Architecture Validation System

Using Assertion Descriptions

Users Manual

9 May 1983

Charles P. Kollar
Computer Science Department
Carnegie-Mellon University

This research was sponsored in part by the Department of the Army under contract number DAA B07-82-C-J164. This document should not be interpreted as representing the official or unofficial policies, either expressed or implied, in part or in whole, of the Department of the Army or the United States Government.

Table of Contents

Preface

An overview and background to the system is presented in the Introduction. The latter portion of this document is a **users manual** to be used to construct computer architecture validation programs.

Please address all comments concerning this document to the author at ARPANET address KOLLAR@CMU-CS-C.

1. Introduction

1.1 The Problem of Computer Architecture Validation.

The problem of computer architecture[1] validation is simply the problem of proving the equivalence of incarnations of the same computer architecture. Since computer architectures are essentially algorithms, the problem becomes one of proving the equivalence of algorithms. This latter problem is dealt with under the heading of 'Program Verification'. Program verification attempts to determine whether the program description is implemented via the formal programming language.

There are several approaches to program verification that should be understood before attempting computer architecture validation. In addition there are constraints that the physical computer architecture imposes that are not found in program verification. The following will briefly discuss the current schools of thought of program verification (Section 1.2). After this a brief overview of the current schools of thought of computer architecture validation will be discussed (Section 1.3) noting the similarities between the two.

1.2 Program Verification.

Program Verification techniques are divided along the lines of how much *formal understanding* of the problem is used. There are three current divisions that are described, in order of decreasing formal understanding, as follows:

1.2.1 Formal Analysis.

Here the program is described as a mathematical model and analyzed axiomatically. This *formalization* allows general assertions about value/set interactions to be demonstrated. Hoare [3] describes a program as possessing a *precondition*, a *function* and an *assertion*. The *precondition* describes the state that is necessary to invoke the function. The *function* describes the action that is necessary to produce the assertion. The *assertion* is the action of the function. Hoare points out that this approach does not detect failure to terminate due to an infinite loop. It should be interpreted as: "provided that the program successfully terminates, the properties of its results are described by the assertion." Hoare's axioms and rules are not implementation dependent. Rather, they focus on the

---

[1] Here, the computer architecture is defined as "that time independent aspect of a computer implementation that the programmer sees."

"conditional" correctness of a program, "relying on an implementation to give a warning if it has had to abandon execution of the program as a result of violation of any implementation limit." He makes mention of including these implementation dependencies but warns of their "reflection in the complexity of the underlying axioms."

The method is general and it is not surprising that the results are likewise general.

1.2.2 Symbolic Execution.

The program is executed symbolically in an effort to understand the data paths through the program. A data path is described by a set of symbolic inputs that derive a set of symbolic outputs. In *symbolic program execution*, the set of all inputs is divided into classes or subsets of inputs. The classes are selected by analyzing the program control flow. The control flow is determined by representing the program in the form of a flow diagram or network structure. One class corresponds to one path along the flow diagram or network. The symbolic program execution method is not a formal correctness proof as is the formal analysis method, since the program is not analyzed axiomatically. The halting problem is not satisfied since, as stated by King [5]: "Often the set of input classes is determined only by those inputs in the control flow".

The term "symbolic" arises from the fact that the real data for program execution is not used. Each symbol corresponds to a set of real data. The input is symbolic and therefore the output is symbolic.

1.2.3 Test Data Selection.

The least general, and least formal, method is that of manual test data selection. Here input is selected to cover certain "intuitively chosen" problem cases. The method does not satisfy the general Halting problem in that typically only a subset of the input criteria is considered. Therefore it is not valid in demonstrating the existence of errors for all input that produces output as does the formal analysis method. This problem was recognized by Dijkstra [2]. "Program testing can be used to show the presence of bugs but never to show their absence."

1.2.3.1 Computer Architecture Test Data Selection.

Several problem areas relating to computer architecture validation have been presented by Lai [6]. His experiments attempt to determine where ambiguities are *most likely* to occur in the specification of a computer architecture. He discusses several likely sources of errors. These are:

1. *Incomplete and imprecise specification:* Where the "Hardware for any implementation does something for the unspecified operations."

2. *Interdependent side-effects:* "An instruction consisting of multiple operations is inherently ambiguous if the order of the operations is not clearly specified and the effect of the instruction depends on this order."

3. *Boundary values:* "Input values that are at the boundaries of different decision regions in the input domain of the instruction."

4. *Missing features:* "Relatively simple features are left out of an implementation due to the oversight or lack of experience of its designers."

1.2.4 Summary

The first approach is considered useful for only the simplest of programs. Therefore, this approach does not lend itself to the validation of computer architectures. Some work has been conducted on the second approach (in relation to computer architectures). A simulator for a computer architecture written in ISPS [4] for the PDP-11 [1] has been analyzed in this manner by Oakley [9]. This involves the creation of an input/output assertion description from the procedural ISPS description. The procedural description has the advantage of being more easily analyzed by humans, while the assertion description gives a clearer understanding of the paths through the computer architecture. If the test data by intuition method is to become more complete, less intuition and more formalization will be required. However if this method is used the problem areas, as discussed by Lai, should be exercised thoroughly.

1.3 Architecture Validation

Due to the physical nature of most computer architectures, validation takes place along one of two lines. The first line requires the hardware implementation of the computer to be modified so that certain machine state is made available. A second computer architecture description (perhaps written in a procedural language such as ISP) is created and declared the "standard" to which all other implementations of this architecture are compared. Machine state of the "standard" and the implementation to be validated are compared as each executes the same series of instructions. This approach has the disadvantage of requiring extra machinery to produce the comparisons. The advantage of this approach is that instructions are "validated" at a high rate. This approach can be categorized as one of brute force. There is usually little attempt to single out potential problem areas and concentrate on them; instructions are typically selected at random. The high instruction throughput and the random selection lead to a 'certain' degree of confidence.

The second method is characterized by low instruction/data throughput but a high degree of concentration on what are considered the problem areas. These are the features of the architecture that should be most prone to implementation error. A typical approach assumes orthogonality of the access modes and the instruction functionality. This allows instructions that modify data 'exclusively' to be tested in a table driven format. This assumption decreases the complexity of the validation program which allows it to be more easily declared correct[2] . In contrast to the first approach, the second approach provides low instruction throughput and a high degree of selection to give a 'certain' degree of confidence.

Confidence is not easily quantified.

1.4 The Proposed Approach.

The following approach allows the computer architecture to be tested along different data paths. As such it should be used in conjunction with the symbolic execution of a procedural description of the computer architecture to be validated. The approach requires the construction of one 'validation program' for each of the input/output assertion paths described by the symbolic execution[3] (See Section 1.2.2.). The issue of 'what actual data to select' within the sets described by the input/output assertions is in part answered by providing tools for the creator of the validation programs. The tools allow the input/output assertion sets to be further divided into sets, with a probability selection distribution associated with each set. This allows the architecture to be described at a somewhat lower input/output assertion level than could be provided by the assertion description for the architecture. The necessity for this arises from the knowledge that the procedural or assertion description does indeed describe the physical architecture at a relatively high level.

Assuming the validity of the data paths described by the high level description with respect to the physical implementation, it becomes necessary to describe the sub-paths. The system discussed within allows these sub-paths to be described yet it does not claim to find them. Since these sub-paths are *architecture dependent*, and indeed implementation dependent, they must be found through a more in-depth knowledge of the structure of the physical implementation. Lacking this, it is necessary to make assumptions about the underlying data paths based on intuition. These

---

[2] A problem with this approach is having to 'prove' the validation program correct (the first approach uses no program *per se*).

[3] Indeed, it is possible to proceed without an input/output assertion description generated from a procedural description. This simply requires the input/output description to be generated from scratch.

assumptions could include choosing the more complicated addressing modes to be used more often than the less complicated, or subseting data paths along the lines of the storage sizes (i.e. word, byte, halfword) of the architecture. The work of Lai (See Section 1.2.3.1.) suggests other sources of implementation error.

1.5 Summary

A system has been devised to generate program modules whose data and addressing modes are chosen with given probabilities. The programs themselves are generated automatically from 'templates'. Each 'template' describes one input/output assertion path through the architecture. The potential data set of the instruction is divided into weighted subsets, as are the potential access modes of the instruction to be validated.

