UNLIMITED

(2)

Report No. 89012

Report No. 89012

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

THE ALGEBRA OF THE NODEN ANALYSER

Author: C H Pygott

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

August 1989

UNLIMITED

CONDITIONS OF RELEASE

0051916 BR-111545

********** U

********** Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 89012

TITLE: THE ALGEBRA OF THE NODEN ANALYSER

AUTHOR: C H Pygott

DATE: August 1989

SUMMARY

NODEN is a suite of programs designed to perform hardware analysis on moderately complex blocks of logic, to prove the correspondence between the specification and implementation of a circuit. It is intended that circuits to be analysed should be described in the NODEN Hardware Description Language, NODEN_HDL (either directly or by translation from other hardware description languages such as ELLA, HILO etc). The following paper describes the basic features of NODEN and the circuits it can reason about. The bulk of the paper describes the operations performed by the analyser in terms of set operations.

There is also a discussion of the possible representations that can be used for sets, and the operations on them. This leads to a comparison of the performance of a number of different analysers, based on different internal representations, when used on an actual application.

Contents

1 Introduction

NODEN is a hardware verification system that has grown out of RSRE's work on the VIPER micro-processor [Cullyer 87] The importance of hardware verification is seen as being twofold; firstly there is a class of applications in safety and security critical areas for which any unanticipated behaviour of the system (be it caused by hardware or software design errors or hardware failure) is unacceptable. Secondly, even for those applications which are not critical, an efficient design verification system could lead to reduced design testing costs and reduce the risk of devices being supplied to customers with undiscovered errors, with the inherent cost of reworking and loss of customer confidence. The work to be described covers certain aspects of the hardware design verification problem. Software verification and hardware fault tolerance are subjects of different research.

Whilst verifying the VIPER design [Cohn 87], it was found that there is a natural division between the high level abstract descriptions of a circuit and those levels closer to the implementation. In VIPER, four levels of description were produced, from the most abstract top level specification, through the major state (micro-program) and block models, to the gate level implementation. Proof of the correctness of the design was therefore in three parts; proving the correspondence between the major state model and the top level specification, the block and major state models, and finally between the gate level implementation and the block model. The first two of these are similar, particularly as in both cases the two descriptions involved have different views of time. In the top level specification, each machine instruction is described as an atomic event, but is implemented in the major state model by the execution of a sequence of micro-instructions. Similarly, whilst the major state model sees each micro-instruction as an atomic event, the block model sees them implemented as a sequence of clock cycles. This leads to these two proofs being performed in a similar manner, involving a machine assisted algebraic technique using the HOL theorem checker [Gordon 85].

Whilst the gate level to block model proof could be done using the same technique, there is a difference between this proof and the preceding pair, as the gate level and block models have the same view of time. These two descriptions are therefore simply different expressions of the same behaviour. The block model is a 'human orientated' view of the requirement in a comparatively abstract form, whilst the gate level description should be functionally identical but constructed from primitive functions that have some physical significance. This enables the use of a more automated proof style. For VIPER, a technique called intelligent exhaustive testing was developed [Pygott 85]. Whilst this worked successfully for VIPER, it was shown that it could be pessimistic, leading to correct circuits being rejected as erroneous [Pygott 88]. NODEN was the answer to this problem. It provides an automatic proof of the correspondence (or otherwise) between the block level specification of a circuit and its gate level implementation.

Before the details of NODEN are considered, it should be said that there is an alternative route to correct circuits, which does not depend on post-design verification, but produces the design by meaning-maintaining transformations on the specification. Such techniques applied to the top levels of description are still in the research stage [Brumfitt 87], but a

number of tools such as GATEMAP [Pitty 88] and LOCAM [Praxis 89] exist to synthesise a circuit from its block level description. However the use of such tools does not remove the need for NODEN and similar tools. As the synthesis tools are so complex, they will never be amenable to software verification, so for critical applications it would be difficult to trust them without an independent check on their operation. Also, the designer using these tools may decide to optimise the resulting circuit for a number of reasons, so a verification route is required to prove the legality of the optimisation.

The rest of this paper will provide an introduction to NODEN, its type model and the functions it supports. This is followed by a detailed description of the semantics of the built-in functions and constructs. As the usefulness of a tool such as NODEN depends upon the complexity of circuits it can analyse and how long that analysis takes, the next section describes a number of machine representations that can be used to implement the analyser, and finally the results of applying NODEN to a practical problem (VIPER) are discussed.

2 The NODEN type model

Circuits to be verified are presented to the analyser in the NODEN Hardware Description Language, NODEN_HDL. This is essentially a sub-set of ELLA [Morison 84], but with some additional information needed for verification. This section and the one that follows are intended to provide an overview of the facilities of NODEN, rather than provide a detailed syntactic description (which can be found in [Pygott 89]). One reason for this is that the circuits may never be directly expressed in NODEN_HDL, but may be specified in a hardware description language such as ELLA, with the implementation being described in a netlist language such as HILO, with these descriptions being translated into NODEN_HDL. This overview is therefore intended to indicate those feature of such description languages that can be reasoned about by NODEN.

There are four primitive types of object in a NODEN description; unconstrained integers, constrained integers, wire values and enumeration values.

In principle, an unconstrained integer can have any non-negative value from 0 to infinity. In practice, the upper bound of the integer range has been fixed at $2^{40} - 1$. It should be noted that the analyser cannot cope with integers this big, as the size of the internal expressions generated quickly become too large to handle. This means that a block such as a 32-bit arithmetic unit should be described as a number of slices, and the behaviour of the slices when joined together determined by an algebraic theorem prover.

In addition to the unconstrained integer type, NODEN also allows the user to define constrained integer types (such as 0..15 or 31..56). The only restrictions on the lower and upper bounds are that they must both be legal unconstrained integers, and the upper bound must be larger than the lower bound.

A wire type is a model of a value that can occur on a single wire, such as a boolean signal. A wire type is therefore indivisible. This should be contrasted with an enumeration type,

which represents the value of a group of signals. For example, the control lines to an ALU may form a set of signals. At the specification stage, the individual behaviour of the signals is not important, what is important is that some pattern of signals is required for an *ADD*, a different pattern for *SUB* etc. The enumeration type allows the specification to be written in an abstract manner, with the implication that in the implementation the single enumeration type will be mapped onto a number of physical wire values.

Any of the above four primitive types may be grouped into arrays of signals. For example an array of six signals, each of the wire type called *bool* is said to be of type *[6]bool*. Any combination of wire types, enumeration types, integers and arrays can be combined to form compound objects (or structures). For example, a value called *complex* may be of type:

```
(bool, word4, (tristate, bool), bool)
```

It is a user definable option as to whether arrays and structures are indexed from element 1 as the least significant (as ELLA and most HDL's) or 0 (as for HOL). If the lower bound of arrays is 1, then in the above example, *complex[1]* is of type *bool*, *complex[2]* is of type *word4* etc. Multiple levels of indexing are achieved such that *complex[3][1]* is of type *tristate*. Note that if *word4* is an array, the first member of the array is also indexed in the same manner, ie as *complex[2][1]*.

NODEN_HDL allows arrays and structures to be sliced, so that *complex[1..2]* would be a structure of type *(bool,word4)*. Arrays of the same wire or enumeration type may also be concatenated using the `CONC` operator.

As well as values made up of the members of a type, each type is assumed to have separate 'don't care' and 'illegal' (or 'undefined') values. That is, the user may specify a signal to be the dont-care value of a particular type under some circumstances, with the implication that the implementation can deliver any value under those circumstances. This is indicated by the use of the type name as a 'wild card' value, and should be contrasted with the interpretation of an incomplete description (for example the optional ELSE part omitted from an IF..THEN..ELSE..FI statement), when the analyser will regard the statement as having an undefined value of the appropriate type under the circumstances for which no value is defined (section 5.4). For example, the NODEN_HDL statement:-

```
IF a=b THEN c ELSE bool FI
```

delivers the value c when $a = b$, and the boolean 'don't care' value otherwise. If this is part of the specification of the circuit, this says that the implementation must deliver the value c when $a = b$, but can deliver any value at other times. This should be compared with the interpretation of:-

```
IF a=b THEN c FI
```

which delivers the value c when $a = b$, and is otherwise undefined. If this is part of the specification of a circuit, this would be regarded as illegal (if a could ever be unequal to

b), as the specification must deliver a defined value under all input conditions.

The majority of a NODEN description consists of type definitions and functions. Type definitions allow the user to define new wire, enumeration and constrained integer types. The unconstrained integer type is always available.

A new wire type is declared as:-

TYPE *typename* = WIRE (*name* | *name* | ..... *name*).

typename is the name of the type, whilst the names in brackets (separated by '|'s) are the values that that type can have. The predefined boolean type is effectively defined as:-

```
TYPE bool = WIRE ( t | f ).
```

A new enumeration type is declared as:-

TYPE *typename* = NEW *type* (*name* = #*boollist* | .....) .

typename is the name of the enumeration type. All enumeration types correspond to a group of booleans in the implementation, the *type* indicates the size of that group. Each of the members of the enumeration type are specified as *name*=#*boollist*, where the *name* is the name of the type member and the *boollist* is the representation of that member in the implementation, and consists of a list of the appropriate number of 0's (false), 1's (true) and x's (dont-care). The mapping of arrays of booleans to enumeration types and vice versa is discussed in sections 5.6 and 5.7. For initial definition and simulation, the representations can be omitted as in the first of the following examples (in which case the members will be given a representation equivalent to a binary count from 0):-

```
TYPE alucontrol = NEW [4]bool ( add | sub | shift ),
     counter    = NEW word2   (reset=#0x | inc=#10 | load=#11).
```

A new constrained integer type is declared as:-

TYPE *typename* = INT [*integer_expression* .. *integer_expression*].

as in:-

```
TYPE four_bit = INT[0..15].
```

3 NODEN functions

Behaviour is described to NODEN in terms of functions. Each function has a set of inputs and outputs, and may contain internal states (memory). The purpose of the function descriptions is to define how the outputs' behaviour depends upon the values of the inputs, and the current values of any internal states. The function description must also define how the next value of the internal states is derived from the same information. That is, as in ELLA there is an implied global clock that synchronises all state transitions, and the

descriptions of functions with internal states define the value the state variables will take after the next implied clock. This means that NODEN can only reason about synchronous logic.

