SADL

A SYMBOLIC ARCHITECTURE DESCRIPTION LANGUAGE

A Thesis Presented in Partial Fulfilment

of the requirements for the Degree of

Master of Arts in Computer Science

at Massey University

Thomas William Livingstone

1985

Abstract

This thesis develops a new language capable of specifying computer architecture at the symbolic, or assembly language level.

The thesis first provides a representative sample of current, or proposed, computer description languages and discusses four of the languages and their merits with regard to the symbolic approach. Next, a model is proposed of computer architecture at the level which is visible to an executing sequence of instructions. This model is based on the assembly language level of computer architecture. Next, the Symbolic Architecture Description Language (SADL) is described. Finally, Build, a LISP program which takes SADL architecture descriptions and generates functions and data structures for use in simulating architectures, is described.

Acknowledgements

I would like to thank the following for their support and assistance during this thesis:

Nola Simpson, for being my supervisor;

Paul Lyons, for his excellent criticisms of the manuscript;

The staff of the Massey University Computer Centre for their cooperation.

Table of Contents

1 Introduction

This thesis proposes a language for symbolically specifying the execution environment of assembly language programs. The assembly language level of description was chosen as it is the most abstract level which is still capable of specifying the instruction set functionality of a computer. Higher level abstractions, such as compilers and interpreters, no longer allow explicit access to the physical machine state, while lower level descriptions have little meaning to the software engineer.

Computer Design, once an area of individual artistic expression, is becoming the result of systematic cooperation between the members of a team, often a large team, frequently aided by automated design tools. Members of the design team must be able to communicate with each other, and with their design tools, without ambiguity, and to this end a number of formal languages have been developed for the description of computer systems.

It has become a truism that a computer system consists of a number of layers, each describable in terms of a particular model. In this thesis, we shall find the level described by the ISP (Instruction Set Processor) model [Bell71] to be the most useful. A computer architecture defined in terms of this model would comprise:

(i) a set of registers,

(ii) a memory which contains the encoded instructions,

(iii) a set of functions which

(a) produce the effective address for obtaining and storing the operands and

(b) specify the actions required to implement the instructions.

(iv) a finite state machine which defines the loading, interpretation and execution of instructions defined for the architecture.

There are two approaches to modelling an architecture at the ISP level. The traditional method (adopted in the specification language ISPS [Barb81]) is a mechanical view: the architecture is viewed as a structure consisting of registers and decoding functions which operate on the machine code of the architecture.

The second approach is a symbolic view: it is derived from the Assembly Language model of architecture. It ignores the mechanics of encoding and decoding - the instruction is only ever represented in symbolic form - and models the decode and execute cycle as a language interpretation cycle.

Why use the symbolic approach ?

1. It is the natural tool for software engineers.

A software engineer who programs an architecture directly (as opposed to using a high level language) makes use of the symbolic level and an Assembler. The costs of programming in machine code versus assembly language and the functional equivalence of the two means that machine code programming has been superceded by assembly language programming, except possibly for some extremely specialised applications.

2. It is a natural pedagogic tool.
   Because people are familiar with the symbolic approach to architecture, it is easier to comprehend architectures when expressed symbolically. This is important when attempting to learn new architectures, when comparing two architectures or when evaluating an architecture.

3. It allows direct simulation of the symbolic program.
   The normal process when simulating the execution of programs on a particular architecture is to write the programs (normally in assembly language), translate them into the machine code for the target architecture and run them on a simulator which emulates the instruction and register sets of the target machine.

   Having the architecture specified symbolically bypasses the translation phase as the assembly language program may be executed directly by the simulator. This saves programmer time and therefore saves money. Balanced against this is the increased cost in processor time of executing an interpreted program rather than a compiled program. Also, the symbolic tracing of instruction execution is simplified and protection mechanisms against faulty programs are easier to install; for instance it would be impossible for a running program to try executing data, an occurrence common in out-of-control machine code programs.

4. It can fully specify the register set of an architecture, and external lines may also be modelled indirectly as registers. The symbolic approach allows the register set of an architecture to be specified to the same detail as the mechanical approach to ISP specification. Thus there is no expressive capability lost when using the symbolic approach over the mechanical approach.

5. Fundamental to the symbolic approach is the fact that each machine instruction has one equivalent symbolic instruction and that the functionality of both is the same. This is a widely recognised view of pure assembly language (as opposed to macro-assembly language).

Section 1.2 of chapter 1 examines four languages which are used, or have been proposed for use in describing the instruction set processor level. Two of the languages, LISP and VDL, deal with instruction set processors at the symbolic level while the other two languages, Pascal and ISPS, deal with the machine code level.

Chapter 2 proposes a model of computer architecture which is centred on the view of an executing program within a machine. The model is based upon the stored program concept with a single execution unit and single instruction and data streams; this excludes architectures based upon array and vector processing as well as systolic architectures.

Chapter 3 defines both the syntax and semantics of the Symbolic Architecture Description Language (SADL) and shows the capabilities and restrictions of the current version of the language.

Chapter 4 describes software which processes a description in SADL and produces a set of data structures and functions which may be used to simulate the architecture when provided with an assembly language program. It is an application intended to test the validity of SADL.

1.1 Multi-level Architectures and Virtual Machines

One of the major concepts that has evolved in computing in the last fifteen years has been the view of a computer system as a layered hierarchy of abstract machines. At the top of the hierarchy are user applications and at the bottom is the physical specification of the electronic components which combine to form the hardware.

Each level may be viewed (more or less) as a complete architecture independent of those levels in the hierarchy either above or below it. This view is invaluable in simplifying the task of designing or analysing computer systems.

There are differing views as to what constitutes each layer, but Siewiorek, Bell and Newell [Bell71,Siewiorek82] have proposed a layering that suits the author's purposes and is quite widely recognised. I shall refer to this as the Bell model.

In the Bell model there are four main levels which are subdivided into sublevels. The main levels are: Circuit level, Logic level, Program level, PMS level.

The only level of relevance to the software engineer is the program level, because this level is broken down into the ISP (Instruction Set Processor) sublevel, and the High Level Language sublevel which is itself broken down into Operating System, Run-time System, Application Routines and Applications Systems sublevels.

Example 1.1

```
---------------------------------------------------------|
PMS                                                      |
---------------------------------------------------------|
Program      | High Level     | Applications Systems     |
             | Language       |--------------------------|
             |                | Applications Routines    |
             |                |--------------------------|
             |                | Run-time System          |
             |                |--------------------------|
             |                | Operating System         |
             |----------------|--------------------------|
             | Instruction                               |
             | Set Processor                             |
---------------------------------------------------------|
Logic                                                    |
---------------------------------------------------------|
Circuit                                                  |
---------------------------------------------------------|
```

The Assembly Language sublevel fits into the hierarchical view just above the ISP sublevel and below the Operating System sublevel (although Tanenbaum [Tanenbaum76] views the assembler level as being above the operating system level).

The reasons for placing Assembly Language at this point in the hierarchy are these:

(i) In the abstraction process, information is hidden or lost. Anything that may be specified by an Assembly Language program may be specified in greater detail at the ISP sublevel; this indicates that the Assembly level is an abstraction of the ISP sublevel.

(ii) Similarly, an Operating System is a composition of concepts expressible in Assembly Language. Its component subroutines, coroutines, and programs are built up from assembler-level instructions, either directly or (as in the case of UNIX and Burroughs' MCP which are written in high level languages) indirectly.

Where do compilers, which bypass the assembler level and directly produce code at the ISP level, fit into the model? Their mapping from a particular level in the hierarchy of abstract machines to another, lower level may bypass one or more levels. However the number of levels which a compiler bypasses does not invalidate the hierarchical structuring of abstract machines.

1.2 Current Architecture Description Languages

There currently exist a considerable number of languages for describing computer architectures at various levels. Most of these straddle the Register Transfer and the ISP levels. There seem to be almost no generally recognised languages which approach the ISP level from the language (or symbolic) direction.

Subrata Dasgupta [Dasgupta82] surveys a group of languages which he calls Computer Design and Description Languages (or CDDLs). The survey concentrates on ISPS, S*A and the CONLAN extensible language system.

Two points made by Dasgupta are significant. The first is that at the time of writing (1982) CDDLs had not been generally accepted by the computer design community. The second point is that the majority of CDDLs that have been proposed have fallen into the Register Transfer level of description. This is partly true of most of the languages described here although they all have applicability at the ISP level. Only LISP and VDL have the ability to specify architecture at the symbolic level.

1.2.1 ISPS

ISPS is the single most influential language for specifying processor architecture in terms of the instruction set and registers.

When Bell and Newell first formulated their layered model of abstract machine descriptions they developed a pedagogic language with which they illustrated the instruction set processor model. The language that they created was called ISP, the same name as the level of abstract machine that was being described. ISP was a descriptive tool rather than a formal language and thus suffered from shortcomings which led to a short period of evolution. The resulting language was adopted for use in various applications [Barb81] and has come to be known as ISPS. In its most recent form it was used by Bell, Newell and Siewiorek in the update of the original Bell and Newell text.

In ISPS an architecture consists of collections of <u>entities</u> and <u>carriers</u> the interfaces between which and behaviour of which are described. An entity is a unit of hardware. A carrier is a storage location (register or memory) used for communicating between entities. The interface describes the number and types of carriers used to store and transmit information between entities.

The behavioural aspects of entities are described by procedures which specify the operations carried out by each entity.

A carrier is described by naming the carrier and specifying its word dimension and array dimension.

Example 1.2 - M\Memory[0:4095]<0:11>.

specifies a carrier named M with an alias Memory; the alias is intended to document the meaning or usage of a carrier's name. The square brackets indicate that the carrier is an array of cells where the first cell is named 0 and the last cell is named 4095; N.B. the cell indices are names not numbers. The angle brackets indicate the structure of the individual words to be 12 bits named 0, 1, ... , 11.

Procedures contain data, control operations, and local entities which may be of arbitrary complexity. Each procedure has associated with it a carrier of the same name as the procedure and with a structure specified by the procedure. This is the mechanism for explicitly returning information from procedures; when a procedure is invoked it performs its operations and the value returned from the procedure is accessible from the carrier of the same name.

ISPS is designed to describe more than just the instruction set processor view of the architecture, it is able to describe architecture from the logical level up to the ISP level. In this respect ISPS has been significantly expanded in its purpose from the original language ISP.

However, the following description of ISPS will deal only with those aspects of the language that are used to describe instruction set processors.

An ISPS description is broken into a series of sections of the form:

```
        ** section.name **
        <declaration>,
        <declaration>,
            ...
        ** section.name **
        <declaration>,
        <declaration>,
            ...
```

This grouping of sections is purely a documentation device and the section headings have no semantic content. Section names are arbitrary names intended to convey information about the declarations immediately following. A section is a list of declarations separated by commas.

There are two types of declaration: the carrier declaration and the procedure declaration - The carrier declaration has the form of the memory declaration given previously with the additional feature that a carrier may be associated (as a synonym) with part of another carrier:

Example 1.3 - The PDP-8 extended accumulator is defined thus

```
        LAC <O:12>,
           L\Link<>          := LAC<O>,
           AC\Accumulator    := LAC<1:12>
```

The expression <> indicates a single, unnamed bit. Notice that there are three distinct declarations in the above example and that the indentation is purely a typographical tool chosen to convey the idea that the second two declarations are associated with the first.

Example 1.4 - PDP-8 Page zero format:

```
        P.O\Page.Zero[O:127]<O:11>    := M[O:127]<O:11>,
        A.I\Auto.Index[O:7]<O:11>     := P.O[8:15]<O:11>
```

In this example the first declaration associates a carrier with a part of the memory array (see Example 1.2) while the second declaration associates another carrier with part of the first carrier. This illustrates how carriers may be hierarchically structured.

Procedures are of the form:

```
        procedure.name :=
           BEGIN
              <statements>
           END
```

Statements may be grouped either sequentially or concurrently. A sequential ordering is indicated by the keyword NEXT while concurrent clustering is indicated by a semicolon (";").

A sequential group of statements:

```
    <statement group 1> NEXT
    <statement group 2> NEXT
         ...
```

A concurrent group of statements:

```
    <statement 1> ;
    <statement 2> ;
         ...
```

The statements separated by semicolons may execute concurrently in an asynchronous manner but all statements immediately preceding a

NEXT must be completed before any statements following may begin executing.

The following example is a complete ISPS function and illustrates several aspects of the language.

Example 1.5 - PDP-8 effective address calculation:

```
    eadd\effective.address<0:11> :=
         begin
         Decode pb =>
              begin
              0 := eadd = '00000 @ pa,             !page zero
              1 := eadd = last.pc<0:4> @ pa        !current page
              end NEXT
         if ib =>                                  !indirect bit
              begin
              if eadd<0:8> eqv #001                !auto index
                   => M[eadd] = M[eadd] + 1 NEXT
              eadd = M[eadd]                       !indirect addr.
              End
         End,
```

ib is the indirection bit; pb is the page bit.

The Decode operation is equivalent to the Pascal CASE statement. The => token serves as a delimiter for both the Decode and the conditional constructs; In the former it delimits the carrier being decoded (and is redundant), in the latter it is the equivalent of then in Pascal. The := token delimits constants in the Decode construct. The IF test is false if the expression being tested resolves to zero. The operator @ is the concatenation operator.

Note that eadd has a carrier component as part of its declaration. A typical use of the carriers associated with procedures would be:

Example 1.6 - PDP-8 Increment and Skip if Zero instruction

```
        BEGIN
           M[eadd] = M[eadd()] + 1 NEXT
           IF M[eadd] eql 0 => PC = PC + 1
        END
```

This instruction increments the addressed memory.

The effective address is computed once by invoking the function eadd() and from then on the value of the effective address is available from the carrier eadd.

In Example 1.5 the memory is accessed in an assignment statement where both sides of the assignment refer to the carrier eadd not the function eadd. One of the problems with ISPS is this ambiguity as to whether the function is being invoked or the associated carrier is being referenced.

Note that procedures may have parameters, though this is not shown.

Other components of ISPS are:

logical operators: and or not xor eqv

arithmetic operators: + - * / mod

relational operators: eql lss leq neq geq gtr tst

shift operators: sl0 sl1 sld slr sr0 sr1 srd srr

number bases: ' (binary), # (octal), " (hexadecimal).

= is the logical assignment operator. Truncation or <u>zero</u> extension is performed on the value on the right hand side to fit the destination on the left hand side.

<= is the transfer operator. Truncation or <u>sign</u> extension is performed on the right hand side to fit the destination on the left.

The three ways of exiting a procedure invocation are <u>leave</u>, <u>restart</u>, and <u>resume</u>.

Leave entityname -

terminates the named entity. The only restriction is that the statement must occur within the dynamic scope (activation) of the named entity.

resume entityname -

returns control to the specified entity.

restart entityname -

terminates and reactivates the named entity (effectively a combined leave and resume).

The arithmetic operators are modified in their function by representation modifiers. The following arithmetic representations are supported:

Modifier	Representation
{TC}	Two's complement
{OC}	One' complement
{SM}	Signed magnitude
{US}	Unsigned magnitude

The usage is:

```
M[eadd] = M[eadd()] +{SM} 1
```

The modifier affects the arithmetic operator immediately preceding it.

The control clauses are specified by IF, REPEAT and DECODE.

```
IF <expression> => <stmt>
```

If the expression does not evaluate to zero then <stmt> is invoked.

```
        REPEAT <stmt>
```

The statement is continously executed. If it is to terminate then one of the control transfer statements LEAVE, RESTART, RESUME must be present within <stmt>.

```
        DECODE <carrier> => <selector block>
```

This statement evaluates the contents of <carrier> and executes the appropriately labelled branch of the <selector block>.

ISPS is a flexible language with considerable expressive power both for instruction set processor descriptions and for the lower levels of the abstract machine hierarchy.

ISPS has been highly successful and has been applied to:

The evaluation and certification of instruction set processors.
VLSI design automation.
Automatic generation of assemblers.
Production Quality Compiler Compilers.
Symbolic execution of ISPS descriptions.
Functional fault simulation.

Despite this wide application ISPS is not perfect. Dasgupta lists several drawbacks of the language, two of which are relevant to this thesis.

The first is stylistic. ISPS employs familiar symbols in an unfamiliar manner. The examples given are _:=_ and _;_. The first is normally used as the assignment operator whereas ISPS uses it to delimit the names of entities or labels. The _;_ is almost universally used to denote sequential ordering but in ISPS it is used for the opposite purpose of specifying concurrent execution.

The second drawback is a more limiting one - ISPS has few data types. Dasgupta identifies the "register" and the "memory" as the only data types supported by ISPS. The author feels that a type "bit", the one indivisible unit of storage should be included too. There is no facility in ISPS for producing composite data structures from collections made up of the base types as in Pascal and this does tend to limit the ease with which complex register structures may be described.

The main limitation of ISPS, in the context of this thesis, is that ISPS has no facilities for integrating the symbolic Assembly Language level into the ISPS description. The essence of ISPS is to describe the machine code view of the ISP model. The lack of Assembly Language constructs means that an Assembly Language program is unable to be represented within ISPS without extensions to the language.