In addition, the automation of this process allows certain methods to be employed that would otherwise be too tedious and time consuming to 'hand code'. These methods include the execution of different instruction sequences to add a dimension of instruction interaction to the testing. For example, a result produced by the previous instruction can be used as a source operand for the instruction currently being validated. The use of several methods to both *set and test operands*, either sources or destinations, decreases reliance on the 'functionality' of any one method of testing or path through the architecture. In this manner different paths through the architecture are used in areas of the program that are 'assumed'[4] to be previously validated.

[4] Instruction sequences that are used to "validate" other instructions should themselves be valid.

2. Validation System

2.1 Justification of the Approach

For the purposes of the system of validation used here it is necessary to think of a computer as a finite state machine. The state of the machine can be characterized by the value of all architecture variables such as the program counter, the registers, the memory, as well as architecture variables not available directly to the programmer. The exact definition of a state for our purposes will be qualified below. In addition it is necessary to effect a state transformation via a *state transformation function*. The state transformation function will be defined as requiring two inputs, a state and an event, and producing as output a state. That is:

$$\text{NEXT-STATE}(S_i, \text{EVENT}) \Rightarrow S_j$$

Or simply:

$$\text{EVENT}(S_i) \Rightarrow S_j$$

Which states that it is the event itself which is responsible for the *state transformation* and is thus the state transformation function.

To continue this discussion it will be necessary to define several terms. A *pre-condition* consists of the machine state prior to the occurrences of some event. A *post-condition* consists of the machine state at the termination of some event. An *event* is therefore a means of effecting a state change in the computer. An event would include: the execution of an instruction, the execution of some portion of an instruction, or some action initiated external to the computer such as a I/O request. That is:

$$\text{EVENT}(S_{pre}) \Rightarrow S_{post}$$

This being the case, the task of validating the computer architecture may be described as follows: Given all possible *events* it is necessary to determine that all possible *pre-conditions* of these events produce the "expected" *post-conditions*. To effect this it will be necessary to:

1. Find a means to capture machine states.

2. Find a means to compare machine states.

3. Find a means to initiate an event.

The method chosen here to capture and compare machines states as well as to initiate events[5] has been determined, in part, by a constraint placed on the validation system: no external hardware

---

[5] As well as the exact definition of a *state* and *event* for our purposes.

probes are permitted to examine states on the "sub-instruction" level. Therefore states must be captured and compared at the "instruction" level. This determines what may be considered a "state" for validation purposes. Here the "state" consists of those aspects of a computer architecture which may be examined and compared at the instruction level of the architecture. If the system is permitted to ignore I/O events, the events which may be initiated are those of the instruction level. A scheme must therefore be devised for establishing a state (as we have defined it) as well as testing for the existence of a state after the event of an instruction execution.

Several assumptions have been made which allow this task to become manageable. These are:

1. Instructions which are used to establish the required *pre-condition* are indeed valid (will modify the intended state variables correctly) and will produce no "unexpected" side effects.

2. The instruction subject to validation will modify only those state variables that it is "expected" to modify (though it may not modify these correctly), and in doing so will produce no "unexpected" side effects.

That is an instruction which is "expected" to modify only the value of the state variables, the operand, and the condition codes,[6] would not unexpectedly add 5 to the stack pointer and place the result in virtual memory address 5432 hex.

These assumptions about instruction execution behavior permit the architecture validation system to proceed in the following manner:

1. The required precondition S_{pre} may be established by applying several instruction level state transformations to a known current state $S_{current}$. That is:

$$\begin{aligned} &EVENT(S_{current}) \Rightarrow S_k \\ &EVENT(S_k) \Rightarrow \ldots \\ &EVENT(\ldots) \Rightarrow S_{pre} \end{aligned}$$

   This follows from the first assumption in that the state transformation function is guaranteed to produce the desired result with no "unexpected" side effects.

2. The expected post-condition S_{post} may be tested for by examining only those state variables which the instruction is "expected" to modify. This follows from the second assumption in that the state transformation function under test is guaranteed to effect only these state variables.

---

[6] For example, the NEBULA INC instruction which is expected to modify an operand (by incrementing it's value), and modify the condition codes (based on the resulting value of the operand).

2.2 Overview

The state of the architecture which an instruction under test may modify is sub-divided along certain conceptual divisions or boundaries. Currently these boundaries or sub-states[7] are:

1. The memory and registers (MR), which are modified by the instruction through the operand fetch mechanism, or access mode (AM).

2. The condition codes (CC), which are modified by the instruction directly.

3. The program counter (PC), which is modified by the instruction directly.

4. Exception has been generated (EX), which is modified by the instruction directly.

The instruction manipulates data (effects change in state of the memory and registers (MR)) through the AM's. For example, given a three operand instruction of the form:

```
INSTR  AM1, AM2, AM3
```

Two pieces of data, located in the memory or register (MR) state variables, might be fetched through AM1 and AM2 with the result returned through AM3. In the instance where instructions produce *arithmetic* results it is clear that the CC's are an integral part of the *true result*[8] . The CC's exist in part due to the problems associated with *implementing* the mathematical concept of a number. This is true in particular for the *truncate* and *carry* CC's. However for validation purposes the condition codes CC's are treated as a separate state variable.

The effect of state modification by an instruction execution event which is under test

$$\text{TEST-EVENT}(S_{pre}) \Rightarrow S_{post}$$

can be divided along the aforementioned state divisions. That is:

$$\begin{aligned} \text{TEST-EVENT}(S_{MR\text{-}pre}) &\Rightarrow S_{MR\text{-}post} \\ \text{TEST-EVENT}(S_{CC\text{-}pre}) &\Rightarrow S_{CC\text{-}post} \\ \text{TEST-EVENT}(S_{PC\text{-}pre}) &\Rightarrow S_{PC\text{-}post} \end{aligned}$$

Accordingly the required state precondition $S_{X\text{-}pre}$ maybe realized as follows:

$$\begin{aligned} \text{EVENT}(S_{MR\text{-}current}) &\Rightarrow \ldots \Rightarrow \text{EVENT}(S_{MR\text{-}pre}) \\ \text{EVENT}(S_{CC\text{-}current}) &\Rightarrow \ldots \Rightarrow \text{EVENT}(S_{CC\text{-}pre}) \\ \text{EVENT}(S_{PC\text{-}current}) &\Rightarrow \ldots \Rightarrow \text{EVENT}(S_{PC\text{-}pre}) \end{aligned}$$

---

[7] These particular sub-states were chosen as a result of their being easily accessed or monitored at the NEBULA computer architecture *instruction level*. In somes sense this issue is architecture dependent. However the state of most architectures can be subdivided into, at least, the first three sub-states. These are the MR, CC, and PC sub-states.

[8] *True result* in the mathematical sense of an operation such as "add" or "subtract"

In addition serial events may be initiated to verify the post-conditions: $S_{MR\text{-}post}$, $S_{CC\text{-}post}$, $S_{PC\text{-}post}$, and $S_{EX\text{-}post}$. The only restraint placed on these events is that they do not perturb any state that has already been established as part of the precondition, or has not already been verified as part of the expected post-condition.

This results in three sub-divisions of the validation procedure: the *setup*, the *execution*, and the *check*. For example, a particular AM is chosen from the list of legal AM's for an instruction according to a given probability distribution. The AM has a *value* associated with it (the MR state modified) according to a given probability distribution. The *setup* involves the generation of code sequences that cause the AM to fetch the associated *value* (MR value) at instruction execution. The *setup* also includes the generation of code to establish the proper CC state, as well as the proper PC state. The *execution* phase causes a code sequence for the instruction to be generated that reflects the chosen AM's (MR state) and PC state. In the *check* phase, a code sequence is generated that determines whether the instruction execution produced the expected resulting state.

In like fashion each potential state modification of the instruction undergoes one or all of the above sub-division phases. Sub-division phases, for all divisions, take place at one time. That is the setup phases for all divisions take place, then the execution phases, and then the check phases.

3. The Interpreter

The validation system interpreter is written in a dialect LISP [7]. The kernel of the validation system is a *modified* LISP interpreter. It was derived from one described by Steele and Sussman [10].