3.1 Function signatures

NODEN recognises three basic function types; auxiliary functions (FNs), BLOCKs and CIRCUITs. Auxiliary functions are all those functions used to construct the description of a circuit, but which have no particular physical significance. BLOCKs, on the other hand, are functions which are identifiable in both the specification and implementation. That is, for any circuit that is to be verified, the specification description will contain a BLOCK which defines the required behaviour in terms of primitive operations and if required, auxiliary functions. The implementation description will contain a BLOCK that represents the implementation of the circuit. This will have the same inputs and outputs as the specification BLOCK (and the same internal states).

It has already been said that some circuits maybe too complex for NODEN to analyse, but the NODEN_HDL description may describe how these more complex circuits can be constructed from the BLOCKs that it can verify. This is done by CIRCUIT functions. So the description of a 32-bit arithmetic unit, which cannot be analysed directly, can be partitioned into 4-bit slices (represented as a BLOCK that can be verified) and a CIRCUIT which describes how the 32-bit ALU is constructed from eight of the 4-bit slices. CIRCUITs are ignored by the NODEN analyser, but may be used as part of the input to an algebraic prover in order to reason about the more abstract levels of description of a system.

Auxiliary functions and BLOCKs have similar signatures, in that both define the name of the function, the type of its inputs and their local names, and the type of its outputs. The only difference is that a BLOCK's outputs are named whilst an auxiliary function's are not. For example:-

```
FN EXAMPLE_FN = (bool: count clear, word4: reg) -> (word4,bool):
   ........

BLOCK B1 = ([4]bool: bus,  bool: reset) -> (bool: op1, tristate: op2):
   ........
```

In the first example, an auxiliary function is defined to take two boolean values named *count & clear* and a *word4* value named *reg*, and deliver a structure of type $(word4, bool)$. Similarly the `BLOCK B1` is defined as having two inputs and delivering a structure of type $(bool, tristate)$, with the two members of this structure being named as indicated. These names are used by the analyser and comparison programs to refer to a particular signal in their output listings. That is, they are not used in the definition of the function.

3.2 Function bodies

The body of a function either consists of a single NODEN statement, the value of which is the value delivered by the function, or a collection of local definitions and a final output statement. For example:-

```
FN A_NOT_B = (bool: a b) -> bool: a AND (NOT b).

FN A_INV_B = (bool: a b) -> bool:
   BEGIN LET not_b   = NOT b.
         LET a_not_b = a AND not_b.
         OUTPUT a_not_b
   END.
```

These two functions describe identical behaviour, where **NOT** and **AND** are two auxiliary functions. In general, any function is applied to a set of variables as in `F1(a,b,c,d)`, but monadic functions such as **NOT** do not need the brackets, and dyadic functions such as **AND** can be in-fixed as shown above. **NOT** and **AND** are actually two of the built-in functions that will be described later.

NODEN_HDL supports two conditional statements, **CASE** and **IF**. The **CASE** statement is of the form:-

CASE *exp*0 **OF** *value*1 : *exp*1, *value*2 : *exp*2, ... **ELSE** *expn* **ESAC**

Where *exp*0 is an expression that delivers a wire type, an enumeration type or an integer value. *value*1, *value*2 etc are different members of the type delivered by *exp*0. When *exp*0 has the value *value*1 the statement delivers *exp*1, when *exp*0 is *value*2 the statement delivers *exp*2 etc.. Not all the members of the type need explicitly appear as limb selectors, and if *exp*0 can ever have a value that doesn't match one of the selectors, the statement delivers the value *expn*. The **ELSE** *expn* part of the **CASE** statement is optional and if absent leads to the result of the statement being undefined for any value of *exp*0 not covered by the limb selectors. NODEN also supports an 'IF-THEN-ELSE-FI' construct, such that:-

```
IF •xp1 THEN •xp2 ELSE •xp3 FI
```

is identical to:-

```
CASE •xp1 OF t: •xp2, ELSE •xp3 ESAC
```

3.3 State variables

In order to introduce state information, a special class of functions (known as DELAY functions) are required. These are created as in:-

```
DELAY DELAY_BOOL = bool.
```

This creates a function called **DELAY_BOOL** with a single input of type *bool* and a single output of the same type. The output of this function is the input value delayed by one clock cycle.

As so far described, no statement can contain a value that has not already been declared, either as a member of a particular type, as an input to the function or as an expression named by a previous **LET**s. However, without some other mechanism it would be impossible to describe circuits that have feedback. For example, if **D_TYPE** is a function that is to model

a d-type latch with data and gate inputs and a single output (q), then what is required is that the output after the next implied clock should follow *data* if *gate* is true, or otherwise should be the same as the current output state. This cannot be written as:-

```
FN D_TYPE = (bool: data gate) -> bool:
   BEGIN LET q = DELAY_BOOL(IF gate THEN data ELSE q FI)
         OUTPUT q
     END.
```

as the name q is used on the right hand side of the assignment before the assignment is completed. To overcome this, it is possible to use a **MAKE** statement to indicate that a function of a particular type is to be defined later in the description, but that the name associated with that function is immediately available. Before the end of the function being defined, the inputs to any functions made in this way must be defined by **JOIN** statements. These are identical to the **MAKE** and **JOIN** statements in ELLA. The above example can therefore be legally expressed as:-

```
FN D_TYPE = (bool: data gate) -> bool:
   BEGIN MAKE DELAY_BOOL: q.
         JOIN IF gate THEN data ELSE q FI -> q.
         OUTPUT q
     END.
```

The NODEN_HDL compiler checks that any loops created as above contain at least one DELAY function, so that their behaviour is synchronous. Whilst it is perfectly possible to describe a circuit such as a pair of cross coupled NAND gates, these form an asynchronous memory device, and so the circuit cannot be analysed.

3.4 Predefined functions

NODEN supports a number of predefined functions. These can be grouped into four classes; boolean, comparisons, numerics and mappings. The basic boolean operations consist of inversion (**NOT**), **AND** and **OR**. However, to simplify (and improve the efficiency of) translating from a gate level hardware description language such as HILO to NODEN_HDL, these are extended to three, four and eight input **AND** and **OR** functions (**AND3**, **OR4** etc) and their inverses (**NAND**, **NOR8** etc). There is also a predefined boolean selection function **SEL2**, such that:-

```
SEL2(a,b,c) ≡ IF a THEN b ELSE c FI
```

NODEN supports the six possible numeric comparison operators between unconstrained integer values (ie ==, /=, <, <=, >, >=). It also supports equality and inequality (== & /=) operations between values of the same wire or enumeration type, and between arrays of wire or enumeration values. All these operations deliver boolean results.

There are basically two numeric operations defined in NODEN, + and −. Both are defined between unconstrained integers and deliver unconstrained integer results. It should be remembered that all unconstrained integers are defined to be positive, but the − operator may lead to a negative value being delivered. This is usually an error and will raise

an exception in the analyser to say the analysis has failed. The exception detection and handling mechanism is discussed in section 6.4.2.

NODEN supports three groups of mapping operators; arrays of booleans to/from unconstrained integers, constrained integers to/from unconstrained integers, and enumeration types to/from arrays of booleans. Arrays of booleans are mapped to integers with the **VAL***n* operators, that is **VAL4** maps an array of four booleans to an integer (in the range 0 to 15) etc. **WORD***n* operations provide the converse functions. It should be noted that **WORD***n* operations, like −, may raise an exception, if the value being mapped can ever be too large to be represented by *n* booleans (eg **WORD4 16**).

The other four classes of mapping functions are not automatically available, but have to be requested and named. If **F1** is to be a mapping function from an enumeration type *counter* to an array of two booleans, it is specified as:-

```
MAP F1 = counter -> [2]bool.
```

Mapping from constrained to unconstrained integers is always possible, and simply informs the NODEN compiler to change the type of a value. However, mapping from unconstrained to constrained integers involves checking that the unconstrained value is always in the constrained range, and raising an exception if it can ever fall outside that range.

Mapping between enumeration types and arrays of booleans is always possible (with no exceptions), and uses the representation defined for each member of the enumeration type when it was declared.

3.5 NODEN_HDL example

As an example of the use of NODEN, here is the specification of the behaviour of the SN74163 synchronous 4-bit counter [Texas]. The first four lines define an integer type with a range 0 to 15, a delay function for that type and mapping functions between that type and the unconstrained integer type **_integer**. The auxiliary function **INC4** then describes the effect of incrementing an integer of type *fourbit*. Note that the case of *v* having its maximum value, 15, is treated separately, so that the addition operation will never generate a value that causes an exception to be raised by **WORD_4**.

The circuit of interest is described by the BLOCK, SN74163. This defines the inputs into the circuit, the outputs from it, and the internal states (defined by the **LATCH4** function *count*). The next value of *count* is than defined in terms of the input values and the current state of *count*, and finally the values of the outputs are specified.

The implementation of this circuit would be defined in a similar manner (not shown here), with a function representing the counter being defined as a BLOCK with the same signature as SN74163. The contents of this block would represent the network of primitive gates used to implement the counter.

```
TYPE fourbit = INT [0..15].
```

```
DELAY LATCH4 = fourbit.

MAP  VALUE4 = fourbit -> _integer.
MAP  WORD_4 = _integer -> fourbit.

FN INC4 = (fourbit: v) -> fourbit:
   WORD_4( IF (VALUE4 v)==15 THEN 0 ELSE (VALUE4 v) + 1  FI ).

BLOCK SN74163 = (fourbit: datain,  bool: loadbar clearbar penb tenb) ->
                (fourbit: current, bool: ripple):
   BEGIN MAKE LATCH4: count.
         LET next_count =   IF clearbar == f  THEN WORD_4 0
                          ELIF loadbar  == f  THEN datain
                          ELIF penb AND tenb  THEN INC4 count
                                              ELSE count
                            FI.
         JOIN next_count -> count.
         OUTPUT (count, tenb AND ((VALUE4 count) == 15) )
     END.
```

4 The use of multi-valued logic

The function of the NODEN analyser is to find the way in which each of the outputs of a block depends upon the block's inputs. To this end, the next state of a memory element is regarded as an additional output, and its current state as a further input to the block. If the set of all possible input values to a block defines its input space, what is required for each output is an expression that indicates the set of states from the input space that causes the output to have some particular value. Before a detailed description of the operations performed by the NODEN analyser is undertaken, some consideration should be given to the effect of multi-valued logic on set manipulation.