Because ISPS represents a machine code description, the extension to ISPS would need to be a complete description driven Assembler. This is a non-trivial exercise. There are problems of instruction naming, access methods and their assembler formats, as well the generality necessary to support a wide variety of architectures that indicate it is easier to build the assembly language definition and then derive the ISPS description from it. Attempts at standardising even Assembly Language mnemonics have not produced entirely satisfactory results [Fischer79,Distler82]; standardisation of access methods would be much more difficult.

1.2.2 The Vienna Definition Language

The Vienna Definition Language (VDL) was originally designed to specify the syntax and the semantics of PL/1. It is a language "for defining interpreters rather than compilers" [Wegner72]. LISP, and in particular the technique of language definition illustrated by the APPLY function, was an important influence in determining the approach to language definition of VDL. This is noticeable when examining VDL expression trees.

VDL has subsequently been applied to the specification of other languages such as Algol-68 but has not generally been widely applied. One author [Spitzen76] derides the VDL description of PL/1 as being "lengthy, unintuitive, and itself a program."

Another, [Lee73], describes the use of VDL as a tool for describing a machine at various levels of abstraction down to the register transfer level. The architecture of the example given in Lee's paper was too limited to fully evaluate the applicability of VDL to describing arbitrary computer systems but there is certainly reason to believe that VDL does indeed have the power.

The stumbling block appears to be the general lack of acceptance of VDL and the "unintuitive" structure of the language. This structure would probably not be so much of a problem to people who have extensive grounding in Language Theory. It is also possible that VDL has been ignored not because of inherent limitations of the language itself but rather because it is associated with the generally unsuccessful languages PL/1 and Algol 68.

In VDL an architecture is modelled as a finite state machine with a set of states containing information on the data being manipulated (registers) and the instructions which define the transformations to be executed over the data. A function will interpret and execute the instructions in the current state of the machine.

Within the definitional machine (the VDL program defining the target architecture) there is a component known as the "control stack". This stack contains the set of instructions which are awaiting execution and which, when executed, model the execution of instructions in the target architecture. The "control stack" may be viewed as a tree in which the nodes are definitional instructions.

Only instructions at the leaf nodes may be executed; this means that an instruction at a given node in the tree is inhibited from execution until all its offspring instructions have been executed (and therefore removed).

Definitional instructions are executed either as macro-expansions or as state-modifiers. In a macro-expansion the instruction replaces itself in the control stack (tree) by a subtree, thus modelling the transition in definitional level or the sequencing of operation. State modifiers alter the state of the machine (other than the control stack), thus modelling operations upon registers.

A definitional instruction may contain several definitions but only one is applicable at any one time.

The general form of a definitional instruction is:

```
        instr(q1, q2, ... ,qn) =

              p1 -> group1
                  ...
                  ...
              pm -> groupm
```

p1...pm are predicate expressions that select alternative actions (group1...groupm). q1...qn are parameters that may occur in pi or groupi and that are replaced by values before the instruction is executed. The execution of the instruction causes the current state to be transformed by the action groupi corresponding to the first true predicate pi.

Where a group is a macro-expansion, the notation shows the set of instructions which replace the instruction being executed. The form which is used indicates the structural relations between the instructions.

- indentation indicates a lower level in the tree.
- comma (",") indicates continuation of a level.
- semi-colon (";") indicates completion of a level except where the instruction that the semi-colon follows is the last in the macro-expansion in which case it is unnecessary.

Example 1.7

```
inst-1;
   inst-2;
      inst-3;
         inst-4
```

The order of execution is from the leaf node (inst-4) to the root node (inst-1).

Example 1.8

```
inst-1;
   inst-2,
   inst-3,
   inst-4
```

Instructions at the same level are executed asynchronously, so inst-2, inst-3, inst-4 will each execute independently but inst-1 will not be able to execute until all of the other instructions have completed.

Normally, the control tree structure is represented linearly using braces to indicate subtrees:

Example 1.9

```
inst-1;
   inst-2,
   inst-3;
      inst-4,
      inst-5
```

is equivalent to:

```
{inst-1 {inst-2 inst-3 {inst-4 inst-5} } }
```

State-modifying definition groups specify changes to the state of the definitional machine. Each group consists of a set of selector : value pairs. Selectors represent states in the machine and the values are functions with parameters. A typical state modifying instruction would be:

```
pc_to_mar =
     s-mar : s-pc( E )
```

which means "replace the contents of the s-mar component of the state E by the contents of the s-pc component of the state E.

To overcome problems of timing with the pairs (which are asynchronous) the new state is defined to be a copy of the old state rather than a modification of it.

Example 1.10 - 3-bit rotate

```
shift =
     bit-0.s-acc : bit-1.s-acc(E)
     bit-1.s-acc : bit-2.s-acc(E)
     bit-2.s-acc : bit-0.s-acc(E)
                                    ( "." means component of)
```

If the new state were not defined to be a copy of the old state then the above instruction group would be meaningless because the original value of one of the bit components <u>must</u> be lost, there being no guarantee that all operations will (or can) occur at the same instant.

In [Lee73] a simple computer architecture (the "Blue Machine") is described which is similar to that of a PDP-8. Its state may be defined by the predicates:

Example 1.11

```
is-E = ( <s-mem : is-memory>,
         <s-mbr : is-word>,
         <s-acc : ( <s-link : is-bit>,
                    <s-body : is-word> ) >,
         <s-mar : ( <s-ma : is-word-address>,
                    <s-pa : is-page-address> ) >,
         ... )
```

where each of the pairs specifies the name of the branch on which the component is located and the structure of the component. The above example describes the architecture at the register transfer level.

```
is-word = ( { <bit(i) : is-bit> | 0 <= i <= 11 } )
```

defines a structure composed of a set of pairs, the object of each of which is a bit and the selector of which is the form bit(i) where the value of <u>i</u> is in the range {0,11}. This effectively defines a 12-bit word.

This explanation cannot do justice to the power of VDL and is only intended to impart a feeling for the way that VDL may be applied to architecture description.

1.2.3 LISP

Lisp has been put forward as a language suitable for specifying computer instruction sets [Cragon83]. It is stated that the LISP environment has the ability to describe components of the architecture, such as registers, operations and control, symbolically with the benefit of being able to edit the architecture using the interactive editor available as part of the LISP environment. The example architecture given in Cragon's paper indicates that this is so, but the architecture being modelled is reasonably simple.

The basis of the argument is that the functionality of the instructions may be directly encoded using LISP functions.

```
Example 1.12 - for the RISC-1 instruction:  ADD  RS,S2,RD
               the defined operation  is :  RD <- RS + S2
```

this may be encoded in LISP as :

```
(DEFUN ADD (RS S2 DEST)
   (SETQ RD (+ RS S2))            RD <- RS + S2
   (STORE (REG (EARD DEST)) RD) store RD
   (SETQ PC (ADD1 PC))            advance Program Counter
)
```

The RISC architecture is described in [Patterson82].

The operations specified (such as addition, subtraction and the logical operators) are performed using the operators available within MACLISP. The implementation restricts the wordlength of the architecture being modelled to less than the wordlength supported by the LISP environment.

Memory and array registers are defined by declaring them to be LISP arrays.

Example 1.13

```
(ARRAY MEM T (EXPT 2 16))
(ARRAY REG T 138)
```

The T indicates that each element may contain an arbitrary s-expression (list).

Writes to memory are accomplished by:

```
(STORE (MEM EA) X)
```

where X is the data and EA is the effective address.

A series of functions specifies the control operations of the architecture; The assembly language format (contents of MEM) is:

```
(OP SCC DEST SOURCE1 IMM SOURCE2)
```

OP is the opcode mnemonic;

SCC is the "set condition codes" enable bit;

DEST is the destination register address;

SOURCE1 is the first source operand register address;

IMM indicates whether or not the SOURCE2 field is a register address or constant value.

The functions:

(DEFUN IFS (PC) ...) - loads the instruction register with the symbolic instruction located in the memory location pointer to by the program counter.

(DEFUN DECODE (IR) ...) - extracts the values from the field entries for the instruction.

(DEFUN DISPATCH () ...) - This is a single case statement which invokes a different function for each instruction of the architecture.

(DEFUN SET-PSW (RD) ...) - A two bit program status word was defined in the article with this function being used to set the values. The model of the architecture is dependent for its information on the fact that the LISP precision is greater than the precision of the destination register in the target architecture as the psw is modified separately from, and after, each instruction execution. This could not apply to a two, or less, operand architecture as information would be lost.

The function RUN emulates the finite state machine which causes the initial status of the machine to be set up and the IFS, DECODE, DISPATCH loop to be continuously executed until a STOP instruction is encountered.

The result is a specification of the ISP for RISC-1 which is directly executable within a LISP environment and so may be immediately evaluated and modified in an iterative manner.

This solution appears to be an ad-hoc one. It has a number of limitations, some of which are not mentioned in the paper. The limitations are:

1. The RISC architecture is not typical of computer architectures as the register structure, the effective address calculations, and the operations performed by the instructions are unusually simple; the RISC-1 is only slightly more complex than a Motorola 6800 or Intel 8085 microprocessor.

2. Using the arithmetic precision of MACLISP limits the architectures which may be specified. Architectures with words longer than 32 bits may not be specified using the numeric precision available in MACLISP. This eliminates the CDC 6600 and the Burroughs 6000 family, for example.

3. The use of LISP arrays for defining register arrays would cause problems in specification. This is admitted in the paper where a 64 Kword subset of the 32 bit address space is used because of MACLISP's inability to support arrays larger than 64 Kwords. A sparse matrix implementation could be one approach to solving this problem.

4. The assembler format is not properly defined. There is no mapping from the assembly language format shown in the instruction specification table to the representation stored in the memory registers. A front-end would be required to take assembly language statements and extract the operands (from text indicating the effective address calculation method) that are stored in the memory word.

5. All instructions fit within a single word; this is increasingly unrepresentative of modern computer architectures where the number of operands varies from

instruction to instruction. Processors which would be unable to be defined because of this limitation include most microprocessors and some minicomputers, such as the Prime 750.

6. LISP is not an intuitive language for specifying instruction set processors. It is possible to specify very similar architectures using completely different specification functions. The converse may also be true.

   The reasons for this are twofold: first is that LISP is not one single language but a group of dialects, each with their own peculiarities; specifications written in LISP would have no hope of being portable. Second is that LISP is a general purpose language with functional redundancy built into it; in different dialects of LISP there are three forms of choice function (COND, IF, CASE), a similar number of loop functions, and various methods of extracting items from lists and performing assignments. Special purpose languages, such as ISPS, have the benefit of being targeted at a specific application and being able to eliminate the redundancy in LISP.

If LISP is to be used as a specification language then the following aspects of its use should be standardised:

1. A non-redundant subset of LISP functions to be used when specifying an architecture.
2. The register specification technique (LISP arrays or sparse arrays).
3. The calculation of effective addresses.
4. The numeric precision of arithmetic and logic operations.

The final points made in [Cragon83] are that the functional specification in LISP may be expanded in detail as the model descends through the levels of abstract machine description. LISP shares this feature in common with VDL and as such has much to recommend its use as a specification language. It is also a significantly more flexible language for describing architectures than the traditional approaches such as ISPS and Pascal.

Even if the drawbacks of LISP were removed, the author feels that LISP is not an attractive description tool because of its visual style and textual density; people who are not used to LISP notation would find it an impossibly obscure way of specifying an architecture.

1.2.4 PASCAL

[Wakerly80] has suggested that the Pascal programming language, with some extensions, could be suitable for specifying computer instruction sets and points out that, although the extended Pascal has no more power or functionality than ISPS, it is a more familiar tool and so is more useful in teaching situations. The extensions are the following:

Numbers: unsigned binary, octal and hexadecimal numbers are recognised.

Data types: the data type BIT has been added to the language.

Arrays: Pascal has been extended to allow for subarrays, defined as "an ordered, contiguous subset of the array" to be referenced. Subarrays are restricted to one dimensional arrays.

Operators: the concatenation operator "¦" has been added. It produces a bit array the length of which is the sum of the lengths of the arrays that have been concatenated. The addition ("+") and subtraction ("-") operators have been extended to perform two's complement arithmetic on bit arrays.

Built-in Functions: The following standard functions have been added to the language -

BINT - converts a bit array into an unsigned integer.

BITS - converts a non-negative integer into a bit array of specified length.

BCOM - complements the elements of a bit array.

BSHL - performs a left shift on the elements of a bit array.

BSHR - performs a right shift.

BAND - performs a logical AND on the elements of two bit arrays of the same length.

BOR - performs a logical OR on two bit arrays.

BXOR - performs the exclusive-OR on the two arrays.

BADD - converts two bit arrays to unsigned integers and performs an unsigned addition upon them.

Type conversion: The elements of an expression with a mixture of bit arrays, integers and constants are converted to bit arrays before being evaluated. For assignment of a bit array to an integer, the bit array is converted to an integer before being assigned. For assignment of an integer to a bit array, the integer is converted to a bit array before being assigned.

In many respects Pascal is a good language for specifying ISP's. It is a mainstream language well enough known not to cause people too much trouble in comprehending descriptions. It has a rich set of data types and structures capable of expressing complex machine states. It is capable of structural abstraction with its TYPE facility and it is capable of defining functional behaviour of arbitrary complexity.

The language is reasonably compact. A fully functional specification of the PDP-8 architecture was 144 lines of Pascal code [Wakerly80] versus 175 lines of ISPS code [Siewiorek82] so the two are approximately equal in information density, the difference being attributable to differing coding styles.

The limitations of Pascal are mainly those ones designed into it by Nicklaus Wirth; its lack of flexibility regarding data type coercion, its lack of string handling facilities and its limitations with regard to input and output.

A more important flaw with the proposed extensions as they stand (for the purposes of this thesis) is that there is no facility for tying Assembly Language descriptions into the model of the architecture. Pascal is not a good language for performing that function largely because of its lack of string manipulation capabilities.

Pascal is intended to describe the ISP level only although, like ISPS, it is able to express lower aspects of the architecture.

Architectures which may be modelled are limited by the arithmetic precision and wordlength of the host architecture. In the paper this is defined to be 64 bits or greater and as such would be unlikely to limit the range of architectures able to be described by the language.

Like LISP, the extended Pascal computer description may be directly executed and evaluated, but unlike LISP it needs to go through a translation process first. Also unlike Cragon's LISP approach the extended Pascal system works purely at the ISP level and so an assembly language program must also go through a process of translation to turn it into a bit stream which is then loaded into the appropriate registers before execution.

In the description of the PDP-8 architecture given in [Wakerly80] the states of the machine are represented by variables while the behavioural aspects are represented by procedures and functions:

Example 1.14 - The PDP-8 Effective Address calculation procedure.

```
{Calculate Effective Address Register}
PROCEDURE CalcEAR;
  BEGIN
      IF IR [pb] = 0
      THEN    {page 0}
         EAR := 0 [0::4] ¦ IR [pa]
      ELSE        {current page}
         EAR := lastPC [0::4] ¦ IR [pa];
      IF IR [ib] = 1 THEN       {indirect address}
        BEGIN
          IF EAR [0::8] = 1 THEN
            MEM [EAR] := MEM [EAR] + 1; {auto increment}
          EAR := MEM [EAR];
        END;
   END;
```

A comparison of this example with Example 1.5 shows immediately the similarities and differences between extended Pascal and ISPS.

The major value of extended Pascal is in the wealth of data structures available and the resulting structural complexity that may be described along with the structural abstraction capability available with the TYPE facility. These two facilities are shared only with the programming language C. C has the additional advantages, though, of having flexible string handling facilities.

Pl/1 has more flexible string handling facilities and better I/O facilities than Pascal but lacks the data abstraction capability.

1.3 Summary

In this chapter I have stated the goal of this thesis and have described the way in which some existing languages contribute to this goal. Each language has been shown to be deficient in some particular way for our purposes: ISPS and Pascal are mechanistic languages without the language structures to support symbolic specification; VDL, though a powerful symbolic language for specifying interpreters, is not widely known and has a structure which is widely dissimilar to the mainstream programming languages; LISP is less powerful than VDL but has a similar functionality although the style is again sufficiently dissimilar to mainstream programming languages to be difficult to learn.

The goal of the thesis has been stated as being an attempt to devise a language which allows the symbolic definition of arbitrary ISP architectures. None of the languages described present a coherent model of symbolic ISPs although VDL comes close by subsuming the ISP model into its general model of language interpreters.

2 A Conceptual Model of Architecture

The model as formulated is intended to describe the parts of a computer that a running program "sees" at the level of symbolic machine instructions (the "assembler" level). There is a one for one correspondence between instructions at this level and the instructions executed by the physical machine but the detail of how the instructions are encoded is avoided and so the model is significantly simpler than other descriptive models such as ISPS. This view of architecture is oriented toward the software engineer.

In the conceptual model an architecture consists of four domains:

the instruction set domain
the register set domain
the access method domain
the data types domain

A domain is an autonomous component of an architecture; the name is drawn from an analogy with a four dimensional matrix where the instruction set, register set etc. each make up a single domain.

The execution of an instruction involves making changes to the register domain (also called the state space). Instruction execution starts from a known state in the register domain and continues until another position is reached which inhibits execution. Each domain is discussed separately below, as is the model of instruction loading and execution.

2.1 A Model of Instruction Execution

Fundamental to our model of execution is the concept that an "instruction execution cycle" is indivisible. This is actually the case in many computers, especially microprocessors, but not in some more complex computers, such as those with virtual memory.

