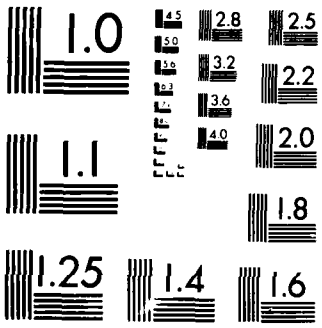
AD-A155 167 THE FORMAL SPECIFICATION OF AN ABSTRACT MACHINE: DESIGN AND IMPLEMENTATION(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA J M YURCHAK DEC 84 1/2

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(2)

AD-A155 167

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DTIC
ELECTE
JUN 18 1985
S D
B

THESIS

THE FORMAL SPECIFICATION OF AN ABSTRACT MACHINE:
DESIGN AND IMPLEMENTATION

by

John Marshall Yurchak
December 1984

Thesis Advisor: Daniel Davis

DTIC FILE COPY

Approved for public release; distribution is unlimited

85 5 22 033

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Formal Specification of an Abstract Machine: Design and Implementation		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John Marshall Yurchak		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE December 1984
		13. NUMBER OF PAGES 151
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

abstraction, formal specifications, algebraic semantics, abstract machine, software probability, formal specifications

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The high cost of porting software from one machine to another stems from the ad hoc way in which the programmer's problem solving abstraction interacts with the machine's physical resource abstraction. If this interaction could be formalized, the well known semantic gap would at least be better understood, if not narrowed significantly. In this study, we apply techniques borrowed from contemporary (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

research in abstract data type specification to design, specify, and implement the physical resources of an abstract machine called AM.

Accession For

Availability Codes

Avail and/or

Dist Special

A-1

Approved for public release, distribution unlimited

The Formal Specification of an Abstract Machine: Design and Implementation

by

John Marshall Yurchak
Lieutenant, United States Navy
B.A., Lycoming College, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

Author: John M. Yurchak

Approved by: Daniel Davis, Thesis Advisor

Bruce J. MacLennan, Second Reader

Bruce J. MacLennan, Chairman,
Department of Computer Science

Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

The high cost of porting software from one machine to another stems from the *ad hoc* way in which the programmer's problem solving abstraction interacts with the machine's physical resource abstraction. If this interaction could be formalized, the well known *semantic gap* would at least be better understood, if not narrowed significantly. In this study, we ~~apply~~ techniques borrowed from contemporary research in abstract data type specification to design, specify and implement the physical resources of an abstract machine called AM.

TABLE OF CONTENTS

I. INTRODUCTION 7
A. THE PORTABILITY PROBLEM 7
1. Abstraction 8
2. The Semantic Gap 8
B. THREE WAYS TO NARROW THE SEMANTIC GAP 9
1. Formalism 9
2. Representation Independence 9
3. Intent Expressive Resource Abstraction 10
C. METHODOLOGY 10
II. THEORY 13
A. ABSTRACT DATA TYPES 13
B. ALGEBRAS 14
C. ALGEBRAIC SPECIFICATIONS 15
D. TERM, INITIAL AND FINAL ALGEBRAS 17
E. ALGEBRAIC SEMANTICS 19
III. ISSUES 23
A. FIRST PRINCIPLES 23
1. Assumptions 23
2. Notation and Syntax 23
3. The Limitations of Algebraic Specifications 23
4. Finiteness 24
5. Parameters 25
6. Errors 25
7. Proving Correctness 25
B. HIGH LEVEL LANGUAGES 26
1. Implementation and the Semantic Gap 26
2. The Chicken and the Egg 26
C. THE PHYSICAL RESOURCE 27
1. The Instruction Set 27
2. Overlap 28
D. SPECIFICATIONS 29
1. Notation, Syntax and Semantics 29
2. Parameterized Specifications 31
3. Extension 32
IV. DESIGN 34
A. THE SPECIFICATION LANGUAGE 34

1. The Macro Preprocessor 34
B. THE MACHINE 35
C. THE SPEC 38
1. Macro Definitions 38
2. The Atomic Types 39
3. Machine Primitives 42
4. State 44
5. Execution 45
6. Remarks 46
V. IMPLEMENTATION 49
A. IMPLEMENTING DATA TYPES 49
B. MAPPING OPERATORS TO FUNCTIONS 53
C. ERROR HANDLING 54
D. EXECUTION 54
E. OBSERVATIONS 58
VI. CONCLUSIONS 60
A. FUTURE WORK 60
APPENDIX A: A GRAMMAR FOR ALGEBRAIC SPECIFICATIONS 61
APPENDIX B: THE SPECIFICATION FOR AM 64
APPENDIX C: A SIMPLE ASSEMBLER FOR AM 101
LIST OF REFERENCES 149
INITIAL DISTRIBUTION LIST 151

I. INTRODUCTION

We address the problem of formalizing the relationship between hardware and software resources by demonstrating a practical methodology for precisely specifying the manner in which the two may interact. After a brief statement of the problem and related topics, we discuss the theory behind our research and some of the issues affecting our results. Finally, we describe in some detail the manifestation of our efforts, a specification and implementation of an abstract machines we call AM.

A. THE PORTABILITY PROBLEM

It is well known that porting large programs from one machine to another is an expensive ordeal. It is also well known that once the software has been moved to the new machine, it is anybody's guess whether or not it will work as before[1]. Even if our program seems to work, we may find it consumes more resources than we expected. Indeed, this may be just as bad as if it did not work at all.

There are a number of reasons why the portability problem is getting worse, not better:

- Most architectures, even those which profess to be "language directed", reflect a bias toward making the machine look like what the programmer wants, or toward some engineering goal, such as maximizing the number of devices.
- Both languages and machines are related to the data they manipulate in an implementation dependent way.
- Language and hardware designers pursue their conflicting goals to the detriment of the poor compiler writer, who, with imprecise tools and methodologies is faced with the job of implementing ambiguous semantics on an informally designed resource.

Although these and other factors do adversely contribute to the imperfect task of moving software from one machine to another, they add their weight to other difficult issues in language design, computer architecture and software

[1]We assume, probably unjustifiably, that it worked correctly before we tried to move it.

engineering. This study confines itself to treating the issues surrounding the interaction between the programmer's view of the world as a problem, and the architect's view of the world as a resource.

1. Abstraction

Abstraction describes the separation of the defining properties of an object from other, unnecessary details about it. A programmer is primarily concerned with solving a problem. Appropriately, the tools at his disposal, programming languages, development aids, the programming environment, form a *problem solving abstraction.* The hardware (and some of the software) on which this problem solving abstraction is implemented, however, is an abstraction of a different sort. Addresses, registers, ports, most of the operating system service routines, all provide more or less efficient ways to manipulate the physical resources of the machine -- they form a *physical resource abstraction.*

The fuzzy area between these two abstractions, sometimes simplistically perceived as the boundary between hardware and software, exposes a number of shortcomings in language design and computer architecture collectively termed the *semantic gap.*

2. The Semantic Gap

The semantic gap manifests itself anywhere a problem solving abstraction touches a physical resource abstraction. A detailed description may be found in Myers (1982). He observes that the semantic gap contributes to the cost of software development, software unreliability, inefficiency, complexity, and the distortion of programming languages. Certainly no single development or methodology will eliminate this problem.

Narrowing the semantic gap requires significant changes in the fundaments of computer architecture and language design. We chose to concentrate on three factors which significantly contribute to this problem:

- Informally described semantics.
- Representation dependent data types.
- Arbitrarily designed instruction set architectures.

The implication, of course, is that through increased formalism, the introduction of representation independent data, and a more thoughtful treatment of the

instruction set, the semantic gap can be narrowed. The balance of this thesis is devoted to describing a methodology for doing just that.

B. THREE WAYS TO NARROW THE SEMANTIC GAP

1. Formalism

The benefits of formalism in the design process have been amply revealed in countless articles treating this issue from the standpoint of software engineering. Our concern will be limited to formalism as it applies to the specification of an abstraction. Various specification methodologies exist, many of which have been used with more or less success in projects of practical significance. But we caution the reader that by "formal" we mean a mathematical rigour rooted in proven theory. The idea of formalism as often applied to software engineering will not do here. A *formal specification* is a complete description of the meaning of an object. It forms the basis for an abstraction and is ultimately a bridge over the semantic gap.

The benefits of formalism in which we are most interested are:

- It provides a firm basis for proving our assertions about a specification and its implementation.
- It encourages a discipline on the part of the designer to be rigorously precise.
- It compels us to find ways of describing things which are representation (implementation) independent.

2. Representation Independence

Conventional machines force us, as programmers, to develop our own abstractions of data. At a time when we are most concerned with developing clean algorithms the architecture obligates us to worry about status registers and word length. Certainly someone must ultimately deal with these physical properties of the hardware, but this should not fall as an *obligation* upon programmer. The programmer should be free to ignore unnecessary detail.

We will attempt to minimize the dependence of data upon its representation through the use of *abstract data types*. Our notion of data is very general and includes, for example, program instructions.

3. Intent Expressive Resource Abstraction

Conventional architectures do not permit us to unambiguously express our intent in a program. Artificial data types combined with typical resource models force ambiguity and the overloading of data structures. Stack frames are a good example of this. The semantics of the frame combine those of an array and those of a stack. Meanwhile, the whole thing is implemented in memory, with the data types overlayed on an array of fixed length cells.

We claim that applying methods similar to those used to describe abstract data types, we can describe an abstraction of the physical resources of a machine which benefits not only from the formalism used to specify it, but also permits the implementor to clearly interpret the intent of programs written for it.

C. METHODOLOGY

The goal of this research is to contribute something of practical significance to the study of software portability by treating an area which has been largely ignored -- the design of a formal abstraction for the machine itself. We have innumerable high level programming languages, programming environments, graphics languages, database machines, file systems, operating system command interpreters, a whole host of different abstractions tailored to the task of providing us with just enough information to do everything we need to do, and nothing more. So why, then, have we failed to develop abstractions for the hardware resources, upon which we are so dependent, which are more than just a collection of registers, opcodes and some arbitrary rules about how they interact. A more difficult but certainly more important task than actually defining the abstraction is developing a methodology for producing more.

Our method has been to take a naive approach toward all areas of the design and implementaion process not directly related to the specification itself. We do this for two reasons. First, we can take for granted the large body of research in programming languages and computer architecture -- we are designing neither a language nor a processor, even though *ad hoc* examples were required to complete the implementation. Second, the research is intended to benefit programmers. Since it is unreasonable to expect those who may use this method to understand the theory behind the specification, the key to understanding the reasons for our

design decisions lies in the way we coded it. Thus, cleverness has been eschewed in favor of clarity.

Our task in this thesis, then, is to examine a wide range of issues which impinge on the process of designing and implementing the specification of a machine, and then to describe how we went about actually doing it.

Another area where we run into trouble is the number of terms induced by the specification. If the number of terms is infinite, then the specification is not even representable, let alone computable. This problem can easily creep into a specification unnoticed.

5. Parameters

To reduce the enormous complexity of some types of specifications, designers have turned to the use of *parameterized specifications*. The basic idea is to write a specification whose signature is a template for other specifications. A typical example would be the type string. We might want strings of other objects besides characters. Rather than duplicate what is essentially the same specification, it is *parameterized* and instantiated when needed in the specification. The need for parameterized specifications goes beyond a simple savings in the effort of writing a specification. Some objects are simply not describable without them.

Parameterized specifications are still not well understood. Most of the underlying theory concerning them is under debate. We have minimized our use of them, as a result.

6. Errors

What to do when objects which are not members of the right carrier sets find their way into operations is a real problem, as is the "propagation" of errors throughout the specification. A number of ways of handling this have been proposed, most notably by Goguen (1978), but without much success. We have found that in implementation this is not a problem, however, and have generally taken the point of view that the most important thing to specify is *where* errors are explicitly detected rather than what to do about them once they are.

7. Proving Correctness

We have not mentioned provability except to say that a formal design methodology tends to support formal methods of proof. It is beyond the scope of this study to treat this issue in detail, but we will return to it in the discussion of our implementation.

Unfortunately, there are many accepted features of conventional processors which are extemely difficult if not impossible to describe with algebraic specifications. Take, for example, the typical primitive of all machine types -- the bit. The reader is encouraged to attempt to write a semantic description of two's complement arithmetic, or operation of a status register. True, one of the goals we have stated is representation independence. But for this we give up the *freedom* to design anything we might conceive.

Another important limitation is the difficulty with modality. How does one specify *when* an operation is to occur. In operators whose arity is greater than one, the arguments are assumed to "arrive" simultaneously, and side effects are not allowed. A number of techniques have been suggested as to how timing might be formally expressed (Giegerich 1983) but we use only the simplest. Modality is expressed in terms of parametric dependencies.

prog(a,q) = xeq(atomofinstr(fetchm(a,q),a,q));

In the above example, the operator **fetchm** is applied "before" **atomofinstr**, which is applied before **xeq**.

4. Finiteness

No matter what we describe with a formal specification, any *implementation* of the specification must be finite. Hence a problem: how do we describe a finite limitation of an infinite set of objects. Consider again Figure 3.1. It specifies a type with a countable infinity of objects. But we have no machines to represent an infinity of numbers. The problem does not stop there. A specification for the natural data type, which will look much the same as the one for integer, will also describe a countable infinity of objects. In a world accustomed to twice as many signed as unsigned integers, this will come as a great shock!

Obviously, any actual machine *will* be finite. And although the problem we have just described may seem more metaphysical than physical, it forces us to realize that we will never be able to fully implement a specification. More important, it also requires us to deal with boundary conditions in ways which may not preserve our methodology.

A. FIRST PRINCIPLES

Any formal specification **must** begin with some assumptions, "first principles", upon which the whole methodology is based. We have already discussed the mathematical basis for our specification method. Let us look now at its implications.

1. Assumptions

At some point we will have to describe the operations our abstract machine will perform. Some of these operations can be defined explicitly. Most, however, will be defined in terms of certain primitives, the meanings of which we simply take for granted. For example, it is probably a good idea for AM to be able to perform integer arithmetic. So, we specify the integer data type. Figure 3.1 gives us just about everything we need, except for one thing. The specification does not describe, nor should it describe, the way in which strings from an alphabet may be uniquely associated to the elements of the type. As written, the specification obligates us to refer to the "number" 5 as

succint(succint(succint(succint(succint(zeroint)))))

Not very convenient.

Thus, we consciously limit the scope of our formalism in the interest of practicality. Our use of a formal specification is intended to improve our understanding of the meaning of a physical resource, not elementary number theory.

2. Notation and Syntax

Although the notation ultimately used to express the specification is arbitrary, the need for automatic parsing means the usual syntactic and semantic considerations facing the designers of any programming language are before us as well. In addition, since, as we will show, the specification of anything useful is likely to be complex, we may also need to choose notation which supports automatic program generation, or at least macro processing.

3. The Limitations of Algebraic Specifications

Once we are commited to a formal design methodology, it will be difficult to justify departures from the method. This means we have limited ourselves to designing objects which can be describe with the semantics we have defined.

III. ISSUES

The elegance with which algebraic specifications solve the mechanical problem of specifying an abstraction does nothing to help solve a whole host of other problems affecting our design. In fact, the use of a formal methodology has imposed constraints which would not normally affect more conventional architectures. We treat these and other issues now.

```
spec integer
is
        extend
                boolean
        with
                sort
                        int;
                op
                        predint: int → int;
                        succint: int → int;
                        sumint: int,int → int;
                        zeroint: → int;
                        eqint: int,int → bool;
                        gtint: int,int → bool;
                axiom
                        predint(succint n) = n;
                        comutative(sumint,int);
                        associative(sumint,int);
                        sumint(n,zeroint) = n;
                        sumint(n,succint m) = succint(sumint(n,m));
                        equivrel(eqint,int);
                        irreflexive(gtint,int);
                        transitive(gtint,int);
        end extend;
end integer;
```

Figure 3.1: A Spec for the Abstract Type **Integer**

- We can prove a Turing machine is describable by these algebras. Therefore, we can describe a computer with these algebras.

final quotient algebra, then we say the candidate describes the "final algebra semantics" of the specification. If the congruence matches neither the initial nor the final quotient algebra, then the candidate does not describe the meaning of the specification.

Now that we know how to determine whether or not our specification describes something "real", the next step is to show that the object(s) it describes has the properties we intended. Rather than discuss the details, we direct the reader again to Goguen (1978) and Davis (1984), and instead list some of the key results of this theory below:

- A specification is an abstraction of a concrete object. It forms a *template* for an algebra which must ultimately describe the *meaning* of the specification.
- A *term algebra* contains only those terms composed of operators without free variables. A *free algebra* permits terms with free variables.
- The axioms of a specification are really equations between terms in a free algebra.
- The *initial algebra semantics* of a specification is defined by the class of algebras whose signature is given in the specification, with the property that any two formal terms are provably equal from the axioms if and only if the corresponding expressions in the algebra of that class evaluate to the same constant.
- The *final algebra semantics* of a specification is defined by the class of algebras whose signature is given in the specification, with the property that any two formal terms are consistent with the axioms if and only if the corresponding expressions evaluate to the same constant.
- Any two final or any two initial algebras for a specification are isomorphic.
- The object a specification specifies is computable if and only if the class of initial algebras and the class of final algebras are the same. Likewise, any time one can show all formal terms reduce to a 0-ary term (a constant), then the initial and final algebraic semantics must be the same.
- An algebra is *effectively computable* when its signature matches that of the specification, its carrier sets are enumerable, and the operations defined by its operators can be described using algorithms.
- Any time one can show the class of initial and final algebras is not the same, there exists at least one algebra which is not effectively computable.
- Any two specifications which can be shown to produce the same class of algebras are equivalent (semantically).

One final result of this theory forms a key part of the foundation for believing it possible to describe an abstract machine with algebraic specifications:

The second congruence is such that two terms, t and t' are congruent if and only if the equation $t = t'$ is *consistent* with the axioms. An equation is consistent if, by adding it to the set of axioms, the resulting set of terms is not *trivial*. A set of terms is considered trivial when, for all terms, any two terms of the same sort are provably equal.

A correspondence H associating the carrier sets and operators of one algebra A to the carrier sets and operators of another algebra B can be shown to be a homomorphism, provided:

- The correspondence between carrier sets preserves the sort type.
- The correspondence between operators preserves the characteristic.
- The standard property of homomorphisms holds:

$$H(o(t_1, t_2, \ldots, t_n)) = H(o)(H(t_1), H(t_2), \ldots, H(t_n))$$

where each t_i is an element of sort s_i in A.

There is a canonical homomorphism from a term algebra to an algebra A, with the same signature, which can be determined by evaluating each formal term in the term algebra through its corresponding term in A.

E. ALGEBRAIC SEMANTICS

We learn to ascribe meaning to abstraction by associating with that abstraction concrete objects. In some sense, we "know" the meaning of concepts like *table* and *tree* when we can recognize the class of objects captured by those concepts[2]. In the "world" of algebraic specifications, the concrete objects are the algebras. They form the manifestation of the *meaning* of a specification. As we have said, a specification induces three congruences: a congruence defined by the algebra on the evaluated terms, a congruence on the initial quotient algebra and a congruence on the final quotient algebra. We can determine whether or not a candidate algebra captures the meaning of a specification by examining the properties of the congruence it induces. If the congruence is identical with the initial quotient algebra, then we say the candidate describes the "initial algebra semantics" of the specification. Likewise, if the congruence is identical with the

[2] The author would be happy to discuss the epistemic implications of this statement with the philosophically minded reader some other time.

the the set of terms forms a language on the alphabet obeying the following grammar:

- For each sort s in S add the production

$$<T_S> \rightarrow <T_{S_s}>$$

  where T_S is the set of all terms which can be created from the signature which contain no free variables and T_{S_s} is the set of all terms in T_S of sort s. Note that T_s and T_{S_s} are both term algebras.

- For each operator of characteristic

$$o : s_1, s_2, \ldots, s_n \rightarrow s$$

  add the production

$$<T_{S_s}> \rightarrow \text{'op('} <T_{S_{s_1}}> \text{','} \ldots \text{','} <T_{S_{s_n}}> \text{')'}$$

  where 'op' is a name uniquely associated to o.