The modifications include:

- Implementing the original interpreter in MacLISP [8], a dialect of LISP developed at MIT.
- Adding a *production* construct:
- Adding an *equation solving* construct.
- Adding certain *primitive operations* which are to be used by the validation system. These *primitive operations* are divided into two categories as follows.
    - General LISP primitive operations.
    - Validation system specific primitive operations which are described in the following chapters.

The interpreters I/O capabilities are described in Chapter 7.

3.1 Functions

A function definition is a list consisting of: the keyword FUNCTION, the Function Name *Fname*, a list of Format Parameters *FPj*, and an Symbolic Expression *Sexpr*. That is:

(FUNCTION *Fname* (*FP1 FP2 ...*) *Sexpr*)

The arguments to a function, the Actual Parameters, are bound to the Formal parameters at the time of the functions evaluation.

Functions are composed of calls to *General Primitive, Validation System Specific Primitive, or User defined* functions. Several *Primitive Functions* currently exist and are enumerated in Section 3.1.3. The *Validation System Specific Primitive* functions are described in the following chapters.

3.1.1 Function Call

A *Function call* is distinguished from a *Production call* in that the first element in the function call must evaluate to an ATOM which corresponds to the function name. That is, a function call is a list consisting of: a Function Name *Fname* followed by some number of Actual Parameters *ActPj*. That is:

(*Fname ActP1 ActP2 ...*)

A production call is a list consisting of: a list of Activation Parameters *APn* followed by some number of Actual Parameters *ActPn*. That is:

((*AP1 AP2 ...*) *ActP1 ActP2 ...*)

3.1.2 Special Forms

The following is a list of special forms which are of general use.

PROGN Takes any number of arguments and evaluates the arguments in the order given. The value returned is that of the last function evaluated.

REPEAT Takes any number of arguments. The first argument is a repeat count. The remaining arguments are evaluated in PROGN fashion the number of times specified by the repeat count. If the repeat count is less than one the remaining arguments are NOT evaluated.

LAMBDA Function call binding mechanism without the function name (As in LISP).

SETQ Takes two arguments. The first argument is not evaluated. The value of the second argument is assigned to the first argument.

COND — The traditional LISP conditional mechanism.

3.1.3 General Primitive Functions

The following is a list of primitive functions which are of general use.

EQ — Takes two arguments. Returns 'T' if the arguments are identical, otherwise it returns 'NIL'.

CONS — Takes two arguments and returns the CONS of the arguments.

LIST — Takes any number of arguments and returns a list of the arguments.

CAR — Takes one argument and returns the *left* hand portion of its *CONS* cell.

CDR — Takes one argument and returns the *right* hand portion of its *CONS* cell.

ASSQ — Takes two arguments. The first argument must evaluate to an atom. The second argument is a list of CONS cells. The function scans the CAR of each CONS cell until one matches the first argument. The function then returns that CONS cell.

NULL — Takes one argument and returns 'T' if the value of the argument is 'NIL', otherwise it returns 'NIL'.

QUOTE — Takes one argument, an item to be quoted. The item quoted is not evaluated. That is the character string that is quoted (argument) is to be taken as a literal.

\- — Takes one or more arguments. If one argument is given the two's complement negative of that argument is returned. If more than one arguments are given then the first argument has subtracted from it all the remaining arguments. The subtraction is two's complement.

\+ — Takes two or more arguments. The arguments are added together and their sum is returned.

1- — Takes one argument. The value returned is that of the argument minus one.

1 + — Takes one argument. The value returned is that of the argument plus one.

* — Takes two or more arguments. The arguments are multiplied together and their product is returned.

//	Takes one or more arguments. If one argument is given, it's reciprocal is returned. If more than one argument is given, the first argument is divided by the following arguments and the quotient is returned.
\	Takes two arguments. The remainder of the first argument divided by the second argument is returned.
\\	Takes two arguments. The value returned is the greatest common denominator of the arguments.
↑	Takes two arguments. The value returned is that of the first argument raised to the power of the second argument.
ROT	Takes three arguments. The first argument is rotated by the value specified by the second argument. If the second argument is positive then the first argument is rotated right. If the second argument is negative then the first argument is rotated left. The third argument specifies the size of the register that the rotation takes place in as a power of two.
LSH	Takes two arguments. The first argument is shifted by the value specified by the second argument. If the second argument is positive then the first argument is multiplied by two raised to the power of the second argument. If the second argument is negative then the first argument is divided by two raised to the power of the second argument.
COB	Takes one argument. The value returned is the count of the powers of two that the argument is composed of.[9]
ABS	Takes one argument. The value returned is the absolute value of the argument.
NOT	Takes one argument. The value returned is the ones complement negative of the argument.
AND	Takes three arguments. The value returned is the logical AND of the first and second arguments. The third argument specifies the size of the register that the AND operation takes place in.
OR	Takes three arguments. The value returned is the logical OR of the first and second arguments. The third argument specifies the size of the register that the OR operation takes place in.

[9] Historically *count one bits*.

XOR Takes three arguments. The value returned is the logical XOR of the first and second arguments. The third argument specifies the size of the register that the XOR operation takes place in.

3.2 Production Construct

A production definition is a list consisting of: the keyword **PRODUCTION**, a list of Activation Parameters *APn*, a list of Formal Parameters *FPn*, and an Symbolic Expression *Sexpr*. That is:

(PRODUCTION (*AP1 AP2 ...*) (*FP1 FP2 ...*) *Sexpr*)

3.2.1 Activation Parameters

The Activation Parameters in the production definition are compared, considered as (unordered) sets, with the Activation Parameters in a production call. A list of productions is constructed whose *AP's* are equal (as a set) to the production call *AP's*. From this list one production is selected and executed. If no production can be found with the above properties a list is constructed corresponding to the productions whos *AP's* are a superset of the production call *AP's*. From this list one production is selected and executed. If no production can be found with the above properties an error is generated.

3.2.2 Formal Parameters

The Formal Parameters are bound to the Actual Parameters when the production is invoked.

3.2.3 Production Call

A production call is a list consisting of: a list of Activation Parameters *APn* followed by some number of Actual Parameters *ActPn*. That is:

((*AP1 AP2 ...*) *ActP1 ActP2 ...*)

3.2.4 Example

The production call:

```
((friend-of george) clyde)
```

will invoke a production from the set of productions which contain 'friend-of' and 'george' in their activation parameters.

Of the following production definitions only the first two productions will be in this set (Here no exact match is found). One will be chosen arbitrarily for execution.

```
(PRODUCTION (george sam friend-of neighbor) (person) Sexpr)
(PRODUCTION (friend-of george guy) (pal) Sexpr)
(PRODUCTION (friend-of sam guy) (person) Sexpr)
(PRODUCTION (pet-of sam) (name) Sexpr)
```

Of the following production definitions only the last production will be in the set (Here an exact match is found). It will be chosen for execution.

```
(PRODUCTION (george sam friend-of neighbor) (person) Sexpr)
(PRODUCTION (friend-of george guy) (pal) Sexpr)
(PRODUCTION (friend-of sam guy) (person) Sexpr)
(PRODUCTION (pet-of sam) (name) Sexpr)
(PRODUCTION (george friend-of) (person) Sexpr)
```

3.3 Equation Solving

An Equation definition consists of the string EQUATION followed by the Equation Key *eq-key* followed by a list of Formal Parameters *FPn* followed by the Equation Form *eq-form*.

(EQUATION *eq-key* (*FP1 FP2 ...*) *eq-form*)

The *eq-key* is used to refer to the equation definition in an invocation or call. A call consists of a list containing the Equation Key *eq-key* followed by the Actual Parameters *APn*. The value of the Actual Parameters *APn* are bound to the Formal Parameters *FPn* in the Equation form *eq-form* of the instruction definition.

(*eq-key AP1 AP2 ...*)

The Equation Form is a Symbolic Expression (Sexpr) which contains function calls and assignments (a special function). The special function ' = = ' (the assignment) allows assignment to Formal Parameters *FPn* (variables) within the Equation Form. All variables are local to the Equation Form. Assignment may only take place if the value of the Actual Parameter bound to the Formal Parameter in the equation is '&NIL' and in addition assignment only takes place once.

