is allowed). The number of entries in the TAC decide how many object-address pairs can be traced.

Two argumentless instructions, ENABLE-TAC and DISABLE-TAC are also provided to respectively enable and disable access tracing using the TAC. One major dissimilarity between the TAC and the TLB needs special mention. The TAC is explicitly loaded (and unloaded by overwriting!) with the SET-TRAP instruction. In contrast, the TLB entries are filled and removed dynamically by the address mapping logic to capture the locality of reference.

Using a cache for trapping the access address and the access mode in a specified mode provides us with a desirable ability that is absent on existing architectures [31] -- the ability
to trap on a 'write' or 'execute' access. It also enables code to be traced without the need for inserting special 'marking' instructions. (In traditional systems, the marking instruction may be simply a call instruction or a software trap instruction; SWARD [46] uses a conditionally-enabled marking instruction for the same purpose).

5.6.3 Tracing events

As part of the debugging/tracing process, it is useful to include mechanisms to detect the occurrence of the following events [31], [57]:

a) Execution of conditional or unconditional branch instructions that result in a branch.
b) Execution of a procedure call instruction.
c) Flow of control to a specific point in the program.
d) Completion of the execution of the current instruction.

In our architecture, appropriate mechanisms are included to trace events (a) and (b). Event (c) can be easily traced using the TAC, described in Section 5.6.2. However, the number of entries in the TAC will limit the number of events that can be traced in this category. In this section, we will describe an alternate mechanism for tracing events in the category (c).

Notice that events in categories (a) and (b) will result in a discontinuity in the value of the 'program counter' offset register, PFC for local branches or for local calls (using CALL) or a change in PC (for non-local jumps or protected calls). In terms of the hardware activities, both events (a) and (b) cause a value to be loaded into PF (instead of incrementing it) or into PC. The hardware necessary for detecting events (a) and (b) is thus fairly simple. A
programmable mechanism is, however, necessary to enable and disable the circuitry that generates exceptions on events (a) and (b). We will use three bits in the CU status register to enable the mechanism that raise exceptions on events (a), (b), and (c). The SET-CONTROL instruction, which is privileged, cannot be used for the purpose of setting or resetting these bits in the CU-STATUS registers. It is preferable that the ability to enable or disable exceptions corresponding to events in the categories (a), (b), (c), and (d) from any operational mode (user or privileged). For this reason, the instructions SET-EVENT and RESET-EVENT are provided. These instructions have the following form:

\[
\text{SET-EVENT literal}
\]

and

\[
\text{RESET-EVENT literal}
\]

where the literal field, in its four lower bits, specify the event flags to be turned on or off in the CU status register.

To detect events in category (c), we will mark the desired portions in the code within a code object or SCO using the instruction MARK, which has the form:

\[
\text{MARK literal}
\]

where the literal identifies the position marked. When exceptions corresponding to event (c) are enabled, execution of the MARK instruction generates an exception, after putting the identifier for the position into the system argument register. When exceptions corresponding to event (c) are disabled, the effect of the MARK instruction is to do nothing. Notice that, in order to trace events of category (c) in this way, we have to insert MARK instructions at appropriate points, and compile the corresponding code object or SCO. The preferred approach for tracing events in category (c) would be to use the TAC, as described in Section
5.6.2. When enough TAC entries are not available at any time, the approach using the MARK instruction can be used.

Exceptions triggered on event (d) are useful in implementing the single-stepping function. Many modern architectures, including SWARD incorporate similar mechanisms for event tracing [40], [46], and the mechanism incorporated in our architecture for event tracing in the manner described in this section is not novel in any sense.

5.7 Exception Handling

In our system, we group exceptions (also called faults or interrupts) into the following three categories:

(i) System-handled exceptions, which require the intervention of the operating system for servicing the exception. Examples of exceptions in this category are: TLB fault, page fault, out-of-range offset fault, invalid capability fault and so on.

(ii) Exceptions generated due to errors in programs (such as overflow, underflow, divide-by-zero, subrange violation, reference to uninitialized word).

(iii) Exceptions defined by the programmer. Many languages such as CLU [41] and Ada [63] permit the programmer to define exception conditions, test for these conditions explicitly and raise an exception when the conditions are valid.

Exceptions falling under groups (ii) and (iii) can be handled by programmer-supplied routines, and many languages like PL/I, CLU, Modula, and Ada allows the programmer to define these handlers. In this section, we will describe how exceptions falling all of these
groups are handled in our system.

Two mechanisms are provided in our architecture for raising exceptions:

a) An implicit mechanism that uses dedicated hardware to raise exceptions in groups (i) and (ii). When the hardware or firmware detects the conditions appropriate to these exceptions, it interrupts normal instruction execution (provided interrupts of that level are enabled), moves appropriate arguments into the system argument register and transfers control to a pre-specified location. This location corresponds to the address of the first-level interrupt handler routine in the operating system. The hardware necessary for doing this is fairly standard and deserves no special mention.

b) An explicit technique for raising exceptions falling under the category (iii) that involves the execution of the SYSCALL instruction with appropriate arguments.

Once an exception falling under group (ii) or (iii) is generated, it is necessary to locate an appropriate handler routine for the exception. Before we discuss the support provided in our architecture for locating the handler for an exception in category (ii) or (iii), it is necessary to review the approaches taken for locating a handler in some contemporary programming languages.

In CLU, exception handlers can be associated with a procedure [41], whereas in Ada handlers can be associated with a block or a procedure or a package [63]. In CLU, exceptions raised within a procedure can be handled only by the caller. In contrast, the search for an appropriate handler in Ada is not restricted to the immediate caller, it proceeds down the dynamic environment. We will now describe the support incorporated in our architecture for accommodating these diverse policies for locating the correct handler for an exception.
We will associate a handler-list with each block or procedure or package, as appropriate. This handler list is actually a part of the code object or SCO that contains the block or procedure or a part of the SCO that corresponds to the package. The handler list contains information for locating the handlers declared within the block or procedure or package. The compiler is responsible for generating the handler lists from the exception handlers declared within the block or procedure or package.

A handler list contains the following words:

- A word containing a numerical value that gives the number of handlers specified in this list;
- Two words containing a capability to the code object or SCO containing the code for these handlers (code for the handlers and the corresponding handler list is located within the same code object or SCO that contains the body of the block or procedure or package with which the handler is associated);
- A single word entry for each handler, comprised of two fields: a twelve-bit code identifying the exception served by the handler and a twenty-bit offset locating the body (code) of the handler within the code object or SCO specified by the capability in the end and 3rd word of the handler list.

We now describe how these handler lists are used to locate the appropriate handler for an exception.

Whenever a block is entered or a local procedure call is made (using CALL) or when an external procedure in a package is called (using PCALL), a pointer to the associated handler list is pushed onto a stack, hereafter called the handler stack, implemented by a
generic object. On an exit from a block or a local procedure or an external procedure, the handler list pointer is popped off this stack. This pointer is a three-word entity, comprising of a capability to the code object or SCO containing the handler list, and an offset to locate the list within the code object to SCO. The generic object implementing the stack of handler list pointers will be hereafter referred to as the HLSO (handler list stack object). For execution efficiency, two control registers, CH and FH are provided to hold the capability to the HLSO and the offset for the top of the handler stack. Compiler generated codes are responsible for pushing and popping handler list pointers from the handler stack. From the description given above, it should be obvious that dynamic environment for locating the handlers is correctly recorded in the handler stack within the HLSO. The task of locating an appropriate handler can simply be accomplished by searching the handler lists located by pointers obtained by running down the handler stack. To mimic Ada semantics, this search starts within the handler list specified by the topmost pointer in the stack. To model the semantics in CLU, only the handler list specified by the second pointer from the top in the handler stack is searched. In any case, the searching algorithm is fairly straightforward and can be easily microprogrammed. A mode bit in the CU-status register will control how the search proceeds (CLU vs. Ada semantics).

A decision has to be made about where to resume when an exception is serviced. (Again, programming languages featuring exception handling facilities specify the point of resumption, and this varies from one language to another.) The actions corresponding to this can be implemented entirely in software.
5. 8 Some Final Comments

In the last two chapters, we discussed the novel mechanisms and instructions in the proposed architecture. We did not discuss many other instructions or features that would be useful in implementing a real system centered on this architecture. Two notable things that we did not discuss were the problem of input and output and the problem of filing objects on long-term storage. Both are non-trivial issues. We, however feel that the architecture, as discussed has the necessary mechanisms for implementing these functions. Input/output, for example, can be simply accomplished by receiving from or sending to ports mapped onto I/O devices. Long-term storage functions can be implemented entirely in software. Another important consideration in processor design today is the ability to set up multiprocessor configurations. The basic mechanism in the architecture does allow the possibility of 'software-transparent multiprocessing' [46] whereby processors can be incrementally added to the system to improve performance, without the need to change the operating system. We expect to explore these possibilities in the future.