A conventional boolean expression of three variables a, b and c, such as $d \Leftarrow a + \bar{b} \cdot c$ is interpreted as "d is true if a is true or if b is false and c is true".[2] This interpretation depends upon a number of conventions; firstly that unless otherwise indicated expressions are written to indicate when the value is true, and that the name of a variable means the conditions under which that variable is true, whilst the name of a variable with a bar over it means the conditions under which that variable is false. What is actually being described by this expression is the set of states, from the set of eight possible states of a b and c, for which d is true. This provides a complete definition of d, as being a boolean value, it can only ever have the states true or false, so any input state that is not a member of the set defined by the above expression must lead to d being false.

[2]Note that in writing an expression the effect of taking the union of two sets will be shown as $+$, whilst the union operator between expressions is written as $\cup$. Similarly the effect of the intersection operator is shown as $\cdot$ whilst the operator is $\cap$. As is conventional it will be assumed that intersection binds more tightly than union

Whilst the above convention is acceptable for purely boolean expressions there are two reasons why it is desirable to be able to reason about values with more than two states. Firstly, some signals commonly met in electronics naturally have more than two states. A *tristate* signal, as implied by its name, has three possible states; *high*, *low* and *z*. Secondly, it may be desirable to describe the behaviour of some device in terms of an abstract variable with a number of possible states, which will be implemented as a collection of simpler (possibly boolean) signals. For example the control input to an arithmetic unit may be said to have the states *add*, *subtract*, *shift* etc, whilst the implementation consists of a set of boolean signals, with a defined state representing each of the abstract control values. If e and f are tristate values, it is quite sensible to state that $f_{high} \Leftarrow e_{high} \cdot \bar{b} + e_z \cdot c$, to mean that "$f$ has the value *high* when e is *high* and b is false or when e has the value z and c is true". It should be noticed that this is not a complete description of f, as when the inputs are in a state which is not a member of the above set, it is clear that f must be *low* or z, but it is not known which. A second expression is needed to complete the description.

In general, a value with n possible states can be completely defined by $n-1$ expressions which describe the condition under which it is in $n-1$ of its possible states. It will be in its n^{th} state for any input state with is not covered by these expressions. In practice NODEN does not make use of this property, but defines all n states. This avoids the necessity of implementing a set difference operator to find when a value can be in the n^{th} state.

Therefore, a value v with n possible states can be defined by n expressions v_i ($i = 1$ *to* n), where v_i defines the set of inputs states which leads to v being in state i. Clearly for any given input state, v can only be in one particular state. It must therefore be the case that the intersection between any pair of expressions v_i and v_j is the empty set (0). Also all input states must define a result state, so the union of all of the n expressions must equal the set of all possible input states (1).

That is:- $i \neq j \Rightarrow (v_i \cap v_j) = 0$ and $\bigcup_{i=1}^{n} v_i = 1$

These properties can be used to check the correct operation of the NODEN analyser.[3]

5 Non-numeric analyser operations

The NODEN compiler generates a series of instructions to the analyser, which correspond to a collection of built in functions and operations. Whilst it is not the intention of the following two sections to describe the detailed formats of these instructions, the algebra performed by them is the basis for all the claims made for the NODEN verification suite and so will be described in detail.

This section considers those actions that can be performed on non-numeric (ie wire and enumeration) types, whilst section 6 considers numerics. Neither section will explicitly describe structures of values, as these can easily be constructed and manipulated as a series

[3] These properties only hold for non-numeric values, for numerics see section 6.1

of primitive non-numeric or numeric values. Hence the formation of structures '(-,-,-)' or '- CONC -' and the selection and slicing of values from structures '-[-]' or '-[-..-]' will *not be considered explicitly.*

The actions and operations to be described will be in terms of the set expressions that represent when a value is in a particular state. These operations are independent of the representation of the expressions. A number of possible representations will be discussed in section 7.

5.1 Representation of values

Wire and enumeration values are represented in the same manner. For a type with k members m_1 to m_k, a value is represented as a set of $k+2$ expressions. The first k of these indicate when the value is in states m_1 to m_k, whilst the remaining two indicate when the value has either been declared to be irrelevant (the don't care or X state) or when it is undefined (the U state). That is, a value a of k states is represented by a set of $k+2$ expressions: $V_{a=m_i}$ $(i = 1 \text{ to } k)$, X_a and U_a.

Boolean values (which have two states 't' and 'f') will be treated as a special case. Whilst these could be represented as a set of expressions:- $[V_{a=t},\ V_{a=f},\ X_a,\ U_a]$, these occur so frequently in the following descriptions that it is convenient to rename the first two expressions. A boolean value, a, is therefore represented as:- $[T_a,\ F_a,\ X_a,\ U_a]$.

When a constant value of a type with k members is required (including don't care and undefined), a structure of $k+2$ expressions is created, with the required member equal to the universal set (1) and all other expressions being the empty set (0). So for a boolean value, $[1, 0, 0, 0]$ is the representation of the constant 'true'; $[0, 0, 1, 0]$ is the boolean don't care etc.

5.2 Input values

An input value for a block is the value the NODEN analyser associates with a particular input signal. It should be noted that there is a subtle difference between the conceptual input and the analyser input value. If a is an input to the block under consideration and is of a type with k members, then the analyser value representing a, as previously discussed, will consist of $k+2$ expressions. However, the don't care and undefined states are properties of the analysis process, not of the input. An actual input signal must be in one of its k member states. The analyser value representing a must therefore define a set of expressions that show a in its k legal states but never in the don't care or undefined state.

For example, if a is a boolean, the analyser value associated with a would be:- $[a,\ \bar{a},\ 0,\ 0]$.

5.3 Boolean operations

If a is a boolean value (with the representation discussed above), then the inverse of a is:-

$NOT\ a \Rightarrow [F_a,\ T_a,\ 0,\ X_a \cup U_a]$

Note whenever a is either in its don't care or undefined state, the result is undefined. It is arguable that the inverse of the don't care state should be don't care, but it was felt that this was contrary to the intended meaning. It was intended that don't care should imply that under some set of conditions the value would not be used, but applying NOT to the value is using it, hence the result should be undefined rather than don't care. Whilst either definition of NOT is sensible, the one above is the one implemented by the analyser.

The basic boolean combinatorial operation is AND. This is defined as:-

$AND(a,b) \Rightarrow [T_a \cap T_b,\ F_a \cup F_b,\ 0,\ ((X_a \cup U_a) \cap (T_b \cup X_b \cup U_b)) \cup (T_a \cap (X_b \cup U_b))]$

That is, it is true whenever a and b are true, false whenever a or b is false, and is undefined under all other circumstances.

The NODEN analyser also supports $AND3$, $AND4$ and $AND8$, being three, four and eight input AND functions. These can be constructed from the above function in the expected manner.

The analyser supports $NAND$, $NAND3$, $NAND4$ and $NAND8$, being the above four functions followed by NOT.

The analyser supports the OR operator, where:-

$OR(a,b) \Rightarrow NOT((NOT\ a)\ AND\ (NOT\ b))$

As with AND, this is extended to provide $OR3$, $OR4$, $OR8$, NOR, $NOR3$, $NOR4$ and $NOR8$ operations.

The final boolean operations directly supported by the analyser is a select function, where:-

$SEL2(a,b,c) \Rightarrow (a\ AND\ b)\ OR\ ((NOT\ a)\ AND\ c) \equiv$ IF a THEN b ELSE c FI

5.4 Conditional statements

The basic conditional statement in NODEN_HDL has the form:-

```
CASE a OF
     m1: e1,
     m3: e3,
     m4: e4
  ELSE other
ESAC
```

If a is of a type with k members m_1 to m_k, then the CASE statement above can be regarded as an operator applied to a and k expressions E_{e1} to E_{ek}, such that whenever a has

the value m_1 the result E_{e1} is delivered, etc. Should the source text not supply a result expression for all the members of a, then the default result E_{other} is used. Thus in the above example, if a is of a type with five members, E_{other} is used as E_{e2} and E_{e5}. Should the 'ELSE -' clause be absent in the source text E_{other} is taken to be an undefined value of the appropriate type.

Each of the expressions E_{e1} to E_{ek} and E_{other} must be of the same type, and if this is a non-numeric type with l members, then the result expression R will also be of the same non-numeric type. R will be in a particular state whenever the result expression selected by a is in the same state, [4] that is:-

$$for\ i\ =\ 1\ to\ l;\quad V_{R=m_i}\ \Leftarrow\ \bigcup_{j=1}^{k} V_{a=m_j}\ \cap\ V_{E_j=m_i}$$

This also holds for the don't care state:-

$$X_R\ \Leftarrow\ \bigcup_{j=1}^{k} V_{a=m_j}\ \cap\ X_{E_j}$$

Finally, the result is in an undefined state whenever the selected expression is undefined or whenever a is itself undefined or don't care:-

$$U_R\ \Leftarrow\ X_a\ \cup\ U_a\ \cup\ \bigcup_{j=1}^{k} V_{a=m_j}\ \cap\ U_{E_j}$$

If the result expression is a structure of primitive values, then the effect of the CASE statement can be determined by applying the above operator to each of the primitive values in turn. The effect of the result or selector expressions being a numeric value is discussed in section 6.5.