- For each free variable X of sort s, add the production

$$<T_{S_s}> \rightarrow \text{'}X\text{'}$$

The reader will note that the grammar just described is LL(1), and thus can be parsed very efficiently, particularly by automatically generated parsers. This is the theoretical basis for the methodology described in Guttag (1978a).

Now, the set of axioms induces two canonical congruences on T_S, which in turn induce two quotient algebras on T_S, called the *initial quotient algebra* and the *final quotient algebra.* The first congruence is such that two terms, t and t' are congruent if and only if the assertion $t = t'$ can be proven from the axioms. The following rules apply (Davis 1984):

- Any axiom is a proven equation. Any conditional axiom is a valid rule of inference for proving equations from proven equations.
- If, in a proven equation, every occurence of a free variable is replaced by a single term of the same sort, the resulting equation is proven.
- If, in an equation, some term is replaced by a term which is provably equivalent, the resulting equation is proven.
- Any equation derived from proven equations using the reflexive, symmetric or transitive laws for equality is proven.

From these it can be shown that the relation defined by all pairs of provably equal terms is a congruence.

boolean type, i.e.,the set {t,f}. and if, using conventional notation. we associate the operators in the specification for *bool* with some representative operators

true	t
false	f
and	&
not	~

then the equations

$$\sim(\&(x,\&(t,y)) = \sim(\&(x,y))$$
$$\&(\sim x,\&(y,f)) = f$$

both satisfy the axioms of the specification. Important to note, however, is that the symbols we choose to associate with the operators in a specification are completely arbitrary. If we instead make the following associations

bool	{a}	
true	→a	
false	→a	(constant ops returning 'a')
and	(x,y)→a	(trivial binary op)
not	(x)→a	(trivial unary op)

it can be easily shown that this algebra also satisfies the axioms, but we would not admit that it is representative of the boolean type. Thus, we make a distinction between an algebraic specification and an algebra. What then is the *meaning* of a specification? It is the class of algebras which is *uniquely* associated to that specification. The precise nature of this association will be discussed in Section E.

D. TERM, INITIAL AND FINAL ALGEBRAS

Given a specification (S,E), with signature S and axioms E, our next problem is to show that there are indeed algebras with that signature which satisfy the axioms. Using the specification for boolean as an example, the *term algebra* is the set of all term expressions which can be constructed without violating the characteristic of an operator in the specification. This set of terms is obtained using a technique know as the Herbrand construction.

If we view terms as strings on an alphabet of operator names, some useful punctuation symbols and a finite set of symbols for the names of free variables.

any others we might need. **Not** and **and** are those tranditionally encountered in computer hardware. Thus, so far we have:

```
sort
        bool
ops
        true: → bool
        false: → bool
        not: bool → bool
        and: bool,bool → bool
```

Now we need some axioms to describe the *semantics* of the operators on values of type **bool**.

```
not(true) = false
not(not(b)) = b
and(true,b) = b
and(false,b) = false
and(b1,b2) = and(b2,b1)
and(and(b1,b2),b3) = and(b1,and(b2,b3))
```

These axioms explicitly describe all the essential properties of the operators. Notice that no explicit mention is made of the possible composition of a set of boolean values. We know **true** and **false** must be in it, but they may not be unique.

The syntax used in Figure 2.1 was chosen to permit automatic compilation. To date, no such compiler has been written, but a syntax directed editor operating on a similar syntax is available (Lilly 1984).

Algebraic specifications are composed of a *signature*, which includes the operators and sorts, and *axioms*. Axioms are simply equations between *terms* (expressions) made up of operators and/or free variables from the specification. Axioms may be conditional. A specification is thus a pair (S,E) where S is the signature and E is a set of axioms.

An algebra's signature matches the specification if there is a one to one correspondence between the sorts and operators of the specification and the carriers and operators of the algebra, and if the operations on elements of the carriers are consistent with the semantics specified by the axioms in the specification. If we associate the name *bool* to what we normally accept as the

and result are known as the *characteristic* of the operator. A sort is an index into a set of carriers. The carrier indexed by the sort represents the data type, while the sort represents the "name" of that type, like "integer", "boolean" or "character". Each operator is thus an explicitly defined function which accepts zero or more typed arguments and returns a typed result. The type of each argument may differ from every other argument as well as from the result type.

Any usefully complex data type requires the use of free variables. Their presence introduces the possibility that the type we specify may not by finitely describable. Note, "finitely describable" does not mean "descibes a finite number of objects". It means the description is itself finite, i.e., the number of operators and the number of axioms is finite. It is sometimes difficult to find a finite description of a type, since many mathematical and logical operations assume a non-finite application. Simple iteration is an example of this.

Since we hope to describe something in a way which is representation independent, we must be certain we introduce no representational bias into the specification. If we ultimately hope to show how to use these methods to describe hardware in a representation independent way, we certainly do not want to require the use of a particular architecture unless we can demonstrate its generality.

C. ALGEBRAIC SPECIFICATIONS

A specification is a *template* for the sets and operators in the algebra. The semantics of the type are specified using *axioms*, which are provable equations constructed from operators and free variables. The template makes no assumptions about the elements of the sets in the algebra, or about how operators are applied to manipulate the elements. This information is furnished by the axioms. Let us now return to Figure 2.1.

In the case of the specification for boolean, we require a single set to hold the values of the type. Call it **bool**. Next, we specify two *constants*[1] to represent the two possible values of any object in the type. Call them **true** and **false**. Then we select the smallest set of operators from which we know we can derive

[1] Really 0-ary operators.

This leads us to a number of important facts about abstract data types:

- The *meaning* of an abstract type specification does not depend upon the notation used to express it.
- A type may imply other operators than those mentioned explicitly in the specification.
- Two "equivalent" data types may not only "look" different, but may be specified using fundamentally different operators and axioms.

Proving two abstract types to be equivalent is not trivial. To do this we must be able to show that the set of possible expressions built using the operators from one specification is in some way "equivalent" to the set produced with operators from the other.

An expression is constructed from the operators and values of the type using composition of operators. Here, using prefix notation, are two correctly formed boolean expressions:

not(and(true,false))
and(true,and(false,not(true)))

Note, they do not contain variables, but are constructed solely from operators. Below is an expression with *free variables*:

and(not(x),or(true,y))

Evaluating an expression is straightforward, but the presence of free variables makes it much more difficult to prove assertions about the specification of the type.

The notation we used in Figure 2.1 is not arbitrary. It reflects the fundamental theory upon which our method of specification is based.

B. ALGEBRAS

An algebra is an aggregate of operators and sets. The sets describe argument and result types for each operator, while the operators define the ways in which arguments may be manipulated to form results. The general form of an operator is:

$$o : a_1, a_2, a_3, \ldots, a_m \rightarrow a_n$$

where each a_i is a *carrier* set of *sort* s_i. The set and arrangement of arguments

```
spec boolean
is
        sort
                bool;

        primitive
        op
                true: → bool;
                false: → bool;
                not: → bool;
                and: bool,bool → bool;

        axiom
                false = not true;
                not(not(b)) = b;
                and(true,b) = b;
                and(false,b) = false;
                and(b1,b2) = and(b2,b1);
                and(and(b1,b2),b3) = and(b1,and(b2,b3));
end boolean;
```

Figure 2.1: A Spec for the Abstract Type **Boolean**

permits us to specify both problem solving and physical resource abstractions with equal facility. Hence, we begin with the specification of data types, not because this is the *best* place to start, but rather because we can refer to the large body of research on the subject.

A typical example of an abstract data type is boolean (Figure 2.1). The reader should have no trouble seeing that all the classical rules of inference, as well as the operators **or** and **implies** may be derived from the given axioms and primitive operators. In this sense, Figure 2.1 represents a minimal specification of the type **boolean**. A student of logic will note here that several other combinations of primitive operators will permit all the others to be derived. We further note that the names chosen for the operators, indeed for the type itself, are purely arbitrary.

II. THEORY

There are many ways to write a specification. A high level language is an example of a specification with more or less ambiguous semantics. To achieve true portability, we must be able to demonstrate the following properties in our implementation:

- The specified semantics *actually implemented* on the source machine are completely unambiguous.
- The implementation on the source machine is "correct".

Thus, our method of specification must be formal enough to permit proofs of correctness. Although the knowledgable reader will know that the provability of usefully complex specifications has so far been unrealized, research conducted in parallel with this study (Griffin 1984) has given us reason to be optimistic.

The requirement for unambiguous precision and provability leads us naturally to a mathematical basis for our specification. Here we find a significant body of research already in place in the area of abstract data type specification. Goguen (1978) and Guttag (1978) treat this topic in great detail. We will not do so here. Instead we give an overview of the important concepts of abstract data types and the underlying theory of specifications as a preface to a treatment of the issues. The following discussion is a paraphrase of Davis (1984). The reader is directed to that paper for more details.

A. ABSTRACT DATA TYPES

A data type is a class of objects together with a set of operations which may be performed on those objects. An *abstract* data type is a precise description of a class of objects in terms of the *semantics* of the operations which may be performed on the class. Our reasons for considering abstract data types are twofold. First, in any specification of a hardware resource, some mention of the data types it manipulates must be made. Second, although an abstract data type is primarily a problem solving resource, it may imply a physical resource as well. A stack is a good example of this. Our goal is to arrive at a technique which

B. HIGH LEVEL LANGUAGES

The vast majority of large software systems are written in a high level language. Dialectal variations between languages notwithstanding, the problem of porting software from one hardware environment to another does not involve mapping one high level abstraction to another. It is much more complex. It involves the translation of a *relation* between the semantics of a language and its implementation on one machine into a similar but not identical relation on another machine. The properties of this relation form part of the semantic gap.

1. Implementation and the Semantic Gap

Implementing a problem solving abstraction on conventional machines is somewhat haphazard due in part to difficulties in mapping the semantics of the language onto the semantics of the hardware. It is unfortunately true that there is often no way at all to describe the meaning of a problem solving abstraction in terms of the physical resource. This occurs because language designers do not want to acknowledge the existence of the resource provided by the engineers and because engineers do not see the physical resource as part of a higher level abstraction.

In general, when we push a physical resource abstraction up to meet a problem solving abstraction, the class of languages which may be efficiently implemented on that hardware is narrowed. Likewise, when problem solving abstraction is pulled down toward the physical resource, we lose the ability to elegantly map our problems into a program.

The task of implementing a language should acknowledge the semantic gap, not contribute to it. We must therefore find ways to describe the precise relationship between the language semantics and the resource. Our methodology should make this easier.

2. The Chicken and the Egg

One problem we will always have to deal with is where to start. Do we begin by defining a general purpose problem solving abstraction, or by defining the resource? We have chosen the latter for several reasons. First, we have come to realize that it is easy to dream up problem solving abstractions which are simply unimplementable. Ada may be a prime example of this. Second, an

understanding of the fundamental characteristics of the resource reveals much more about the relationship between a language and its implementation than an understanding of the semantics of the language. Third, we have gotten pretty good at describing language abstractions, but have only scratched the surface at trying to formally describe a physical resource. Thus, we devote our efforts to describing an abstract machine.

C. THE PHYSICAL RESOURCE

The ideas behind the concept of a *memory* or a *display* are not really well understood. We know they are complex physical structures, but we have great difficulty formalizing what it means to fetch or store values. The primary cause of this difficulty is the design process itself.

The hardware design process is a battle against the clock, against rising complexity, against shrinking space, where opportunity is expediency, and unused space is a crime. That must change if the semantic gap is to be narrowed significantly.

Complex components imply complex semantics. Complex semantics imply even more complex specifications. The conventional design goals of minimizing circuit complexity, and of maximizing the regularity and orthogonality of the instruction set architecture do not really address the issue of semantic complexity. If our goal is to increase software portability, a way must be found to coalesce the many conflicting considerations affecting the design process. Admittedly, the hardware-software relationship is only one of these considerations.

1. The Instruction Set

An important question one might ask is, what effect does the specification methodology have upon the design of the instruction set architecture? If we are prevented from describing the instructions we need for an application, the whole method loses much of its appeal. Interestingly, we have found that the content of the instruction set is much more related to the types of data we are able to describe, rather than to the method of algebraic specification. Representation independence renders meaningless instructions like shift and rotate because the level of abstraction is necessarily higher. In essence, the

physical resource is made to look more like a primitive problem solving abstraction.

We found also that the typical issues facing the designer of an instruction set, such as timing, opcode size, addressing modes, and such, tend to become moot. However, consideration of regularity and orthogonality -- programming language issues -- remain as important as ever. This reinforces the observation that emphasis upon the resource *as* a resource is considerably diminished when the machine is modeled as an abstraction.

2. Overlap

The term *overlap* is used here in reference to the manner in which machine data types are overlaid upon a common resource, the memory. Overlap occurs in conventional machines because data structures are overloaded to prevent the waste of valuable computer resources. For a typical word size of 32 bits, the practice is to assign the character type to a byte (8 bits), short integers to a half word, long integers to a word, and standard and extended precision floating point numbers to a word and double word respectively. Without even mentioning the problem of alignment[1] it should be clear to even the casual reader that describing the semantics of such a memory system would be a mess. We borrow from Giegerich (1983) to illustrate this point. We define a function *overlap* which accepts two cell identifiers and returns true when overlap exists between the cells, false if not. If we assume even alignment for 16-bit words and 32-bit longwords, then the overlap axiom relating 16 and 32-bit words would be

$$overlap\,(M\,16\;a\;,M\,32\;b\;) = (b \leqslant a \leqslant b+2)$$

where a and b are addresses, and, of course, *overlap* is commutative. Now, imagine having to specify a set of overlap axioms relating each data type to every other data type, and then having to specify them *everywhere* they applied to axioms throughout the specification! What makes this even worse is it can be shown that, for certain configurations of memory, there may be an infinity of such axioms. Therefore, we avoid overlap.

[1] Many machines require types larger than one byte to be aligned on an even address.

D. SPECIFICATIONS

Algebraic specifications impose restrictions upon the class of objects we can describe. Although a benefit from this is that it forces us to think very carefully about the objects we are attempting to specify, it is important not to allow the methodology to restrict our thinking. That this can easily happen has been demonstrated over and over again with programming languages. Experienced programmers are masters of idiom. But mastering the "tricks" of particular specification language should not be considered a goal.

1. Notation, Syntax and Semantics

Although the notation is theoretically arbitrary, the design of a specification language is at least as difficult as designing a programming language, perhaps more so. Abstract algebra already has a body of accepted notation, and familiarity with it tends to bias one's ideas about how to go about designing a language. Some of the key points to remember are:

- The grammar/syntax should support automated parsing.
- The language should not make it easy for the writer to specify things which cannot possibly describe physical objects (such as an object with an infinite number of terms).
- The language should be human readable since anything usefully complex will be difficult enough to understand without requiring the reader to wade through syntax to determine the meaning of a specification.

The relationship between a language and the semantics it is intended to express is often difficult to understand. Indeed, this fact is one of the reasons for this study. That the meaning of a block of statements in a specification depends upon a complex mathematical theory does not make this relationship any easier to discern. Notation and syntax should, in the worst case, have no effect whatsoever upon the expressibility of the abstraction.

In a programming language, the symbols which make up a program represent abstract objects with which most of us are familiar. The fact that a specification language "looks" and "feels" like a programming language is not necessarily a good thing. On the pro side, similarities between an algebraic specification language and procedural programming languages help those unfamiliar with the methodology to understand how to describe abstractions.

Unfortunately, this "understanding" is tainted by the knowledge most of us have gained through years of experience. It will not do to explain to a budding specification writer, "You can't use the name of a sort as an argument to an operator because a sort is just an index into a set of carriers."

There is one very important difference between a programming language and a specification language. The semantics of a programming language construct or of a particular statement in a program may be ambiguous for any number of reasons. The language may be poorly defined, there may be several "dialects" in use, and of course, the compiler writer may have erred during the implementation. Although the latter case is still possible in the implementation of a specification, one thing is certain -- the meaning of a particular axiom is completely defined. We may not *know* what we have written, we may think it means something it does not, we may even have expressed a built-in ambiguity[2], but the true *meaning* of an axiom is completely determined by the underlying theory we discussed in Chapter 2. The problem is figuring out what that meaning *is*. Unfortunately, one of the most important results of actually designing and implementation a specification is that we discover there is just no *easy* way to find this out. We cannot even be certain that an *incorrectly* specified abstraction will be guaranteed to fail when it is implemented, because any implementation is at best a finite instantiation of a subclass of objects described in the specification. One implementation may work fine because the values which uncover the ambiguity are simply not defined, while another, less restrictive implementation may not work at all. We will return to this issue again in our discussion of the implementation.

We have already noted that errors are difficult to handle in algebraic specifications. It is not that they are difficult to express, nor is it that it is difficult to determine where errors might occur. Rather, it is that a formal treatment of errors usually results in an explosion of extra terms due to a tremendous increase in the number and complexity of axioms, which must be modified to account for these "boundary conditions". All we have to say about

[2] An axiom which evaluates to two different terms, depending upon the order of evaluation, is explicitly ambiguous.

this is, we realize it is a diffcult problem which must be solved, and that we have no good solution for it.

2. Parameterized Specifications

The more complex the object we are attempting to describe, and particularly, the more general a class of *operations*, the more likely it is that a parameterized specification will be required. Since the meaning and method of expressing parameterized specifications are highly disputed, we have used it only once in our specification -- to describe a data type for character strings.

Parameterized specifications provide an additional level of abstraction to those we described in Chapter 2. They specify a template onto which the sorts and operators of another specification must be mapped. This mapping is one-to-one. The axioms and operations expressed in the body of the parameterized specification become available to the parameterized type when it is instantiated. Parameterized specifications make the already difficult task of determining the properties of the carrier sets even more difficult.

```
spec A
end A

spec B
        extend A
end B

spec C
        extend A
end C

spec D
        extend B,C
end D
```

Figure 3.2: The Problem with Extension

3. Extension

The concept of *extension* is somewhat analogous to the way attributes may be added to an object in an object oriented language, such as Smalltalk. An existing specification is "extended" with the addition of more sorts, operators and/or axioms. Extension is required because the process of building the abstract specification involves continuously adding to existing specifications, moving from low level primitives, through higher and higher levels of abstraction. The reader will note that this is a classic example of bottom up design. The algebraic specification methodolgy we use here requires it.

A serious problem with extension involves the proliferation and duplication of specifications through the abstraction hierarchy. Figure 3.2 illustrates this. Notice that specs B and C are extensions of A. But D extends B and C, so there are now two "copies" of A in D. The analogy to scoping an a programming language looks attractive, but is very weak, if not incorrect. It is closer to the concept of multiple inheritance in an object-oriented language. When we say extension adds new operators, axioms and sorts to an existing specification, we really mean "adds new objects and rules to an existing collection of objects and rules. Illustrated in Figure 3.2 is the addition of a specification to itself (A on A). What effect does this have upon the semantics we are describing? Most references do not treat this problem.

IV. DESIGN

Athough the literature is filled with examples used to illustrate how one might specify abstractions, few provide a practical treatment of the problem of designing a working system. This study has been an attempt to bring the theory down to earth -- to show that it really *is* possible to use algebraic specifications to design something which we not only can talk about, but which we can actually use.

A. THE SPECIFICATION LANGUAGE

Appendix A contains a high level grammar for our specification language, which is similar to examples found in the literature, with changes to give it the feel of a programming language. A "module" in the specification is called a *spec*. The entire specification forms a hierarchy of specs which are related to one another through the operation of *extension* which we described in the previous chapter. Each spec may introduce zero or more new sorts, operators and/or axioms, which may be added to an existing spec through extension, or which may form the primitives of a new "branch" of the hierarchy. Although it is conceivable that one might specify an object composed of disjoint specs, this is not the usual case. Extension provides the only means of relating the carriers and operators described in two different specs[1].

Our language also permits the use of parameterized specifications, although we minimize their use because their properties are not well understood.

