RC22503 (W0206-143) June 27, 2002
Computer Science

IBM Research Report

Formalization of the DE Language

Warren A. Hunt Jr
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Formalization of the DE Language

Warren A. Hunt, Jr.

IBM Austin Research Laboratory
11400 Burnet Road, M/S 904-6H014
Austin, TX 78758, USA

E-mail: whunt@austin.ibm.com

Abstract. We have formalized a hierarchical, occurrence-oriented hardware description language and we have developed a proof theory to permit the mechanical verification of hardware descriptions. Using the ACL2 logic, we defined a well-formedness predicate that recognizes syntactically acceptable descriptions, and a symbolic simulator that defines a cycle-based simulation semantics.

1 Introduction

We present a formal description and proof theory for a hierarchical, occurrence-oriented hardware description languages than can be used to verify hardware designs or to optimize existing hardware designs in a proveably correct manner. The rigorous definition of a hardware description language (HDL) is difficult because of the many different ways in which such languages are used; for example, a HDL may be used to specify a simulation semantics, define what circuits can be specified, restrict allowable names, enforce circuit interconnect types, estimate power consumption, and provide layout or other manufacturing information. Using the ACL2 logic [8] we have formally described a HDL that can be used to optimize library-based, gate-array designs, by symbolic propagation of their usage constraints. And using the ACL2 automated theorem-proving system we have created a system for mechanically specifying and verifying HDL-based designs.

Our HDL is designed to permit the rigorous description of hierarchical finite-state machines (FSMs). We call our language **DE** (**D**ual-**E**val) because of the two-pass approach that we employ for the language recognizers and evaluators. **DE** is actually a general-purpose language for specifying FSMs; users may define their own language primitives. Later we exhibit the definition of some typical hardware primitives, such as logic gates and storage devices.

We recognize a valid **DE** description with an ACL2 predicate; this predicate defines the permissible syntax, names, and hierarchy, of valid descriptions. Additional restrictions are imposed to ensure that combinational circularities and other anomalies are not permitted. A FSM is defined in the **DE** language by defining a well-formed (net)list of modules. Each module specifies its inputs, outputs, internal state, and a list of occurrences. Each occurrence refers to other

defined modules or to primitive modules. Each module and every occurrence within a module can be annotated.

The hierarchical structure of the **DE** language imposes syntactic restrictions on allowed descriptions. The two-pass approach employed for evaluation requires that we enforce a level-order on module references and disallows combinational circularities. A well-formed **DE** netlist has a strict naming convention that ensures every module, primitive, and wire is uniquely named – these naming restrictions also permit emulation of faulty modules and wires.

We begin our presentation by describing some related efforts before presenting some **DE** language examples. We next present the formal syntax of a **DE** netlist and show how primitive modules are defined. We have defined a number of different evaluators (conventional simulators, symbolic simulators, dependency simulators, etc.) but we just present the conventional simulator. Different evaluators are simply defined with a primitive evalutor of the type desired. We conclude by describing possible uses for the **DE** language.

2 Related Work

The idea of embedding one formal system within another has been well established by logicians. The key technique is the explicit demonstration by Goedel [2] that given the primitive recursive functions, we can then use them to write a proof checker for the mathematical system *Principia Mathematica* of Whitehead and Russell, with the formulas and proofs codes as integers. This basic technique has been used widely to show that a simulator for computational framework X can be written in the language of computational framework Y; this can be done using partial recursive functions, Turing machines, Lisp programs, etc., which led to Church's thesis that these are all different ways of characterizing what can be computed.

The hardware verification community has taken two approaches [4] to defining the semantics of circuits: shallow and deep embedding. Shallow embedding defines the semantics of a circuit description by translating it into some formal language. Deep embedding uses a formal language to define the syntax and semantics of an HDL by embedding its definition and representation into the formal language used. Shallow embedding has been usefully employed in many efforts [5, 6, 11], but the formal languages used to represent the systems investigated were not designed to represent hardware circuits. The overloading of the formal systems to represent hardware circuits does not permit the syntactic analysis of the circuits so represented – it is not possible to treat the circuit descriptions as data so a program may be used to analyze its suitability.

The **DE** language presented here has been defined by deeply embedding it inside a primitive recursive language that is a subset of Lisp [9]. The formalization of the **DE** language is quite similar in style to the embedding of the `DUAL-EVAL` HDL in the NQTHM [1]. Tom Melham used the HOL system [3] to deeply embedding some elements of a hardware description language [3]. Boyer and Hunt attempted to deeply embed a subset of VHDL in the ACL2 logic, but

this specification grew to more than 100 pages of formal mathematics, and its usefulness became suspect. In short, deeply embedding an HDL into another language brings great analytical power at the cost of having to manage all of the logical formalisms required – but these formalisms represent the real complexity that are inherit in such languages and their associated analysis and simulation systems. In short, to make such embedding useful, a serious effort needs has to be made to ensure an absolute economy of complexity.

Deep embedding uses explicit values to represent circuits, which are, in turn, given a meaning by a function or predicate defined in the same formal system used to represent the circuits. The **DE** language employs the ACL2 logic constants to represent circuits and ACL2 functions to define the language recognizers and evaluators. Conventional HDLs, such as Verilog and VHDL, actually employ the deep embedding approach; a Verilog of VHDL circuit description is represented as data in a file that is analyzed by programs written in other languages. These programs read files containing Verilog or VHDL and perform an analysis of such data files; for instance, such a data file is read and then a meaning is given to these files by a simulator. Unfortunately, this powerful approach is implemented with languages that have no formal semantics; that is, C is often used to read such data files and implement a simulator for the data files. Since C does not have a formal semantics, we do not have a proof theory or formal system to analyse any circuit so used.

The **DE** language has a deep embedding within the ACL2 language, which is a formal language with an associated proof theory. This permits the rigorous syntactic and semantic definition of the **DE** language, and such a definition allows us to use the ACL2 language to write formal specifications of circuits so represented. In addition, the ACL2 proof theory provides circuits represented within the **DE** language a proof theory defined in terms of ACL2's proof theory. In addition, the mechanization of the ACL2 system provides a mechanical theorem prover that can be used to mechanically verify **DE** HDL descriptions.

The language presented here owes much to the `DUAL-EVAL` language defined by Bishop Brock and the author for the FM9001 microprocessor verification and manufacture [7]. The `DUAL-EVAL` language was designed to syntactically similar to LSI Logic's NDL gate-array description language. The semantics of `DUAL-EVAL` attempted to mimic the behavior of a globally-clocked hardware system that emulated a FSM. The **DE** language is similar in character to `DUAL-EVAL`, but is subtly different in many ways, and its definition is much smaller, which makes reasoning it easier. In addition, the **DE** language permits user-defined primitives and a different structuring of language state-holding elements.

3 Example

The use of the `DE` language is similar to that of other hardware description languages. Circuits are specified in a hierarchical manner, and the syntactic form of the hierarchical circuit description also defines the hierarchical structure

of a description's associated state. Here we give several examples of `DE` language circuits, and describe some of the restrictions imposed by the `DE` language.

For the moment, consider the operation of the electrical circuit represented in Figure 3.1. Electrical circuits are always active; that is, every electrical circuit eagerly performing its operation as specified by electrical and physical laws. Such circuit operation can be observed by building such a circuit and measuring its behavior with an oscilloscope or similar instrument.

Our **DE** language definition is a tremendous abstraction of physical reality. The **DE** language defines finite-state machines composed where the definition of the primitive elements is provided by a user of the **DE** language. For this presentation, we assume the definition of Boolean connectives and state-holding elements; the definition of some of these primitives will be presented later. Issues such as clocking, wire delay, race conditions, power, heat, have been largely ignored.

Here we introduce the **DE** language informally. The **DE** language hierarchically defines Mealy machines: the outputs and next state of every module is a function of its inputs and internal state. By successively repeating the evaluation of an identified FSM, the **DE** system can be used to emulate typical finite-state machine operation. **DE** language definitions are written in a Lisp-style syntax using the syntax permitted for constant expressions; that is, modules definitions are represented as Lisp data and not Lisp function definitions, macros, or other such constructs. We first give an example of several combinational circuits, where we exhibit some of the restrictions our evaluation approach imposes. Later we exhibit a sequential circuit.

3.1 Combinational Modules

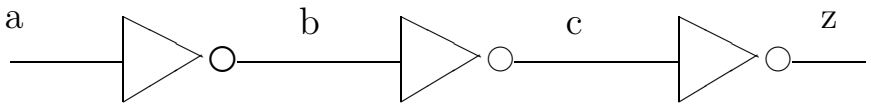

Fig. 1. Schematic of Three Inverters

Consider the circuit shown in Figure 3.1 with three inverters. This circuit can be represented as follows.

```
(three-inverters
 (type . module)
 (ins  a)
 (sts   )
 (outs z)
 (wires b c)
 (occs
```

```
(occ-0  (b)  INVERT  (a))
(occ-1  (c)  INVERT  (b))
(occ-2  (z)  INVERT  (c))
))
```

A module is identified by a name, in this case `three-inverters`. Each module is composed of a set of key-value pairs that whose entries depend on the type of the module. The type of the above module definition is `module`. All modules have inputs, states, and outputs lists, identified by `ins`, `sts`, and `outs`, respectively. The definition a `module` type of module will also include occurrences that defines the relationship between the inputs, outputs, and internal module states. As will be discussed later, state elements can themselves are structured. In this case, module `three-inverters` does not contain any internal state.

Each occurrence of a defined module must reference other defined modules. Modules with a `primitive` type contain a definition of their semantics, which will be shown in Section 5. Each occurrences must have a unique occurrence name, a list of outputs, a module reference, and a list of inputs. The three references to the `INVERT` module shown above reference some other defined module, which we will assume mimics the behavior of an inverter circuit.

Occurrences in a module have four entries: their occurrence name, an output list, a module reference, and an input list. In a given module, all of the occurrence names must be distinct. The `wires` field identifies names for the internal wire references.

The `DE` language evaluation semantics define the outputs of a module as a function of a module's inputs and internal state. The next state of a module is also a function of a module's inputs and internal state. Evaluation is not continuous, but it is discrete. Thus, there is an implicit notion of time; that is, time is broken into discrete steps that do not have anything to do with real time. This kind of finite-state machine approach is widely used in the development of digital systems and this kind of abstraction has been used successfully for years – we do not further defend our choice.

Module evaluation begins by binding input values to a module's inputs, binding state values to a module's states. Then, each occurrence is evaluated in their order of appearance. An occurrence is evaluated by binding its inputs and state to the specified arguments and then evaluating the reference itself. This is actually a recursive module evaluation. For the module defined above, input `a` will be *inverted* and the *wire* `b` will get the logical negation of the value bound to `a`. This process repeats twice more, finally, resulting with `z` being bound to the logical negation of the value bound to `a`. Any module that can be evaluated in a single pass is called combinational.

Although the definition of the following version of the `three-inverters` is syntactically identical, we do not permit such a definition. We require module inputs to be defined before they are referenced, and in the definition just below, occurrence `occ-1` appears before `occ-0` which means input `b` does not have a defined value when it is used. That is, we require that all internal variables can be correctly set by an in-order evaluation.

```
(three-inverters
 (type . module)
 (ins  a)
 (sts   )
 (outs z)
 (wires b c)
 (occs
  (occ-1  (c)  INVERT  (b))
  (occ-0  (b)  INVERT  (a))
  (occ-2  (z)  INVERT  (c))
  ))
```

Purely combinational modules that only refer recursively to other combinational or primitive modules with at most one output, can be sorted in such a way that a single-pass evaluation is possible. However, it may not be possible to sort a module containing references to other combinational modules that have more than one output.

Once again, consider the three inverter circuit shown in Figure 3.1. Imagine that instead of only having a one-input, one-output `INVERT` primitive, we also have a two-input, two-output `INVERT-2` module defined as follows.

```
(INVERT-2
 (type . module)
 (ins  a b)
 (sts   )
 (outs y z)
 (wires)
 (occs
  (occ-0  (y)  INVERT  (a))
  (occ-1  (z)  INVERT  (b))
  ))
```

There is no way we can use a reference to the `INVERT-2` module in the definition of the `three-inverters` module because references to modules must have all of their inputs defined when they are referenced.[1] It is important to realize that this restriction may make it impossible to represent a schematic diagram using the **DE** HDL.

[1] We have considered allowing an identical reference to the same module so long as it has the same inputs, states, and outputs. Such a reference would also have the same occurrence name, thus identifying it as, in some sense, a redundant module reference. In this case, each time such a module was evaluated, it would be evaluated even if some of its inputs were not defined. The trick would not to use an output of such a module before its inputs were defined. At present, we do not permit such module definitions.

3.2 Sequential Modules

Sequential modules, that is FSMs, inherently contain loops, which means that their definition will also contain loops. We permit such definitions so long as each combinational loop is broken by a state-holding element. The single-step evaluation of FSMs defined with the **DE** language proceeds in phases: first the combinational circuits are evaluated and the values of the internal wire references as collected. Then a second phase of combinational evaluation is made where every wire value is known at the beginning of the second evaluation pass. In this way, inputs to state-holding devices are known, whereas they may not have been known of the first pass.

Well-formed sequential modules impose further ordering requirements on the definition of modules and a netlist. The simplest kind of sequential module is a delay element, that one that has a single output whose value is only a function of its internal state and a single input that is used to compute the module's next state. A reference to such a module can be placed anywhere because it output can be computed without its input to be known; that is, its output is only a function of its input. In the example just below, the `F1` is such a delay element.

```
(SN74175
 (type . module)
 (ins   x0  x1  x2  x3)
 (sts   s0  s1  s2  s3)
 (outs  y0  y1  y2  y3)
 (wires  )
 (occs
  (s0  (y0)  F1  (x0))
  (s1  (y1)  F1  (x1))
  (s2  (y2)  F1  (x2))
  (s3  (y3)  F1  (x3)))
 ))
```

The `SN74175` module is a sequential module containing only four one-bit, state-holding elements. Notice the inclusion of the four occurrence names in the `sts` argument; these are used to associate a data structure containing the values of the internal state-holding elements to the specific syntactic reference for a particular module.

Evaluation of a sequential module proceeds in phases; the first phase (or pass) being identical to the evaluation of the combinational modules. A second pass is made to update the values of the state-holding elements themselves, and this is returned with the same structure and order as specified by the `sts` argument. The ordering requirements are the same as for combinational modules; the output of a primitive state-holding module can be thought of as a primary input for the purposes of checking whether module occurrences are well ordered. There are some additional restrictions that will later be seen when we present the formal definition of a well-formed netlist.

3.3 A Larger Example

Just to give a feel for the definition a larger module, we present a circuit that computes propagate and generate outputs as well as three carry outputs. This circuit is used for generating carry look-ahead values when used in conjunction with the 74181 ALU. A schematic of this circuit can be found in many TTL databooks [10].

```
(SN74182
 (type . module)
 (ins  c~ x0 x1 x2 x3 y0 y1 y2 y3)
 (sts )
 (outs cn+x~ cn+y~ cn+z~ x y)
 (wires  w0  w1  w2  w3  w4  w5  w6  w7  w8  w9
        w10 w11 w12 w13 w14 w15 w16 w17 w18)
 (occs
  ( occ-0  (w0)     INVERT ( c~))
  ( occ-1  (w1)     AND2   ( y0  x0))
  ( occ-2  (w2)     AND2   ( w0  y0))
  ( occ-3  (w3)     AND2   ( y1  x1))
  ( occ-4  (w4)     AND3   ( y0  y1  x0))
  ( occ-5  (w5)     AND3   ( w0  y0  y1))
  ( occ-6  (w6)     AND2   ( y2  x2))
  ( occ-7  (w7)     AND3   ( y1  y2  x1))
  ( occ-8  (w8)     AND4   ( y0  y1  y2  x0))
  ( occ-9  (w9)     AND4   ( w0  y0  y1  y2))
  (occ-10 (w10)     AND2   ( y3  x3))
  (occ-11 (w11)     AND3   ( y2  y3  x2))
  (occ-12 (w12)     AND4   ( y1  y2  y3  x1))
  (occ-13 (w13)     AND4   ( y3  y2  y1  y0))
  (occ-14 (w14)     OR4    ( x0  x1  x2  x3))
  (occ-15 (w15)     NOR2   ( w1  w2))
  (occ-16 (w16)     NOR3   ( w3  w4  w5))
  (occ-17 (w17)     NOR4   ( w6  w7  w8  w9))
  (occ-18 (w18)     OR4    (w10 w11 w12 w13))
  (renm-0 (cn+x~)   RENAME (w15))
  (renm-1 (cn+y~)   RENAME (w16))
  (renm-2 (cn+z~)   RENAME (w17))
  (renm-3 (y)       RENAME (w18))
  (renm-4 (x)       RENAME (w14))
 ))
```

It is not possible to use this definition with an appropriate definition of the 74181 because of the multiple output problem described above even though the definition of this module is well-formed.

4 Syntax and Arity

The syntax for a circuit is represented as a well-formed netlist, which is a list of well-formed modules. Each module is itself well-formed given that its internal structure is also well-formed. Here we exhibit the formal syntactic requirements for a netlist.

We give the definition for a well-formed netlist in a bottom-up fashion. The definitions of our recognizers for a well-formed netlist proceeds in the following order: an occurrence, a list of occurrences, a module body, a module, and finally a netlist. We begin by giving the `sx-occp` predicate that recognizes a well-formed occurrence.

```
(defun sx-occp (occ)
  (declare (xargs :guard t))
  (and (consp occ)
       (consp (cdr occ))
       (consp (cddr occ))
       (consp (cdddr occ))
       (let ((o-name  (o-name occ))
             (o-outs  (o-outs occ))
             (o-fn    (o-fn   occ))
             (o-ins   (o-ins  occ)))
         (and
          (symbolp       o-name)    ;;  Occurrence Name
          (symbol-listp  o-outs)    ;;  Outputs
          (if (atom o-fn)
              (symbolp o-fn)        ;;  Function Name
            (sx-lambdap o-fn))      ;;  Lambda Definition
          (eqlable-listp o-ins)     ;;  Inputs -- allows constants
          ))))
```

Each occurrence is composed of at least four fields: its name, outputs, reference, and inputs. Including additional fields is permitted; these additional fields are typically used for annotations, such as, to indicate that this reference should have a certain strength or size or placement. The occurrence name must be a symbol and the outputs must be a list of symbols. The reference field is either a symbol that refers to another module or it is a lambda expression that defines itself. Lambda expressions are used to define modules that are to be considered primitive modules. A syntactically well-formed set of occurrences is just a list of well-formed occurrences.

```
(defun sx-occsp (occs)
  (declare (xargs :guard t))
  (if (atom occs)
      (eq occs nil)
    (and (sx-occp  (car occs))
         (sx-occsp (cdr occs)))))
```

Each module is composed a number of fields describing its interface and a list of occurrences. Each occurrence reference may refer to some other defined module or be a lambda expression. For modules with a primitive tag field, we do not check for internal wire declarations. A module is an association list of key-value pairs. The six local variables declared in the first `let` just below look up the values for each of the keys we require a module to have. We require the input, output, and state names to all be symbols and we do not permit duplicates. If there the module is not primitive, there must be distinct wire names.

```
(defun sx-module-bodyp (body)
  (declare (xargs :guard (symbol-alistp body)))
  (let ((m-type  (m-type  body))
        (m-ins   (m-ins   body))
        (m-outs  (m-outs  body))
        (m-sts   (m-sts   body))
        (m-wires (m-wires body))
        (m-occs  (m-occs  body)))
    (and

         (symbol-listp m-ins)
         (no-duplicatesp-eq m-ins)

         (symbol-listp m-outs)
         (no-duplicatesp-eq m-outs)

         (symbol-listp m-sts)
         (no-duplicatesp-eq m-sts)

         (or (eq m-type 'primitive)
             (and (symbol-listp m-wires)
                  (no-duplicatesp-eq m-wires)))

         (sx-occsp      m-occs))))
```

A module must have a name and a well-formed body as checked by the predicate `sx-module-bodyp` above. Although we permit four different types of modules, we usually only use `primitive` and `module` types.

```
(defun sx-modulep (module)
  (declare (xargs :guard t))
  (and
   (consp module)
   (let ((m-name  (m-name module))
         (m-body  (m-body module)))
     (and
      (symbolp m-name)
      (symbol-alistp m-body)
```

```
    (let ((m-type (m-type m-body)))
      (and
       (symbolp m-type)

       (case m-type
         (black-box         t)
         (primitive         (sx-module-bodyp m-body))
         (defined-primitive (sx-module-bodyp m-body))
         (module            (sx-module-bodyp m-body))
         (otherwise         nil)))))))
```

Finally, a netlist is just a proper list of well-formed modules.

```
(defun sx-netp (netlist)
  (declare (xargs :guard t))
  (if (atom netlist)
      (eq netlist nil)
    (and (sx-modulep (car netlist))
         (sx-netp (cdr netlist)))))
```

There are actually a litany of ever more restrictive predicates whose conjunction actually defines a well-formed netlist. We will not present all of them, but the next one checks the arity of a netlist. Both the basic syntax check and the arity check are required properties of every netlist, no matter whether the netlist represents one FSM or many distinct FSMs. The arity check just ensure that the for every module reference, the number of inputs, outputs, and state references have the same arity.