We are now finished with the description of architectural features in our system that are necessary for the realization of the goals listed in the introduction of this thesis.

In the next chapter, we will try to make an assessment of the proposed design, identify the contributions made by this thesis, and describe the scope and extent of future, related work.
6. ASSESSMENT AND CONCLUSIONS

The functional characteristics of the proposed architecture were described in reasonable amount of detail in the last two chapters. However, two fundamental questions remain unanswered: how does our architecture compare with similar ones? In what sense is our architecture unique and is it better (or worse) than similar architectures? We will try to answer these questions in the first two sections of this chapter and thus provide an assessment of the proposed design. In section 6.3, we will explicitly list the contributions made by this thesis. Section 6.4 represents the conclusions that can be drawn from the thesis, while the last section describes further work that needs to be done to develop the proposed design further, and to design and evaluate an integrated system based on this architecture.

6.1 An Assessment of the Functional Characteristics

In this section, we will describe the unique characteristics of our design by comparing its functional characteristics with those of similar architectures.

6.1.1 Unique aspects of the architecture

As indicated in Chapter 1, our fundamental goal was to design an architecture that would incorporate the following features toward the realization of software quality enhancement:
• Efficient support for detecting the common run-time errors,

• Efficient support for programming language constructs that are known to enhance software reliability by imposing a programming style,

• Support efficient mechanisms for making the system secure against programming errors or errors of other types,

• Efficient support for program debugging and tracing,

• Efficient support for program-specified exception handling.

To date, only two architectures, beside ours, come close to realizing these goals. These are the iAPX 432 [29] and the SWARD system [46], which were discussed, as occasion demanded, in the earlier chapters. We will indicate the unique features of our architecture by comparing the functional characteristics of our architecture with those for the iAPX 432 and SWARD.

Figure 6. 1 shows how our architecture fares in comparison with the iAPX 432 and SWARD in terms of run-time error detection capabilities. SWARD is our closest competitor in this respect; SWARD, however, checks for subrange violations explicitly at run-time using the RANGE-CHECK instruction.

The features incorporated in the three architectures under comparison for supporting techniques or programming language constructs for enhancing software reliability are shown in Figure 6. 2. All three architectures support the features listed. However, as we will see later, there are considerable differences in the efficiency (execution speed) with which these mechanisms are supported on these three machines. One particular weakness of SWARD in implementing abstract data types needs to be mentioned. In the iAPX 432 and SWARD,
<table>
<thead>
<tr>
<th>Error Type</th>
<th>iAPX 432</th>
<th>SWARD</th>
<th>Proposed Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference to uninitialized location</td>
<td>Not detected</td>
<td>Detected</td>
<td>Detected</td>
</tr>
<tr>
<td>Subrange violation</td>
<td>Not detected</td>
<td>Requires explicit checks using instruction</td>
<td>Detected</td>
</tr>
<tr>
<td>Dangling references</td>
<td>Detected</td>
<td>Detected</td>
<td>Detected</td>
</tr>
<tr>
<td>Intermodule parameter inconsistencies</td>
<td>Not detected</td>
<td>Detected on every call using explicit instruction</td>
<td>Detected optionally using explicit instruction</td>
</tr>
</tbody>
</table>

Figure 6.1 Common Run Time Error Detection Abilities

<table>
<thead>
<tr>
<th>Feature</th>
<th>iAPX 432</th>
<th>SWARD</th>
<th>Proposed Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data abstraction</td>
<td>Supported</td>
<td>Supported but inefficient</td>
<td>Supported</td>
</tr>
<tr>
<td>Small protection domains</td>
<td>Supported but inefficient</td>
<td>Supported but inefficient</td>
<td>Supported</td>
</tr>
<tr>
<td>Programming using modules</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Enforcement of minimum privileges</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
</tbody>
</table>

Figure 6.2 Features for Encouraging Good Programming Style

instantiation of an abstract type can share the object that contains the procedures implementing the abstraction. In contrast, the implementation in SWARD insists on integrating a copy of the programs implementing the abstraction with the representation object for every representation making the mechanism very unwieldy (dynamic instantiations, for example,
would require a long time to be spent in copying the abstraction program!). This is due to
the absence of any mechanism in SWARD for implementing capability amplification and
deamplification.

Capability-based addressing is the basic mechanism employed in the iAPX 432,
SWARD and our architecture for implementing a system immune to deliberate or intentional
error. In this respect, we need to compare our architecture with some well-known capability
architectures, as well. These are the C.mmp/Hydra system [67], CAP [66], Cm*/Star OS
[20], the IBM S/38 [28] and the Plessey PP 250 [15]. We will briefly compare the functional
characteristics for ensuring system security against programming and other errors in these
systems with our architecture.

All of these architectures have the mechanism required to implement the principle of
least privilege and small protection domain. They differ in the way some of the following
peripheral problems associated with protection are addressed.

1) The revocation problem: After granting a capability to an object, it may be necessary
to withdraw it later. This is the revocation problem [47]. All the capability architec-
tures under comparison have the basic mechanism for solving this problem. CAP,
SWARD, and our architecture provide a direct and simple solution to this problem by
providing indirect capabilities (Section 4.11).

2) The confinement problem [67]: once a capability is granted to a user, it may be neces-
sary to prevent the propagation of this capability and information accessible using this
capability from the user to other users. This is referred to as the confinement problem.
Only the C.mmp/Hydra system and the Cm* provides a partial solution to this prob-
lem. We have not attempted to solve this problem in our architecture since our primary
concern is not information security.

3) This illegal reference problem: when object names in a capability are re-used, it is possible for a user, holding a capability to a deleted object, to use it to address a newly created object with the same name and thus compromise the security of the system. This is the illegal reference problem, and is a strong possibility in systems like the iAPX 432, CAP, PP 250, and Cm*Star OS systems, which use indices of re-usable mapping table entries as names of objects in capabilities. The solution in these systems is to scavenge capabilities to the deleted object, as discussed in Chapter 2. C.mmp/Hydra, the IBM S/38, SWARD, and our architecture are all free of the illegal reference problem, since object names are derived from monotonically increasing clocks or counters.

Thus, our architecture compares very favorably with the functional characteristics of C.mmp/Hydra [67], which is the only architecture to date that has solved most of the problems associated with systems security in a capability-addressed environment.

Figure 6.3 shows how our architecture compares with respect to debugging and tracing abilities with the iAPX 432 and SWARD. Our architecture has some definite advantages over the others in this respect. Unlike the other architectures, we can:

- Specify trace points in programs without the need for inserting special marker instructions at these points and the need to recompile the program after the insertions.
- Detect access to specified locations in any specified access mode. (Even with markers inserted, the iAPX 432 and SWARD cannot detect 'write' access to locations.)
<table>
<thead>
<tr>
<th>Traps</th>
<th>iAPX 432</th>
<th>SWARD</th>
<th>Our System</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Requires explicit marks at points to be traced</td>
<td>Requires explicit marks at points to be traced</td>
<td>Explicit marking not needed</td>
</tr>
<tr>
<td>Read access to a specific location</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Write access to a specific location</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Call</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Successful conditional branch</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Completion of an instruction (used for single stepping)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 6.3 Tracing and Debugging Capabilities

The 80386 microprocessor from Intel provides debugging support that is somewhat similar to those provided in our architecture. It allows the tracing of access to any one of four programmer-specified locations in the specified access mode. However, it lacks the ability to detect access to any part of a specified segment. Nor does it have the ability to trace successful branches without the use of markers.

Like our architecture, both the iAPX 432 and SWARD provide architectural support for accommodating user-defined exception handlers. The mechanism used for locating the appropriate exception handler in all of these systems are fairly complex and all provide the same degree of support. Thus, in terms of their ability to support exception handling, all three architectures are at par.
6.2 Performance Comparison

In this section, we compare the performance of our architecture with similar architectures. Before doing so, we first clarify what we mean by 'performance', and then provide a basis for fair comparison of performance.

