M.I.T. Laboratory for Computer Science

March 28, 1979

Computer Systems Research Division

Request for Comments No. 169

The Architecture of an Object Based Personal Computer

Allen W. Luniewski

Attached is a copy of my recently accepted doctoral thesis proposal.

Massachusetts Institute of Technology

Cambridge, Massachusetts

Proposal for Thesis Research in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

Title: The Architecture of an Object Based Personal Computer

Submitted by: Allen W. Luniewski
350 Engamore Lane, #205
Norwood, Massachusetts
02062

Signature of Author

Date of Submission: March 26, 1979

Brief Statement of the Problem

This thesis will explore the problem of designing the architecture of a personal computer that directly supports the notions of data abstraction and control abstraction, the key notions of structured programming. Both kinds of abstraction allow the programmer to formulate solutions to problems in terms of higher level entities, the abstractions, and then later implement these high level entities in terms of lower level entities. This method of problem solution encourages the production of correct solutions and the architecture, by encouraging the use of abstractions, thus helps the programmer produce correct solutions to problems. The goal of the architecture is to separate the implementation issues of structured programming languages from the higher level language issues. The thesis will show that high level languages can be easily compiled onto the architecture. The thesis will also show that efficient implementations of the architecture, without concern for the higher level language issues, exist for personal computers. The environment of a personal computer has been chosen since the present trends of hardware technology indicate a future trend towards distributed computing and personal computing. The thesis will take advantage of the "personal" nature of personal computers by either avoiding, or by solving in a simple manner, many of the problems that are difficult to solve in the context of a shared general purpose computer.

In recent years there has been a tremendous interest in the area of structured programming. This interest has taken many forms including the development of languages that support and encourage the structured programming methodology. These languages have been implemented on classical von Neumann computers. I propose to specify the architecture of a new, personal computer - one that will make the implementation of these structured programming languages easy. The architecture will encourage and support the use of abstractions, the key notion of structured programming. It will put the user in an object oriented world; a world in which everything is an entity provided by some data abstraction, an object, and will not let the user out of that world. It will be the architecture of a personal computer in order to explore the impact that the personal computing environment has on such an architecture.

The next two sections will introduce the notions of structured programming and personal computing. This will be followed by a description of the proposed research problem and a proposed solution to that problem.

Introduction - Structured Programming

The desire to produce correct, easily maintained programs has been around since the first days of computers. In recent years computer scientists have investigated ways to make this desire a reality. This has resulted in the programming methodology known as structured programming.

Although sometimes referred to as "go-to less" programming[7], structured programming is better regarded as a methodology or philosophy of programming whose general theme is "abstraction". The basic idea is to have the programmer think in terms of higher level entities in order to formulate the solution to a problem instead of thinking only in terms of very low level entities such as bits and integers. Later these high level entities are implemented in terms of lower level entities and these in terms of still lower level entities until, finally, a working program in terms of the lowest level primitives of the language is written. This process has variously been referred to

as top-down design or stepwise refinement[4,6,19]. These higher level entities are the abstractions of structured programming. Abstractions come in two basic forms - control abstractions and data abstractions.

Control abstractions are intended to abstract away from the means of control of flow provided in most computers, the go to statement, and provide the user with other means of specifying the flow of control in his program. Some control abstractions have been provided in all higher level languages including even Fortran. The notion of procedural abstraction, one of the most common forms of control abstraction, is provided by Fortran in the form of subroutines and functions. These allow the programmer to make use of common, parameterized pieces of code that, in principle at least, satisfied some input-output specification. The programmer may then use that procedure knowing only its specification (e.g. the abstraction that it implements) and not its implementation. The do construct of Fortran is a useful abstraction that allows the programmer to specify the repetitive execution of a sequence of statements while stepping an integer variable with each execution of the body of the loop. It is an abstraction in two ways. First, it embodies a common pattern of code:

```
      <initialize>
loop: <body>
      <increment>
      if <test> then go to loop
```

in one language-provided construct. Thus, in this sense, it is an abstraction of a common usage pattern of more primitive constructs. It also abstracts the notion of a parameterized body of code that is executed for sequential values of its parameter. This is an especially powerful notion when used in connection with arrays. It is, however, the only control abstraction provided for use within program units, all other control amounts to providing a (conditional) transfer instruction. Subsequent languages have improved upon this situation by adding additional control abstractions for use within a program unit. Features such as while loops, case statements and fancier looping constructs were added to make the programmer's task easier. Recent languages such as CLU[12]

and Alphard[20] have provided control abstractions that are closely associated with data abstractions. These data abstractions are in the form of looping constructs that permit repetitive execution of a body of code while assigning new values to a variable of abstract type on each such execution. Thus the presence of control abstractions in languages is well established and fundamental to current programming techniques.