The netlist arity predicate is defined in a style precisely parallel with the style of the netlist syntax predicate given above. In fact, the arity predicate assumes that the netlist to be checked has a well-formed syntax as required by the `sx-netp` predicate. The arity predicate is also presented in a bottom-up fashion. We first define the arity of a module reference, where we require that the inputs and outputs have the same arity as the inputs and outputs of the module being referenced.

```
(defun ar-module-ref (ref o-outs o-ins)
  (declare (xargs :guard (and (sx-modulep ref)
                              (symbol-listp o-outs)
                              (eqlable-listp o-ins))))
  (let* ((ref-body (m-body ref))
         (ref-outs (m-outs ref-body))
         (ref-ins  (m-ins  ref-body)))
    (and (= (len ref-outs) (len o-outs))
         (= (len ref-ins)  (len o-ins)))))
```

For the case where instead of a module name we find a lambda expression, we require that the input and output arity of lambda definition matches the number of inputs and outputs for the occurrence in which the lambda expression appears.

```
(defun ar-lambdap (le o-outs o-ins)
  (declare (xargs :guard (and (sx-lambdap le)
                              (symbol-listp o-outs)
                              (eqlable-listp o-ins))))
  (let ((l-args (l-args le))
        (l-body (l-body le)))
    (and (consp l-body)
         (eq (car l-body) 'list)
         ;; State always returned, thus one more output.
         (= (len (cdr l-body)) (1+ (len o-outs)))
         ;; State always an agrument, thus one more input.
         (= (len l-args)       (1+ (len o-ins))))))
```

The arity check for an occurrence is primarily a selection of the two arity predicates, `ar-module-ref` and `ar-lambdap`.

```
(defun ar-occp (occ netlist)
  (declare (xargs :guard (and (sx-occp occ)
                              (sx-netp netlist))))
  (let-names
   (o-outs o-fn o-ins) occ
   (if (atom o-fn)
       (let ((ref (assoc-eq o-fn netlist)))
         (and (sx-modulep ref)     ;; Should be removed by proof.
              (ar-module-ref ref o-outs o-ins)))
     (ar-lambdap o-fn o-outs o-ins))))
```

The arity of a list of occurrences is just a check that each occurrence has the correct arity.

```
(defun ar-occsp (occs netlist)
  (declare (xargs :guard (and (sx-occsp occs)
                              (sx-netp netlist))))
  (if (atom occs)
      t
    (and (ar-occp  (car occs) netlist)
         (ar-occsp (cdr occs) netlist))))
```

The arity of the body of a module is nothing more than checking the arity of the occurrences.

```
(defun ar-module-bodyp (body netlist)
  (declare (xargs :guard (and (symbol-alistp body)
                              (sx-module-bodyp body)
                              (sx-netp netlist))))
  (ar-occsp (m-occs body) netlist))
```

Similarly, the arity of a module is nothing more than an arity check of the body of the module.

```
(defun ar-modulep (module netlist)
  (declare (xargs :guard (and (sx-modulep module)
                              (sx-netp netlist))))
  (let* ((m-body  (m-body module))
         (m-type  (m-type m-body)))
    (case m-type
      (black-box         t)
      (primitive         (ar-module-bodyp m-body netlist))
      (defined-primitive (ar-module-bodyp m-body netlist))
      (module            (ar-module-bodyp m-body netlist))
      (otherwise         nil))))
```

The arity check for a netlist imposes an ordering requirement on a netlist. The arity of each module is checked with respect the the rest of the netlist. This means that there cannot be any circular references in the definitions of modules.

```
(defun ar-netp (netlist)
  (declare (xargs :guard (sx-netp netlist)))
  (if (atom netlist)
      t
    (and (ar-modulep (car netlist) (cdr netlist))
         (ar-netp (cdr netlist)))))
```

5 Primitive Modules

The `DE` language does not contain any primitive modules; primitive modules are defined by the user. Here is a definition for the `INVERT` primitive referenced earlier. There are several interesting things to note in this definition. First, the `type` field identifies it as a `primitive`, and there is a `library` designation. The inputs and outputs are identified as usual, but this definition also includes a crude measure of its cost in two-input gates. Finally, there is a single occurrence with a lambda expression that defines the semantics of the inverter.

```
(INVERT   (type . primitive)
          (library . basic)
          (ins  a)
          (sts  )
          (outs z)
          (two-input-gates . 1)
          (occs
           (st (z) (lambda (s x)
                     (list s (not x)))
               (a))))
```

The evaluation of all **DE** primitive modules actually returns both the next state, which is the first value returned, and outputs, which rest of the outputs. In this case, the state evaluation is vacuous. Note that the input `a` is actually bound

to `x` in the lambda form and the evaluator always deposits the local state in `s`. Here, `z` is the only output, and it will be bound to the expression `(not x)`.

However, the next state for this module for our one-bit delay element, is the input `in` and its output is the current state.

```
(F1       (type . defined-primitive)
          (library . basic)
          (ins  in)
          (sts  st)
          (outs z)
          (two-input-gates . 10)
          (occs
           (st (z) (lambda (s i)
                     (list i s))
               (in))))
```

6 The DE Evaluator

The definition of the evaluator is composed to two groups, each containing two mutually recursive functions. These four functions implement the entire evaluation of the outputs and next-state values for any well-formed hierarchical FSM defined using the **DE** language, except for the evaluation of lambda expressions used to evaluate lambda expressions.

We first present the evaluator for lambda expression. The lambda evaluator is a pair of mutually recursive functions. It can be seen what lambda expressions can be evaluated by inspection.

```
(mutual-recursion

(defun sn-ev (term alst)
  (declare (xargs :guard (and (sx-lexprp term)
                              (symbol-alistp alst))))
  (if (atom term)
      (cdr (assoc-eq term alst))
    (let ((fn   (car term))
          (args (cdr term)))
      (case fn
        (quote (car args))
        (list  (sn-ev-lst args alst))
        (car   (if (consp (sn-ev (car args) alst))
                   (car (sn-ev (car args) alst)) nil))
        (cdr   (if (consp (sn-ev (car args) alst))
                   (cdr (sn-ev (car args) alst)) nil))
        (if    (if (sn-ev (car args) alst)
                   (sn-ev (cadr args) alst)
                 (sn-ev (caddr args) alst)))
```

```
        (not   (not (sn-ev (car args) alst)))
        (buf   (if (sn-ev (car args) alst) t nil))
        (and   (and-lst  (sn-ev-lst args alst)))
        (nand  (nand-lst (sn-ev-lst args alst)))
        (or    (or-lst   (sn-ev-lst args alst)))
        (nor   (nor-lst  (sn-ev-lst args alst)))
        (xor   (xor-lst  (sn-ev-lst args alst)))
        (nxor  (nxor-lst (sn-ev-lst args alst)))
        (otherwise nil)))))

(defun sn-ev-lst (term-lst alst)
  (declare (xargs :guard (and (sx-lexpr-lstp term-lst)
                              (symbol-alistp alst))))
  (if (atom term-lst)
      nil
    (cons (sn-ev (car term-lst) alst)
          (sn-ev-lst (cdr term-lst) alst))))
)
```

The function below just binds the actual parameters and the state to the formal parameters of a lambda expression.

```
(defun sn-eval (le ins sts)
  (declare (xargs :guard (and (sx-lambdap le)
                              (true-listp ins)
                              (= (len (cons sts ins))
                                 (len (l-args le))))))
  (let ((args        (cons sts ins))
        (formal-args (l-args le))
        (l-body      (l-body le)))
    (sn-ev l-body (pairlis\$ formal-args args))))
```

We actually have defined a number of different evaluators. Here we only present the evaluator that operates like a conventional simulator. The function `flg-eval` selects the appropriate evaluator. We do not present the other evaluators.

```
(defun flg-eval (flg fn ins sts)
  (declare (xargs :guard (and (symbolp flg)
                              (sx-lambdap fn)
                              (true-listp ins)
                              (= (len (cons sts ins))
                                 (len (l-args fn))))
                  :verify-guards t))
  (case flg
    (ins-sts     (is-eval fn ins sts))
    (wire-check  (wc-eval fn ins sts))
```

```
    (sim         (sm-eval fn ins sts))
    (dep         (dl-eval fn ins sts))
    (simplify    (sl-eval fn ins sts))
    (simp-with-x (wx-eval fn ins sts))
    (otherwise  nil)))
```

This completes the definition of the evaluator for the simulator.

The following four functions completely define the evaluation of a netlist of modules, no matter which type of primitive evaluation is specified. That is, the functions presented in this section constitute the entire definition of a simulator for the `DE` language. This definition is small enough to allow us to reason with it mechanically. The functions `se` and `se-occ` perform the first phase of an evaluation; that is, these two functions, given values for the primary inputs and a structure state argument, compute the outputs of a reference to a module.

```
(mutual-recursion

(defun se (flg fn ins sts netlist)
  (declare (xargs :measure (se-measure fn netlist)
                  :guard (ge-static flg fn ins sts netlist)
                  :verify-guards nil))
  (if (consp fn)
      (cdr (flg-eval flg fn ins sts))
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
        (let-names
         (m-ins m-outs m-sts m-occs) (m-body module)
         (let ((wire-alist  (pairlis\$ m-ins ins))
               (sts-alist   (pairlis\$ m-sts sts))
               (new-netlist (delete-assoc-eq-netlist fn netlist)))
          (assoc-eq-values
           m-outs
           (se-occ flg m-occs wire-alist sts-alist
                   new-netlist))))))))

(defun se-occ (flg occs wire-alist sts-alist netlist)
  (declare (xargs :measure (se-measure occs netlist)
                  :guard (ge-occ-static flg occs wire-alist
                                        sts-alist netlist)
                  :verify-guards nil))
  (if (endp occs)
      wire-alist
    (let-names
     (o-name o-outs o-fn o-ins) (car occs)
     (let* ((ins      (assoc-occ-ins-values o-ins  wire-alist))
            (sts      (assoc-eq-value       o-name sts-alist))
```

```
            (new-wire-alist
             (append
              (pairlis\$ o-outs (se flg o-fn ins sts netlist))
              wire-alist)))
       (se-occ flg (cdr occs) new-wire-alist sts-alist
                   netlist)))))
)
```

Similarly, the functions `de` and `de-occ` perform the second phase of a module evaluation; that is, these two functions, given values for the primary inputs and a structure state argument, compute the next state of a reference to a module. Note, that the definition of `de` contains a reference to the function `se-occ`; this reference computes the value of the wires for the module whose next state is being computed.

```
(mutual-recursion

(defun de (flg fn ins sts netlist)
  (declare (xargs :measure (se-measure fn netlist)
                  :guard (ge-static flg fn ins sts netlist)
                  :verify-guards nil))
  (if (consp fn)
      (car (flg-eval flg fn ins sts))
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
        (let-names
         (m-ins m-sts m-occs) (m-body module)
         (let* ((wire-alist     (pairlis\$ m-ins ins))
                (sts-alist      (pairlis\$ m-sts sts))
                (new-netlist (delete-assoc-eq-netlist fn netlist))
                (all-wire-alist (se-occ flg m-occs wire-alist
                                        sts-alist new-netlist)))
           (assoc-eq-values
            m-sts
            (de-occ flg m-occs all-wire-alist sts-alist
                    new-netlist))))))))

(defun de-occ (flg occs wire-alist sts-alist netlist)
  (declare (xargs :measure (se-measure occs netlist)
                  :guard (ge-occ-static flg occs wire-alist
                                        sts-alist netlist)
                  :verify-guards nil))
  (if (endp occs)
      sts-alist
    (let-names
     (o-name o-fn o-ins) (car occs)
```

```
    (let* ((ins      (assoc-occ-ins-values o-ins  wire-alist))
           (sts      (assoc-eq-value       o-name sts-alist))
           (new-sts-alist
            (cons (cons o-name (de flg o-fn ins sts netlist))
                  sts-alist)))
      (de-occ flg (cdr occs) wire-alist new-sts-alist netlist)))))
)
```

The `de-sim` function is not a part of the definition of the **DE** language, but it provides a way to simulate a FSM by repeatedly calling the `de` function.

```
(defun de-sim (flg fn ins-list sts netlist)
  (if (atom ins-list)
      sts
    (de-sim flg fn (cdr ins-list)
            (de flg fn (car ins-list) sts netlist)
            netlist)))
```

Finally, the `de-sim-sts-outs` function just returns the last state and outputs from a `de-sim` evaluation.

```
(defun de-sim-sts-outs (flg fn ins-list sts netlist)
  (let ((ins-last (last ins-list))
        (sts-last (de-sim flg fn ins-list sts netlist)))
    (list (se flg fn ins-last sts-last netlist)
          sts-last)))
```

This concludes the definition of the **DE** language. The remarkable things regarding it definition are its brevity, it is formal, and it provides the basic of a proof theory for any FSM defined using the **DE** language. By using the ACL2 theorem prover, we can mechanically verify properties about any system defined using the **DE** language.

7 Conclusion

The formal syntax and semantics of the **DE** language were presented. The **DE** language is a hierarchical, occurrence-oriented HDL which can be used to represent hardware or software systems. A similar system, `DUAL-EVAL`, was used to prove the correctness of the FM9001 microprocessor netlist. We hope to extend the ease of using the **DE** system, by proving a series of lemmas that extend the ACL2 proof theory with rewrite rules and meta lemmas.