The term 'performance' can be interpreted in many ways in the context of evaluating the effectiveness of an architecture. One common interpretation of performance relies on low-level metrics such as instruction execution times, instruction throughput, memory traffic, and so on. Myers refers to the performance interpreted on the basis of low-level metrics as 'performance-in-the-small' [46]. Another interpretation of performance uses high-level metrics, such as the execution time of benchmark programs and cost-effectiveness metrics. Such measures provide a more global view of the implications of the architecture. Given our design goals, viz. the design of an architecture for enhancing software reliability, an appropriate high-level measure of performance for our system would be the factor by which software reliability is increased when our architecture is used. This, of course, would entail the construction of software systems for our architecture and at least a simulator for our machine -- two problems well beyond the initial scope of this thesis. Another major problem in estimating performance in this way is to quantify software reliability -- a difficult, if not impossible task in itself.

Estimating the performance of our system in terms of benchmark execution times is neither meaningful, nor feasible in the scope of this thesis. High-level benchmarks that are available are designed for traditional architectures and are not written in terms of the language constructs or mechanisms that would normally be part of a well-designed and
reliable piece of software. Thus, available high-level benchmarks are of very limited value in the context of evaluating the performance of an architecture such as ours.

We will therefore have to be content, at least initially, with evaluating the low-level performance of our system. The low-level performance estimates would be useful to us for several reasons:

1) A fairly accurate estimate of the execution time of an instruction on our system can be made from the algorithmic description of its corresponding microprogram,

2) Execution times of instructions on some of the similar architectures are available [8], [10], [20], [46], [67], and thus the low-level performance of these systems can be easily compared with ours,

3) Since many of the instructions incorporated in our architecture have a fairly high-level of operational semantics, their execution times would be fairly representative of the impact these instructions would have on the high-level performance.

In the next section, we provide a common basis for comparing the low-level performance of our architecture with similar ones.

6. 2. 1 A basis for comparing low-level performance

Given the fact that we are comparing the instruction set level performance of our architecture with similar ones, a fundamental question still remains. What is a meaningful way of comparing the execution times of instructions in two architectures? This is a valid question, since the execution time of an instruction simply does not depend on the architecture alone. It also depends on how the architecture is organized and implemented, and thus on things like
the technology used for the implementation, clock speed, bus bandwidth and other factors.

We will provide a fair basis for comparing the instruction set level performance of our architecture and similar ones by doing either or both of the following, wherever possible.

a) Instruction execution times would be expressed in terms of clock cycles rather than in absolute figures.

b) Instruction execution times would be scaled by the time required to execute a typical, simple instruction (such as the instruction for adding two numbers). We will, after [10], refer to the time to execute the simple instruction as ‘tick time’, and the execution time of any instruction divided by the tick time, as the ‘scaled execution time’.

Approach (a) has been used in many recent publications ([8], for example), while approach (b) has been used in [10], for comparing low-level performances.

6.2.2 A comparison of instruction execution times

The capability mechanism is an intrinsic part of our architecture and, in a way, is responsible for making our architecture grossly different from other ones. The capability mechanism is also used to implement many of the design goals listed in the introduction of this thesis. We will therefore compare the performance of some key instructions related to the capability mechanism with similar instructions related to the capability mechanism with similar instructions in other architectures.

Figure 6.4 shows how the instruction PCALL, used for effecting a protected procedure call in our architecture, compares with similar instruction or kernel primitives in some other systems. All of the times given in Figure 6.4 corresponds to the best-case situation. Our
<table>
<thead>
<tr>
<th>Architecture / System</th>
<th>Time to execute protected call</th>
<th>Time needed to create/clear context object</th>
<th>Tick time</th>
<th>Scaled time for protected proc. call</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWARD$^1$</td>
<td>0.04 ms</td>
<td>Preallocated</td>
<td>0.0005 ms</td>
<td>20</td>
</tr>
<tr>
<td>C.mmp/ Hydra$^2$</td>
<td>70 ms</td>
<td></td>
<td>0.14 ms</td>
<td>500</td>
</tr>
<tr>
<td>Cm$^3$ / Star OS$^3$</td>
<td>0.22 ms</td>
<td>Preallocated</td>
<td>0.0007 ms</td>
<td>31</td>
</tr>
<tr>
<td>iAPX 432$^4$</td>
<td>(982 cycles)</td>
<td>(334 cycles)</td>
<td>0.0058 ms</td>
<td>33.7</td>
</tr>
<tr>
<td></td>
<td>0.196 ms</td>
<td>0.066 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAP$^5$</td>
<td>0.224 ms</td>
<td>unavailable</td>
<td>0.003 ms</td>
<td>80</td>
</tr>
<tr>
<td>Our system</td>
<td>84 cycles</td>
<td>50 cycles</td>
<td>5 cycles</td>
<td>16.9</td>
</tr>
</tbody>
</table>

Figure 6. 4 Protected Procedure Call Times

...system emerges as a clear winner, both in terms of the number of cycles used and in terms of scaled execution time.

Figure 6. 5 shows the time needed to execute a protected procedure call on the iAPX 432 and our system, passing four integer parameters. It also shows the time needed to make a conventional procedure call, passing the same number of arguments on two traditionally...
rooted architectures. Excluding the times for our system, the remainder of the information
given in Figure 6. 5 comes from [8]. The performance of our system is very close to that of
the conventional processors, which execute unprotected calls only. One reason for this is the
use of registers for the parameter transfers during a protected procedure call in our system.

Amplification and de-amplification of a capability to a representation object are impor-
tant operations in any capability-based system. Figure 6. 6 shows how our system compares
with the Cm* /StarOS system in this respect. No information was available regarding the
execution time of similar instructions on the iAPX 432. For the C.mmp/Hydra system, the
times for amplifying and de-amplifying a capability were not available separately.
Amplification is an intrinsic part of the protected procedure call mechanism in Hydra [67].
SWARD does not have any instruction for amplification and de-amplification for reasons
described in Section 2. 4.

Our system, like many similar architectures, provides the ability of interprocess com-
munication via messages. Figure 6. 7 depicts the times needed to send and receive messages
in our system and similar ones. The 'send' and 'receive' times shown in this figure
correspond to the best case situation in all of the systems, when the sender or receiver

<table>
<thead>
<tr>
<th>System</th>
<th>Scaled execution time for &quot;AMPLIFY&quot; (min./max.)</th>
<th>Scaled execution time for &quot;DE-AMPLIFY&quot; (min./max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our system</td>
<td>10/21</td>
<td>8/19</td>
</tr>
<tr>
<td>Cm* /Star OS</td>
<td>18/NA</td>
<td>18/NA</td>
</tr>
</tbody>
</table>

Figure 6. 6 AMPLIFY/DE-AMPLIFY Times
processes are not suspended. In terms of scaled operation times, the iAPX 432 is our closest competitor, so far as the 'send' time is concerned. The 'receive' time (scaled) on our system is about half that for the iAPX 432.

The ability to create objects dynamically is an important requirement in all capability addressed system. Figure 6. 8 depicts how our system compares to two other systems in this respect. (Other systems were not considered in this comparison, since object creation times on these are misleading or not available.) Again, our system does exceptionally well in this context, both in terms of absolute number of cycles and scaled time. For comparison, the time needed to clean an already pre-allocated context object in the iAPX 432 architecture is shown in this figure.

There are several other aspects of low-level performance that cannot be compared in a meaningful way without the lack of a detailed simulator (and system software) for our system and due to the lack of published results for other systems. Nevertheless, we will mention some of these aspects of low-level performance here.