The reason for this is that the model is sequential: the instruction cycle consists of loading the next instruction to be executed, checking the asynchronous instructions (interrupts etc.) and executing any which are valid, then executing the synchronous instruction which has been loaded. During execution of the instruction, no other instruction may be active.

There is no facility at all in the model for describing concurrent processing. All instructions are processed sequentially in the model, and the primitive operations within each instruction are executed sequentially. If concurrency does exist in the real architecture it may be converted to an equivalent sequential model.

The architecture starts in some arbitrary but known state in which the Instruction Pointer points at the first instruction. The instruction is interpreted and executed and the Instruction Pointer is modified to point to the next instruction to be executed. This continues until an instruction is executed which inhibits further interpretation and execution of instructions. Before the execution of each synchronous instruction any pending asynchronous instructions are interpreted and executed.

The above requires there to be a special register designated as the Instruction Pointer. There is only one of these at any point in time, although any register may act as the Instruction Pointer.

In an orthogonal model the instruction sequencing must be described in terms of access methods and register sets. Instruction sequencing is the specification of the method and order of accessing of instructions within the register space.

Instructions may explicitly modify the Instruction Pointer and thus cause changes in the normal flow of control. If the change in the flow of control is to be temporary (as in the case of a subroutine call followed by a return instruction) the current instruction must have available the address of the next instruction before the current instruction is executed. This is achieved in ISPS by assigning the value of the Instruction Pointer (PC) to a register called LAST.PC immediately before modifying the instruction pointer; in this instance the true instruction pointer is LAST.PC not PC .

2.2 The Register Set Domain

The register set domain represents the state space of an architecture. All locations explicitly addressable by a program plus those registers required to model external events or implicit internal events are contained in this domain.

The Register is the indivisible addressable unit for the architecture. A register has a single dimension of word size.

A Register Array is a contiguous group of registers with a generic name. The whole register array may be referred to by name alone while individual registers within the array may be referred to by the name followed by an expression yielding a positive index into the array. Registers are a special case of the register array.

In the model there is no distinction made between register arrays that are contained within the processor and those external to it. This adds versatility when considering non-Von Neumann architectures without adding undue complexity to the simpler architectures. This is because of the increased flexibility in such things as addressing and instruction location.

Example 2.1

The Intel 8051 microprocessor has three distinct address spaces - the ROM, on-chip RAM, and off-chip RAM. The same address may refer to any one of the three address spaces depending upon the value of a selection register; a program may be located in any or all of these address spaces. It is in this sort of architecture that the distinction between on-chip register arrays and off-chip register arrays is shown to be invalid.

It is convenient to divide the register domain up into three organisational classes:

- The "explicit" register array which is specified in the manufacturer's data sheets and is explicitly addressable by instructions. This is the programmable state space of the architecture.

- The "implicit" register array which is used by instructions though not specified in the manufacturer's data. Implicit register arrays are used to model state changes which are not part of the explicit register set. An example of its use is to model external interrupts.

- The "referred" register array which provides a mechanism for reordering the explicit and implicit register arrays into logical groups for addressing purposes. The specification of the mapping of referred register arrays onto the physical state space should be in terms of access method expressions in order to maintain flexibility. Referred register arrays complete the modelling of a computer's state space.

No distinction need be made in the model between any of the register classes. In the model they co-exist and operate in the same manner; their membership of an individual class is transparent to the operation of instructions. Data may be moved from any register to any other register as long as the normal register transfer restrictions are adhered to.

When data is transferred to any register it will also be implicitly transferred to all other registers which map onto the target register. This is an important point to remember during any implementation of the model.

All register arrays may be described in terms of two dimensions:

- word size - this describes the width of individual registers in bits. This size may be a bit count or it could specify the range of selectable bits within the word thus giving an implicit ordering to the bits in the register (indicating whether the most significant bit is the right-most or left-most bit).

- array size - this describes the number of registers in the register array. Again, the array size may be a size indicator or an address range indicator.

The word size dimension is distinct from the array size dimension in that the ordering of the bits contains an implicit ranking of importance with the largest numbered bit being defined as the most significant.

In addition, it is possible for register arrays to overlap. If this is the case then the intersection must be specified.

The model requires that any intersection between two register arrays must be complete: one must be a subset of the other. It is partly for this reason that referred register arrays are necessary.

Because there is no restriction on the number of registers which may refer to the same physical location, the partial intersection of two register arrays may be specified by making them both be semi-disjoint subsets of a third (referred) register array. This enables the model of register intersection to be implemented without undue complexity.

Implicit registers may be treated as real registers; they may often reflect a real register in the internal structure of the architecture. They are needed because there are state transitions in an architecture which are not reflected directly in the explicit register domain, but which affect the operation of the architecture and so must be modelled.

The register set domain is probably the least complex domain of the symbolic instruction set processor model.

2.3 The Instruction Set Domain

An instruction is a specification of the way in which the state space is to be modified. This specification is normally in the form of assignment operations with either unary or binary operators

Example 2.2

```
    assignment     R[n] <- R[ n-1 ]
    unary operator R[n] <- NOT R[n]
    binary op.     R[n] <- R[n] - 1
```

The right hand side of the assignment is an expression which yields a value; there is no limit on the complexity of the expression. The left hand side must be a register or the concatenation of several registers. If more than one operation is performed by the instruction then each operation is expressed individually with the separate operations forming a sequential list:

Example 2.3 - Z8000 LDD instruction

```
MEM [ R[n] ] <- MEM [ R[m] ];
        R[n] <- R[n] - 2 ;
        R[m] <- R[m] - 2 ;
        R[o] <- R[o] - 1 ;
           0 <- ( R[o] = 0 )
```

As there is no facility in the model for expressing concurrent operations those operations which are concurrent must be converted to a sequential model before being expressed.

Example 2.4 - 8085 exchange instruction:

```
xchg        HL <-> DE
```

To model this sequentially it is necessary to introduce a new register:

```
TEMP <- HL ;
HL <- DE ;
DE <- TEMP
```

There are two classes of instruction; they have been named synchronous and asynchronous.

A synchronous instruction occupies register space, is located by the Instruction Pointer and is interpreted and executed. It is the programmable component of an architecture.

An asynchronous instruction does not occupy register space; it is not dependent on being selected by the Instruction Pointer before being executed but is event driven. It is associated with an instantiation expression (a boolean expression generally involving a value in a control register) and is executed when that expression becomes true. It is generally limited to executing before the interpretation and execution of a synchronous instruction, although in some architectures some asynchronous instructions are able to break in on an executing instruction. Interrupts and traps are asynchronous instructions.

Before the interpretation of any synchronous instruction all pending asynchronous instructions must be executed. An asynchronous instruction is pending when its instantiation expression is true. Very often registers associated with asynchronous instructions and their instantiation expressions are not listed in the manufacturer's data; they may be implicit registers needed to satisfy the requirements of the model.

Asynchronous instructions often cause a temporary transfer of control and so they must be able to store the location of the next instruction (i.e. the Instruction Pointer) in order to return control to the original instruction sequence. This is why they are defined to occur before the next synchronous instruction. If this condition did not apply then it would be impossible for asynchronous instructions (such as an INT instruction) to alter the Instruction Pointer in a controlled manner to coordinate program execution.

In the event of more than one instantiation expression becoming true at the same time, the model is indeterminate. For this reason each asynchronous instruction must have a priority associated with it. The priority ordering may be explicitly encoded in the instantiation expression or it may be implicit in the ordering of the asynchronous instructions.

In any implementation of the above model of instructions the following components would be essential:

- a name for the instruction
- a description of the operations performed

In addition synchronous instructions require the following:

- a list of the access method combinations permitted (every element in the list would contain an access method name for each variable in the instruction)

- a list of data type combinations permitted
- An instruction template which describes how the instruction appears within the program text and lists the operands associated with it

Asynchronous operations require:

- An instantiation expression description

Depending on the needs of the implementation a way of distinguishing between synchronous and asynchronous instructions is necessary; the exact mechanism is not pertinent to the discussion of the model but rather to the language used to implement the model.

All operations performed by the instructions are specified by a small group of primitive operators from which more complex operations may be built. These operations are as follows:

addition	+	and	AND
subtraction	-	or	OR
multiplication	*	not	NOT
division	/	concatenation	||
modulus	MOD	exponentiation	**
assignment	<-	exclusive or	XOR
left shift	LSH	right shift	RSH
sign extension	EXT		

Note that the problem of differing data types has not been resolved. According to the model all data type information is contained within the data type domain but common experience with the above operators indicates that they are used only with specific data types.

A distinction needs to be made between operand data types within the model and data types associated with the primitive operators.

The primitive operators interpret a particular value differently depending on whether they are logical or arithmetic operators. All operators treat operands as vector values (the values are pure binary magnitude values with no sign component) except for the subtraction operator (-) which treats operands as two's complement numbers.

The only operator which causes the state space to be modified is the assignment operator (<-). All other operators are functional; they return a value which is a function of the operator as applied to the operands.

In concatenation, two or more registers of $\underline{n}$ bits are logically concatenated along the word boundary to produce a single register of $\underline{n}$ times $\underline{m}$ bits where $\underline{m}$ is the number of registers being concatenated. There is no requirement that any of the registers being concatenated be of the same word size but all register arrays must have the same array size. Thus two register arrays, one eight bits by four words and the other sixteen bits by four words may be successfully concatenated to form a single register array of twenty-four bits by four words.

The left shift operator propagates each bit in a register one position to the left. The original value of the left-most bit is lost and the value of the right-most bit remains unchanged.

With right shift each bit is simultaneously copied to the next bit to the right. The left-most bit remains unaltered and the original value of the right-most bit is lost.

Both left and right shifts are independent of the most significant bit polarity of the register. Left and right shift are unary operators.

Assignment is possible between registers of differing sizes. When the assignment is necessary between two different length locations then the value held in the source register is either truncated or extended as may be required to match exactly the size of the target register. The truncation or extension is with respect of the most significant bit.

Assignment recognises vector values only, so if the source register must be extended it will be zero extended. The EXT operator extends the most significant bit of the source register or expression to an arbitrarily long wordsize which is then truncated to fit into the target register.

Example 2.5

ACCA is an 8 bit register, ACCX is a 16 bit register:

```
after    ACCA <- &10001011;
         ACCX <- ACCA

ACCX will contain &0000000010001011

after    ACCA <- &10001011;
         ACCX <- ext ACCA

ACCX will contain &1111111110001011

after    ACCA <- &00001011;
         ACCX <- ext ACCA

ACCX will contain &0000000000001011
```

2.4 The Access Method Domain

When an instruction is interpreted and executed it may have variable operands. If so then there must exist an access method for specifying how each actual operand value is derived from the variable selector.

Each access method description contains an access method expression which indicates the transformations required to obtain the operands. The expression tree consists of register specifiers, constants, parameter substitutes and basic operators.

A register specifier locates an individual element of a particular register array.

Because the instruction operands are variables which are assigned specific values in a program, actual values must be substituted for the formal parameters of each instruction occurring within a program. Parameter substitutes are the formal parameters. When a program is executed by the architecture the formal parameters of the instruction are replaced by the actual parameters of the instruction and an operand constant is derived.

In modern complex instruction set computers, the number of operands for each instruction and even the number of components of each operand may vary considerably. This is because many modern architectures, especially microprocessor architectures tend to cram as much functionality into each instruction as possible. The following are examples of modern, complex instructions:

Example 2.6

Motorola 68000: LINK and UNLINK instructions

Intel 8086: REPT MOVS and LOOP instructions

VAX: CASE SOBGEQ INSQUE REMQUE

In many computers the state space may be modified as a side effect of using a particular access method; thus access methods are able (like instructions) to perform operations affecting the state space. At this point the difference between instructions and access methods becomes somewhat blurred (although the side effects are generally less complex than for instructions).

The primitive operations specified in <u>instructions</u> alter the register space in a manner independent of the access methods used whereas the side effects of <u>access methods</u> alter the register space in a manner independent of the instructions using them.

It is quite possible that both the access method and the instruction will alter the same registers. The alterations cannot be concurrent as that would make the system inconsistent, thus we require temporal information incorporated into the access method expression. This could be done by splitting the access method expression into three related components; the first component would contain operations performed before the instruction is executed, the second would contain the operand derivation expression, and the third part would contain the operations performed subsequent to the execution of the instruction.

Another approach would be to have a special identifier representing the derived operand value and to have an arbitrary sequence of operations of which one, and only one, must assign a value to the derived-operand identifier.

This model is designed to be equivalent to ISP architecture at the symbolic (assembler) level and so each instruction will indicate, for each operand, the access method associated with that operand. Because there are many assembler languages in the world (more than one per machine architecture) it is necessary for the access method model to contain a template which enables it to determine the access method being used by a particular instruction in a program and to extract the actual parameter values from the operand expression.

It is probably the implementation of the access method descriptions which will provide the greatest scope for variation in terms of implementation possibilities.

2.5 The Data Type Domain

Of the four addresses described earlier, the first three may take on a variety of data types. The data types of source and target and even source1 and source2 need not match although it is extremely rare for them not to.

Some common data types are:

- unsigned binary
- two's complement
- one's complement
- binary coded decimal (packed or unpacked)
- ascii (seven/eight bits)
- ebcdic (eight bits)
- floating point (a whole host of these)

Data types are significant to the architecture because they alter the side effects of an instruction (such as the status registers affected) and they alter the operations of instructions themselves.

In the model of the data type domain, there are no assumptions about the data type and any side effects caused by the use of a particular data type must be explicitly stated. Every value is nominally an absolute binary value. Once the instruction has been executed normally the data type expression is invoked to coerce the target to have the correct value for the appropriate data type. As the data type expression may be required to perform arbitrary manipulations on the target it must have the same expressive capacity as the instruction specification expression with the same primitive operations affecting the register space and possessing full parameter substitution capabilities.

The distinction between an instruction expression and a data type expression is marginal and it is quite possible to restrict the model to three domains by including both the instruction domain and the data type domain together. In fact this is normally done when specifying ISP's so that for each separate data type there is a separate instruction to perform any given function. This is demonstrated in the Motorola 68000 where the MOV instruction (for example) is specified separately for 8-bit, 16-bit and 32-bit data words.

In assembly language models of architecture the problem of data types as a separate domain is rarely apparent due to the restricted set of data types available and the tendency to specify a single permissible data type for each instruction. The data type domain has been included in this initial exposition of the model for completeness and may not be included in implementations of the model.

3 SADL - The Symbolic Architecture Description Language

SADL has been created to implement the model of computer architecture proposed in the previous chapter. There are some deviations from the model for the purposes of ease of implementation but the bulk of the language conforms to the model.

The full syntax of SADL is given in Appendix 1 but the following description utilises extracts from the syntax to illustrate the use of the language. The syntax uses extended Backus Naur Form where square brackets ([]) are used to indicate optional items and braces ({}) are used to indicate items that may be iterated zero or more times. In addition, parentheses are used to override the normal precedence associated with BNF. Terminal symbols of the language being defined are underlined for clarity. This is to help distinguish terminal symbols from non-terminals in the BNF description and is not part of SADL.

The SADL description consists of a processor specification (Pdescr) optionally followed by the execution cycle specification (Executor).

```
        <sadl> ::= <pdescr> [ <executor> ] .
```

Pdescr provides a symbolic description of the register set, addressing modes, and instruction set of the architecture. In situations where the instruction execution cycle must also be modelled, a description of the load and execute loop for the architecture may be provided by the executor section of SADL.

3.1 The Basics of the Language

The following components of SADL are so pervasive that it is necessary to explain them before a comprehensive description of the language and its relationship to the model of Chapter 2 is possible.

The alphabet of SADL consists of the ASCII character set from " " to "~" (ASCII characters 32 to 126) inclusive. All other characters are treated as spaces; SADL is a free format language except that end-of-line is a token separator.

Numbers are unsigned constants using decimal, hexadecimal, or binary representation.

```
<number> ::= <dec num> | <bin num> | <hex num>

<bin num> ::= & ( 0 | 1 ) { 0 | 1 }

<dec num> ::= <digit> { <digit> }

<hex num> ::= # <hdigit> { <hdigit> }
```

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<hdigit> ::= <digit> | A | B | C | D | E | F

A prefix of "&" signals a binary number while a prefix of "#" signals a hexadecimal number.

There are two classes of operator: the unary operators and the binary operators.

<unop> ::= + | - | not | lsh | rsh | ext | sizeof

+	positivity operator (redundant)
-	negation operator (two's complement)
not	logical complement
lsh	logical left shift by one bit (right most bit remains unaltered)
rsh	logical right shift by one bit (left most bit remains unaltered)
ext	sign extension to arbitrary length
sizeof	length of operand, either minimum number of bits necessary to store the value or the defined bit length of a register location

The unary operators have the highest precedence of the operators. All unary operators have the same precedence.

```
<binop> ::= + | - | * | / | ** |
            and | or | || | mod |
            <boolop>

<boolop> ::= = | > | < | >= | <= | <>
```

+	unsigned addition
-	two's complement subtraction
*	unsigned multiplication
/	unsigned division
and	logical product
or	logical sum
||	bit string concatenation
**	exponentiation
mod	remainder of division
<boolop>	these operators return one (1) if the relation between their operands is true otherwise they return zero (0).