Data abstractions are the other kind of abstraction. They allow the programmer to formulate solutions to problems in terms of high level entities, data objects, provided by data abstractions, without worrying, initially at least, about the details of their representation or how operations on those entities are implemented. Data abstractions have been provided in even the earliest languages such as Fortran. Whereas the underlying hardware in reality provides only bit strings, Fortran provides the notion of typed cells in memory.[1] The user may have integer, real and boolean variables and can perform the appropriate operations upon those variables. He does not have to worry about the details of the implementation of those types. Later languages added new types such as character strings, procedure variables and pointers. In addition Fortran allows the user to group data of the same type into aggregates called arrays. Such aggregates allow the grouping of related data in a way that allows easy, indexed access to any piece of that data. Languages such as Cobol, Algol-W and PL1 introduced the notion of structures (also called records). These allow the programmer to collect related data into one place and name and use the whole collection of data as one entity. For instance, the payroll information for an employee might be one record in a program instead of a collection of variables that just happen to all be part of one logical entity - the employee's payroll information. Still later, languages such as Simula[2], CLU and Alphard introduced the notion of user defined data abstractions. Whereas in previous

1. The fact that the type mechanism is easily circumvented is not relevant at this point.

languages the only data abstractions (or types) provided were those that the language designer deemed necessary, these latter languages allow the user to build his own data abstractions. Thus a user who is writing a payroll program may have a data abstraction that is a "company-payroll" that contains entities "payroll-record" instead of thinking in terms of data files and integer and character variables. Clu and Alphard differ from Simula in that they enforce the implicit specifications of the data abstractions; that is, they permit the user of a data abstraction to use the entities provided by that abstraction, the objects, only in the ways defined by that abstraction. Thus we see that the notion of data abstraction is also deeply rooted in the history of programming languages and has evolved through the years.

Introduction - Personal Computing

The areas of distributed computing and personal computers are also relevant to this proposed thesis. In recent years, as the price of hardware has continued to decline, there has been a great increase in interest in these areas. The basic idea is to place a large amount of computing power into a relatively inexpensive computer. This leads to visions of a computer in every office or, perhaps, a computer on everyone's desk. These computers would serve local needs but would also need to communicate with each other. Such communication would be needed to pass information from one computer to another in order to mimic the channels of communication present in the non-computer world. The need for centralized computers would be limited to special applications such as a centralized file machine or a machine with special capabilities such as an extremely fast "number crunching" processor. In short, for many applications the era of the shared general purpose computer seems to be over; the shared general purpose computer is about to be replaced by collections of personal computers connected by a communication network of some sort with a small collection of special purpose computers to serve these personal computers in special

cases.

For the purposes of this proposed thesis the most important attribute of personal computers is their simplicity. This simplicity is a result of two factors. First, personal computers must be simple in order to keep their cost down - only relatively inexpensive computers are candidates for personal computers. Second, personal computers seem to be intrinsically simpler than the more traditional general purpose computer because certain functions of general purpose computers, in particular protection, resource allocation and accountability, are relatively less important in a personal computer since the owner of a personal computer is generally its only user.

Distributed processing itself is relatively unimportant to this thesis. The major issues in distributed processing center around when and how computers interact with each other. Such issues are at a much higher level than the architecture to be designed in this proposed thesis and so are relevant only in that the architecture to be proposed in this thesis should not impose a limitation on the ways in which these issues can be addressed.

Proposed Research

To summarize the previous two sections, three notions are crucial to the proposed research. Data abstractions allow the user to create objects for which only the specification need be known and not the implementation of the object. Control abstractions allow the user to abstract certain stylized patterns of specifying the flow of control in programs. These two notions are the pillars of the area of structured programming. Finally the trend towards personal and distributed computing must be reckoned with in proposing any new computer architecture.