<table>
<thead>
<tr>
<th>System</th>
<th>Send time (actual/scaled)</th>
<th>Receive time (actual/scaled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmmp/Hydra</td>
<td>3.5 ms/25</td>
<td>2.9 ms/21</td>
</tr>
<tr>
<td>Cm*/Star OS</td>
<td>0.15 ms/21</td>
<td>0.18 ms/26</td>
</tr>
<tr>
<td>iAPX 432</td>
<td>310 cycles/10.3</td>
<td>320 cycles/10.6</td>
</tr>
<tr>
<td>SWARD</td>
<td>21.8 ms/43.6</td>
<td>NA</td>
</tr>
<tr>
<td>Our System</td>
<td>51 cycles/10.2</td>
<td>29 cycles/5.8</td>
</tr>
</tbody>
</table>

Figure 6. 7  Send/Receive Times
<table>
<thead>
<tr>
<th>System</th>
<th>Total time</th>
<th>Scaled time</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmmp/Hydra</td>
<td>30 ms</td>
<td>214</td>
<td>Average time</td>
</tr>
<tr>
<td>Cm*/Star OS</td>
<td>5.6 ms</td>
<td>800</td>
<td>Minimum time; 480 instructions for clearing, estimated 30 for allocation</td>
</tr>
<tr>
<td>Our system</td>
<td>50 cycles</td>
<td>7.8</td>
<td>Minimum time; Allocation and clearing time included</td>
</tr>
<tr>
<td>iAPX 432</td>
<td>384 cycles</td>
<td>11.4</td>
<td>Preallocated active object time to clear and initialize 5 capability slots</td>
</tr>
</tbody>
</table>

Figure 6.8 Time to Create Small Objects

The capability generation scheme in our architecture ensures fast capability translation. A capability translation without the assistance of the TLB takes twelve cycles in our system, whereas, a TLB based translation takes just one cycle. If we assume a very reasonable TLB hit ratio of 80%, the effective address translation time is just 3.4 cycles (0.8 * 13 + 0.2 * 1 cycles). In the iAPX 432, translating a capability reference, even with the lower level mapping table entries in the cache, takes 89 cycles [25].

In Section 4.6.1, we had mentioned the possibility of storing the starting physical address of a small object (less than a page long), instead of the address of its PMT entry in the CMT as an obvious way of improving capability translation time for small objects. If this is done, the capability translation time for small objects reduces to just 7 cycles. Since objects in a capability-based system are typically small, this approach would result in considerable improvement in performance. Any real implementation of our architecture would
definitely embody this optimization.

As mentioned in Chapter 2, a major source of the overhead in the performance of many capability-based systems comes from the swapping of an object from primary memory to secondary storage, since this entails a conversion between two forms of capabilities. This overhead is not present in our system, since capability conversions are not needed during swaps of active objects.

Finally, some comments are due on the bread-and-butter instructions in our architecture—the instructions that are responsible for data manipulation. Most of these take a single cycle to execute. Two notable exceptions are the load and store instructions, which take two cycles each. SWARD and the iAPX 432 are particularly handicapped in terms of executing the data manipulation instructions, which employ a storage-to-storage format.

6.2.3 Explanation of the results of performance comparison

The comparison of some aspects of low-level performance in the last section clearly indicates the superiority of our system will respect to similar machines. In this section, we will try to explain why our system does so well.

Let us first consider the case of protected procedure calls. In the other systems, a major overhead of the call lies in the creation of the context object (as done in Hydra), or the 'cleaning' of an already used pre-allocated object (as done by the iAPX 432). In our system, we get around this problem by using a novel technique for creating the context object at a very high speed. This technique essentially involves the pre-allocation of a virtual storage space for the context object and use of a separate hardware unit (the GCU) for cleaning deallocated page frames concurrently with program execution on the EU/CU combination.
Another factor contributing to the superior performance of the protected procedure call instruction in our system lies in the use of registers for holding a pointer (a capability-offset pair) to the context stack object. Substantial efficiency also results from the use of registers for inter-context parameter transfers. Even if we neglect the time needed to create the context object in Hydra, a substantial amount of (scaled) time is still needed for effecting a protected procedure call in Hydra. This is because of the generality (perhaps unnecessary) that is built into the call mechanism in Hydra. In Hydra, as part of the call, all parameter capabilities are amplified, regardless of whether they are going to be used in the called context or not.

The performances of the AMPLIFY and DEAMPLIFY are superior to that for the Cm*/StarOS system (and are likely to be superior to similar instructions in other architectures) for several reasons. First, the amplification mechanism is very simple but effective, and controls access to the underlying implementation by using a bit flag within the representation capability itself. Second, the association between a representation object and its corresponding TDO is made by storing the TDO's capability within the representation object itself. When the type correspondence is to be checked by the AMPLIFY or DEAMPLIFY instruction, the capability of the TDO is available within the CU, as part of the 'program counter', in register PC. This obviates any need to fetch the TDO capability, and thus reduces the execution time for AMPLIFY and DEAMPLIFY considerably. In systems like the Cm*/StarOS, a considerable amount of time is spent in establishing the type correspondence.

The implementation of port objects, the 'send' and 'receive' instructions in our architecture are straightforward and similar to the approach taken in SWARD, whose absolute 'send'
time in cycles is close to ours (in comparison with others). Although, the scaled time for the iAPX 432 is close to ours in this respect, in terms of absolute time in cycles, the ‘send’ time the iAPX 432 is very inferior to ours. When processes are suspended on a ‘send’ or ‘receive’ on a port in the iAPX 432, the performance is likely to be inferior to ours. (We cannot corroborate this claim due to the lack of relevant published results.) This is due to the overgenerality built into the ‘send’ and ‘receive’ instructions in the iAPX 432 and due to the use of carrier objects [29] which serve as proxies for a swapped-out process object on a port queue. (Recall that swapping is a very costly process in the iAPX 432; carrier objects are used to easily locate swapped out process objects on secondary storage.)

In general, most of the instructions related to the capability mechanism in our system are not overly general or complex. It is this lack of overgenerality, coupled with the use of novel techniques that make instructions in our system superior to their counterparts in the other similar architectures.

### 6.3 Contributions Made by the Thesis

In this thesis, we identified some desirable architectural features for enhancing software reliability and then proceeded on to define a capability-based architecture incorporating these mechanisms. In doing so, we made the following contributions:

- We identified the sources of some major performance problems in existing capability-based architectures in the form of overhead for capability translation, object swapping, invalid capability reclamation, dynamic object creation and overgenerality of various related instructions.
We proposed a technique for generating object names for use in capabilities that guarantees fast capability translation, ensures a simplified structure for the mapping capability translation cache (TLB) and provides a compact way of linking the mapping entries for an object and its refinements. The name generation scheme proposed in this thesis relies on a property found to be valid in all capability-addressed systems -- the number of active objects at any time is fairly small and stable.

We proposed a new and very efficient technique for creating small objects dynamically (in microcode) that relies on the pre-allocation of a large virtual space and cleans deallocated page frames in the background using the garbage collection co-processor. Larger objects can also be created dynamically, with the assistance of a nominal amount of systems software, using this technique.

We have proposed the use of registers explicitly in a capability-addressed environment for storing both data and capabilities and thus exploit the storage hierarchy in a more economical way. (Both the CAP [66] and the PP250 [15] use explicit registers for holding offsets that locate capabilities within memory resident object and for holding translated capabilities, respectively. These architectures do not store capabilities in user-accessible registers.)

We have designed a novel scheme for protected procedure calls that uses special registers for inter-context parameter transfers. This, together with the ability to create a context object dynamically with a very small performance overhead, results in a significantly improved context (domain) switching mechanism.

We have proposed the design of a simple but effective mechanism for implementing abstract data types that is free of the performance problems found in similar
mechanisms in other capability-based architecture. In particular, the mechanism has all
the power of the Hydra abstraction mechanism, albeit it is considerably simpler and
efficient.

- We have designed a very simple mechanism for interprocess communication that uses
two simple instructions for implementing both blocking and non-blocking 'send' and
'receive' operations, and permit the incorporation of either FIFO or priority-based queu­
ing policies.

- To address the problem of detecting run-time errors like subrange violation, access to
uninitialized locations, reference using invalid capabilities, we have proposed a fairly
complete design of a mechanism for detecting errors of these types. The mechanism
imposes no overhead on the instruction execution time. We have also suggested a sim­
ple technique for detecting intermodule parameter inconsistencies, which uses an expli­
cit test for consistency.

- We have also suggested a simple technique for accommodating user-specified exception
handling. The technique relies on some assistance from compiler generated code to
keep track of the dynamic scope of exception handlers and can be easily adapted to
cater to a variety of language-specific rules for locating a handler. The technique sug­
gest for locating a handler can be very easily microcoded.

- Last, but not least, we have addressed the non-trivial problem of processor complexity
and VLSI implementation constraints. We have suggested an effective way of partition­
ing the complex functions in our architecture over two chips, the capability unit (CU)
and the execution unit (EU). The CU is responsible for capability translation, page­
mapping and the implementation of the few complex instructions related to the
capability mechanism. The EU, on the other hand is responsible for implementing simple, register-based data manipulation instructions, controlling the CU and the garbage collector unit (GCU) and issuing instruction fetch requests. The apparent delay in communicating between the EU and the CU is reduced by operating these two units in a pipelined fashion.