The order of precedence for the binary operators is:

**	highest priority
* / mod	
+ -	
> >= < <=	
= <>	
and	
or	
||	lowest priority

The parameter substitution symbol serves the role of a placeholder indicating that some other text is to be substituted for the placeholder during evaluation. It is important to note that the substitution is textual and that the text is evaluated only after all substitutions have been performed.

```
<param substn> ::= $ [ <dec num> ]
```

The optional decimal number is necessary where there is more than one parameter substitution within a given context.

Example 3.1 - $1 <- $2 + $3

The value expression is central to the functional specification of architectures.

```
<value exprn> ::= [<unop>] <value group>
                  { <binop> [<unop>] <value group> }
```

The value expression describes how a value is obtained from an architecture to be used in statements and conditions. It is the SADL analogue of an arithmetic expression in a programming language such as Pascal or PL/1. The value expression yields a bit string of the minimum length necessary to represent the value produced.

The value group specifies where each value is obtained from:

```
<value group> ::= <reg selector> |
                  <param substn> |
                  ( <value exprn> ) |
                  <number>
```

The parentheses are for altering the order of evaluation of value expressions from that required by the operator precedence rules. The <param substn> indicates that a value is to be substituted into that position during evaluation; the value must be either a numeric constant or a valid name within the specified context.

The <reg selector> specifies that the value is to be obtained from a particular element of a named register array:

```
<reg selector> ::= <r name> [ [ <value exprn> ] ]
```

The <value exprn> in brackets returns a numerical index that specifies which element of the register array contains the desired value. If the register array contains only a single element, then the selector expression may be omitted.

Example 3.2 - some value expressions:

```
REG1
REG1 [ A ]
REG1 [ $ + 6 ]
REG1 + REG2 [ $1 + 1 + REG3 ] - $2 - 4
REG1 * ( 4 + REG2 [ $1 ] )
$1 + #F
&0110
```

3.2 The Processor Description

The processor description associates a name for the architecture with three sections which describe respectively the register set, the access methods (addressing modes) and the instruction set. Each of these is referred to as a domain.

```
<pdescr>  ::=  architecture  <ar name>  is
                      <rset domain>
                      <amset domain>
                      <iset domain>
```

Each domain consists of a header followed by one or more domain entries. None of the domains may be omitted, nor may any of them be null, or the architecture would not be capable of being programmed.

The first domain describes the register space of the architecture. All state variables defined for the architecture are described in this domain.

The second domain describes the access methods available to synchronous instructions in the architecture. It is broken up into two sections; in the first, the access methods themselves are declared while an optional sub-domain allows groups of access methods to be collectively referred to using a single name.

The final domain describes both the asynchronous and the synchronous instructions. It is broken up into two sub-domains with the first sub-domain, which describes the asynchronous instructions, being optional. The second sub-domain, the synchronous instructions, is mandatory.

3.2.1 The Register Domain

This section consists of a series of one or more register declarations of the form:

```
<reg defn>  ::=  <r name>  is
                           <dim exprn>
                           [  <mapping exprn>  ]
                           end
```

```
<dim exprn>  ::=  <array spec>  <word spec>
```

All register names must be unique within the register domain.

Typical register declarations would look like:

Example 3.3

```
MEM is [ 0 #FFFF ] < 7 0 > end
IO  is [ 0 #FF ] < 7 0 > end
```

A register array contains n elements where each element has a word length of m bits. The square brackets ([]) denote the size of the array while the angle brackets (<>) denote the size of words within the array.

3.2.1.1 The Array Clause

The array component of a register is described with:

```
<array spec>  ::=  [
                   [  <range bounds>  |
                      <cell list>
                   ]
                   ]
```

```
<range bounds>  ::=  <lower bound>  <upper bound>
```

```
<cell list>  ::=  <cell name>  {  ,  <cell name>  }
```

There are three legal ways of specifying the size of the array. The first one is to explicitly state the index numbers of the first and last elements of the array as in:

Example 3.4 - REG1 is [0 7]<> end

This indicates that REG1 is an 8 element array with the first element accessed by an index of 0 and the last element accessed by an index of 7; indices outside the range 0 - 7 are not valid.

The other method of defining the array parameters is to label each cell of the array explicitly, thus:

Example 3.5

```
        REG1 is [ CELL1 , CELL2 , CELL3 , CELL4 ]<> end
```

REG1 is defined to be a four element array where CELL1 refers to the first element of the array and CELL4 refers to the last element of the array.

It was shown in the description of <u>register selectors</u> that it is necessary to allow the elements of REG1 to be accessed using a numerical index as well as by cell name. For this reason an enumerated register array is defined to have an index of 0 for the first element and a final index of <u>n</u> where <u>n+1</u> is the number of elements in the array; so for REG1 the element REG1 [CELL1] is the same as REG1 [0] and REG1 [CELL4] is the same as REG1 [3].

It is important to note that the names used to enumerate the elements of a register array have a scope restricted to that register. This means that the same name may be used for a register name as well as being repeatedly used within different register definitions to enumerate elements.

Example 3.6 - for the following definitions:

```
REG1 is [ A, B, C ]<> end
REG2 is [ D, E, A ]<> end
A    is [ F, A ]<> end
```

The name A used to enumerate REG1 is distinct from the name A used to enumerate REG2 and both are distinct from the register array A and its enumerator name A. It is not legal to use the same name twice within the enumeration list of a single register array so:

Example 3.7 - `REG1 is [ A, X, B, X, C ]<> end`

is not correct because a reference to REG1 [X] would not be sufficient to locate an individual element. However, as long as the X elements are never going to be uniquely accessed this problem would never arise. This situation is common among computer architectures for special purpose registers such as the Status register in which not all elements of the array have meaning. In this context non-unique element enumerators could be valid but the author feels that the ambiguity created is undesirable in a formal specification language.

When a register array has only one element its index need not be supplied and the shorthand form [] may be used; thus:

Example 3.8 - REG1 is []<> end

specifies a register array with a single element (called simply a register). This is the third legal way of describing the array dimensions of a register.

3.2.1.2 The Word Clause

The word clause describes the dimensions of the individual elements of a register array:

<word spec> ::= $\underline{<}$ [<msb> <lsb>] $\underline{>}$

There are two ways of specifying the word size of a register. The first method is to explicitly state the number of the most significant bit and the number of the least significant bit of a register. Thus:

Example 3.9 - REG1 is []<7 0> end

specifies a register with an 8 bit wordsize where the most significant bit is numbered 7 and the least significant bit is numbered 0.

Note that the order of significance for the array specification and for the word specification are different. For the array specification the elements have an increasing significance from left to right but the bits specified by the word clause have an increasing priority from right to left. The reason for this is that it follows the convention of custom; it is almost universally adopted that the bits of a word (like the digits of a number) have an increasing significance as they progress to the left. For arrays though, the elements are conventionally ordered so that the first element is on the left and the index number of the elements increases as the array is scanned to the right; this is in accordance with the way people write and parse text. As the choice of ordering is arbitrary the author settled on a form which is consistent with the way people are used to treating the respective structures. This is in contrast with ISPS in which both the word and the array description are based on the left most element/bit being associated with the lowest numerical value.

The second method of specifying the wordsize of a register is to omit the explicit delimiters of the word as in:

```
Example 3.10 -         REG1 is [] <> end
```

This indicates a register size of one bit and is directly analagous to the shorthand form for the array specification.

3.2.1.3 The Mapping Clause

The mapping clause describes the area of intersection between the array being defined and those other registers which occupy the same register space in the architecture:

```
<mapping exprn>  ::=  maps
                      <r mapdef>
                      {  ||  <r mapdef>  }

<r mapdef>  ::=  <r name>  [  <m array spec>  ]

<m array spec>  ::=  [  <init addr>  <term addr>  ]

<init addr>  ::=  <number>  |  <cell name>

<term addr>  ::=  <number>  |  <cell name>
```

The following illustrates the possible use of mapping to define register synonyms:

Example 3.11

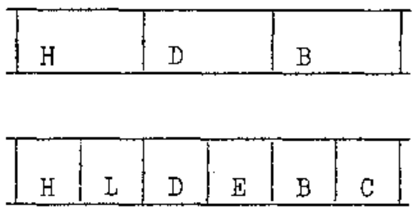
```
        B is []<7 O> end
        C is []<7 O> end
        D is []<7 O> end
        E is []<7 O> end
        H is []<7 O> end
        L is []<7 O> end
        RP is [ B, D, H ]<15 O>
                            maps C || B || E || D || L || H end
```

The virtual register array RP is declared as being a three element array where each element has a word length of 16 bits and this is defined to map onto the concatenation of the 8-bit registers B,C,D,E,H,L.

This method of specifying the intersection between registers differs significantly from the model proposed in chapter 2. The mapping mechanism is much simpler and allows a straightforward implementation with only a small loss in flexibility; the new mechanism is effectively a subset of the access method expression mechanism and may be expanded in a susbsequent version of SADL.

To perform this mapping, the model of a register array in chapter 2 is expanded so that, in addition to being an array of n elements where each element has m bits, we must view the register array as a contiguous bit stream from the least significant bit of the first element to the most significant bit of the last element of the register array. Thus, for RP (above) bit 15 of RP[B] is adjacent to and one position less significant than bit 0 of RP[D].

The concatenation of registers C ¦¦ ... ¦¦ H where C is the least significant register and H is the most significant register may then be mapped to RP by a simple superimposition of bits:



```
RP[ B, D, H ] maps

C ¦¦ B ¦¦ E ¦¦ D ¦¦ L ¦¦ H end
```

To simplify the implementation there are some restrictions that need to be enforced.

1. All registers named in the mapping expression must have been previously defined; this is necesary to allow one pass validation of the mapping expression.

2. One or more target registers or register arrays must map to each element of the source register array and must map exactly on word boundaries.

This means that for each element of the source register there are exactly _n_ target register elements with no bits in either the source or the target registers remaining unassigned.

3. Closure is enforced. All elements of a register array which is mapped _must_ be assigned to target registers.

Registers arrays C, D, E (below) illustrate the possible combinations allowed for mapping. Essentially, the rule is that the number of contiguous bits represented by the mapping expression (the concatenation of registers and part registers) must be equal to _m_ * _n_ where _m_ is the number of elements in the source register array and _n_ is the length (in bits) of each word in the source register array.

Example 3.12

```
A is []<7 0> end
B is [0 3]<7 0> end
C is []<7 0> end
D is [0 1]<7 0> maps B[1 2] end
E is [0 4]<7 0> maps A ¦¦ B end
F is [1 4]<7 0> maps A ¦¦ B[0 2] end
G is [0 2]<15 0> maps A ¦¦ B ¦¦ C end
H is [0 1]<15 0> maps C ¦¦ A ¦¦ B[1 2] end
```

As shown above, a subset of the elements of a register array may be involved in the mapping expression; where this is the case the first number or identifier is the first element included in the mapping and the second number or identifier is the last element included in the mapping. All elements between the first and last indicated elements are included in the mapping.

In addition to all the explicit registers (those directly accessible to the assembly language programmer) there may be registers which are either implicit (as in the case of the Interrupt Enable register on the Intel 8080) or are necessary to define the behaviour of certain aspects of the architecture (such as external inputs, interrupt lines, reset lines etc.). These implicit registers are included in the declared register set of the architecture and use the same syntax and semantics as explicit registers.

3.2.2 The Access Method Domain

Once all the registers of an architecture have been defined, the access methods, which describe the derivation of operands, must be defined. The Access Method domain contains a series of one or more access method declarations; these declarations define all access methods available to the architecture, their functionality, their parameters and how the values associated with those parameters may be extracted from the operand field of an assembly language program.

When all access methods have been declared, an optional subsection of the Access Method domain may be declared. This section is called the Access Method Class and is an organisational tool to enable a group of access methods to be referred to by a single name. This reduces the amount of coding required for architectures which have regular instruction sets with large numbers of access methods.

3.2.2.1 The Access Method Description

The Access Method domain consists of one or more access method declarations of the form:

```
<am descr>  ::=  <am name>  is  <am exprn seq>
                            from  <template>
                            [  size  <bitsize>  ]  end
```

An access method description associates a name with a sequence of access method statements, a template indicating how the values of the operands are to be extracted from the operand field, and the additional length of the instruction attributable to the selection of the particular access method.

With most microprocessor based architectures the instructions are of varying length each dependent upon the choice of access method used for the instruction. The <bitsize> is the number of bits by which the instruction length is increased by selection of the specific access method.

The <template> is a pattern matching tool whereby the text of the operand is extracted from surrounding constant text, which serves merely to indicate which of several potential access methods has been selected.

The Access Method Statement

The access method statement sequence is a series of assignment statements of which one and only one derives the operand for an instruction; the others cause side effects to the use of the access method.

The syntax of the access method statement is:

```
        <am exprn seq> ::= { <am assign stmt> ; }
                           <am param stmt>
                           { ; <am assign stmt> }

        <am param stmt>  ::=  OPERAND
                              <-
                              <dest selector>
```

```
<am assign stmt>  ::=  <reg selector>
                       <-
                       <value exprn>


<dest selector>  ::=  <dest exprn>
                     { || <dest exprn> }


<dest exprn>  ::=  <reg selector>  |
                   <param substn>
```

For statements which describe side effects the destination of the assignment must always be a register and the source is always a value expression.

The assignment which derives the operand is recognised by the presence of the keyword OPERAND as the destination of the assignment; the source is a destination selector which means that OPERAND is assigned either a concatenation of register locations where operands may be extracted from or placed, or the explicit value of the operand parameter as extracted from the template. The latter represents the access method known as immediate addressing, where the instruction operand is part of the instruction itself.

The statements are temporally ordered, so that the first statement occurs before the second statement which occurs before the third statement and so on. Statements defined before the assignment of

OPERAND represent side effects preceding the derivation of the access method operand and statements following the assignment of the OPERAND represent succeeding side effects.

The Template

The template is a series of one or more parameter substitutions optionally surrounded by arbitrary text:

```
        <template> ::= <const item> | <param substn>
                       { <const item> | <param substn> }

        <const item> ::=  <special char>  |
                          <identifier>  |
                          <number>
```

Any non-blank text may surround the parameters with the exception of the "$" symbol which must be represented by "$$". If there is more than one parameter substitution then they must be numbered uniquely.

Example 3.13

Motorola 68000 pre-decrement access method is indicated by:

```
-(A$)
```

where the $ (or $<number>) indicates the portion of the text which is the actual operand value.

For an actual operand of the form:

```
-(A6)      the operand value is 6.
```

3.2.2.2 The Access Method Class Section

The Access Method Class section is intended to be an organisational mechanism for referring collectively to a group of access methods. It is a shorthand form which associates a name with several access methods.

```
<am class>  ::=  access classes :
                 <amc descr>
                 {  ;  <amc descr>  }
```

```
<amc descr>  ::=  <amc name>
                  is
                  <am name>
                  {  <am name>  }
```

The only restriction necessary is that the AMC is viewed as part of the Access Method domain and for this reason the AMC names must be distinct from the names of the individual access methods.

Whenever an access method name occurs in the access method field of an instruction it is equivalent to listing the access methods named by that access method class within the field; in this sense it can be viewed as a macro-definition.

3.2.3 The Instruction Domain

The Instruction domain consists of two sections. The first section is optional; this is the Asynchronous Instruction Set and describes all instructions which conform to the model of asynchronous instructions proposed in Chapter 2.

The second section, which is necessary for the architecture to be programmable, describes the explicit, synchronous instruction set of the architecture. The instructions described in this section conform to the Chapter 2 model of synchronous instructions.

Common to both forms of instruction is the instruction statement sequence. This is a set of instruction statements which cause changes to the register space of the architecture. Because they are common to both instruction sets they are described first.

3.2.3.1 The Instruction Statement

The instruction statement defines the functionality of the assembly language instructions in terms of the operator set of SADL. There are four forms of instruction statement:

```
<istmt>  ::=  <assign stmt> |
              <cm stmt> |
              <cond stmt> |
              <loop stmt>
```

Instructions may be temporally ordered as an instruction sequence:

```
<istmt seq> ::= <istmt> { ; <istmt> }
```

The assignment statement (<assign stmt>) effects changes to the register space by assigning the result of a value expression to a register or concatenation of registers.

```
<assign stmt>  ::=  <dest selector>  <-  <value exprn>
```

Parameter substitutions may occur on either the right or left hand sides of the assignment operator. Parameter substitutions within the value expression were described in Section 3.1 .

The destination selector must be either a register or a concatenation of registers.

```
<dest selector> ::= <dest exprn> { || <dest exprn> }

<dest exprn> ::= <reg selector>   |   <param substn>
```

Any parameter substitution that occurs within a destination selector must evaluate to a register selector. It is possible to determine the validity of the parameter substitution as all access methods must have been defined before the instruction domain is evaluated.

Note that for asynchronous instructions, parameter substitutions are not legal because there are no instruction operands to substitute into the value expression or the destination selector.

The codem statement (<cm stmt>) invokes the named code macro:

```
<cm stmt>  ::=  do  <cm name>
                [ ( <param exprn>
                    { , <param exprn> }
                  ) ]


<param exprn>  ::=  <number> |
                    <reg selector> |
                    <param substn>
```

The code macro declaration is described in Section 3.2.3.3 (below); its use is analagous to that of a procedure call in Pascal though its operation is not. The parameters are textually substituted into the macro for evaluation purposes.