The proposed research combines these three notions: its goal is to design the architecture of a new computer - a personal computer that directly supports the notions of data abstractions and control abstractions. The design will be at the instruction set processor level; that is, it will specify the interface that an assembly language programmer would see.

The architecture to be specified in this thesis can be interpreted in two general ways. First it can be taken as the architecture of a new computer. That is, it is assumed that at some point in time the ideas of this thesis will appear as a computer - a piece of hardware. The other way of looking at this thesis is as specifying an intermediate language into which languages such as CLU and Alphard can be compiled. This point of view leads to treating the architecture as a means of writing a machine independent compiler for these languages. A third, and unifying, way of looking at the architecture of this thesis is that the architecture is a way of separating the implementation issues of high level languages such as CLU and Alphard from the higher level language design issues.

The implementation of a high level language, when done in terms of this architecture, frees the language designer and compiler writer from having to worry about many of the issues normally associated with such tasks, such as storage management, procedure calls and the implementation of the basic types, since the proposed architecture will be a "nice" environment onto which to compile the high level languages. Most of the implementation issues should be hidden by the architectural interface.[1] Thus the architecture serves as an implementation independent intermediate language for high level language compilers.

---

1. Not all implementation issues for a particular language, though, are likely to be handled in an architecture unless the architecture is specifically designed for that language.

The implementer of the architecture is presented with a well defined, bounded problem since the architecture is not open ended. The implementer can choose data representations without regard to the needs of the high level languages but only with regard to the peculiarities of the particular machine or hardware base being used. Thus the architecture is suitable for implementation by interpreting it in software or by actually building a new piece of hardware.

The major result of this thesis will be the demonstration that an architecture exists that separates implementation issues from high level language design issues. The next section of this proposal will outline some concrete design goals for the architecture. These goals have been chosen for two general reasons. First they specify an architecture that addresses the issues of the previous section - structured programming and personal computers. Secondly the goals have the effect of specifying an architecture that achieves the separation mentioned in the previous paragraphs.

Design Goals

The architecture to be designed must be suitable for a personal computer due to current trends towards the use of personal computers and not large, centralized computers. This limits the complexity allowed in the architecture since an overly complex architecture will not be implementable at reasonable cost and thus would be unsuitable as a personal computer for economic (but not technical) reasons. Therefore this thesis will make an effort to show that the features of the architecture are implementable in a fairly simple manner[1] and thus suitable for a personal computer. This means that the resulting implementation must be efficient. Storage

---

1. This does not mean that a cost-benefit analysis will be performed on each feature of the architecture. Rather, subjective arguments will be presented that the architecture meets this goal.

overhead must be low; the storage used by the implementation[2] must not be an excessive fraction of memory capacity. The CPU throughput must be reasonable. This means that only a relatively few cycles of the low level, implementing hardware should be needed to execute an instruction of the proposed architecture. To put it differently, the cost, in economic terms, of a computation using this architecture should not significantly differ from the cost of the same computation on more traditional architectures.

It must be a "structured programming" architecture. That is, it must support and encourage the use of the primary notions of structured programming - abstractions, both data and control abstractions. Such encouragement is important because it encourages those programming techniques and practices that lead to correct programs - one of the prime goals of the programming task.

As part of supporting structured programming the architecture must support data abstractions - those provided by the architecture as well as user defined data abstractions. There must be no distinction, to the user of data abstractions, between the use of built-in data abstractions and the use of user defined data abstractions; all provide objects of some abstract type with a specific set of operations on those objects. Such distinctions, if permitted, may lead to giving second class status to user defined data abstractions and would certainly reduce program generality since a program that accepts objects of unknown type would have to act differently based upon whether the object was of a builtin or user defined type.

---

2. An implementation may use storage for code of the implementation (especially in the case of an interpretive implementation) or for bookkeeping purposes (e.g. to remember the type of an object).