6.4 Conclusions

A number of architectural features can be identified for enhancing software reliability. These include mechanisms for efficiently detecting run-time errors, mechanisms for program debugging and tracing, support for user-specified exception handling and mechanisms to enforce a programming discipline and encourage good programming styles. We have presented the design of an architecture that incorporates these mechanisms using capability-based addressing, tagging and a few simple, special hardware mechanisms. Given the need to support these mechanisms, we addressed the issue of processor complexity by taking into account realistic VLSI implementation constraints. The final design that evolved uses programmable registers for holding capabilities and data. There are two major functional units in the proposed design, the execution unit and the capability unit. The execution unit contains the data registers and is responsible for instruction sequencing and implementation of data operations on register-based operands. The capability unit contains the capability registers and is responsible for capability translation, memory management functions and the sole implementation of a few relatively complex instructions related to the capability mechanism. There are a very few simple instructions that are implemented by the capability unit and the
A key consideration in the design of the capability mechanism was simplicity of design and the need to overcome the performance problems of existing capability-based architectures. A comparison of the instruction-set level performance of the proposed architecture indicates that we have successfully met these design objectives. Feature-by-feature, our architecture does significantly better than at least two other architectures (the SWARD system and the Intel iAPX 432) which were designed with software reliability enhancement in mind.

Although, we did not specify exact details of the micro-architecture, our instruction design was based micro-architectural features that have already been implemented in existing VLSI architectures. Finally, we must admit that in designing the proposed architecture we were somewhat fortunate to learn from the experience (and mistakes) of the designers of earlier and similar systems.

6.5 Further Work

Although the definition of the architecture proposed in the thesis and a preliminary evaluation of its low-level performance is fairly complete, further work needs to be done to study the effectiveness of the architecture in enhancing software reliability. To do this, we first need to define the micro-architecture necessary to implement the (macro) architecture and perform a detailed simulation of the system. This would enable us to experiment with various tunable parameters, such as TLB size, number of registers and so on. Although not incorporated in the proposed design, an interesting experiment would be to use bank-switched register sets to allow fast task or domain switching. A micro-architecture simulator would
also open up the possibility of studying the effectiveness of the pipelined operations of the Execution Unit and the Capability Unit. The algorithms presented in this thesis for the various instructions can then be optimized further for this specific micro-architecture. Once this is done, supporting systems software needs to be designed and appropriate benchmarks will have to be identified, or designed to evaluate the benchmark performance of the architecture. Designing an appropriate benchmark for an architecture such as ours is a major research problem in itself. The ultimate test would be, of course, to build a prototype, either as customized VLSI chips or in terms of bit-slice logic, and use the architecture as part of a real system.
7. REFERENCES


8. ACKNOWLEDGEMENTS

This is the usual part of a dissertation where the author gets an opportunity to express his or her gratitude to the people who, in some way or other, have contributed in the generation of this thesis. I would like to make the most of this opportunity.

First and foremost, I am forever indebted to Prof. Robert M. Stewart for his kind guidance, helpful suggestions and constant encouragement throughout the course of this research. I particularly benefited from his ability to provide a global perspective of the technical aspects of this research at various points during our discussions. More than that, he was a friend and support that I could lean on at various times during my four years at Iowa State.

I am also very much indebted to members of my POS committee, Dr. S. C. Kothari, Prof. A. E. Oldehoeft, Prof. A. V. Pohm, Dr. G. O. Strawn and Dr. C. T. Wright, for technical discussions, comments and helpful suggestions. In particular, I would like to cite my discussions with Dr. Kothari, Prof. Oldehoeft and Prof. Pohm at various points during my research. Needless to say, these discussions had a lasting effect on this dissertation. I would also like to express my gratitude to Dr. G. M. Prabhu and Prof. T. A. Smay for acting as substitute committee members during my oral examination.

I remain forever indebted to my friend, Raj Sunderraman, for devoting a substantial part of his time and efforts in helping me during the final production phase of this thesis. A similar vote of thanks goes to my friend, Lakhsman Mukavilli, for acting as my proxy at Iowa State to help me complete the final formalities. I would also like to express my gratitude to
Mrs. LaDena Bishop of the thesis office for her editorial suggestions.

Last, but not the least, I would like to credit my wife Indrani and my son Saugata (Raja) for providing the right environment at home and for their endless patience during the course of this work.
9. APPENDIX I: SELECTED MICRO-PROGRAMS

In this appendix, we describe the details of some of the instructions in Chapters 3 and 4 by presenting the corresponding micro-programs. Some comments are due concerning the notations:

1. MAR indicates the memory address register in the Capability Unit.

2. T0 thru T9 are 33-bit working registers used by the microprogram.

3. CUSR is the status register of the CU.

4. At the end of each step, the best case execution time in cycles, where applicable, is shown within parentheses.

Refer to Section 3.5.1 for the assumed micro-architectural characteristics.
The Capability Translation Micro-Routine

Max-line:  last line number on a page (= page size -1)
CMT-base:  starting physical address of CMT.

STEPS

1) Add last 15 bits of uid in issued capability with CMT-base and put the result in MAR.
   Start memory access (which uses the address in MAR), to read the first word of the desired CMT entry.  (1)

2) Check if the access mode (over) requested is allowed by the access code in the issued capability.  If there is a violation, set appropriate exception flag (Disallowed mode) in the CUSR and generate an exception.  (1)

3) Move fetched word to T0, increment MAR and start the memory read operation for the second word.  T0 contains the fields F0 thru F5a of a CMT entry.  (Refer to Section 5.6.1.) (1)

4) Generate an exception after setting appropriate exception flags in the CUSR under any of the following conditions:
   If F0 is zero (invalid CMT entry).
   If F3 does not match the object type requited by the operation that generated the capability reference (type incompatibility).
   If the offset specified by the reference exceeds F4 (size violation).  (1)

5) Move fetched word to T1, increment MAR and start the memory read operation to fetch the 3rd word of the CMT entry.  T1 now contains the second word of the CMT entry, comprising of field F5b.  (1)

6) Compare the last 5-bits in T0 (field F5a), concatenated with the contents of T1 (field F5b) with the most significant 37 bits in the issued capability.  If there is a mismatch, set the appropriate exception flag in the CUSR (Invalid Capability) and generate an exception.  (1)

7) Move the fetched word to T2.  T2 contains the field F6.  (1)

8) Add the logical page number specified on the reference to T2 and move the result to MAR.  Start the memory read operation to fetch the 1st word forming the PMT entry.  (1)

9) Move the fetched word (1st word of a PMT entry) into T0.  Generate an exception, after setting appropriate flags in the CUSR under the following conditions.
If the presence bit is off and the initial fault flag is set (initial fault -- requires a clean frame to be allocated).

If both the presence bit and the initial fault flag are reset (normal page fault). (1)

10) Concatenate the frame number from the PMT entry fetched in step 9 with the line number specified in the access request and move the result to MAR. (1)

11) Establish a TLB entry for this page as follows:

   Invalidate the least recently used entry in the TLB (using the LRU hardware).

   Compare the page number in the reference with the last page. If the reference is to the last page, set the line limit field in the TLB to the line number specified in field F4, otherwise set it to max-line. Set the page number field in the TLB to the page number specified in the reference.

   Set the frame number field in the TLB to the frame number obtained from the PMT entry. (2)

12) Access memory in the requested mode (using the address in the MAR) and restart the operation that generated this reference.
Micro-routine for: CREATE Cj, Ck

2^p is the page size

1) If Fj div (2^p) > 0, raise an exception, setting appropriate code ('attempt to create an object larger than a page'); save Fj in T1. (1)

2) Check if the type field in Cj specifies a VSO object -- if not, set appropriate exception code ('attempt to use a non-VSO in CREATE') and raise an exception; clear Fj. (1)

3) Translate the capability reference (Cj, Fj) and save the resulting address in MAR. (The controller specifies a read-write access for this translation -- violations can cause exceptions.) (1)

4) Execute the allocation algorithm of Figure 4.9. Assume that PMT_base is returned in registers T2, respectively. (As shown in Figure 4.9, this step may generate some exceptions.) (17)

5) Obtain a uid into register Ck for the object being created from the system-maintained list of available uids. Start the uid-generator if the number of uids in this list falls below a prespecified threshold after allocating a uid for the object being created. (9)

6) Obtain the address of a clean frame in T3 from the pool of cleaned frames maintained by the system. Generate an exception ('unable to allocate frame') if a clean frame is not found. (9)

7) Set up the frame address in the PMT entry located in step 4. Set the presence flag in this entry and reset the initial fault flag. (2)