We avoid a detailed description of the syntactic sugar, since this is essentially arbitrary. The semantics and overall structure, however, is not. For example, all symbols must be unique. No symbol may be used unless it has first appeared as the name of a spec, in a sort definition, or to the left of a colon in an operator definition. This rule guarantees that at no time are the properties of the object inferred from the name ambiguous. Thus, the structure of a specification is much

[1] There are several other operations by which two specifications may be related. They are discussed in Fasel (1983). We do not use them here.

like a Pascal program, but more restrictive. There are no self referential specs. and no use of a spec before it has been defined.

The language also introduces the idea of *primitive*, *derived*, *hidden* and *error* operators. Primitive operators are those which *must* be implemented to provide a full instantiation of the specification. Derived operators are simply that -- derived from the primitives. The implementor may elect to ignore these, secure in the knowledge that their functions may be performed by composition of primitives. In our specification for boolean, **or** and **implies** are derived operators. Error operators accept no arguments. They are guaranteed to return a value of the result sort which must be an error. The need for such operators and their limitations are described in detail in Goguen (1978). We found them, in practice, to be a nuisance. We will return to the issue of errors in our discussion of the implementation. Hidden operators are those to which the programmer has no access. They represent abstractions of the machine required to express a certain semantics but nothing more.

1. The Macro Preprocessor

One of the things we quickly realized as the specification became more complex was that the writer of a specification spends a lot of time writing the same thing, over and over again. This occurs whenever the specification calls for the description of a number of general purpose operators which operate on elements of a number of different carries through the use of a mapping function. Our **fetch** and **store** operators are an example of this. They are capable of storing and retrieving values of any type to and from primary storage. All the AM data types map into a common type, *value*, which is passed to or returned from **fetch** and **store**. The spec which describes the mapping function for each type is virtually identical except for the names of operators and sorts. Thus, we introduced a partially defined, imaginary macro preprocessor which provides for macros with parameters. The reader will see examples of its use throughout the specification.

The basic form of a macro definition is

replace "text..." with "other text..."

Since the lexics of our specification language does not require quotes, they are

used as delimiters for definition and equivalence strings. A macro with arguments looks like

replace(A,B,....Z) "text....." with "other text....."

where the formal parameters *must* be capital letters. Since we do not allow uppercase letters within the spec itself, an uppercase letter denotes a formal parameter to a macro. Thus for the definition

```
replace(A)
        "convert(A)"
with
        "atomofA: val → S"
```

then the string

convert(bool)

would be replaced by

atomofbool: val → bool

wherever it appeared.

B. THE MACHINE

AM is a abstract machine whose overall concept is based upon a simple design put forward by Fasel (1980). Appendix B contains the specification which describes it, and Appendix C contains the programmer's manual for a simple assembler which produces native, relocatable AM object code.

Now that we have the theory upon which to base a specfication, the next important question to answer is, what do we design? Our stated goal has been to contribute to solving the portability problem by attacking the semantic gap. But, not only must we design a machine, we must also remember and analyze the *process* of designing it. Therefore, we treat now this process, discussing our fundamental design decisions and the reasons behind them.

At the time of this writing there are many examples of advanced special and general purpose architectures. Some of the big names are RISC (Patterson 1982) and various language directed architectures (Waite 1975, Hoffman 1982 and Myers 1982). After a survey of these and other references, we decided to put off

talking about the architecture until we can see what sorts of things we might describe *and* understand with our specification language.

In his PhD thesis, Fasel (1980) describes a simple abstract machine called SAM (Single Accumulator Machine). After wading through his spec and some preliminary attempts to specify a few objects of greater complexity, we decided to model a conventional architecture. We have two very good reasons for this. First, since we are all familiar with the typical Von Neumann processor, we should be more likely to find good ways to formally describe it. Second, this same familiarity should make it more likely for others to understand our specification.

The next step is determining where to start. This is not too difficult. The operation of every machine can be reduced to a complex sequence of simple operations. At a level of abstraction below the basic data element and its primitives we should be required to specifiy the semantics of processing elements and control stores. At a level above the basic data element we would merely be adding another to the long list of Von Neumann programming languages. Therefore, we use as a basis for the specification, the primitive data types. In the interests of simplicity, we chose five: boolean, natural (unsigned), integer, character and string. These form the atomic data types, referred to hereafter as *atoms*.

Data types implemented on conventional architectures exhibit a built-in dependence upon the way in which values are represented in hardware. This arises naturally from design goals which stress storage efficiency, and leads to several undesirable properties. First, machine data structures are overloaded. Given an arbitrary address, not only can we not tell what type of data we have accessed, we can not even determine with certainty if we have accessed all of it. We might be in the middle of a floating point number or on the end of a character string. Second, nothing prevents a programmer from treating one type of data as another. Third, the "state" of a machine is impossible to analyze. The endless string of bits characterizing the "meaning" of a program at a particular instant provides no hope for proving something about the program's correctness. We therefore offer an architecture which will rationalize the

relationship between data and the machine, but which can be implemented easily either by emulation or through direct hardware means.

An abstract machine which solves these problems must have the following properties:

- In the organization of primary storage, the next logical data item is in the next logical address.
- Except as formally specified, no data type may be accessed in any way as another.
- Given any arbitrary logical address, the value stored there *and its type* can always be determined.

Hence, we use a tagged architecture with some very special characteristics[2], which takes away some of the programmer's freedom to "twiddle" bits. The resource provided by this architecture will now be partitioned into functional areas along the lines of a conventional machine.

Typical resources available at the instruction set level include the primary storage, high speed registers, stacks, I/O ports and perhaps a heap. We will define abstractions for each of these. Here, again, we see a marked difference between the conventional view of the physical resource and that *imposed* by our specification method. Ports, stacks and the heap are usually thrown right in with the rest of the program and data. In fact, as we have said in Chapter 1, stacks are often accessed as arrays. AM treats each of these resources as a black box. One may push, pop and read the top of a stack, but the stack pointer is inaccessible, as are any values below the top element (unless one pops the stack to reach them). We thus remove another freedom once enjoyed by the programmer -- that of treating one type of data structure as another.

A conventional instruction set forms an abstraction closely tied to the representation of data in the hardware. Our architecture makes this impossible. Instead, whatever instruction set we design will become much closer to a primitive problem solving abstraction. Again in the interest of simplicity and understanding, we define an instruction set which should be thoroughly familiar to most readers who have programmed in assembly language.

[2] A proposal for a hardware implementation with these properties is given in Yurchak (1984).

The restrictions upon the programmer's freedom which we have discussed are justified because by giving up the ability to do almost anything we can imagine, we gain the ability to explicitly specify our *intent* during the course of a program. The specification does not specify what a resource is or how it is implemented. It *does* specify exactly what the resource means and how to use it.

AM is an abstraction of a conventional Von Neumann resource with some unconventional properties. The primary (only) machine element is called a *value*. All data primitives (atoms) map into values. Primary storage is an array of one or more memory segments, each of which may contain an arbitrary number of cells. Each cell is capable of "containing" any legal data value. Both programs and data may reside together in a single segment. For high speed storage, there are one or more register segments, each of which contains an arbitrary numbr of registers. Again, every register is capable of containing any type of data. AM also has one or more stacks, a heap, and a crude file system. We will discuss the details in the next section.

The basic atomic data types are augmented by several others needed for the execution of programs. These are memory addresses, register addresses, stack addresses, file addresses and instructions.

C. THE SPEC

The specification for AM is contained in Appendix B. The language used to describe it obeys the grammar found in Appendix A. We will discuss the specification in some detail since portions are nonintuitive.

1. Macro Definitions

At the top of the specification are listed a number of macro definitions. We concern ourselves for the time being with just those definitions pertaining to the properties of relations. The intended properties of certain operators will require that we express axioms for commutativity, transitivity, etc., throughout the specification. Rather than write this out repeatedly, we define macros with appropriate parameters which permit a more readable and explicit expression of these properties. Take, for example, the equality operator for integers,

eqint: int,int → bool;

which returns **true** if the arguments are equivalent, **false** if not. We should like to express that **eqint** is an equivalence relation on objects of type **int**. Thus, we need the following axioms:

```
eqint(i,i) = true;
eqint(i,j) = eqint(j,i);
implies(and(eqint(i,j),eqint(j,k)),eqint(i,k)) = true;
```

But there are relations like this one throughout our specification. Thus we define macros like

```
replace(X,S)
        "equivrel(X,S)"
with
        "for i in S
                X(i,i) = true;
        for i,j in S
                X(i,j) = X(j,i);
        for i,j,k in S
                implies(and(X(i,j),X(j,k)),X(i,k)) = true"
```

which permits us to write, in the case of **eqint**,

```
equivrel(eqint,int);
```

We then read this as "**eqint** is an equivalence relation on **int** ". Note that we are not required to explicitly specify the type of free variables, since this can normally be determined by context. We do so in the interest of clarity, since *there can be no doubt for which type* **eqint** *is an equivalence relation.*

For the reader who doubts that the more complex macros described in this specification will work, a modified version of the familiar M4 macro preprocessor[3] will correctly deal with every macro found in our specification.

2. The Atomic Types

The basic data types form the primitive objects of the problem solving abstraction. The programmer's algorithm must in some way be mapped into these abjects. Boolean is described first because, not only is it a data type available for use by the programmer, it is also part of the specification itself.

[3] See Kernighan and Ritchie, *The M4 Macro Preprocessor*, Bell Laboratories, Murray Hill, New Jersey, July 1974.

iodifying a few constants. Only one module of our interpreter need be ecompiled to make this alteration.

5. MAPPING OPERATORS TO FUNCTIONS

It seems natural, although incorrect, to look at the operators in a spec as unctions. *However, in the implementation*, this makes perfect sense. Figure 5.4 sts the code for the AM module which implements the boolean type. The eader files which provide the constant definitions are omitted here. Notice that, vhere possible, we rely upon the operations provided by the C language, rather

```
BOOL atomofbool(v)
VAL  v;
{
        BOOL b;

        if (v.type != BOOL_VAL)
                error("value not of type BOOL - %x",v.type);
        b.type = BOOL_TYPE;
        b.val = v.boolval.val;
        return(b);
}

VAL  valofbool(b)
BOOL b;
{
        VAL  v;

        if (b.type != BOOL_TYPE)
                error("atom not of type BOOL - %x",b.type);
        v.boolval.type = BOOL_VAL;
        v.boolval.val = b.val;
        return(v);
}
```

Figure 5.5: Error Handling

```
BOOL true = { BOOL_TYPE. 1 };
BOOL false = { BOOL_TYPE. 0 };

BOOL not(a)
BOOL a;
{
        a.val = !a.val;
        return(a);
}

BOOL and(a,b)
BOOL a,b;
{
        a.val = (a.val && b.val);
        return(a);
}

BOOL or(a,b)
BOOL a,b;
{
        a.val = (a.val || b.val);
        return(a);
}


BOOL eqbool(a,b)
BOOL a,b;
{
        a.val = (a.val == b.val);
        return(a);
}


BOOL nebool(a,b)
BOOL a,b;
{
        a.val = (a.val != b.val);
        return(a);
}
```

Figure 5.4: Operator-Function Mapping

```
typedef struct {
        int     size;
        VAL     **val;
} memseg;
typedef struct {
        int     num;
        VAL     **val;
} regseg;
typedef struct {
        int     size;
        int     sp;
        VAL     **val;
} stkseg;
typedef struct {
        int     stat;
        int     mode;
        int     type;
        int     val;
} fileseg;

#define _NUMMEMSEG   1024
#define _NUMUSRSEG   2
#define _NUMREGSEG   1
#define _NUMSTKSEG   1
#define _NUMFILES    16

memseg      _mem[_NUMMEMSEG] = {
        1024, 0,
        1024, 0 };
regseg _reg[_NUMREGSEG] = {
        32, 0 };
stkseg _stk[_NUMSTKSEG] = {
        512,512,0 };
fileseg _file[_NUMFILES] = {
        1,RMODE,CHAR_VAL,0,
        1,WMODE,CHAR_VAL,1,
        1,WMODE,CHAR_VAL,2 }
```

Figure 5.3: The Physical Resource

interpreter. Each atom is represented as a structure consisting of a 16-bit tag field, and a value field. The size of the value field varies with the type. Each sort in the specification is assigned a sixteen bit code. Whenever an atom is created, or copied, it is tagged with the appropriate code.

By using a fixed size tag field as the first field in each record, we build in some additional robustness, since even in the event of a mistyped structure being copied into the formal parameter of a function, we can rely upon the first word to be a valid code (the type).

The next step is to describe the structure for machine values, which must be capable of containing any atom. This is more difficult. We resort here to subterfuge. Our specification method relies upon the *extend* operation to build more and more complex specifications. Unfortunately, there are few Von Neumann languages which permit additions to the definition of a data type once the compiler has seen it. In C, we cannot specify directly two structures which contain each other. So, we resort to the technique illustrated in Figure 5.2. The problem is caused by the type **instr** which represents the opcode returned when each instruction operator is invoked. These **instr** atoms must contain values for their operands (as part of the opcode), but are themselves values, since we must be able to store and fetch instructions. How else would we get a program into memory and execute it? The solution is to fool C into thinking we are talking about pointers to structures instead of structures themselves. This works fine since we implement an instruction opcode as a structure whose value field is a pointer to the opcodes.

The primary physical resources are also defined as structures (Figure 5.3). Registers, primary storage and stacks are represented as arrays of arrays of pointers to values. The reader should note that a simple change to the constants in the header files can completely alter the configuration of the machine. We can specify an arbitrary number of arbitrarily long memory segments and register segments, and an arbitrary number of different sized stacks. Files are represented as usual as an array of structures containing status information and an input/output buffer. The number and type of files can also be changed by

A. IMPLEMENTING DATA TYPES

As in any piece of substantial software, we look first at the data structures required to support our algorithm (which in this case was represented by the specification). We chose C as it appeared to provide the easiest translation from the specification. In retrospect, Lisp would work very well, too.

AM is a tagged architecture. Each data element must be self descriptive. The most likely construct to provide this is a structure (record), and this is what we used. Figure 5.1 lists some fragments from the header files used by our

```
typedef short opcode;

typedef struct {
        short  type;
        union value  *val;
} INSTR;

typedef union value {
        short  type;
        opcode        opcodeval;
        BOOL boolval;
        INT   intval;
        NAT   natval;
        CHARcharval;
        STRING        stringval;
        MEMADDR memaddrval;
        REGADDR  regaddrval;
        STKADDR  stkaddrval;
        FIL   fileval;
        INSTR         instrval;
        MOP  mopval;
        DOP  dopval;
        RELOP         relopval;
        BOP  bopval;
} VAL;
```

Figure 5.2: Machine Values

V. IMPLEMENTATION

AM is implemented as a finite state machine interpreter. It comprises approximately 12000 lines of C code, including the assembler. Details of the assembler are treated in Appendix C. The overall concept is quite simple. A text file representing an assembly language program is translated by the assmbler into a relocatable object module. A loader, part of the AM interpreter, loads this object module into the appropriate cells, and AM executes it.

There are only four issues of real interest concerning the details of the implementation. These are the representation of data types, the mapping of operators in the specification to functions in the interpreter, the handling of errors, and the execution of a program.

The AM interpreter is a fairly large program by most standards. We feel it notable that the period of time from completion of the specification to a working version of the interpreter spanned just two weeks! We attribute this level of productivity largely to the existence of the specification, which left absolutely no doubt about the meaning of operations. Once a few mechanical obstacles had been bridged, writing the program was largely repetition.

```
#define NAT_TYPE        0x0002

typedef unsigned intnat;

typedef struct {
        short  type;
        nat    val;
} NAT;
```

Figure 5.1: Type Definitions

To completely specify the effects of errors a set of axioms must be supplied for each operator. We avoid this in the interest of simplicity, but caution the reader that errors have not been properly treated here.

We guarantee there are errors in our specification, and encourage to reader to do as we have done: stare at the specification, and thoroughly test its implementation. We note that the design and implementation of a specification language brings with it the host of problems which follow more conventional programming languages. We do not have a way of determining if the specification is correct, let alone whether or not it describes what we want it to.

However, we have demonstrated how something of useful complexity can be described using algebraic specifications. By moving toward the problem solving abstraction from the resource side, we require data to be manipulated in a representation independent way. We have also shown that, by capturing the true meaning of the machine's data structures in a specification, we remove the semantic ambiguities usually encountered where a resource oriented instruction set meets a problem oriented language. The instruction set becomes a medium through which we may unambiguously express our intent in a program.

This axiom is used to "fire off" a program. The progression from one instruction to another is given, as one might suspect, in the axioms for each instruction. For example, consider the move memory-to-memory instruction (mov_m_m). The axiom which defines its semantics is

```
xeq(mov_m_m(m1,m2),m,q) =
        prog(nextmemaddr(m),storem(fetchm(m1,q),m2,q));
```

which means:

> The state resulting from the execution of **mov_m_m** with operands m1 and m2 at address m in state q is equivalent to the state of the program at the next address, after what is fetched from m1 is stored in m2.

Notice that the q's in the axiom are identical (refer to the same state). The reader should see that, through this axiom, we have fetched, decoded and executed an instruction, and incremented the program counter (m). The other axioms in the spec express exactly the same relationship between **xeq** and **prog**.

Sequencing must be expressed as a nesting of operators. Thus, the execution of an AM program amounts to a non-deterministic recursion between **xeq** and **prog**. Cleverness on the part of the implementor is required to enable AM to execute programs of useful length.

6. Remarks

The reasons for various distinctions among objects which, in a conventional design, would more intuitively be lumped together are often subtle. However, they reflect a conscious effort to capture the abstraction of a machine at a level low enough to provide a degree of flexibility in writing the axioms which define its semantics. The higher the level of abstraction, the more difficult it is to infer a direct correspondence between the resource and the specification which describes it.

We should at this point rationalize our error operations and move as many as possible into a dedicated error spec, where they can be properly handled as a whole. The specification in Appendix B does not reflect this. The result is that error ops and axioms are scattered throughout the specification.

Heap operators are provided to permit the dynamic allocation of arbitrary sized memory segments for constructing linked lists, frames and other structures. **lalloc** returns the identifier to a segment of n cells. **lfree** deallocates it, returning it to the heap. The **indir** operator has been designed expressly to permit up-level addressing through a list of frames allocated using **lalloc**.

The file operators are not really part of AM. They resemble a set of typical operating system service calls. We provide them to enable AM to communicate with the outside world. Files are much more interesting than the other structures, since their semantics resemble the operation of an infinite sequence generator. How, for instance, do we specify the semantics of a removable cartridge disk drive? What is read off the file may in no way be related to what was written on it.

5. Execution

At this point AM is essentially complete. At our disposal are all the tools we need to build programs to manipulate data any way we want. Missing, however, is a means for executing programs. What we have just described is a fairly typical Von Neumann architecture with the bounds removed. We have noted in the previous chapter how difficult it is to express the passage of time. How do we express the sequential execution of a program? Our solution is derived from that used by Fasel (1980).

Define two operators, we call them **prog** and **xeq**, which are co-recursive. The semantics of execution are given by

```
prog: memaddr,state → state;
xeq: instr,memaddr,state → state;

prog(m,q) = xeq(atomofinstr(fetchm(m,q)),m,q);
```

where m is a memory address (in this case representing the value of the program counter) and q is the current state of execution. The axiom can be interpreted like this:

> At any moment, the state of the program at address m in state q is equivalent to the execution of the instruction stored at that address.

of diagram or table showing the characteristic bit patterns of the opcode, with "holes" to be filled in by the operands to the instruction. The operators specified in the aminstructions spec correspond precisely to these diagrams. They accept zero or more "operands" and return an atom representing the aggregate opcode. The AM assembler uses these operators to construct an AM object program. Notice, there are no axioms in this spec. After a brief study the reader will see that these instructions are similar to those found on a large number of popular processors. For a description of the naming conventions, see Appendix C.

The only thing left to do is to relate the objects we have just described to **val** so that they can be stored and fetched in the same manner as the atomic data types. To do this we again invoke the **newtype** macro.

4. State

The next spec forms the heart of AM. It describes the semantics of the the physical resource. A new sort is introduced, **state**, which at any moment represents the state of execution. Every operator whose result depends upon the current state of execution must accept a state value as an argument, and every operator which alters the state must return a state value. Thus a familiar pattern develops throughout the operators in this spec. For example, to examine the value stored in a register or memory cell, we must provide not only the address, but also the current state of the machine. The state is not altered. However, when a value is stored, a new state must be returned.

initam returns a constant representing the "initial state" of the machine. By implication, all values in the machine in state **initam** are "undefined". This is an error condition, specified with the value **undef**. The axioms make it impossible to fetch from a cell whose value is undefined.