The data abstractions provided by the hardware must be strongly typed. That is, the operations allowed on an object created by a builtin data abstraction are restricted to those supplied by that abstraction. Moreover, that data abstraction may only operate on objects that it created. This is an important design goal for at least two reasons. First, this is the definition of data abstractions; it expresses all of the attributes that make something a data abstraction - anything less makes a sham of the notion of typed, architecture supplied objects. Second, it hides the implementation of the basic types. With this requirement the implementer of the architecture is free to choose any set of implementations for the basic types that he wants without regard for how the implementation of one type interacts with another type at the architectural level. Such considerations might be necessary if strong typing were not imposed and might make the designer's task more difficult.

The architecture must also allow the user to create his own data abstractions. Failure to allow user defined data abstractions can only stifle the use of data abstractions, whose use is generally accepted as important to good programming technique, by making their use difficult. The architecture must make no artificial constraints on the form of user defined data abstractions. That is, the architecture must not dictate right from wrong. Rather, it must gently point the way towards a good programming style[1] while at the same time giving the programmer the rope to hang himself if he so desires. If a designer is to impose restrictions of this sort, he must be very confident that the banned usage patterns are not useful. In a discipline as young as computer science, where the set of applications is constantly expanding, such confidence is at best ill founded.[2]

---

1. At least a style that this author feels is "good".
2. This is not to say that the imposition of restrictions for experimental purposes, to see if the world does fall apart, is inappropriate.

The architecture must support control abstractions as part of meeting the goals of structured programming. This means more than just providing a procedure call mechanism - procedures are just one example of control abstractions. One way of meeting this goal is to provide a large class of control abstractions and then hope that they are the right set. I, however, propose to design the architecture so that the user can build his own control abstractions, control abstractions of arbitrary form. Of course, some control abstractions will be provided by the architecture itself. Once again, the architecture must not dictate right from wrong, rather it should only point the way towards good programming styles. Thus the mechanism that allows control abstractions to be constructed should encourage "good" or "structured" control abstractions. This implies that this goal should probably not be met by the simple addition of conditional transfer instructions to the architecture. In fact the presence of a conditional transfer instruction will have to be carefully motivated if it is to appear at all since such an instruction tends to encourage "bad" programming styles[7].

The architecture should address issues normally associated with operating systems. The major reason for including such issues in the architecture is that the architecture itself can serve as an operating system; there will be little, if any, need to impose a layer of software between the programmer and the architecture. Moreover, if such a layer is ever needed this architecture should tend to make it small and relatively simple. When viewed as the intermediate language for compilers, an architecture with this feature will hide operating system peculiarities from the compiler. The important operating system issues that need to be included in the architecture seem to be protection, processes, the permanent (long term) storage of objects and the allocation of space and CPU resources.

• The architecture must be comprehensive and coherent. It must be a real system and not a toy system. Even though this architecture may be hidden from users by compilers, care must be taken to avoid creating an architecture lacking the expressive power needed to easily implement typical applications. The features must also form some coherent background - they must not look like a hodge-podge of features that were chosen just because each of them solves some particular problem that is felt to be important. These, again, are intangible goals and their satisfaction will be argued subjectively in the thesis.

Although the goal of efficiency has been alluded to previously, it should be stated as an explicit goal. It must be possible to implement the architecture in a reasonable manner. The amount of extra storage used by the implementation to keep track of the objects in the system must not be excessive. The instruction execution speed at the architectural level must be adequate for the user to perform his computations. This means that if the architecture is implemented in software (or microcode) then the computation needed to perform one instruction of the architecture must not be excessive. Similarly if the architecture is implemented in hardware (e.g. via digital logic and not via some software) the amount and cost of the hardware needed to implement the architecture must not be excessive. In considering these efficiency issues it must always be kept in mind that the architecture is doing much more than is done at an architectural level of typical computers.

Finally, in undertaking a design effort such as this it is important to decide on either a broad based investigation or an investigation that centers on some small set of issues. The proposed thesis will be a broad based one. This is really forced by the previous goal of having a comprehensive architecture. In particular, if some really difficult problem in designing the architecture should come up it will not be unreasonable to choose some solution, incomplete or not

completely satisfactory perhaps, and simply point out the limitations of the proposed solution and go on.

Proposed Solution

Some work has already been done in an attempt to design the architecture and meet the goals of the previous section. This preliminary solution was presented in another paper[13]. This section will outline some of the important aspects of the preliminary solution as it will form the basis for the solution presented in the thesis.