The NODEN_HDL language also contains a boolean conditional statement:-

```
IF •1 THEN •2 ELSE •3 FI
```

This is simply a special case of the CASE statement described above, and is analysed as:-

```
CASE •1 OF t: •2 ELSE •3 ESAC
```

Similarly the following two nested conditionals are equivalent:-

```
IF •1 THEN •2 ELIF •3 THEN •4 ELSE •3 FI
```

[4] The expression $V_{a=m_j}$ represents the conditions under which a particular limb of the case statement is delivered. These are the conditions used to determine if any errors in the evaluation of the limb are to be reported. See sections 6.4.2, 6.7 and 6.6

```
and   CASE •1 OF
          t: •2
        ELSE CASE •3 OF
                 t: •4
               ELSE •5
             ESAC
      ESAC
```

5.5 Equality operator

For all wire and enumeration types, NODEN accepts an equality operator '=='. The result of applying this operator between two values a and b (which are of the same type, and have k members), is a boolean value. The result R is true whenever a and b are in the same legal states (m_1 to m_k), false whenever they are in different legal states, and undefined whenever a or b are undefined or don't care. That is:-

$$T_R \Leftarrow \bigcup_{i=1}^{k} V_{a=m_i} \cap V_{b=m_i}$$

$$F_R \Leftarrow \bigcup_{i=1}^{k} \bigcup_{j=1}^{k} V_{a=m_i} \cap V_{b=m_j} \quad (i \neq j)$$

$$X_R \Leftarrow 0$$

$$U_R \Leftarrow X_a \cup U_a \cup X_b \cup U_b$$

The inequality operator is defined as the inverse of the above, ie:-

$$a/=b \equiv NOT(a == b)$$

An equality operator is also defined between arrays of values of the same type. If a and b are arrays of values of length n, so that the members of the array are $a[1]$ to $a[n]$ and $b[1]$ to $b[n]$, then $a == b$ is defined as:-

$$(a[1] == b[1])\ AND\ (a[2] == b[2])\ AND\ ---AND\ (a[n] == b[n])$$

Again the inequality operator is simply NOT applied to the above

5.6 Mapping enumeration types to booleans

For all enumeration types there exists an operator to map a value of that type onto an array of booleans (with the representation defined or implied in the source text). If a is of an enumeration type with k members (m_1 to m_k), and a mapping is defined from this type to an array of n booleans, then internally a set of representation functions exist, REP. Where $REP(m_i, j)$ delivers true (t) if the j^{th} bit of member m_i is defined to be true; delivers false (f) if the j^{th} bit m_i is defined to be false; and delivers a boolean don't care (*bool*) if the j^{th} bit m_i is defined to be irrelevant. If the boolean equivalent of a is a

set of n boolean values R_1 to R_n, then for $j = 1$ *to* n:-

$$
\begin{aligned}
R_j \Leftarrow\ & CASE\ a\ OF \\
& \quad m_1 : REP(m_1, j), \\
& \quad\quad \vdots \\
& \quad m_k : REP(m_k, j) \\
& ESAC
\end{aligned}
$$

There is however one difference between this effective CASE statement and the normal NODEN_HDL CASE described in section 5.4, whenever the selector a is irrelevant, the result is also irrelevant (and not undefined).

For example, if a type is defined to have three states mapped onto two booleans as follows:-

```
TYPE control = NEW [2]bool (reset = #1x | load = #00 | inc = #01)
```

Then the two boolean values equivalent to a value a of the type *control* are:-

$$
\begin{aligned}
R_1 &\Leftarrow [V_{a=reset},\ V_{a=load} \cup V_{a=inc},\ X_a,\ U_a] \\
R_2 &\Leftarrow [V_{a=inc},\ V_{a=load},\ V_{a=reset} \cup X_a,\ U_a]
\end{aligned}
$$

5.7 Mapping booleans to enumeration types

There is a mapping corresponding to the inverse of the above, mapping an array of booleans onto an enumeration type. This is effectively defined as a conditional statement of the form:-

```
  IF em1 THEN m1
ELIF em2 THEN m2
       :
ELIF emk THEN mk
  FI
```

Where each of the k expressions em_i is a series of AND operations applied between members of the array of booleans R_1 to R_n. For a result equal to m_i; if $REP(m_i, j)$ delivers t, then the value R_j is present in the expression em_i; if $REP(m_i, j)$ delivers f, then the inverse of R_j is present in em_i and if $REP(m_i, j)$ delivers *bool*, then R_j is not present in em_i. That is for the previous example, if $r1$ and $r2$ are the pair of boolean values to be mapped onto an enumeration type *control*, the mapping function is equivalent to:-

```
  IF      r1               THEN reset
ELIF (NOT r1) AND (NOT r2) THEN load
ELIF (NOT r1) AND      r2  THEN inc
  FI
```

or

$$[T_{R_1},\ F_{R_1} \cap F_{R_2},\ F_{R_1} \cap T_{R_2},\ 0,\ X_{R_1} \cup U_{R_1} \cup (F_{R_1} \cap (X_{R_2} \cup U_{R_2}))]$$

Note that this function is not strictly the converse of the enumeration to boolean function described in the previous section, as the don't care enumeration type maps onto an array of don't care booleans, but any boolean being don't care maps onto the enumeration type undefined.

6 Numeric analyser operations

6.1 Representation of values

NODEN is concerned with two classes of numeric values; constrained integers with a defined range (*lwb* to *upb*), and unconstrained integers which in principle can have any value from zero to infinity. It should be noted that all numeric values are strictly non-negative and both constrained and unconstrained integers have the same representation.

Numeric inputs and outputs of a block are always constrained integers, because these must be represented in the physical implementation by a finite set of signals, and so must have a finite number of states. Inside a block, all operations are performed on unconstrained integers. If not, each numeric type would require its own set of arithmetic operators, and the expected type of an arithmetic operation would quickly become ambiguous in a description with several numeric types. As input and output numerics are always constrained but all operators act on unconstrained values, it would be expected that mapping functions are required to turn constrained integers into unconstrained, and vice versa. These are described in section 6.6.

As all unconstrained integers are always derived by arithmetic operations from constrained integers or constants, they in fact always have a known finite range. Therefore any integer a has a maximum value $maxval_a$. If n_a is the number of bits needed to represent $maxval_a$ in binary form (ie $n_a = \top log_2(maxval + 1)$),[5][6] then a numeric representation consists of $2 \times n_a$ expressions defining when the n_a bits of the binary representation are in their true and false states ($BIT_{a=t_i}$ and $BIT_{a=f_i}$, for $i = 1$ *to* n_a). Like non-numeric values, numerics may also be defined to be irrelevant (don't care or X_a) or may be undefined (U_a). So an integer a with a maximum value of 2 (or 3) is represented as two bits, don't care and undefined:-

$$[BIT_{a=t_1},\ BIT_{a=f_1},\ BIT_{a=t_2},\ BIT_{a=f_2},\ X_a,\ U_a]$$

Each bit $BIT_{a=t_i}$ and $BIT_{a=f_i}$, together with X_a and U_a, effectively forms a boolean value. So it must be the case that, for $i = 1$ *to* n_a:-

$$BIT_{a=t_i} \cap BIT_{a=f_i} = 0 \text{ and } BIT_{a=t_i} \cap X_a = 0 \text{ and } BIT_{a=t_i} \cap U_a = 0 \text{ and}$$
$$BIT_{a=f_i} \cap X_a = 0 \quad \text{and } BIT_{a=f_i} \cap U_a = 0 \text{ and } \quad X_a \cap U_a = 0$$

also

[5] Note that $\top a$ is the first integer greater than or equal to a

[6] The integer literal 0 is treated as a special case, with $n_0 = 1$ rather than 0

$$BIT_{a=t_i} \cup BIT_{a=f_i} \cup X_a \cup U_a = 1$$

These properties can be used to check the operation of the analyser, in the same way the use of the equivalent properties for non-numerics can be used, as described in section 4. In addition, for any pair of bits i and j:-

$$BIT_{a=t_i} \cup BIT_{a=f_i} = BIT_{a=t_j} \cup BIT_{a=f_j}$$

Literal numeric values are therefore represented by a binary value, with any non-significant leading 0's removed. Hence 4 is represented as [0, 1, 0, 1, 1, 0, 0, 0]. All numeric values are represented with at least one bit, so the don't care numeric value is:- [0, 0, 1, 0].

6.2 Input values

If v is a numeric input to a block, with a range 0 to upb, then the analyser will represent this as a set of n bits, where $n = \top log_2(upb + 1)$. If $upb = 2^n - 1$, then the mapping between these bits and v is obvious. For example if $upb = 7$, then v is mapped onto a set of three bits (say a, b, c), such that v is represented as:-

$[a,\ \bar{a},\ b,\ \bar{b},\ c,\ \bar{c},\ 0,\ 0]$

If $upb \neq 2^n - 1$, then the first $upb - 1$ states of v will be mapped onto the equivalent states of the bits, whilst the final state of v will map onto the remaining states of the bits. So if $upb = 5$, then this value will also be represented by three bits, such that:-

$0 \equiv \bar{a} \cdot \bar{b} \cdot \bar{c}$ and $1 \equiv a \cdot \bar{b} \cdot \bar{c}$ etc to $4 \equiv \bar{a} \cdot \bar{b} \cdot c$ and $5 \equiv a \cdot \bar{b} \cdot c + \bar{a} \cdot b \cdot c\ + a \cdot b \cdot c$

Hence the value of v is represented as:-

$[a + b \cdot c,\ \bar{a} \cdot \bar{c} + \bar{a} \cdot \bar{b},\ b \cdot \bar{c},\ \bar{b} + c,\ c,\ \bar{c},\ 0,\ 0]$

If the range of v is not 0 to upb, but lwb to upb, then the input value is derived by making a value for an input in the range 0 to $upb - lwb$ as described above, and adding a literal value lwb to it (see section 6.3). So the value of an input with a range 1 to 6 (mapped onto three bits) would be:-

$[\bar{a} \cdot \bar{c} + \bar{a} \cdot \bar{b},\ a + b \cdot c,\ \bar{a} \cdot b + a \cdot \bar{b} + a \cdot c,\ \bar{a} \cdot \bar{b} + a \cdot b \cdot \bar{c},\ c + a \cdot b,\ \bar{a} \cdot \bar{c} + \bar{b} \cdot \bar{c},\ 0,\ 0]$