Fetch and store for memory and registers is self-explanatory. Worth noting are the axioms which relate them as inverses.

The stack operators are also straightforward. Notice it is impossible to alter the stack pointer (which is only implied in the operators) except by pushing or popping a value. The axioms relate the operators, and make it an error to pop or access the top of an empty stack.

Next we specify the operators for accessing registers. This is easier, since we may not perform address "arithmetic" on register addresses. We need only have a way for obtaining all of them, given the first. As with memaddresses, we provide an operator for deciding whether or not two register addresses are equal. Notice that we draw a distinction between addresses and integer displacements, although operators like **offset** allow mixed use of these and others sorts in precisely defined ways.

The stack is a little more interesting. We do not want the programmer to have access to the "inside" of the stack, nor do we want to provide facilities for altering the stackpointer. We therefore provide an operation for returning the stack pointer, and for determining whether or not two stack pointers are equal, but no more. The anticipated push and pop operations cannot be defined here for the same reasons we have not defined store and fetch operations -- we have yet to define all the objects which might be stored or fetched, and we have no concept of a machine state. This will be treated shortly.

The spec for files offers the same "black box" abstraction as the stack. We want to give the programmer access only through a carefully designed set of as yet unspecified primitives. Therefore, the only referencing primitive is that for obtaining a file's address.

AM's intrinsic operator codes are next defined in the amoperators spec. These give the programmer access to the atomic operators provided with each data type. Each such atomic operator is mapped to a corresponding operator in amoperators (its machine code). We introduce a new sort for each type of operator (monadic, dyadic, relational, etc.) and the operators themselves. A set of **apply**... operators are also specified. These will accept an instrinsic op of the appropriate type, plus one or more argument values, and return a result. They form AM's arithmetic and logic unit (ALU). Also defined here are sets of relational ops for those types in which they have meaning. These will provide the programmer with the primitives for conditional branch instructions.

The next spec defines the instruction set as a set of operators which all return an atom of the sort **instr**. They are the opcode templates. In a typical assembly language manual, the description of each instruction includes some sort

- It is the responsibility of the specification writer to explicitly define the effects of errors on the object being described.

In our specification, the only proper way to handle conversion errors is to provide a set of axioms defining the result of an expression containing opposing **atomof**... and **valof**... operators whose sorts do not match. This is handled in the error spec.

So, the atomic types are introduced as machine data type. Strings are a special case, since we must first instantiate the paramaterized spec for strings of characters, and then relate it to **val**. This is done with spec charstring and spec str.chartype. Note the dot notation, similar to an aggregate structure reference, used to denote the relation of the chartype spec to the sort **str**.

3. Machine Primitives

We must now specify an abstraction of the operations of the machine itself. We need to be able to reference values, specify arithmetic and logical operations, and define instruction opcodes. We start with identifiers.

The concept of *identifier*, as we use it here, refers to the name of an abstract data structure composing some physical resource, such as a memory segment, a stack, or a file. Identifiers are needed to allow us to reference these structures as complete objects. The only operation we need is a comparison for equality for each type.

We then write specifications for each of the types of addresses we will need, one for each AM data structure. The memaddresses spec defines the operators used to reference values in primary storage. Given the identifier of a memory segment, the base address is returned by **startmemaddr**. Successive and previous addresses may be obtained using **nextmemaddr** and **prevmemaddr** respectively. Note, there is no previous address to **startmemaddr**. This condition is defined as an error in the axioms. **offset** permits arbitrary values to be referenced as integer displacements from another. Its semantics is defined recursively. Note how the memaddresses spec defines an abstraction which exhibits the properties we required for our machine -- that the next (previous) data item is in the next (previous) logical address.

Now that the atomic types are specified, we must define AM's basic element of storage, the **value**. The relationship of the atomic types to machine values must be expressed in terms of **value**. The spec is trivial. It introduces a single sort, **val** and an error op **typerr** to express the condition corresponding to a type conversion error. Now, examine the macro **newtype** at the top of the specification. It expands a statement of the form

newtype(sortname,specname);

into an actual spec defining a new data type to AM which is an extension of the atom's spec and **value**. Within this spec are the key operators and axioms which imply AM's tagged architecture. Using integer as an example, **valofint** accepts an atom of type **int** and returns a **val**. **atomofint** accepts a **val** and returns an **int** atom. The special properties of the operator **atomofint** are expressed in the axiom

atomofint(valofint(x)) = x;

which relates **atomof...** and **valof...** as inverses. Thus, given any value of type **val**, **atomof...** will extract an atom of the appropriate type.

Here we must deal once again with errors. Operators are *not* functions, and their arguments are *not* parameters. An operator's characteristic determines the types of objects it can accept, and the type of object it returns. It is an abstract object which defines a *protocol* of communication with respect to other abstract objects in a specific way. It is not precisely an error for a value of the wrong sort to appear as an argument for an operator. It has no meaning at all. In fact, algebraic specifications provide no way of expressing the relationship of other objects to the characteristic of an operator. This is one of the stumbling blocks of the methodology. Goguen (1978) discusses this in great detail. Unfortunately, in the *real* objects defined by the abstraction, there may come a time when an object described with one spec appears where an object of another type is expected. Therefore, we avoid a rigorous treatment of errors by substituting for a theory the following rules:

- If any value violates the characteristic of an operator, that operator returns an error of the type corresponding to its return type.

Many axioms in other parts of the specification require boolean to express their meaning.

Note that in this and every other spec, the spec name is distinct from the name of any sort. Any similarities in them are purely arbitrary. The name given to a spec denotes an abstract object, the aggregate of sorts and operators and axioms. The name given a sort is an index into a set of carriers. It denotes a specific set of values which, together with operators, forms an abstract data type. In any but the most simple specifications, it will be very difficult to point to a single thing and say "This is the data type so-and-so." Throughout this thesis we loosly refer to "the type int" or "the type integer". This is imprecise, but for lack of a convenient way of expressing ourselves, we shall continue to freely mix these terms. The reader is warned to examine Chapter 2 again if this point is unclear.

In the spec for boolean, **or** and **implies** are specified as derived operators. We provide them for convenience only. DeMorgan's axioms may be omitted as well.

Natural is then expressed as an extension of boolean, and integer as an extension of natural. A typical set of operators is provided. We do not specify multiplication or division, although using conditional axioms this is not too difficult. Integer extends boolean to permit conversions to be specified. AM allows conversions between no other types. Note that the *zero* values of natural and integer are distinct, as are all other members of their respective carriers.

The spec for character defines 128 ASCII codes. The symbol for each character (each a 0-ary operator returning a constant value) includes the bracketing single quotes.

String is expressed as a parameterized specification. The parameter template must be matched in a one-to-one correspondance by some other spec before a string type may be instantiated. Thus, we may have strings of anything, so long as a spec exists with a single sort and two operators whose semantics exactly matches axioms in the parameter template. The syntax we use to express the mapping of sorts to sorts and operators to operators is awkward but necessary to prevent the description of impossible objects.

than slow down an already slow interpreter with axiomatic implementations of the operators.

One of the design decisions we must make is whether to pass structures or pointers to structures throughout the program. Pointers are faster from the standpoint of parameter passing, but make it difficult to determine when to free unwanted values. Passing structures is safer, because a new copy of the data is made within each function, but it is slow. We choose to be slow, but safe -- we pass structures

As the implementation proceeds to more and more complex specifications, the program relies less and less upon C and more and more upon the bulk of operators which we have defined. In fact, the more complex operators are implemented as calls to previously defined functions which almost directly mimic the axioms from which they are derived. We will illustrate this shortly.

C. ERROR HANDLING

All errors are fatal, but they need not be. Those errors which are not must be defined explicitly in the specification. As we have said, a more detailed treatment of errors would be an area for further study.

AM flags most errors in the operators which perform data conversions. This is a natural place for this to occur, since it is difficult to see how the type of a data element may be changed at any other time. Figure 5.5 lists a fragment which implements the boolean conversion routines. The routine error() does not return, but terminates execution after writing the error message to stderr. Notice that, even if a much larger structure was passed to atomofbool() or valofbool(), the error would be detected and handled gracefully.

This type of error checking is also performed in the functions which implement data operations.

D. EXECUTION

The final point of interest involves actually executing a program. The method is also illustrative of the way in which the program mimics the axioms of the specification. Here, too, we resort to subterfuge to implement in a finite way a specification which could require the expendature of an infinite resource (an

```
#include <setjmp.h>

jmp_buf     _context;

MEMADDR cond();

main(argc,argv)
char   *argv[];
{
        int     ap;

        for (ap=1; ap < argc; ap++) {
                if (*argv[ap] == '-') {
                        if (*(argv[ap]+1) == 't')
                                traceflag = 1;
                }
        }

        initam();
        amload();

        setjmp(_context);
        Q = prog(_pc,Q);

        exit(0);
}

STATE       prog(m,q)
MEMADDR m;
STATE       q;
{
        q = xeq(atomofinstr(fetchm(m,q)),m,q);
}

STATE       xeq(i,m,q)
INSTR       i;
MEMADDR m;
STATE       q;
{
        opnd  *p;

        if (i.type != INSTR_TYPE)
                error("attempt to execute non-instruction - %x",i.type);
```

```
        p = i.val;
        switch (getopcode(p[0].opcodeval)) {
        /*
                a case and semantics for each valid opcode
                goes in here
        */
        default:
                error("attempt to execute an illegal instruction - %x",
                        p[0].opcodeval);
        }
        longjmp(_context,1);
}

MEMADDR cond(b,m1,m2)
VAL  b;
MEMADDR m1,m2;
{
        return(b.boolval.val? m1: m2);
}
```

Figure 5.6: Program Execution

implied stack in this case). The problem is the corecursive relationship between the functions xeq() and prog(). We eliminate this problem by never actually returning from xeq(). We rely on a dangerous but effective C idiom, setjmp() and longjmp(). Figure 5.6 illustrates.

In main(), initam() configures AM and invokes all of the initialization operators. amload() loads a program from secondary storage into the appropriate cells as directed by the linker directives in the object module. Setjmp() then saves the state of the "real" machine. The variable _pc is the program counter which is set inside amload(). Now everything is set. The program is loaded and ready to run.

prog() is now called. Notice that prog simply invokes xeq(). Recall now the axiom which defines the semantics of execution.

prog(m,q) = xeq(atomofinstr(fetchm(m,q)),m,q);

```
case MOV_M_M:
        q = storem(
                fetchm(
                        p[1].memaddrval,
                        q
                ),
                p[2].memaddrval,
                q
        );
        _pc = nextmemaddr(_pc);
        break
```

Figure 5.7: The Semantics for mov_m_m

The value of a language which permits usefully long and descriptive names is obvious in this case. Within xeq() a large case statement decodes the instruction and executes it according to the semantics provided for that case. This semantics is very closely modeled on the axioms in the specification. Figure 5.7 lists one such case and its accompanying semantic action. Compare it to the axiom for **mov_m_m**.

```
xeq(mov_m_m(m1,m2),m,q) =
        prog(
                nextmemaddr(m),
                storem(
                        fetchm(m1,q),
                        m2,
                        q
                )
        );
```

The similarities are not accidental. This should make the point that it is beneficial for the implementation language to permit such a close modeling of the specification. Obviously, this made the implementation easier to write, easier to debug and easier to understand.

E. OBSERVATIONS

AM is slow (about as fast as the average Basic interpreter). But we have been unable to make it fail in 2 months of testing. AM refuses to do anything which has not been expressly defined in the specification from which it is implemented. This is encouraging.

As stated earlier, coding went extremely quickly (about 3000 lines a week). We attribute this to the presence of the specification, which was a template for the program, and C, which translates nicely from our specification language. We can make a case here for a rule which would require that the specification language be syntactically and structurally similar to the implementation languages.

The next step would be to implement the interpreter in microcode on a writable control store. This may imply a change in the specification language syntax.

We designed and implemented a Von Neumann resource, but need not have done this. This methodology should be amenable to a wide variety of architectures and implementations. In fact, if an architecture appears to be particularly unsuited to formal specification, it should become suspect. We strongly believe that because the methodology suggests a tagged, non-overlapping storage organization, this tells us something about the way we *should* be designing machines.

VI. CONCLUSIONS

We have noted that a semantic gap exists where concepts which are primarily resource oriented clash with those which are primarily problem oriented. We have described a theory upon which to base a method for formally specifying the meaning of an abstract machine and the resource it represents. We have shown how something usefully complex can be described with this method, and that it can be successfully implemented.

So, what have we learned? As in all cases where physical objects and their observable properties must be abstracted, algebraic specifications describe only fragments of the physical world. The writer of the specification is faced with the difficult task of eliding unnecessary detail from a collection of facts and assumptions while capturing the essential semantics, and nothing more. This is difficult for a number of reasons:

- Designing a specification is at least as difficult as designing a programming language, with a similar set of issues and problems.
- The writer is obligated to understand and abide by a set of precise restrictions imposed by the theory upon which the specification method is based.
- There are no developments tools to support this methodology.
- The problem of testing and proving a specification correct is, as yet, unresolved.
- No method has been developed for finding the differences, if any, between the semantics actually defined by a specification, and those intended by the writer.
- The fact that any implementation can be only a finite instantiation of a specification poses a similar set of problems to those surrounding the acceptance of language and hardware standards.

These difficulties not withstanding, we cannot avoid the rising complexity of hardware and software, nor can we ignore the ways in which resource dependence adversely affects software portability. We have explored a method for describing and thinking about machines in a rational way, which permits us to better

understand the relationship between software, and the resource upon which it is implemented.

A. FUTURE WORK

Algebraic specifications provide a plausible method for formally describing a physical resource abstraction -- this we have demonstrated. We suggest the following areas for continuing research:

- Implement a specification in microcode, using a writable control store.
- Port the abstract machine interpreter to a number of different physical resources.
- Implement a high level language on the abstract machine, and test its portability between several implementations of the machine.
- Rationalize the treatment of errors within a specification.
- Develop an abstraction for a file system and a bit-mapped display.
- Write a compiler which can perform syntactic and semantic analyses on a specification, determine its properties, and generate a test suite of terms to validate it.
- Examine a variety of architectures as to their describability using the algebraic specification methodology.

APPENDIX A: A GRAMMAR FOR ALGEBRAIC SPECIFICATIONS

```
abstraction:
        (abstraction spec)?

spec:
        (spechead | parmhead) specbody specend

spechead:
        nameblk 'is'

parmhead:
        nameblk 'parm' specbody 'is'

specend:
        'end' specname

nameblk:
        'spec' specname

specbody:
        extension
      | specblk

extension:
        extendblk specblk 'end' 'extend'

extendblk:
        'extend' specnames 'with'

specnames:
        specname
      | specnames ',' specname

specblk:
        useblk
      | sortblk? opblk axiomblk?

useblk:
        'use' specname '(' specname ')' mapping? specblk 'enduse'

mapping:
        'where' eqivlist

equivlist:
        equivalence ';'
      | equivlist equivalence ';'
```

```
equivalence:
        sortname 'is' sortname
      | opname 'is' opname

sortblk:
        'sort' sortnames

sortnames:
        sortname ';'
      | sortnames sortname ';'

opblk:
        primblk? dervblk? errblk?

primblk:
        'primitive' 'op' ops

ops:
        op ';'
      | ops op ';'

op:
        opname ':' arglist? '->' sortname

arglist:
        sortname
      | arglist ',' sortname

dervblk:
        'derived' 'op' ops

erblk:
        'error' 'op' ops

axiomblk:
        'axiom' axioms

axioms:
        axiom ';'
      | axioms axiom ';'

axiom:
        ('for' varlist 'in' sortname )? termexpr '=' termexpr

termexpr:
        factor
```

```
        | opname '(' factors ')'

factors:
          factor
        | factors ',' factor

factor:
          opname
        | freevar

varlist:
          freevar
        | varlist ',' freevar
```