The conditional statement allows a sequence of instructions to be performed dependent upon a condition as expressed by a value expression:

```
<cond stmt>  ::=  if  <value exprn>
                  then  <istmt seq>
                  [ else  <istmt seq> ]  endif
```

If the value expression produces a non-zero result then the condition is deemed to be true, otherwise the condition is deemed to be false. Alternatively, the conditional statement may be used to select between two separate instruction sequences depending on whether the result is true or false.

The loop statement causes a sequence of statements to be repeatedly executed while the value expression yields a non-zero result.

```
        <loop stmt> ::= while <value exprn>
                           do <istmt seq> done
```

The value expression will generally be a register value which must be explicitly modified within the loop in order to terminate the repetition. The value expression is evaluated before execution of the instruction statement sequence parenthesised by the do and done keywords so that the minimum number of times the loop is executed may be zero.

3.2.3.2 The Asynchronous Instruction Set

The Asynchronous Instruction Set consists of a series of asynchronous instruction declarations:

```
        <asynch instr>  ::=  <i name>  is  <istmt seq>
                                       upon  <value exprn>
                                       end
```

The instructions model events and are hardwired into the architecture. They are described indirectly by the manufacturer under such headings as interrupt handling, reset operations etc.

The order of definition of the Asynchronous instructions is important because the priority of evaluation is based upon the ordering. This departs from the model in Chapter 2 because the priority encoding is implicit in the order of declaration rather than being an explicit part of the instantiation expression.

The first asynchronous instruction defined has the highest priority while the last instruction defined has the lowest priority. No two instructions can have the same priority; this is because the instructions are executed upon the occurrence of an event (such as external interrupts, or overflow from an addition) and if the priority of all possible concurrent events was not strictly defined then the architecture would exhibit nondeterministic behaviour when two or more events occurred simultaneously.

Though asynchronous instructions do not occupy register space they do alter it since the body of the asynchronous instruction is a sequence of assignment statements.

There are no parameters and therefore no access methods associated with asynchronous instructions.

The Instantiation Expression

The activation clause of an asynchronous instruction (signified by the keyword UPON) defines the condition which signals that the instruction may be executed. The instantiation expression is a value expression where a non-zero result indicates a current instruction.

Physical events external to the architecture, such as an interrupt line going low, are modelled within the architecture by the presence of registers which contain values representing the state of the external components. This is consistent with the abstract view of the architecture. Thus an interrupt pin on a processor may be modelled as a register which is 0 when the register is not asserted and 1 when the interrupt has been asserted.

Note that there is no implicit resetting of event registers within the architecture; either the register is reset externally when the event is no longer true (as in the case of signal-level-based interrupts) or the register must be explicitly reset by the asynchronous instruction handling the event (as is the case with signal-transition-based interrupts). The distinction is between the use of registers to model states (pin level high, pin level low) and discrete events (a transition has occurred signalling an interrupt).

3.2.3.3 The Synchronous Instruction Set

The Synchronous Instruction Set consists of a series of declarations of code macros and assembler instruction descriptions. All code macros must be declared before the first synchronous instruction declaration:

```
<synch domain>  ::=  synchronous
                     instructions  :
                     [  <codem list>  ]
                     <synch instr>
                     {  <synch instr>  }
```

The Code Macro List

The code macro list consists of a sequence of all code macros defined for the architecture:

```
<codem list>  ::=  <code macro>  {  <code macro>  }

<code macro>  ::=  codem  <cm name>
                   [  <cm param>  ]
                   is  <istmt seq>  endm
```

```
<cm param>  ::=  ( <param substn>
                     { <param substn> } )
```

A code macro declaration associates a name with a sequence of instruction statements. The code macro may have parameters; if so, the text of the actual parameters is substituted for the formal parameters, which are enclosed in parentheses. The normal rules which apply to instruction statements elsewhere in the synchronous instruction set also apply here.

The code macro is a way of reducing the amount of coding necessary to describe an architecture by extracting commonly used sequences of instructions and referring to them by a single name. Formal parameters are included because it was found that sequences of instructions tended to recur but utilised different registers.

Example 3.14 - Motorola 6800 instructions:

```
ADDA      is   ACCA <- ACCA + $1
ADDB      is   ACCB <- ACCB + $1
```

both cause the same sequence of operations to be performed upon the status bits of the processor as side effects but the registers in question are different in each case. With parameters it is possible to avoid the duplication of effort necessary to describe the architecture.

Example 3.15 - the <u>sign</u> macro for the M6800 would be:

```
CODEM sign ($1) IS
    ccr[n] <- ( $1 and #80 ) = 0
    ENDM
```

and its invocation would be:

```
DO sign (ACCA) ;
DO sign (ACCB)
```

<u>The Synchronous Instruction</u>

Each synchronous instruction associates a unique (within the instruction domain) name with a sequence of instruction statements.

```
<synch instr>  ::=  <i name>  is
                    <istmt seq>
                    [  from  <template>
                       using  <amlist>  ]
                    [  size  <bitsize>  ]  end
```

The instruction statement sequence has already been described.

The optional "from" clause specifies how operands are extracted from the operand field of the instruction, using the template mechanism described in Section 3.2.2.1, and the combinations of access methods that may legally be used with the instruction.

The optional "size" clause indicates the length, in bits, of the instruction before taking into account the extra length necessitated by the use of some of the access methods.

Parameter Substitution

When an assembly language instruction has operand variables there must be a "from" clause in the instruction declaration to indicate how the operand values may be extracted from the operand field text and also a "using" clause to indicate what combinations of access methods are legal for the instruction.

The operands are extracted from the operand field using the template mechanism as used for access methods. Again, the substitution is a textual one; all interpretation of text is performed in the value expression and the destination selector expressions.

The access method list is a series of tuples where the number of elements in the tuple equals the number of parameter substitutions in the template. Each tuple is a series of access method names or access method class names:

```
<amlist> ::= <am tuple> { ; <am tuple> }


<am tuple> ::= <am name> | <amc name>
                    { <am name> | <amc name> }
```

The following two examples illustrate the declaration of instructions:

Example 3.16 - Motorola 6800 ADDA instruction declaration

```
ADDA is
          ACCA <- ACCA + $
     from $
     using IMMED8;
           DIR;
           INDEX;
           EXTND
     size 8 end
```

The above instruction declaration states that the instruction named ADDA assigns the sum of register ACCA and the operand variable ($) to the register ACCA. The operand field has no text surrounding the operand and there are four access methods which are valid for the instruction. The instruction is 8 bits long plus however many bits are contributed by the different access methods.

Example 3.17 - Intel 8085 MOV instruction

```
        MOV is
                $1 <- $2
            from $1,$2
            using REG8 REG8;
                  REG8 INDIRECT;
                  INDIRECT REG8
            size 8 end
```

In this instruction there are two operand variables separated by a comma in the operand field of the instruction. The value of the second operand is assigned to the first operand, which must therefore be a destination selector. The size is 8 bits.

There are three valid combinations of access method. There is a one-to-one correspondence between the parameter substitutions and the elements of each tuple. In the above example this means that the text before a comma in the operand field of the instruction is passed to the access method REG8, or INDIRECT while the text following the comma is passed to the access method INDIRECT or REG8.

3.3 The Executor Description

The executor section of SADL is optional and necessary only when the load and execute cycle of an architecture is to be modelled; this would be the case whenever a simulation of the architecture is to be carried out.

The executor defines an implicit loop which loads the instruction and then executes it.

```
        <executor> ::= executor
                       [<istmt seq>]
                       load  <reg selector>
                       [<istmt seq>]
                       exec
                       [<istmt seq>]
                       end
```

The load keyword specifies where the next instruction is located while the exec keyword causes the instruction to be evaluated. Before and after the load and after the exec an arbitrary number of primitive instructions may be specified for various housekeeping chores that are not part of individual instructions, such as incrementing the instruction pointer.

The asynchronous instruction list is scanned as the first action by exec. This is in accordance with the model of Chapter 2.

3.4 Using SADL

SADL has been used to describe four complete architectures and a fifth architecture has been partially explored. The four fully described architectures are the Intel 8085 and 8086 microprocessors, the National Semiconductor SC/MP, and the Motorola 6800. The 8085 SADL description is included as Appendix 2 of this thesis.

With the exception of the 8086, all the above architectures are first generation 8-bit microprocessors. The architecture which was partially explored was the Motorola 68000.

Experience with the 8086 and 68000 architectures has indicated possible shortcomings in the language that require further study.

Access method specification in SADL can quickly become unwieldy because of the lack of variable length data structures or data typing. Both the 68000 and the 8086 allow structures of differing lengths for each of their major access methods and this translates into many more access method specifications than are desirable; for both architectures the number of access methods could be cut in half if a separate clause indicating the possible lengths of the operand were included in SADL.

The 8086 also indicates that SADL's simple approach to the calculation of instruction length may not always be satisfactory. SADL calculates the instruction length as the sum of the length components of the instruction and the access methods it uses. The 8086 instruction length is calculated on the combination of access methods used. For instance, a particular access method on the 8086 contributes 0 bits to the length of instruction when used by itself (in a single operand instruction) but contributes 3 bits to the length of instruction when used in combination with some, but not all, of the other access methods.

Finally, improvements may be necessary with regard to the parameter substitution mechanism if the practice, fostered by Intel, of allowing the components of a particular access method to be specified in any order, becomes common. This is not a problem of architecture specification but of the formats for the assembler. A fully generalised facility for accepting arbitrary assembly languages is beyond the scope of this thesis; the current approach by SADL is that it does not attempt to be able to handle any arbitrary assembler format but provides an interface which will allow the most widely used style to be recognised.

4 Building Programs from SADL

Build is a program written in Salford LISP version 17 [Salford83], on a Prime 750, with the purpose of generating data structures and functions for simulating a symbolic architecture specified using SADL.

Build works by parsing a SADL description using the top down approach and builds the data structures and functions as it parses.

LISP was chosen as the implementation language largely because of its ability to generate programs which may be executed within the LISP environment. The interactive debugging facilities provided by Salford LISP were also a major consideration.

Build is not a simulator. Rather it constructs the machine dependent routines upon which a simulator interface may be provided. This splitting of the simulation routines from the simulator interface means that the same interface may be used for any architecture.

4.1 Data Structures

LISP has a single data structure, the s-expression. There are two substructures of the s-expression : the atom and the list. An atom is a name; numbers are pseudo-atoms, as they are treated for the most part in the same way as atoms. A name is an arbitrary string of characters while a number in Salford LISP is held in Prime double precision floating point form. The distinction between names and numbers is the source of some problems in LISP as will be shown in Section 4.3 .

In SADL each register, access method, and instruction is named. This simplifies implementing the data structures in LISP as each domain may be a list of names where each name has associated with it several properties which are relevant to the domain.

Example 4.1

> an asynchronous instruction has a name, an instruction sequence and an instantiation expression. The last two items may be treated as properties of the name by using the LISP property list facilities.

```
Asynch name - property: instruction sequence
            - property: instantiation expression
```

There are six global variables containing lists of declared names:

```
REG_LIST@,
AM_LIST@,AMC_LIST@,
ASYNC_LIST@,
CODEM_LIST@, SYNC_LIST@.
```

LISP atoms are global except when declared explicitly within a PROG or as formal parameters of a LAMBDA or NLAMBDA expression. As SADL permits the same name to be used in each of the domains, a mechanism must be used to ensure that the properties assigned to each name by each domain do not conflict. This is done by using the Property List mechanism of LISP and naming the properties such that every property over the entire architecture is unique.

The Property List mechanism in LISP works by creating a list of property names and the values associated with that property; the first and all other odd-numbered members of the property list are the property names while the second and all other even-numbered members of the property list are lists of values associated with the property. The property list always has an even number of members.

The following describes the data structures for the individual domains.

4.1.1 The Register Domain

REG_LIST@ - a list of register array names:

(... H L HL B C BC ...)

Each name has a property list with the following properties:

LSW - index of the beginning array element

MSW - index of the terminating array element

LSB - index of the least significant bit

MSB - index of the most significant bit for the register

CELLS - an ASSOC list of registers and their values

MAPLIST - a list of register arrays that the named register array maps onto

MSW, LSW, MSB, LSB each have a hexadecimal number as their value.

An ASSOCiation list is a list of two-element lists such as (... (c 1) (b 2) (a 3)) which may be used by the LISP ASSOC function. ASSOC searches the lists trying to match the first member of each sublist with a specified value; the second element of the first sublist to successfully match the value is returned; this is peculiar to version 17 of Salford LISP. If none of the sublists match then NIL is returned. This is a fast and easy technique for implementing a sparse array, one solution to the problem of implementation restrictions with arrays that was described in Chapter 1 [Cragon83].

The contents of the list which represents the value of the CELLS property varies with time and the declaration of the register. If the register array contains only one element the list will be initially empty. If the register array contains more than one element then each of the elements will be present with an initial value of NIL:

```
( (B nil) (C nil) ...) .
```

As each register becomes initialised by being written to, the second value will be replaced by the binary representation of the value so that the above will become:

```
( (B &01001110) (C &00011101) ...) .
```

For the special situation where a register array contains instructions, the registers which hold those instructions have, as their value, a link to the instruction being held:

```
( (0 | ) (1 | ) ...)
     |      |
     |      |
     -->    -->( instr opr1,opr2 )
```

The above is for the case where MAPLIST is null (an empty list). If MAPLIST is not null then it contains a list of the register names that are mapped. Each member of the MAPLIST may be a name, in which case it must already be declared as a register, or it may be a three member list in which the first member is the predeclared register name while the second and third members are the lower and upper array elements for mapping.

Example 4.2

```
    - maps A || B || C end          gives       ( A B C )
    - maps A || B [0 3] || C        gives       ( A (B 0 3) C)
    - maps A || B [X Z] || C        gives       ( A (B X Z) C)
```

Where a register mapping exists, only the register(s) which are mapped <u>onto</u> hold actual values. Those which are mapped <u>from</u> have NIL value fields in their CELLS entries and their values are obtained by indirect reference to the target registers.

4.1.2 The Access Method Domain

This domain is represented by two lists:

AM_LIST@ - a list of access method names:

(DIR8 DIR16 IMMED8 ...)

AMC_LIST@ - a list of access method class names

Each name in the AM_LIST@ has three properties:

AM_MATCH - a template for extracting the operands from the text for a specific access method.

AM_SIZE - a hexadecimal value specifying how many extra bits long the instruction is because of the access method used.

AM_EXPRN - this is a LAMBDA expression which simulates the behaviour of the access method. The functional aspects are discussed in Section 4.4.5 .

The AM_MATCH list consists of a series of parameter identifiers separated by lists containing constant items which surround the parameters.

Example 4.3

```
- 68000  (A$)+      AM_MATCH:  ( ( "(" A ) $ ( ")" + ) )
- 8080   $                  :  ( $ )
- SC/MP  $1($2)             :  ( $1 ( "(" ) $2 ( ")" )
```

AMC_LIST@ has a single property AM_CLASS. The value of the property is simply a list of the access method names which are considered to be part of the access method class.

4.1.3 The Instruction Set domain

This domain has three lists representing subdomains:

ASYNC_LIST@ - a list of asynchronous instruction names;

CODEM_LIST@ - a list of code macro names;

SYNC_LIST@ - a list of synchronous instruction names;

The names in ASYNC_LIST@ have two properties associated with them:

ASYNC_EXPRN - a PROGN which implements the instruction sequences defined for the asynchronous instruction.

UPON - the value expression which determines whether the instruction is able to be invoked or not.

It is essential that the ASYNC_LIST@ contains the instructions in the order that they are declared as the simulator should pass down the list evaluating the UPON property value until a non-null value is returned upon which the ASYNC_EXPRN is invoked. If the order of the asynchronous instructions is not maintained then the implicit priority contained within the declaration order is lost.

Each name in CODEM_LIST@ has a single property associated with it: CM_EXPRN. The value is an NLAMBDA expression which simulates the operation of the code sequences specified in SADL. The NLAMBDA is necessary because text is being passed which must be substituted into the code macro when it is executed.

SYNC_LIST@ names have four properties:

I_SIZE - the size in bits of the instruction (stored in Hexadecimal)

I_MATCH - a template for recognising the instruction and extracting operands. The template has the same format as AM_MATCH.

AM_LIST - a list containing one or more lists. Each sublist contains the names of the access methods for the instruction's operands.

SYNC_EXPRN - a LAMBDA expression which simulates the operation of the instruction.

4.2 Constructing Tokens

The functions GET_CHAR and GET_TOKEN provide a clean interface through which the remainder of the parsing routines may obtain the next valid token. After an initial call on GET_TOKEN, the next valid token will always be available as well as the next character subsequent to that token.

There are three classes of SADL token:

Identifiers - all strings starting with a letter and containing only letters, numbers or the characters ".", "$" and "_" . Identifier tokens include SADL keywords.

Numbers - any string of characters conforming to the SADL syntax for numbers.

others - any string of characters forming valid SADL tokens but are not included in the above two categories.