6.3 The addition operation

To find the sum of two numeric values, a and b, a binary ripple add algorithm is used. This algorithm assumes that both numeric values have the same number of significant bits n. If this is not the case, the missing bits of the shorter value are constructed to be never true and false whenever any one of the defined bits (say bit 1) is either true or false. Note that these constructed bits are not always false, as the union of the conditions under which they are true, false, don't care and undefined must be 1, and none of these four conditions may overlap. So if a has fewer bits than b, each missing bit j is constructed as:- $BIT_{a=t_j} \Leftarrow 0$

and $BIT_{a=f_i} \Leftarrow BIT_{a=t_1} \cup BIT_{a=f_1}$

Each bit of the result, r, is constructed as the sum of the corresponding bits from a and b and a ripple carry from the previous bit. The carry out from each bit is also evaluated. That is for $i = 1$ *to* n:-

$$BIT_{r=t_i} \Leftarrow (BIT_{a=t_i} \cap BIT_{b=f_i} \cap CARRY_{f_{i-1}}) \cup (BIT_{a=f_i} \cap BIT_{b=t_i} \cap CARRY_{f_{i-1}}) \cup (BIT_{a=f_i} \cap BIT_{b=f_i} \cap CARRY_{t_{i-1}}) \cup (BIT_{a=t_i} \cap BIT_{b=t_i} \cap CARRY_{t_{i-1}})$$

$$BIT_{r=f_i} \Leftarrow (BIT_{a=f_i} \cap BIT_{b=f_i} \cap CARRY_{f_{i-1}}) \cup (BIT_{a=t_i} \cap BIT_{b=t_i} \cap CARRY_{f_{i-1}}) \cup (BIT_{a=t_i} \cap BIT_{b=f_i} \cap CARRY_{t_{i-1}}) \cup (BIT_{a=f_i} \cap BIT_{b=t_i} \cap CARRY_{t_{i-1}})$$

$$CARRY_{t_i} \Leftarrow (BIT_{a=t_i} \cap BIT_{b=t_i} \cap CARRY_{f_{i-1}}) \cup (BIT_{a=t_i} \cap CARRY_{t_{i-1}}) \cup (BIT_{b=t_i} \cap CARRY_{t_{i-1}})$$

$$CARRY_{f_i} \Leftarrow (BIT_{a=f_i} \cap CARRY_{f_{i-1}}) \cup (BIT_{b=f_i} \cap CARRY_{f_{i-1}}) \cup (BIT_{a=f_i} \cap BIT_{b=f_i} \cap CARRY_{t_{i-1}})$$

where $CARRY_{t_0} = 0$ and $CARRY_{f_0} = (BIT_{a=t_1} \cup BIT_{a=f_1}) \cap (BIT_{b=t_1} \cup BIT_{b=f_1})$

Note that $CARRY_{f_0} \neq 1$, as this may lead to sets of expressions that violate the rules already expressed for the union and intersection of bits of a value, but is false only under those conditions for which a and b both have defined values.

If the carry out from the n^{th} bit can ever be true ($CARRY_{t_n} \neq 0$), then the result has one more significant bit than the operands, such that:-

$BIT_{r=t_{n+1}} \Leftarrow CARRY_{t_n}$ and $BIT_{r=f_{n+1}} \Leftarrow CARRY_{f_n}$

The result can never be irrelevant ($X_r \Leftarrow 0$) and is undefined whenever either of the operands is undefined or don't care:-

$U_r \Leftarrow X_a \cup U_a \cup X_b \cup U_b$

6.4 Subtraction and comparison operators

6.4.1 General difference operator

The difference between two numeric values a and b is found in a similar manner to the above addition operator. However, it is used for two distinct purposes, firstly to evaluate $a - b$, and secondly to compare two numeric values as in $a > b$. for this reason, the general difference algorithm will be described first, and then the following two sections will consider its use.

The difference algorithm used is the well known method of adding the inverse of the second operand to the first operand, with the carry into the first bit equal to 1. In order to distinguish between the sum and difference algorithms, the difference operator will be

described as delivering n DIFF values (rather than BIT's) and the carry expressions will be replaced by their inverse, BORROW. As with addition, n is the number of significant bits in the longer of the operands, and the shorter operand is padded to the same length with values calculated in the same manner.

The difference operator is evaluated as, for $i = 1$ *to* n:-

$$DIFF_{t_i} \Leftarrow (BIT_{a=t_i} \cap BIT_{b=t_i} \cap BORROW_{t_{i-1}}) \cup (BIT_{a=f_i} \cap BIT_{b=f_i} \cap BORROW_{t_{i-1}}) \cup (BIT_{a=f_i} \cap BIT_{b=t_i} \cap BORROW_{f_{i-1}}) \cup (BIT_{a=t_i} \cap BIT_{b=f_i} \cap BORROW_{f_{i-1}})$$

$$DIFF_{f_i} \Leftarrow (BIT_{a=f_i} \cap BIT_{b=t_i} \cap BORROW_{t_{i-1}}) \cup (BIT_{a=t_i} \cap BIT_{b=f_i} \cap BORROW_{t_{i-1}}) \cup (BIT_{a=t_i} \cap BIT_{b=t_i} \cap BORROW_{f_{i-1}}) \cup (BIT_{a=f_i} \cap BIT_{b=f_i} \cap BORROW_{f_{i-1}})$$

$$BORROW_{f_i} \Leftarrow (BIT_{a=t_i} \cap BIT_{b=f_i} \cap BORROW_{t_{i-1}}) \cup (BIT_{a=t_i} \cap BORROW_{f_{i-1}}) \cup (BIT_{b=f_i} \cap BORROW_{f_{i-1}})$$

$$BORROW_{t_i} \Leftarrow (BIT_{a=f_i} \cap BORROW_{t_{i-1}}) \cup (BIT_{b=t_i} \cap BORROW_{t_{i-1}}) \cup (BIT_{a=f_i} \cap BIT_{b=t_i} \cap BORROW_{f_{i-1}})$$

where $BORROW_{t_0} = 0$ and $BORROW_{f_0} = (BIT_{a=t_1} \cup BIT_{a=f_1}) \cap (BIT_{b=t_1} \cup BIT_{b=f_1})$

6.4.2 The subtraction operator

Evaluating $a - b$ is rather different to evaluating the sum of two values, as this is the first operation discussed which has an exception condition. As all NODEN numeric values are defined to be strictly non-negative, if a can ever be less than b, there is a set of results which cannot be represented by a numeric value. This will normally lead to the analyser reporting an error (as will be described later). In any event, the result of the subtraction is undefined for negative values (ie when $BORROW_{t_n} \neq 0$). Conversely, the general difference operator delivers the correct values for the result bits under the conditions indicated by $BORROW_{f_n}$.

Hence, if $r = a - b$, then the BIT values of r are, for $i = 1$ *to* n:-

$$BIT_{r=t_i} \Leftarrow BORROW_{f_n} \cap DIFF_{t_i} \text{ and } BIT_{r=f_i} \Leftarrow BORROW_{f_n} \cap DIFF_{f_i}$$

The result is never irrelevant ($X_r \Leftarrow 0$) and is undefined whenever a or b are undefined or irrelevant, or when the result would be negative:-

$$U_r \Leftarrow X_a \cup U_a \cup X_b \cup U_b \cup BORROW_{t_n}$$

It was said earlier that if the result of a subtraction can ever be negative then usually an error is reported. Precisely what happens depends upon where in the source text the subtraction being evaluated occurred. If a is of a numeric type with lower bound 0, and the NODEN_HDL source contains the statement:-

```
LET a_minus_1 = a - 1.
```

An error will be reported by the analyser to the effect that analysis of the current block has failed whilst evaluating a subtraction on a particular line of the source text, under the condition for which $a = 0$. However, if the subtraction is in a conditional statement, as in:-

```
LET a_limit = IF a==0 THEN 0 ELSE a - 1 FI.
```

then no error will be reported. This is because the analyser evaluates an expression which indicates the conditions under which the result limb of a conditional statement will be delivered. If the result limb can raise an exception, it checks whether the exceptional result could ever occur before indicating an error. Hence in the above example, the subtraction is performed under the conditions when $a \neq 0$, so the exception when $a = 0$ can never be raised, hence no error is reported.

There are a set of unusual circumstances that may lead to an error not being reported immediately. If the source text contains a statement such as:-

```
LET a_dec = IF a /= 0 THEN a - 1 ELSE a - 1 FI.
```

no error is reported because the analyser recognises that $a - 1$ need only be evaluated once. As the first occurrence is under the circumstances when $a \neq 0$, no error is reported. However this does not lead to an erroneous result as any output from the block that uses *a_dec* will be shown as being undefined when $a = 0$. This will lead to an error being reported if the source text is a specification (as all specification outputs must be defined for all input values), or if the description is an implementation, a mismatch will be reported when the implementation is compared with the specification.

6.4.3 Comparison operators

NODEN supports six relational operations between numeric values (>, >=, <, <=, == and / =). The two principle operations are < and ==. Both are evaluated using the difference operation previously described. $a < b$ is true whenever $a - b$ is negative (ie in the conditions indicated by $BORROW_{t_n}$), false when $a - b$ is positive (ie in the conditions indicated by $BORROW_{f_n}$), never irrelevant and undefined whenever either of the operands are irrelevant or undefined. That is:-

$$a < b \Leftarrow [BORROW_{t_n},\ BORROW_{f_n},\ 0,\ X_a \cup U_a \cup X_b \cup U_b]$$

a and b are equal whenever all the DIFF bits are false. Hence the equality operator is defined as:-

$$a == b \Leftarrow [\bigcap_{i=1}^{n} DIFF_{f_i},\ \bigcup_{i=1}^{n} DIFF_{t_i},\ 0,\ X_a \cup U_a \cup X_b \cup U_b]$$

The other comparisons are then constructed as:-

```
a /= b   is equivalent to   NOT (a == b)
a <= b   is equivalent to       (a <  b) OR (a == b)
a >  b   is equivalent to   NOT((a <  b) OR (a == b))
a >= b   is equivalent to   NOT (a <  b)
```

6.5 Conditional statements

6.5.1 Non-numeric selector

Numerics may occur in CASE statements in three ways; as the result of a CASE with a non-numeric selector, or in a CASE statement with a numeric selector and either a numeric or non-numeric result.