The most basic notion is that everything in the architecture is an object - there is no escaping the object oriented view of the world. Moreover, it is a strongly typed object oriented view in that only the type manager[1] for a type may manipulate objects of that type. This mechanism may not be circumvented in any way.[2]

Everything in the machine is an object including the most basic objects provided by the architecture. In particular, integers, booleans and characters are objects and are provided by the architecture. This points out the fact that the architecture must support very small objects. Efficient support for such small objects is an important problem that must be addressed in the thesis. In particular, issues such as space fragmentation (potentially both internal and external), low CPU overhead for object creation and low overhead (both in space and time) for space reclamation must be addressed.

---

1. A type manager is the piece of code implementing a data abstraction. In CLU it is called a cluster while in Alphard it is called a form.
2. Unless of course the data abstraction itself provides a means to do so.

Associated with every object is its type. The type of an object is the name of the type manager that provided the object (e.g. the type manager that created the object). Whenever an operation is to be performed on an object, the type manager for that object is called to perform the operation. Since everything in the system is an object, it follows that the only instruction needed in the architecture is a call-type-manager instruction. This works in all cases since everything, including procedures, type managers and collections of code are objects and can only be operated on by the appropriate type manager.

Another notion is that the basic naming mechanism of the processor is a capability one. That is, objects are named by unforgeable names called capabilities. The only way to refer to an object is if the user has been given a capability for that object. Capabilities do not contain access specifications in them, rather a capability gives full access to the object it refers to.[1] Access restriction is achieved in another manner. Similarly, capabilities do not contain type codes - every object is marked with its type.

It is necessary for objects to name other objects. For every object, except procedures that are being executed, an object refers to another object by containing a capability for that second object. Thus one object refers to another object by naming it and not by physically containing it. This allows an object to be shared among many other objects.

For executing procedures the mechanism is different since they have a clear need for a dynamic naming environment.[2] Associated with every executing procedure is an environment in which the names used by the procedure are interpreted. This dynamic environment is needed, at the least, for parameters and also is useful for containing the names of the objects used by the

---

1. However, this object may be an indirect object that restricts access to the eventual object
2. Such a need is not as clear for data objects.

particular invocation of the procedure. The environment of an executing procedure is divided into two parts - the local name space and the global name space.

To refer to objects, a procedure uses a name in its code and specifies that the name is to be interpreted relative to either the global name space or to the local name space. The name is an index into the specified name space. The indexed entry in the name space contains a capability for the object to be referenced. Thus to refer to the object X in figure 1, the currently executing procedure would use the name <local-name-space, 2> while to refer to the object Y the procedure would use the name <global-name-space, 3>. The association between an executing procedure and local and global name spaces is implicit from the act of calling a procedure and need not be further specified by the architecture.

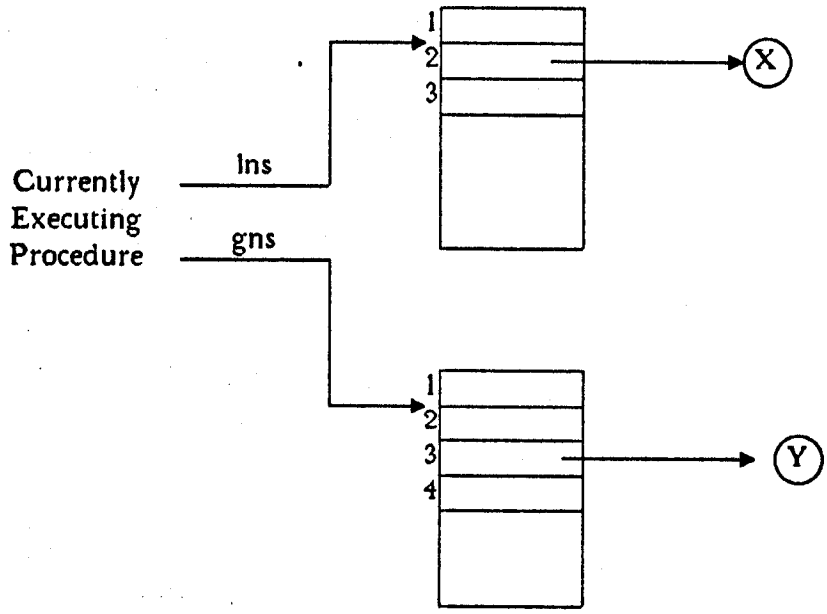

Figure 1. Example of a procedure referencing objects.