References

1. Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.
2. Kurt Gödel. Some Metamathematical Results on the Completeness and Consistency, On Formally Undecidable Propositions of *Principia Mathematica* and Related System *I*, and On Completeness and Consistency. In *From Frege to Gödel*, pages 592–617, Jean Van Heijenoort editor, Harvard University Press, 1967.
3. M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
4. Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with Embedding Hardware Description Languages in HOL. Proceedings of *Theorem Provers in Circuit Design*, pages 129–156, IFIP Transactions A-10, Elsevier Science Publishers, 1992.
5. M.J.C. Gordon. Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware. Technical Report 77, University of Cambridge, Computer Laboratory, September 1985.
6. Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*. LNCS 795, Springer-Verlag, 1994.
7. Warren A. Hunt, Jr. and Bishop C. Brock. A Formal HDL and Its Use in the FM9001 Verification. In C.A.R. Hoare and M.J.C. Gordon, editors, Mechanized Reasoning and Hardware Design, pages 35–48, Prentice-Hall International Series in Computer Science, Engle wood Cliffs, N.J., 1992.
8. Matt Kaufmann and J Strother Moore. ACL2: An Industrial Strength Version of NQTHM. Proceedings of the *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34, IEEE Computer Society Press, June 1996.
9. Guy Steele. Common Lisp: The Lanugage, Second Edition. Digital Press, 1990.
10. The TTL Data Book for Design Engineers, Second Edition. Texas Instruments, 1976.
11. Phil J. Windley and Micheal L. Coe. A Correctness Model for Pipelined Microprocessors, in *Theorem Provers in Circuit Design : Theory, Practice, and Experience*, Lecture Notes in Computer Science 901, Springer Verlag, pages 33-51, 1995.

```
#|
From Matt Kaufmann...

I'm not sure what you mean.  Perhaps you mean that the file contains a list of
forms, and you want VAR assigned to that list.  In that case, the book file-io
at the end of this email will do the trick.  (I think I'll put this into
Version 2.6.)  Here is a demo.

ACL2 !>(include-book "file-io")
; Fast loading /u/kaufmann/temp/file-io.fasl

Summary
Form:  ( INCLUDE-BOOK "file-io" ...)
Rules: NIL
Warnings:  None
Time:  0.39 seconds (prove: 0.00, print: 0.00, other: 0.39)
 "/u/kaufmann/temp/file-io.lisp"
ACL2 !>(read-list "foo.lisp" 'top state)
 ((IN-PACKAGE "ACL2")
  (DEFUN FOO (X) (+ 3 (CAR X)))
  (DEFUN BAR (X) (+ 3 X)))
ACL2 !>(er-let* ((my-list (read-list "foo.lisp" 'top state)))
(assign VAR my-list))
 ((IN-PACKAGE "ACL2")
  (DEFUN FOO (X) (+ 3 (CAR X)))
  (DEFUN BAR (X) (+ 3 X)))
ACL2 !>(@ var)
((IN-PACKAGE "ACL2")
 (DEFUN FOO (X) (+ 3 (CAR X)))
 (DEFUN BAR (X) (+ 3 X)))
ACL2 !>(er-let* ((my-list (read-list "a-missing-file" 'top state)))
(assign VAR my-list))


ACL2 Error in TOP:  Unable to open file a-missing-file for :object
input.

ACL2 !>(@ var)
((IN-PACKAGE "ACL2")
 (DEFUN FOO (X) (+ 3 (CAR X)))
 (DEFUN BAR (X) (+ 3 X)))
ACL2 !>

P.S. Oops -- now I realize that you wanted to evaluate the forms.  That is a
bit harder.  I've shown you how to collect up the forms, so your question
reduces to: How can one evaluate a list of forms and collect up the results?
I'm assuming that each of the forms returns a value of type *, i.e., a single
value which is not state (nor a stobj), the function my-file-file in the book
at the end of this email will work.  Here is a demo.

ACL2 !>(read-list "ev-test.lisp" 'top state) ; just to see what we've got
 ((+ A B) (- C A))
ACL2 !>(my-ev-file "ev-test.lisp" '((a . 7) (b . 2) (c . 10)) state)
 (9 3)
ACL2 !>

|#

(in-package "ACL2")

(program)

(set-state-ok t)

(defun collect-objects (list channel state)
  (mv-let (eofp obj state)
```

```
  (read-object channel state)
  (if eofp
      (mv (reverse list) state)
    (collect-objects (cons obj list) channel state))))

; Return (value result) where result is the list of top-level forms in file
; fname:
(defun read-list (fname ctx state)
  (mv-let (channel state)
  (open-input-channel fname :object state)
          (if channel
              (mv-let (result state)
                      (collect-objects () channel state)
                      (let ((state (close-input-channel channel state)))
                        (value result)))
            (er soft ctx
                "Unable to open file ~s0 for :object input."
                fname))))

(defun pprint-object-or-string (obj channel state)
  (if (stringp obj)
      (pprogn
       (newline channel state)
       (princ$ obj channel state)
       (newline channel state))
    (mv-let (col state)
            (fmt "~px~|" `((#\x . ,obj))  channel state ())
            (declare (ignore col))
            state)))

(defun write-objects (list channel state)
  (if (consp list)
      (pprogn (pprint-object-or-string (car list) channel state)
              (write-objects (cdr list) channel state)
              state)
    state))

; Pretty-print the given list of forms to file fname, except that strings are
; printed without any formatting.
(defun write-list (list fname ctx state)
  (mv-let (channel state)
  (open-output-channel fname :character state)
          (if channel
              (mv-let
               (col state)
               (fmt1 "Writing file ~x0~%" (list (cons #\0 fname))
                    0 (standard-co state) state nil)
               (declare (ignore col))
               (let ((state (write-objects list channel state)))
                 (pprogn (close-output-channel channel state)
                         (value :invisible))))
            (er soft ctx
                "Unable to open file ~s0 for :character output."
                fname))))

; (Downcase form) causes the execution of form but where printing is in
; :downcase mode.  Form must return an error triple.
(defmacro downcase (form)
  `(state-global-let*
    ((print-case :downcase))
    ,form))

; Same as write-list above, but where printing is down in downcase mode:
(defun write-list-downcase (list fname ctx state)
  (downcase (write-list list fname ctx state)))

; The following is for a given form that returns a single, non-stobj value.  It
; returns
```

```
(defun my-ev (form alist ctx state)
  (er-let* ((pair (simple-translate-and-eval
                   form alist nil "A top-level form" ctx (w state) state)))
           (value (cdr pair))))

(defun my-ev-list (list alist acc ctx state)
  (if (endp list)
      (value (reverse acc))
    (er-let* ((val (my-ev (car list) alist ctx state)))
             (my-ev-list (cdr list) alist (cons val acc) ctx state))))

(defun my-ev-file (fname alist state)
  (let ((ctx 'top-level))
    (er-let* ((list (read-list fname ctx state)))
             (my-ev-list list alist nil ctx state))))

(logic)
```

```
;;;  macros.lisp                              Warren A. Hunt, Jr.

; NOTE:  Why does <Meta>-Q replace the first colen of a package-name
;        symbol with a "."?

(in-package "ACL2")
(deflabel macros-defuns-section)

; Macro expansion is a subtle process.  Macros are expanded during
; translation (from untranslated terms to translated terms), and the
; process of translation can be found in the mutually recursive
; function calls between functions ``ACL2::MACROEXPAND1'' and
; ``ACL2::TRANSLATE11''.  Macros are expanded from the outside to the
; inside.  When the translation encounters a function call, and the
; function symbol is determined to identify a macro definition, then
; the call of the macro is replaced by its macro body and the result
; of this substitution is evaluated.  When the evaluation is complete,
; then this entire result is once again submitted to the translation
; process.  Expanding the ``FACTORIAL-MACRO'' below (with :TRANS1 and
; :TRANS) is instructive.


(defmacro factorial-macro (x)
  (declare (xargs :guard (and (integerp x)
                              (<= 0 x))))
  (if (= x 0)
      1
    (list '* x `(factorial-macro ,(1- x)))))


; Something to simplify the writing of let expressions.

(defun one-let-form (name dt)
  (list name (list name dt)))

(defun make-let-list (names dt)
  (if (atom names)
      nil
    (cons (one-let-form (car names) dt)
          (make-let-list (cdr names) dt))))

(defmacro let-names (names dt body)
  `(let ,(make-let-list names dt) ,body))


(deftheory macros-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'macros-defuns-section)))
```

```
;;;  help.lisp                                Warren A. Hunt, Jr.

;;;  This little book is a tiny collection of various things.

(in-package "ACL2")

(deflabel help-defuns-section)

;;;  Global ACL2 Flag Settings...

(set-state-ok t)


;;;  A couple of stupid macros to ease the loading of files.

(defmacro swd (dir)
  (declare (xargs :guard t))
  `(assign swd ,dir))

(defmacro lwd (file)
  (declare (xargs :guard t))
  `(ld (string-append (@ swd) ,file)))

(defmacro bvl (variable new-value)
  (declare (xargs :guard t))
  `(mv-let
    (erp result state)
    (assign ,variable ,new-value)
    (declare (ignore result))
    (value (not erp))))

(defmacro bvl-file (variable file-name)
  `(er-let*
     ((my-list (read-list ,file-name 'top state)))
     (bvl ,variable my-list)))

(defmacro ! (x y)
  (declare (xargs :guard (symbolp x)))
  `(assign ,x ,y))

;;;  Help with enabling and disabling theories.

(defmacro en-dis (a b)
  (declare (xargs :guard (and (symbol-listp a)
                              (symbol-listp b))))
  `(set-difference-theories (enable ,@a) ',b))

;;;  Macros to simplify specifications.

(defmacro declare-guard-t ()
  (declare (xargs :guard t))
  `(declare (xargs :guard t)))

(defmacro sc (x)
  (declare (xargs :guard t))
  (if (atom x)
      ""
    (list 'string-append
          (car x)
          `(sc ,(cdr x)))))

(defmacro b-fix (x)
  (declare (xargs :guard t))
  `(if ,x t nil))

(defmacro natp (x)
```

```
  (declare (xargs :guard t))
  `(and (integerp ,x) (<= 0 ,x)))

(defmacro two-symbol-listp (a b)
  (declare (xargs :guard t))
  `(and (symbol-listp ,a)
        (symbol-listp ,b)))

#|
(defmacro consp-n (x n)
  (declare (xargs :guard (natp n)))
  (if (= n 0)
      `(eq ,x nil)
    (if (= n 1)
        `(consp ,x)
      (list 'and
            `(consp ,x)
            `(consp-n (cdr ,x) ,(- n 1))))))
|#

(defmacro consp-n (x n)
  (declare (xargs :guard (natp n)))
  (if (= n 0)
      `(eq ,x nil)
    (list 'and
          `(consp ,x)
          `(consp-n (cdr ,x) ,(- n 1)))))

(defmacro true-listp-consp-n (x n)
  (declare (xargs :guard (natp n)))
  (list 'and
        (list 'true-listp `,x)
        `(consp-n ,x ,n)))

(defmacro consp-at-least-n (x n)
  (declare (xargs :guard (natp n)))
  (if (= n 0)
      `t
    (list 'and
          `(consp ,x)
          `(consp-at-least-n (cdr ,x) ,(- n 1))))))

(defmacro true-listp-consp-at-least-n (x n)
  (declare (xargs :guard (natp n)))
  (list 'and
        (list 'true-listp `,x)
        `(consp-at-least-n ,x ,n)))

;;;  We now give the definitions for several simple Boolean functions.
;;;  We define NAND and NOR in such a way that they can accept an
;;;  arbitrary number of inputs.

(defmacro nand (&rest args)
  (declare (xargs :guard t))
 `(not (and ,@args)))

(defmacro nor (&rest args)
  (declare (xargs :guard t))
 `(not (or ,@args)))

;;;  The atoms of our Lisp-based Verilog description language are
;;;  symbols and natural numbers.  We define function to aid their
;;;  manipulation.

#|

(defun member-eq (x lst)
```

```
  #-small-acl2-image
  ":Doc-Section ACL2::Programming

  membership predicate, using ~ilc[eq] as test~/

  ~c[(Member-eq x lst)] equals the longest tail of ~c[lst] that
  begins with ~c[x], or else ~c[nil] if no such tail exists.~/

  ~c[(Member-eq x lst)] is provably the same in the ACL2 logic as
  ~c[(member x lst)] and ~c[(member-equal x lst)], but it has a stronger
  ~il[guard] because it uses ~ilc[eq] for a more efficient test for whether
  ~c[x] is equal to a given member of ~c[lst].  Its ~il[guard] requires that
  ~c[lst] is a true list, and moreover, either ~c[x] is a symbol or
  ~c[lst] is a list of symbols.  ~l[member-equal] and
  ~pl[member].~/"

  (declare (xargs :guard (if (symbolp x)
                             (true-listp lst)
                           (symbol-listp lst))))
  (cond ((endp lst) nil)
        ((eq x (car lst)) lst)
        (t (member-eq x (cdr lst)))))

|#


(defun member-eql (x lst)
  (declare (xargs :guard (if (eqlablep x)
                             (true-listp lst)
                           (eqlable-listp lst))))
  (cond ((endp lst) nil)
        ((eql x (car lst)) lst)
        (t (member-eql x (cdr lst)))))

(defun all-member-eql (xs lst)
  (declare (xargs :guard (if (eqlable-listp xs)
                             (true-listp lst)
                           (and (true-listp xs)
                                (eqlable-listp lst)))))
  (if (endp xs)
      t
    (and (member-eql (car xs) lst)
         (all-member-eql (cdr xs) lst))))

(defun all-member-eq (xs lst)
  (declare (xargs :guard (if (symbol-listp xs)
                             (true-listp lst)
                           (symbol-listp lst))))
  (if (atom xs)
      t
    (and (member-eq (car xs) lst)
         (all-member-eq (cdr xs) lst))))

(defun no-member-eq (xs lst)
  (declare (xargs :guard (if (symbol-listp xs)
                             (true-listp lst)
                           (symbol-listp lst))))
  (if (atom xs)
      t
    (and (not (member-eq (car xs) lst))
         (no-member-eq (cdr xs) lst))))

(defun lst-all-member-eq (lst-xs lst)
  (declare (xargs :guard (symbol-listp lst)))
  (if (atom lst-xs)
      t
    (and (all-member-eq (car lst-xs) lst)
         (lst-all-member-eq (cdr lst-xs) lst))))
```

```
(defmacro assoc-eq-value (x alist)
  (declare (xargs :guard t))
  `(cdr (assoc-eq ,x ,alist)))

;;;  The NO-DUPLICATESP-EQ function is an inefficient way to check
;;;  that a bag is actually a set -- it takes O(n^2) in the bag size.
;;;  Much more efficient (using raw Common Lips) would be the marking
;;;  of a symbol's property list, which would require O(n) time with
;;;  respect to the bag size.

(defun no-duplicatesp-eq (l)
  (declare (xargs :guard (symbol-listp l)))
  (cond ((endp l) t)
        ((member-eq (car l) (cdr l)) nil)
        (t (no-duplicatesp-eq (cdr l)))))

(defun symbol-treep (tree)
  (declare (xargs :guard t))
  (if (atom tree)
      (eq tree nil)
    (and (consp tree)
         (consp (cdr tree))
         (symbolp (car tree))
         (symbol-treep (cadr tree))
         (symbol-treep (cddr tree)))))

(defun insert-symbol-into-tree (sym tree)
  (declare (xargs :guard (and (symbolp sym)
                              (symbol-treep tree))))
  (if (atom tree)
      (cons sym (cons nil nil))
    (if (eq sym (car tree))
        nil
      (if (symbol-< sym (car tree))
          (let ((add-left (insert-symbol-into-tree sym (cadr tree))))
            (if add-left
                (cons (car tree)
                      (cons add-left (cddr tree)))
              nil))
        (let ((add-right (insert-symbol-into-tree sym (cddr tree))))
            (if add-right
                (cons (car tree)
                      (cons (cddr tree) add-right))
              nil))))))

(defthm symbol-treep-insert-symbol-into-tree
  (implies (and (symbolp sym)
                (symbol-treep tree))
           (symbol-treep (insert-symbol-into-tree sym tree))))

(defun insert-symbol-lst-into-tree (symbols tree)
  (declare (xargs :guard (and (symbol-listp symbols)
                              (symbol-treep tree))))
  (if (atom symbols)
      tree
    (let ((new-tree (insert-symbol-into-tree (car symbols) tree)))
      (if new-tree
          (insert-symbol-lst-into-tree (cdr symbols) new-tree)
        nil))))

(defthm symbol-treep-insert-symbol-lst-into-tree
  (implies (and (symbol-listp symbols)
                (symbol-treep tree))
           (symbol-treep (insert-symbol-lst-into-tree symbols tree))))

(defun tree-for-set (l)
```

```
  (declare (xargs :guard (symbol-listp l)))
  (insert-symbol-lst-into-tree l nil))

;(defthm no-duplicatesp-eq-iff-tree-for-set
;  (implies (symbol-listp l)
;           (iff (tree-for-set l)
;                (no-duplicatesp-eq l))))


;;;  J Moore's property-list based version of the NO-DUPLICATESP
;;;  function.

; Section 1.  The Definitions

; Here is my fast version of no-duplicatesp.  Props is an private ACL2
; property list world, different from the one we use.  I just mark
; every element of lst and see if I have ever seen it as I go.  The
; function extend-world "installs" the property list behind the scenes
; in Lisp in a way strictly analogous to ACL2 arrays.  It is logically
; the identity function but makes getprop go fast if everything is up
; to date.

(defun fast-no-duplicatesp1 (lst props)
  (declare (xargs :guard (and (symbol-listp lst)
                              (worldp props))))
  (cond
   ((endp lst) t)
   ((getprop (car lst) 'mark nil 'fast-no-duplicatesp-world props)
    nil)
   (t (fast-no-duplicatesp1 (cdr lst)
                            (extend-world 'fast-no-duplicatesp-world
                                          (putprop (car lst)
                                                   'mark
                                                   t
                                                   props))))))

(defun fast-no-duplicatesp (lst)
  (declare (xargs :guard (symbol-listp lst)))
  (let ((ans (fast-no-duplicatesp1 lst nil)))
    (prog2$
     (retract-world 'fast-no-duplicatesp-world nil)
     ans)))

; The retract-world just undoes all the marks.

; Section 2. Proving It Correct

; Now I prove that fast-no-duplicatesp is equal to no-duplicatesp, the
; ACL2 function that does it in quadratic time.  Ignore all the lemmas
; until the last one of this section.  They just develop the necessary
; invariants.

(defthm sgetprop-means-fast-no-duplicatesp1-fails
  (implies (and (member e lst)
                (sgetprop e 'mark nil 'fast-no-duplicatesp-world props))
           (not (fast-no-duplicatesp1 lst props))))

(defun all-marked-t (props)
  (declare (xargs :guard (worldp props)))
  (cond ((endp props) t)
        (t (and (eq (cadr (car props)) 'mark)
                (eq (cddr (car props)) t)
                (all-marked-t (cdr props))))))

(defthm sgetprop-means-in-strip-cars
  (implies (all-marked-t props)
           (iff (sgetprop x 'mark nil name props)
                (member x (strip-cars props)))))
```

```
(defthm member-append
  (iff (member e (append a b))
       (or (member e a)
           (member e b))))

(defthm duplicatesp-preserved-by-append
  (implies (no-duplicatesp (append x y))
           (and (no-duplicatesp x)
                (no-duplicatesp y))))

(defthm no-duplicatesp-append-cons
  (equal (no-duplicatesp (append a (cons e b)))
         (and (not (member e a))
              (not (member e b))
              (no-duplicatesp (append a b)))))

(defthm fast-no-duplicatesp1-is-no-duplicatesp-append
  (implies (and (no-duplicatesp (strip-cars props))
                (all-marked-t props))
           (equal (fast-no-duplicatesp1 lst props)
                  (no-duplicatesp (append (strip-cars props) lst)))))

; Isn't this pretty?

(defthm fast-no-duplicatesp-is-no-duplicatesp
  (equal (fast-no-duplicatesp lst)
         (no-duplicatesp lst)))

(in-theory (disable fast-no-duplicatesp))


(defun remove-duplicates-eq (l)
  (declare (xargs :guard (symbol-listp l)))
  (cond ((endp l) nil)
((member-eq (car l) (cdr l))
         (remove-duplicates-eq (cdr l)))
(t (cons (car l) (remove-duplicates-eq (cdr l))))))

(defun disjoint-eq (a b)
  (declare (xargs :guard (or (and (symbol-listp a)
                                  (true-listp b))
                             (symbol-listp b))))
  (if (atom a)
      t
    (if (member-eq (car a) b)
        nil
      (disjoint-eq (cdr a) b))))

(defun subset-eq (a b)
  (declare (xargs :guard (or (and (symbol-listp a)
                                  (true-listp b))
                             (symbol-listp b))))
  (if (atom a)
      t
    (and (member-eq (car a) b)
         (subset-eq (cdr a) b))))

(defun set-equal-eq (a b)
  (declare (xargs :guard (two-symbol-listp a b)))
  (and (subset-eq a b)
       (subset-eq b a)))

(defun member-eq-names (names lst)
  (declare (xargs :guard (two-symbol-listp names lst)))
  (if (endp names)
      t
    (and (member-eq (car names) lst)
```

```
        (member-eq-names (cdr names) lst))))

(defun assoc-eq-values (syms alist)
  (declare (xargs :guard (and (symbol-listp syms)
                              (alistp alist))))
  (if (endp syms)
      nil
    (cons (assoc-eq-value (car syms) alist)
          (assoc-eq-values (cdr syms) alist))))

(defun delete-assoc-eq-element (name alist)
  (declare (xargs :guard (or (and (symbolp name)
                                  (alistp alist))
                             (symbol-alistp alist))))
  (if (atom alist)
      nil
    (if (eq name (caar alist))
        (cdr alist)
      (cons (car alist)
            (delete-assoc-eq-element name (cdr alist))))))

(defthm alistp-delete-assoc-eq-element
  (and (implies (alistp alist)
                (alistp (delete-assoc-eq-element name alist)))
       (implies (symbol-alistp alist)
                (symbol-alistp (delete-assoc-eq-element name alist)))))


(defun delete-assoc-eq-netlist (name alist)
  (declare (xargs :guard (and (symbolp name)
                              (symbol-alistp alist))))
  (if (atom alist)
      nil
    (if (eq name (caar alist))
        (cdr alist)
      (delete-assoc-eq-netlist name (cdr alist)))))

(defthm symbol-alistp-delete-assoc-eq-netlist
  (implies (symbol-alistp netlist)
           (symbol-alistp (delete-assoc-eq-netlist sym netlist))))

(defun delete-eq (name lst)
  (declare (xargs :guard (and (symbolp name)
                              (true-listp lst))))
  (if (atom lst)
      nil
    (if (eq name (car lst))
        (cdr lst)
      (cons (car lst)
            (delete-eq name (cdr lst))))))

(defthm symbol-listp-delete-eq-symbol-from-symbol-listp
  (implies (symbol-listp x)
           (symbol-listp (delete-eq a x))))

#|
(defun set-difference-eq (a b)
  (declare (xargs :guard (two-symbol-listp a b)))
  (if (atom b)
      nil
    (set-difference-eq (delete-eq (car b) a) (cdr b))))
|#

;;;  Some facts about pairlis$

(defthm alistp-symbol-listp-symbol-alistp-are-true-list
  (implies (or (alistp        lst)
               (symbol-listp  lst)
```

```
              (symbol-alistp lst))
          (true-listp lst)))

(defthm alistp-pairlis$
  (alistp (pairlis$ symbols values)))

(defthm symbol-alistp-pairlis$
  (implies (symbol-listp symbols)
           (symbol-alistp (pairlis$ symbols values))))

(defthm alistp-append
  (implies (and (alistp x)
                (alistp y))
           (alistp (append x y))))

(defthm symbol-listp-append
  (implies (and (symbol-listp x)
                (symbol-listp y))
           (symbol-listp (append x y))))

(defthm symbol-alistp-append
  (implies (and (symbol-alistp x)
                (symbol-alistp y))
           (symbol-alistp (append x y))))

(defun symbol-pair-listp (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      (eq lst nil)
    (and (consp (car lst))
         (symbolp (caar lst))
         (symbolp (cdar lst))
         (symbol-pair-listp (cdr lst)))))

(defthm symbol-pair-listp-is-alistp
  (implies (symbol-pair-listp x)
           (and (alistp x)