The behaviour of a CASE statements with a non-numeric selector and result has been described in section 5.4. The behaviour of the CASE statement with a non-numeric selector and a numeric result is very similar. So for a selector a of a type with k members (m_1 to m_k) and result expressions E_1 to E_k that are numerics with n significant bits,[7] then the bits of the numeric result r are, for $i = 1$ *to* n:-

$$BIT_{r=t_i} \Leftarrow \bigcup_{j=1}^{k} V_{a=m_j} \cap BIT_{E_j=t_i}$$

$$BIT_{r=f_i} \Leftarrow \bigcup_{j=1}^{k} V_{a=m_j} \cap BIT_{E_j=f_i}$$

also:-

$$X_r \Leftarrow \bigcup_{j=1}^{k} V_{a=m_j} \cap X_{E_j}$$

$$U_r \Leftarrow X_a \cup U_a \cup \bigcup_{j=1}^{k} V_{a=m_j} \cap U_{E_j}$$

6.5.2 Numeric selector

If the selector a is a numeric value with n significant digits, then the basic form of the CASE statement is:-

```
CASE a OF
     1: e1,
     3: e3,
     4: e4
  ELSE other
ESAC
```

The CASE statement can be regarded as an operator acting on a and 2^n result expressions, E_0 to E_{2^n-1}, such that when a has the value 0, E_0 is delivered etc. As in the non-numeric CASE selector, any missing expressions are replaced by E_{other}, and if E_{other} is not defined an undefined value of the appropriate type is delivered. It should be noted that any selector i, where $i > 2_n - 1$ can never be selected, and takes no part in the result.

[7] where n is the length of the result limb with the most significant digits. Any shorter numeric result limbs are padded to the same length as described in section 6.3

For $i <= 2_n - 1$, a selection value $V_{a=i}$ is defined as the set of conditions for which a has the value i. That is:-

$$V_{a=i} \Leftarrow \bigcap_{j=1}^{n} BIT_{a=?_j}$$

where $BIT_{a=?_j} = BIT_{a=t_j}$ if bit j of i is true, or $BIT_{a=f_j}$ if bit j of i is false (lsb = 1). This is the value that is used to test if exceptions are to be reported.

So for a non-numeric result, with l members:-

$$\begin{aligned} for\ j = 1\ to\ l;\quad V_{r=m_j} &\Leftarrow \bigcup_{i=0}^{2^n-1} V_{a=i} \cap V_{E_i=m_j} \\ X_r &\Leftarrow \bigcup_{i=0}^{2^n-1} V_{a=i} \cap X_{E_i} \\ U_r &\Leftarrow X_a \cup U_a \cup \bigcup_{i=0}^{2^n-1} V_{a=i} \cap U_{E_i} \end{aligned}$$

and for a numeric result, with m significant digits:-

$$\begin{aligned} for\ j = 1\ to\ m;\quad BIT_{r=t_j} &\Leftarrow \bigcup_{i=0}^{2^n-1} V_{a=i} \cap BIT_{E_i=t_j} \\ BIT_{r=f_j} &\Leftarrow \bigcup_{i=0}^{2^n-1} V_{a=i} \cap BIT_{E_i=f_j} \\ X_r &\Leftarrow \bigcup_{i=0}^{2^n-1} V_{a=i} \cap X_{E_i} \\ U_r &\Leftarrow X_a \cup U_a \cup \bigcup_{i=0}^{2^n-1} V_{a=i} \cap U_{E_i} \end{aligned}$$

6.6 Mapping between constrained and unconstrained integers

In NODEN_HDL, mapping functions can be defined to convert a constrained numeric value into the equivalent unconstrained value, and vice versa. The first of these is used solely by the NODEN_HDL compiler to get its type model correct and has no effect on the value as represented in the analyser. On the other hand, mapping from a value which is of the unconstrained numeric type to the equivalent value in a constrained numeric type, does involve some evaluation by the analyser, as there is the possibility of an exception being raised if the value being converted can ever lie outside the range of the constrained type.

If F1 is a function defined to map between unconstrained numeric values and values of a type in the range *lwb* to *upb*, then the NODEN_HDL statement:-

```
LET r = F1 a.
```

is effectively evaluated as:-

```
LET temp  =  IF a >= lwb THEN a    FI.
LET    r  =  IF a <= upb THEN temp FI.
```

The effect of the above is to make the result undefined for all values of a that lie outside the expected range of the result (ie $a < lwb$ or $a > upb$). As with subtraction, a ever being outside the expected range is regarded as an exceptional condition, and will normally lead to an error being reported (unless the exception is raised inside a conditional that prevents it).

6.7 Mapping between integers and booleans

NODEN_HDL defines a set of functions VAL_n and $WORD_n$ that map from an array of n booleans to an unconstrained integer, and from an unconstrained integer to an array of n booleans respectively.

If $r = VAL_n\ a$, where a is an array of n booleans (with members $a[1]$ to $a[n]$) and r is an unconstrained integer, then r is undefined when any member of a is either irrelevant or undefined. That is, if E_{legal} represents the set of conditions for which r is properly defined, then for $i = 1$ *to* n:-

$$BIT_{r=t_i} \Leftarrow E_{legal} \cap T_{a[i]} \text{ and } BIT_{r=f_i} \Leftarrow E_{legal} \cap F_{a[i]}$$

also

$$X_r \Leftarrow 0$$
$$U_r \Leftarrow \bigcup_{i=1}^{n} X_{a[i]} \cup U_{a[i]}$$

where

$$E_{legal} \Leftarrow \bigcap_{i=1}^{n} T_{a[i]} \cup F_{a[i]}$$

If $r = WORD_n\ a$, where a is an unconstrained integer (with m significant bits) and r is an array of n booleans (with members $r[1]$ to $r[n]$), then if $n >= m$ then the function is unexceptional and the n booleans are evaluated as, for $i = 1$ *to* n:-

$$r[i] \Leftarrow [BIT_{a=t_i},\ BIT_{a=f_i},\ X_a,\ U_a]$$

As in addition and subtraction, if n is greater than m, any 'missing' bits are constructed to be never true, and false under the condition under which the other bit pairs are defined (ie $BIT_{a=t_i} \cup BIT_{a=f_i}$).

If m is greater than n, then the value of a can be greater than the maximum value that can be represented by n bits under all circumstances for which any of the bits of a greater than n are true. The values of r are therefore constructed to be undefined under these circumstances. So for $i = 1$ *to* n:-

$$r[i] \Leftarrow [BIT_{a=t_i} \cap \bigcap_{j=n+1}^{m} BIT_{a=f_j},\ BIT_{a=f_i} \cap \bigcap_{j=n+1}^{m} BIT_{a=f_j},\ X_a,\ U_a \cup \bigcup_{j=n+1}^{m} BIT_{a=t_j}]$$

Should there be any values of a that cannot be represented in n bits an exception will be raised (as previously described). This is frequently met when modelling n bit counters. If *count* is the state of a four bit counter, its next state can legally be defined as:-

```
LET next_state =  IF (VAL4 count) == 15
                    THEN WORD4 0
                    ELSE WORD4((VAL4 count) + 1)
                  FI.
```

whilst

```
LET next_state =  WORD4((VAL4 count) + 1).
```

would be lead to the analysis failing and an exception being raised.

7 Data representation and processing

The usefulness of the NODEN analyser depends upon the complexity of descriptions that it can analyse (given finite machine resources) and the time taken to perform that analysis. These properties in turn depend upon the representations used to store the expression data, and the algorithms used to manipulate them. As will be shown there is a trade-off between representations that efficiently use memory but which are slow to process, and those that are fast but occupy more space. Three analysers have been written, two using a disjunctive normal form representation and the other using a modified Shannon form. The performance of these analysers on a practical application will be discussed in section 8. The following sections will discuss the representations that have been used in the analysers, and the Shannon factorised form representation (on which the modified Shannon form was based).

7.1 Disjunctive Normal Form

Each expression in this representation is written as a list of terms, where each term is the intersection of a number of variables (or their inverse). That is, each term represents a set of states from the input space and the expression is the union of these sets. For example, the expression :- $a + \bar{b} \cdot c$ consists of the union of two terms, one covering all states in the input space for which a is true, and the other covering all states for which b is false and c true.

For an expression of n boolean variables, each term can be represented by a pair of boolean sets A_i and S_i (i = 1 to n). $A_i = 0$ means that variable i has no effect on the term, whilst $A_i = 1$, $S_i = 0$ means that the inverse of variable i is part of the term, and $A_i = 1$, $S_i = 1$ means that variable i is part of the term. The two terms in the above expression would therefore be represented as $a \Rightarrow A(1,0,0), S(1,0,0)$ and $\bar{b} \cdot c \Rightarrow A(0,1,1), S(0,0,1)$. That is A is an Activity mask and S indicates the Sense of active bits.

The two expression operations used by the NODEN analyser are union and intersection. The union of two disjunctive normal form expressions can be found by concatenating the two lists of terms that form the two expressions, to form a single list representing the result. For intersection the following algorithm is used. Each term from the expression E' (represented by A', S') is compared with each term from the second expression E'' (A'', S''). That is, if E' has t' terms and E'' has t'' terms, there are potentially $t' \times t''$ terms in the result. For each pair of terms under consideration, a term will be added to the result if any active variables in common between the terms are in the same sense. For example, $a \cdot b \cap \bar{b} \cdot c$ will not add anything to the result, whilst $a \cdot \bar{b} \cap \bar{b} \cdot c$ will add a term $a \cdot \bar{b} \cdot c$. That is a pair of terms will add to the result if for $i = 1$ *to* $n \Rightarrow A'_i \wedge A''_i \wedge (S'_i \neq S''_i) = 0$. [8] If required, the term to be added to the result (A^r, S^r) is $A^r \Leftarrow A' \vee A''$, $S^r \Leftarrow A^r \wedge (S' \vee S'')$.