8) Locate the CMT entry for the newly created object based on the last 15-bits of its uid and initialize the various fields as follows:
F2 := 000; (not locked)
F3 := 'generic';
F4 := T1 (size of object);
F5 := most significant 37 bits of uid;
F6 := T2;
F7 := 0 (object carved out of VSO);
F8 := last 15 bits of uid of VSO in Cj;
F9 := 0; (9)

9) Complete the capability in Ck, by writing a fully-privileged access code and the object type into the respective fields. (1)
Micro-routine for: DELETE Cj

1) Check if the capability in C has the 'destroy' privilege. If not, raise an exception after setting appropriate exception code (unauthorized attempt to delete object). (1)

2) Use the last 15-bits in the uid of the capability in C to look up the corresponding CMT entry. Store the last two words of this entry in registers T1 and T2 respectively. (5)

3) If the most significant bits in the uid of the capability and the CMT do not match, raise an exception, setting appropriate condition code (attempt to use invalid capability). (T1 and T2 contains fields F6 and fields F7 through F10 of the CMT entry, respectively, at this point.) (Refer to section 5.6.1 for a Description of these fields.) (1)

4) Examine field F7 in T2. If this is 1, make a system call and exit. In this case, the object was created through a system call and appropriate system routines are called to complete the deletion process. (If F7 is zero, the object was created using a CREATE instruction, and the routine continues.) (1)

5) Use the CMT offset specified in field F8 (of T2) to locate the CMT entry for the VSO from which this object was carved out. Extract the base address of the VSO from field F6 of this entry and save it in the MAR. (3)

6) Use the value in field F6 in T2 to locate the PMT entry for the object being destroyed, and fetch the first word of the PMT entry in T3. (2)

7) Attach the number of the frame occupied by the object being deleted to the list of page frames to be cleaned by the Garbage Collector Unit (GCU) and signal the GCU. (7)

8) Set the initial fault flag and turn the presence bit on in the PMT entry located in step 6. (2)

9) Deallocate the CMT entry for the object by setting field F0 to zero; update the bit vector used by the uid generator for CMT allocations. (4)

10) Purge the TLB entry for this object, if any. (1)

11) Compute the logical page number, l, occupied by the object being deleted, in the VSO-space as:

   $$ l := \text{address of PMT entry (field F6 in T2)} - \text{base address of VSO (in MAR)} - 3 $$

   (Recall that the VSO header occupies 3 words). (2)

12) Execute the deallocation algorithm of Figure 4.10. (7)
Micro-routines for: CONVERT Cj, Fj and BIND_TYPE Cj, Fj

1) Check if the type seal flag (bit 22) in the capability is set. If set, raise an exception, setting appropriate exception code (object already bound to a type).

2) (For BIND_TYPE only): Check if the object type specified in Fj is one of 'generic', 'data', 'SCO'. (For CONVERT only): Check if the object type specified is one of 'generic', 'data', 'code', 'SCO', 'port' or 'system'. Raise an exception, setting appropriate condition codes if otherwise (attempt to create illegal type).

3) Write the type specified in Fj into the respective field in the capability in Cj, as well as in the corresponding CMT entry.

4) Set the type seal flag in the capability in Cj to 1.
Micro-routine for: PUSH Cj

1) Translate the capability reference (CC, CF). (1)
2) Write first half of capability in Cj to this address; increment CF; translate the capability reference (CC, CF), corresponding to the new value of CF. (2)
3) Write the second half of the capability in Cj to the last translated address; increment CF.

Note: Context object overflow exceptions may be generated during address translation in steps 1 and 2.

Micro-routine for: PULL Cj

1) Decrement CF. (If CF < 0, the hardware will raise an exception ('context object underflow') automatically.) (1)
2) Translate the capability reference (CC, CF). (1)
3) Read first half of capability in word addressed into lower half of Cj: Decrement CF; (If CF < 0, hardware will raise an exception ('context object underflow')); translate the capability reference (CC, CF). (2)
4) Read second half of capability in word addressed in the upper half of Cj.

Note: In steps 3 and 4, the fetch hardware may generate an exception if the words read are not, respectively, the first and second words of a capability ('attempt to read non-capability into capability-register').
Micro-routing for: PCALL Cj

1) Check if Cj contains a SCO-capability with the raf bit reset. If otherwise, generate an exception ('attempt to use improper code object capability in domain switch'). Concurrently, translate (CC, CF). (1)

2) Save the following status and control registers in the activation stack within the current context object: (12)
   a) PC and PF
   b) CU-status (assumed conservatively as 1 word)
   c) EU-status (assumed conservatively as two words)

3) Save Cj in T0, T1 (temporary registers); clear registers D8 through D23, C8 through C15, F8 through F15. (This 'clear' step actually resets the valid tag associated with these registers and is done with the assistance special hardware, overlapping with step 2 above.) (0)

4) Push CC, CF onto the Context Stack Object (CSO) (7)

5) Create a context object (using the algorithm for CREATE), and return its capability in CC. (50)

6) Establish a TLB entry for this object and clear CF. (4)

7) Translate the capability in T2, T3, specifying an offset equal to the number in the 5-bit access field. (1)

8) Fetch the contents of this location into PF; concurrently, move T2, T3 to PC (Establishes 'new' program counter) and transfer to the fetch microroutine. (1)
Micro-routine for: PRET

1) Destroy the context object of the callee (pointed by CC). (This uses the same algorithm as DELETE.) (36)
2) Translate the capability reference (CCS, CCF). (1)
3) Use this translated address to restore CC and CF from the context stack object. (6)
4) Translate the reference (CC, CF). Concurrently, clear all but the parameter registers. (1)
5) Use the translated address to restore the following status and control registers: FC, PF, CU-status, EU-status, and transfer control to the fetch microroutine. (12)
Micro-routine for: ASSOCIATE-SCO Cj, Dk

1) Raise on exception if Cj does not specify a capability to a generic object ('incompatible object for SCO-association'); clear T0. (1)

2) Translate the capability reference (Cj, FRj); move the last 5-bits in Dk (which specifies the access privilege needed on amplification) to T0. (1)

3) Move the contents of PC (which contains a capability to the currently executing code object or SCO) to T1, T2. Raise an exception if the capability in T1, T2 does not refer to a SCO. ('Attempt to associate with a non-SCO'.) (1)

4) Set the sequence bit (bit 31) in T2 (which now contains the second word forming the capability in PC) to zero; remove all access privileges from the capability in T1, T2. (1)

5) Store T1, T2 and T0 (in this order) within the generic object, starting at the address, obtained in step 2 above (after translation). (2)

NOTES:
1) This instruction stores the 'nulled' SCO capability within the generic object, as well as the access code needed for amplification. The SCO capability is stored permanently -- it cannot be retrieved for use as a capability by any (macro) instruction, since the second word in this capability has an incorrect sequence bit.

2) Circuitry incorporated within the Capability Unit will raise an exception if the capability in Cj does not have 'write' privilege or has its raf bit cleared, in steps 1 or 5, as appropriate.
Micro-routine for: AMPLIFY Cj

1) Translate the capability reference (Cj, Fj); disable the circuitry that checks the sequence bits of a fetched capability. (11)

2) Fetch the two words starting at the address obtained in step 1 (after translation) as a capability into registers T0 and T1. (Circuitry associated with the fetch buffer will generate an exception if these words are not initialized or do not represent capability items. The sequence bits will not be checked, since the sequence checker was disabled in step 1.) (4)

3) Fetch the next word into T2 as a non-capability item. (Exceptions similar to those listed in step 2 above, appropriate for non-capability items will be generated for violations); raise an exception if the sequence bits in both T0 and T1 are not zero. ('Attempt to use a non-null capability in amplification'.) (2)

4) Compare the uid of the capability in PC with those in T0, T1. Generate an exception if the uids are unequal. ('Attempt to amplify by unassociated SCO'.) (2)

5) Copy the last 5-bits in the word stored in T2 to the access code of the capability in Cj; set the raf-bit in the capability in Cj. (1)

Micro-routine for: DEAMPLIFY Cj

1) Same as step 1 for the AMPLIFY instruction. (11)

2) Same as step 2 for the AMPLIFY instruction. (4)

3) Raise an exception if the sequence bits in both words are not zero. ('Attempt to use a non-null capability in deamplification'.) Compare the uids of the capability in PC and that in T0, T1. Raise an exception if the uids are unequal. ('Illegal attempt to deamplify'.) (2)