                (symbol-alistp x)))
  :rule-classes :forward-chaining)

(defun exprp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (or (symbolp x)
          (natp x))
    (let ((fn   (car x))
          (args (cdr x)))
      ;;  This will need to generalized.
      (and fn args nil))))

(defun symbol-expr-listp (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      (eq lst nil)
    (and (consp   (car lst))
         (symbolp (caar lst))
         (exprp   (cdar lst))
         (symbol-expr-listp (cdr lst)))))

(defthm symbol-expr-listp-is-alistp
  (implies (symbol-expr-listp x)
           (and (alistp x)
                (symbol-alistp x)))
  :rule-classes :forward-chaining)


(defthm strip-cars-symbol-alistp
  (implies (or (symbol-alistp x)
```

```
              (symbol-pair-listp x))
          (symbol-listp (strip-cars x))))

(defun lst-cons (x lst)
  (declare (xargs :guard t))
  (if (atom lst)
      nil
    (cons (cons x (car lst))
          (lst-cons x (cdr lst)))))

(defthm alistp-listp-lst-cons
  (and (true-listp (lst-cons x lst))
       (alistp     (lst-cons x lst))))

(defthm symbol-listp-lst-cons
  (implies (symbolp x)
           (symbol-alistp (lst-cons x lst))))


;;;  A function that sort of works like MEMBER-EQUAL.

(defun assoc-eq-rest (x alist)
  (declare (xargs :guard (if (symbolp x)
                             (alistp alist)
                           (symbol-alistp alist))))
  (cond ((endp alist) nil)
        ((eq x (caar alist)) alist)
        (t (assoc-eq-rest x (cdr alist)))))

(defthm assoc-eq-works-like-car-assoc-eq
  (equal (car (assoc-eq-rest x alist))
         (assoc-eq x alist)))

;;;  Some simple list recognizers...

(defun all-true-listp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (eq x nil)
    (if (atom (car x))
        (all-true-listp (cdr x))
      (and (all-true-listp (car x))
           (all-true-listp (cdr x))))))

(defthm true-listp-cdr-all-true-listp
  (implies (all-true-listp x)
           (and (true-listp (cdr x))
                (all-true-listp (cdr x)))))

(defmacro symbolp-or-natp (x)
  (declare (xargs :guard t))
  `(let ((eval-of-x ,x))
         (or (symbolp eval-of-x) (natp eval-of-x))))

(defun all-atoms-are-symbolp-or-natp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (symbolp-or-natp x)
    (and (all-atoms-are-symbolp-or-natp (car x))
         (all-atoms-are-symbolp-or-natp (cdr x)))))

(defun symbolp-natp-or-true-listp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (eq x nil)
    (let ((first (car x))
          (rest  (cdr x)))
      (if (atom first)
```

```
          (and (symbolp-or-natp first)
               (symbolp-natp-or-true-listp rest))
     (and (symbolp-natp-or-true-listp first)
          (symbolp-natp-or-true-listp rest))))))


(defthm two-equal-things
  (equal (symbolp-natp-or-true-listp x)
         (and (all-true-listp x)
              (all-atoms-are-symbolp-or-natp x))))


(defthm all-true-listp-forward-true-listp
  (implies (all-true-listp x)
           (true-listp x))
  :rule-classes :forward-chaining)

;;;  Some functions to manipulate symbols.

(defun minimum-symbol (current-symbol symbols)
  (declare (xargs :guard (and (symbolp current-symbol)
                              (symbol-listp symbols))))
  (if (atom symbols)
      current-symbol
    (if (symbol-< current-symbol (car symbols))
        (minimum-symbol current-symbol (cdr symbols))
      (minimum-symbol (car symbols) (cdr symbols)))))

(defthm symbolp-minimum-symbol
  (implies (and (symbolp sym)
                (symbol-listp lst))
           (symbolp (minimum-symbol sym lst))))

(defun sort-symbols (lst)
  (declare (xargs :guard (symbol-listp lst)))
  (if (atom lst)
      nil
    (if (atom (cdr lst))
        lst
      (let ((min-sym (minimum-symbol (car lst) (cdr lst))))
        (if (member min-sym lst)
            (cons min-sym
                  (sort-symbols (delete-eq min-sym lst)))
          lst)))))

(defthm symbol-listp-sort-symbols
  (implies (symbol-listp lst)
           (symbol-listp (sort-symbols lst))))

(defthm remove-duplicates-eq-returns-symbol-listp
  (implies (symbol-listp lst)
           (symbol-listp (remove-duplicates-eq lst))))

(defun symbol-list-listp2 (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      (eq lst nil)
    (and (symbol-listp (car lst))
         (symbol-list-listp2 (cdr lst)))))

(defun sort-symbols-lst (lst)
  (declare (xargs :guard (symbol-list-listp2 lst)))
  (if (atom lst)
      nil
    (cons (sort-symbols (car lst))
          (sort-symbols-lst (cdr lst)))))

(defthm symbol-list-listp2-sort-symbols-lst
```

```
  (implies (symbol-list-listp2 lst)
           (symbol-list-listp2 (sort-symbols-lst lst))))

(defun str-to-chars-reverse (str pos)
  (declare (xargs :guard (and (stringp str)
                              (natp pos)
                              (< pos (length str)))))
  (if (zp pos)
      (list (char str 0))
    (cons (char str pos)
          (str-to-chars-reverse str (1- pos)))))

(defthm characterp-char
  (implies (and (stringp str)
                (natp n)
(< n (length str)))
   (characterp (char str n)))
  :hints (("Goal" :in-theory (disable nth))))

(defthm character-listp-str-to-chars-reverse
  (implies (and (stringp str)
                (natp pos)
                (< pos (length str)))
           (character-listp (str-to-chars-reverse str pos)))
  :hints (("Goal" :in-theory (disable char))))


(defun str-to-chars (str)
  (declare (xargs :guard (stringp str)))
  (let ((len-str (length str)))
    (if (< len-str 1)
        nil
      (reverse (str-to-chars-reverse str (1- len-str))))))

(defthm character-listp-str-to-chars
  (implies (stringp str)
           (character-listp (str-to-chars str))))

(defun cut-char-list-at-underbar (char-list)
  (declare (xargs :guard (character-listp char-list)))
  (if (atom char-list)
      nil
    (if (equal #\_ (car char-list))
        nil
      (cons (car char-list)
            (cut-char-list-at-underbar (cdr char-list))))))

(defthm character-listp-cut-char-list-at-underbar
  (implies (character-listp char-list)
           (character-listp (cut-char-list-at-underbar char-list))))

(defun string-from-list-of-chars (char-list)
  (declare (xargs :guard (character-listp char-list)))
  (coerce char-list 'string))


(defun truncate-symbol-at-underbar (sym)
  (declare (xargs :guard (symbolp sym)))
  (let* ((str-name (symbol-name sym))
         (str-root (string-from-list-of-chars
                    (cut-char-list-at-underbar
                     (str-to-chars str-name)))))
    (intern str-root "ACL2")))

(defthm symbolp-truncate-symbol-at-underbar
  (symbolp (truncate-symbol-at-underbar sym)))
```

```
(defun truncate-symbol-at-underbar-list (sym-list)
  (declare (xargs :guard (symbol-listp sym-list)))
  (if (atom sym-list)
      nil
    (cons (truncate-symbol-at-underbar (car sym-list))
          (truncate-symbol-at-underbar-list (cdr sym-list)))))

(defthm symbol-listp-truncate-symbol-at-underbar-list
  (implies (symbol-listp sym-list)
           (symbol-listp (truncate-symbol-at-underbar-list sym-list))))


(defun nat-to-list (n)
  (declare (xargs :guard (natp n)))
  (if (zp n)
      nil
    (let ((ones (mod n 10))
          (rest (floor n 10)))
      (if (< rest n)
          (cons ones
                (nat-to-list rest))
        (list ones)))))

(defthm integer-listp-nat-to-list
  (integer-listp (nat-to-list x)))

(defun num-0-to-9-to-char (n)
  (declare (xargs :guard (integerp n)))
  (case n
    (0 '#\0)
    (1 '#\1)
    (2 '#\2)
    (3 '#\3)
    (4 '#\4)
    (5 '#\5)
    (6 '#\6)
    (7 '#\7)
    (8 '#\8)
    (9 '#\9)
    (otherwise '#\?)))

(defun nat-list-to-list-of-chars (x)
  (declare (xargs :guard (integer-listp x)))
  (if (atom x)
      nil
    (cons (num-0-to-9-to-char (car x))
          (nat-list-to-list-of-chars (cdr x)))))

(defthm character-listp-nat-list-to-list-of-chars
  (implies (integer-listp x)
           (character-listp (nat-list-to-list-of-chars x))))

(defun nat-to-string (n)
  (declare (xargs :guard (natp n)))
  (string-append
   "-"
   (if (zp n)
       "0"
     (string-from-list-of-chars
      (reverse (nat-list-to-list-of-chars
                (nat-to-list n)))))))

(defun make-symbol-with-number (sym n)
  (declare (xargs :guard (and (symbolp sym)
                              (natp n))))
  (let* ((str-name   (symbol-name sym))
         (str-number (nat-to-string n)))
    (intern (string-append str-name str-number) "ACL2")))
```

```
;;;  Some functions to collect things...

(defun list-of-true-listp (x)
  (declare (xargs :guard t))
  (if (atom x)
      t
    (and (true-listp (car x))
         (list-of-true-listp (cdr x)))))

(defun get-all-nth (n xs)
  (declare (xargs :guard (and (natp n)
                              (list-of-true-listp xs))))
  (if (atom xs)
      nil
    (cons (nth n (car xs))
          (get-all-nth n (cdr xs)))))

(defun get-all-nthcdr (n xs)
  (declare (xargs :guard (and (natp n)
                              (list-of-true-listp xs))))
  (if (atom xs)
      nil
    (cons (nthcdr n (car xs))
          (get-all-nthcdr n (cdr xs)))))

(defun collect-all-nthcdr (n xs)
  (declare (xargs :guard (and (natp n)
                              (list-of-true-listp xs))))
  (if (atom xs)
      nil
    (append (nthcdr n (car xs))
            (collect-all-nthcdr n (cdr xs)))))

(defthm true-listp-collect-all-nthcdr
  (true-listp (collect-all-nthcdr n xs)))

(defun collect-cdr-assoc-eq (syms alst)
  (declare (xargs :guard (if (symbol-listp syms)
                             (alistp alst)
                           (symbol-alistp alst))))
  (if (atom syms)
      nil
    (cons (cdr (assoc-eq (car syms) alst))
          (collect-cdr-assoc-eq (cdr syms) alst))))

(defun collect-if-car-eq-to-symbol (sym items)
  (declare (xargs :guard (or (and (symbolp sym)
                                  (alistp items))
                             (symbol-alistp items))))
  (if (atom items)
      nil
    (if (eq sym (caar items))
        (cons (car items)
              (collect-if-car-eq-to-symbol sym (cdr items)))
      (collect-if-car-eq-to-symbol sym (cdr items)))))

(defthm alistp-collect-if-car-eq-to-symbol
  (implies (or (alistp xs)
               (symbol-alistp xs))
           (alistp (collect-if-car-eq-to-symbol n xs))))

(defun collect-if-car-eq-to-lst-syms (syms items)
  (declare (xargs :guard (and (symbol-listp syms)
                              (alistp items))))
  (if (atom items)
      nil
```

```
    (if (member-eq (caar items) syms)
        (cons (car items)
              (collect-if-car-eq-to-lst-syms syms (cdr items)))
      (collect-if-car-eq-to-lst-syms syms (cdr items)))))

(defthm alistp-collect-if-car-eq-to-lst-syms
  (implies (or (alistp xs)
               (symbol-alistp xs))
           (alistp (collect-if-car-eq-to-lst-syms syms xs))))

(defun collect-symbols (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      nil
    (if (not (symbolp (car lst)))
        (collect-symbols (cdr lst))
      (cons (car lst)
            (collect-symbols (cdr lst))))))

(defthm symbol-listp-collect-symbols
  (symbol-listp (collect-symbols lst)))

(defun symbol-alist-listp (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      (eq lst nil)
    (and (symbol-alistp (car lst))
         (symbol-alist-listp (cdr lst)))))

(defun symbol-alistp-symbol-alistp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (eq x nil)
    (and (consp (car x))
         (symbolp (caar x))
         (symbol-alistp (cdar x))
         (symbol-alistp-symbol-alistp (cdr x)))))

(defthm cdr-assoc-eq-is-alistp
  (implies (symbol-alistp-symbol-alistp x)
           (and (symbol-alistp (cdr (assoc-eq fn x)))
                (alistp (cdr (assoc-eq fn x))))))

(defthm not-assoc-eq-of-alistp-is-nil
  (implies (and (symbol-alistp-symbol-alistp primitives)
                (symbolp fn)
                (symbolp fn2)
                (not (consp (assoc-eq fn2
                                      (cdr (assoc-eq fn primitives))))))
           (not (assoc-eq fn2
                          (cdr (assoc-eq fn primitives))))))

(defthm symbol-alistp-symbol-alistp-forward
  (implies (symbol-alistp-symbol-alistp x)
           (and (symbol-alistp x)
                (alistp x)))
  :rule-classes (:forward-chaining :rewrite))

(defun nat-to-v (x l)
  (declare (xargs :guard (and (natp x)
                              (natp l))))
  (if (zp l)
      nil
    (cons (if (= (rem x 2) 1) t nil)
  (nat-to-v (floor x 2) (1- l)))))

(defthm boolean-listp-nat-to-v
  (implies (and (natp x)
```

```
              (natp l))
           (boolean-listp (nat-to-v x l))))


;;;  Identify a set of symbols for the helper functions.

(deftheory help-defuns
  (set-difference-theories (current-theory :here)
                           (current-theory 'help-defuns-section)))
```

```
;;;  set-ops.lisp                              Warren A. Hunt, Jr.

; This little book contains set operations.

(in-package "ACL2")

(deflabel set-ops-defuns-section)

; Here we go...

(defthm symbol-listp-delete-eq
  (implies (symbol-listp s)
           (symbol-listp (delete-eq sym s))))

(defthm not-member-eq-delete-eq
  (implies (and (symbol-listp s)
                (no-duplicatesp s))
           (not (member-eq sym (delete-eq sym s)))))

(defthm delete-eq-non-existent-symbol
  (implies (and (symbol-listp s)
                (not (member-eq sym s)))
           (equal (delete-eq sym s) s)))

(defthm not-member-eq-delete-eq-two-sym
  (implies (and (symbol-listp s)
                (not (member-eq sym1 s)))
           (not (member-eq sym1 (delete-eq sym2 s)))))

(defthm no-duplicatesp-eq-delete-eq
  (implies (and (symbolp sym)
                (symbol-listp s)
                (no-duplicatesp-eq s))
           (no-duplicatesp-eq (delete-eq sym s))))

(defun delete-all-eq (sym s)
  (declare (xargs :guard (or (symbolp sym)
                             (symbol-listp s))))
  (if (atom s)
      nil
    (if (eq sym (car s))
(delete-all-eq sym (cdr s))
      (cons (car s)
            (delete-all-eq sym (cdr s))))))

(defthm symbol-listp-delete-all-eq
  (implies (symbol-listp s)
           (symbol-listp (delete-all-eq sym s))))

(defthm not-member-eq-delete-all-eq
  (implies (symbol-listp s)
           (not (member-eq sym (delete-all-eq sym s)))))

(defthm delete-all-eq-non-existent-symbol
  (implies (and (symbol-listp s)
                (not (member-eq sym s)))
           (equal (delete-all-eq sym s) s)))

(defthm not-member-eq-delete-all-eq-two-sym
  (implies (and (symbol-listp s)
                (not (member-eq sym1 s)))
           (not (member-eq sym1 (delete-all-eq sym2 s)))))

(defthm no-duplicatesp-eq-delete-all-eq
  (implies (and (symbolp sym)
                (symbol-listp s)
```

```
               (no-duplicatesp-eq s))
          (no-duplicatesp-eq (delete-all-eq sym s))))

(defun setp (s)
  (declare (xargs :guard t))
  (and (symbol-listp s)
       (no-duplicatesp-eq s)))

(defun set-complement (s1 tset)
  (declare (xargs :guard (or (symbol-listp s1)
                             (and (true-listp s1)
                                  (symbol-listp tset)))))
  (if (atom tset)
      nil
    (if (member-eq (car tset) s1)
(set-complement s1 (cdr tset))
      (cons (car tset)
            (set-complement s1 (cdr tset))))))

(defthm symbol-listp-set-complement
  (implies (symbol-listp tset)
           (symbol-listp (set-complement s1 tset))))

;;;  Additional challenges...

;(defthm no-duplicatesp-eq-set-complement
;  (implies (setp tset)
;           (no-duplicatesp-eq (set-complement s1 tset))))


(defun set-union (s1 s2)
  (declare (xargs :guard (and (symbol-listp s1)
                              (symbol-listp s2))))
  (if (atom s1)
      s2
    (if (member-eq (car s1) s2)
(set-union (cdr s1) s2)
      (cons (car s1) (set-union (cdr s1) s2)))))

(defthm symbol-listp-set-union
  (implies (and (symbol-listp s1)
                (symbol-listp s2))
           (symbol-listp (set-union s1 s2))))

;(defthm no-duplicatesp-eq-set-union
;  (implies (and (setp s1)
;                (setp s2))
;           (no-duplicatesp-eq (set-union s1 s2))))

(defun set-intersection (s1 s2)
  (declare (xargs :guard (and (symbol-listp s1)
                              (true-listp s2))))
  (if (atom s1)
      nil
    (if (member-eq (car s1) s2)
(cons (car s1)
              (set-intersection (cdr s1) s2))
      (set-intersection (cdr s1) s2))))

(defthm symbol-listp-set-intersection
  (implies (and (symbol-listp s1)
                (symbol-listp s2))
           (symbol-listp (set-intersection s1 s2))))

;(defthm no-duplicatesp-eq-set-union
;  (implies (setp s1)
;           (no-duplicatesp-eq (set-union s1 s2))))
```

```
(defun set-eq (s1 s2)
  (declare (xargs :guard (and (symbol-listp s1)
                              (symbol-listp s2))))
  (and (all-member-eq s1 s2)
       (all-member-eq s2 s1)))


(defun set-union-list (lst)
  (declare (xargs :guard (symbol-list-listp2 lst)))
  (if (atom lst)
      nil
    (set-union (car lst)
               (set-union-list (cdr lst)))))

(defthm symbol-list-listp2-set-union-list
  (implies (symbol-list-listp2 lst)
           (symbol-listp (set-union-list lst))))


(defun module-union (a b)
  (declare (xargs :guard (and (symbol-alistp a)
                              (symbol-alistp b))))
  (if (atom a)
      b
    (if (assoc-eq (caar a) b)
        (module-union (cdr a) b)
      (cons (car a)
            (module-union (cdr a) b)))))

(defthm symbol-alistp-module-union
  (implies (and (symbol-alistp a)
                (symbol-alistp b))
           (symbol-alistp (module-union a b))))

(defun module-union-list (lst)
  (declare (xargs :guard (symbol-alist-listp lst)))
  (if (atom lst)
      nil
    (module-union (car lst)
                  (module-union-list (cdr lst)))))

(defthm symbol-alistp-module-union-list
  (implies (symbol-alist-listp lst)
           (symbol-alistp (module-union-list lst))))

(defun union-equal-list (lst)
  (declare (xargs :guard t
                  :verify-guards nil))
  (if (atom lst)
      nil
    (if (true-listp (car lst))
        (union-equal (car lst)
                     (union-equal-list (cdr lst)))
      nil)))

(defthm true-listp-module-union-list
  (true-listp (union-equal-list lst)))

(verify-guards union-equal-list)

(defun lst-symbol-listp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (null x)
    (and (symbol-listp (car x))
         (lst-symbol-listp (cdr x)))))
```

```
(defthm symbol-listp-of-assoc-eq-lst-symbol-listp
  (implies (and (symbol-alistp alst)
                (lst-symbol-listp alst)
                (symbolp name)
                (assoc-eq name alst))
           (symbol-listp (assoc-eq name alst))))

(defthm symbol-listp-of-cdr-assoc-eq-of-lst-symbol-listp
  (implies (and (symbol-alistp alst)
                (lst-symbol-listp alst)
                (symbolp name)
                (assoc-eq name alst))
           (symbol-listp (cdr (assoc-eq name alst)))))


(defun listify-elements (x)
  (declare (xargs :guard t))
  (if (atom x)
      nil
    (cons (list (car x))
          (listify-elements (cdr x)))))

;;;  Identify a set of symbols for this file.

(deftheory set-ops-defuns
  (set-difference-theories (current-theory :here)
                           (current-theory 'set-ops-defuns-section)))
```

```
;;;  boolean-ops.lisp                          Warren A. Hunt, Jr.

; Some helper definitions for defining the simulator.

(in-package "ACL2")

(deflabel boolean-ops-defuns-section)

(defun b-and (x y)
  (declare (xargs :guard t))
  (if x (if y t nil) nil))

(defun b-or (x y)
  (declare (xargs :guard t))
  (if x t (if y t nil)))

(defun and-lst (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      t
    (and (car lst)
 (and-lst (cdr lst)))))

(defun nand-lst (lst)
  (declare (xargs :guard t))
  (not (and-lst lst)))

(defun or-lst (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      nil
    (or (car lst)
(or-lst (cdr lst)))))

(defun nor-lst (lst)
  (declare (xargs :guard t))
  (not (or-lst lst)))

(defun xor (a b)
  (declare (xargs :guard t))
  (if a (if b nil t) (if b t nil)))

(defun xor-lst (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      nil
    (xor (car lst)
 (xor-lst (cdr lst)))))

(defun nxor-lst (lst)
  (declare (xargs :guard t))
  (not (xor-lst lst)))

;;;  Identify a set of symbols for this book.

(deftheory boolean-ops-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'boolean-ops-defuns-section)))
```

```
;;;  syntax.lisp                              Warren A. Hunt, Jr.

; This book defines the acceptable syntax for the netlist format to
; which we apply our netlist simplifier.

; We now present our recognizer for netlists upon our constrain
; propagation system will operate.  Where possible, we have left the
; syntax the same as that recognized by the "raw" syntax checker.

(in-package "ACL2")
; (include-book "help")

(deflabel syntax-defuns-section)

;  For now, constants are some of the atoms.

(defun constp (constant)
  (declare (xargs :guard t))
  (and (atom constant)
       (or (symbolp  constant)
           (integerp constant)
           (stringp  constant))))

(mutual-recursion

(defun sx-lexprp (term)
  (declare (xargs :guard t))
  (if (atom term)
      (symbolp term)
    (let ((fn   (car term))
          (args (cdr term)))
      (case fn
        (quote (and (consp-n args 1)
                    (constp (car args))))
        (list  (sx-lexpr-lstp args))
        (car   (and (consp-n args 1)
                    (sx-lexpr-lstp args)))
        (cdr   (and (consp-n args 1)
                    (sx-lexpr-lstp args)))
        (if    (and (consp-n args 3)
                    (sx-lexpr-lstp args)))
        (not   (and (consp-n args 1)
                    (sx-lexpr-lstp args)))
        (buf   (and (consp-n args 1)
                    (sx-lexpr-lstp args)))
        (and   (sx-lexpr-lstp args))
        (nand  (sx-lexpr-lstp args))
        (or    (sx-lexpr-lstp args))
        (nor   (sx-lexpr-lstp args))
        (xor   (sx-lexpr-lstp args))
        (xnor  (sx-lexpr-lstp args))
        (otherwise nil)))))

(defun sx-lexpr-lstp (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      (null lst)
    (and (sx-lexprp (car lst))
         (sx-lexpr-lstp (cdr lst)))))
)

(defthm symbol-listp-add-to-set-eq
  (implies (and (symbolp sym)
                (symbol-listp lst))
           (symbol-listp (add-to-set-eq sym lst))))
```

```
(mutual-recursion

(defun collect-alist-refs (term vars)
  (declare (xargs :guard (and (sx-lexprp term)
                              (symbol-listp vars))
                  :verify-guards nil))
  (if (atom term)
      (add-to-set-eq term vars)
    (let ((fn   (car term))
          (args (cdr term)))
      (case fn
        (quote vars)
        (list  (collect-alist-lst-refs args vars))
        (not   (collect-alist-lst-refs args vars))
        (buf   (collect-alist-lst-refs args vars))
        (and   (collect-alist-lst-refs args vars))
        (or    (collect-alist-lst-refs args vars))
        (xor   (collect-alist-lst-refs args vars))
        (xnor  (collect-alist-lst-refs args vars))
        (otherwise nil)))))

(defun collect-alist-lst-refs (lst vars)
  (declare (xargs :guard (and (sx-lexpr-lstp lst)
                              (symbol-listp vars))
                  :verify-guards nil))
  (if (atom lst)
      vars
    (collect-alist-lst-refs (cdr lst)