The functions and assignments in the Equation Form are "solved" by iteration over the Equation Form. When the system encounters a function it will be executed "produce a non-&NIL result" only when *all* of it's parameters are bound to a non-&NIL value (otherwise it produces &NIL). There is therefore no explicit ordering of execution. This iterating over the Equation Form continues until all the Formal Parameter bindings are non-&NIL at the end of an iteration (The maximum number iterations is one greater than the number of Formal Parameters. When this number if iterations is exceeded an error is generated.).

3.3.1 Example

The following example describes an equation which will produce a result which is dependent upon the values of the actual parameters of the call.

```
(EQUATION x (a b c)
  (LIST
    (== a (COND ((prime-p b) 'prime) (T 'not-prime)))
    (== b (random c))))
```

If the call is of the form:

```
(x '&NIL '&NIL 1024)
```

on the first iteration the actual parameter bound to 'a' is &NIL but the equation ' = = a' depends on

the result of the equation ' = = b' (the value of 'b' is &NIL), therefore insufficient information is available to solve this equation. The actual parameter bound to 'b' is &NIL so the equation ' = = b' is solved for (since 'c' has a non &NIL binding). On the second iteration 'b' has a non &NIL binding and the equation ' = = a' may be solved. A list will be produced the first item of which is either 'PRIME' or 'NOT-PRIME', depending on the value of the function 'prime-p' (this function returns T if the argument is a prime number or NIL if the argument is not a prime number). The second item of the list will be a number selected randomly in the range '0' to 'c-1'.

If the call is of the form:

```
(x '&NIL 20 1024)
```

On the first iteration the actual parameter bound to 'a' is &NIL and 'b' has a non &NIL bind, therefore the equation ' = = a' is solved. The actual parameter bound to 'b' is non &NIL (the equation has been "solved previously") so the equation ' = = b' is NOT solved. A list will be produced whose first item is 'NOT-PRIME' and whose second item item is the number 20. That is:

```
(NOT-PRIME 20)
```

4. Second Level Validation Program *Template*

4.1 The Validation Program

The *Level II Validation Program* consists of several parts which are described in the following sections. The parts which constitute a program may be placed directly in the program or in separate files, being referenced through the **READ** function (See Section 7). The extension of a Second Level Validation Program file shall be **HLV**.

4.2 Level II Instruction Definitions

The *Level II Instruction Definitions* file shall have the extension L2. A Level II Instruction Definition consists of Equations.

An example of a *Level II Instruction Definition* for the *NEBULA INC* instruction which specifies a *post condition* of a *positive condition code*, where *no exception should take place* is as follows:

```
(equation inc-p (size value-pre value-post cc-pre cc-post)
 (list 'no-exception--sd ''inc
  (list 'quote (list
   (list 'am
         (list (== size (SET bhw))
               (== value-pre (- value-post 1))
               (== value-post (RANGE 1 (1- (CDRASSQ size bhw-size))))))
   (list 'cc
         (== cc-pre (SET cc-all))
         (== cc-post 'p))))))
```

The above equation may be invoked with the following.

```
(inc-p '&NIL '&NIL '&NIL '&NIL '&NIL)
```

The equations will be solved to produce a *Level I Instruction Definition* which is actually a function call. It is important to note that each Level II Instruction Definition represents a "set" of Level I Instruction Definitions (one-to-many), only one of which is produced for each *Level II Instruction Call*. One possible *Level I Instruction Definition* produced from the above *Level II Instruction Definition* above is as follows:

```
(NO-EXCEPTION--SD 'INC '((AM (BYTE 5 6)) (CC N P)))
```

4.3 Level II Validation System Specific Primitive Functions

SET Takes one argument which is a list. The function returns a randomly chosen top level element of the list.

RANGE Takes one argument which is a list. The elements of the list are CONS cells. The values of a CONS cell describes a range. The CAR of the cell describes a lower limit. The CDR of the cell describes an upper limit. The function randomly chooses one of the CONS cells of the list. It then returns a randomly chosen number in the range described by that CONS cell.

CDRASSQ A short hand notation for the following.

```
(CDR (ASSQ item list))
```

RANGE-EQCAR Takes two arguments, an ATOM *key* and a *list*. If the *key* is EQ to the CAR of a 'sublist' of the list then the RANGE operation is done on the CDR of the 'sublist'.

SET-INRANGE Takes two arguments, a NUMERIC *key* and a *list*. If the numeric *key* is IN the RANGE of the CAR of a 'sublist' (the CAR of the 'sublist' is a CONS cell which contains a range pair) then the SET operation is done on the CDR of the 'sublist'.

AFFECT Takes three arguments, an ATOM *code*, an ATOM *code modifier*, and an *Affect list*. If the *code* is EQ to the CAR of a 'sublist' of the list and the list represented by the CADR of the 'sublist' contains the *code modifier* then the CADDR of the 'sublist' is returned.

4.4 Level II Table Definitions

The following tables are used by the *Level II Validation System Specific Functions* described above. These tables are designed for the NEBULA architecture and do not constitute all the tables used to validate the NEBULA architecture. These tables are included here to give some flavor of the types of data structures which are necessary to construct *Level II Validation* systems.

4.4.1 Condition Code Tables

These are used to group similar condition codes into sets. In general it is necessary to select one condition code from the set. This is done with the SET function. Some NEBULA specific examples of condition code tables are.

```
;;;
;;; List of all the NEBULA condition codes:

(setq cc-all '(
               p p-c p-t p-tc
               z z-c z-t z-tc
               n n-c n-t n-tc
               zn zn-c zn-t zn-tc))

;;;
;;; List of all the NEBULA condition codes that are NOT zero:

(setq cc-tz '(
              p p-c p-t p-tc
              n n-c n-t n-tc
              zn zn-c zn-t zn-tc))
```

4.4.2 Size Range Tables

It is sometimes necessary to select a number from a range which corresponds to the different data sizes that the architecture supports. This is done with the RANGE function. Some NEBULA specific examples are.

```
;;;
;;; Range distribution table.

(setq range-size '(
                   (0 . 127)
                   (128 . 32767)
                   (32768 . 214748367)))
```

4.4.3 Value Size Correspondence Tables

It is sometimes necessary to determine which addressing mode *sizes* will accommodate a particular value. This is done through the SET-INRANGE function. Some NEBULA specific examples are.

```
(setq range-bhw '(
                  ((0 . 127) byte halfword word)
                  ((128 . 32767) halfword word)
                  ((32768 . 214748367) word)))

(setq range-bh '(
                 ((0 . 127) byte halfword)
                 ((128 . 32767) halfword)))
```

4.4.4 Maximum Value for a Size Tables

It is sometimes necessary to determine what the maximum value for a particular access mode size is. Or what is the greatest power of two of a value that a size will accommodate. This is done through the function CDRASSQ. Some NEBULA specific examples are.

```
;;;
;;; Size table.

(setq bhw-size '(
                 (byte . 255)
                 (halfword . 32767)
                 (word . 214748367)))

;;;
;;; What power of two (w↑2) table ...

(setq bhw-am↑2 '(
                 (byte . 8.)
                 (halfword . 16.)
                 (word . 32.)))
```

4.4.5 Access Mode Size Tables

It is sometimes necessary to choose a particular access mode size. This is done through the SET function. Some NEBULA specific examples are.

```
(setq bhw '(byte halfword word))
```

4.4.6 Condition Code Affect table

It is sometimes necessary to modify a condition code to reflect some aspects of another condition code. This is done through the function AFFECT. Some NEBULA specific examples are:

```
(setq cc-affect '(
                  (p-↑t  (p-tc n-tc z-tc zn-tc) p-c)
                  (n-↑t  (p-tc n-tc z-tc zn-tc) n-c)
                         ...
                  (p-↑t  (p-c n-c z-c zn-c) p-c)
                  (n-↑t  (p-c n-c z-c zn-c) n-c)
                         ...
                  (z      (p n z zn) z)
                  (zn     (p n z zn) zn)
                         ...
                  (-↑t    (p-tc p-c)   p-c)
                  (-↑t    (zn-t  zn)   zn)
                         ...
                  (-t    (p-tc p-c)   p-tc)
                  (-t    (p-t  p)     p-t)
                  (-t    (zn-t  zn)   zn-t)))
```

5. First Level Validation Program *Template*

5.1 The Validation Program

The *Level I Validation Program* consists of several parts which are described in the following sections. The parts which comprise the program may either be placed directly in the program or in separate files and referenced through the **READ** function (See Section 7). The Level I Validation program file shall have the extension **MLV**.