4) Remove all the access privileges in the capability in Cj (by AND-ing 00000 onto the access code field of the capability in Cj); clear the raf-bit of the capability in Cj. (1)
Micro-routine for: CREATE-REFINEMENT Cj, Ck

Let R be the refinement object that is created by this instruction, from a parent object, P

1) Generate an exception, after setting appropriate exception codes under any of the following conditions:
   a) The type of P (whose capability is in Cj) is not one of 'generic', 'code' or 'refinement'.
   b) The capability to P, given in Cj, does not have the 'destroy' privilege.
2) Fetch the CMT entry for P into registers T0, T1, T2 and T3. Generate an exception, setting appropriate exception codes under any one of the following conditions:
   a) If the starting address for the refinement R (specified in Fj) is higher than the size of the parent, P.
   b) If the last address in R (which is the sum of contents of Fj and Fk minus 1) is higher than the size of the parent minus 1. (These two tests ensure that the address space for R is totally subsumed in the address space if P.)
3) Obtain an uid for R, store it in the last 52-bits in Ck. Copy the access codes & type fields in Cj to the corresponding bit positions in Ck.
4) Check if bit F9 in the CMT entry for P (fetched into T0 through T3) is zero. If F9 contains a one (then refinements were already created from the parent), go to step 6.
5) Store the last 15-bits in Ck (which locates the CMT entry for R) in field F10 of the fetched entry for P; also set F9 in this entry to 1. Store the updated CMT entry of P back (from T0 thru T3) and go to step 7.
6) (Executed if P has siblings.) Use field F10 in the CMT entry for P (in T3) to locate the CMT entry for the first child of P. From this entry, follow the sibling chain (field F10) till a CMT entry with a null sibling link is located. Insert a one into field F9 and the last 15-bits in Ck into field F10 of this entry.
7) Form the CMT entry for R as follows:
   (i) Store a zero in field F1.
   (ii) Store the type code for 'refinement' in F3.
   (iii) Store the contents of Fk in field F4a (size of refinement).
   (iv) Complete the starting virtual address of R in the virtual address space of the earliest ancestor as follows. If the parent P is not a refinement, this is simply the contents of Fj. Otherwise (if P is itself a refinement), this is the sum of the contents of fields F4b and F6a concatenated (from the CMT entry for P) and the contents of Fj. Store this address in fields F4b and F6a for R.
(v) If P is a refinement object itself, copy the contents of field F6b of the parent's CMT entry to the same field in the entry for R. Otherwise, copy the last 15 bits in Cj to F6b.

(vi) Set field F7 to zero (no next sibling for R).

(vii) Set field F9 to zero (R has no children).
Micro-routine for: LOCK Cj

- The interpretation of the lock bits (field F2 of a CMT entry, Section 5.6.1) are as follows:
  - 0XX (X= don't care) -- object is unlocked,
  - 100 -- object is locked and no process is waiting for it to be unlocked,
  - 11X -- object is locked and at least one process is waiting for it to be unlocked.

- Both the LOCK and UNLOCK instructions bypass the TLB entry, if any, for the object.

1. Generate an exception if the capability in Cj is invalid or if it does not have either the 'read' or the 'write' privilege and the 'lock' privilege. (1)

2. Multiply the last fifteen-bits in Cj by four (by shifting left two places) and add the result to the base address of the CMT; store this result in MAR. (The MAR now contains the address of the first word forming the CMT entry for the object being locked.) Copy MAR to T2. (1)

3. Using the address T0 and T1. (At this point fields F1 through F5b of the CMT entry are in T0 and T1.) (4)

4. Check if field F1 is zero; concurrently, compare the contents of fields F5a and F5b with the most significant thirty-seven bits in the uid of the capability in Cj; generate an 'attempt to use invalid capability' exception if F1 is zero (obsolete CMT entry) or if the result of the comparison indicates that the comparands are unequal. (2)

5. Copy T2 to MAR, and indivisibly perform (i) through (iv) below:
   (i) Read the word located by MAR into T0; copy T0 to T1. (Both T0 and T1 contain the first word of the object's CMT entry.) (2)
   (ii) If field F2 in T0 has the value '0XX' (X= don't care), write '100' to the same field in T1, otherwise write '110' into the same field (viz., F2) in T1. (1)
   (iii) Write T1 back to the address pointed to by MAR. (2)
   (Since instructions are interruptible at the microinstruction level, these three steps are performed indivisibly. One way to do this would be to acquire mastership of the memory bus and disable all interrupts.)
   (iv) If field F2 of T1 contains a '100', copy Cj and the code necessary to request suspension of the executing process into the systems register and generate an exception. (1)
Micro-routine for: UNLOCK Cj

See description of LOCK Cj for applicable notes.

1 through 4. Identical to corresponding steps for LOCK. (8)
5. Copy T2 to MAR and indivisibly perform (i) through (iii) below:
   (i) Read the contents of the word addressed into T0; copy T0 to T1. (2)
   (ii) If field F2 in T0 is equal to '100' set the field F2 in T1 to '000' and write T1 back to
        the location pointed to by MAR. (3)
   (iii) If field F2 in T0 is equal to '110', copy Cj and the code necessary to request revival of
        a waiting process into the systems parameter register and generate an exception. Other­
        wise (field F2 in T0 = 0XX), generate an error exception ('attempt to unlock already
        unlocked object'), after setting appropriate condition codes. (1)
Micro-routine for: SEND Cj, Ck

- The capability register Cj and the corresponding offset register Fj specifies the message to be sent to the port specified by the capability in Ck.
- Fj specifies the message size (long or short) and a sixteen-bit literal message. If the message requires more than sixteen bits, Cj specifies a generic object containing the rest of the message. Short messages (up to sixteen bits) are sent via Fj only-- in such cases, the contents of Cj are irrelevant.
- Fj also specifies if the sending process is to be awakened when the message is read by the receiving process.

1. Generate an exception if the capability in Ck is invalid ('attempt to use invalid capability of port operand') or if Ck lacks 'write' privilege ('insufficient port privilege') or if Ck does not contain a capability to a port object ('attempt to send to non-port type').
   (2)
2. Save Fk in TO; clear Fk. (1)
3. Translate the capability reference (Ck, FK) into the MAR. (1)
4. Using the translated address, fetch the first two words in the header (maximum and actual queue sizes) into T4 and T2, respectively. (4)
5. If T1=T2, then set the 'negative' condition code flag, reset the 'positive' condition code flag, (Queue is full), and exit. (1)
6. (Queue is not full): Increment T1 and write it back to the location pointed to by MAR. (This updates actual queue size.) (2)
7. Increment MAR and fetch the contents of the addressed location in T2. (T2 now contains the third word in the header of the port object that gives the queing priority, waiting status and the offset for the first element in the free list.) Copy MAR to T1. (3)
8. (Now insert the message into the first free element):
   (i) Copy the offset for the first free element (from T2) to Fk; translate the capability reference (Ck, Fk). (2)
   (ii) Store the contents of Fj at the resulting location; increment Fk and translate the reference (Ck, Fk). (2)
   (iii) If the message is short (determined by checking the 'size' flag of Fj), add two to Fk, translate the reference (DRk, Fk), go to step (vi). (2)
   (iv) Store the first word of the capability in Cj at the resulting address; increment Fk and translate the capability reference (Ck, Fk). (2)
(v) Store the second word of the capability in \( C_j \) at the resulting address; increment \( F_k \) and translate the capability reference \( (C_k, F_k) \). (2)

(vi) Store the first word of the capability to the process executing send from control register TC, after removing all access privileges, at the resulting address; increment \( F_k \) and translate the reference \( (C_k, F_k) \). (3)

(vii) Store the second word of the capability in TC at the resulting address; increment \( F_k \) and translate the capability reference \( (C_k, F_k) \). (2)

(viii) Copy \( T_2 \) to \( T_5 \) and fetch the last twenty bits of the addressed location into the corresponding bit positions in \( T_2 \); copy the MAR to \( T_3 \). (3)

(ix) Copy \( T_1 \) to MAR, and store \( T_2 \) into the resulting location. (This updates the pointer to the free list.) (2)

(x) Clear \( T_4 \); obtain the priority of the executing process from control register TP, and store this into the most significant eight bits of \( T_4 \). (1)

9. Determine the queuing priority from the contents previously fetched in \( T_2 \) (in step 7). If it is 'process-priority-based', go to step 11. (1)

10. (Queueing priority is FIFO): Add the newly formed element to the end of the queue as follows:

   (i) Increment \( T_1 \) and move it to MAR (to make MAR point to the fourth word in the port, containing the offset of the last element); fetch the word addressed by MAR into \( T_2 \). (2)

   (ii) Add five to the contents of \( T_2 \) and transfer the sum to \( F_k \); translate the capability reference \( (C_k, F_k) \). (This gives the address of the last word in the last element in the queue.) (2)

   (iii) Fetch the word at the resulting address in \( T_2 \); write the lower twenty bits of \( T_5 \) (offset of newly-created element) into the lower twenty bits of \( T_2 \), keeping the rest of \( T_2 \) unaffected. (3)

   (iv) Store \( T_2 \) back into the location addressed by MAR. (This links the newly formed entry to the end of the queue.) (2)

(v) Copy \( T_3 \) to MAR; store \( T_4 \) at the resulting address and go to step 12. (3)

11. (Queueing priority is process priority): Add the newly-created element to its correct position within the queue as follows:

   (i) Add two to the contents of \( T_1 \) and copy \( T_1 \) to MAR. (MAR now points to the port word containing the offset of the first queue element.) (1)

   (ii) Fetch the contents of the addressed word in \( T_1 \); copy \( T_1 \) to \( T_6 \). (3)

   (iii) Move \( T_1 \) plus 5 to \( F_k \); translate the capability reference \( (C_k, F_k) \) and fetch the addressed word in \( T_2 \). (4)

   (iv) If the priority of the executing process (in control register TP) exceeds the priority listed in the most significant eight-bits of \( T_2 \), go to step (vi). (1)

(v) (Continue searching): Copy \( T_1 \) to \( T_6 \); transfer lower twenty-bits in \( T_2 \) to \( T_1 \) and go to step (iii) above. (2)
(vi) (Insert newly-created element following the one pointed by T6): Move T6 plus 5 to Fk; translate the reference (Ck, Fk) and write the lower twenty-bits of T5 (offset of the newly-created element) to the resulting address. (4)

(vii) Copy the lower twenty-bits in T1 to the lower twenty-bits in T4, keeping the rest of T4 unaffected; move T3 to MAR (to make it point to the last word in the newly-created element); write T4 to the resulting address. (3)

12. Set the 'positive' condition flag and reset the 'negative' condition flag; restore Fk by copying T0 to Fk. (2)

13. If T7 indicates that any process was waiting the arrival of a message, generate an exception to make a system call to revive one such process. (Note that an initial queue size of zero does not necessarily indicate that receiving processes were waiting for a message to arrive!) (1)
Micro-routine for: RECEIVE Cj, Ck

Refer to description of SEND for applicable notes.

1. Generate an exception if the capability in Ck is invalid ('attempt to use invalid capability for port operand') or if it lacks 'read' privilege ('insufficient port privilege') or if it is not a capability to a port object ('attempt to receive from non-port type'). (2)

2. Save Fk in T0; move 1 to Fk. (1)

3. Translate the capability reference (Ck, Fk) into the MAR. (1)

4. Using the translated address, fetch the second word in the port header into T1. (2)

5. If T1 is not zero (queue is not empty), decrement T1 and write it back to the address in MAR, go to step 7. (3)

6. (Queue is empty): Increment MAR and fetch the word addressed in T2; set the 'negative' condition code flag and reset all other condition code flags; if Fj is zero, set the 'waiting receiver' flag in the corresponding field in T2 and write T2 back into the port object using the address in MAR; move T0 to Fk; unlock the TLB entry for the port object and exit. (6)

7. Add two to the contents of MAR (to make it point to the fifth word in the port header), and fetch the addressed word into T1. (T1 now contains the offset of the first queue element.) (2)

8. Copy MAR to T2 (to save the address of the fifth word in the port header for future use); move the lower twenty bits in T1 to Fk and translate the capability reference (Ck, Fk) into MAR. (2)

9. Fetch the addressed word into Fj. (This copies the message type, message size and the sixteen bit literal message into Fj.) (2)

10. Check the message size field in Fj. If the message is short, go to step 12. (1)

11. (Message is long): Increment Fk and translate the capability reference (Ck, Fk); fetch the addressed word in the upper half of Cj; increment Fk and translate the capability reference (Ck, Fk); fetch the addressed word into the lower half of Cj. (This copies the capability to the generic object containing the rest of the message into Cj.) (6)

12. Check the message type field in Fj; if the message does not specify that the sending process be awakened when the message is being read (by a RECEIVE), add two to Fk and go to step 14. (2)

13. Increment Fk and translate the capability reference (Ck, Fk); fetch the addressed word into the second system parameter register; increment Fk and translate the reference (Ck, Fk); fetch the addressed word into the third system parameter register (to load the capability to the sending process object into the system parameter registers); move a code to request the revival of the above process into the first system register. (6)
14. Increment Fk, translate the capability reference (Ck, Fk) and fetch the addressed word in T1. (3)

15. Move T1 to MAR; read the word addressed into T2 (this reads the third word in the port's header into T2); copy the last twenty bits in T1 to the lower twenty bits in T2 (keeping the rest of T2 unaltered) and write the contents of T2 to the address in MAR. (This updates the pointer to the first element in the queue.) (6)

16. If the message type field indicates that the sender be revived, generate an exception to make a system call to revive the sending process (specifying the parameters in the system registers).
10. APPENDIX II: ABBREVIATIONS AND GLOSSARY

AVSL

Available Virtual Space List: A list that indicates the location of available and allocated logical pages within a large virtual space defined by a virtual storage object. This list is implemented as a bit-map within the virtual storage object.

CISC

Complex Instruction Set Computer: A computer with an instruction set characterized by a large number of instructions and/or high-level instruction semantics.

CMT

Capability Mapping Table: A system-maintained table used to map capabilities to physical addresses. This table is always resident in the primary memory and can map capabilities to at most 32 K live objects. A capability translation requires a simple, direct lookup of this table.

CSO

Context Stack Object: An object of 'system' type that serves as a stack of capability-offset pairs to context objects in the dynamic calling environment. This object thus holds the information used for the restoration of the caller's protection domain on return from a protected procedure call.

CU

Capability Unit: The co-processor that implements all major functions related to the capability mechanism. The capability and offset registers are located within this co-processor chip, along with a simple arithmetic/logic unit and an address translation buffer. The capability unit also serves the role of a memory management unit and implements a paged object space.

EU

Execution Unit: The main processor in the system. This unit is responsible for implementing the data manipulation operations and control sequencing. It uses the other co-processors as slaves. The Execution Unit has a register-oriented, reduced instruction set and can thus be characterized as a RISC. Most of the data manipulation instructions execute in a single cycle.
GCU

Garbage Collector Unit: The co-processor that implements the garbage collection operations concurrently with the operation of the other units. It is also responsible for clearing de-allocated page frames before their re-allocation in order to prevent the accidental inheritance of capabilities.

HLL

High Level (Programming) Language.

HLSO

Handler List Stack Object: An object of 'system' type that holds pointers in the form of capability-offset pairs to the exception handler tables within the code objects or SCOs in the dynamic calling environment.

TLB

Translation Look-aside Buffer: A fully-associative cache maintained within the Capability Unit to hold recently translated capability-page references.

PMT

Page Mapping Table: A table used to map references to logical pages within an object. These tables are located within the virtual storage object whose virtual space subsumes the logical address space of the object.

RISC

Reduced Instruction Set Computers: Computers with a simple register-oriented instruction set that permits a very area-efficient VLSI implementation, leaving more room on the chip for mechanisms that speed up the instruction execution rate.

SCO

Structured Code Object: An object containing executable code, with up to 32 pre-specified entry points. These objects are used in the implementation of abstract types and modules with well-defined interfaces.

TAC

Trace Address Cache: A cache implemented within the Capability Unit to generate exceptions on the occurrence of pre-specified events in real-time. This cache can be loaded under program control and is extremely useful as a de-bugging or instrumentation aid.
uid

*Universal Identifier:* The system-wide unique name for an active object. This name is independent of the disposition of the object in the storage hierarchy.

**vNA**

*von Neumann Architecture:* The architecture of traditional computers, embodying a 'typeless', co-ordinate-addressed, linear storage.

**VSO**

*Virtual Storage Object:* An object that defines a large virtual space, the status of logical pages within this space and their virtual-to-physical address mappings. The virtual storage object is always resident in the primary memory. Small objects are 'carved out' of the virtual space defined by the virtual space object.