The local name space contains the names of (capabilities for) the objects local to this procedure activation - the parameters to this procedure activation, the names of objects that this particular procedure needs to know about (e.g. subroutines that it needs to call or static data that it needs to reference) as well as the the names of the objects created by this procedure activation.

The global name space is the global environment for a collection of procedure activations. It defines the virtual machine that the programs are to run in. It contains the names of global type managers (e.g. the names of the integer and boolean type managers) and the names of globally known objects (e.g. the name of the boolean objects true and false). It will also contain the names of whatever objects the user feels need to be globally known (e.g. the name of the PL1 runtime library). The intent is that the global name space stays the same for a sequence of procedure calls while a new local name space is created for each procedure activation.

This brings up the question of how to initialize the local name space of a newly called procedure. Associated with every procedure is a template local name space that is specified at the time the procedure is created. At procedure call time a new name space is created, the contents of the template name space are copied into the newly created name space and then the names of the parameters are copied into conventional places in the newly created name space. At this point the procedure begins execution in the environment provided by the current global name space and the newly created local name space.

A unique feature of the architecture is that type extension and access specification are provided by the same basic mechanism. The details of this mechanism are inappropriate for this document, however the basic idea can be presented. In both access specification and type extension the idea is to give the referencing object a capability that only allows it to see a particular view of the referenced object. In the case of type extension the object sees an object of the extended type

and cannot see "inside" of the object to its representation. In the case of access specification the referencing object sees an object on which some operations are not possible. For instance, if a bibliography is to be given to another object to refer to, that second object should only see the bibliography and the operations associated with bibliographies and not the fact that the bibliography is actually a collection of characters arranged in a particular way or some other more esoteric data structure. If this same bibliography is to be given to a program and that program is only to be allowed to look up references in the bibliography and not allowed to insert new references in the bibliography, then that program does not see a full fledged bibliography. Rather, it sees a bibliography on which the "add-ref" operation is not permitted. In both cases the object using the bibliography sees a different view of some underlying object. Thus the common mechanism provided by the architecture is an object-viewer mechanism. It permits the creator of the object-viewer to specify the view of the object that is "behind" the object-viewer (or window) that those looking through the object-viewer are to see.

The proposed architecture starts to meet the goals outlined in the previous section. It provides a coherent object oriented view of the world. The proposed thesis will continue the design of the architecture along the lines begun in the work just described in a continuing effort to meet the goals outlined earlier in this proposal.

Uniqueness of the Approach and Related work

This work has its background in three areas - structured programming, capability based machines and personal computers. In this section I will review some of the relevant literature and point out the ways in which the proposed research differs from work done by others.

The area of structured programming is a rich one. For our purposes it is only necessary to note that the fundamental ideas in structured programming are those of abstraction and top-down design.

Wirth[19] is the first writer to have formalized the notion of top-down design. Others [4,6,8] have discussed the need and form of structured programming. One of the aspects of this debate has been the undesirability of the go-to statement and the desirability of more structured control forms such as case statements, conditional loops and iterators. The proposed research is unique in that it attempts to provide a framework in which it is possible to develop all such control structures from a small collection of primitives in a manner that is consistent with the notions of top-down design and structured programming.

Another aspect of structured programming is the notion of data abstraction. Data abstractions have been around since the earliest programming languages. Simula, with its **class** mechanism, was the first language to allow the user to build his own data abstractions. It was deficient in that the language provided no mechanism by which the user could build strongly typed data abstractions - the abstraction mechanism could always be circumvented. CLU, Alphard and Euclid[11] are three recent programming languages that support strongly typed data abstractions. I have little to add to the notion of data abstraction as provided by these three languages. The architecture will be unique in that it will directly support the notion of data abstractions - both builtin and user defined.

The proposed research is similar to these language design efforts in that an environment to support structured programming is to be developed. It differs in that the architecture of a computer is to be developed. This means that issues such as protection and the permanent storage of objects, that can be ignored in language designs for the most part, must be addressed.