5.2 Level I Instruction Definitions

The *Level I Instruction Definition* is a function call to an *Instruction Type function*. One *Instruction Type function* should exist for each "type" of instruction. This "type" classification reflects various broad aspects of an instruction. Some of the aspects reflected in the "type" for the NEBULA architecture might be.

- The instructions takes exceptions.
- The instructions does NOT take exceptions.
- The instruction takes branches.
- The instruction does not take branches.
- The instruction has one source and one destination.
- The instruction has two sources and one destination.
- The instruction has one source.

5.2.1 Instruction Type Function

This function is used to determine the "order of execution" of the productions which process the *assertions* about the instruction (state) to be validated. The following is an example of an *Instruction Type Function* for the NEBULA architecture to handle Level I Programs of the form:

```
(no-exception--s-s-d name
  ((am (size pre-condition-value)
       (size pre-condition-value)
       (size post-condition-value))
   (cc pre-condition post-condition)))
```

The instruction "type" reflected here has the following constraints:

- The instructions must take no exceptions.
- The operand type is of the form: Source, Source, Destination.

The *Instruction Type Function* is as follows:

```
(function no-exception--s-s-d (name form)
  (progn
   (deallocate 'register) ;        Free up the registers between tests
   ((lambda (am1 am2 am3 cc1 cc2)
      ((lambda (s1 s2 d)
         (progn
          (('cc 'pre 'no-exception) (cdr (assoc cc1 cc-table)))
          (('no-exception--s-s-d name) s1 s2 d)
          (('cc 'post 'no-exception) (cdr (assoc cc2 cc-table))
                                     module-no)
          (('am 'post 'signed) (cadr am3) d module-no)))
       (('am 'value 'source (car am1)) (cadr am1))
       (('am 'value 'source (car am2)) (cadr am2))
       (('am 'destination (car am3)))))
    (car (cdar form))
    (cadr (cdar form))
    (caddr (cdar form))
    (car (cdadr form))
    (cadr (cdadr form)))
   name))
```

The instructions of a certain type are specified by a production which contains, as the activation parameter, the "type" and the instruction name. Some examples of this for the NEBULA architecture are.

```
;;;
;;; productions of the form 'NO-EXCEPTION--S-S-D' ...

(production (no-exception--s-s-d add) (s1 s2 d)
  (stream '|  ADD | s1 '|,| s2 '|,| d))

(production (no-exception--s-s-d sub) (s1 s2 d)
  (stream '|  SUB | s1 '|,| s2 '|,| d))

(production (no-exception--s-s-d mul) (s1 s2 d)
  (stream '|  MUL | s1 '|,| s2 '|,| d))

(production (no-exception--s-s-d div) (s1 s2 d)
  (stream '|  DIV | s1 '|,| s2 '|,| d))

(production (no-exception--s-s-d mod) (s1 s2 d)
  (stream '|  MOD | s1 '|,| s2 '|,| d))
```

5.2.2 State Descriptors

The *State Descriptors* are used to specify the *assertion* about that state. There is no general form for the *State Descriptor* since an *Instruction Type function* may be constructed to handle any form. Some examples of *State Descriptors* for the NEBULA architecture are.

- The following is an AM *State Descriptor* for instructions which have one access mode of type: Source-Destination.

```
(am (size pre-condition-value)
    (size post-condition-value))
```

- The following is an AM *State Descriptor* for instructions which have two access modes of type: Source, Source-destination.

```
(am (size pre-condition-value)
    (size pre-condition-value post-condition-value))
```

- The following is an AM *State Descriptor* for instructions which have three access modes of type: Source, Source, Destination.

```
(am (size pre-condition-value)
    (size pre-condition-value)
    (size post-condition-value))
```

- The following is a CC *State Descriptor* for instructions which modify the condition codes.

```
(CC cc-pre-condition cc-post-condition)
```

Some of the valid *States* for the NEBULA architecture are.

AM	The memory/register (access mode) state key.
BRANCH	The take branch state key (boolean).
CC	The condition code state key.
EXCEPTION	The take exception state key (boolean).

5.2.3 Code Sequence

The Code Sequence is typically composed of *STREAM* function calls. The *STREAM* function is used to generate the Level 0 Validation Program text. The Code Sequence may also contain *Production Calls* and *Level I Function calls.*

5.2.4 Production Classification

The machine specific code can be divided as follows:

1. **Pre-diagnostic code.** Code emitted at the beginning of the diagnostic program module. This code can set the starting address of the program, specify a starting label or whatever.

2. **Post-diagnostic code.** Code emitted at the end of the diagnostic program module. For

example, the assembler may require an end statement and a starting address for the diagnostic program module.

3. **Set code.** Code used to place a value at a certain location, or set the *precondition* of a certain state. The location can be a memory address, a register, the condition codes, or whatever.

4. **Target instruction.** The code here is the instruction under test.

5. **Check code.** Code used to verify that a **location** contains a certain **value**, or check the *post condition* of a certain state. The location can be a memory address, a register, the condition codes, or whatever.

In general productions are created to handle the above areas (See Section 5.4 and 5.3).

5.3 Access Mode Productions

The function of an access mode (addressing mode) is to retrieve certain data from the MR (Memory/Register) state, for use by the processor. The validation program has the responsibility of setting up access modes to *point* to certain pieces of data (value range) in the MR space. That is, given a certain piece of data (value) and an access mode, the system must *Emit* the appropriate sequence of instructions to cause that access mode to 'evaluate to' the specified data (value) when the instruction is executed. The access mode definitions are created to do just this. However, rather than pointing to just one piece of data, the AM is given a "range" of values that it will *theoretically* be allowed to point or evaluate to. The "value range" is therefore a *set* of values.

5.3.1 Format of a Value Range

The *value range* is a list of three items: the *type*, the *value*, and the *size*. The *type* is created by the first attempt to allocate it. That is, there are no 'wired in' types. Some *types* used in the NEBULA architecture are.

1. NUMBER - type number or literal.

2. MEMORY - type memory address.

3. REGISTER - type register.

The value is a list that designates the form that the value is to take. The value may be a range of values, or it may be one value. A range of values is described by a list beginning with the identifier 'RANGE' and followed by *range-pair<s>*[10]*/probability* list. That is:

(RANGE (*range-pair<s> prob*) ... (*range-pair<s> prob*))

The range-pair is a dotted pair[11] consisting of the (inclusive) lower bound followed by the (inclusive) upper bound of the range. That is:

(*lower-bound . upper-bound*)

The probability fields are interpreted as being relative to the other probabilities specified. Actual selection probabilities are determined by normalizing the numbers given. The *probability* may be omitted, leaving only the *range-pair<s>*. In this case the range descriptor takes on the form:

---

[10] One or more range-pair.

[11] A left parenthesis followed by a value followed by a period followed by a value followed by a right parenthesis.

(RANGE *range-pair<s>*)

In this case the probability of selection is evenly distributed over the range-pair<s>.

The single value form consists of a list whose first element is the identifier 'VALUE' followed by the numeric value. This can be seen as a special case of the range form, and is included for the sake of convenience. That is:

(VALUE *190*)
(RANGE (*190* . *190*))
(RANGE (*190* . *190*) (*190* . *190*))

all mean the same thing.

The *size* is a number which describes the size in bytes that the value will occupy in the memory or register array. The *size* of a *value* located in a memory byte would be 1, in a half word 2, in a word 4, and in a double word 8.

The *value range* corresponding to a number whose value is 5 and is located in a half word would be:

(NUMBER (VALUE *5*) *2*)

The *value range* corresponding to the number whose range is -8 through 123 and is located in a word would be:

(NUMBER (RANGE (*-8* . *123*)) *4*)

5.3.2 AM Form

The AM form (or AM) is defined as a Lisp EXPR with one variable. The variable is the RANGE that the AM evaluates to upon instruction execution. The function must return an expression known as an AMEXPR-LIST. The AMEXPR-LIST is a list containing two lists, these are the EMIT-EXPR and the RANGE-EXPR.

The EMIT-EXPR is a list that describes how the AM is placed or emitted into the instruction stream. The list consists only of *literal* expressions.

The RANGE-EXPR has a form similar to the range and, in general, points to that place, in the MR state, where the RANGE has been stored. Thus a range (that may be a range of one value) is associated with an AM. The AM is said to evaluate to this range or fetch the value of (or pointed to by) the RANGE-EXPR range.