APPENDIX B: THE SPECIFICATION FOR AM

```
! amspec

replace(X,S)
    "equivrel(X,S)"
with
    "for i,j,k in S
        X(i,i) = true;
        X(i,j) = X(j,i);
        implies(and(X(i,j),X(j,k)),X(i,k)) = true"


replace(X,S)
    "reflexive(X,S)"
with
    "for i in S
        X(i,i) = true"


replace(X,S)
    "commutative(X,S)"
with
    "for i,j in S
        X(i,j) = X(j,i)"


replace(X,S)
    "transitive(X,S)"
with
    "for i,j,k in S
        implies(and(X(i,j),X(j,k)),X(i,k)) = true"


replace(X,S)
    "associative(X,S)"
with
    "for i,j,k in S
        X(i,X(j,k)) = X(X(i,j),k)"


replace(X,S)
    "irreflexive(X,S)"
with
    "for i in S
        X(i,i) = false"


replace(X,S)
    "symmetric(X,S)"
with
    "for i,j in S
        implies(X(i,j),X(j,i)) = true"
```

```
replace(X,S)
    "antisymmetric(X,S)"
with
    "for i,j in S
        implies(and(X(i,j),X(j,i)),(i = j))    true"


replace(S,T)
    "newtype(S,T)"
with
    "spec Stype
    is
        extend
            T,
            value
        with
            primitive
            op
                atomofS: val → S;
                valofS: S → val;
            error
            op
                Serr: → S;
            axiom
                for x in val
                    atomofS(valofS(x)) = x;
                atomofS(typerr) = Serr;
        end extend;
    end Stype"


spec boolean
is
    sort
        bool;
    primitive
    op
        true: → bool;
        false: → bool;
        not: bool → bool;
        and: bool,bool → bool;
    derived
    op
        or: bool,bool → bool;
        implies: bool,bool → bool;
    axiom
        false = not(true);
        not(not(b)) = b;
        and(true,b) = b;
        and(false,b) = false;
        not(and(b1,b2)) = or(not(b1),not(b2));
        or(b1,b2) = not and(not(b1),not(b2));
        not(or(b1,b2)) = and(not(b1),not(b2));
        or(true,b) = true;
        or(false,b) = b;
        commutative(and,bool);
        commutative(or,bool);
        implies(b1,b2) = not(and(b1,not(b2)));
end boolean;
```

```
spec natural
is
    extend
        boolean
    with
        sort
            nat;
        primitive
        op
            prednat: nat → nat;
            succnat: nat → nat;
            sumnat: nat,nat → nat;
            zeronat. → nat;
            eqnat: nat,nat → bool;
            gtnat: nat,nat → bool;
        axiom
            prednat(zeronat) = zeronat;
            prednat(succnat(n)) = n;
            commutative(sumnat,nat);
            associative(sumnat,nat);
            sumnat(n,zeronat) = n;
            sumnat(n,succnat(m)) = succnat(sumnat(n,m));
            equivrel(eqnat,nat);
            irreflexive(gtnat,nat);
            transitive(gtnat,nat);
            gtnat(succnat(n),n) = true;
    end extend;
end natural;
```

```
c integer

 extend
    boolean,
    nat
 with
    sort
        int;
    primitive
    op
        predint: int → int;
        succint: int → int;
        sumint: int,int → int;
        zeroint: → int;
        eqint: int,int → bool;
        gtint: int,int → bool;
        ntoi: nat → int;
        iton: int → nat;
    error
    op
        nconverr: → nat,
    axiom
        predint(succint(n)) = n;
        commutative(sumint,int);
        associative(summint,int);
        sumint(n,zeroint) = n;
        sumint(n,succint(m)) = succint(sumint(n,m));
        equivrel(eqint,int);
        irreflexive(gtint,int);
        transitive(gtint,int);
        gtint(succint(n),n) = true;
        ntoi(zeronat) = zeroint;
        ntoi(succnat(n)) = sumint(succint(zeroint),ntoi(n));
        iton(zeroint) = zeronat;
        iton(succint(n)) =
            if or(gtint(n,zeroint),eqint(n,zeroint))=true
            then
                sumnat(succnat(zeronat),iton(n));
            else
                nconverr;
  end extend;
nd integer;
```

```
am

xtend
   amstate,
   memaddrtype,
   regaddrtype,
   stkaddrtype,
   instrtype
with
   sort
      type;
   primitive
   op
      prog: memaddr,state → state;
   hidden
   op
      xeq: instr,memaddr,state → state;
      cond: val,memaddr,memaddr → memaddr;
      whattype: val → type;
      typeundef:    type;
      typebool: → type;
      typechar: → type;
      typenat: → type;
      typeint: → type;
      typestring.char: → type;
      typememaddr: → type;
      typefile: → type;
      typeinstr: → type;
      eqtype: type,type → bool;
      isundef: → bop;
      isbool: → bop;
      ischar: → bop;
      isnat: → bop;
      isint: → bop;
      isstring.char: → bop;
      isinstr: → bop;
      ismemaddr: → bop;
      isfile: → bop;
   axiom
      whattype undef = typeundef;
      whattype valofbool(b) = typebool;
      whattype valofchar(c) = typechar;
      whattype valofnat(n) = typenat;
      whattype valofint(i) = typeint;
      whattype valofstring.char(s) = typestring.char;
      whattype valofmemaddr(m) = typememaddr;
      whattype valoffile(f) = typefile;
      whattype valofinstr(i) = typeinstr;
      replace(S)
         "isops(S);"
      with
         "applybop(isS,v) =
            if eqtype(whattype v,typeS) = true then
               valofbool true;
            else
               valofbool false;
            endif;"
      isops(bool),
      isops(char),
      isops(nat),
      isops(int);
```

```
            nonallocerr;
        storem(v,offset(n,startmemaddr(lalloc(n1,q))),lfree(lalloc(i,q),q2))
            = staterr;
        offset(n,offset(n1,startmemaddr(lalloc(n2,q)))) =
            if or(gtint(n,ntoi(n2)),eqint(n,ntoi(n2))) = true
            then
                accesserr;
            else
                offset(sumint(n,n1),
                startmemaddr(lalloc(n2,q)));
        indir(zeronat,m) = m;
        indir(succnat(n),m) = atomofmemaddr(fetchm(indir(n,m),q));
        infile(f,openfile(s,f,wmode,x,q)) = ioerr;
        infile(f,initam) = ioerr;
        infile(f,close(d,q)) = ioerr;
        outfile(v,f,close(f,q)) = staterr;
        outfile(v,f,initam) = staterr;
        outfile(f,openfile(s,f,m,chardata,q)) = staterr;
        outfile(v,f,openfile(s,f,rmode,x,q)) = staterr;
        closefile(f,openfile(s,f,n,x,q)) = q;
        openfile(s,f,n,openfile(s,f,m,x,q)) = staterr;
    end extend;
end amstate;
```

```
spec amstate
is
    extend
        aminstructions,
        identifiers
    with
        sort
            state;
        primitive
        op
            initam: → state;
            storer: val,regaddr,state → state;
            fetchr: regaddr,state → val;
            storem: val,memaddr,state → state;
            fetchm: memaddr,state → val;
            initstk: stkaddr,state → state;
            topstk: stkaddr,state → val;
            pushstk: val,stkaddr,state → state;
            popstk: stkaddr,state → state;
            lalloc: nat,state → memid;
            lfree: memid,state → state;
            indir: nat,memaddr → memaddr;
            infile: file,state → val;
            outfile: val,file,state → state;
            openfile: str char,file,int,int,state → state;
            closefile: file,state → state;
            rmode: → int;
            wmode: → int;
            rwmode: → int;
            openerr: → int;
            openok: → int;
            valdata: → int;
            chardata: → int;
            undef: → val;
        error
        op
            ioerr: → val;
            staterr: → state;
            emptystkerr: → val;
            undflowstkerr: → state;
            nonallocerr: → val;
            accesserr: → memaddr;
        axiom
            implies(eqmemaddr(a1,a2),fetchm(a1,storem(v,a2,q)) = v)
                = true;
            implies(not(eqmemaddr(m1,m2)),fetchm(m1,storem(v,m2,q)) = fetchm(m1,q))
                = true;
            fetchm(m,initam) = undef;
            storem(fetchm(m,q),m,q) = q;
            implies(eqregaddr(r1,r1),fetchr(r1,storer(v,r2,q)) = v)
                = true;
            implies(not(eqregaddr(r1,r2)),fetchr(r1,storer(v,r2,q)) = fetchr(r2,q))
                = true;
            fetchr(r,initam) = undef;
            storer(fetchr(r,q),r,q) = q;
            topstk(s,pushstk(v,s,q)) = v;
            popstk(s,pushstk(v,s,q)) = q;
            topstk(s,initstk(s)) = emptystkerr;
            popstk(s,initstk(s)) = undflowstkerr;
            popstk(s,initam) = undflowstkerr;
            fetchm(offset(n,startmemaddr(lalloc(n1,q))),lfree(lalloc(n,q),q2))
```

```
        push_m: memaddr,stkaddr → instr;
        push_pcr: int,stkaddr → instr;
        push_ri: regaddr,stkaddr → instr;
        push_rid: regaddr,int,stkaddr → instr;
        push_ridn: regaddr,nat,int,stkaddr → instr;
        pushi: val,stkaddr → instr;
        pop_r: stkaddr,regaddr → instr;
        pop_m: stkaddr,memaddr → instr;
        pop_pcr: stkaddr,int → instr;
        pop_ri: stkaddr,regaddr → instr;
        pop_rid: stkaddr,regaddr,int → instr;
        pop_ridn: stkaddr,regaddr,nat,int → instr;
        popx: stkaddr → instr;
        jmp: memaddr → instr;
        jmp_ri: memaddr → instr;
        jmp_r: regaddr → instr;
        bra: int → instr;
        bra_r: regaddr → instr;
        if: relop,regaddr,regaddr,memaddr → instr;
        ifi: relop,regaddr,val,memaddr → instr;
        ifte: relop,regaddr,regaddr,memaddr,memaddr → instr;
        iftei: relop,regaddr,val,memaddr,memaddr → instr;
        if_pcr: relop,regaddr,regaddr,int → instr;
        ifi_pcr: relop,regaddr,val,int → instr;
        ifte_pcr: relop,regaddr,regaddr,int,int → instr;
        iftei_pcr: relop,regaddr,val,int,int → instr;
        test: bop,regaddr,memaddr → instr;
        testm: bop,memaddr,memaddr → instr;
        teste: bop,regaddr,memaddr,memaddr → instr;
        testme: bop,memaddr,memaddr,memaddr → instr;
        test_pcr: bop,regaddr,int → instr;
        testm_pcr: bop,memaddr,int → instr;
        teste_pcr: bop,regaddr,int,int → instr;
        testme_pcr: bop,memaddr,int,int → instr;
        stop → instr;
        jsr: memaddr,stkaddr → instr;
        jsr_ri: memaddr,stkaddr → instr;
        jsr_r: regaddr,stkaddr → instr;
        bsr: int,stkaddr → instr;
        bsr_r: regaddr,stkaddr → instr;
        rts: stkaddr → instr;
        link: regaddr,nat → instr;
        unlink: regaddr → instr;
        open: stkaddr → instr;
        close: stkaddr → instr;
        read: stkaddr → instr;
        write: stkaddr → instr;
        org: → instr;
        extern: → instr;
        globl: → instr;
        mbegin: → instr;
        mend: → instr;
    end extend;
end aminstructions;

newtype(memaddr,memaddresses);
newtype(regaddr,regaddresses);
newtype(stkaddr,stkaddresses);
newtype(file,files);
newtype(instr,aminstructions);
```

```
spec aminstructions
is
    extend
        amoperators,
        memaddresses,
        regaddresses,
        stkaddresses
    with
        sort
            instr;
        primitive
        op
            dyads: dop,regaddr,regaddr → instr;
            dyadsi: dop,val,regaddr → instr;
            dyad: dop,regaddr,regaddr,regaddr → instr;
            dyadi: dop,val,regaddr,regaddr → instr;
            monads: mop,regaddr → instr;
            monad: mop,regaddr,regaddr → instr;
            monadi: mop,val,regaddr → instr;
            offst: int,regaddr → instr;
            mov_m_m: memaddr,memaddr → instr;
            mov_pcr_pcr: int,int → instr;
            mov_ri_m: regaddr,memaddr → instr;
            mov_ri_pcr: regaddr,int → instr;
            mov_rid_m: regaddr,int,memaddr → instr;
            mov_rid_pcr: regaddr,int,int → instr;
            mov_ridn_m: regaddr,nat,int,memaddr → instr;
            mov_ridn_pcr: regaddr,nat,int,int → instr;
            mov_m_ri: memaddr,regaddr → instr;
            mov_pcr_ri: int,regaddr → instr;
            mov_m_rid: memaddr,regaddr,int → instr;
            mov_pcr_rid: int,regaddr,int → instr;
            mov_m_ridn: memaddr,regaddr,nat,int → instr;
            mov_pcr_ridn: int,regaddr,nat,int → instr;
            mov_ri_ri: regaddr,regaddr → instr;
            mov_rid_ri: regaddr,int,regaddr → instr;
            mov_ridn_ri: regaddr,nat,int,regaddr → instr;
            mov_ri_rid: regaddr,regaddr,int → instr;
            mov_ri_ridn: regaddr,regaddr,nat,int → instr;
            mov_rid_rid: regaddr,int,regaddr,int → instr;
            mov_ridn_rid: regaddr,nat,int,regaddr,int → instr;
            mov_rid_ridn: regaddr,int,regaddr,nat,int → instr;
            mov_ridn_ridn: regaddr,nat,int,regaddr,int,int → instr;
            movi_m: val,memaddr → instr;
            movi_pcr: val,int → instr;
            movi_ri: val,regaddr → instr;
            movi_rid: val,regaddr,int → instr;
            movi_ridn: val,regaddr,nat,int → instr;
            movi_r: val,regaddr → instr;
            mov_r_r: regaddr,regaddr → instr;
            mov_m_r: memaddr,regaddr → instr;
            mov_pcr_r: int,regaddr → instr;
            mov_ri_r: regaddr,regaddr → instr;
            mov_rid_r: regaddr,int,regaddr → instr;
            mov_ridn_r: regaddr,nat,int,regaddr → instr;
            mov_r_m: regaddr,memaddr → instr;
            mov_r_pcr: regaddr,int → instr;
            mov_r_ri: regaddr,regaddr → instr;
            mov_r_rid: regaddr,regaddr,int → instr;
            mov_r_ridn: regaddr,regaddr,nat,int → instr;
            push_r: regaddr,stkaddr → instr;
```

```
            valofS(gtS(atomofS v1,atomofS v2))"
        monadic(boolnot,not,bool);
        dyadic(booland,and,bool);
        dyadic(boolor,or,bool);
        dyadic(natsum,sumnat,nat);
        dyadic(intsum,sumint,int);
        dyadic(charstrlen,lenstr.char,str.char);
        dyadic(charconcat,catstr.char,str.char);
        relationalops(nat);
        relationalops(int);
        relationalops(char);
        relationalops(str.char);
    end extend;
end amoperators;
```

```
spec amoperators
is
    extend
        booltype,
        nattype,
        inttype,
        chartype,
        str.chartype
    with
        sort
            mop;
            dop;
            relop;
            bop;
        primitive
        op
            applymop: mop,val → val;
            applydop: dop,val,val → val;
            applyrel: relop,val,val → val;
            applybop: bop,val → val;
            boolnot: → mop;
            booland: → dop;
            boolor: → dop;
            natsum: → dop;
            intsum: → dop;
            charstrlen: → mop;
            charconcat: → dop;
            charmakestr: → mop;
            charheadstr: → mop;
            chartailstr: → mop;
            replace(S)
                "relationalops(S)"
            with
                "Sgt: → relop;
                Seq: → relop"
        error
        op
            operr: → val;
            moperr: → mop;
            doperr: → dop;
            reloperr: → relop;
        axiom
            applymop(m,typerr) = operr;
            applydop(d,v,typerr) = operr;
            applydop(d,typerr,v) = operr;
            replace(M,O,S)
                "monadic(M,O,S)"
            with
                "applymop(M,v) =
                    valofS(O(atomovS v))"
            replace(D,O,S)
                "dyadic(D,O,S)"
            with
                "applydop(D,v1,v2) =
                    valofS(O(atomofS v1,atomofS v2))"
            replace(S)
                "relations(S)"
            with
                "applyrel(Seq,v1,v2) =
                    valofS(eqS(atomofS v1,atomofS v2));
                applyrel(Sgt,v1,v2) =
```

```
spec regaddresses
is
    extend
        identifiers,
        boolean
    with
        sort
            regaddr;
        primitive
        op
            startregaddr: regid → regaddr;
            nextregaddr: regaddr → regaddr;
            eqregaddr: regaddr,regaddr → bool;
        axiom
            equivrel(eqregaddr,regaddr);
            eqregaddr(startregaddr(i1),startregaddr(i2)) = eqregid(i1,i2);
            eqregaddr(startregaddr(i),nextregaddr(a)) = false;
            eqregaddr(nextregaddr(a1),nextregaddr(a2)) = eqregaddr(a1,a2);
    end extend;
end regaddresses;


spec stkaddresses
is
    extend
        identifiers,
        boolean
    with
        sort
            stkaddr;
        primitive
        op
            eqstkaddr: stkaddr,stkaddr → bool;
            stkpointer: stkid → stkaddr;
        axiom
            equivrel(eqstkaddr,stkaddr);
            eqstkaddr(stkpointer(i1),stkpointer(i2)) = eqstkid(i1,i2);
    end extend;
end stkaddresses;


spec files
is
    extend
        identifiers,
        boolean
    with
        sort
            file;
        primitive
        op
            getfile: fid → file;
            eqfile: file,file → bool;
        axiom
            equivrel(eqfile,file);
            eqfile(getfile(i1),getfile(i2)) = eqfid(i1,i2);
    end extend;
end devaddresses;
```

```
spec identifiers
is
    sort
        memid;
        regid;
        stkid;
        fid;
    primitive
    op
        mem: → memid;
        reg: → regid;
        stk: → stkid;
        eqmemid: memid,memid → bool;
        eqregid: regid,regid → bool;
        eqstkid: stkid,stkid → bool;
        eqfid: fid,fid → bool;
    axiom
        equivrel(eqmemid,memid);
        equivrel(eqregid,regid);
        equivrel(eqstkid,stkid);
        equivrel(eqfid,fid);
end identifiers;


spec memaddresses
is
    extend
        identifiers,
        boolean
    with
        sort
            memaddr;
        primitive
        op
            startmemaddr: memid → memaddr;
            nextmemaddr: memaddr → memaddr;
            preamemaddr: memaddr → memaddr;
            eqmemaddr: memaddr,memaddr → bool;
            getmemid: memaddr → memid;
            offset: int,memaddr → memaddr;
        error
        op
            memaddrerr: → memaddr;
        axiom
            equivrel(eqmemaddr,memaddr);
            preamemaddr(nextmemaddr(m)) = m;
            preamemaddr(startmemaddr(i)) = memaddrerr;
            eqmemaddr(startmemaddr(i1),startmemaddr(i2)) = eqmemid(i1,i2);
            eqmemaddr(startmemaddr(i),nextmemaddr(a)) = false;
            eqmemaddr(nextmemaddr(a1),nextmemaddr(a2)) = eqmemaddr(a1,a2);
            offset(zeroint,m) = m;
            offset(succint(n),m) = nextmemaddr(offset(n,m));
            offset(predint(n),m) = preamemaddr(offset(n,m));
            eqmemid(i,getmemid(offset(n,startmemaddr(i))));
    end extend;
end memaddresses;
```

```
spec value
is
    sort
        val;
    error
    op
        typerr: → val;
end value;

newtype(bool,boolean);
newtype(int,integer);
newtype(nat,natural);
newtype(char,character);


spec charstring
is
    extend
        chartype
    with
        use
            string(character)
        where
            char is lm;
            eqchar is eqlm;
            gtchar is gtlm;
    end extend;
end charstring;


spec str.chartype
is
    extend
        charstring
    with
        primitive
        op
            atomofstr.char: val → str.char;
            valofstr.char: str.char → val;
        error
        op
            str.charerr: → str.char;
        axiom
            for x in val
                atomofstr.char(valofstr.char(x)) = x;
            atomofstr.char(typerr) = str.charerr;
    end extend;
end str.chartype;
```

```
spec string
parm
    extend
        boolean
    with
        sort
            lm;
        primitive
        op
            eqlm: lm,lm → bool;
            gtlm: lm,lm → bool;
        axiom
            equivrel(eqlm,lm);
            irreflexive(gtlm,lm);
            transitive(gtlm,lm);
    end extend;
is
    extend
        natural,
        boolean
    with
        sort
            str;
        primitive
        op
            nullstr: → str;
            makestr: lm → str;
            catstr: str,str → str;
            lenstr: str → nat;
            headstr: str → lm;
            tailstr: str → str;
            eqstr: str,str → bool;
            gtstr: str,str → bool;
        axiom
            lenstr(nullstr) = zeronat;
            lenstr(makestr(l)) = succnat(zeronat);
            lenstr(catstr(s1,s2)) = sumnat(lenstr(s1),lenstr(s2));
            headstr(makestr(l)) = l;
            tailstr(makestr(l)) = nullstr;
            headstr(catstr(makestr(l),s)) = l;
            tailstr(catstr(makestr(l),s2)) = s2;
            headstr(nullstr) = strerr;
            tailstr(nullstr) = nullstr;
            catstr(catstr(s1,s2),s3) = catstr(s1,catstr(s2,s3));
            catstr(nullstr,s) = catstr(s,nullstr) = s;
            equivrel(eqstr,str);
            irreflexive(gtstr,str);
            transitive(gtstr,str);
            implies(eqlm(l1,l2),eqstr(makestr(l1),makestr(l2))) = true;
            implies(gtlm(l1,l2),gtstr(makestr(l1),makestr(l2))) = true;
            gtnat(lenstr(makestr(l)),lenstr(nullstr)) = true;
            implies(gtnat(lenstr(s1),lenstr(s2)),gtstr(s1,s2)) = true;
            if not eqstr(lenstr(s1),zeronat) then
                gtnat(lenstr(catstr(s1,s2),lenstr(s2)) = true;
            else
                eqnat(lenstr(catstr(s1,s2),lenstr(s2)) = true;
            end if;
    end extend;
end string;
```

```
            gtchar(ENQ,EOT) = true;
            gtchar(EOT,ETX) = true;
            gtchar(ETX,STX) = true;
            gtchar(STX,SOH) = true;
            gtchar(SOH,NUL) = true;
    end extend;
end character;
```

```
gtchar('z',....,'a') = true;
gtchar('a','`') = true;
gtchar('`','_') = true;
gtchar('_','^') = true;
gtchar('^','[') = true;
gtchar('[',']') = true;
gtchar(']','Z') = true;
gtchar('Z',....,'A') = true;
gtchar('A','@') = true;
gtchar('@','?') = true;
gtchar('?','>') = true;
gtchar('>','=') = true;
gtchar('=','<') = true;
gtchar('<',';') = true;
gtchar(';',':') = true;
gtchar(':','9') = true;
gtchar('9',....,'0') = true;
gtchar('0','/') = true;
gtchar('/','.') = true;
gtchar('.','-') = true;
gtchar('-',',') = true;
gtchar(',','+') = true;
gtchar('+','*') = true;
gtchar('*',')') = true;
gtchar(')','(') = true;
gtchar('(',''') = true;
gtchar(''','&') = true;
gtchar('&','%') = true;
gtchar('%','$') = true;
gtchar('$','#') = true;
gtchar('#','"') = true;
gtchar('"','!') = true;
gtchar('!',SP) = true;
gtchar(SP,US) = true;
gtchar(US,RS) = true;
gtchar(RS,GS) = true;
gtchar(GS,FS) = true;
gtchar(FS,ESC) = true;
gtchar(ESC,SUB) = true;
gtchar(SUB,EM) = true;
gtchar(EM,CAN) = true;
gtchar(CAN,ETB) = true;
gtchar(ETB,SYN) = true;
gtchar(SYN,NAK) = true;
gtchar(NAK,DC4) = true;
gtchar(DC4,DC3) = true;
gtchar(DC3,DC2) = true;
gtchar(DC2,DC1) = true;
gtchar(DC1,DLE) = true;
gtchar(DLE,SI) = true;
gtchar(SI,SO) = true;
gtchar(SO,CR) = true;
gtchar(CR,FF) = true;
gtchar(FF,VT) = true;
gtchar(VT,LF) = true;
gtchar(LF,HT) = true;
gtchar(HT,BS) = true;
gtchar(BS,BEL) = true;
gtchar(BEL,ACK) = true;
gtchar(ACK,ENQ) = true;
```

```
spec character
is
    extend
        boolean
    with
        sort
            char;
        primitive
        op
            'A','B','C',...,'Z': → char;
            'a','b','c',...,'z': → char;
            '!','@','#','$','%','^','&','*','(',')': → char;
            '-','_','+','=','~','`','{','}','[',']': → char;
            '''','"',':',';',',','.','<','>','?','/': → char;
            ' ',': → char;
            '1','2','3','4','5','6','7','8','9','0': → char;
            NUL: → char;
            SOH: → char;
            STX: → char;
            ETX: → char;
            EOT: → char;
            ENQ: → char;
            ACK: → char;
            BEL: → char;
            BS: → char;
            HT: → char;
            LF: → char;
            VT: → char;
            FF: → char;
            CR: → char;
            SO: → char;
            SI: → char;
            DLE: → char;
            DC1: → char;
            DC2: → char;
            DC3: → char;
            DC4: → char;
            NAK: → char;
            SYN: → char;
            ETB: → char;
            CAN: → char;
            EM: → char;
            SUB: → char;
            ESC: → char;
            FS: → char;
            GS: → char;
            RS: → char;
            US: → char;
            SP: → char;
            DEL: → char;
            eqchar: char,char → bool;
            gtchar: char,char → bool;
        axiom
            equivrel(eqchar,char);
            irreflexive(gtchar,char);
            transitive(gtchar,char);
            gtchar('DEL','~') = true;
            gtchar('~','}') = true;
            gtchar('}','`') = true;
            gtchar('`','{') = true;
            gtchar('{','z') = true;
```

```
isops(string.char);
isops(memaddr);
isops(instr);
isops(file);
equivrel(eqtype,type);
implies(
    eqtype(whattype(v1),whattype(v2)) = false,
    applyrel(inteq,v1,v2) = valofbool(false)
    ) = true;
implies(
    eqtype(whattype(v1),whattype(v2)) = false,
    applyrel(nateq,v1,v2) = valofbool(false)
    ) = true;
implies(
    eqtype(whattype(v1),whattype(v2)) = false,
    applyrel(chareq,v1,v2) = valofbool(false)
    ) = true;
implies(
    eqtype(whattype(v1),whattype(v2)) = false,
    applyrel(string.chareq,v1,v2) = valofbool(false)
    ) = true;
cond(valofbool(true),a1,a2) = a1;
cond(valofbool(false),a1,a2) = a2;

prog(a,q) = xeq(atomofinstr(fetchm(a,q),a,q));

xeq(dyads(o,r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applydop(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            r2,
            q
        )
    );
xeq(dyadsi(o,v,r1),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applydop(
                o
                v,
                fetchr(r1,q)
            ),
            r1,
            q
        )
    );
xeq(dyad(o,r1,r2,r3),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applydop(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
```

```
            r3,
            q
        )
    );
xeq(dyadi(o,v,r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applydop(
                o,
                v,
                fetchr(r1,q)
            ),
            r2,
            q
        )
    );
xeq(monads(o,r1),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applymop(
                o,
                fetchr(r1,q)
            ),
            r1,
            q
        )
    );
xeq(monad(o,r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applymop(
                o,
                fetchr(r1,q)
            ),
            r2,
            q
        )
    );
xeq(monadi(o,v,r1),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applymop(o,v),
            r1,
            q
        )
    );
xeq(offst(i,r),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            valofmemaddr(
                offset(
                    i,
                    atomofmemaddr(
                        fetchr(r,q)
                    )
                )
```

```
            ),
            r,
            q
        )
    );
xeq(mov_m_m(m1,m2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(m1,q),
            m2,
            q
        )
    );
xeq(mov_pcr_pcr(i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(i1,m),
                q
            ),
            offset(i2,m),
            q
        )
    );
xeq(mov_ri_m(r,m1),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                atomofmemaddr(
                    fetchr(r,q)
                ),
                q
            ),
            m1,
            q
        )
    );
xeq(mov_ri_pcr(r,i),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                atomofmemaddr(
                    fetchr(r,q)
                ),
                q
            ),
            offset(i,m),
            q
        )
    );
xeq(mov_rid_m(r,i,m1),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i,
```

```
                atomofmemaddr(
                    fetchr(r,q)
                )
            ),
            q
        ),
        m1,
        q
    )
);
xeq(mov_rid_pcr(r,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i1,
                    atomofmemaddr(
                        fetchr(r,q)
                    )
                ),
                q
            ),
            offset(i2,m),
            q
        )
    );
xeq(mov_ridn_m(r,n,i,m1),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i,
                    indir(
                        n,
                        atomofmemaddr(
                            fetchr(r,q)
                        )
                    )
                ),
                q
            ),
            q
        )
    );
xeq(mov_ridn_pcr(r,n,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i1,
                    indir(
                        n,
                        atomofmemaddr(
                            fetchr(r,q)
                        )
                    )
                ),
                q
```

```
            ),
            offset(i2,m),
            q
        )
    );
xeq(mov_m_ri(m1,r),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(m1,q),
            atomofmemaddr(fetchr(r,q))
            ,q
        )
    );
xeq(mov_pcr_ri(i,r),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(i,m),
                q
            ),
            atomofmemaddr(
                fetchr(r,q)
            ),
            q
        )
    );
xeq(mov_m_rid(m1,r,n),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(m1,q),
            offset(
                n,
                atomofmemaddr(
                    fetchr(r,q)
                )
            ),
            q
        )
    );
xeq(mov_pcr_rid(i1,r,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(i1,m),
                q
            ),
            offset(
                i2,
                atomofmemaddr(
                    fetchr(r,q)
                )
            ),
            q
        )
    );
xeq(mov_m_ridn(m1,r,i1,i2),m,q) =
    prog(
```

```
        nextmemaddr(m),
        storem(
            fetchm(m1,q),
            offset(
                i2,
                indir(
                    i1,
                    atomofmemaddr(
                        fetchr(r,q)
                    )
                )
            ),
            q
        )
    );
xeq(mov_pcr_ridn(i1,r,n,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(i1,m),
                q
            ),
            offset(
                i2,
                indir(
                    n,
                    atomofmemaddr(
                        fetchr(r,q)
                    )
                )
            ),
            q
        )
    );
xeq(mov_ri_ri(r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                atomofmemaddr(
                    fetchr(r1,q)
                ),
                q
            ),
            atomofmemaddr(
                fetchr(r2,q)
            ),
            q
        )
    );
xeq(mov_rid_ri(r1,i,r2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i,
                    atomofmemaddr(
                        fetchr(r1,q)
                    )
```

```
                ),
                q
            ),
            atomofmemaddr(
                fetchr(r2,q)
            ),
            q
        )
    );
xeq(mov_ridn_ri(r1,i1,i2,r2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(
                            fetchr(r1,q)
                        )
                    )
                ),
                q
            ),
            atomofmemaddr(
                fetchr(r2,q)
            ),
            q
        )
    );
xeq(mov_ri_rid(r1,r2,n),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                atomofmemaddr(
                    fetchr(r1,q)
                ),
                q
            ),
            offset(
                n,
                atomofmemaddr(
                    fetchr(r2,q)
                )
            ),
            q
        )
    );
xeq(mov_ri_ridn(r1,r2,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                atomofmemaddr(
                    fetchr(r1,q)
                ),
                q
            ),
            offset(
```

```
            i2,
            indir(
                i1,
                atomofmemaddr(
                    fetchr(r2,q)
                )
            )
        ),
        q
    )
);
xeq(mov_rid_rid(r1,i1,r2,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i1,
                    atomofmemaddr(
                        fetchr(r1,q)
                    )
                ),
                q
            ),
            offset(
                i2,
                atomofmemaddr(
                    fetchr(r2,q)
                )
            ),
            q
        )
    );
xeq(mov_ridn_rid(r1,i1,i2,r2,i3),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(
                            fetchr(r1,q)
                        )
                    )
                ),
                q
            ),
            offset(
                i3,
                atomofmemaddr(
                    fetchr(r2,q)
                )
            ),
            q
        )
    );
xeq(mov_rid_ridn(r1,i1,r2,i2,i3),m,q) =
    prog(
        nextmemaddr(m),
```

```
        storem(
            fetchm(
                offset(
                    i1,
                    atomofmemaddr(
                        fetchr(r1,q)
                    )
                ),
                q
            ),
            offset(
                i3,
                indir(
                    i2,
                    atomofmemaddr(
                        fetchr(r2,q)
                    )
                )
            ),
            q
        )
    );
xeq(mov_ridn_ridn(r1,i1,i2,r2,i3,i4),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(
                            fetchr(r1,q)
                        )
                    )
                ),
                q
            ),
            offset(
                i4,
                indir(
                    i3,
                    atomofmemaddr(
                        fetchr(r2,q)
                    )
                )
            ),
            q
        )
    );
xeq(movi_m(v,m1),m,q) =
    prog(
        nextmemaddr(m),
        storem(v,m1,q)
    );
xeq(movi_pcr(v,i),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            v,
            offset(i,m),
```

```
            q
        )
    );
xeq(movi_ri(v,r),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            v,
            atomofmemaddr(
                fetchr(r,q)
            ),
            q
        )
    );
xeq(movi_rid(v,r,n),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            v,
            offset(
                n,
                atomofmemaddr(
                    fetchr(r,q)
                )
            ),
            q
        )
    );
xeq(movi_ridn(v,r,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            v,
            offset(
                i2,
                indir(
                    i1,
                    atomofmemaddr(
                        fetchr(r,q)
                    )
                )
            ),
            q
        )
    );
xeq(movi_r(v,r),m,q) =
    prog(nextmemaddr(m),storer(v,r,q));
xeq(mov_r_r(r1,r2),m,q) =
    prog(nextmemaddr(m),storer(fetch(r1,q),r2,q));
xeq(mov_m_r(m1,r),m,q) =
    prog(nextmemaddr(m),storer(fetchm(m1,q),r,q));
xeq(mov_pcr_r(i,r),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            fetchm(offset(i,m),q),
            r,
            q
        )
    );
xeq(mov_ri_r(r1,r2),m,q) =
```

```
    prog(
        nextmemaddr(m),
        storer(
            fetchm(
                atomofmemaddr(
                    fetchr(r1,q)
                )
            ),
            r2,
            q
        )
    );
xeq(mov_rid_r(r1,n,r2),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            fetchm(
                offset(
                    n,
                    atomofmemaddr(
                        fetchr(r1,q)
                    )
                ),
                q
            ),
            r2,
            q
        )
    );
xeq(mov_ridn_r(r1,i1,i2,r2),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(
                            fetchr(r1,q)
                        )
                    )
                ),
                q
            ),
            r2,
            q
        )
    );
xeq(mov_r_m(r,m1),m,q) =
    prog(nextmemaddr(m),storem(fetchr(r,q),m1,q));
xeq(mov_r_pcr(r,i),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchr(r,q),
            offset(i,m),
            q
        )
    );
xeq(mov_r_ri(r1,r2),m,q) =
```

```
    prog(
        nextmemaddr(m),
        storem(
            fetchr(r1,q),
            atomofmemaddr(
                fetchr(r2,q)
            ),
            q
        )
    );
xeq(mov_r_rid(r1,r2,n),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchr(r1,q),
            offset(
                n,
                atomofmemaddr(
                    fetchr(r2,q)
                )
            ),
            q
        )
    );
xeq(mov_r_ridn(r1,r2,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchr(r1,q),
            offset(
                i2,
                indir(
                    i1,
                    atomofmemaddr(
                        fetchr(r2,q)
                    )
                )
            ),
            q
        )
    );
xeq(push_r(r,s),m,q) =
    prog(nextmemaddr(m),pushstk(fetchr(r,q),s,q));
xeq(push_m(m1,s),m,q) =
    prog(nextmemaddr(m),pushstk(fetchm(m1,q),s,q));
xeq(push_pcr(i,s),m,q) =
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(
                offset(i,m),
                q
            ),
            s,
            q
        )
    );
xeq(push_ri(r,s),m,q) =
    prog(
        nextmemaddr(m),
        pushstk(
```

```
            fetchm(
                atomofmemaddr(
                    fetchr(r,q)
                ),
                q
            ),
            s,
            q
        )
    );
xeq(push_rid(r,n,s),m,q) =
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(
                offset(
                    n,
                    atomofmemaddr(
                        fetchr(r,q)
                    )
                ),
                q
            ),
            s,
            q
        )
    );
xeq(push_ridn(r,i1,i2,s),m,q) =
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(
                            fetchr(r,q)
                        )
                    )
                ),
                q
            ),
            s,
            q
        )
    );
xeq(pushi(v,s),m,q) =
    prog(nextmemaddr(m),pushstk(v,s,q));
xeq(pop_r(s,r),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storer(
                topstk(s,q),
                r,
                q
            )
        )
    ),
```

```
xeq(pop_m(s,m1),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                m1,
                q
            )
        )
    );
xeq(pop_pcr(s,i),m,q) =
    prog(
        nextmemaddr(m),
        poptsk(
            s,
            storem(
                topstk(s,q),
                offset(i,m),
                q
            )
        )
    );
xeq(pop_ri(s,r),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                atomofmemaddr(
                    fetchr(r,q)
                ),
                q
            )
        )
    );
xeq(pop_rid(s,r,n),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                offset(
                    n,
                    atomofmemaddr(
                        fetchr(r,q)
                    )
                ),
                q
            )
        )
    );
xeq(pop_ridn(s,r,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
```