                            (collect-alist-refs (car lst) vars))))
)

(defun collect-alist-refs-induction (x vars)
  (if (atom x)
      (list x vars)
    (list (collect-alist-refs-induction (car x) vars)
          (collect-alist-refs-induction (cdr x)
                                        (collect-alist-refs (car x) vars))
          (collect-alist-refs-induction (cdr x) vars))))

(defthm symbol-listp-collect-alist
  (and (implies (and (sx-lexprp x)
                     (symbol-listp vars))
                (symbol-listp (collect-alist-refs x vars)))
       (implies (and (sx-lexpr-lstp x)
                     (symbol-listp vars))
                (symbol-listp (collect-alist-lst-refs x vars))))
  :hints
  (("Goal" :induct (collect-alist-refs-induction x vars))))

(verify-guards collect-alist-refs)

(defun sx-lambdap (le)
  (declare (xargs :guard t))
  (and (consp le)
       (consp (cdr le))
       (consp (cddr le))
       (null (cdddr le))
       (eq (car le) 'lambda)
       (let ((formal-args (cadr le))
             (lexpr       (caddr le)))
         (and (symbol-listp formal-args)
              (sx-lexprp lexpr)
              (subset-eq (collect-alist-refs lexpr nil)
                         formal-args)))))

(defun l-args (le)
  (declare (xargs :guard (sx-lambdap le)))
  (cadr le))
```

```
(defun l-body (le)
  (declare (xargs :guard (sx-lambdap le)))
  (caddr le))


;  Accessors.

(defun m-name (module)
  (declare (xargs :guard (consp module)))
  (car module))

(defun m-body (module)
  (declare (xargs :guard (consp module)))
  (cdr module))

(defun m-type (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'type alst)))

(defun m-ins  (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'ins  alst)))

(defun m-outs (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'outs alst)))

(defun m-deps (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'deps alst)))

(defun m-sts (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'sts alst)))

(defun m-wires (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'wires alst)))

(defun m-fn (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'fn alst)))

(defun m-transistors (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'transistors alst)))

(defun m-occs (alst)
  (declare (xargs :guard (alistp alst)))
  (cdr (assoc-eq 'occs alst)))

(defun o-name (occ)
  (declare (xargs :guard (consp occ)))
  (car occ))

(defun o-outs (occ)
  (declare (xargs :guard (and (consp occ)
                              (consp (cdr occ)))))
  (cadr occ))

(defun o-fn (occ)
  (declare (xargs :guard (and (consp occ)
                              (consp (cdr occ))
                              (consp (cddr occ)))))
  (caddr occ))

(defun o-ins (occ)
```

```
  (declare (xargs :guard (and (consp occ)
                              (consp (cdr occ))
                              (consp (cddr occ))
                              (consp (cdddr occ)))))
  (cadddr occ))


;  Syntax check for occurrences and modules

;  In an occurrence inside a module, the inputs may only be
;  symbols are natural numbers.

(defun sx-occp (occ)
  (declare (xargs :guard t))
  (and (consp occ)
       (consp (cdr occ))
       (consp (cddr occ))
       (consp (cdddr occ))
       (let ((o-name  (o-name occ))
             (o-outs  (o-outs occ))
             (o-fn    (o-fn   occ))
             (o-ins   (o-ins  occ)))
         (and
          (symbolp       o-name)    ;;  Occurrence Name
          (symbol-listp  o-outs)    ;;  Outputs
;          (pseudo-termp (cons o-fn o-ins))
          (if (atom o-fn)
              (symbolp o-fn)        ;;  Function Name
            (sx-lambdap o-fn))      ;;  Lambda Definition
;          (symbol-listp  o-ins)     ;;  Inputs
          (eqlable-listp o-ins)     ;;  Inputs -- allows constants
          ))))

(defun sx-occsp (occs)
  (declare (xargs :guard t))
  (if (atom occs)
      (eq occs nil)
    (and (sx-occp  (car occs))
         (sx-occsp (cdr occs)))))

(defun sx-module-bodyp (body)
  (declare (xargs :guard (symbol-alistp body)))
  (let ((m-type  (m-type  body))
        (m-ins   (m-ins   body))
        (m-outs  (m-outs  body))
        (m-sts   (m-sts   body))
        (m-wires (m-wires body))
        (m-occs  (m-occs  body)))
    (and

         (symbol-listp m-ins)
         (no-duplicatesp-eq m-ins)

         (symbol-listp m-outs)
         (no-duplicatesp-eq m-outs)

         (symbol-listp m-sts)
         (no-duplicatesp-eq m-sts)

         ;;  I need to re-think the testing for wire names.
         ;;  When Dana and I were working on the simplifier,
         ;;  it became apparent that this test is not sufficient!!!

         (or (eq m-type 'primitive)
             (and (symbol-listp m-wires)
                  ;; Need additional checks for wires.
                  ;; This failed defined primtives because of
                  ;; empty wire list.
```

```
                (no-duplicatesp-eq m-wires)))

        (sx-occsp      m-occs))))

(defun sx-modulep (module)
  (declare (xargs :guard t))
  (and
   (consp module)
   (let ((m-name  (m-name module))
         (m-body  (m-body module)))
     (and
      (symbolp m-name)
      (symbol-alistp m-body)
      (let ((m-type (m-type m-body)))
        (and
         (symbolp m-type)

         (case m-type
           (black-box         t)
           (primitive         (sx-module-bodyp m-body))
           (defined-primitive (sx-module-bodyp m-body))
           (module            (sx-module-bodyp m-body))
           (otherwise         nil))))))))

(defun sx-netp (netlist)
  (declare (xargs :guard t))
  (if (atom netlist)
      (eq netlist nil)
    (and (sx-modulep (car netlist))
         (sx-netp (cdr netlist)))))


(defthm symbol-alistp-sx-netp
  (implies (sx-netp netlist)
           (symbol-alistp netlist))
  :rule-classes :forward-chaining)

(defthm symbol-alistp-cdr-assoc-eq
  (implies (and (sx-netp netlist)
                (assoc-eq fn netlist))
           (symbol-alistp (cdr (assoc-eq fn netlist))))
  :rule-classes :forward-chaining)

(defthm sx-modulep-assoc-eq-netlist-rewrite
  (implies (and (symbolp fn)
                (sx-netp netlist)
                (assoc-eq fn netlist))
           (sx-modulep (assoc-eq fn netlist))))

(defthm sx-modulep-assoc-eq-netlist-forward
  (implies (and (assoc-eq fn netlist)
                (symbolp fn)
                (sx-netp netlist))
           (sx-modulep (assoc-eq fn netlist)))
  :rule-classes :forward-chaining)

;;;  Debugging functions...

(defun sx-net-listp (netlist)
  (declare (xargs :guard t))
  (if (atom netlist)
      nil
    (cons (sx-modulep (car netlist))
          (sx-net-listp (cdr netlist)))))

(defun sx-occs-listp (occs)
  (declare (xargs :guard t))
  (if (atom occs)
```

```
      nil
    (cons (sx-occp  (car occs))
          (sx-occs-listp (cdr occs)))))

(defun pick-module-and-check-sx-occs (n netlist)
  (sx-occs-listp  (nth n netlist))))

;;;  Identify a set of symbols for this book.

(deftheory syntax-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'syntax-defuns-section)))
```

```
;;;  arity.lisp                              Warren A. Hunt, Jr.

; This book defines a mechanism to check the arity of a netlist.

(in-package "ACL2")
; (include-book "syntax")

(deflabel arity-defuns-section)

(defun ar-module-ref (ref o-outs o-ins)
  (declare (xargs :guard (and (sx-modulep ref)
                              (symbol-listp o-outs)
                              (eqlable-listp o-ins))))
  (let* ((ref-body (m-body ref))
         (ref-outs (m-outs ref-body))
         (ref-ins  (m-ins  ref-body)))
    (and (= (len ref-outs) (len o-outs))
         (= (len ref-ins)  (len o-ins)))))

(defun ar-lambdap (le o-outs o-ins)
  (declare (xargs :guard (and (sx-lambdap le)
                              (symbol-listp o-outs)
                              (eqlable-listp o-ins))))
  (let ((l-args (l-args le))
        (l-body (l-body le)))
    (and (consp l-body)
         (eq (car l-body) 'list)
         ;; State always returned, thus one more output.
         (= (len (cdr l-body)) (1+ (len o-outs)))
         ;; State always an agrument, thus one more input is supplied.
         (= (len l-args)       (1+ (len o-ins))))))

(defun ar-occp (occ netlist)
  (declare (xargs :guard (and (sx-occp occ)
                              (sx-netp netlist))))
  (let-names
   (o-outs o-fn o-ins) occ
   (if (atom o-fn)
       (let ((ref (assoc-eq o-fn netlist)))
         (and (sx-modulep ref)     ;; Should be removed by proof.
              (ar-module-ref ref o-outs o-ins)))
     (ar-lambdap o-fn o-outs o-ins))))

(defun ar-occsp (occs netlist)
  (declare (xargs :guard (and (sx-occsp occs)
                              (sx-netp netlist))
                  :guard-hints
                  (("Goal"
                    :in-theory (en-dis () (sx-occp ar-occp))))))
  (if (atom occs)
      t
    (and (ar-occp  (car occs) netlist)
         (ar-occsp (cdr occs) netlist))))


(defun ar-module-bodyp (body netlist)
  (declare (xargs :guard (and (symbol-alistp body)
                              (sx-module-bodyp body)
                              (sx-netp netlist))))
  (ar-occsp (m-occs body) netlist))


(defun ar-modulep (module netlist)
  (declare (xargs :guard (and (sx-modulep module)
                              (sx-netp netlist))
                  :guard-hints
```

```
                ((\"Goal\"
                  :in-theory (en-dis () (sx-module-bodyp ar-module-bodyp))))))
  (let* ((m-body  (m-body module))
         (m-type  (m-type m-body)))
    (case m-type
      (black-box         t)
      (primitive         (ar-module-bodyp m-body netlist))
      (defined-primitive (ar-module-bodyp m-body netlist))
      (module            (ar-module-bodyp m-body netlist))
      (otherwise         nil))))

(defun ar-netp (netlist)
  (declare (xargs :guard (sx-netp netlist)
                  :guard-hints
                  ((\"Goal\"
                    :in-theory (en-dis () (sx-modulep ar-modulep))))))
  (if (atom netlist)
      t
    (and (ar-modulep (car netlist) (cdr netlist))
         (ar-netp (cdr netlist)))))


;;;  Debugging functions...

(defun ar-occs-listp (occs netlist)
  (declare (xargs :guard (and (sx-occsp occs)
                              (sx-netp netlist))
                  :guard-hints
                  ((\"Goal\"
                    :in-theory (en-dis () (sx-occp ar-occp))))))
  (if (atom occs)
      nil
    (cons (ar-occp       (car occs) netlist)
          (ar-occs-listp (cdr occs) netlist))))

(defun ar-net-listp (netlist)
  (declare (xargs :guard (sx-netp netlist)
                  :guard-hints
                  ((\"Goal\"
                    :in-theory (en-dis () (sx-modulep ar-modulep))))))
  (if (atom netlist)
      nil
    (cons (ar-modulep (car netlist) (cdr netlist))
          (ar-net-listp (cdr netlist)))))

(defun pick-module-and-check-ar-occs (n netlist)
  (ar-occs-listp (m-occs (m-body (nth n netlist))) netlist))

;;;  Identify a set of symbols for this book.

(deftheory arity-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'arity-defuns-section)))
```

```
;;;  wire-check.lisp                          Warren A. Hunt, Jr.

; For each different flag given to SE and DE (see file "de.lisp") we
; have different primitive evaluators.  This evaluator checks to
; ensure that all signals are defined.  This evaluator assumes that
; the primary inputs all contain T symbols.  The state should be
; well-formed and contain T symbols as well.

; If any not T symbol is returned, then the wire check failed.

(in-package "ACL2")

(deflabel wire-check-defuns-section)

(mutual-recursion

(defun wc-ev (term alst)
  (declare (xargs :guard (and (sx-lexprp term)
      (symbol-alistp alst))))
  (if (atom term)
      (cdr (assoc-eq term alst))
    (let ((fn   (car term))
  (args (cdr term)))
      (case fn
(quote t)
(list  (wc-ev-lst args alst))
        (car   (if (consp (wc-ev (car args) alst))
                   (car (wc-ev (car args) alst)) nil))
        (cdr   (if (consp (wc-ev (car args) alst))
                   (cdr (wc-ev (car args) alst)) nil))
        (if    (and (wc-ev (car args) alst)
                    (wc-ev (cadr args) alst)
                    (wc-ev (caddr args) alst)))
(not   (wc-ev (car args) alst))
(buf   (wc-ev (car args) alst))
(and   (and-lst (wc-ev-lst args alst)))
(nand  (and-lst (wc-ev-lst args alst)))
(or    (and-lst (wc-ev-lst args alst)))
(nor   (and-lst (wc-ev-lst args alst)))
(xor   (and-lst (wc-ev-lst args alst)))
(nxor  (and-lst (wc-ev-lst args alst)))
(otherwise nil)))))

(defun wc-ev-lst (term-lst alst)
  (declare (xargs :guard (and (sx-lexpr-lstp term-lst)
      (symbol-alistp alst))))
  (if (atom term-lst)
      nil
    (cons (wc-ev (car term-lst) alst)
  (wc-ev-lst (cdr term-lst) alst))))
)

(defun wc-eval (le ins sts)
  (declare (xargs :guard (and (sx-lambdap le)
      (true-listp ins)
      (= (len (cons sts ins))
                                 (len (l-args le))))))
  (let ((args        (cons sts ins))
(formal-args (l-args le))
(l-body      (l-body le)))
    (wc-ev l-body (pairlis$ formal-args args))))


;;;  Identify a set of symbols for this book.

(deftheory wire-check-section
```

```
(set-difference-theories (current-theory :here)
                         (current-theory 'wire-check-defuns-section)))
```

```
;;;  ins-sts.lisp                              Warren A. Hunt, Jr.

; For each different flag given to SE and DE (see file "de.lisp") we
; have different primitive evaluators.  This evaluator checks to
; ensure that "the shape" of the input arguments are correct.  This
; evaluator can be called with any valid arguments that would be given
; to the simulator.

; Lambda IN expressions are expected to have ATOMs as input arguments.
; Lambda STS expressions are (for now) expected to have a singleton
; atom.

; This is another subtle issue and will likely need to be revisited.
; It relates directly to whether we can have primitives with large
; amounts of state, like a memory, or whether such structured
; state-holding devices must be defined using primitives.

; In fact, the issue runs deeper than just that whether state can be
; structured.  It is clear that bit-vector inputs will be required to
; handle many of the circuits that should optimized by SeqSimp.  In a
; second pass, we will need to generalize every argument to permit the
; use of vectors.

(in-package "ACL2")

(deflabel ins-sts-defuns-section)

(mutual-recursion

(defun is-ev (term alst)
  (declare (xargs :guard (and (sx-lexprp term)
      (symbol-alistp alst))))
  (if (atom term)
      (atom (cdr (assoc-eq term alst)))
    (let ((fn   (car term))
  (args (cdr term)))
      (case fn
(quote t)
(list  (is-ev-lst args alst))
        (car   (if (consp (is-ev (car args) alst))
                   (car (is-ev (car args) alst)) nil))
        (cdr   (if (consp (is-ev (car args) alst))
                   (cdr (is-ev (car args) alst)) nil))
        (if    (and (is-ev (car args) alst)
                    (is-ev (cadr args) alst)
                    (is-ev (caddr args) alst)))
(not   (is-ev (car args) alst))
(buf   (is-ev (car args) alst))
(and   (and-lst (is-ev-lst args alst)))
(nand  (and-lst (is-ev-lst args alst)))
(or    (and-lst (is-ev-lst args alst)))
(nor   (and-lst (is-ev-lst args alst)))
(xor   (and-lst (is-ev-lst args alst)))
(nxor  (and-lst (is-ev-lst args alst)))
(otherwise nil)))))

(defun is-ev-lst (term-lst alst)
  (declare (xargs :guard (and (sx-lexpr-lstp term-lst)
      (symbol-alistp alst))))
  (if (atom term-lst)
      nil
    (cons (is-ev (car term-lst) alst)
  (is-ev-lst (cdr term-lst) alst))))
)

(defun is-eval (le ins sts)
```

```
  (declare (xargs :guard (and (sx-lambdap le)
      (true-listp ins)
      (= (len (cons sts ins))
                                (len (l-args le))))))
  (let ((args        (cons sts ins))
(formal-args (l-args le))
(l-body      (l-body le)))
    (is-ev l-body (pairlis$ formal-args args))))


;;;  Identify a set of symbols for this book.

(deftheory ins-sts-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'ins-sts-defuns-section)))
```

```
;;;  sim-int.lisp                            Warren A. Hunt, Jr.

; For each different flag given to SE and DE (see file "de.lisp")
; we have different primitive evaluators.  This is the evaluator
; for simulation using 1s and 0s instead of Lisp's T and NIL.

; For a while, I thought the type of the objects returned by the
; simulator would be restricted.  For example, at first I thought SN-EV
; (below) would only return a Boolean value, while SN-EV-LST would
; return a list of Boolean values.  This wasn't clear thinking.
; SN-EV-LST will clearly return a list, but SN-EV can return anything
; constructed by an evaluated TERM or for that matter is on the ALST.

; This free wheeling approach to term evaluation means that the other
; types of evaluators must be constructed carefully.

(in-package "ACL2")

(deflabel sim-int-defuns-section)

(mutual-recursion

(defun sn-ev (term alst)
  (declare (xargs :guard (and (sx-lexprp term)
      (symbol-alistp alst))))
  (if (atom term)
      (cdr (assoc-eq term alst))
    (let ((fn   (car term))
  (args (cdr term)))
      (case fn
(quote (car args))
(list  (sn-ev-lst args alst))
        (car   (if (consp (sn-ev (car args) alst))
                   (car (sn-ev (car args) alst)) nil))
        (cdr   (if (consp (sn-ev (car args) alst))
                   (cdr (sn-ev (car args) alst)) nil))
        (if    (if (sn-ev (car args) alst)
                   (sn-ev (cadr args) alst)
                 (sn-ev (caddr args) alst)))
(not   (not (sn-ev (car args) alst)))
(buf   (if (sn-ev (car args) alst) t nil))
(and   (and-lst  (sn-ev-lst args alst)))
(nand  (nand-lst (sn-ev-lst args alst)))
(or    (or-lst   (sn-ev-lst args alst)))
(nor   (nor-lst  (sn-ev-lst args alst)))
(xor   (xor-lst  (sn-ev-lst args alst)))
(nxor  (nxor-lst (sn-ev-lst args alst)))
(otherwise nil)))))

(defun sn-ev-lst (term-lst alst)
  (declare (xargs :guard (and (sx-lexpr-lstp term-lst)
      (symbol-alistp alst))))
  (if (atom term-lst)
      nil
    (cons (sn-ev (car term-lst) alst)
  (sn-ev-lst (cdr term-lst) alst))))
)

(defun sn-eval (le ins sts)
  (declare (xargs :guard (and (sx-lambdap le)
      (true-listp ins)
      (= (len (cons sts ins))
                                  (len (l-args le))))))
  (let ((args        (cons sts ins))
(formal-args (l-args le))
(l-body      (l-body le)))
```

```
    (sn-ev l-body (pairlis$ formal-args args))))

;;;  Identify a set of symbols for this book.

(deftheory sim-int-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'sim-int-defuns-section)))
```

```
;;;  simulate.lisp                          Warren A. Hunt, Jr.

; For each different flag given to SE and DE (see file "de.lisp")
; we have different primitive evaluators.  This is the evaluator
; for simulation.

; For a while, I thought the type of the objects returned by the
; simulator would be restricted.  For example, at first I thought SM-EV
; (below) would only return a Boolean value, while SM-EV-LST would
; return a list of Boolean values.  This wasn't clear thinking.
; SM-EV-LST will clearly return a list, but SM-EV can return anything
; constructed by an evaluated TERM or for that matter is on the ALST.

; This free wheeling approach to term evaluation means that the other
; types of evaluators must be constructed carefully.

(in-package "ACL2")

(deflabel simulate-defuns-section)

(defun sm-ev-fix (x)
  (declare (xargs :guard t))
  (logbitp 0 (ifix x)))

(defun sm-ev-fix-lst (x)
  (declare (xargs :guard t))
  (if (atom x)
      nil
    (cons (sm-ev-fix (car x))
          (sm-ev-fix-lst (cdr x)))))

(mutual-recursion

(defun sm-ev (term alst)
  (declare (xargs :guard (and (sx-lexprp term)
      (symbol-alistp alst))))
  (if (atom term)
      (cdr (assoc-eq term alst))
    (let ((fn   (car term))
  (args (cdr term)))
      (case fn
(quote (car args))
(list  (sm-ev-lst args alst))
        (car   (if (consp (sm-ev (car args) alst))
                   (car (sm-ev (car args) alst)) nil))
        (cdr   (if (consp (sm-ev (car args) alst))
                   (cdr (sm-ev (car args) alst)) nil))
        (if    (if (sm-ev (car args) alst)
                   (sm-ev (cadr args) alst)
                 (sm-ev (caddr args) alst)))
(not   (not (sm-ev (car args) alst)))
(buf   (if (sm-ev (car args) alst) t nil))
(and   (and-lst  (sm-ev-lst args alst)))
(nand  (nand-lst (sm-ev-lst args alst)))
(or    (or-lst   (sm-ev-lst args alst)))
(nor   (nor-lst  (sm-ev-lst args alst)))
(xor   (xor-lst  (sm-ev-lst args alst)))
(nxor  (nxor-lst (sm-ev-lst args alst)))
(otherwise nil)))))

(defun sm-ev-lst (term-lst alst)
  (declare (xargs :guard (and (sx-lexpr-lstp term-lst)
      (symbol-alistp alst))))
  (if (atom term-lst)
      nil
    (cons (sm-ev (car term-lst) alst)
```

```
  (sm-ev-lst (cdr term-lst) alst))))
)

(defun sm-eval (le ins sts)
  (declare (xargs :guard (and (sx-lambdap le)
      (true-listp ins)
      (= (len (cons sts ins))
                                (len (l-args le))))))
  (let* ((args        (cons sts ins))
         (formal-args (l-args le))
         (l-body      (l-body le)))
    (sm-ev l-body (pairlis$ formal-args args))))

;;;  Identify a set of symbols for this book.

(deftheory simulate-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'simulate-defuns-section)))
```

```
;;;  depends.lisp                              Warren A. Hunt, Jr.

; Here we define the dependencies for a netlist reference.  Each input
; and state-holding element is seeded with a list containing symbols.
; The primary outputs and the next state-holding elements will
; contains symbols from inputs and current state-holding elements that
; can combinationally effect the outputs and next state.

(in-package "ACL2")

(deflabel depends-defuns-section)

(mutual-recursion

(defun dl-ev (term alst)
  (declare (xargs :guard (and (sx-lexprp term)
      (symbol-alistp alst))
                  :verify-guards t))
  (if (atom term)
      (cdr (assoc-eq term alst))
    (let ((fn   (car term))
  (args (cdr term)))
      (if (eq fn 'quote)
          nil
        (let ((dl-ev-args (dl-ev-lst args alst)))
          (case fn
            (list  dl-ev-args)
            (car   (if (consp (car dl-ev-args))
                       (car (car dl-ev-args)) nil))
            (cdr   (if (consp (car dl-ev-args))
                       (cdr (car dl-ev-args)) nil))
            (if    (union-equal-list dl-ev-args))
            (not   (dl-ev (car args) alst))
            (buf   (dl-ev (car args) alst))
            (and   (union-equal-list dl-ev-args))
            (nand  (union-equal-list dl-ev-args))
            (or    (union-equal-list dl-ev-args))
            (nor   (union-equal-list dl-ev-args))
            (xor   (union-equal-list dl-ev-args))
            (nxor  (union-equal-list dl-ev-args))

            (otherwise nil)))))))

(defun dl-ev-lst (term-lst alst)
  (declare (xargs :guard (and (sx-lexpr-lstp term-lst)
      (symbol-alistp alst))
                  :verify-guards t))
  (if (atom term-lst)
      nil
    (cons (dl-ev (car term-lst) alst)
  (dl-ev-lst (cdr term-lst) alst))))
)

(defun dl-eval (le ins sts)
  (declare (xargs :guard (and (sx-lambdap le)
      (true-listp ins)
      (= (len (cons sts ins))
                                (len (l-args le))))
                  :verify-guards t))