Furthermore it is essential that the set of uses of the architecture be as open ended as possible. This means that the architecture to be developed must provide a more extensible environment than languages such as the ones cited above. In particular, no aspect of the architecture must prevent the user of the architecture from building programs that contain features not anticipated by the architecture. The architecture must not attempt to dictate a programming style, only encourage one. This is an important and non-trivial difference between the design of a machine architecture and the design of a language.

Another area that this research is related to is the area of capability based machines since the proposed architecture is a capability based one. Capabilities were originally proposed in [5] and further elaborated in [9]. The proposed architecture uses capabilities that serve only as unique names objects and so in that respect differs from other efforts in this area such as Hydra[21], CAP[18] and CAL-TSS[10], all of which associated type and protection with capabilities. In all of these systems, the most basic object provided was a data segment composed of uninterpreted bits (essentially). Thus capabilities could name nothing smaller than a segment. This means that data types such as integers, real numbers and character strings, while supported by the architecture, were not treated as objects by the hardware - the goodwill of the programmer and compiler enforced constraints were all that protected them as data abstractions. The proposed research is also unique in that the notion of object is carried down to the bottom level of the system - everything is an object. By carrying the notion of object to the very bottom of the system, the small-object problem is made even more important and difficult to solve. The proliferation of small objects that the system must keep track of is immense. Lisp systems, see [1,14,17] for instance, have examined the issue of reclaiming storage when many small objects are present. Bishop[3] has proposed a mechanism to ease the problem of garbage collecting large amounts of storage containing many objects. Permanent storage of objects has not, however, been a major issue. The whole problem of

cataloging and keeping track of many small referencable objects is important, yet it has not been directly addressed in the literature. A preliminary solution to this aspect of the small object problem, based upon the ideas present in Bishop's thesis in this area, in the context of a computer system rather than in the context of a language, will be one of the results of this thesis.

There has been some work done in the area of language directed design of computer architectures. McKeeman[15] has argued rather forcibly for the need for computer architectures based upon language considerations. McMahan[16] in his PhD thesis has designed the architecture of a machine that is based heavily upon the ideas of Algol-68. This thesis differs from the previous work in two ways. First it has its language oriented roots in broad language design principles (e.g. structured programming) and not a specific language. It also differs in that the architecture to be designed in this proposed thesis addresses many issues normally associated with operating systems.

The proposed research will design a personal computer as opposed to a general purpose, shared computer. Current personal computer efforts have concentrated on providing the user with a classical von-Neumann computer at a reasonable cost. This research is unique in that an object-based personal computer is to be developed. Since it is a personal computer, many problems of building shared computers are avoided. Protection is a much less important issue in a personal computer since it is only necessary to provide facilities that protect the user from himself and not from other users. This should result in a much simpler protection mechanism for most uses while in those rare cases in which the user needs a more sophisticated protection mechanism a more expensive mechanism is available. The other important issue that can be avoided to a certain extent in a personal computer is resource allocation. Again, in a personal computer the goal is to protect the user from himself and not from other users. This should allow <u>some</u> resource allocation,

to be built into the architecture. In particular, a process mechanism can be provided since allocation of processor resources in a fair manner is not overly important.[1] The allocation of storage is also not overly important as the only goals are to provide a facility so that the user can protect himself from a runaway program - a program must not be allowed to use excessive amount of storage. In both of these last two issues efficiency of CPU usage is an issue. The user is not interested in a mechanism that results in inordinate CPU overhead. The thesis must address the issue of providing a sufficient mechanism at reasonable cost. The combination of these observations in a personal computer is unique and should produce an interesting design.

In summary this research has its origins in the areas of structured programming and personal computing. It is unique in the combination of these two areas into one research project. The need to provide a very extensible architecture is also unique. The approaches to be taken in providing protection and resource allocation control are also unique.

Thesis Plan

The thesis will consist of three distinct parts. The first part of the thesis will consist of a discussion of the actual architecture. This will comprise the major portion of the actual document since the description of an architecture is fairly difficult and a thorough understanding of the architecture is essential if the remainder of the thesis is to be understood. This area of the research is basically complete. Remaining issues center around I/O, control issues (including exception handling) and storage management.

---

1. This is not to say that the process mechanism should be so restrictive that the user is unable to perform scheduling of processes if he should need to.