A function named AMEXPR is provided that will produce an AMEXPR-LIST from other AMEXPR-LISTS as well as literal expressions and numeric values. The AMEXPR is a function of two variables, an EMIT-EXPR (or list of EMIT-EXPR's) and RANGE-EXPR. The EMIT-EXPR can contain any of the following:

1. **Literal.** Causes a string to be emitted literally. The string is to be enclosed in double quotes (").

2. **EMIT-EXPR.** The entries of the EMIT-EXPR (argument) are placed, at the point of reference, into the EMIT-EXPR that the AMEXPR function generates.

3. **AMEXPR-LIST.** In this case the EMIT-EXPR from the AMEXPR-LIST is placed, at the point of reference, into the EMIT-EXPR that the AMEXPR function generates.

The RANGE-EXPR may be any one of the following:

1. **RANGE-EXPR.**

2. **AMEXPR-LIST.** In this case the RANGE-EXPR form the AMEXPR-LIST is extracted. The AMEXPR-LIST found here is typically produced be another AM form.

An example of an AM definition for the NEBULA memory byte, register, and register indirect byte mode is as follows:

```
(PRODUCTION (am memory byte value source source-destination index)
            (range)
 ((LAMBDA ($1)
    ((LAMBDA ($2)
       (PROGN
        (STREAM '|  MOVL  | (('am 'literal 'byte) range) '|,| $2)
        $2))
     (AMEXPR (list $1 '↑B) $1)))
  (ALLOCATE 'memory
            (list 'RANGE (cons buffer-bottom buffer-top))
            byte-size)))

(PRODUCTION (am register word value source source-destination index)
            (range)
 ((LAMBDA ($1 $2)
    ((LAMBDA ($3)
       (PROGN
        (STREAM '|  MOVL  | $1 '|,| $3)
        $3))
     (AMEXPR (list '% $2) $2)))
  (('am 'literal 'word 'value) range)
  (ALLOCATE 'register
            (list 'RANGE (cons register-bottom register-top))
            register-size)))

(PRODUCTION (am indirect byte value source source-destination index)
            (range)
 ((LAMBDA ($1)
    (AMEXPR (list '@ (('am 'register 'word 'value) $1) '↑B) $1))
  (('am 'memory 'byte 'value) range)))
```

In this example of the register indirect byte mode, the range is placed in a memory byte location. A register then takes on the range of the evaluation of that memory byte. The EMIT-EXPR is the "@" followed by the EMIT-EXPR of the AMEXPR-LIST generated by the 'register' production followed by the literal "↑B". The RANGE-EXPR is the RANGE-EXPR of the AMEXPR-LIST of the evaluation of the 'memory byte' production.

5.4 Setup and Check Productions

In the *set* and *check* routines, implementing more than one method of accomplishing the appointed task is recommended.

It is necessary to exercise as many different paths in the architecture as possible to build confidence in the check and set code sequences.

5.4.1 Access Mode Checking

For example, the following code may be used for an AM check of the NEBULA architecture[12] .

```
(PRODUCTION (am post signed) (value address module)
 ((LAMBDA (label)
    (PROGN
     (STREAM '|  CMP  #| value '|,| address)
     (STREAM '|  BEQL  | label)
     (STREAM '|  DISPLAY #|
             module
             '|,#1 ;          Failed access mode test|)
     (STREAM label '|:|)))
  (MAKE-LABEL 'amp)))
```

In this code sequence **access-mode** and **value** are both **EMIT-EXPR's**. The purpose of this code is to compare the literal value with the contents of the location and halt if they are different. Another code sequence that performs this task could be written as:

```
(PRODUCTION (am post signed) (value address module)
 ((LAMBDA (pass fail)
    (PROGN
     (STREAM '|  CMP  #| value '|,| address)
     (STREAM '|  BNE  | fail)
     (STREAM '|  B    | pass)
     (STREAM fail
             '|:  DISPLAY #|
             module
             '|,#1 ;       Failed access mode test|)
     (STREAM pass '|:|)))
  (MAKE-LABEL 'AMP)
  (MAKE-LABEL 'AMF)))
```

[12] Use of both methods above will catch errors in which the conditional branches *always* branch.

5.4.2 Condition Code Setup and Checking

The following productions may be used for the Condition Code Setup for the NEBULA architecture.

```
(production (cc pre no-exception) (value)
 (stream '|  SETCC #| value))

(production (cc pre exception) (value)
 (stream '|  SETCC #| value '|.OR.↑X10|))
```

The following productions may be used for the Condition Code Checking for the NEBULA architecture.

```
(production (cc post no-exception) (value module)
 ((lambda (label)
    (progn
     (stream '|  CALL  GETCC,%1|)
     (stream '|  CMPU  %1,#| value)
     (stream '|  BEQL  | label)
     (stream '|  DISPLAY #|
             module
             '|,#2 ;        Failed condition code test|)
     (stream label '|:|)))
  (make-label 'CCP)))

(production (cc post no-exception) (value module)
 ((lambda (pass fail)
    (progn
     (stream '|  CALL  GETCC,%1|)
     (stream '|  CMPU  %1,#| value)
     (stream '|  BNE   | fail)
     (stream '|  B     | pass)
     (stream fail
             '|: DISPLAY #|
             module
             '|,#2 ;    Failed condition code test|)
     (stream pass '|:|)))
  (make-label 'CCP)
  (make-label 'CCF)))
```

5.5 Level I Validation System Specific Primitive Functions

STREAM Takes any number of arguments. Each argument is output to the instruction stream in the following manner.

1. **Literal.** Causes the string to be emitted into the instruction stream literally. The string is to be enclosed in double quotes.

2. **Number.** Causing the text equivalent of that number to be emitted into the instruction stream.

3. **EMIT-EXPR.** The entries of the EMIT-EXPR (argument) are placed, at the point of reference, into the instruction stream.

4. **AMEXPR-LIST.** In this case the EMIT-EXPR from the AMEXPR-LIST is placed, at the point of reference, into the instruction stream.

5. **Variable.** Containing either a literal, a number, an EMIT-EXPR, or an AMEXPR-LIST.

AMEXPR Takes two arguments: an emit-list and a value-list. This function creates an *Expression* for the *AM* object (AMEXPR).

ALLOCATE Takes three arguments: a *type*, a *range* and a *size*. This function creates a value-list from its arguments.

DEALLOCATE Takes one argument: a *type*. This function *deallocates* all allocations made to that type.

AM- Takes any number of arguments. Each argument can be either a *number* an *AMEXPR* or a *VALUE-LIST*. This subtracts the first argument from the other arguments.

AM+ Takes any number of arguments. Each argument can be either a *number* an *AMEXPR* or a *VALUE-LIST*. This function adds the first argument to the other arguments.

AM* Takes any number of arguments. Each argument can be either a *number* an *AMEXPR* or a *VALUE-LIST*. This function multiplies the first argument with the other arguments.

MAKE-LABEL Takes one argument. This function generates labels for the assembly language program. It does this by creating a symbol which consists of a *unique* number appended to the argument.

5.6 Level I Table Definitions

In most instances, these tables are used in the "mapping" of certain constants in the Level I Programs to the code emitted in the Level 0 (target assembly language) Program.