The function GET_TOKEN skips over leading blanks, and uses the first non-blank character encountered to select the appropriate token building routine. The function has a single input parameter which may be used to restrict the range of tokens that may be recognised. If the input parameter is not specified then every token that is successfully constructed will be returned as a valid token. If the input parameter is the atom ID then the token that is constructed must be a member of the set of valid identifiers. If the input parameter is the atom NUMBER then the token must be a member of the set of valid numbers. If the input parameter is any other atom then the token and the parameter must be the same.

Example 4.4

```
        (GET_TOKEN) with <token> returns <token>
        (GET_TOKEN 'ID) with <identifier> returns <identifier>
        (GET_TOKEN 'NUMBER) with <number> returns <number>
        (GET_TOKEN 'end) with "end" returns "end"
```

If the token is not valid, either because it is not a valid SADL symbol or because it is not of the expected type as indicated by the input parameter, then a value of NIL is returned by GET_TOKEN otherwise the token is returned. In either case the position of the input stream is updated.

The token, regardless of whether or not it is valid within GET_TOKEN, is stored in the global variable TOKEN@ for access by the parsing routines. If the character stream was not a legal SADL token then TOKEN@ will be NIL.

4.3 Handling Symbolic Numbers

A major problem in modelling arbitrary architectures with any programming language is the possible inadequacy of number representation in the language. In Salford LISP numbers allow exact representation of integers up to 2 ** 45. While this precision may be adequate for the majority of architectures it is unable to represent all architectures (e.g. Burroughs B6700, CDC Cyber series). To overcome this problem all numbers are stored in Build as names and are manipulated symbolically. This does have an adverse effect on performance and for this reason, Build currently performs all arithmetic in decimal. With the ability to compile LISP expressions it would be feasible to perform the arithmetic symbolically. A bug in the version of LISP used to develop Build prevented use of the compilation facility.

The three classes of number (binary, hexadecimal and decimal) are stored the same way that they are represented in SADL: a binary number is prefixed by "&" and a hexadecimal number is prefixed by "#"; decimal numbers have no prefix.

The prefixes for binary and hexadecimal numbers cause them to be treated as names rather than numbers by LISP so their manipulation is straightforward. Decimal numbers are a significant problem because of LISP's distinction between names and numbers. The arithmetic operators in Salford LISP only work on atoms stored in numeric format; 6 is a number whereas "6" is not. If a list of numeric digits is imploded into a single atom (1 2 3) -> 123 then the atom is treated as a non-numeric atom. On the other hand, if a name consisting of digits only is exploded then the digits in the list are converted to numeric format even though the original atom was not.

This inconsistency has led to the inconsistency in the current version of BUILD, that hexadecimal and binary numbers are non-numeric atoms while decimal numbers are numeric atoms. This decision is partly because it was the easiest to implement and partly because it results in improved performance. The disadvantage is that while the precision of the binary and hexadecimal numbers is unlimited, the precision of decimal numbers is limited.

With the three different types of number, the system needs to be able to convert numbers from one format to another. This is performed by the function CONVERT.

CONVERT has two input parameters. The first parameter is the TO_TYPE argument indicating what type of number is to be returned. Values are: BIN to return a binary number; HEX to return a hexadecimal number and any other value returns a decimal number. The second input parameter is the number to be converted. The number in the required format is returned as the value of CONVERT.

4.4 Building LISP functions and PROGs

The construction of LISP functions, PROGs and PROGNs is very straightforward since both LISP expressions have very well defined and similar constructs.

For functions it is:

```
(LAMBDA <parameter list>  <expression list> )
```

For PROGs it is:

```
(PROG <local variable list> <expression list> )
```

For PROGNs it is:

```
(PROGN <expression list> )
```

Before describing the techniques of constructing the above expressions it is necessary to show how a LISP expression can be generated from SADL instruction statements.

4.4.1 Converting Value Expressions to LISP

The function PREFIX converts infix SADL expressions into a prefix, LISP-oriented representation using the shunting yard algorithm.

```
Example 4.5 -          A + B * C - D

              becomes (- (+ A (* B C)) D)
```

The output has the form of a LISP expression. Because each of the SADL operators is a function in Build, the above expression may be evaluated to return a result.

The routines which generate the above expressions are the parsing routines VALUE_EXPRN and VALUE_GROUP. VALUE_EXPRN parses the right-hand side of an assignment statement and calls PREFIX with either an operator or the result of invoking VALUE_GROUP. This process follows the structure of the syntax definition for SADL.

4.4.1.1 The Value Group

A value expression is a series of value groups separated by binary operators with optional unary operators prefixing each value group. Each of the identifiers A, B, C, D in Example 4.5 is a value group.

A value group is either a register selector, a numeric constant, a parameter substitution or a parenthesised value expression.

Register selectors are the most complex members of the value groups. The formats which a register selector may take are:

```
        <rname>  or  <rname> [ <selector expression> ]
```

The general form of a register selector is the NLAMBDA expression:

```
        ( VALUE_OF <rname>  <selector> )
```

VALUE_OF is a function which extracts the value from the named register or from the specified element of the register array when <selector> is not null. It is also able to accept and return numbers; this is necessary when handling parameter substitutions. VALUE_OF performs the indirection necessary for registers which are mapped and always returns the binary representation of the register contents.

The selector expression may be a numeric constant, a symbolic constant when the register array is enumerated, or a value expression returning a value within the valid addressing range for the register array.

Parameter substitutions, when occurring as part of a value group, may represent either a register selector (as returned by the access method) or they may be numbers.

In either case they are prefixed with the function VALUE_OF which must be able to interpret the parameter and return the appropriate value. This is due to the necessity of being able to extract the value from a register selector.

Example 4.6 - 8080 MOV is $1 <- $2

```
      generates: (LAMBDA ($1 $2)
                     ( "<-" ($1) (VALUE_OF $2) )
                 )                 ^
                                   |
                                   \_________ value group
```

Value expressions are straightforward as they become LISP function expressions through recursive parsing:

Example 4.7 - 6800 NEGA is ACCA <- 0 - ACCA

```
        generates:  (LAMBDA ()
                        ("<-" (acca) ("-" 0 (VALUE OF acca) ) )
                    )
                         value groups
```

Decimal constants may be included directly into the LISP expression (the 0 in example 4.7 above) but binary and hexadecimal numbers must be quoted to force the name rather than the value cell of the name to be passed to the operator functions. Decimal constants may be quoted or unquoted.

4.4.2 The Destination Selector

Whereas the right hand side of an assignment statement represents a value, the left hand side represents a location, or a series of locations, where the value is to be stored.

In SADL the destination selector is a register or the concatenation of several registers. It may also be a parameter substitution which equates to a register selector.

The function DEST_SELECTOR produces the data structure representing the destination of an assignment. DEST_SELECTOR processes the current token and continues to build a destination list while the next token is the concatenation operator. When the token is a register the function REG_SELECTOR is called. This function returns either the register name or (if the register is an array) a list where the first member is the register name and the second is the selector expression.

Example 4.8 -

register:	REG_SELECTOR returns:
HL	hl
MEM [O]	(mem O)
MEM [HL]	(mem (VALUE_OF hl)
REG8 [B]	(reg8 b)

Note that in line four of the above example B is an enumerated cell name, not a register name.

DEST_SELECTOR produces a list of destinations in the order that they are specified:

```
Example 4.9 -     HL ¦¦ MEM [O] ¦¦ DE <- ...

    would return  (hl (mem O) de)
```

The NDEFUN "<-" scans the list in reverse order assigning the result of the value expression to the target registers. The function also handles the indirection due to register mapping so that the model remains consistent.

Where a destination is a parameter substitution it is evaluated in order to extract the real target registers.

4.4.3 SADL statements

Instruction statements in SADL have one of the following forms.

```
        <target> <- <value expression>

        IF <value> THEN <statement sequence>
                   ELSE <statement sequence> ENDIF

        DO <code macro> <parameters>

        WHILE <value> DO <statement sequence> DONE
```

The basic SADL statement is the assignment statement. Section 4.4.2 indicates how a value expression and a destination selector may be combined to construct the two sides of the assignment. It is then a matter of enclosing the two expressions within a list with the assignment operator.

The functions I_STMT, for instructions, and AM_STMT, for access methods, convert SADL statements into LISP expressions through the invocation of DEST_SELECTOR and VALUE_EXPRN. The local variables DEST and VAL hold the destination expression and the value expression respectively and the final statement:

```
( RETURN ( LIST '"<-" DEST VALUE ) )
```

combines them into a single LISP expression.

I_STMT is more complex than AM_STMT as it must provide for the IF, DO and WHILE statements as well as the assignment statement. The first token of the statement determines which is the appropriate function to handle a particular statement. The functions are described below.

4.4.3.1 COND STMT

This function constructs a conditional expression using Salford LISP's IF function.

COND_STMT has three local variables for temporary storage. They are IF_PART, THEN_PART and ELSE_PART. Each variable holds the expression for the appropriate clause of the IF statement. IF_PART holds the value expression. Both the THEN_PART and the ELSE_PART are lists where each member of the list is an expression corresponding to a single statement. They are of the form:

```
        ( ... ( statement2 ) ( statement1 ) )
```

The ELSE_PART may be null if there is no else clause to the IF statement.

Finally, when the IF statement has been successfully parsed the component parts are amalgamated into a single expression. The following is executed as the final statement of COND_STMT. The ELSE section (lines six and seven) is omitted if there is no else clause.

```
    (RETURN (LIST 'IF
                  (LIST 'NEQUAL
                       (LIST 'CONVERT '"'DEC" IF_PART)
                       0)
                  (REVERSE THEN_PART)
                  'ELSE
                  (REVERSE ELSE_PART)
            ))
```

Note that the THEN_PART and ELSE_PART lists are accumulated in reverse order and so must be reversed to produce a correct ordering.

The result of the value expression is converted to decimal so as to allow the inbuilt function NEQUAL to be used. This improves performance but restricts the values which may be tested.

The following example shows how the SADL if-then-else construct is converted to LISP:

Example 4.10 - 8080 JM $ instruction

The SADL IF statement

```
IF ccr [s] then
    pc <- $ endif
```

is expressed in Salford LISP as:

```
(IF (NEQUAL (CONVERT 'DEC (VALUE_OF ccr s)) 0)
   ("<-" (pc) (VALUE_OF $) )
)
```

4.4.3.2 CODEM STMT

The presence of the keyword "do" causes the function CODEM_STMT to be called. The next token is the name of the code macro being invoked while the presence of parentheses indicates actual parameters to the code macro.

Example 4.11

```
        do ACplus ( ACCA )
        translates to:  (DO_CM acplus acca)
```

DO_CM evaluates a parameter list containing the name of the code macro followed by the actual parameters to it. DO_CM is an NLAMBDA and so its arguments are not evaluated before being passed.

If there is more than one parameter to the code macro then all parameters must be combined in a list.

Example 4.12

```
        do SOME ( ACCX ACCY )       :  (DO_CM some (accx accy))
```

4.4.3.3 WHILE STMT

This function is the analogue of the COND_STMT. It makes use of Salford LISP's WHILE expression:

```
        ( WHILE <condition> <statement list> )
```

The function WHILE_STMT has two local variables WH_PART and DO_PART. The first takes the list returned by VALUE_EXPRN, which is invoked for the conditional expression of the While statement. The DO_PART is a list whose members each make up a SADL statement.

The expression returned is:

```
(WHILE (NEQUAL (CONVERT 'DEC <value exprn>) 0)
     <statement sequence>
)
```

The statement sequence should modify registers in the value expression otherwise the loop will execute indefinitely.

4.4.4 SADL Instructions in LISP

Now that the method of constructing SADL statements has been described it is possible to show an entire instruction. For example:

Example 4.13 - 8080 instruction

```
XCHG is TMP <- HL;
        HL <- DE;
        DE <- TMP
     size 8 end
```

This causes the following to be generated:

```
(LAMBDA ()
   ( "<-" (tmp) ( VALUE_OF hl ) )
   ( "<-" (hl) ( VALUE_OF de ) )
   ( "<-" (de) ( VALUE_OF tmp ) )
)
```

Example 4.14 - 8080 LDA $

```
LDA is A <- $
    from $ using DIR8
    size 8 end
```

generates the following function:

```
(LAMBDA ($)
   ( "<-" (a) (VALUE_OF $) )
)
```

The above is created by the function ADD_SYNC which builds synchronous instructions. Asynchronous instructions are different as they have no parameters and can therefore be PROGNs rather than LAMBDAs.

Local variables are: EXPRN_SEQ which holds the sequence of statement expressions; PARAM_LIST which contains the parameters declared in the "from" clause; AM_LIST and AM_TUPLE which are used to construct the access method tuple list.

When the instruction has been successfully parsed the following statement ties the components into a LAMBDA expression which is placed on the instruction's property list:

```
(APPEND (LIST 'LAMBDA (REVERSE PARAM_LIST))
        (REVERSE EXPRN_SEQ))
```

Note the use of the REVERSE function again.

The instantiation expression is a value expression which returns T or NIL depending upon its truth. It is extracted from the property list, evaluated and, if not null, the PROGN is extracted and evaluated.

4.4.5 The Access Method Function

Access methods have the form of LAMBDA expressions but in addition they have an internal PROG expression. This is because the SADL keyword OPERAND is treated as a local variable which takes on the text of the destination selector and is the value returned when the access method LAMBDA is called. The format is as follows:

```
(LAMBDA <parameter list>
   (PROG (OPERAND)
      <statement list>
      (SETQQ OPERAND <destination selector> )
      <statement list>
      (RETURN OPERAND)
   )
)
```

The SETQQ function quotes both its arguments and so assigns OPERAND the text of the destination selector rather than its value. This is necessary as the text must be inserted into the instruction LAMBDA when the instruction is evaluated.

4.4.6 The Code Macro Function

This is an NLAMBDA function as the parameters, if any, are text which should be substituted into the appropriate places in the statement list. If there are no parameters, the parameter variables referred to in the code macro reference those variables within the scope of the calling instruction. This is in accordance with the scoping rules of LISP.

The format of the code macro is identical to the format of the SYNC_EXPRN (see section 4.1.3) except for the substitution of NLAMBDA in place of LAMBDA.

4.4.7 The Executor Function

This is a PROG function attached to the global variable EXEC@. The expression contains a PROG expression with the local variable INSTR. This variable holds the text of the next instruction to be executed.

```
(PROG (INSTR)
   <statement sequence>
   (SETQ INSTR (LOAD (REG_SELECTOR)))
   <statement sequence>
   (EXEC INSTR)
   <statement sequence>
)
```

The EXEC function causes INSTR to be parsed and executed. It also causes the asynchronous instruction list to be scanned for valid instructions.

EXEC@ must be called each time an instruction is to be executed. A call to EXEC@ is equivalent to starting an instruction cycle in the hardware of the architecture.

4.5 SADL Operators as Functions

The binary and unary operators of SADL as well as the assignment operator ("<-") are all functions in Build. All of the functions are LAMBDA expressions except for the assignment operator which is a special case.

The assignment operator must be an NLAMBDA expression because it must not evaluate the destination list. It evaluates the right hand side by subjecting the value expression to the EVAL function thus forcing an extra level of evaluation.

Both binary and unary operators accept any number and return a number in symbolic binary format. The length of the binary number returned depends on the operator. Boolean operators return a binary zero or binary one, a single digit. The _length_ operator returns a _decimal_ value. The _ext_ operator returns a binary number of "infinite length"; this is actually some long implementation dependent length like 128 or 256 characters. In Salford LISP the length is in excess of 600 characters.

The remaining operators return a result that is the same length as the operand, or the larger of the two operands. If the operands were decimal or hexadecimal then the binary format contains as many bits as are necessary to represent the number as passed; this means that leading zeros in decimal and hexadecimal numbers are significant to the representation.

Note that the unary operators "+" and "-" invoke the same functions as the binary operators "+" and "-". The functions check to see whether the second parameter is null to decide whether to behave as a unary operation or a binary operation.

Some operators process the symbolic binary numbers in that form while others (the arithmetic operators) first convert the input into decimal before applying the inbuilt LISP operators and then convert the result back to binary. This is a temporary solution to the problem of performance.

The operators which process symbolic binary first split the numbers into lists of digits and then perform list walks in combination with list surgery. Nothing more sophisticated than comparison or cutting and pasting is involved.

Example 4.15 - the right shift operator

```
(DEFUN rsh (OP)
   (SETQ OP (CDR (EXPLODE (CONVERT 'BIN OP))))
   (SETQ OP (CDR (REVERSE (CONVERT 'BIN OP))))
   (SETQ OP (REVERSE OP))
   (SETQ OP (CONS (CAR OP) OP))
   (IMPLODE (CONS '& OP))
)
```

Example 4.16 - the logical AND operator

```
(DEFUN and (OP1 OP2)
   /* ensure the operands are binary
   (SETQ OP1 (CDR (EXPLODE (CONVERT 'BIN OP1))))
   (SETQ OP2 (CDR (EXPLODE (CONVERT 'BIN OP2))))
   /* extend the shorter operand to the length of the larger
   (WHILE (LESSP (LENGTH OP1) (LENGTH OP2) )
      (SETQ OP1 (CONS 0 OP1))
   )
   (WHILE (LESSP (LENGTH OP2) (LENGTH OP1) )
      (SETQ OP2 (CONS 0 OP2))
   )
   (SETQ OP1 (REVERSE OP1))
   (SETQ OP2 (REVERSE OP2))
   /* perform the and operation
   (SETQ OP1 (MAPCAR (LAMBDA (X Y)
                         (COND ((OR (ZEROP X)
                                    (ZEROP Y) )
                               0)
                               (T 1)
                         ))
                      OP1 OP2
              ))
   (IMPLODE (CONS '& (REVERSE OP1)))
)
```

The assignment operator is by far the most sophisticated operator as it must perform several functions. It causes evaluation of its second parameter to extract a value which is assigned to the local variable RSLT. It then parses the destination list accessing locations in which to store the value. Because of the possibility that each of the destination registers is mapped, a function GET_BASE is called. The function accepts a register name or register selector expression and returns a name or expression which consists of the registers which are mapped to by the input register array. As GET_BASE is recursive any level of mapping is supported; this is consistent with the semantics of SADL. After GET_BASE has been applied to every member of the destination list a new destination list exists with only unmapped registers.