The next two parts of the thesis will show that the architecture actually does separate implementation issues from higher level, language issues. The first aspect of this is showing that languages such as Clu and Alphard, that encourage the structured programming philosophy, can easily be implemented using this architecture. It is believed that this can be demonstrated with a few sample programs that exercise many of the language features.

Finally the thesis will show that the architecture is implementable. This part of the thesis should propose an implementation suitable for a personal computer. In particular a reasonable implementation of the basic types must be proposed, an efficient (both space and time-wise) space allocation procedure developed and an effective space reclamation procedure (e.g. garbage collection) proposed.

Although little research has been done in these last two areas, the architecture has been developed with a particular implementation in mind. Thus the completion of these areas should not be difficult.

The major remaining work of this thesis should be completed by April 1979. The remainder of spring 1979 and the 1979 summer term will be required to write up the results of this thesis. As no implementation of the ideas in this thesis are contemplated as part of the thesis, no major amounts of computer resources are anticipated although small amounts may be needed for experiments concerning some of the ideas in the thesis.

References

[1] Baker, H.G. Jr., "List Processing in Real Time on a Serial Computer", *Communications of the ACM 21*, 4 (April 1978), pp. 280-294.

[2] Birtwistle, G.M., *Simula Begin*, Auerbach Publishers Inc., Philadelphia, Pa., 1973.

[3] Bishop, P.B., "Computer Systems with a Very Large Address Space and Garbage Collection," M.I.T. Laboratory for Computer Science report TR-178, May 1977.

[4] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., *Structured Programming*, Academic Press, Inc., New York, New York, 1973.

[5] Dennis, J.B. and Van Horn, E.G., "Programming semantics for multiprogrammed computations," *Communications of the ACM 9*, 3 (March 1966), pp. 143-155.

[6] Dijkstra, E.W., "A Constructive Approach to the Problem of Program Correctness", *BIT*, 8 (1968), pp. 174-186.

[7] Dijkstra, E.W., "Go To Statement Considered Harmful", *Communications of the ACM 11*, 3 (March 1968), pp. 147-148.

[8] Dijkstra, E.W., "The Humble Programmer", *Communications of the ACM 15*, 10 (October 1972), pp.859-866.

[9] Fabry, R.S., "Capability-based addressing," *Communications of the ACM 17*, 7 (July 1974), pp. 403-412.

[10] Lampson, B. and Sturgis, H., "Reflections on an Operating System Design", *Communications of the ACM 19*, 5 (May 1976), pp. 251-265.

[11] Lampson, B., et. al., "Report on the Programming Language Euclid", *SIGPLAN Notices 12*, 2 (February 1977).

[12] Liskov; B.H., et al., "The CLU Reference Manual," CSG Memo * 161, M.I.T. Laboratory for Computer Science, July, 1978.

[13] Luniewski, Allen W., "AESOP: An Architecture for an Object Based Machine," MIT laboratory for Computer Science, Computer Systems Research Division, Request for Comments Nr. 165, July 22, 1978.

[14] McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Communications of the ACM 3*, 4 (April 1960), pp.184-195.

[15] McKeeman, W.M., "Language Directed Computer Design", *AFIPS Conference Proceedings*, 1967 Fall Joint Computer Conference, pp. 413-417.

[16] McMahan, Larry N., "Language Directed Computer Architecture", PhD Thesis, Rice University Department of Electrical Engineering, 1975.

[17] Steele, G.L. Jr., "Multiprocessing Compactifying Garbage Collection", *Communications of the ACM 18*, 9 (September 1975), pp. 495-508.

[18] Walker, R.D.H., "The Structure of a Well Protected Computer," Ph.D. Dissertation, University of Cambridge, England, December, 1973.

[19] Wirth, N., "Program Development by Stepwise Refinement", *Communications of the ACM 14*, 4 (April 1971), pp. 221-227.

[20] Wulf, W.A., "ALPHARD: Towards a language to support structured programming," Carnegie-Mellon University Dept. of Computer Science, April 1974.

[21] Wulf, W., et. al., "HYDRA: The Kernel of a Multiprocessor Operating System", *Communications of the ACM 17*, 6 (June 1974), pp. 337-345.