Unfortunately this representation is not canonical, and it can easily be seen how the algorithms described above could lead to multiple copies of the same term appearing in the representation of an expression. Whilst this would not affect the correctness of the result, it would cause the representation to be wasteful of memory, and more importantly as the time to perform the intersection operation is proportional to the product of the number of terms in the expressions, redundant terms can seriously degrade the processing time. This means that the algorithms described above must be followed by a simplification routine to remove redundant terms. It is also desirable that the simplifier should recognise simplifications such as:- $a \cdot \bar{b} + a \cdot b \equiv a$. The simplification algorithm is in three parts:-

1. If all the active variables in one term are also active in the corresponding sense in a second term, then the second term is either the same or more specific than the first and so can be eliminated. For example:- $a \cdot \bar{c} + a \cdot \bar{c} \equiv a \cdot \bar{c}$ and $a \cdot \bar{c} + a \cdot b \cdot \bar{c} \equiv a \cdot \bar{c}$.

2. If the same variables are active in two terms and all but one of these variables is in the corresponding sense, then the pair of terms can be replaced by a single term with the 'mismatched' variable eliminated. For example:- $a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} \equiv a \cdot \bar{c}$.

3. If the variables which are active in common between two terms are in corresponding senses, and if one term has a single additional variable active, then the second term can be made more specific by including the inverse of the additional variable in the first term. For example:- $a \cdot b \cdot c + a \cdot b \cdot d \cdot e \equiv a \cdot b \cdot c + a \cdot b \cdot \bar{c} \cdot d \cdot e$. This helps eliminate terms which are included in the union of several other terms. For example:- $\bar{a} \cdot b + b \cdot c + a \cdot c \equiv \bar{a} \cdot b + a \cdot b \cdot c + a \cdot c \equiv \bar{a} \cdot b + a \cdot c$.

The advantage of this representation is that the boolean sets A and S for each term can be represented efficiently by packing 32-bits into each 32-bit machine word and then using word-wide AND and OR operations to evaluate the intersection operation. However, the simplification process described above requires the behaviour of individual bits of the sets A and S to be examined and manipulated, and this is not such an efficient process. As will be seen in section 8 the analyser built using this representation uses very little data space, but takes a considerable time to evaluate complex functions.

[8] Note that the operator $\wedge$ is *AND* (or intersection) between boolean values (or sets of boolean values) and $\vee$ is *OR* (or union)

Although the above simplification algorithm will remove most redundant terms, it is not perfect and an expression such as:- $\bar{a} \cdot \bar{c} + \bar{b} \cdot c + a \cdot b + a \cdot \bar{b} \cdot \bar{c} + \bar{a} \cdot b \cdot c$ will be regarded as irreducable, even though it is identically true. This problem becomes more probable as the number of variables increases, and is of particular significance if the test that 'the union of all expressions which define a result should be true' is performed. This leads to the need for a further algorithm to prove that an expression such as the above is indeed true. This is done by constructing two expressions; one including all the terms from the original in which a selected variable is true or absent (that variable being eliminated from all the terms), and the other including those terms in which the selected variable is false or absent. These two expressions should then themselves be universally true, if the original was universally true. If the simplification algorithm already described cannot reduce these expressions to true, then they can themselves be split into two expressions by selecting a second variable. If this process is repeated for all variables, and there are still some expressions which are not true, then the original was not true. In the above example, if a is selected as the first variable to be eliminated, then this leads to two expressions:-
a true or absent $\Rightarrow \bar{b} \cdot c + b + \bar{b} \cdot \bar{c} \Rightarrow 1$ and a false or absent $\Rightarrow \bar{c} + \bar{b} \cdot c + b \cdot c \Rightarrow 1$

A second disadvantage is that all the variables described so far have been boolean, whereas NODEN needs to process multi-valued logic variables. In order to reason about such non-boolean values a mapping is needed between the actual variable and a set of booleans. For a variable with m possible states, this can be mapped onto a set of s booleans, where $s = \top \log_2 m$. These can be regarded as representing states 0 to $2^s - 1$. The first $m - 1$ states of the variable are mapped onto states 0 to $m - 1$ of the booleans, whilst the final state can be mapped onto states m to $2^s - 1$. So for a variable of six states, this could be mapped onto a set of three booleans (a, b, c). The first five states being represented as; $\bar{a} \cdot \bar{b} \cdot \bar{c}$, $\bar{a} \cdot \bar{b} \cdot c$, $\bar{a} \cdot b \cdot \bar{c}$, $\bar{a} \cdot b \cdot c$ *and* $a \cdot \bar{b} \cdot \bar{c}$; whilst the final state is $a \cdot b + a \cdot \bar{b} \cdot c$.

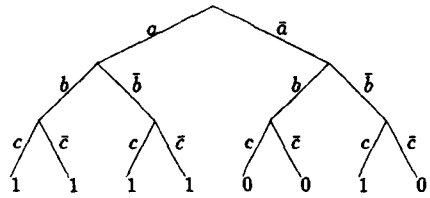
7.2 Shannon factorised form

Shannon factorised form [Bryant 86] regards the variables upon which an expression depends as having a fixed order. It then describes an expression of n boolean variables as a pair of expressions (each of $n - 1$ variables) with the first variable factored out. That is if expression E depends upon a, b and c (in that order), then E is expressed as $[E_a\ E_{\bar{a}}]$.[9] Where E_a represents the expression E with a factored out, and $E_{\bar{a}}$ represents the expression E with $\bar{a}$ factored out. E_a and $E_{\bar{a}}$ are themselves represented as a pair of expressions with b and $\bar{b}$ factored out, and the resulting four expressions (which depend solely on c are also factored into pairs of expressions, which are either universally true (1) or false (0). It should be noted that for any expression there is only one possible Shannon factorised form representation, ie the representation is canonical.

For example, the expression that is represented in disjunctive normal form as $a + \bar{b} \cdot c$, is first factorised to remove a and $\bar{a}$, $[1, \bar{b} \cdot c]$. These two expressions are then factorised to remove b and $\bar{b}$, leading to the expression pairs [1,1] and [0, c]. These are further factorised to remove c and $\bar{c}$ to give the Shannon factorised form:- [[[1, 1], [1, 1]], [[0, ^], [1, 0]]]. This might more easily be thought of as a tree as shown in Figure 1.

[9] That is $E \equiv a \cdot E_a + \bar{a} \cdot E_{\bar{a}}$

Figure 1: A Shannon factorised form 'tree'



This representation has a number of desirable properties. If E' and E'' are two expressions in Shannon factorised form, with values $[E'_a, E'_{\bar{a}}]$ and $[E''_a, E''_{\bar{a}}]$ respectively, then it is quite simple to show that $E' \cap E'' \equiv [E'_a \cap E''_a, E'_{\bar{a}} \cap E''_{\bar{a}}]$. Similarly $E' \cup E'' \equiv [E'_a \cup E''_a, E'_{\bar{a}} \cup E''_{\bar{a}}]$. The operators $\cap$ and $\cup$ applied to the 'leaf' values 0 and 1 have the expected results. This means that the union or intersection of two Shannon form expressions of n variables is always found by $2^{n+1} - 1$ simple operations, and that the result is always in a canonical form. Hence, unlike the disjunctive normal form representation, no simplification stage is required and the universal truth of an expression is easily determined. Therefore this representation can potentially be processed very quickly.[10]

There are however a number of disadvantages with this representation. Firstly, as with disjunctive normal form, there is no direct representation of multi-valued logic, although a mapping similar to that discussed previously can be implemented. The second and more serious problem is that each expression is formed of a fixed number of nodes ($2^n - 1$), each pointing to two further nodes. For efficient processing, these need to be implemented as some form of pointer, so it is likely that each expression will require $2 \times 2^n - 1$ words of memory. This is not a problem if n is small, but if n is large the representation may become very inefficient in its use of memory, particularly if not all variables are significant in a particular expression. In one of the blocks described in section 8, a set of expressions were evaluated where each could depend on upto 239 variables, although in fact no single expression actually depended upon more than 20. What has therefore been developed is a more memory efficient version of the Shannon factorised form, that maintains the desirable properties of being a canonical form and having fast union and intersection evaluation times, but which often uses far less memory.

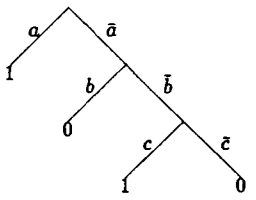
7.3 Modified Shannon form

The modified Shannon form representation depends upon the observation that if a is potentially the first variable in an expression who's Shannon factorised form is $[E_a, E_{\bar{a}}]$, then if the expression is actually independent of a, $E_a = E_{\bar{a}}$. A modified Shannon form expression can therefore be constructed from an indication of the first significant variable in the expression, and the two factored expressions with that variable removed (as in the Shannon factorised form). That is if b is the first significant variable in expression E, then

[10] This representation also has the property (not used by NODEN) that the inverse of E: $\neg E \equiv [\neg E_a, \neg E_{\bar{a}}]$. That is the inverse can be evaluated by inverting all the 'leaf' expressions

$E \equiv [b,\ E_b,\ E_{\bar{b}}]$.[11] These sub-expressions are also in modified Shannon form, or are the 'leaf' values 1 and 0.

For example, the expression represented in disjunctive normal form as $a + \bar{b} \cdot c$, can be represented in modified Shannon form as:- $[a,\ 1,\ [b,\ 0,\ [c,\ 1,\ 0]]]$ or graphically as:-



If any expression $[i,\ E_i,\ E_{\bar{i}}]$ in which $E_i = E_{\bar{i}}$ is replaced by E_i, it can be shown that this representation is canonical.