  (let ((args        (cons sts ins))
(formal-args (l-args le))
(l-body      (l-body le)))
    (dl-ev l-body (pairlis$ formal-args args))))


;;;  Identify a set of symbols for this book.
```

```
(deftheory depends-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'depends-defuns-section)))
```

```
;;;  simplify.lisp                              Warren A. Hunt, Jr.

; Here we define a very crude simplifier.  This simplifier is very
; similar to the dependency evaluator, and maybe this simplifier and
; dependency evaluator should be merged.

(in-package "ACL2")

(deflabel simplify-defuns-section)

(defconst *sl-1* '(t))
(defconst *sl-0* '(nil))

(defun sl-1p (lst)
  (declare (xargs :guard t))
  (equal lst *sl-1*))

(defun sl-0p (lst)
  (declare (xargs :guard t))
  (equal lst *sl-0*))

(defun sl-not (lst)
  (declare (xargs :guard t))
  (if (sl-1p lst)
      *sl-0*
    (if (sl-0p lst)
        *sl-1*
      lst)))

(defun sl-and (l1 l2)
  (declare (xargs :guard (and (true-listp l1)
                              (true-listp l2))))
  (if (or (sl-0p l1) (sl-0p l2))
      *sl-0*
    (if (sl-1p l1)
        l2
      (if (sl-1p l2)
          l1
        (union-equal l1 l2)))))

(defun sl-and-lst (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      *sl-1*
    (if (true-listp (car lst))
        (sl-and (car lst)
                (sl-and-lst (cdr lst)))
      nil)))

(defun sl-nand-lst (lst)
  (declare (xargs :guard t))
  (sl-not (sl-and-lst lst)))

(defun sl-or (l1 l2)
  (declare (xargs :guard (and (true-listp l1)
                              (true-listp l2))))
  (if (or (sl-1p l1) (sl-1p l2))
      *sl-1*
    (if (sl-0p l1)
        l2
      (if (sl-0p l2)
          l1
        (union-equal l1 l2)))))

(defun sl-or-lst (lst)
  (declare (xargs :guard t))
```

```
  (if (atom lst)
      *sl-0*
    (if (true-listp (car lst))
        (sl-or (car lst)
               (sl-or-lst (cdr lst)))
      nil)))

(defun sl-nor-lst (lst)
  (declare (xargs :guard t))
  (sl-not (sl-or-lst lst)))

(defun sl-xor (l1 l2)
  (declare (xargs :guard (and (true-listp l1)
                              (true-listp l2))))
  (if (sl-1p l1)
      (sl-not l2)
    (if (sl-0p l1)
        l2
      (if (sl-1p l2)
          (sl-not l1)
        (if (sl-0p l2)
            l1
          (union-equal l1 l2))))))

(defun sl-xor-lst (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      *sl-0*
    (if (true-listp (car lst))
        (sl-xor (car lst)
                (sl-xor-lst (cdr lst)))
      nil)))

(defun sl-nxor-lst (lst)
  (declare (xargs :guard t))
  (sl-not (sl-xor-lst lst)))

(defun sl-if (test t1 t0)
  (declare (xargs :guard t))
  (if (sl-1p test)
      t1
    (if (sl-0p test)
        t0
      (if (and (symbol-listp test)
               (symbol-listp t0)
               (symbol-listp t1))
          (union-equal test
                       (union-equal t0 t1))
        nil))))


(mutual-recursion

(defun sl-ev (term alst)
  (declare (xargs :guard (and (sx-lexprp term)
                              (symbol-alistp alst))
                  :verify-guards t))
  (if (atom term)
      (cdr (assoc-eq term alst))
    (let ((fn   (car term))
          (args (cdr term)))
      (if (eq fn 'quote)
          nil
        (let ((sl-ev-args (sl-ev-lst args alst)))
          (case fn
            (list  sl-ev-args)
            (car   (if (consp (car sl-ev-args))
                       (car (car sl-ev-args)) nil))
```

```
            (cdr   (if (consp (car sl-ev-args))
                       (cdr (car sl-ev-args)) nil))
            (if    (sl-if (sl-ev (car args)   alst)
                          (sl-ev (cadr args)  alst)
                          (sl-ev (caddr args) alst)))
            (not   (sl-not (sl-ev (car args) alst)))
            (buf   (sl-ev (car args) alst))
            (and   (sl-and-lst  sl-ev-args))
            (nand  (sl-nand-lst sl-ev-args))
            (or    (sl-or-lst   sl-ev-args))
            (nor   (sl-nor-lst  sl-ev-args))
            (xor   (sl-xor-lst  sl-ev-args))
            (nxor  (sl-nxor-lst sl-ev-args))
            (otherwise nil)))))))

(defun sl-ev-lst (term-lst alst)
  (declare (xargs :guard (and (sx-lexpr-lstp term-lst)
                              (symbol-alistp alst))
                  :verify-guards t))
  (if (atom term-lst)
      nil
    (cons (sl-ev (car term-lst) alst)
          (sl-ev-lst (cdr term-lst) alst))))
)

(defun sl-eval (le ins sts)
  (declare (xargs :guard (and (sx-lambdap le)
                              (true-listp ins)
                              (= (len (cons sts ins))
                                 (len (l-args le))))
                  :verify-guards t  ))
  (let ((args        (cons sts ins))
        (formal-args (l-args le))
        (l-body      (l-body le)))
    (sl-ev l-body (pairlis$ formal-args args))))


;;;  Identify a set of symbols for this book.

(deftheory simplify-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'simplify-defuns-section)))
```

```
;;;  simp-with-x.lisp                              Warren A. Hunt, Jr.

; Here we define a very crude simplifier.  This simplifier is very
; similar to the dependency evaluator, and maybe this simplifier and
; dependency evaluator should be merged.

(in-package "ACL2")

(deflabel simp-with-x-defuns-section)

(defmacro xp (a) `(eq ,a 'x))

(defun wx-args-okp (a)
  (declare (xargs :guard t))
  (if (atom a)
      (or (eq a nil)
          (eq a t)
          (xp a))
    (symbol-listp a)))

(defun combine-args (a b)
  (declare (xargs :guard t))
  (if (atom a)
      (if (xp a) 'x b)
    (if (atom b)
        (if (xp b) 'x a)
      (if (and (symbol-listp a)
               (symbol-listp b))
          (set-union a b)
        'x))))

(defun wx-not (a)
  (declare (xargs :guard t))
  (if (atom a)
      (if (xp a)
          'x
        (not a))
    a))

(defun wx-and (a b)
  (declare (xargs :guard t))
  (if (atom a)
      (if (xp a)
          (if (eq b nil) nil 'x)
        (if a b nil))
    (if (atom b)
        (if (xp b)
            'x
          (if b a nil))
      (combine-args a b))))

(defun wx-and-lst (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      t
    (wx-and (car lst)
            (wx-and-lst (cdr lst)))))

(defun wx-nand-lst (lst)
  (declare (xargs :guard t))
  (wx-not (wx-and-lst lst)))

(defun wx-or (a b)
  (declare (xargs :guard t))
  (if (atom a)
      (if (xp a)
```

```
          (if (eq b t) t 'x)
        (if a t b))
    (if (atom b)
        (if (xp b)
            'x
          (if b t a))
      (combine-args a b))))

(defun wx-or-lst (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      nil
    (wx-or (car lst)
           (wx-or-lst (cdr lst)))))

(defun wx-nor-lst (lst)
  (declare (xargs :guard t))
  (wx-not (wx-or-lst lst)))

(defun wx-xor (a b)
  (declare (xargs :guard t))
  (if (atom a)
      (if (xp a) 'x
        (if a (wx-not b) b))
    (if (atom b)
        (if (xp b) 'x
          (if b (wx-not a) a))
      (if (and (symbol-listp a)
               (symbol-listp b))
          (set-union a b)
        'x))))

(defun wx-xor-lst (lst)
  (declare (xargs :guard t))
  (if (atom lst)
      nil
    (wx-xor (car lst)
            (wx-xor-lst (cdr lst)))))

(defun wx-nxor-lst (lst)
  (declare (xargs :guard t))
  (wx-not (wx-xor-lst lst)))

(defun wx-if (tst tst-1 tst-0)
  (declare (xargs :guard t))
  (if (atom tst)
      (if (xp tst) 'x
        (if tst tst-1 tst-0))
    (combine-args tst
                  (combine-args tst-1 tst-0))))

(mutual-recursion

(defun wx-ev (term alst)
  (declare (xargs :guard (and (sx-lexprp term)
                              (symbol-alistp alst))
                  :verify-guards t))
  (if (atom term)
      (cdr (assoc-eq term alst))
    (let ((fn   (car term))
          (args (cdr term)))
      (if (eq fn 'quote)
          nil
        (let ((wx-ev-args (wx-ev-lst args alst)))
          (case fn
            (list  wx-ev-args)
            (car   (if (consp (car wx-ev-args))
                       (car (car wx-ev-args)) nil))
```

```
            (cdr   (if (consp (car wx-ev-args))
                       (cdr (car wx-ev-args)) nil))
            (if    (wx-if (wx-ev (car args)   alst)
                          (wx-ev (cadr args)  alst)
                          (wx-ev (caddr args) alst)))
            (not   (wx-not (wx-ev (car args) alst)))
            (buf   (wx-ev (car args) alst))
            (and   (wx-and-lst  wx-ev-args))
            (nand  (wx-nand-lst wx-ev-args))
            (or    (wx-or-lst   wx-ev-args))
            (nor   (wx-nor-lst  wx-ev-args))
            (xor   (wx-xor-lst  wx-ev-args))
            (nxor  (wx-nxor-lst wx-ev-args))
            (otherwise nil)))))))

(defun wx-ev-lst (term-lst alst)
  (declare (xargs :guard (and (sx-lexpr-lstp term-lst)
                              (symbol-alistp alst))
                  :verify-guards t))
  (if (atom term-lst)
      nil
    (cons (wx-ev (car term-lst) alst)
          (wx-ev-lst (cdr term-lst) alst))))
)

(defun wx-eval (le ins sts)
  (declare (xargs :guard (and (sx-lambdap le)
                              (true-listp ins)
                              (= (len (cons sts ins))
                                 (len (l-args le))))
                  :verify-guards t  ))
  (let ((args        (cons sts ins))
        (formal-args (l-args le))
        (l-body      (l-body le)))
    (wx-ev l-body (pairlis$ formal-args args))))


;;;  Identify a set of symbols for this book.

(deftheory simp-with-x-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'simp-with-x-defuns-section)))
```

```
;;;  flg-eval.lisp                                Warren A. Hunt, Jr.

; The primitive evaluators...

(in-package "ACL2")

(deflabel flg-eval-defuns-section)


(defun flg-eval (flg fn ins sts)
  (declare (xargs :guard (and (symbolp flg)
                              (sx-lambdap fn)
      (true-listp ins)
      (= (len (cons sts ins))
                                 (len (l-args fn))))
                  :verify-guards t))
  (case flg
    (ins-sts     (is-eval fn ins sts))
    (wire-check  (wc-eval fn ins sts))
;    (sn          (sn-eval fn ins sts))
    (sim         (sm-eval fn ins sts))
    (dep         (dl-eval fn ins sts))
    (simplify    (sl-eval fn ins sts))
    (simp-with-x (wx-eval fn ins sts))
    (otherwise  nil)))


;;;  Identify a set of symbols for this book.

(deftheory flg-eval-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'flg-eval-defuns-section)))
```

```
;;;  measure.lisp                              Warren A. Hunt, Jr.

(in-package "ACL2")

(deflabel measure-section)

;;;  We define a measure function CONS-SIZE that we use in the
;;;  definition of our DE simulators.

(defun cons-size (x)
  (declare (xargs :guard t))
  (if (atom x)
      0
    (+ 1 (cons-size (car x)) (cons-size (cdr x)))))

(defthm cons-size-cons
  (implies (consp x)
   (and (< 0 (cons-size x))
                (< (cons-size (car x))
                   (cons-size x))
(< (cons-size (cdr x))
                   (cons-size x))
(< (cons-size (caddar x))
   (cons-size x))))
  :rule-classes :linear)

(defthm cons-size-fn-delete-assoc-eq-netlist
   (implies (assoc-eq fn netlist)
            (< (cons-size (delete-assoc-eq-netlist fn netlist))
               (cons-size netlist)))
   :rule-classes (:linear :rewrite))

#|
(defthm cons-size-no-fn-delete-assoc-eq-netlist
   (implies (and (alistp netlist)
                 (not (assoc-eq fn netlist)))
            (equal (cons-size (delete-assoc-eq-netlist fn netlist))
                   (cons-size netlist)))
   :rule-classes (:linear :rewrite))
|#

;;;  The measure function for the DE recursions.

(defun se-measure (ins-or-occurs netlist)
  (declare (xargs :guard t))
  (cons (1+ (cons-size netlist))
        (cons-size ins-or-occurs)))


(deftheory measure
  (set-difference-theories (current-theory :here)
   (current-theory 'measure-section)))
```

```
;;;  de-help.lisp                              Warren A. Hunt, Jr.

; Here are some helper functions...


; Important Issue!!!

; For now, the input to an occurrences inside a module may either be a
; natural number or a symbol.  Natural numbers are considered to be
; constants, where symbols refer to previously defined values.  We
; have some special functions that permit natural number constants to
; appear in netlists.  These next functions translate embedded
; non-symbol constants.

; NOTE!!!  The issue of having anything but symbols as input to an
; occurrence in a module needs to be thought through in its entirety.
; The ASSOC-OCC-INS-VALUES function is deeply intertwined with the
; definition of SE and DE and the entire SeqSimp system.  This is also
; the function that should extract vector values if they were to
; appear as an input in an occurrence.

(in-package "ACL2")

(deflabel de-help-section)


(defun assoc-occ-ins-values (elements alist)
  (declare (xargs :guard (and (eqlable-listp elements)
                              (alistp alist))))
  (if (endp elements)
      nil
    (cons (if (symbolp (car elements))
              ;; The value for a symbol is looked up in the
              ;; associaton list
              (assoc-eq-value (car elements) alist)
            ;;  Other constants are converted into T or NIL.
            (logbitp 0 (ifix (car elements))))

          (assoc-occ-ins-values (cdr elements) alist))))


(deftheory de-help
  (set-difference-theories (current-theory :here)
   (current-theory 'de-help-section)))
```

```
;;;  de-guard.lisp                              Warren A. Hunt, Jr.

; Here we give an operational definition for the semantics of our
; netlist langauge.

; This language is designed to represent FSMs, which are represented
; as modules.  A netlist is a list of well-formed modules.  To evaluate
; a module requires its inputs and state as well as any associated
; module definitions.

(in-package "ACL2")

(deflabel de-guard-defuns-section)


(defun ge-static (flg fn ins sts netlist)
  (declare (xargs :guard t))
  (and (symbolp flg)
       (if (consp fn)
           (sx-lambdap fn)
         (symbolp fn))
       (true-listp ins)
       (or (consp fn)
           (true-listp sts))
       (sx-netp netlist)
       (ar-netp netlist)))

(defun ge-occ-static (flg occs wire-alist sts-alist netlist)
  (declare (xargs :guard t))
  (and (symbolp flg)
       (sx-netp netlist)
       (ar-netp netlist)
       (sx-occsp occs)
       (ar-occsp occs netlist)
       (symbol-alistp wire-alist)
       (symbol-alistp sts-alist)))


; After thinking about things, I realize the guard is dynamic; that
; is, it must be checked for every initial call as the length of the
; inputs must equal the number of inputs for the module to be
; evaluated and the structure of the state must well-formed with
; respect to the module being evaluated.

; Thus, the guards for the SE and DE should be primitive independent
; except for the arity of the returned values.  The guards for SE and
; DE will look very much like these functions, but it should be
; possible to prove that many checks are unnecessary once we know that
; the arity of the netlist is OK.  That is, it should be possible to
; conclude, once the top-level call is known to be OK, that all
; subsequent calls are also OK.

; NOTE:  Many of the test below can be eliminated by using static
; netlist properties.

(mutual-recursion

(defun ge (flg fn ins sts netlist)
  (declare (xargs :measure (se-measure fn netlist)
                  :guard (ge-static flg fn ins sts netlist)
                  :verify-guards nil))
  (if (consp fn)
      (and (= (len (cons sts ins))
              (len (l-args fn)))
           (let ((primitive-ans (flg-eval flg fn ins sts)))
             (and (consp primitive-ans)    ;;  State
```

```
                (cdr primitive-ans))))   ;;  At least one value.
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
(let-names
 (m-ins m-outs m-sts m-occs) (m-body module)
         (and
          (true-listp m-ins)
          (true-listp m-sts)
          (= (len m-ins) (len ins))
          (= (len m-sts) (len sts))
          (sx-occsp m-occs)
          (let ((wire-alist  (pairlis$ m-ins ins))
                (sts-alist   (pairlis$ m-sts sts))
                (new-netlist (delete-assoc-eq-netlist fn netlist)))
            (and
             (sx-netp new-netlist)
             (ar-netp new-netlist)
             (ar-occsp m-occs new-netlist)
             (symbol-alistp wire-alist)
             (symbol-alistp sts-alist)
             (let ((new-alist (ge-occ flg m-occs wire-alist sts-alist
                                      new-netlist)))
               (and (symbol-listp m-outs)
                    (symbol-alistp new-alist)
                    (assoc-eq-values m-outs new-alist)))))))))

(defun ge-occ (flg occs wire-alist sts-alist netlist)
  (declare (xargs :measure (se-measure occs netlist)
                  :guard (ge-occ-static flg occs wire-alist sts-alist
                                        netlist)
                  :verify-guards nil))
  (if (endp occs)
      wire-alist
    (let-names
     (o-name o-outs o-fn o-ins) (car occs)
     (and (symbolp o-name)
          (symbol-listp o-outs)
          (symbolp o-fn)
          (eqlable-listp o-ins)
          (alistp wire-alist)
          (alistp sts-alist)
          (let* ((ins       (assoc-occ-ins-values o-ins  wire-alist))
                 (sts       (assoc-eq-value       o-name sts-alist)))
            (and
             (true-listp ins)
             (true-listp sts)
             (let ((result    (ge flg o-fn ins sts netlist)))
               (and
                (true-listp result)
                (true-listp o-outs)
                (= (len o-outs) (len result))
                (let ((new-pairs (pairlis$ o-outs result)))
                  (and
                   (alistp new-pairs)
                   (let ((new-wire-alist (append new-pairs wire-alist)))
                     (ge-occ flg (cdr occs)
                             new-wire-alist sts-alist netlist))))))))))))
)

(verify-guards ge)


;;;  Identify a set of symbols for this book.

(deftheory de-guard-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'de-guard-defuns-section)))
```

```
;;;  de.lisp                                  Warren A. Hunt, Jr.

; Here we give an operational definition for the semantics of our
; netlist langauge.

; This language is designed to represent FSMs, which are represented
; as modules.  A netlist is a list of well-formed modules.  To
; evaluate a module requires its inputs and state as well as the
; definition of the module, which in turn may require the definitions
; of other modules.

(in-package "ACL2")

(deflabel de-defuns-section)

; Currently, we have a static guard that is not sufficient to carry
; out the guard proofs for the functions below.  However, they do
; catch some problems, so we have left them.

; To prove the guards will be quite a bit of work.  We leave this for
; now to continue...

(mutual-recursion

(defun se (flg fn ins sts netlist)
  (declare (xargs :measure (se-measure fn netlist)
                  :guard (ge-static flg fn ins sts netlist)
                  :verify-guards nil))
  (if (consp fn)
      (cdr (flg-eval flg fn ins sts))
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
(let-names
 (m-ins m-outs m-sts m-occs) (m-body module)
 (let ((wire-alist  (pairlis$ m-ins ins))
       (sts-alist   (pairlis$ m-sts sts))
              (new-netlist (delete-assoc-eq-netlist fn netlist)))
          (assoc-eq-values
           m-outs
           (se-occ flg m-occs wire-alist sts-alist new-netlist))))))))

(defun se-occ (flg occs wire-alist sts-alist netlist)
  (declare (xargs :measure (se-measure occs netlist)
                  :guard (ge-occ-static flg occs wire-alist sts-alist
                                        netlist)
                  :verify-guards nil))
  (if (endp occs)
      wire-alist
    (let-names
     (o-name o-outs o-fn o-ins) (car occs)
     (let* ((ins      (assoc-occ-ins-values o-ins  wire-alist))
            (sts      (assoc-eq-value       o-name sts-alist))
            (new-wire-alist
             (append
              (pairlis$ o-outs (se flg o-fn ins sts netlist))
              wire-alist)))
       (se-occ flg (cdr occs) new-wire-alist sts-alist netlist)))))
)

(mutual-recursion

(defun de (flg fn ins sts netlist)
  (declare (xargs :measure (se-measure fn netlist)
                  :guard (ge-static flg fn ins sts netlist)
                  :verify-guards nil))
```

```
  (if (consp fn)
      (car (flg-eval flg fn ins sts))
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
(let-names
 (m-ins m-sts m-occs) (m-body module)
        (let* ((wire-alist     (pairlis$ m-ins ins))
               (sts-alist      (pairlis$ m-sts sts))
               (new-netlist    (delete-assoc-eq-netlist fn netlist))
               (all-wire-alist (se-occ flg m-occs wire-alist
                                       sts-alist new-netlist)))
          (assoc-eq-values
           m-sts
           (de-occ flg m-occs all-wire-alist sts-alist
                   new-netlist)))))))

(defun de-occ (flg occs wire-alist sts-alist netlist)
  (declare (xargs :measure (se-measure occs netlist)
                  :guard (ge-occ-static flg occs wire-alist sts-alist
                                        netlist)
                  :verify-guards nil))
  (if (endp occs)
      sts-alist
    (let-names
     (o-name o-fn o-ins) (car occs)
    (let* ((ins      (assoc-occ-ins-values o-ins  wire-alist))
           (sts      (assoc-eq-value       o-name sts-alist))
           (new-sts-alist
            (cons (cons o-name (de flg o-fn ins sts netlist))
                  sts-alist)))
      (de-occ flg (cdr occs) wire-alist new-sts-alist netlist)))))
)

(defun de-sim (flg fn ins-list sts netlist)
  (if (atom ins-list)
      sts
    (de-sim flg fn (cdr ins-list)
            (de flg fn (car ins-list) sts netlist)
            netlist)))

(defun de-sim-sts-outs (flg fn ins-list sts netlist)
  (let ((ins-last (last ins-list))
        (sts-last (de-sim flg fn ins-list sts netlist)))
    (list (se flg fn ins-last sts-last netlist)
          sts-last)))

;;;  Some test functions.


(defun de-sim-list (flg fn ins-list sts netlist)
  (if (atom ins-list)
      (list sts)
    (cons sts
          (de-sim-list flg fn (cdr ins-list)
                       (de flg fn (car ins-list) sts netlist)
                       netlist))))

;;;  Identify a set of symbols for this book.

(deftheory de-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'de-defuns-section)))
```

```
;;;  collect.lisp                              Warren A. Hunt, Jr.

; To collect statistics about our hardware circuits we have defined a
; mutually recursive set of functions that only return a single answer.

(in-package "ACL2")

(deflabel collect-defuns-section)

(mutual-recursion

(defun sec (flg fn netlist)
  (declare (xargs :measure (se-measure fn netlist)
                  :verify-guards nil))
  (if (consp fn)
      0
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          0
(let-names
 (m-type m-occs) (m-body module)
         (if (member-eq m-type '(primitive defined-primitive))
             (cdr (assoc-eq flg (m-body module)))
           (sec-occ flg m-occs (delete-assoc-eq-netlist fn netlist))))))))

(defun sec-occ (flg occs netlist)
  (declare (xargs :measure (se-measure occs netlist)
                  :verify-guards nil))
  (if (endp occs)