5.6.1 Condition Code Definition Table

The valid *CC-pre-exec-state* and *post-exec-state* specifiers for the NEBULA architecture are defined as doted pairs. The first element is the name of the *exec-state* specifier. The second element is numerical representation (to be emitted) of that specifier as reflected in the condition codes. The *exec-state* specifiers for the NEBULA architecture would be defined as follows:

```
(p      . "↑X0")
(z      . "↑X1")
(n      . "↑X2")
(zn     . "↑X3")
(p-t    . "↑X4")
(t      . "↑X5")
(n-t    . "↑X6")
(zn-t   . "↑X7")
(p-c    . "↑X8")
(z-c    . "↑X9")
(n-c    . "↑X0A")
(zn-c   . "↑X0B")
(p-tc   . "↑X0C")
(z-tc   . "↑X0D")
(tc     . "↑X0E")
(zn-tc  . "↑X0F")
```

5.6.2 Exception Code Definition Tables

The definition of the EXCEPTION state specifiers is similar to that of the CC state specifiers. The valid NEBULA EXCEPTION state specifiers are as follows:

```
(Specification-Error . "↑.1")
(Illegal-Mode . "↑.2")
(Illegal-Parameter . "↑.3")
(Illegal-Register . "↑.4")
(Illegal-Write . "↑.5")
(Illegal-Size . "↑.6")
(Illegal-Address . "↑.7")
(Operand-Size . "↑.8")
(Context-Alignment . "↑.9")
(Context-Base . "↑.10")
(Segment-Specifier . "↑.11")
(Supervisor-Check . "↑.12")
(Task-Load-Error . "↑.13")

(Illegal-Divisor . "↑.16")
(Truncation . "↑.17")
(Range-Error . "↑.18")
(Invalid-Operation . "↑.19")
(Divide-By-Zero . "↑.20")
(Floating-Overflow . "↑.21")
(Floating-Underflow . "↑.22")
(Floating-Inexact . "↑.23")
(Unordered . "↑.24")

(Task-Failure . "↑.32")
(Break . "↑.33")
(Instruction-Break . "↑.34")
(Call-Break . "↑.35")
```

6. A Program Example

6.1 Level II Program

The following is a *Level II validation template* for the NEBULA ADD instruction in which the condition code results should be positive.

```
(equation add-positive (s1-range s1-size s1-value s2-range s2-size s2-value
                        d-range d-size d-value cc-range cc-pre cc-post)
          (list 'NO-EXCEPTION--S-S-D ''ADD
           (list 'quote (list
            (list 'AM
                  (list (== s1-size (SET-INRANGE s1-value s1-range))
                        (== s1-value (- d-value s2-value)))
                  (list (== s2-size (SET-INRANGE s2-value s2-range))
                        (== s2-value (RANGE (cons 0 d-value))))
                  (list (== d-size (SET d-range))
                        (== d-value
                            (RANGE (cons 1
                                         (1- (CDRASSQ d-size bhw-size))))))))
            (list 'CC
                  (== cc-pre (SET cc-range))
                  (== CC-POST 'p))))))
```

The following is a *Level II program* which makes use of the above *Level II template*.

```
;;;
;;; Read this into 'HLV' to make a 'MLV' program.
;;;

(read funs) ;               random LISP functions that are 'not primops'.
(read table) ;              Level II TABLEs.
(read eq) ;                 Level II PROGRAM.

(function make-llv ()
  (progn
   (printl '(read funs)) ; random LISP functions that are 'not primops'.
   (printl '(read instr)) ;Level I 'Instruction' production definition file.
   (printl '(read am)) ;   Level I 'Access Mode' production definition file.
   (printl '(read pp)) ;  Level I 'Pre and Post' production definition file.
   (printl '(read top)) ;  Level I 'Top Level' productions and functions.
   (prints1 '(file |llv.neb|)) ; The Lvl I program will produce a Lvl 0 prog
   (printl '|(('module 'pre 'no-exceptions) 1 module-no)|)
   (repeat 10
           (printl (add-positive range-bhw '&NIL '&NIL
                                 range-bhw '&NIL '&NIL
                                 bhw '&NIL '&NIL
                                 cc-all '&NIL '&NIL)))
   (printl '|(('module 'post 'no-exceptions) module-no)|)
   (printl '(close)))) ;   Close the Level 0 program that was just created.
```

```
(file makllv) ;          Make a 'Level I' program.
(make-llv) ;             Invoke the above function.
(close) ;                Close the Level I program that was just created.
```

6.2 Level I Result

The following is one instance of a *Level I Validation Program* that would be produced, by the *validation system*, from the above *Level II Validation Program*. Many such instances will be produced, by the *validation system*, to realize a complete validation program.

```
(READ FUNS)
(READ INSTR)
(READ AM)
(READ PP)
(READ TOP)
(FILE |llv.neb|)
(('module 'pre 'no-exceptions) 1 module-no)
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (WORD 52103075) (WORD 127824650) (WORD 179927725)) (CC Z-TC P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (BYTE 54) (WORD 93) (BYTE 147)) (CC P-TC P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (HALFWORD 3657) (HALFWORD 20148) (HALFWORD 23805)) (CC Z-TC P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (WORD 4379) (HALFWORD 3327) (HALFWORD 7706)) (CC P-C P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (WORD 96) (HALFWORD 492) (HALFWORD 588)) (CC Z-TC P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (WORD 3270609) (WORD 802639) (WORD 4073248)) (CC N-TC P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (BYTE 197) (WORD 6357) (HALFWORD 6554)) (CC P-T P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (WORD 1896) (WORD 15903) (HALFWORD 17799)) (CC P-TC P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (WORD 8321) (HALFWORD 2274) (HALFWORD 10595)) (CC N-C P)))
(NO-EXCEPTION--S-S-D 'ADD
 '((AM (WORD 6781824) (WORD 24707157) (WORD 31488981)) (CC Z P)))
(('module 'post 'no-exceptions) module-no)
(CLOSE)
```