Given the expected definitions of union and intersection between 'leaf' expressions and other expressions (ie $0 \cup E \Rightarrow E$, $1 \cup E \Rightarrow 1$, $0 \cap E \Rightarrow 0$ and $1 \cap E \Rightarrow E$), then if E' and E'' are in modified Shannon form, with values $[i',\ E'_i,\ E'_{\bar{i}}]$ and $[i'',\ E''_i,\ E''_{\bar{i}}]$ respectively, then the union and intersection operations are defined as follows. If *op* is either $\cup$ or $\cap$, then $E'\ op\ E''$ is:-

$$\text{if } i' = i'' \Rightarrow [i',\ E'_i\ op\ E''_i,\ E'_{\bar{i}}\ op\ E''_{\bar{i}}]$$

$$\text{or } i' < i'' \Rightarrow [i',\ E'_i\ op\ E'',\ E'_{\bar{i}}\ op\ E'']$$

$$\text{or } i' > i'' \Rightarrow [i'',\ E'\ op\ E''_i,\ E'\ op\ E''_{\bar{i}}]$$

It should be remembered that the variables are assumed to be ordered; so $i' = i''$ means i' is the same as i'', $i' < i''$ means i' occurs earlier in the order of variables than i'', and $i' > i''$ means i' occurs later in the order than i''. Essentially if $i' = i''$, the evaluation of union and intersection is identical to Shannon factorised form, but if one variable precedes the other, the fact that in Shannon factorised form both sub-expressions of the 'missing' variable would be the same is used.

Each node in a modified Shannon form expression occupies three words of memory. So for a pathological case, such as a parity generator, the modified Shannon form expression will contain as many nodes as the Shannon factorised form, and so will occupy 50% more memory. However in the vast majority of practical examples, this representation is more memory efficient than the factorised form. The functions needed to evaluate union and intersection are slightly more complex, but as the number of nodes in an expression is probably reduced, the execution time may actually be shorter.

[11] where $E = b \cdot E_b + \bar{b} \cdot E_{\bar{b}}$

One further advantage is that the above representation can easily be adapted to directly represent multi-valued variables. If E is an expression whose first significant variable a is a tristate signal (with state *high*, *low* and z), then E can be represented by the modified Shannon form expression:- $[a,\ E_{a=high},\ E_{a=low},\ E_{a=z}]$. If in general, E^x has the first significant variable v^x, which has n^x possible states, then the modified Shannon form expression consists of v^x and n^x expressions E^x_i ($i = 1$ *to* n^x). The definitions of the union and intersection operators given above can be extended so that for E' and E'', E' *op* E'':-

$$\text{if } v' = v'' \Rightarrow v' \,\&\, n' \textit{ expressions} = E'_i \textit{ op } E''_i, i = 1 \textit{ to } n'$$

$$\text{or } v' < v'' \Rightarrow v' \,\&\, n' \textit{ expressions} = E'_i \textit{ op } E'', i = 1 \textit{ to } n'$$

$$\text{or } v' > v'' \Rightarrow v'' \,\&\, n'' \textit{ expressions} = E' \textit{ op } E''_i, i = 1 \textit{ to } n''$$

In all cases, if all the sub-expressions are identical, then the modified Shannon form expression can be replaced by one of the sub-expessions, (eg $[v,\ E,\ E,\ E] \equiv E$).

8 Practical application of the NODEN analyser

As was said earlier, the usefulness of NODEN depends upon the size of circuit it can analyse and how long it takes to do so. In order to investigate the properties of the representations discussed in the previous section, three analysis and comparison programs have been written and applied to the blocks of the VIPER microprocessor. These results are shown in Table 1.

The first program suite, 'Disjunctive #1', is based on the disjunctive normal form representation described in section 7.1. It would be expected that this would produce compact representations, but that the complex simplification algorithm would mean that it would be slow for large expressions. The second program suite, 'Disjunctive #2', uses the same representation but a less complex simplifier. The simplification step described in section 7.1.3 is omitted. This greatly speeds the simplification algorithm (as most of the bits of the representation can be processed in word-wide units and not individually), but at the same time the representations will contain more redundant terms and are therefore going to occupy more space. It would therefore be expected that this representation should run faster than the first, but occupy more memory. However this will not be universally true, as it is possible the extra redundant terms in the representation may increase the processing load to the point at which this representation is actually slower than the original.

The final suite of programs are based on the modified Shannon representation (section 7.3). It would be expected that this would be very much faster than the disjunctive normal form programs, but occupy far more memory. Table 1 shows the three program suites applied to blocks from VIPER (described below). For each block, the specification and implementation were analysed and the results compared. The storage requirement is the maximum amount of memory used by one of these three processes, whilst the time taken is the sum of the three processing times (CPU time on a DEC VAX 6220).

Block	Size			Gates	Disjunctive #1		Disjunctive #2		Shannon	
	Ip	Dy	Op		Store	Time	Store	Time	Store	Time
MINOR	4	3	3	16	1.4	1.6	1.4	1.3	1.5	1.2
TIMEOUT	3	6	1	32	1.9	3.2	1.9	2.4	2.2	1.8
MAJOR	10	4	5	43	3.1	19.7	3.1	15.5	21.0	28.1
TIMING	10	0	4	53	2.1	10.0	2.1	11.9	3.8	3.2
BANDSTOP	21	2	2	53	2.8	103.3	3.5	39.7	12.1	5.1
4 BIT ALU	14	0	12	62	12.1	1502.0	13.1	403.9	207.5	179.0
REGSEL	118	0	32	236	27.8	33.3	28.7	28.5	22.2	22.2
DECODER	18	21	26	295	10.0	288.8	14.0	563.7	81.9	124.5
DATAREG	71	168	180	763	94.0	519.1	97.1	290.5	75.1	71.4
TOTAL						2481.0		1357.4		436.5

Size = Inputs : delays : outputs (all boolean equivalents)
All storage in Kwords, all times in seconds

Table 1: NODEN applied to the VIPER microprocessor

The VIPER microprocessor is broken down into a number of blocks [Pygott 86], the majority of which are shown in Table 1. MINOR and TIMEOUT are essentially three and six bit counters respectively. MAJOR is a sixteen state finite state machine. TIMING is a piece of combinatorial logic from nine inputs to four outputs. BANDSTOP calculates the next state of the B and STOP flags from various information from the ALU. '4 BIT ALU' is a four bit slice of the ALU, with 13 arithmetic and logical functions. REGSEL is a 4-way, 32-bit wide multiplexer. DECODER is the micro-instruction decoder, and finally DATAREG is a register file with 4 32-bit and 2 20-bit registers.

As can be seen from Table 1, the overall behaviour of the three program suites is as predicted. The second disjunctive form programs are faster than the first, except for TIMING and DECODER, where the increased expression size slows the processing down more than the reduction in the simplifier algorithm gains. The modified Shannon form suite is faster than either of the disjunctive normal form suites, with the notable exception of the MAJOR block. This is anomalous because the MAJOR block is defined in terms of a CASE statement applied to the current state of the MAJOR finite state machine. However, the way the specification is written causes this controlling variable to appear as the last input defined. As can be seen from the definition of the Shannon representation, it is very sensitive to the order of inputs, and in this case this leads to rather inefficient processing.

9 Conclusions

The NODEN hardware analysis suite represents a practical tool to perform hardware verification on moderately complex blocks of logic. As has been demonstrated on the VIPER blocks, the component parts of a complex design can be verified in a matter of CPU minutes. This means that the more labour intensive algebraic theorem provers or checkers need only be used to show the correct behaviour of the assembled blocks.

References

[Brumfitt 87] Brumfitt P.J, Flynn M.J, Patel M, Holden T; A hardware synthesis methodology. IEE colloquium on VLSI system design; specification and synthesis, October 1987, Digest No: 1987/89

[Bryant 86] Brayant R.E; Graph based algorithms for boolean function manipulation. IEEE transactions on computers, 35(8):677-691,1986

[Cohn 87] Cohn A; A proof of correctness of the VIPER microprocessor: The first level. Proc of the hardware verification workshop, Calgary Canada, Jan 1987

[Cullyer 87] Cullyer W.J, Pygott C.H; Application of formal methods to the VIPER microprocessor. IEE proceedings Vol 134, Pt E, No 3, May 1987

[Gordon 85] Gordon M.J; HOL: a machine orientated formulation of higher-order logic. Univ Cambridge Computing Lab, Technical Report 68

[Morison 84] Morison J.D, Peeling N.E, Thorp T.L; ELLA: Hardware description or specification? Proc IEEE international conference CAD-84, Santa Clara Nov 1984

[Pitty 88] Pitty E.B; A critique of the GATEMAP logic synthesis system. International workshop on logic and architecture synthesis for silicon compilers, Grenoble May 1988

[Praxis 89] The LOCAM hardware synthesis system is supplied (and will be documented) by Praxis Ltd, 20 Manvers Rd, Bath.

[Pygott 85] Pygott C.H; Formal proof of correspondence between the specification of a hardware module and its gate level implementation. RSRE Report 85012, 1985

[Pygott 86] Pygott C.H; VIPER: The electronic block model. RSRE Report 86008, 1986

[Pygott 88] Pygott C.H; NODEN_HDL: An engineering approach to hardware verification. The fusion of Hardware Design and Verification, Milne G.J (ed), Elsevier Science Publishers B.V. IFIP 1988

[Pygott 89] Pygott C.H; The NODEN Hardware Description Language. RSRE Report 89011, July 1989

[Texas] The TTL Data Book for design engineers. Texas Instruments Ltd

DOCUMENT CONTROL SHEET

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	REPORT 89012		U/C

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS AND RADAR ESTABLISHMENT ST ANDREWS ROAD GREAT MALVERN WORCESTERSHIRE WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title THE ALGEBRA OF THE NODEN ANALYSER

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
PYGOTT C H			1989.8	32

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract NODEN is a suite of programs designed to perform hardware analysis on moderately complex blocks of logic, to prove the correspondence between the specification and implementation of a circuit. It is intended that circuits to be analysed should be described in the NODEN Hardware Description Language, NODEN HDL (either directly or by translation from other hardware description languages such as ELLA, HILO etc). The following paper describes the basic features of NODEN and the circuits it can reason about. The bulk of the paper describes the operations performed by the analyser in terms of set operations.

There is also a discussion of the possible representations that can be used for sets, and the operations on them. This leads to a comparison of the performance of a number of different analysers, based on different internal representations, when used on an actual application.