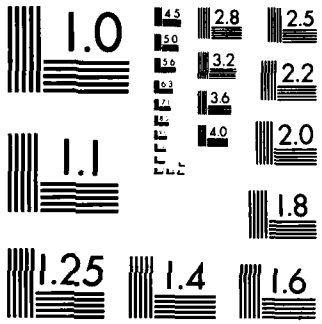
AD-A155 167 THE FORMAL SPECIFICATION OF AN ABSTRACT MACHINE: DESIGN AND IMPLEMENTATION(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA J M YURCHAK DEC 84 2/2

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
                topstk(s,q),
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(
                            fetchr(r,q)
                        )
                    )
                ),
                q
            )
        )
    );
xeq(popx(s),m,q) =
    prog(nextmemaddr(m),popstk(s,q));
xeq(jmp(m1),m,q) =
    prog(m1,q);
xeq(jmp_mi(m1),m,q) =
    prog(atomofmemaddr(fetchm(m1,q)),q);
xeq(jmp_r(r),m.q) =
    prog(atomofmemaddr(fetchr(r,q)),q);
xeq(bra(n),m,q) =
    prog(offset(n,nextmemaddr(m)),q);
xeq(bra_r r,m,q) =
    prog(offset(atomofint(fetchr(r,q)),nextmemaddr(m)),q);
xeq(if(o,r1,r2,m1),m,q) =
    prog(
        cond(
            applyrel(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            m1,
            nextmemaddr(m)
        ),
        q
    );
xeq(ifi(o,r,v,m1),m,q) =
    prog(
        cond(
            applyrel(
                o,
                fetchr(r,q),
                v
            ),
            m1,
            nextmemaddr(m)
        ),
        q
    );
xeq(ifte(o,r1,r2,m1,m2),m,q) =
    prog(
        cond(
            applyrel(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            m1,
```

```
            m2
        ),
        q
    );
xeq(iftei(o,r,v,m1.m2),m,q) =
    prog(
        cond(
            applyrel(
                o,
                fetchr(r,q),
                v
            ),
            m1,
            m2
        ),
        q
    );
xeq(if_pcr(o,r1,r2,n),m,q) =
    prog(
        cond(
            applyrel(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m)
        ),
        q
    );
xeq(ifi_pcr(o,r,v,n),m,q) =
    prog(
        cond(
            applyrel(
                o,
                fetchr(r,q),
                v
            ),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m)
        ),
        q
    );
xeq(ifte_pcr(o,r1,r2,i1,i2),m,q) =
    prog(
        cond(
            applyrel(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            offset(i1,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
        ),
        q
    );
xeq(iftei_pcr(o,r,v,i1,i2),m,q) =
    prog(
        cond(
            applyrel(
                o,
```

```
                fetchr(r,q),
                v
            ),
            offset(i1,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
        ),
        q
    );
xeq(test(o,r1,m1),m,q) =
    prog(
        cond(
            applybop(o,fetchr(r1,q)),
            m1,
            nextmemaddr(m)
        ),
        q
    );
xeq(testm(o,m2,m1),m,q) =
    prog(
        cond(
            applybop(o,fetchm(m2,q)),
            m1,
            nextmemaddr(m)
        ),
        q
    );
xeq(teste(o,r1,m1,m2),m,q) =
    prog(cond(applybop(o,fetchr(r1,q)),m1,m2),q);
xeq(testme(o,m3,m1,m2),m,q) =
    prog(cond(applybop(o,fetchm(m3,q)),m1,m2),q);
xeq(test_pcr(o,r1,n),m,q) =
    prog(
        cond(
            applybop(o,fetchr(r1,q)),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m);
        ),
        q
    );
xeq(testm_pcr(o,m2,n),m,q) =
    prog(
        cond(
            applybop(o,fetchm(m2,q)),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m)
        ),
        q
    );
xeq(teste_pcr(o,r1,i1,i2),m,q) =
    prog(
        cond(
            applybop(o,fetchr(r1,q)),
            offset(i1,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
        ),
        q
    );
xeq(testme_pcr(o,m3,i1,i2),m,q) =
    prog(
        cond(
            applybop(o,fetchm(m3,q)),
```

```
            offset(i1,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
        ),
        q
    );
xeq(stop,m,q) = prog(m,q) = q;
xeq(jsr(m1,s),m,q) =
    prog(m1,pushstk(valofmemaddr(nextmemaddr(m)),s,q));
xeq(jsr_mi(m1,s),m,q) =
    prog(
        atomofmemaddr(fetchm(m1,q)),
        pushstk(valofmemaddr(nextmemaddr(m)),s,q)
    );
xeq(jsr_r(r,s),m,q) =
    prog(
        atomofmemaddr(fetchr(r,q)),
        pushstk(valofmemaddr(nextmemaddr(m)),s,q)
    );
xeq(bsr(n,s),m,q) =
    prog(
        offset(n,nextmemaddr(m)),
        pushstk(valofmemaddr(nextmemaddr(m)),s,q)
    );
xeq(bsr_r(r,s),m,q) =
    prog(
        offset(
            atomofint(fetchr(r,q)),
            nextmemaddr(m)
        ),
        pushstk(valofmemaddr(nextmemaddr(m)),s,q)
    );
xeq(rts s,m,q) =
    prog(atomofmemaddr(topstk(s,q)),popstk(s,q));
xeq(link(r,n),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            valofmemaddr(
                startmemaddr(lalloc(n,q))
            ),
            r,
            storem(
                fetchr(r,q),
                startmemaddr(lalloc(n,q),q)
            )
        )
    );
xeq(unlink(r),m,q) =
    prog(
        nextmemaddr(m),
        lfree(
            getmemid(
                atomofmemaddr(fetchr(r,q))
            ),
            storer(
                fetchm(
                    atomofmemaddr(fetchr(r,q)),
                    q
                ),
                r,
            q
```