      0
    (let-names
     (o-fn) (car occs)
     (+ (sec flg o-fn netlist)
        (sec-occ flg (cdr occs) netlist)))))
)


;;;  Identify a set of symbols for this book.

(deftheory collect-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'collect-defuns-section)))
```

```
;  library-help.lisp                          Warren A. Hunt, Jr.

; Here are some definitions and lemmas used to help define the
; ASIC libraries.


(in-package "ACL2")

(deflabel library-help-defuns-section)


(defun update-alist (name value alst)
  (declare (xargs :guard (and (symbolp name)
                              (symbol-alistp alst))))
  (if (atom alst)
      nil
    (if (eq (caar alst) name)
        (acons name value (cdr alst))
      (cons (car alst)
            (update-alist name value (cdr alst))))))

(defun merge-data (existing-data new-data)
  (declare (xargs :guard (and (symbol-alistp existing-data)
                              (symbol-alistp new-data))))
  (if (atom new-data)
      existing-data
    (merge-data (update-alist (caar new-data)
                              (cdar new-data)
                              existing-data)
                (cdr new-data))))

(defun make-primitives (root type library new-primitives basic-prims)
  (declare (xargs :guard (and (symbolp root)
                              (symbolp type)
                              (symbolp library)
                              (symbol-alistp-symbol-alistp new-primitives)
                              (symbol-alistp-symbol-alistp basic-prims))))
  (if (atom new-primitives)
      nil
    (acons (caar new-primitives)
           (merge-data (cdr (assoc-eq root basic-prims))
                       (cons (cons 'type type)
                             (cons (cons 'library library)
                                   (cdr (car new-primitives)))))
           (make-primitives root
                            type
                            library
                            (cdr new-primitives)
                            basic-prims))))


;;;  Identify a set of symbols for this book.

(deftheory library-help-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'library-help-defuns-section)))
```

```
;  basic-prims.lisp                          Warren A. Hunt, Jr.

; After thinking about how to make the primitive evaluator, I have
; decided to use lambda expressions.  The lambda evaluator always
; returns both the next state and the outputs.  For a lambda
; evaluation, the state is always supplied as the first argument, and
; the next state is always returned as the first element of a list.
; The state argument can be as complex as necessary, but the lambda
; evaluator must return a state even if the primitive does not
; contain state.

; The function ``ACL2::PSEUDO-TERMP'' is used to help recognize
; well-formed function calls with additional restrictions that I
; impose.  These function calls appear in occurrences.

; The primitives here are (in some sense) the lowest level primitives.
; Library elements will likely be defined in terms of these
; primitives, but need not be.  Here we defiine basic primitives, and
; we define some macros for easing the definition of defining the
; primitives for a library.

(in-package "ACL2")

(bvl basic-primitives
     '((AND2     (type . primitive)
                 (library . basic)
                 (ins  a b)
                 (sts  )
                 (outs z)
                 (two-input-gates . 1)
                 ;; I need to understand the issue of depth and how
                 ;; it relates to a primitive.  Contrast this "depth"
                 ;; with the expression for the AO21 primitive!!!
                 (depth (1+ (max a b)))
                 (occs
                  (st (z) (lambda (s x y)
                            (list s (and x y)))
                      (a b)))
;                  (simplfications
;                   (implies (or (null a)
;                                (null b))
;                            (equal (occ-name (z) 'AND2 (a b))
;                                   (occ-name (z) 'the-constant-false ())))
;                   (implies (and (eq a t)
;                                 (consp b))
;                            (equal (occ-name (z) 'AND2 (a b))
;                                   (occ-name (z) 'RENAME (b))))
;                   (implies (and (consp a)
;                                 (eq b t))
;                            (equal (occ-name (z) 'AND2 (a b))
;                                   (occ-name (z) 'RENAME (a)))))
                 )

       (AND3     (type . primitive)
                 (library . basic)
                 (ins  a b c)
                 (sts  )
                 (outs z)
                 (two-input-gates . 2)
                 (occs
                  (st (z) (lambda (s x y z)
                            (list s (and x y z)))
                      (a b c))))

       (AND4     (type . primitive)
                 (library . basic)
```

```
          (ins  a b c d)
          (sts  )
          (outs z)
          (two-input-gates . 3)
          (occs
           (st (z) (lambda (s w x y z)
                     (list s (and w x y z)))
               (a b c d))))

(AO21     (type . primitive)
          (library . basic)
          (ins  a1 a2 b)
          (sts  )
          (outs z)
          (two-input-gates . 2)
          (depth (1+ (max (1+ (max a1 a2)) b)))
          (occs
           (st (z) (lambda (s a1 a2 b)
                     (list s (or (and a1 a2) b)))
               (a1 a2 b))))

(AO22     (type . primitive)
          (library . basic)
          (ins  a1 a2 b1 b2)
          (sts  )
          (outs z)
          (two-input-gates . 3)
          (occs
           (st (z) (lambda (s a1 a2 b1 b2)
                     (list s (or (and a1 a2) (and b1 b2))))
               (a1 a2 b1 b2))))

(AO222    (type . primitive)
          (library . basic)
          (ins  a1 a2 b1 b2 c1 c2)
          (sts  )
          (outs z)
          (two-input-gates . 5)
          (occs
           (st (z) (lambda (s a1 a2 b1 b2 c1 c2)
                     (list s (or (and a1 a2) (and b1 b2) (and c1 c2))))
               (a1 a2 b1 b2 c1 c2))))

(AO2222   (type . primitive)
          (library . basic)
          (ins  a1 a2 b1 b2 c1 c2 d1 d2)
          (sts  )
          (outs z)
          (two-input-gates . 7)
          (occs
           (st (z) (lambda (s a1 a2 b1 b2 c1 c2 d1 d2)
                     (list s (or (and a1 a2) (and b1 b2)
                                 (and c1 c2) (and d1 d2))))
               (a1 a2 b1 b2 c1 c2 d1 d2))))

(AOI21    (type . primitive)
          (library . basic)
          (ins  a1 a2 b)
          (sts  )
          (outs z)
          (two-input-gates . 2)
          (occs
           (st (z) (lambda (s a1 a2 b)
                     (list s (nor (and a1 a2) b)))
               (a1 a2 b))))

(AOI22    (type . primitive)
          (library . basic)
```

```
          (ins  a1 a2 b1 b2)
          (sts  )
          (outs z)
          (two-input-gates . 3)
          (occs
           (st (z) (lambda (s a1 a2 b1 b2)
                     (list s (nor (and a1 a2) (and b1 b2))))
               (a1 a2 b1 b2))))

(AOI222   (type . primitive)
          (library . basic)
          (ins  a1 a2 b1 b2 c1 c2)
          (sts  )
          (outs z)
          (two-input-gates . 5)
          (occs
           (st (z) (lambda (s a1 a2 b1 b2 c1 c2)
                     (list s (nor (and a1 a2) (and b1 b2) (and c1 c2))))
               (a1 a2 b1 b2 c1 c2))))

(AOI2222  (type . primitive)
          (library . basic)
          (ins  a1 a2 b1 b2 c1 c2 d1 d2)
          (sts  )
          (outs z)
          (two-input-gates . 7)
          (occs
           (st (z) (lambda (s a1 a2 b1 b2 c1 c2 d1 d2)
                     (list s (nor (and a1 a2) (and b1 b2)
                                  (and c1 c2) (and d1 d2))))
               (a1 a2 b1 b2 c1 c2 d1 d2))))

(AOI33    (type . primitive)
          (library . basic)
          (ins  a1 a2 a3 b1 b2 b3)
          (sts  )
          (outs z)
          (two-input-gates . 5)
          (occs
           (st (z) (lambda (s a1 a2 a3 b1 b2 b3)
                     (list s (nor (and a1 a2 a3) (and b1 b2 b3))))
               (a1 a2 a3 b1 b2 b3))))

(BUFFER   (type . primitive)
          (library . basic)
          (ins  a)
          (sts  )
          (outs z)
          (two-input-gates . 2)
          (occs
           (st (z) (lambda (s x)
                     (list s (buf x)))
               (a))))

(CLK      (type . primitive)
          (library . basic)
          (ins  a)
          (outs z)
          (sts)
          (info (clock-driver . t))
          (two-input-gates . 2)
          (occs
           (st (z) (lambda (s x)
                     (list s (not (not x))))
               (a))))

;;  Clock Pulse Generator
(CLKCHP   (type . primitive)
```

```
          (library . basic)
          (ins  enn osc eni gate)
          (outs z)
          (sts)
          (info (pulse-generator . t))
          (two-input-gates . 9)
          (occs
           (st (z) (lambda (s enn osc eni gate)
                     (list s (buf (nand
                                   (and enn (not osc))
                                   (nand
                                    (or (buf (not osc)) ; buf is Delay
                                        (not eni))
                                    gate)))))
               (enn osc eni gate))))

(CLKI     (type . primitive)
          (library . basic)
          (ins  a)
          (outs z)
          (sts)
          (info (clock-driver . t))
          (two-input-gates . 3)
          (occs
           (st (z) (lambda (s x)
                     (list s (not (not (not x)))))
               (a))))

;;  Clock Splitter
(CLKSPC   (type . primitive)
          (library . basic)
          (ins  pg1 b c en osc)
          (outs zb zc)
          (sts)
          (two-input-gates . 9)
          (occs
           (st (y z)
               (lambda (s pg1 b c en osc)
                 (list s
                       (not (nand (nand c (nand osc en)) b))
                       (not (nand pg1
                                  (not (nand c
                                             (nand osc en)))))))
               (pg1 b c en osc))))

;;  Clock Splitter
(CLKSPL   (type . primitive)
          (library . basic)
          (ins a b c en osc)
          (outs zb zc)
          (sts)
          (two-input-gates . 10)
          (occs
           (st (y z)
               (lambda (s a b c en osc)
                 (list s
                       (not (nand (nand c (nand osc en)) b))
                       (not (not (nand (not a)
                                       (nand c
                                             (nand osc en)))))))
               (a b c en osc))))

(COMP2    (type . primitive)
          (library . basic)
          (ins  a1 a2 b1 b2)
          (sts  )
          (outs z)
          (two-input-gates . 6)
```

```
          (occs
           (st (z) (lambda (s a1 a2 b1 b2)
                     (list s (and (xnor a1 b1) (xnor a2 b2))))
               (a1 a2 b1 b2))))


(DELAY4   (type . primitive)
          (library . basic)
          (ins  a)
          (sts  )
          (outs z)
          (info (delay . t))
          (two-input-gates . 4)
          (occs
           (st (z) (lambda (s x)
                     (list s (buf x)))
               (a))))

(DELAY6   (type . primitive)
          (library . basic)
          (ins  a)
          (sts  )
          (outs z)
          (info (delay . t))
          (two-input-gates . 6)
          (occs
           (st (z) (lambda (s x)
                     (list s (buf x)))
               (a))))


(INVERT   (type . primitive)
          (library . basic)
          (ins  a)
          (sts  )
          (outs z)
          (two-input-gates . 1)
          (occs
           (st (z) (lambda (s x)
                     (list s (not x)))
               (a))))

(LMX0001  (type . primitive)
          (library . basic)
          (ins  a b c d0 d1 i sd)
          (outs l2)
          (sts s)
          (two-input-gates . 10)
          (occs      ;;;  This needs to be checked!!!
           (st (l2) (lambda (s a b c d0 d1 i sd)
                      (list i s))
               (a b c d0 d1 i sd))))

(LPH0001  (type . primitive)
          (library . basic)
          (ins  a b c d i)
          (outs l2)
          (sts s)
          (two-input-gates . 10)
          (occs      ;;;  This needs to be checked!!!
           (st (l2) (lambda (s a b c d i)
                      (list i s))
               (a b c d i))))

(LPH0101  (type . primitive)
          (library . basic)
          (ins  a b c d i)
          (outs l1 l2)
```

```
          (sts s)
          (two-input-gates . 10)
          (occs      ;;;  This needs to be checked!!!
           (st (l1 l2) (lambda (s a b c d i)
                         (list i s s))
               (a b c d i))))

(LPH4001  (type . primitive)
          (library . basic)
          (ins  a b c d i r1n)
          (outs l2)
          (sts s)
          (two-input-gates . 10)
          (occs      ;;;  This is wrong!!!
           (st (l2) (lambda (s a b c d i r1n)
                      (list i s))
               (a b c d i r1n))))

(LTL0001  (type . primitive)
          (library . basic)
          (ins  a b c d e i)
          (outs l2)
          (sts s)
          (two-input-gates . 10)
          (occs      ;;;  This needs to be checked!!!
           (st (l2) (lambda (s a b c d e i)
                      (list i s))
               (a b c d e i))))

(MUX21    (type . primitive)
          (library . basic)
          (ins sd d0 d1)
          (outs z)
          (sts)
          (two-input-gates . 2)
          (occs
           (st (z) (lambda (s sd d0 d1)
                     (list s (buf (if sd d1 d0))))
               (sd d0 d1))))

(MUX21I   (type . primitive)
          (library . basic)
          (ins sd d0 d1)
          (outs z)
          (sts)
          (two-input-gates . 2)
          (occs
           (st (z) (lambda (s sd d0 d1)
                     (list s (not (if sd d1 d0))))
               (sd d0 d1))))

(MUX41    (type . primitive)
          (library . basic)
          (ins sd1 sd2 d0 d1 d2 d3)
          (outs z)
          (sts)
          (two-input-gates . 5)
          (occs
           (st (z) (lambda (s sd1 sd2 d0 d1 d2 d3)
                     (list s (if sd1
                                 (if sd2 d3 d2)
                                 (if sd2 d1 d0))))
               (sd1 sd2 d0 d1 d2 d3))))

(NAND2    (type . primitive)
          (library . basic)
          (ins  a b)
          (sts  )
```

```
          (outs z)
          (two-input-gates . 1)
          (occs
           (st (z) (lambda (s x y)
                     (list s (not (and x y))))
               (a b))))

(NAND3    (type . primitive)
          (library . basic)
          (ins a b c)
          (outs z)
          (sts)
          (two-input-gates . 2)
          (occs
           (st (z) (lambda (s x y z)
                     (list s (not (and x y z))))
               (a b c))))

(NAND4    (type . primitive)
          (library . basic)
          (ins a b c d)
          (outs z)
          (sts)
          (two-input-gates . 3)
          (occs
           (st (z) (lambda (s w x y z)
                     (list s (not (and w x y z))))
               (a b c d))))

(NOR2     (type . primitive)
          (library . basic)
          (ins  a b)
          (sts  )
          (outs z)
          (two-input-gates . 1)
          (occs
           (st (z) (lambda (s x y)
                     (list s (not (or x y))))
               (a b))))

(NOR3     (type . primitive)
          (library . basic)
          (ins  a b c)
          (sts  )
          (outs z)
          (two-input-gates . 2)
          (occs
           (st (z) (lambda (s x y z)
                     (list s (not (or x y z))))
               (a b c))))

(NOR4     (type . primitive)
          (library . basic)
          (ins  a b c d)
          (sts  )
          (outs z)
          (two-input-gates . 3)
          (occs
           (st (z) (lambda (s w x y z)
                     (list s (not (or w x y z))))
               (a b c d))))

(OA21     (type . primitive)
          (library . basic)
          (ins  a1 a2 b)
          (sts  )
          (outs z)
          (two-input-gates . 2)
```

```
         (occs
          (st (z) (lambda (s a1 a2 b)
                    (list s (and (or a1 a2) b)))
              (a1 a2 b))))

(OA22    (type . primitive)
         (library . basic)
         (ins  a1 a2 b1 b2)
         (sts  )
         (outs z)
         (two-input-gates . 3)
         (occs
          (st (z) (lambda (s a1 a2 b1 b2)
                    (list s (and (or a1 a2) (or b1 b2))))
              (a1 a2 b1 b2))))

(OA222   (type . primitive)
         (library . basic)
         (ins  a1 a2 b1 b2 c1 c2)
         (sts  )
         (outs z)
         (two-input-gates . 5)
         (occs
          (st (z) (lambda (s a1 a2 b1 b2 c1 c2)
                    (list s (and (or a1 a2) (or b1 b2) (or c1 c2))))
              (a1 a2 b1 b2 c1 c2))))

(OA2222  (type . primitive)
         (library . basic)
         (ins  a1 a2 b1 b2 c1 c2 d1 d2)
         (sts  )
         (outs z)
         (two-input-gates . 7)
         (occs
          (st (z) (lambda (s a1 a2 b1 b2 c1 c2 d1 d2)
                    (list s (and (or a1 a2) (or b1 b2)
                                 (or c1 c2) (or d1 d2))))
              (a1 a2 b1 b2 c1 c2 d1 d2))))

(OAI21   (type . primitive)
         (library . basic)
         (ins  a1 a2 b)
         (sts  )
         (outs z)
         (two-input-gates . 2)
         (occs
          (st (z) (lambda (s a1 a2 b)
                    (list s (nand (or a1 a2) b)))
              (a1 a2 b))))

(OAI22   (type . primitive)
         (library . basic)
         (ins  a1 a2 b1 b2)
         (sts  )
         (outs z)
         (two-input-gates . 3)
         (occs
          (st (z) (lambda (s a1 a2 b1 b2)
                    (list s (nand (or a1 a2) (or b1 b2))))
              (a1 a2 b1 b2))))

(OAI222  (type . primitive)
         (library . basic)
         (ins  a1 a2 b1 b2 c1 c2)
         (sts  )
         (outs z)
         (two-input-gates . 5)
         (occs
```

```
          (st (z) (lambda (s a1 a2 b1 b2 c1 c2)
                    (list s (nand (or a1 a2) (or b1 b2) (or c1 c2))))
              (a1 a2 b1 b2 c1 c2))))

(OAI2222  (type . primitive)
          (library . basic)
          (ins  a1 a2 b1 b2 c1 c2 d1 d2)
          (sts  )
          (outs z)
          (two-input-gates . 7)
          (occs
           (st (z) (lambda (s a1 a2 b1 b2 c1 c2 d1 d2)
                     (list s (nand (or a1 a2) (or b1 b2)
                                   (or c1 c2) (or d1 d2))))
               (a1 a2 b1 b2 c1 c2 d1 d2))))

(OAI33    (type . primitive)
          (library . basic)
          (ins  a1 a2 a3 b1 b2 b3)
          (sts  )
          (outs z)
          (two-input-gates . 5)
          (occs
           (st (z) (lambda (s a1 a2 a3 b1 b2 b3)
                     (list s (nand (or a1 a2 a3) (or b1 b2 b3))))
               (a1 a2 a3 b1 b2 b3))))



(OR2      (type . primitive)
          (library . basic)
          (ins  a b)
          (sts  )
          (outs z)
          (two-input-gates . 1)
          (occs
           (st (z) (lambda (s x y)
                     (list s (or x y)))
               (a b))))

(OR3      (type . primitive)
          (library . basic)
          (ins  a b c)
          (sts  )
          (outs z)
          (two-input-gates . 2)
          (occs
           (st (z) (lambda (s x y z)
                     (list s (or x y z)))
               (a b c))))

(OR4      (type . primitive)
          (library . basic)
          (ins  a b c d)
          (sts  )
          (outs z)
          (two-input-gates . 3)
          (occs
           (st (z) (lambda (s w x y z)
                     (list s (or w x y z)))
               (a b c d))))

(TERM     (type . primitive)
          (library . basic)
          (ins a)
          (outs)
          (sts)
          (dep)
```

```
                (two-input-gates . 1)
                (occs
                 (st () (lambda (s x)
                          (list s))
                     (a))))

       (XNOR2    (type . primitive)
                 (library . basic)
                 (ins  a b)
                 (sts  )
                 (outs z)
                 (two-input-gates . 2)
                 (occs
                  (st (z) (lambda (s x y)
                            (list s (xnor x y)))
                      (a b))))

       (XNOR3    (type . primitive)
                 (library . basic)
                 (ins  a b c)
                 (sts  )
                 (outs z)
                 (two-input-gates . 4)
                 (occs
                  (st (z) (lambda (s x y z)
                            (list s (xnor x (xor y z))))
                      (a b c))))

       (XOR2     (type . primitive)
                 (library . basic)
                 (ins  a b)
                 (sts  )
                 (outs z)
                 (two-input-gates . 2)
                 (occs
                  (st (z) (lambda (s x y)
                            (list s (xor x y)))
                      (a b))))

       (XOR3     (type . primitive)
                 (library . basic)
                 (ins  a b c)
                 (sts  )
                 (outs z)
                 (two-input-gates . 4)
                 (occs
                  (st (z) (lambda (s x y z)
                            (list s (xor x (xor y z))))
                      (a b c))))

       ))

;;;  Example exhaustive test for AND2 primitive.

(list (let ((inputs (list nil nil)))
        (equal (se 'sim 'and2 inputs nil (@ basic-primitives))
               (list (and-lst inputs))))
      (let ((inputs (list nil t  )))
        (equal (se 'sim 'and2 inputs nil (@ basic-primitives))
               (list (and-lst inputs))))
      (let ((inputs (list t   nil)))
        (equal (se 'sim 'and2 inputs nil (@ basic-primitives))
               (list (and-lst inputs))))
      (let ((inputs (list t   t  )))
        (equal (se 'sim 'and2 inputs nil (@ basic-primitives))
               (list (and-lst inputs)))))

(bvl defined-primitives
```

```
    '(
      ;; Below is the primitive that defines the one-bit flip-flop.
      ;; It is unusual because the SE or DE evaluation of the lambda
      ;; expression defining this module, does not satisfy the usual
      ;; requirement that the SE or DE STS argument is always a list,
      ;; which is, indeed, the point.

      ;; Another reference to this point can be found in the SE-GUARD
      ;; function, where this guard does not require the STS argument
      ;; to be a list (in fact be anything) when the FN argument is a
      ;; lambda (CONS) expression.

      (F1       (type . defined-primitive)
                (library . basic)
                (ins  in)
                (sts  st)
                (outs z)
                (two-input-gates . 10)
                (occs
                 (st (z) (lambda (s i)
                           (list i s))
                     (in))))

      ;; The RENAME defined primitive is used to replace Verilog
      ;; continuous assignment statements.

      (RENAME   (type . defined-primitive)
                (library . basic)
                (ins  a)