6.3 Target Result

The following is an example of a *Validation program* for the NEBULA computer architecture. The program is produced by the Validation System from the *Level I validation* programs. The *Validation program* is output, by the *Validation system*, in the assembly language for the target architecture. The NEBULA code segment between the ellipsis is the code sequence generated, by the validation system, from the above *First level validation* program.

```
; Machine Generated Diagnostic program for the NEBULA architecture.
; Test Module Number 100.
; Module generated in 1 passes.


   .=↑X100060   ; Origin: program runs in supervisor state

;
;  Routine to access the Condition Codes of the caller ...
GETCC:
   .ENTRY NP=1, REG=1
   SPSW    %1
   LSH     #-↑.19,%1,%1
   AND     #↑X0F,%1
   MOVL    %1,?1
   RET


START:
   .ENTRY REG=15, NP=0, SUPV, NOAE

   MOVL   #52103075↑W,874↑W
   MOVL   #1297880708↑W,%12
   MOVL   #-324470065↑W,880↑W
   SETCC #↑XD
   ADD 1246↑W[#-93↑B]↑W,#127824650↑W,-84(%12)↑W[880↑W]↑W
   CALL   GETCC,%1
   CMPU   %1,#↑X0
   BEQL   CCP$1
   DISPLAY #100,#2 ;          Failed condition code test
CCP$1:
   CMP   #179927725,-84(%12)↑W[880↑W]↑W
   BNE   AMF$3
   B     AMP$2
AMF$3:   DISPLAY #100,#1 ;        Failed access mode test
AMP$2:
   MOVL   #54↑B,73↑B
   MOVL   #-1862536012↑W,%5
   MOVL   #1862536095↑W,%13
   MOVL   #93↑W,488↑W
   MOVL   #-1958666778↑W,%8
   MOVL   #1958667266↑W,534↑W
   MOVL   #422↑W,%10
   MOVL   #244↑W,%14
   SETCC #↑XC
   ADD -10(%5)↑B[%13]↑B,@%8↑W(112(%10)↑W)↑W,-16(%14)↑B
   CALL   GETCC,%1
   CMPU   %1,#↑X0
   BNE    CCF$5
   B      CCP$4
CCF$5: DISPLAY #100,#2 ;      Failed condition code test
CCP$4:
   CMP   #147,-16(%14)↑B
   BEQL   AMP$6
   DISPLAY #100,#1 ;          Failed access mode test
```

```
AMP$6:
  MOVL  #3657↑H,342↑H
  MOVL  #467↑W,%9
  MOVL  #-125↑B,654↑B
  MOVL  #654↑W,%4
  MOVL  #20148↑H,39↑H
  MOVL  #1485843947↑W,263↑W
  MOVL  #263↑W,%12
  MOVL  #-1966732779↑W,%5
  SETCC #↑XD
  ADD @%9↑H(@%4↑B)↑H,-1485843908↑H(@%12↑W)↑H,1966732821(%5)↑H
  CALL  GETCC,%1
  CMPU  %1,#↑X0
  BNE   CCF$8
  B     CCP$7
CCF$8: DISPLAY #100,#2 ;    Failed condition code test
CCP$7:
  CMP  #23805,1966732821(%5)↑H
  BEQL  AMP$9
  DISPLAY #100,#1 ;          Failed access mode test
AMP$9:
  MOVL  #4379↑W,480↑W
  MOVL  #299200269↑W,%8
  MOVL  #74↑B,695↑B
  MOVL  #695↑W,%15
  MOVL  #3327↑H,867↑H
  MOVL  #-1757526795↑W,%5
  MOVL  #-13995↑H,26↑H
  MOVL  #-4↑W,%11
  SETCC #↑X8
  ADD -299199863(%8)↑W(@%15↑B)↑W,867↑H,1757541180(%5)↑H(30(%11)↑H)↑H
  CALL  GETCC,%1
  CMPU  %1,#↑X0
  BEQL  CCP$10
  DISPLAY #100,#2 ;         Failed condition code test
CCP$10:
  CMP  #7706,1757541180(%5)↑H(30(%11)↑H)↑H
  BEQL  AMP$11
  DISPLAY #100,#1 ;          Failed access mode test
AMP$11:
  MOVL  #96↑W,566↑W
  MOVL  #566↑W,%4
  MOVL  #224↑W,%12
  SETCC #↑XD
  ADD @%4↑W,#492↑H,100(%12)↑H
  CALL  GETCC,%1
  CMPU  %1,#↑X0
  BEQL  CCP$12
  DISPLAY #100,#2 ;         Failed condition code test
CCP$12:
  CMP  #588,100(%12)↑H
  BNE  AMF$14
  B    AMP$13
AMF$14:  DISPLAY #100,#1 ;       Failed access mode test
AMP$13:
```

```
  MOVL  #802639↑W,%13
  MOVL  #-523630373↑W,%9
  SETCC #↑XE
  ADD #3270609↑W,%13,523630563(%9)↑W
  CALL  GETCC,%1
  CMPU  %1,#↑X0
  BNE   CCF$16
  B     CCP$15
CCF$16: DISPLAY #100,#2 ;    Failed condition code test
CCP$15:
  CMP  #4073248,523630563(%9)↑W
  BEQL  AMP$17
  DISPLAY #100,#1 ;          Failed access mode test
AMP$17:
  MOVL  #197↑B,784↑B
  MOVL  #1696351218↑W,%15
  MOVL  #28179↑H,532↑H
  MOVL  #412↑W,%8
  MOVL  #6357↑W,1018↑W
  MOVL  #956↑W,%12
  MOVL  #62↑B,809↑B
  MOVL  #719↑W,%13
  MOVL  #-1222422648↑W,%14
  MOVL  #611211503↑W,%5
  SETCC #↑X4
  ADD -1696378613(%15)↑B(120(%8)↑H)↑B,@%12↑W(90(%13)↑B)↑W,-105(%14)↑H[%5]↑H
  CALL  GETCC,%1
  CMPU  %1,#↑X0
  BNE   CCF$19
  B     CCP$18
CCF$19: DISPLAY #100,#2 ;    Failed condition code test
CCP$18:
  CMP  #6554,-105(%14)↑H[%5]↑H
  BNE  AMF$21
  B    AMP$20
AMF$21:  DISPLAY #100,#1 ;      Failed access mode test
AMP$20:
  MOVL  #1896↑W,810↑W
  MOVL  #34↑B,545↑B
  MOVL  #1702625594↑W,%11
  MOVL  #15903↑W,972↑W
  MOVL  #198↑W,%13
  MOVL  #102↑B,816↑B
  MOVL  #816↑W,%7
  SETCC #↑XC
  ADD 776↑W(-1702625049(%11)↑B)↑W,972↑W,-128(%13)↑H[@%7↑B]↑H
  CALL  GETCC,%1
  CMPU  %1,#↑X0
  BEQL  CCP$22
  DISPLAY #100,#2 ;        Failed condition code test
CCP$22:
  CMP  #17799,-128(%13)↑H[@%7↑B]↑H
  BNE  AMF$24
  B    AMP$23
AMF$24:  DISPLAY #100,#1 ;      Failed access mode test
```

```
AMP$23:
  MOVL  #8321↑W,%9
  MOVL  #2274↑H,135↑H
  MOVL  #2031628647↑W,%10
  MOVL  #-18367↑H,258↑H
  MOVL  #257↑W,%5
  MOVL  #753↑W,%7
  MOVL  #-49↑B,137↑B
  MOVL  #137↑W,%11
  SETCC #↑XA
  ADD %9,-2031591778(%10)↑H[1(%5)↑H]↑H,-108(%7)↑H[@%11↑B]↑H
  CALL  GETCC,%1
  CMPU  %1,#↑X0
  BNE   CCF$26
  B     CCP$25
CCF$26: DISPLAY #100,#2 ;    Failed condition code test
CCP$25:
  CMP  #10595,-108(%7)↑H[@%11↑B]↑H
  BNE  AMF$28
  B    AMP$27
AMF$28:  DISPLAY #100,#1 ;    Failed access mode test
AMP$27:
  MOVL  #6781824↑W,%6
  MOVL  #24707157↑W,657↑W
  MOVL  #657↑W,%10
  MOVL  #-1141531414↑W,%4
  SETCC #↑X1
  ADD %6,@%10↑W,1141531596(%4)↑W
  CALL  GETCC,%1
  CMPU  %1,#↑X0
  BEQL  CCP$29
  DISPLAY #100,#2 ;        Failed condition code test
CCP$29:
  CMP  #31488981,1141531596(%4)↑W
  BEQL  AMP$30
  DISPLAY #100,#1 ;         Failed access mode test
AMP$30:
  DISPLAY #100,#0 ;     Successful Termination

  .END START+↑X80000001 ;          Run in Supervisor space.
```

7. IO

7.1 READ

The READ function takes the form:

```
(READ device:name.extension)
```

All the Symbolic Expressions (Sexpr) contained in the referenced file are evaluated.

7.2 FILE

The FILE function takes the form:

```
(FILE device:name.extension)
```

This function causes all future output to be directed to the specified file. If the file specification is 'T' then output is directed to the terminal. This command has the side effect of preforming a CLOSE on previous file.

7.3 CLOSE

The CLOSE function takes the form:

```
(CLOSE)
```

The file which is currently open for output is closed and all future IO is directed to the terminal. If the currently opened file is the terminal this command has no effect.

7.4 PRINT

The PRINT function takes the form:

```
(PRINT Sexpr)
```

The *Sexpr* is output to the file named in the **FILE** command. Special characters are *not* printed with vertical bars.

7.5 PRINTL

The PRINTL function takes the form:

```
(PRINTL Sexpr)
```

The *Sexpr* is output to the file named in the FILE command. Special characters are *not* printed with vertical bars. An end of line character is then output to the file named in the FILE command.

7.6 PRINTS

The PRINTS function takes the form:

```
(PRINTS Sexpr)
```

The *Sexpr* is output to the file named in the FILE command. Special characters are printed with vertical bars.

7.7 PRINTSL

The PRINTSL function takes the form:

```
(PRINTSL Sexpr)
```

The *Sexpr* is output to the file named in the FILE command. Special characters are printed with vertical bars. An end of line character is then output to the file named in the FILE command.

Notes

References

1. *PDP-11 Handbook.* Digital Equipment Corporation, Maynard, Massachusetts, 1969.

2. Dahl, Dijkstra, Hoare. *Structured Programming.* Academic Press, 1972.

3. Hoare, C. A. R. "An Axomatic Basis for Computer Programming." *Communications of the ACM 12*, 10 (October 1969).

4. Barbacci, Barnes, Cattell, and Sieworek. Symbolic Manipulation of Computer Descriptions. Dept. of Computer Science, Carnegie- Mellon Univ., 1978.

5. King, James C. "Symbolic Execution and Program Testing." *Communications of the ACM 19*, 7 (July 1976).

6. Lai, Larry Kwok-woon. Error-oriented architecture testing. Proceedings of the NCC, National Computer Confrence, 1979.

7. McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; and Levin, Michael I.. *LISP 1.5 Programmer's Manual.* MIT Press, CAMBRIDGE, 1962.

8. Moon, David. *MacLISP Reference Manual, Revision 0.* M.I.T. Project MAC, CAMBRIDGE, April 1974.

9. John Oakley. Symbolic Execution of Formal Machine Descriptions. Dept. of Computer Science, Carnegie- Mellon Univ., 1979.

10. Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two). AI Memo 453, MIT Artificial Intelligence Lab., CAMBRIDGE, May, 1978.