Scanning the destination list also requires processing any substitution parameters which may be part of the destination list. Any member of the destination list which starts with a $ must be evaluated to obtain the true destination.

Example 4.17

```
        dest:          $1
        value of $1:   (mem 0)
        true dest:     (mem 0)
```

Once the new, unmapped destination list has been constructed, the value must be placed into the appropriate registers and zero extended where necessary. This is a matter of scanning the destination list in reverse order (as left to right is most significant to least significant order) assigning the digits from RSLT, least significant first.

Where a register array is named without any selector expression, all registers in the array are assigned values.

As each bit, or multiple of bits, is assigned from the RSLT it is dropped from the list. When RSLT is null the remaining registers are assigned zeros.

The complexity of the above description is necessary to support the full semantics of the SADL assignment. The most common case is not a destination list but rather a single destination register. In this case GET_BASE is called once with the register selector and the value may be assigned directly.

4.6 An Example

To tie this description together an example of how an instruction would be processed is given. The example is the INC instruction from the Motorola 6800 microprocessor. This instruction causes the memory location specified by the single instruction parameter to be incremented by one.

The SADL definitions for the registers used are:

```
PC is []<15 0> end          /* 16-bit register
MEM is [0 #FFFF]<7 0> end   /* 8-bits * 64K words
```

The SADL instruction definition for INC is:

```
INC is $ <- $ + 1
    from $                          /* entire operand field
    using INDEX;                    /* Index and
          EXTND                     /* Extended addressing
    size 8 end                      /* instruction length
```

The SADL description for the access method used is:

```
EXTND is
         OPERAND <- MEM [$]
      from $                          /* entire operand field
      size 16 end                     /* 8+ 16 = 24 bit instr.
```

When the SADL description has been processed by BUILD the following properties of the various names are defined. The following is the output of DUMP, a procedure which prints out the names of the properties and their values in a (reasonably) pleasing format:

```
pc


MSB #f LSB #0 LSW #0 MSW #0
MAPLIST ()
CELLS ()
```

For the example at least two of the registers of the register array MEM must be occupied. One must contain an instruction while the other contains the value that is being incremented.

```
mem

MSB #7 LSB #0 LSW #0 MSW #ffff
MAPLIST ()
CELLS ( (#f000 &10001001)
 (#0 (inc #f000) )
 (#1 (inc #f000) )
 (#2 (inc #f000) ) )
```

The first member of the CELLS list is the register which is to be modified while the second,third and fourth members are the registers which hold the instruction. Three registers are required because of the access method used by the instruction.

Only the EXTND access method is shown:

```
extnd


AM_SIZE 16

AM_MATCH ($)

AM_EXPRN

   (LAMBDA ($)

      (PROG (OPERAND)

         (SETQQ OPERAND (mem $))

         (RETURN OPERAND)

      )

   )
```

And the INC instruction:

```
inc


I_SIZE 8

I_MATCH ( $)

AM_LIST (( extnd) ( index))

I_EXPRN

   (LAMBDA ($)

      ( <- ($) (+ (VALUE_OF $) 1))

   )
```

The loading of instructions into the register space is the responsibility of the interface which sits on top of the LISP architecture generated by Build. It is assumed that the interface procedure which loads the instructions into the register array also converts constants to hexadecimal representation.

The invocation (EXEC@) causes the instruction to be loaded from the register array element specified. The SADL declaration is:

```
executor
    load MEM [PC];
    PC <- PC + 1;
    exec
    end
```

which is represented in LISP as:

```
EXEC@

(PROG (INSTR)
   (SETQ INSTR (LOAD (mem (VALUE_OF pc))))
   ( <- PC (+ (VALUE_OF pc) 1))
   (EXEC INSTR)
)
```

The EXEC function locates the instruction in the SYNC_LIST@ and locates the appropriate access method. In this case the access method is EXTND. The operands of the instruction are passed to the EXTND property function which returns the value of OPERAND. OPERAND for this particular instance would be:

```
        (mem #f000)
```

This is then passed into the INC instruction which then evaluates as if it were:

```
        (LAMBDA ()
           ( <- ( (mem #f000) ) (+ (VALUE_OF (mem #f000) ) 1))
        )
```

The VALUE_OF function parses the register selector, extracts the value from the register data structure and returns the value &10001001. If mem had mapped to several smaller registers, then the values returned from those registers would be concatenated into a single number.

The "+" function is evaluated with the parameters &10001001 and 1. It returns the value &10001010.

The assignment function parses the destination list for the single target register expression. The function GET_BASE returns the same expression (mem #f000) indicating no mapping.

The first member of the expression is extracted and used to locate the register data structure in REG_LIST@. The properties of the register are then used in the following manner.

A search of the members of the CELLS list is used to attempt to locate the correct cell using the value returned by the selector expression (in this case #f000). If that fails then the number of elements in the list is compared with the number specified using the MSW and LSW properties; if they are the same then it is an enumerated list and the INDEX function may be used to extract the value, otherwise the element has not yet been written to.

If the element is not yet written to, it is CONSed to the beginning of the list, otherwise direct surgery is performed using RPLACA to replace the old version of the cell with the new version of the cell.

In the example the cell #f000 is located and RPLACA is performed on:

```
( (#f000 &10001001)
  (#0 (inc #f000) )
  (#1 (inc #f000) )
  (#2 (inc #f000) ) )
```

with (#f000 &10001010) being the substitute list.

The final result is that CELLS looks like:

```
( (#f000 &10001010)
  (#0 (inc #f000) )
  (#1 (inc #f000) )
  (#2 (inc #f000) ) )
```

This example has shown a simple instruction which has a value expression involving a SADL operator and an assignment. No other examples have been given as no new concepts are necessary to build up more complex instructions.

5 Conclusions

5.1 Summary

In the four preceding chapters I have provided a representative sample of thinking in instruction set processor description languages, have developed my own model of the environment of executing instruction sequences and have produced an architecture description language and an application using that language.

Chapter One explored several types of architecture description language, and cited examples from each area. The advantages and disadvantages of each of the approaches were examined. The most influential of the languages examined has been ISPS and this language was examined in rather greater detail because of this.

Chapter Two developed a model of architecture at the level which is visible to an executing sequence of instructions. The model was influenced by the approaches described in Chapter One but was not based specifically on any of them. It was used as the basis for the language described in Chapter Three.

Chapter Three described the syntax and semantics of SADL, the Symbolic Architecture Description Language. The syntax description given in the chapter is incomplete but is sufficient to allow the semantics of the language to be fully specified. A full syntax using extended BNF notation is included as Appendix 1.

Chapter four described the LISP program Build, an application using SADL. The description of Build showed how a SADL description may be processed to produce data structures and procedures which may form the basis of a simulator, thus allowing architecture-independent simulation.

5.2 The Realization of Design Goals

This thesis had two design goals. The major goal was to design a new language capable of describing instruction set processors in a symbolic form. The language should avoid the details of implementation, but should be able to express the functionality of the architecture fully. SADL accomplishes that goal with some success.

SADL is able to describe a range of architectures without exploring the implementation details and has been successfully used to describe the Intel 8085 and 8086 microprocessors, as well as the National

Semiconductor SC/MP and Motorola 6800 architectures. The language does have limitations though, and these have been described in Chapter Three. Possible improvements to the language outlined in the chapter, included the addition of support for variable length operands and a more sophisticated technique for describing the operand field of assembly language instructions.

The secondary goal was to produce a tool which could serve as the basis for an architecture independent simulator for use in interactive study of architectures from a software engineer's viewpoint. Chapter Four describes Build and provides an example showing how an assembly language instruction is converted into data structures and LISP functions which may then be evaluated to simulate the operation of the instruction. This example indicates the feasibility of Build as the basis of a description driven simulator. Therefore I feel that Build goes some way to satisfying the secondary goal of this thesis.

5.3 Future Directions

The approach to SADL was based very much on the principle that an architecture consists of several independent domains. While this view has been supported by SADL it has resulted in a large language. An interesting possibility is to migrate SADL more towards the approach taken in ISPS (while still retaining the symbolic nature of SADL).

ISPS has a different approach from that of SADL. It recognises a dichotomy between "carriers" (registers) and procedures. Procedures describe all behavioural aspects of the architecture without distinction between access method procedures or instruction procedures. This means that the same syntax and semantics are shared between instructions and access methods, as well as enabling procedures to invoke other procedures (like the SADL code macro statements). This makes the language quite compact.

The disadvantage of this approach is that the distinction between access methods and instructions that exists at the symbolic level is largely lost. In a pedagogic situation this could be a major drawback and it is certainly not the ideal situation for a software engineer who is used to thinking of instructions and their access methods as independent entities.

There are two areas of potential for the development of applications using SADL. One is the application started with the development of Build, that of a symbolic simulator for software engineers to use. This would be a useful tool for two reasons. First, its approach to the architecture is at the level that a software engineer has experience with and so can relate to without extensive training. Second, it is useful as a pedagogic tool for a similar reason.

The other area in which applications could be developed depends upon the fact that the architecture is built up into LISP functions. Because of this it is possible for development engineers to edit an architecture and then immediately simulate the modified architecture to evaluate it. This is essentially the justification that Cragon [Cragon83] put forward for the use of LISP as an architecture specification language.

The other application, and the one that sparked the idea for this thesis originally, is that of automatic translation of instruction sequences from one architecture to functionally equivalent instruction sequences on another architecture. This is an application which keeps recurring as people need to move software from ageing architectures to new systems.

The important phrase is "functionally equivalent". This means that the code sequences may be quite dissimilar so long as their operations, and the registers which hold the values, are consistent within the SADL specification.