```
            )
        );
    xeq(open(s),m,q) =
        prog(
            nextmemaddr(m),
            openfile(
                atomofstr.char(
                    topstk(s,popstk(s,popstk(s,popstk(s,q))))
                ),
                atomoffile(topstk(s,popstk(s,popstk(s,q)))),
                atomofint(topstk(s,popstk(s,q))),
                atomofint(topstk(s,q)),
                popstk(s,q)
            )
        );
    xeq(close(s),m,q) =
        prog(
            nextmemaddr(m),
            closefile(
                atomoffile(topstk(s,q)),
                popstk(s,q)
            )
        );
    xeq(read(s),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                infile(
                    atomoffile(topstk(s,popstk(s,q))),
                    popstk(s,q)
                ),
                atomofmemaddr(topstk(s,q)),
                popstk(s,q)
            )
        );
    xeq(write(s),m,q) =
        prog(
            nextmemaddr(m),
            outfile(
                fetchm(
                    atomofmemaddr(topstk(s,popstk(s,q))),
                    popstk(s,q)
                ),
                atomoffile(topstk(s,q)),
                popstk(s,q)
            )
        );
end extend;
end am;
```

APPENDIX C: A SIMPLE ASSEMBLER FOR AM

1. Introduction

AMASM is an assembler which produces a relocatable load module for AM, an abstract machine interpreter. This document constitutes the reference manual for Version 1.0. It provides a description of the syntax and semantics of the assembler as well as a description of the salient features of the AM machine and a definition of the opcodes executed by AM.

AMASM is, to the extent possible, written in portable C. Readers desiring to port the code to 16-bit machines may have to make slight changes to "defines" since **long** is assumed to occupy 32 bits, and **short** 16 bits.

The input syntax of AMASM is similar to that of other assemblers. It supports symbolic addresses and constants and a typical set of directives, but has no macro capabilities. The assembler accepts an ASCII source file created on a conventional text editor and produces an output file containing relocation information and AM opcodes. The output file may be loaded using the AM loader and executed by AM.

2. Usage

AMASM is invoked with the following command line syntax:

amasm [-t] [-l] *file* ...

AMASM produces a single load module "a.vm", which forms the input to the AM loader. The optional "-t" switch sends debugging trace to "stdout". The optional "-l" switch generates the listing and crossreference file "a.x". Appended to this file is a hex dump of "a.vm".

3. Lexical Conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), literal constants, operators and delimiters.

3.1. Identifiers

Legal identifiers are described by the following regular expression:

[A-Za-z_][A-Za-z0-9_]*

Identifiers consist of a letter or underline "_" followed by a string of zero or more letters, decimal digits and underlines. Upper and lower case are distinct. Identifiers may represent symbolic constants, instruction mnemonics, labels, addresses and type names.

3.2. Operators

The following are considered to be operators:

== != < <= > >=
+ - * / % & |

The meaning of the above symbols varies with context.

3.3. Literal Constants

Decimal and hexadecimal constants are described by the following regular expressions respectively:

```
[-+][0-9]+ | [0-9]+
$[0-9A-Za-z]+
```

Decimal constants consist of an optional sign followed immediatly by one or more decimal digits. Hexadecimal constants consist of the character "$" followed immediately by a string of one or more decimal digits and upper or lower case letters "A" through "F". Numeric constants may represent addresses, integer and natural numbers, boolean and character values.

Character constants consist of a single quote "'", followed either by an ASCII character not a newline or a numeric constant, followed by a closing single quote.

String constants consist of a string of zero or more ASCII characters (except newline) enclosed in double quotes.

3.4. Blanks

Blanks and tabs are ignored by the assembler except where required to separate adjacent constants or identifiers.

3.5. Comments

The character ";" produces a comment. The assembler ignores all further characters on the line up to the terminating newline.

3.6. Delimiters

All other characters found in the input stream are treated as delimiters.

4. Statements

A source program is composed of a sequence of statements which are separated by newlines. There are 3 kinds of statements: directives, instructions and null.

Instructions and null statements may be preceded by a label. Directives may (in some cases, must) be preceded by an identifier.

4.1. Labels & Identifiers

A label consists of an identifier followed by a colon ":". When the assembler encounters a label, the effect is to assign the current value of the location counter to the name.

An identifier preceding a directive is assigned a value whose type depends upon the directive. For instance, the **equate** directive assigns a typed value to an identifier, while the **define storage** directive assigns the current value of the location counter.

Neither labels nor identifiers may be redefined within a single source file.

4.2. Null Statements

A null statement is an empty statement. Although ignored by the assembler, null statements may be preceded by a label.

4.3. Directive Statements

A directive is a command to the assembler to perform some sort of operation which does not involve emitting an executable instruction. Typical directives (also known as "pseudo ops" or "pseudo instructions") allocate storage for variables, make names within the current module visible to other modules and set the location counter. Directives also produce instructions for the AM linker and loader.

Directives consist of a keyword followed by zero or more arguments, depending upon the context. Directives and their syntax are described in more detail in Section 11.

4.4. Instruction Statements

Instruction statements produce the code which is ultimately executed by AM. An instruction may be preceded by a label, and consists of a keyword followed by zero or more arguments, depending upon context.

The AM instruction set and its syntax will be described in detail in Section 13.

5. The Machine

Because AM differs from conventional machines in a number of important ways, some discussion is necessary before introducing the instruction set. Outwardly similar to a number of well known examples, AM instructions form an unconventional set of primitive operations which implement a formally specified semantics. The reasons for this are described below.

AM uses a tagged architecture. Thus, each data element contains, within it, information which uniquely identifies a finite set of legal operations which may be performed upon it, as well as a range of legal values it may take on. This set of operations and values is known formally as a **data type**. AM supports a number of data types. An element of a particular data type will be referred to throughout the rest of this manual as an **atom**.

AM physical resources are partitioned into **segments**. There are several types of segments, and these together form a conventional overall model of the familiar stored program computer. There are memory segments (primary storage), register segments (high-speed memory), stacks, and file segments (secondary storage). Segments are further partitioned into discrete, addressable elements (alternatively, "cells") which will contain atoms during the execution of a program. These elements will be referred to repeatedly as **typed values**. The reason for the distinction between atoms and values will become more clear shortly.

AM is the finite implementation of a formal specification. As such, data elements and the operations which can be applied to them must reflect a mathematical consistency not required by conventional architectures. Since all operations which affect the state of the machine must be able to "communicate" with each other during the execution of a AM program, they must do so using a common object. This object is a value. The memory, the registers, the stack, the files all hold values. Store, fetch, execute, read, write -- any operations which

change the state of the machine -- all operate on values (i.e., storage cells). All other operations, such as "add", "multiply", "and", "or", work on atoms. Atomic operations in AM correspond to those which take place in the temporary registers of the arithmetic and logic unit of a conventional processor.

5.1. Configuration

A unique feature of AM is the ease with which it is possible to reconfigure the machine by partitioning the physical resources in different ways. A typical configuration would be something like this:

2 memory segments
1 register segment (with a useful number of registers)
1 stack
4 files

The configuration chosen should provide a good indication of the types of programs AM is intended to execute.

Note that, in conventional machines, stacks are implemented in primary storage. This constitutes an overloading of data structures which obscures the intent of the user of these structures. It also creates a semantic nightmare for the specification writer. In AM, stacks take their rightful places as separate entities with easy to understand properties.

In addition to the resources listed above, AM has a conventional program counter.

5.1.1. Memory

AM memory is partitioned into segments which may be of unequal but fixed length. A program and its data will reside in memory segments. It is not necessary that code and data share the same segment, nor is it required that code and data be contiguous. The loader will determine from the **origin** directive where to load code and data values.

The AM heap is implemented as a set of operations which allocate and deallocate memory segments.

AM has a rich set of addressing modes which interact with a powerful move instruction which allows the programmer to move a value from "anywhere to anywhere".

5.1.2. Registers

AM registers form the high-speed storage into which operands are placed.

All atomic operations, such as add and divide, require operands to be in registers.

5.1.3. Stack

The AM stack is conventional in every respect except that it is impossible to access any value except the top. Thus, frames are implemented on the heap, not the stack.

AM has a typical set of push and pop instructions for operating on stacks.

5.1.4. Files

Input/output is implemented rather arbitrarily along the lines of system calls to an operating system and should not be considered part of AM itself.

Instructions are provided to open, close, read to and write from a file.

6. Atoms

An atom is a component of a data type. The assembler recognizes the following types of atoms:

boolean
natural
integer
character
string
memory address
register address
stack address
file address

As operands to instruction mnemonics, these atoms form the familiar set of literal and symbolic constants found in typical assembly language programs.

Atoms may appear in the form literal constants:

100
$d0f1
´a´
"this is a string atom"

They may also appear as symbols which take on the value of the atom in some other part of the source program. With few exceptions, anywhere a literal constant may be used, a symbolic constant of the appropriate type may also be used.

The assembler distinguishes between types of atom using syntax and context. The syntax is described below.

6.1. Boolean

A boolean atom has only two values, *true* and *false*. These values are represented to the assembler by the decimal or hexadecimal constants for 1 and 0, respectively.

0
1
$1
$0

are legal boolean atoms.

6.2. Natural

This type represents, as the name implies, the natural (unsigned) numbers. Legal values range from zero to positive infinity. Natural numbers are represented to the assembler as decimal or hexadecimal constants whose values

are greater than or equal to zero.

```
0
$2f5
240
```

are legal natural atoms.

6.3. Integer

Integers range from negative to positive infinity, and are specified as hexadecimal or signed or unsigned decimal constants.

```
-250
0
$ed67f
+10
```

are legal integer atoms.

6.4. Character

Character atoms may take values defined by the ASCII character set. They are represented to the assembler as literal character constants.

```
´a´
´r´
```

are legal character atoms.

6.5. String

String atoms are composed of zero or more concatenated ASCII characters. They are specified as literal strings.

```
"this is a legal string atom"
""
```

are both legal string atoms.

6.6. Memory Address

Memory address atoms consist of two components: a segment address, and an element address. Memory addresses are represented as an ordered pair of unsigned decimal or hexadecimal constants, separated by a colon ":" and enclosed within parentheses "(" ")".

```
(0:100)
```

represents memory segment 0, element 100.

```
(2:$10)
```

represents segment 2, element 16.

Segment and element addresses start at 0. The number and size of available memory segments depends upon the current configuration of AM.

Labels are considered memory address atoms, as are names which appear to left of the **define storage** and **define constant** directives.

6.7. Register Addresses

Register atoms have a syntax identical to that of memory addresses except that a lower case "r" is prepended to the address.

r(0:3)

refers to register segment 0, register 3.

Segment and element addresses start, as with memory addresses, at 0. The number of register segments, and the number of registers within each segment, varies as determined by the current AM configuration.

6.8. Stack Addresses

A stack address has only one component: the segment address. Stack addresses are specified by prepending a lower case "s" to an unsigned decimal or hexadecimal constant enclosed within parentheses.

s(2)

refers to stack segment 2.

Stack addresses begin at 0. The number of stacks depends upon AM's configuration.

6.9. File Addresses

File address atoms may not appear in a program except within typed values. File address atoms are represented as unsigned integer or hexadecimal constants.

File addresses start at 0. The number of files which may be open at one time is determined by the current AM configuration. The first three file addresses (0,1,2) are normally opened automatically by AM when a program is loaded.

7. Typed Values

Some of the atomic types may also appear as typed values in certain instructions and directives. A typed (immediate) value is represented as an ordered pair consisting of a keyword representing the type, and the atom itself, separated by a comma "," and enclosed within curly braces "{""}".

{int,100}

represents the integer value 100.

{addr,(1:100)}

represents memory address value (1:100).

A list of the types which may be used as immediate values alongside the corresponding keywords appears below:

bool - boolean
nat - natural
int - integer
char - character
string - character string
addr - memory address
file - file address

12.9. Immediate Atom

The operand is an atom.

Syntax: A

A - usually an integer or natural

Format:

tag	val

12.10. Stack Direct

The operand is a stack.

Syntax: S

Format:

000B	HHHHHHHH

13. Instruction Set

The AM instruction set is simple but powerful. The rigid data types make it meaningless to specify operations like shift and mask, thus removing some of the programmer's freedom to muck with data in arbitrary ways. The tagged architecture will detect errors like jumping to data, or accessing instructions as data, as well as the more common bounds checking performed by runtime libraries.

13.1. Machine Errors

The following errors are detected by AM during loading and execution:

- attempt to execute a non-instruction
- attempt to execute an illegal instruction
- memory segment not defined
- memory segment overflow
- memory segment underflow
- register segment not defined
- register segment underflow
- register segment underflow
- stack segment not defined
- <file> contains unresolved references
- attempt to convert negative int to nat
- no predecessor to zeronat
- unknown operator to applybop
- unknown operator to applymop
- unknown operator to applydop
- unknown operator to applyrelop
- type error - GT
- type error - GE
- type error - LT
- type error - LE
- no more segment available

Syntax: RN@I

R - holds the current frame pointer
N - a non-negative frame reference
I - an integer frame displacement

(R0@I is equivalent to R@I)

Format:

000A	HHHHHHHH	0002	HHHHHHHH	0003	HHHHHHHH

12.5. Memory Absolute

Syntax: M

M - the operand address

Format:

0009	HHHHHHHH

12.6. Memory Indirect

The address of the operand is in a memory cell.

Syntax: M@

M - a pointer to the operand address

Format:

0009	HHHHHHHH

12.7. Program Counter Relative

The address of the operand is the sum of the program counter and an integer displacement.

Syntax: M

M - the operand address

The specified address must be in the same module as the instruction. The assembler automatically computes the displacement. Program counter relative is specified for a block by placing a **rorg** directive at the top of the block.

Format:

0003	HHHHHHHH

12.8. Immediate Value

The operand is an immediate value.

Syntax: V

V - any typed value

Format:

tag	val

12. Addressing Modes

AM supports 10 addressing modes:

r	- register direct
ri	- register indirect
rid	- register indirect with displacement
ridn	- n-level register indirect with displacement
m	- memory absolute
mi	- memory indirect
pcr	- program counter relative
i	- immediate value
a	- immediate atom
s	- stack direct

Like other more familiar processors, not all AM instructions can use all of the addressing modes.

In addition, AMASM supports address expressions, which provides a rudimentary indexing capability.

12.1. Register Direct

The operand is in a register.

Syntax: R

Format:

000A	HHHHHHHH

12.2. Register Indirect

The address of the operand is in a register.

Syntax: R@

R - holds the operand address

Format:

000A	HHHHHHHH

12.3. Register Indirect with Displacement

The address of the operand is the sum of the address in a register and an integer displacement.

Syntax: R@I

R - holds a base address
I - an integer displacement

Format:

000A	HHHHHHHH	0003	HHHHHHHH

12.4. N-level Register Indirect with Displacement

The address of the operand is the sum of the address obtained from the nth link in a chain of dynamic links and an integer displacement.

Syntax:

extern <name>...

where:
<name> is any legal identifier

Description:

The list of symbols is made visible to the current module and are assumed to be defined elsewhere. An error is flagged if a symbol in the list is not referenced somewhere within the current module. It is also an error for any symbol in the list to be defined within the current module.

Example:

```
        extern expon

        push  {int,100},s(0)
        jsr   expon,s(0)
```

Format:

For each symbol declared external, an **extern** pseudo op is emitted, followed by a string containing the symbol.

0190	18F2	0005	HH...00

Syntax:

globl <name>...

where:
<name> is any legal identifier

Description:

The list of symbols is made visible to external modules. Each <name> in the list must be defined as a memory address somewhere within the current module.

Example:

```
            globl   test,data

test:
            move    (0:0),r(0:0)
            stop
data        ds      10
```

"test" and "data" are made visible to other modules.

Format:

For each symbol declared global, a **globl** pseudo op is emitted, followed by a string containing the symbol, followed by a memory address representing the value of the symbol.

0190	18F3	0005	HH...00	0009	HHHHHHHH

Syntax:

[<name>] dc V...

where:
<name> is an optional identifier

dc permits a list of atoms to follow the type keyword of each value.

Description:

dc allocates and initializes storage from a list of values starting at the current value of the location counter.

Example:

```
data3     dc    {char,'a','b'}
          dc    {string,"this is a string value"}
```

The first **ds** shown allocates 2 character values.

The second allocates a single string value. No identifer was specified.

Format:

A typed value is emitted for each value in the list.

Syntax:

[<name>] ds N [V...]
[<name>] ds [N] V...

where:
<name> is an optional identifier

ds permits a list of atoms to follow the type keyword of each value.

Description:

ds allocates storage for values starting at the current value of the location counter.

- If N is specified and N is greater than or equal to the number of values in the list, space for N values is allocated and the location counter is incremented by N.

- If N is specified and N is less than the number of values in the list, N is ignored.

- If N is not specified, the amount of storage allocated is equal to the number of values in the list. The location counter is incremented by this number.

- If a value list is specified, the allocated cells will be initialized to those values, beginning with the first.

- Cells allocated but not initialized are considered to hold undefined values. It is an error to attempt to read an undefined value.

Example:

```
data1     ds    10
data2     ds    10 {int,100},{nat,0,20,40}
data3     ds    {char,'a','b'}
          ds    {string,"this is a sting value"}
```

The first **ds** allocates 10 values and leaves them undefined. "data1" may be used to index into those values.

The second also allocates 10 values, but initializes the first to the integer 100, and the next 3 to the naturals 0, 20, and 40 The last 6 values are left undefined.

The third **ds** shown allocates 2 character values.

The fourth allocates a single string value. No identifer was specified.

Format:

A typed value is emitted for each value in the list. In addition, **ds** will emit an **org** pseudo op (see **org**) whenever the number of values in the value list is less than N.

Syntax:

rorg [M]

Description:

The location counter is reset to M, if specified; otherwise it remains unchanged. All memory addresses and labels specified after a **rorg** directive up to the next **org** or **rorg** directive are computed as displacements. Code generated after a **rorg** directive up to the next **org** or **rorg** directive is relocatable (program counter independent).

Example:

```
        rorg

        move  {int,100},data
        jsr   stuff
        stop

data    ds    10
```

In the above example, the **move** would be emitted using destination program counter relative addressing.

Format:

0190	18F4	0160	HHHHHHHH

Syntax:

org [M]

Description:

The location counter is reset to M, if specified; otherwise it remains unchanged. All memory addresses and labels specified after an **org** directive up to the next **org** or **rorg** directive not exlicitly expressed as displacements are treated as absolute addresses. Code generated after an **org** directive up to the next **org** or **rorg** directive is not relocatable.

Example:

```
            org
            move  (0:0),r(0:0)
            org   (1:0)
data        ds    {int,100},{nat,0}
```

Format:

0190	18F4	0160	HHHHHHHH

Format:

equ does not cause an emission.

Syntax:

<name> equ <equivalence>

where:
<name> is any legal identifier
<equivalence> is any atom or typed value

Description:

The symbol <name> is assigned the value of <equivalence>. Elsewhere in the source module, the symbol may be used in place of a literal value of the same type as <equivalence> using the following syntax:

- If the symbol represents a memory address **atom**, the symbol may be used directly.
- If the symbol represents a typed (immediate) value, it must be enclosed in curly braces "{" "}".
- If the symbol represents an integer or natural atom, it must be preceded by a pound sign "#".

Example:

```
progseg     equ   (0:0)
dataseg     equ   (1:100)
offset      equ   10
datafile    equ   {file,3}

            org   progseg
            move  {addr,data},r(0:0)
            move  {int,100},r(0:0)@#offset

            push  {string,"test.dat"},s(0)
            push  {datafile}, s(0)
            push  {int,0},s(0)
            push  {int,0},s(0)
            open  s(0)
            stop

            org   dataseg
data        ds    100
```

"progseg" and "dataseg" are equated to memory address atoms.

"offset" is equated to the integer atom 10.

"datafile" is equated to the file address value {file,3}.

memory address - [0160] [HHHHHHHH]

register address - [0170] [HHHHHHHH]

stack address - [0180] [HHHHHHHH]

file address - [01A0] [HHHH]

instruction - [0190] [HHHH] [zero or more operand atoms]

10.3. Object Module Format

The structure of an object module is very simple. The only object always found is a leading **org** directive. Next, if any symbols were declared global or external in the source module, a pseudo instruction will be emitted for each such symbol. The rest of the file contains executable and pseudo instructions emitted as they occur in the source.

11. Assembler Directives

AMASM recognizes the following directives:

equ	- equate
org	- absolute origin
rorg	- relative origin
extern	- external symbol
globl	- global symbol
ds	- define storage
dc	- define constant

Directives do not produce code which will be executed by AM, but they may cause linker/loader instructions to be emitted. The meaning and syntax of each directive is described in the following pages.

AM's tagged architecture. The following conventions will apply:

- All numbers show are in hexadecimal.
- The letter "H" is a place holder signifying any 4-bit value.
- The general form of a typed value is

[tag] [val]

where "tag" is a 16-bit type field, and "val" is an 8 to 32-bit value.

There are two exceptions:

- Character string atoms and values have a 16-bit size field inserted after the type field which indicates the number of characters in the value field (including the terminating null). This size field is omitted in memory (since it is not needed), replaced by a pointer to the string. Both the size field and pointer will be omitted in the format diagrams.
- Instruction values have a 16-bit opcode following the type field, followed by a list of operand values.

A number of the formats listed below are not described elsewhere in this manual since they are either not accessible to the programmer, or are implied by context.

10.1. Atom Formats

boolean - [0001] [HH]

natural - [0002] [HHHHHHHH]

integer - [0003] [HHHHHHHH]

character - [0004] [HH]

character string - [0005] [HH...00]

memory address - [0009] [HHHHHHHH]

register address - [000A] [HHHHHHHH]

stack address - [000B] [HHHHHHHH]

file address - [0011] [HHHH]

monadic operator - [000C] [HHHH]

dyadic operator - [000D] [HHHH]

relational operator - [000E] [HHHH]

boolean comparitor - [0012] [HHHH]

10.2. Value Formats

boolean - [0110] [HH]

natural - [0120] [HHHHHHHH]

integer - [0130] [HHHHHHHH]

character - [0140] [HH]

character string - [0150] [HH...00]

Immediate values are used, as in conventional assembly languages, for loading constants into cells, initializing storage, pushing parameters to subroutines on the stack, and so on.

A special syntax may be applied when expressing typed values for the **define storage** and **define constant** directives. The type keyword may be followed by a list of atoms of the appropriate type, separated by commas.

{int,1,2,3,4,5,6,7,8}

shows an example of this.

8. Expressions

An expression may be substituted anywhere an integer or natural atom is called for. The expression must be a sequence of integer/natural atoms (and symbolic constants equated to integer/natural atoms) separated by operators and grouping symbols which evaluates to an atom of the type called for where the expression is used.

8.1. Expression Operators

Legal operators are (in order of increasing precedence):

| | - or
& - and
+ - - addition and subtraction
* / % - multiplication, division, and modulus
- - unary minus

Expressions may be grouped using parentheses "(" ")".

9. Notation

Throughout the rest of this manual, the following notational conventions will be used to describe the syntax of directives and instructions.

M - memory address atom
R - register address atom
S - stack address atom
I - integer atom
N - natural atom
A - atom
V - typed value
< > - items enclosed within angle brackets are arguments
[ ] - items enclosed in square brackets are optional
<ea> - effective address
<ev> - effective value

10. Data Format

AMASM emits object code and directives using AM I/O modules. The object module is, thus, directly readable by AM. A linker and loader may be written either in a high level language, or AM assembler.

The data and object module formats described below are a direct reflection of

- attempt to free invalid memory segment
- attempt to free non-allocated segment
- stack empty
- stack overflow
- stack underflow
- file already open
- unable to close file
- unable to open <file>
- file already closed
- file not open
- file not open for reading
- file not open for writing
- reading file, type not recognized
- error reading file
- writing file, type not recognized
- invalid memory segment
- memory segment not allocated
- invalid memory address
- invalid register segment
- invalid register address
- invalid stack segment
- invalid file descriptor
- attempt to return head of null string
- value not of type bool
- atom not of type bool
- value not of type int
- atom not of type int
- value not of type nat
- atom not of type nat
- value not of type char
- atom not of type char
- value not of type string
- atom not of type string
- value not of type memaddr
- atom not of type memaddr
- value not of type regaddr
- atom not of type regaddr
- value not of type stkaddr
- atom not of type stkaddr
- value not of type instr
- atom not of type instr
- value not of type file
- atom not of type file
- type error

All machine errors are fatal.

13.2. Assembler Errors

AMASM will detect and report the following errors:

- symbol not an address
- symbol defined locally
- <symbol> does not match declared type
- relative memory indirect not permitted
- symbol not a value
- symbol not an integer
- symbols declared but not referenced
- displacement from external addresses not permitted
- relative addressing not permitted between segments
- out of symbol space
- symbol declared external
- symbol already defined
- symbol not of same type
- impossible value for given type
- syntax error

Assembler errors are not fatal, but will prevent the creation of the object module and, usually, the cross-reference file.

13.3. AM Operations

AM supports a useful set of monadic, dyadic, relational and test operators. These operators are to be used with the monad, dyad, if and test insructions. The mnemonics/symbols for each operator along with the data types to which each may be applied are described below.

13.3.1. Dyadic Operators (DOP's)

cat - string concatenation

cat accepts two string arguments and returns the concatenation of the first onto the second.

add,sub,mul,div - computational operators

These operators accept integer or natural arguments (both of the same type) and return a result of that type. Divide by zero returns an error. **div** discards any remainder.

and,or

and and **or** accept two boolean arguments and return a boolean result.

13.3.2. Monadic Operators (MOP's)

len - string length

len accepts a string and returns its length as a natural number.

not - boolean negation

not accepts a boolean argument and returns its negation.

make - make a string

This operator accepts a character argument and returns a string of length 1.

head - the head of a string

This operator accepts a string and returns the character at its head. It is an error to take the head of an empty string.

tail - the rest of a string

tail accepts a string and returns a string containing all but the first character. The tail of an empty string is the empty string.

13.3.3. Relational Operators (RELOP's)

The relational operators are:

== - equality
> - greater than
>= - greater than or equal to
< - less than
<= - less than or equal to
!= - not equal to

They may be applied to int, nat, char and string.

If == or != are applied to arguments of different types, == returns true, != return false. This applies also to types not listed above. >,>=,< and <= return an error if there arguments are not of the same type.

Relational operators return a boolean result.

13.3.4. Test Operators (BOP's)

These operators permit the programmer to test a cell for type before attempting to access it. These are necessary because AM considers it a fatal error to read from an undefined cell or apply an operator of one type on data of another. The test operators are the same as the type mnemonics, plus a mnemonic for testing undefined values:

bool
nat
int
char
string
instr
addr
file
undef

Test operators accept a typed value and return true if the value is of the specified type, false otherwise. **undef** returns true if a value is undefine. false otherwise.

Syntax:

<dop> Rx,Ry

where:
<dop> is a dyadic operator

Operation:

Ry <dop> Rx --> Ry

Description:

The operation corresponding to <dop> is applied to the operands and the result stored in Ry.

Example:

and r(0:0),r(0:1)

Addressing Modes:

Rx: r

Ry: r

Format:

0190	4801	operands

Syntax:

<dop> V,R

where:
<dop> is a dyadic operator

Operation:

R <dop> V --> R

Description:

The operation corresponding to <dop> is applied to the operands and the result stored in R.

Example:

sub {int,100},r(0:1)

Addressing Modes:

V: i

R: r

Format:

0190	4802	operands

Syntax:

<dop> Rx,Ry,Rz

where:
<dop> is a dyadic operator

Operation:

Ry <dop> Rx --> Rz

Description:

The operation corresponding to <dop> is applied to Rx and Ry and the result stored in Rz.

Example:

add r(0:0),r(0:1),r(0:3)

<dop> Rx,Ry,Ry is equivalent to <dop> Rx,Ry

Addressing Modes:

Rx: r

Ry: r

Rz: r

Format:

0190	4803	operands

Syntax:

<dop> V,Rx,Ry

where:
<dop> is a dyadic operator

Operation:

Rx <dop> V --> Ry

Description:

The operation corresponding to <dop> is applied to V and Rx and the result stored in Ry.

Example:

add {int,100},r(0:0),r(0:1)

<dop> V,Rx,Rx is equivalent to <dop> V,Rx

Addressing Modes:

V: i

Rx: r

Ry: r

Format:

0190	4804	operands

Syntax:

<mop> R

where:
<mop> is a monadic operator

Operation:

<mop> R --> R

Description:

The operator corresponding to <mop> is applied to R and the result stored in R.

Example:

not r(0:0)

Addressing Modes:

R: r

Format:

0190	3807	operands

Syntax:

<mop> Rx,Ry

where:
<mop> is a monadic operator

Operation:

<mop> Rx --> Ry

Description:

The operator corresponding to <mop> is applied to Rx and the result stored in Ry.

Example:

```
not    r(0:0),r(1:0)
```

Addressing Modes:

Rx: r

Ry: r

Format:

0190	4808	operands

Syntax:

<mop> V,R

where:
<mop> is a monadic operator

Operation:

<mop> V --> R

Description:

The operator corresponding to <mop> is applied to the immediate value V and the result stored in R.

Example:

```
not    {bool,flag},r(1:0)
```

Addressing Modes:

V: i

R: r

Format:

0190	4809	operands

Syntax:

offset I.R

R must contain a memory address atom

Operation:

R + I --> R

Description:

The sum of I and the address in R is stored in R.

Example:

```
offset  20,r(0:0)
```

Addressing Modes:

I: a

R: r

Format:

0190	3810	operands

Syntax:

move <ea1>.<ea2>

where:
<ea> must be one of the addressing modes listed below

Operation:

source --> dest

Description:

The value found at the source address is copied into the destination address.

Example:

```
        move r(0:0),data
        move {addr,data},r(0:20)
        move {int,100},r(0:20)@
        move r(0:20)@10,r(0:10)

data:   ds 100
```

Addressing Modes:

<ea1>: r,ri,rid,ridn,m,pcr,i

<ea2>: r,ri,rid,ridn,m,pcr

Format:

[0190] { [H815]...[H83C] } [operands]

Syntax:

push <ea>.S

where:
<ea> is one of the addressing modes listed below

Operation:

source --> S

Description:

The source value is pushed onto stack S. The programmer has no access to the stack pointer.

Example:

```
push {int,100},s(0)
push r(0:10),s(1)
```

Addressing Modes:

<ea>: m,pcr,r,ri,rid,ridn,i

S: s

Format:

0190 { H83D ... H843 } operands

LIST OF REFERENCES

Bergstra, A. and Tucker, J. V., *A Natural Data Type with a Finite Equational Final Semantics Specification But No Effective Equational Initial Semantics Specification*, Bull EATCS, 11, 1980, pp. 23-33.

Bergstra, A. and Tucker, J. V., "Initial and Final Algebra Semantics for Data Type Specifications: Two Characterization Theorems", *SIAM Journal of Computing, Vol. 12, No. 2*, May 1983.

Bundy, A., *The Computer Modelling of Mathematical Reasoning*, Academic Press, New York, 1983.

Burstall, R. M. and Goguen, J. A., "Putting Theories Together to Make Specifications", *Proc. 5th Intl. Joint Conf. on AI*, Cambridge, Mass., Aug 1977, pp. 1045-1058.

Naval Postgraduate School, Tech. Report NPS52 84-022, *A Formal Method for Specifying Computer Resources in an Implementation Independent Manner*, Davis, D., Monterey, Ca., Dec 1984.

Ehrich, H. D., "On the Theory of Specification, Implementation and Parameterization of Abstract Data Types", *Journal of ACM 29, No. 1*, Jan 1982.

Fasel, J., *Programming Languages as Abstract Data Types - Definition and Implementation*, Ph. D. Thesis, Purdue University, Aug 1980.

Ganzinger, H., "Parameterized Specifications: Parameter Passing and Implementation with Respect to Observability", *ACM Trans. on Prog. Lang.and Sys., Vol. 5, No. 3*, Jul 1983, pp. 318-354.

Giegerich, R., "A Formal Framework for the Derivation of Machine-Specific Optimizers", *ACM Trans. on Prog. Lang. and Sys., Vol. 5, No. 3*, Jul 1983, pp. 478-498.

Goguen, J. A., Thatcher, J. W., Wagner, E. G. and Wright, J. B., *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*, Current Trends in Programming Methodology IV, Data Structuring, R. T. Yeh, ed., Prentice-Hall, Englewood Cliffs, N. J., 1978, pp. 80-149.

Syntax:

unlink R

Operation:

R@ --> R

Description:

The value in the base address of the segment pointed to by R is returned in R. The segment is freed.

Example:

```
proc:     link   r(0:5),1
          move   r(0:5)2@4,r(0:0)
          add    {int,100},r(0:0)
          move   r(0:0),r(0:5)2@4
          unlink r(0:5)
          rts
```

Addressing Modes:

R: r

Format:

0190	2897	operand

Syntax:

link R,N

Operation:

R@ --> address@
address --> R

Description:

A segment of N cells is allocated from the heap. The value stored in R is save at the base address of the segment. The segment base address is returned in R.

This instruction is designed to create dynamic links for local environments.

Example:

```
proc:     link   r(0:5),1
          move   r(0:5)2@4,r(0:0)
          add    {int,100},r(0:0)
          move   r(0:0),r(0:5)2@4
          unlink r(0:5)
          rts
```

Above is an example of uplevel addressing.

Addressing Modes:

R: r

N: a

Format:

0190	3896	operands

Syntax:

rts S

Operation:

S --> PC

Description:

Execution resumes at the address popped from stack S.

Example:

```
inrc:     add   {int,1},r(0:0)
          rts   s(0)
```

Addressing Modes:

S: s

Format:

0190	2895	operand

Syntax:

bsr <ev>,S

where:
<ev> is one of the addressing modes listed below

Operation:

PC --> S
PC + <ev> --> PC

Description:

The program counter is pushed onto stack S, and execution resumes at the sum of the program counter and <ev>.

Example:

```
bsr     r(1:0),s(0)
```

Addressing Modes:

<ev>: r,a S: s

Format:

[0190] { [3893],[3894] } [operands]

Syntax:

jsr <ea>,S

where:
<ea> is one of the addressing modes listed below

Operation:

PC --> S
<ea> --> PC

Description:

The program counter is pushed onto stack S, and execution resumes at <ea>.

Following a **rorg** directive, memory absolute is converted automatically to program counter relative.

Example:

```
        jsr     incr,s(0)
```

Addressing Modes:

<ea>: m,mi,r,pcr S: s

Format:

[0190] { [H890]...[H892] } [operands]

Syntax:

stop

Operation:

-

Description:

Execution is terminated.

Addressing Modes:

-

Format:

0190	1880

Format:

0190

{ 6862 , 6863 , 6866 , 6867 , 5872 , 5873 , 5876 , 5877 } operands

Syntax:

if R <relop> <ev>,Mx,My
if <bop> <ea>,Mx,My

where:
<relop> is a relational operator
<bop> is a test operator
<ea> and <ev> are one of the addressing modes listed below

Operation:

```
if R <relop> <ev> then
        Mx --> PC
else
        My --> PC

if <bop> <ea> then
        Mx --> PC
else
        My --> PC
```

Description:

If the comparison is true, execution resumes at Mx; otherwise, at My.

Example:

```
            if    r(0:0) > r(0:1),case1,case2
stuff:      move  r(0:0),data
case1:      jsr   first,s(0)
            if    int r(0:0),case1
            stop
case2:      jsr   second,s(0)
            stop
```

Addressing Modes:

R: r

<ev>: r,i

<ea>: r,m

Mx: m,pcr

My: m,pcr

Syntax:

if R <relop> <ev>,M
if <bop> <ea>,M

where:
<relop> is a relational operator
<bop> is a test operator
<ea> and <ev> are one of the addressing modes listed below

Operation:

if R <relop> <ev> then
M --> PC

if <bop> <ea> then
M --> PC

Description:

If the comparison is true, execution resumes at M; otherwise, with the next instruction.

Example:

```
            move  {int,10},r(0:0)
loop:       if    r(0:0) < {int,1},done
            sub   {int,1},r(0:0)
            jmp   loop
done:       if    int data,loop

data        ds    1
```

Addressing Modes:

R: r

<ev>: r,i

<ea>: r,m

M: m,pcr

Format:

[0190]

{ [5860],[5861],[5864],[5865],[4870],[4871],[4874],[4875] } [operands]

Syntax:

bra <ev>

where:
<ev> is one of the addressing modes listed below

Operation:

PC + <ev> --> PC

Description:

Execution resumes at the sum of the program counter and the effective value.

Example:

bra 100

Addressing Modes:

<ev>: a,r

Format:

0190	{	H853	...	H854	}	operands

Syntax:

jmp <ea>

where:
<ea> is one of the addressing modes listed below

Operation:

<ea> --> PC

Description:

Execution resumes at <ea>.

If **jmp** follows a **rorg** directive, a jump to memory absolute is converted to a branch.

Example:

```
            jmp here
            jmp r(0:0)
here:       jmp (1:150)@
```

Addressing Modes:

<ea>: m,r,mi,pcr

Format:

[0190] { [H850]...[H852] } [operands]

Syntax:

popx S

Operation:

S -->

Description:

The top value of stack S is removed.
It is an error to attempt to remove the top of an empty stack.

Example:

popx s(0)

Addressing Modes:

S: s

Format:

0190	284A	operands

Syntax:

pop S,<ea>

where:
<ea> is one of the addressing modes listed below

Operation:

S --> dest

Description:

The source value is popped off stack S and stored at <ea>. The programmer has no access to the stack pointer.

It is an error to attempt to pop a value from an empty stack.

Example:

```
        pop s(0),r(0:1)
        pop s(0),data

data:   ds 1
```

Addressing Modes:

S: s

<ea>: m,pcr,r,ri,rid,ridn

Format:

[0190] { [H844] ... [H849] } [operands]

Gratzer, G., *Universal Algebra*, Van Nostrand, New York, 1968.

Griffin, R., *An Algorithm to Test for Confluence in a System of Left to Right Rewrite Rules*, Master's Thesis, Naval Postgraduate School, Monterey, Ca., Dec 1984.

Guttag, J. V., Horowitz, E. and Musser, D. R., "Abstract Data Types and Software Validation", *Comm. ACM, Vol. 21, No. 12,* Dec 1978, pp. 1048-1064.

Guttag, J. V., Horowitz, E. and Musser, D. R., *The Design of Abstract Type Specifications,* Current Trends in Programming Methodology IV, Data Structuring, R. T. Yeh, ed., Prentice-Hall, Englewood Cliffs, N. J., 1978, pp. 60-79.

Hoffman, C. M. and O'Donnell, M. J., "Programming with Equations", *ACM Trans. on Prog. Lang. and Sys., Vol. 4, No. 1,* Jan 1982, pp. 83-112.

Lilly, N., *An Algebraic Specification Language and a Syntax Directed Editor*, Master's Thesis, Naval Postgraduate School, Monterey, Ca., Dec 1984.

Myers, G. J., *Advances in Computer Architecture* (2nd Edition), Wiley, New York, 1982, pp. 17-29.

Patterson, D. A. and Sequin, C. H., "A VLSI RISC", *IEEE Computer,* Sep 1982, pp. 8-21.

Tannenbaum, A. S., Klint, P. and Bohm, W., "Guidelines for Software Portability", *Software - Practice and Experience, Vol. 8,* 1978, pp. 681-698.

National Science Foundation Div. of Comp. Research, Report SEG-75-1, *A Preliminary Definition of Janus,* Waite, W. M. and Haddon, B. K., Washington, D. C., 1975.

Yurchak, J. M. and Griffin, R., *A Storage Organization to Support Formal Physical Resource Abstractions,* Unpublished notes, Dec 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Superintendent Attn: Library (Code 0142) Naval Postgraduate School Monterey, California 93943	2
3.	Chairman (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Computer Technology Programs (Code 37) Naval Postgraduate School Monterey, California 93943	1
5.	Daniel Davis (Code 52vv) Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
6.	Lt. John M. Yurchak P.O. Box 268 RD#4 Muncy, Pennsylvania 17756	10
7.	Wayne Moore (Code 3160) Naval Costal Systems Center Panama City, Florida 32407	1
8.	Capt. H. McArthur 135 Johnson St. Red Springs, North Carolina 28377	1
9.	Dr. Charles Getchell Department of Mathematics Lycoming College Williamsport, Pennsylvania 17701	1

END

FILMED

8-85

DTIC