During this thesis I have come to realise the size of this task, but I believe that SADL is a reasonable contribution to its solution.

```
<sadl>      ::=  <pdescr> [ <executor> ] .

<pdescr>   ::=  architecture <ar name> is <rset domain>
                                          <amset domain>
                                          <iset domain>

<ar name>  ::=  <identifier>

<rset domain> ::= registers : <reg defn> { <reg defn> }

<reg defn> ::= <r name> is <dim exprn> [ <mapping exprn> ] end

<r name> ::= <identifier>

<dim exprn> ::= <array spec> <word spec>

<array spec> ::= [ [<range bounds> | <cell list>] ]

<range bounds> ::= <lower bound> <upper bound>

<cell list> ::= <cell name> { , <cell name> }

<cell name> ::= <identifier>

<lower bound> ::= <number>

<upper bound> ::= <number>
```

```
<word spec> ::= < [<msb> <lsb>] >

<msb> ::= <number>

<lsb> ::= <number>

<mapping exprn> ::= maps <r mapdef> { || <r mapdef> }

<r mapdef> ::= <r name> [ <m array spec> ]

<m array spec> ::= [ <init addr> <term addr> ]

<init addr> ::= <number> | <cell name>

<term addr> ::= <number> | <cell name>

<amset domain> ::= access methods : <am descr>
                                    { <am descr> }
                                    [ <am class> ]

<am descr> ::= <am name> is <am exprn seq>
                         from <template>
                         [ size <bitsize> ] end

<am name> ::= <identifier>
```

```
<am exprn seq> ::= { <am assign stmt> ; }
                   <am param stmt>
                   { ; <am assign stmt> }


<am param stmt> ::= OPERAND <- <dest selector>


<am assign stmt> ::= <reg selector> <- <value exprn>


<dest selector> ::= <dest exprn> { || <dest exprn> }


<dest exprn> ::= <reg selector>   |   <param substn>


<template> ::= <const item> | <param substn>
                                { <const item> | <param substn> }


<bitsize> ::= <number>


<am class> ::= access classes : <amc descr> { ; <amc descr> }


<amc descr> ::= <amc name> is <am name> { <am name> }


<amc name> ::= <identifier>


<iset domain> ::= [<asynch domain>] <synch domain>
```

```
<asynch domain> ::= asynchronous instructions : <asynch instr>
                                                  { <asynch instr> }


<asynch instr> ::= <i name> is <istmt seq> upon <value exprn> end


<i name> ::= <identifier>


<istmt seq> ::= <istmt> { ; <istmt> }


<istmt> ::= <assign stmt> | <cm stmt> | <cond stmt> | <loop stmt>


<assign stmt> ::= <dest selector> <- <value exprn>


<cm stmt> ::= do <cm name> [ ( <param exprn> { , <param exprn> } )]


<param exprn> ::= <number> | <reg selector> | <param substn>


<cond stmt> ::= if <value exprn> then <istmt seq>
                                  [ else <istmt seq> ] endif


<loop stmt> ::= while <value exprn> do <istmt seq> done


<synch domain> ::= synchronous instructions : [ <codem list> ]
                                                <synch instr>
                                                { <synch instr> }


<codem list> ::= <code macro> { <code macro> }
```

```
<code macro> ::= codem <cm name> [ <cm param> ] is <istmt seq> endm


<cm name> ::= <identifier>


<cm param> ::= ( <param substn> { <param substn> } )


<synch instr> ::= <i name> is <istmt seq>
                                [ from <template> using <amlist> ]
                                [ size <bitsize> ]  end


<amlist> ::= <am tuple> { ; <am tuple> }


<am tuple> ::= <am name>   |  <amc name>  { <am name>  |  <amc name> }


<executor> ::= executor
                [<istmt seq>]
                load <reg selector>
                [<istmt seq>]
                exec
                [<istmt seq>]
                end


<value exprn> ::= [<unop>] <value group>
                  { <binop> [<unop>] <value group> }
```

```
<value group> ::= <reg selector> |
                  <param substn> |
                  ( <value exprn> ) |
                  <number>


<reg selector> ::= <r name> [ [ <value exprn> ] ]


<param substn> ::= $ [ <dec num> ]


<number> ::= <dec num>  |  <bin num>  |  <hex num>


<identifier> ::= <letter> { <letter> | <digit> | . | $ | _ }


<bin num> ::=  & ( 0 | 1 ) { 0 | 1 }


<dec num> ::= <digit> { <digit> }


<hex num> ::= #  <hdigit> { <hdigit> }


<boolop> ::=  =  |  >  |  <  |  >=  |  <=  |  <>


<unop> ::= + | - | not | lsh | rsh | ext | sizeof


<binop> ::= + | - | * | / | ** |
            and | or | || | mod | <boolop>


<const item> ::=  <special char>  | <identifier>  |  <number>
```

```
<special char> ::= ! ¦ " ¦ # ¦ $$ ¦ % ¦ & ¦ ' ¦ ( ¦ ) ¦ = ¦ ~ ¦ ^ ¦
                   \ ¦ ` ¦ @ ¦ { ¦ [ ¦ ; ¦ : ¦ } ¦ ] ¦ < ¦ > ¦ . ¦
                   ? ¦ _   ¦ , ¦ : ¦ ; ¦ + ¦ - ¦ / ¦ * ¦ |

<digit> ::= 0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9

<hdigit> ::=  <digit>  ¦  A  ¦  B  ¦  C  ¦  D  ¦  E  ¦  F
```

```
architecture I8085 is


/* This description is not authoritative but is
/* for illustrative purposes only.
/* It is taken from [Danhof81].


registers :


    CCR is [CY,X1,P,X2,AC,X3,Z,S]<> end
    A is []<7 0> end
    PSW is []<15 0> maps CCR || A end

    B is []<7 0> end
    C is []<7 0> end
    BC is []<15 0> maps C || B end

    D is []<7 0> end
    E is []<7 0> end
    DE is []<15 0> maps E || D end

    H is []<7 0> end
    L is []<7 0> end
    HL is []<15 0> maps L || H end

    SP is []<15 0> end
    PC is []<15 0> end
```

```
/* external registers

   MEM is [0 #FFFF]<7 0> end
   IO is [0 #FF]<7 0> end

/* virtual registers

   R is [A,B,C,D,E,H,L]<7 0> maps A || B || C || D || E || H || L end
   RP is [B,D,H,SP]<15 0> maps C || B || E || D || L || H || SP end
   INX is [B,D]<15 0> maps BC || DE end
   RPP is [B,D,H,SP,PSW]<15 0> maps RP || PSW end
   IMREG is [M5.5,M6.5,M7.5,MSE,R7.5,X,SOE,SOD]<> end

 /* Asynchronous Instruction Activation registers

   INTR is []<> end
   TRAP is []<> end
   RST7.5 is []<> end
   RST6.5 is []<> end
   RST5.5 is []<> end
```

```
    IEREG is []<> end

    RSTREG is []<> end

    HALTREG is []<> end

    TRAPREG is []<> end

    RST7REG is []<> end

    RST6REG is []<> end

    RST5REG is []<> end


   /* Temporary storage registers


    TMP6 is []<15 0> end

    TMP8 is []<7 0> end

    BITS is [0 7]<> end



access methods :


    REG8 is

            OPERAND <- R [$]

        from $

        size 0 end

    REG16 is

            OPERAND <- RP [$]

        from $

        size 0 end
```

```
REG16P is
        OPERAND <- RPP [$]
    from $
    size 0 end
INDIRECT is
        OPERAND <- MEM [HL]
    from M
    size 0 end
INX is
        OPERAND <- MEM [INX [$] ]
    from $
    size 0 end
IMMED3 is
        OPERAND <- $1 and #7
    from $
    size 0 end
IMMED8 is
        OPERAND <- $ and #FF;
        PC <- PC + 1
    from $
    size 8 end
IMMED16 is
        OPERAND <- $ and #FFFF;
        PC <- PC + 2
    from $
    size 16 end
    DIR8 is
            OPERAND <- MEM [$];
            PC <- PC + 2
        from $
        size 16 end
    DIR16 is
            OPERAND <- MEM [$+1] || MEM [$];
            PC <- PC + 2
        from $
        size 16 end

access classes :

    DIR is DIR16 DIR8 end
    IMMED is IMMED8 IMMED16 end
    INDEX is INX INDIRECT end

asynchronous instructions :

    RESET is
            PC <- 0;
            IEREG <- 0;        /* enable interrupts
            RSTREG <- 0
        upon RSTREG end
```

```
TRAP is

       IEREG <- 1;

       MEM [SP-1] || MEM [SP-2] <- PC;

       SP <- SP - 2;

       PC <- #24;

       TRAPREG <- 0

   upon TRAPREG end

RST7.5 is

       IEREG <- 1;

       MEM [SP-1] || MEM [SP-2] <- PC;

       SP <- SP - 2;

       PC <- #3C;

       RST7REG <- 0

   upon RST7REG and not ( IMREG [M7.5] or IEREG) end

RST6.5 is

       IEREG <- 1;

       MEM [SP-1] || MEM [SP-2] <- PC;

       SP <- SP - 2;

       PC <- #34;

       RST6REG <- 0

   upon RST6REG and not ( IMREG [M6.5] or IEREG) end
```

```
    RST5.5 is
           IEREG <- 1;
           MEM [SP-1] || MEM [SP-2] <- PC;
           SP <- SP - 2;
           PC <- #2C;
           RST5REG <- 0
       upon RST5REG and not ( IMREG [M5.5] or IEREG) end


synchronous instructions :


   codem Z ($) is
           CCR [CY] <- $ = 0
       end
   codem S ($) is
           CCR [S] <- ($ and #80) = #80
       end
   codem P ($) is
           BITS <- $;
           CCR [P] <- 1 + BITS [0] + BITS [1] + BITS [2] + BITS [3]
                        + BITS [4] + BITS [5] + BITS [6] + BITS [7]
       end
   codem ACplus ($1 $2) is
           CCR [AC] <- ($1 and #F) + ($2 and #F) > #F
       end
```

```
codem ACminus ($1 $2) is

        CCR [AC] <- ($1 and #F) < ($2 and #F)

    end

codem CY ($1 $2) is

        CCR [CY] <- $1 < $2

    end


MOV is

        $1 <- $2

    from $1,$2

    using REG8 REG8;

          REG8 INDIRECT;

          INDIRECT REG8

    size 8 end

XCHG is

        TMP6 <- HL;

        HL <- DE;

        DE <- TMP6

    size 8 end

MVI is

        $1 <- $2

    from $1,$2

    using REG8 IMMED8;

          INDIRECT IMMED8

    size 8 end
```

```
LXI is
        $1 <- $2
    from $1,$2
    using REG16 IMMED16
    size 8 end
LDA is
        A <- $1
    from $1
    using DIR8
    size 8 end
LHLD is
        HL <- $1
    from $1
    using DIR16
    size 8 end
LDAX is
        A <- $1
    from $1
    using INX
    size 8 end
STA is
        $1 <- A
    from $1
    using DIR8
    size 8 end
```

```
SHLD is

        $1 <- HL

    from $1

    using DIR16

    size 8 end

STAX is

        $1 <- A

    from $1

    using INX

    size 8 end

ADD is

        do ACPLUS (A $1);

        CCR [CY] || A <- A + $1;

        do P (A);

        do Z (A);

        do S (A)

    from $1

    using REG8;

          INDIRECT

    size 8 end
```

```
ADI is
        do ACPLUS (A $1);
        CCR [CY] || A <- A + $1;
        do P (A);
        do Z (A);
        do S (A)
    from $1
    using IMMED8
    size 8 end
ADC is
        do ACPLUS (A $1);
        CCR [CY] || A <- A + $1 + CCR [CY];
        do P (A);
        do Z (A);
        do S (A)
    from $1
    using REG8;
          INDIRECT
    size 8 end
```

```
ACI is

        do ACPLUS (A $1);

        CCR [CY] || A <- A + $1 + CCR [CY];

        do P (A);

        do Z (A);

        do S (A)

    from $1

    using IMMED8

    size 8 end

SUB is

        do ACMINUS (A $1);

        CCR [CY] || A <- A - $1;

        do P (A);

        do Z (A);

        do S (A)

    from $1

    using REG8;

          INDIRECT

    size 8 end
```

```
SUI is
        do ACMINUS (A $1);
        CCR [CY] || A <- A - $1;
        do P (A);
        do Z (A);
        do S (A)
    from $1
    using IMMED8
    size 8 end
SBB is
        do ACMINUS (A $1);
        CCR [CY] || A <- A - $1 - CCR [CY];
        do P (A);
        do Z (A);
        do S (A)
    from $1
    using REG8;
          INDIRECT
    size 8 end
    SBI is
            do ACMINUS (A $1);
            CCR [CY] || A <- A - $1 - CCR [CY];
            do P (A);
            do Z (A);
            do S (A)
        from $1
        using IMMED8
        size 8 end
    INR is
            do ACPLUS ($1 1);
            $1 <- $1 + 1;
            do P ($1);
            do Z ($1);
            do S ($1)
        from $1
        using REG8;
              INDIRECT
        size 8 end
    INX is
            $1 <- $1 + 1
        from $1
        using REG16
        size 8 end
    DCR is

            do ACMINUS ($1 1);

            $1 <- $1 - 1;

            do P ($1);

            do Z ($1);

            do S ($1)

        from $1

        using REG8;

              INDIRECT

        size 8 end

    DCX is

            $1 <- $1 - 1

        from $1

        using REG16

        size 8 end

    DAD is

            CCR [CY] || HL <- HL + $1

        from $1

        using REG16

        size 8 end
```

```
DAA is
        if (A and #OF) > 9 then
           CCR [CY] || A <- A + 6;
           CCR [AC] <- 1 endif;
        if (A and #FO) > #90
        then CCR [CY] || A <- A + #60 endif;
        do P (A);
        do Z (A);
        do S (A)
    size 8 end
ANA is
        A <- A and $1;
        do P (A);
        do Z (A);
        do S (A);
        CCR [CY] <- 0;
        CCR [AC] <- 1
    from $1
    using REG8;
          INDIRECT
    size 8 end
```

```
ANI is
        A <- A and $1;
        do P (A);
        do Z (A);
        do S (A);
        CCR [CY] <- 0;
        CCR [AC] <- 1
    from $1
    using IMMED8
    size 8 end
XRA is
        A <- (A and not $1) or (not A and $1);
        do P (A);
        do Z (A);
        do S (A);
        CCR [CY] <- 0;
        CCR [AC] <- 0
    from $1
    using REG8;
          INDIRECT
    size 8 end
```

```
XRI is

        A <- (A and not $1) or (not A and $1);

        do P (A);

        do Z (A);

        do S (A);

        CCR [CY] <- 0;

        CCR [AC] <- 0

    from $1

    using IMMED8

    size 8 end

ORA is

        A <- A or $1;

        do P (A);

        do Z (A);

        do S (A);

        CCR [CY] <- 0;

        CCR [AC] <- 0

    from $1

    using REG8;

          INDIRECT

    size 8 end
```

```
ORI is
        A <- A or $1;
        do P (A);
        do Z (A);
        do S (A);
        CCR [CY] <- 0;
        CCR [AC] <- 0
    from $1
    using IMMED8
    size 8 end
CMP is
        TMP8 <- A - $1;
        do CY (A $1);
        do P (TMP8);
        do ACMINUS (A $1);
        do Z (TMP8);
        do S (TMP8)
    from $1
    using REG8;
          INDIRECT
    size 8 end
    CPI is

            TMP8 <- A - $1;

            do CY (A $1);

            do P (TMP8);

            do ACMINUS (A $1);

            do Z (TMP8);

            do S (TMP8)

        from $1

        using IMMED8

        size 8 end

    RLC is

            CCR [CY] || A <- lsh A;

            if CCR [CY]

            then A <- A or 1 endif

        size 8 end

    RRC is

            A || CCR [CY] <- A;

            if CCR [CY]

            then A <- A or #80 endif

        size 8 end

    RAL is

            CCR [CY] || A <- A || CCR [CY]

        size 8 end

    RAR is

            A || CCR [CY] <- CCR [CY] || A

        size 8 end
```

```
CMA is
        A <- not A
    size 8 end
CMC is
        CCR [CY] <- not CCR [CY]
    size 8 end
STC is
        CCR [CY] <- 1
    size 8 end
JMP is
        PC <- $1
    from $1
    using IMMED16
    size 8 end
JNZ is
        if CCR [Z] = 0
        then PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
JZ is
        if CCR [Z] = 1
        then PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
```

```
JNC is
        if CCR [CY] = 0
        then PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
JC is
        if CCR [CY] = 1
        then PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
JPO is
        if CCR [P] = 0
        then PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
JPE is
        if CCR [P] = 1
        then PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
```

```
JP is
        if CCR [S] = 0
        then PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
JM is
        if CCR [S]
        then PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
CALL is
        MEM [SP-1] || MEM [SP-2] <- PC;
        SP <- SP - 2;
        PC <- $1
    from $1
    using IMMED16
    size 8 end
```

```
CNZ is
    if CCR [Z] = 0 then
            MEM [SP-1] || MEM [SP-2] <- PC;
            SP <- SP - 2;
            PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
CZ is
    if CCR [Z] = 1 then
            MEM [SP-1] || MEM [SP-2] <- PC;
            SP <- SP - 2;
            PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
CNC is
    if CCR [CY] = 0 then
            MEM [SP-1] || MEM [SP-2] <- PC;
            SP <- SP - 2;
            PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
```

```
CC is
    if CCR [CY] = 1 then
            MEM [SP-1] || MEM [SP-2] <- PC;
            SP <- SP - 2;
            PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
CPO is
    if CCR [P] = 0 then
            MEM[ SP-1] || MEM [SP-2] <- PC;
            SP <- SP - 2;
            PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
CPE is
    if CCR [P] = 1 then
            MEM [SP-1] || MEM [SP-2] <- PC;
            SP <- SP - 2;
            PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
```

```
CP is
    if CCR [S] = 0 then
            MEM [SP-1] || MEM [SP-2] <- PC;
            SP <- SP - 2;
            PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
CM is
    if CCR [S] = 1 then
            MEM [SP-1] || MEM [SP-2] <- PC;
            SP <- SP - 2;
            PC <- $1 endif
    from $1
    using IMMED16
    size 8 end
RET is
        PC <- MEM [SP+1] || MEM [SP];
        SP <- SP + 2
    size 8 end
RNZ is
    if CCR [Z] = 0 then
            PC <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2 endif
    size 8 end
```

```
RZ is
    if CCR [Z] = 1 then
            PC <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2 endif
    size 8 end
RNC is
    if CCR [CY] = 0 then
            PC <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2 endif
    size 8 end
RC is
    if CCR [CY] = 1 then
            PC <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2 endif
    size 8 end
RPO is
    if CCR [P] = 0 then
            PC <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2 endif
    size 8 end
RPE is
    if CCR [P] = 1 then
            PC <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2 endif
    size 8 end
```

```
RP is
    if CCR [S] = 0 then
            PC <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2 endif
    size 8 end
RM is
    if CCR [S] = 1 then
            PC <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2 endif
    size 8 end
RST is
        MEM [SP-1] || MEM [SP-2] <- PC;
        SP <- SP - 2;
        PC <- $1 * 8
    from $1
    using IMMED3
    size 8 end
PCHL is
        PC <- HL
    size 8 end
PUSH is
        MEM [SP-1] || MEM [SP-2] <- $1;
        SP <- SP - 2
    from $1
    using REG16P
    size 8 end
    POP is
            $1 <- MEM [SP+1] || MEM [SP];
            SP <- SP + 2
        from $1
        using REG16P
        size 8 end
    XTHL is
            TMP6 <- HL;
            HL <- MEM [SP+1] || MEM [SP];
            MEM [SP+1] || MEM [SP] <- TMP6
        size 8 end
    SPHL is
            SP <- HL
        size 8 end
    IN is
            A <- $1
        from $1
        using IO
        size 8 end
    OUT is
            $1 <- A
        from $1
        using IO
        size 8 end
    EI is

            IEREG <- 0

        size 8 end

    DI is

            IEREG <- 1

        size 8 end

    HLT is

            HALTREG <- 1

        size 8 end

    NOP is

            PC <- PC

        size 8 end

    RIM is

            A <- IMREG

        size 8 end

    SIM is

            if IMREG[MSE]

            then IMREG <- A

            else IMREG <- (IMREG and #07) or (A and #F8) endif

        size 8 end


executor

    load MEM [PC]

    PC <- PC + 1

    exec

    end .
```

[Barb81] Instruction Set Processor Specifications (ISPS): The Notation and Its Applications
- Mario R. Barbacci
IEEE Trans. Comp., Vol. C-30, No. 1, Jan 1981

[Bell71] Computer Structures: Readings and Examples
- C. G. Bell
- A. Newell
McGraw-Hill Publ. 1971

[Cattell78] Formalization and Automatic Derivation of Code Generators
- R. G. G. Cattell
Carnegie-Mellon University (April 1978) CMU-CS-78-115
(Ph.D thesis)

[Cattell80] Automatic Derivation of Code Generators from Machine Descriptions
- R. G. G. Cattell
ACM Trans. Prog. Lang. Syst. Vol. 2, No. 2, April 1980

[Cragon83] Executable Instruction Set Specification
- Harvey Cragon
Computer Architecture News, Vol. 11, No. 1, March 1983

[Danhof81] Computer System Fundamentals
- Kenneth J. Danhof
- Carol L. Smith

Addison-Wesley Publ. 1981

[Dasgupta82] Computer Design and Description Languages
- Subrata Dasgupta

Advances in Computers, Vol. 21, 1982

[Distler82] Trial implementation reveals errors in IEEE standard
- R. J. Distler
- M. A. Shaver

Computer, July 1982, pp 76-77

[Fischer79] Microprocessor Assembly Language Draft Standard (IEEE Task P694/D11)
- Wayne P. Fischer

Computer, Dec. 1979, pp 96-109

[Intel81] iAPX 88 Book

Intel Corp. 1981

[Intel84] Microsystem Components Handbook, Vol. 1

Intel Corp. 1984

[Lee73] VDL - A definition System For All Levels

- John A. N. Lee

Proc First Ann. Symp. on Comp. Arch.

Univ. Florida, Gainsville 1973

[Motorola81] Motorola Microprocessors Data Manual

Motorola Inc. 1981

[Mueller76] A Generator for Microprocessor Assemblers and Simulators

- Robert A. Mueller
- Gearold R. Johnson

Proc. IEEE, Vol. 64, No. 6, June 1976

[Osborne81] Osborne 16-bit Microprocessor Handbook

- Adam Osborne
- Gerry Kane

Osborne/McGraw Hill Publ. 1981

[Patterson82] A VLSI RISC

- David A. Patterson
- Carlo H. Sequin

Computer, September 1982, pp 8-21

[Purdum83] C Programming Guide
- Jack Purdum
Que Publ. 1983

[Salford83] The University of Salford Lisp/Prolog Reference Manual
- David Baily
Univ. Salford 1983

[SC/MP76] SC/MP Technical Description
National Semiconductor Corp. 1976

[Siewiorek82] Computer Structures: Principles and Examples
- Daniel P. Siewiorek
- C. Gordon Bell
- Allen Newell
McGraw-Hill Publ. 1982

[Spitzen76] The Specification of Assemblers
- Jay M Spitzen
IEEE Trans. Soft. Eng. Vol. SE-2, No. 1, March 1976

[Tanenbaum76] Structured Computer Organisation
- Andrew S. Tanenbaum
Prentice Hall Publ. 1976

[Wakerly80] Pascal Extensions For Describing Computer Instruction Sets
- John F. Wakerly
Computer Architecture News, Vol. 8, No. 7, Dec 1980

[Wegner72] The Vienna Definition Language
- Peter Wegner
Computing Surveys, Vol. 4, No. 1, March 1972

[Winston81] LISP
- Patrick Henry Winston
- Berthold Klaus Paul Horn
Addison-Wesley Publ. 1981