                (outs z)
                (sts)
                (two-input-gates . 0)
                (occs
                 (st (z) (lambda (s i)
                           (list s (buf i)))
                     (a)))))
    )

(bvl all-primitives (append (@ defined-primitives)
                            (@ basic-primitives)))

(sx-netp (@ all-primitives))
(ar-netp (@ all-primitives))

(list (equal (se 'sim 'f1
                 (list t)
                 (list nil)
                 (@ defined-primitives))
             (list nil))

      (equal (se 'sim 'f1
                 (list nil)
                 (list t)
                 (@ defined-primitives))
             (list t))

      (equal (de 'sim 'f1
                 (list t)
                 (list nil)
                 (@ defined-primitives))
             (list t))

      (equal (de 'sim 'f1
                 (list nil)
                 (list t)
                 (@ defined-primitives))
             (list nil)))
```

```
(list (equal (se 'dep 'f1
                 (list '(a))
                 (list '(s))
                 (@ defined-primitives))
             '((s)))

      (equal (de 'dep 'f1
                 (list '(a))
                 (list '(s))
                 (@ defined-primitives))
             '((a))))
```

```
;  net-test.lisp                              Warren A. Hunt, Jr.

;;;  Test circuit to see if embedded constants work.

(bvl and-gate
     '((test-and
        (type . module)
        (ins  a b)
        (sts)
        (outs q r s t u v w x y)
        (occs
         (occq (q) and2 (0 0))
         (occr (r) and2 (0 1))
         (occs (s) and2 (1 0))
         (occt (t) and2 (1 1))

         (occ0 (u) and2 (a b))
         (occ1 (v) and2 (0 b))
         (occ2 (w) and2 (1 b))
         (occ3 (x) and2 (a 0))
         (occ4 (y) and2 (a 1))))))

(bvl and-gate-netlist
     (module-union-list
      (list (@ and-gate)
            (list (assoc 'and2 (@ all-primitives))))))

(list (sx-netp (@ and-gate-netlist))
      (ar-netp (@ and-gate-netlist)))

(equal (se 'sim 'test-and (list nil t) nil (@ and-gate-netlist))
       (list nil nil nil t nil nil t nil nil))
```

```
;  net-74181.lisp                          Warren A. Hunt, Jr.

; Netlist definition of the SN74181, 4-bit, 16-function ALU.

(bvl sn74181
     '((SN74181
        (type . module)
        (ins  |\c~|
              |a0| |a1| |a2| |a3|
              |b0| |b1| |b2| |b3|
              |m|
              |s0| |s1| |s2| |s3| )

        (sts )
        (outs |f0| |f1| |f2| |f3|
              |\cout~|
              |x| |y|
              |\a=b| )
        (wires  |w0|  |w1|  |w2|  |w3|  |w4|  |w5|  |w6|  |w7|  |w8|  |w9|
               |w10| |w11| |w12| |w13| |w14| |w15| |w16| |w17| |w18| |w19|
               |w20| |w21| |w22| |w23| |w24| |w25| |w26| |w27| |w28| |w29|
               |w30| |w31| |w32| |w33| |w34| |w35| |w36| |w37| |w38| |w39|
               |w40| |w41| |w42| |w43| |w44| |w45| |w46| |w47| |w48| |w49|
               |w50| |w51| |w52| |w53| |w54| |w55| |w56| |w57| |w58| |w59|
               |w60| |w61| |w62| |w63| )

        (occs
         ( |occ_0|  ( |w0|)     INVERT (  |m|))
         ( |occ_1|  ( |w1|)     INVERT ( |b0|))
         ( |occ_2|  ( |w2|)     INVERT ( |b1|))
         ( |occ_3|  ( |w3|)     INVERT ( |b2|))
         ( |occ_4|  ( |w4|)     INVERT ( |b3|))
         ( |occ_5|  ( |w5|)     BUFFER ( |a0|))
         ( |occ_6|  ( |w6|)     AND2   ( |b0|  |s0|))
         ( |occ_7|  ( |w7|)     AND2   ( |s1|  |w1|))
         ( |occ_8|  ( |w8|)     AND3   ( |w1|  |s2|  |a0|))
         ( |occ_9|  ( |w9|)     AND3   ( |a0|  |s3|  |b0|))
         (|occ_10|  (|w10|)     BUFFER ( |a1|))
         (|occ_11|  (|w11|)     AND2   ( |b1|  |s0| ))
         (|occ_12|  (|w12|)     AND2   ( |s1|  |w2| ))
         (|occ_13|  (|w13|)     AND3   ( |w2|  |s2|  |a1|))
         (|occ_14|  (|w14|)     AND3   ( |a1|  |s3|  |b1|))
         (|occ_15|  (|w15|)     BUFFER ( |a2|))
         (|occ_16|  (|w16|)     AND2   ( |b2|  |s0|))
         (|occ_17|  (|w17|)     AND2   ( |s1|  |w3|))
         (|occ_18|  (|w18|)     AND3   ( |w3|  |s2|  |a2|))
         (|occ_19|  (|w19|)     AND3   ( |a2|  |s3|  |b2|))
         (|occ_20|  (|w20|)     BUFFER ( |a3|))
         (|occ_21|  (|w21|)     AND2   ( |b3|  |s0|))
         (|occ_22|  (|w22|)     AND2   ( |s1|  |w4|))
         (|occ_23|  (|w23|)     AND3   ( |w4|  |s2|  |a3|))
         (|occ_24|  (|w24|)     AND3   ( |a3|  |s3|  |b3|))
         (|occ_25|  (|w25|)     NOR3   ( |w5|  |w6|  |w7|))
         (|occ_26|  (|w26|)     NOR2   ( |w8|  |w9|))
         (|occ_27|  (|w27|)     NOR3   (|w10| |w11| |w12|))
         (|occ_28|  (|w28|)     NOR2   (|w13| |w14|))
         (|occ_29|  (|w29|)     NOR3   (|w15| |w16| |w17|))
         (|occ_30|  (|w30|)     NOR2   (|w18| |w19|))
         (|occ_31|  (|w31|)     NOR3   (|w20| |w21| |w22|))
         (|occ_32|  (|w32|)     NOR2   (|w23| |w24|))
         (|occ_33|  (|w33|)     XOR2   (|w25| |w26|))
         (|occ_34|  (|w34|)     XOR2   (|w27| |w28|))
         (|occ_35|  (|w35|)     XOR2   (|w29| |w30|))
         (|occ_36|  (|w36|)     XOR2   (|w31| |w32|))
         (|occ_37|  (|w37|)     NAND2  ( |w0| |\c~|))
         (|occ_38|  (|w38|)     AND2   ( |w0| |w25|))
```

```
        (|occ_39|  (|w39|)     AND3   ( |w0| |w26|  |\c~|))
        (|occ_40|  (|w40|)     AND2   ( |w0| |w27|))
        (|occ_41|  (|w41|)     AND3   ( |w0| |w25| |w28|))
        (|occ_42|  (|w42|)     AND4   ( |w0| |w28| |w26|  |\c~| ))
        (|occ_43|  (|w43|)     AND2   ( |w0| |w29|))
        (|occ_44|  (|w44|)     AND3   ( |w0| |w27| |w30|))
        (|occ_45|  (|w45|)     AND4   ( |w0| |w25| |w30| |w28|))
        (|occ_46a| (|w46a|)    AND4   ( |w0| |w30| |w28| |w26|))
        (|occ_46|  (|w46|)     AND2   (|w46a| |\c~|))
        (|occ_47|  (|w47|)     NAND4  (|w26| |w28| |w30| |w32|))
        (|occ_48a| (|w48a|)    AND4   (|w26| |w28| |w30| |w32|))
        (|occ_48|  (|w48|)     NAND2  (|w48a| |\c~|))
        (|occ_49|  (|w49|)     AND4   (|w25| |w28| |w30| |w32|))
        (|occ_50|  (|w50|)     AND3   (|w27| |w30| |w32|))
        (|occ_51|  (|w51|)     AND2   (|w29| |w32|))
        (|occ_52|  (|w52|)     BUFFER (|w31|))

        (|renm_0|  (|w53|)     RENAME (|w37|))

        (|occ_54|  (|w54|)     NOR2   (|w38| |w39|))
        (|occ_55|  (|w55|)     NOR3   (|w40| |w41| |w42|))
        (|occ_56|  (|w56|)     NOR4   (|w43| |w44| |w45| |w46|))
        (|occ_57|  (|w57|)     NOR4   (|w49| |w50| |w51| |w52|))
        (|occ_58|  (|w58|)     XOR2   (|w53| |w33|))
        (|occ_59|  (|w59|)     XOR2   (|w54| |w34|))
        (|occ_60|  (|w60|)     XOR2   (|w55| |w35|))
        (|occ_61|  (|w61|)     XOR2   (|w56| |w36|))
        (|occ_62a| (|w62a|)    INVERT (|w48|))
        (|occ_62b| (|w62b|)    INVERT (|w57|))
        (|occ_62|  (|w62|)     OR2    (|w62a| |w62b|))
        (|occ_63|  (|w63|)     AND4   (|w58| |w59| |w60| |w61|))

        (|renm_1|  (|f0|)      RENAME (|w58|))
        (|renm_2|  (|f1|)      RENAME (|w59|))
        (|renm_3|  (|f2|)      RENAME (|w60|))
        (|renm_4|  (|f3|)      RENAME (|w61|))
        (|renm_5|  (|\a=b|)    RENAME (|w63|))
        (|renm_6|  (|x|)       RENAME (|w47|))
        (|renm_7|  (|\cout~|)  RENAME (|w62|))
        (|renm_8|  (|y|)       RENAME (|w57|))))
      ))

(bvl net-74181 (module-union (@ sn74181) (@ all-primitives)))

(list (sx-netp (@ net-74181))
      (ar-netp (@ net-74181)))
```

```
;  net-74181-tests.lisp                   Warren A. Hunt, Jr.

;  We will now define some functions that specify the 74181 ALU.  We
;  first define two helper functions, and then we define each output
;  bit as a function.

(defun f74181-q0 (a b s0 s1 s2 s3)
  (declare (ignore s0 s1)
           (xargs :guard t))
  (nor (and b s3 a)
       (and a s2 (not b))))

(defun f74181-q1 (a b s0 s1 s2 s3)
  (declare (ignore s2 s3)
           (xargs :guard t))
  (nor (and (not b) s1)
       (and s0 b)
       a))

(defun f74181-f0 (cin a0 b0 m s0 s1 s2 s3)
  (declare (xargs :guard t))
  (let ((m~ (not m)))
    (xor (nand cin m~)
         (xor (f74181-q0 a0 b0 s0 s1 s2 s3)
              (f74181-q1 a0 b0 s0 s1 s2 s3)))))

(defun f74181-f1 (cin a0 a1 b0 b1 m s0 s1 s2 s3)
  (declare (xargs :guard t))
  (let ((m~ (not m)))
    (xor (nor (and m~ (f74181-q1 a0 b0 s0 s1 s2 s3))
              (and m~ (f74181-q0 a0 b0 s0 s1 s2 s3) cin))
         (xor (f74181-q1 a1 b1 s0 s1 s2 s3)
              (f74181-q0 a1 b1 s0 s1 s2 s3)))))

(defun f74181-f2 (cin a0 a1 a2 b0 b1 b2 m s0 s1 s2 s3)
  (declare (xargs :guard t))
  (let ((m~ (not m)))
    (xor (nor (and m~ (f74181-q1 a1 b1 s0 s1 s2 s3))
              (and m~
                   (f74181-q1 a0 b0 s0 s1 s2 s3)
                   (f74181-q0 a1 b1 s0 s1 s2 s3))
              (and m~
                   (f74181-q0 a0 b0 s0 s1 s2 s3)
                   (f74181-q0 a1 b1 s0 s1 s2 s3)
                   cin))
         (xor (f74181-q1 a2 b2 s0 s1 s2 s3)
              (f74181-q0 a2 b2 s0 s1 s2 s3)))))

(defun f74181-f3 (cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (declare (xargs :guard t))
  (let ((m~ (not m)))
    (xor (nor (and m~ (f74181-q1 a2 b2 s0 s1 s2 s3))
              (and m~
                   (f74181-q1 a1 b1 s0 s1 s2 s3)
                   (f74181-q0 a2 b2 s0 s2 s2 s3))
              (and m~
                   (f74181-q1 a0 b0 s0 s1 s2 s3)
                   (f74181-q0 a1 b1 s0 s1 s2 s3)
                   (f74181-q0 a2 b2 s0 s1 s2 s3))
              (and m~
                   (f74181-q0 a0 b0 s0 s1 s2 s3)
                   (f74181-q0 a1 b1 s0 s1 s2 s3)
                   (f74181-q0 a2 b2 s0 s1 s2 s3)
                   cin))
         (xor (f74181-q1 a3 b3 s0 s1 s2 s3)
              (f74181-q0 a3 b3 s0 s1 s2 s3)))))
```

```
;;;  The "extra outputs"of the 74181.

(defun f74181-a=b (cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (declare (xargs :guard t))
  (and (f74181-f0 cin a0 b0 m s0 s1 s2 s3)
       (f74181-f1 cin a0 a1 b0 b1 m s0 s1 s2 s3)
       (f74181-f2 cin a0 a1 a2 b0 b1 b2 m s0 s1 s2 s3)
       (f74181-f3 cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)))

(defun f74181-prop~ (cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (declare (ignore cin m)
           (xargs :guard t))
  (nand (f74181-q0 a0 b0 s0 s1 s2 s3)
        (f74181-q0 a1 b1 s0 s1 s2 s3)
        (f74181-q0 a2 b2 s0 s1 s2 s3)
        (f74181-q0 a3 b3 s0 s1 s2 s3)))

(defun f74181-gen~ (cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (declare (ignore cin m)
           (xargs :guard t))
  (nor (and (f74181-q0 a3 b3 s0 s1 s2 s3)
            (f74181-q0 a2 b2 s0 s1 s2 s3)
            (f74181-q0 a1 b1 s0 s1 s2 s3)
            (f74181-q1 a0 b0 s0 s1 s2 s3))
       (and (f74181-q0 a3 b3 s0 s1 s2 s3)
            (f74181-q0 a2 b2 s0 s1 s2 s3)
            (f74181-q1 a1 b1 s0 s1 s2 s3))
       (and (f74181-q0 a3 b3 s0 s1 s2 s3)
            (f74181-q1 a2 b2 s0 s1 s2 s3))
       (f74181-q1 a3 b3 s0 s1 s2 s3)))

(defun f74181-cout~ (cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (declare (xargs :guard t))
  (or (not (f74181-gen~ cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3))
      (not (nand (not (f74181-prop~ cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3))
                 cin))))

;;;  We define several output functions with multiple outputs

(defun f74181-fout (cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (declare (xargs :guard t))
  (list (f74181-f0    cin a0 b0 m s0 s1 s2 s3)
        (f74181-f1    cin a0 a1 b0 b1 m s0 s1 s2 s3)
        (f74181-f2    cin a0 a1 a2 b0 b1 b2 m s0 s1 s2 s3)
        (f74181-f3    cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)))

(defthm boolean-listp-f74181-fout
  (boolean-listp (f74181-fout cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3))
  :hints (("Goal" :in-theory
           (disable f74181-f0 f74181-f1 f74181-f2 f74181-f3)))))

(defun f74181-sumout (cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (declare (xargs :guard t))
  (list (f74181-f0    cin a0 b0 m s0 s1 s2 s3)
        (f74181-f1    cin a0 a1 b0 b1 m s0 s1 s2 s3)
        (f74181-f2    cin a0 a1 a2 b0 b1 b2 m s0 s1 s2 s3)
        (f74181-f3    cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
        (f74181-cout~ cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)))

(defthm boolean-listp-f74181-sumout
  (boolean-listp (f74181-sumout cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3))
  :hints (("Goal" :in-theory
           (disable f74181-f0 f74181-f1 f74181-f2 f74181-f3 f74181-cout~))))

(defun f74181-out (cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (declare (xargs :guard t))
```

```
  (list (f74181-f0    cin a0 b0 m s0 s1 s2 s3)
        (f74181-f1    cin a0 a1 b0 b1 m s0 s1 s2 s3)
        (f74181-f2    cin a0 a1 a2 b0 b1 b2 m s0 s1 s2 s3)
        (f74181-f3    cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
        (f74181-cout~ cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
        (f74181-prop~ cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
        (f74181-gen~  cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
        (f74181-a=b   cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)))

(defthm boolean-listp-f74181-out
  (boolean-listp (f74181-out cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3))
  :hints (("Goal" :in-theory
           (disable f74181-f0 f74181-f1 f74181-f2 f74181-f3
                    f74181-cout~ f74181-prop~ f74181-gen~ f74181-a=b))))

(defun sn74181 (args)
  (declare (xargs :guard (true-listp args)))
  (let ((cin (nth  0 args))
        (a0  (nth  1 args))
        (a1  (nth  2 args))
        (a2  (nth  3 args))
        (a3  (nth  4 args))
        (b0  (nth  5 args))
        (b1  (nth  6 args))
        (b2  (nth  7 args))
        (b3  (nth  8 args))
        (m   (nth  9 args))
        (s0  (nth 10 args))
        (s1  (nth 11 args))
        (s2  (nth 12 args))
        (s3  (nth 13 args)))
  (f74181-out cin a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)))


; Below are some functions that check the definitons of the 74181.

(defun test-sn74181 (inputs netlist)
  (equal (se 'sim 'sn74181 inputs nil netlist)
         (sn74181 inputs)))

(defun test-sn74181-n (n netlist)
  (if (zp n)
      (test-sn74181 (nat-to-v n 14) netlist)
    (and (test-sn74181 (nat-to-v n 14) netlist)
         (test-sn74181-n (1- n) netlist))))

(defun test-sn74181-n-list (n netlist)
  (if (zp n)
      nil
    (cons (test-sn74181 (nat-to-v n 14) netlist)
          (test-sn74181-n-list (1- n) netlist))))

(defun help-sn74181 (inputs netlist)
  (list (se 'sim 'sn74181 inputs nil netlist)
        (sn74181 inputs)))


(defun test-sn74181-nth (index inputs netlist)
  (equal (nth index (se 'sim 'sn74181 inputs nil netlist))
         (nth index (sn74181 inputs))))

(defun test-sn74181-n-nth (n index netlist)
  (if (zp n)
      (test-sn74181-nth index (nat-to-v n 14) netlist)
    (and (test-sn74181-nth index (nat-to-v n 14) netlist)
         (test-sn74181-n-nth (1- n) index netlist))))

:comp t
```

```
; (test-sn74181-n (1- (expt 2 14)) (@ net-74181))


#|
; The stuff below was supposed to speed the simulation by precomputing
; the values used by the simulator instead of always calling NAT-TO-V.
; However, it is only a few percent faster than the simple things done
; above.

; What a waste of time.  Sigh.

; The "fast" tests (333 MHz Pentium II Overdrive).

; ACL2>(time (tst-sn74181-n f74181-inputs (1- (expt 2 14)) (@ net-74181)))
; real time : 68.600 secs
; run time  : 64.370 secs
; T

; ACL2>(time (acl2_*1*_acl2::tst-sn74181-n f74181-inputs (1- (expt 2 14)) (@ net-74181)))
; real time : 300.770 secs
; run time  : 295.010 secs
; T

; The "slow" tests (333 MHz Pentium II Overdrive).

; ACL2>(time (test-sn74181-n (1- (expt 2 14)) (@ net-74181)))
; real time : 71.050 secs
; run time  : 66.260 secs
; T

; ACL2>(time (acl2_*1*_acl2::test-sn74181-n (1- (expt 2 14)) (@ net-74181)))
; real time : 303.770 secs
; run time  : 296.500 secs
; T


; This next function is what I wanted, but it's not tail recursive.

;(defun cons-lst (a lst)
;  (declare (xargs :guard (true-listp lst)))
;  (if (endp lst)
;      nil
;    (acons a
;           (car lst)
;           (cons-lst a (cdr lst)))))

(defun rev-help (x acc)
  (declare (xargs :guard (and (true-listp x)
                              (true-listp acc))))
  (if (endp x)
      acc
    (rev-help (cdr x) (cons (car x) acc))))

(defthm true-listp-rev-help
  (implies (and (true-listp x)
                (true-listp acc))
           (true-listp (rev-help x acc))))

(defun rev (x)
  (declare (xargs :guard (true-listp x)))
  (rev-help x nil))

(defthm true-listp-ref
  (implies (true-listp x)
           (true-listp (rev x))))

(in-theory (disable rev))
```

```
(defun cons-lst-tailrec (a lst acc)
  (declare (xargs :guard (and (true-listp lst)
                              (alistp acc))))
  (if (endp lst)
      (rev acc)
    (cons-lst-tailrec a (cdr lst) (acons a (car lst) acc))))

(defthm true-listp-cons-lst-tailrec
  (implies (true-listp acc)
           (true-listp (cons-lst-tailrec a lst acc))))

(defun cons-lst (a lst)
  (declare (xargs :guard (true-listp lst)))
  (cons-lst-tailrec a lst nil))

(defthm true-listp-cons-lst
  (true-listp (cons-lst a lst)))

(in-theory (disable cons-lst))

(defthm true-listp-append
  (implies (true-listp y)
           (true-listp (append x y))))

(defun bv-all-n (n)
  (declare (xargs :guard (natp n)
                  :verify-guards nil))
  (if (zp n)
      nil
    (if (= n 1)
        (list (list nil)
              (list t))
      (let ((bv-all-n-1 (bv-all-n (1- n))))
        (append (cons-lst nil bv-all-n-1)
                (cons-lst  t  bv-all-n-1))))))

(defthm true-listp-bv-all-n
  (true-listp (bv-all-n n)))

(verify-guards bv-all-n)


(defconst *bv-width-74181-inputs* 14)
(defconst *number-of-inputs* (expt 2 *bv-width-74181-inputs*))

(defstobj f74181-inputs
  (element  :type      (array t (16384))
            :initially nil))

(defun initialize-f74181-input-array-help (f74181-inputs n initial-values)
  (declare (xargs :guard (and (natp n)
                              (<= 0 n)
                              (<= n 16384)
                              (true-listp initial-values)
                              (f74181-inputsp f74181-inputs))
                  :verify-guards nil
                  :stobjs (f74181-inputs)
                  :measure (acl2-count initial-values)))
  (if (or (atom initial-values)
          (not (<= 0 n))
          (not (< n 16384)))
      f74181-inputs
    (let ((f74181-inputs (update-elementi n (car initial-values) f74181-inputs)))
      (initialize-f74181-input-array-help f74181-inputs
                                          (1+ n)
                                          (cdr initial-values)))))
```

```
(defun initialize-f74181-input-array (f74181-inputs)
  (declare (xargs :guard (f74181-inputsp f74181-inputs)
                  :verify-guards nil
                  :stobjs (f74181-inputs)))
  (let ((initial-values (bv-all-n 14)))
    (if (not (true-listp initial-values))
        f74181-inputs
      (initialize-f74181-input-array-help f74181-inputs
                                          0
                                          initial-values))))))

(defun tst-sn74181-n (f74181-inputs n netlist)
  (declare (xargs :stobjs (f74181-inputs)))
  (if (zp n)
      (test-sn74181 (elementi n f74181-inputs) netlist)
    (and (test-sn74181 (elementi n f74181-inputs) netlist)
         (tst-sn74181-n f74181-inputs (1- n) netlist))))


:comp t

; (initialize-f74181-input-array f74181-inputs)

; (tst-sn74181-n f74181-inputs (1- (expt 2 14)) (@ net-74181))

|#
```

```
;  net-tests.lisp                          Warren A. Hunt, Jr.


(bvl test-circuit-2
     '((tst-2
        (type . module)
        (ins  |\c~|
              |a0| |a1| |a2| |a3|
              |b0| |b1| |b2| |b3|
              |m|
              |s0| |s1| |s2| |s3| )

        (sts )

        (outs |f0| |f1| |f2| |f3|
              |\cout~|
              |x| |y|
              |\a=b| )

        (occs
         (occ1 (|f0| |f1| |f2| |f3|
                |\cout~|
                |x| |y|
                |\a=b|)
               sn74181
               (|\c~|
                |a0| |a1| |a2| |a3|
                |b0| |b1| |b2| |b3|
                |m|
                |s0| |s1| |s2| |s3|))))))

(bvl test-circuit-2-m-0
     '((tst-2-m-0
        (type . module)
        (ins  |\c~|
              |a0| |a1| |a2| |a3|
              |b0| |b1| |b2| |b3|
              |m|
              |s0| |s1| |s2| |s3| )

        (sts )

        (outs |f0| |f1| |f2| |f3|
              |\cout~|
              |x| |y|
              |\a=b| )

        (occs
         (occ1 (|f0| |f1| |f2| |f3|
                |\cout~|
                |x| |y|
                |\a=b|)
               sn74181
               (|\c~|
                |a0| |a1| |a2| |a3|
                |b0| |b1| |b2| |b3|
                0
                |s0| |s1| |s2| |s3|))))))

(bvl test-circuit-2-m-1
     '((tst-2-m-1
        (type . module)
        (ins  |\c~|
              |a0| |a1| |a2| |a3|
              |b0| |b1| |b2| |b3|
              |m|
```

```
              |s0| |s1| |s2| |s3| )

        (sts )

        (outs |f0| |f1| |f2| |f3|
              |\cout~|
              |x| |y|
              |\a=b| )

        (occs
         (occ1 (|f0| |f1| |f2| |f3|
                |\cout~|
                |x| |y|
                |\a=b|)
               sn74181
               (|\c~|
                |a0| |a1| |a2| |a3|
                |b0| |b1| |b2| |b3|
                1
                |s0| |s1| |s2| |s3|))))))

(bvl net-test-circuit-2
     (module-union-list
      (list (@ test-circuit-2)
            (@ sn74181)
            (@ all-primitives))))

(bvl net-test-circuit-2-m-0
     (module-union-list
      (list (@ test-circuit-2-m-0)
            (@ sn74181)
            (@ all-primitives))))

(bvl net-test-circuit-2-m-1
     (module-union-list
      (list (@ test-circuit-2-m-1)
            (@ sn74181)
            (@ all-primitives))))

(list (sx-netp (@ net-test-circuit-2))
      (ar-netp (@ net-test-circuit-2)))

(list (sx-netp (@ net-test-circuit-2-m-0))
      (ar-netp (@ net-test-circuit-2-m-0)))

(list (sx-netp (@ net-test-circuit-2-m-1))
      (ar-netp (@ net-test-circuit-2-m-1)))

(bvl tst-ins-2-m-nil (list t  nil nil nil nil  t t t t  nil  t t t t))
(bvl tst-ins-2-m-t   (list t  nil nil nil nil  t t t t  t    t t t t))



;;; Works as they should.

(list
 (equal
  (se 'sim 'tst-2     (@ tst-ins-2-m-nil) nil (@ net-test-circuit-2))
  (se 'sim 'tst-2-m-0 (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2-m-0)))

 (equal
  (se 'sim 'tst-2     (@ tst-ins-2-m-nil) nil (@ net-test-circuit-2))
  (se 'sim 'tst-2-m-0 (@ tst-ins-2-m-nil) nil (@ net-test-circuit-2-m-0)))

 (equal
  (se 'sim 'tst-2     (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2))
  (se 'sim 'tst-2-m-1 (@ tst-ins-2-m-nil) nil (@ net-test-circuit-2-m-1)))
```

```
 (equal
  (se 'sim 'tst-2     (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2))
  (se 'sim 'tst-2-m-1 (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2-m-1)))

 (equal
  (se 'sim 'tst-2-m-0 (@ tst-ins-2-m-nil) nil (@ net-test-circuit-2-m-0))
  (se 'sim 'tst-2-m-0 (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2-m-0)))

 (equal
  (se 'sim 'tst-2-m-1 (@ tst-ins-2-m-nil) nil (@ net-test-circuit-2-m-1))
  (se 'sim 'tst-2-m-1 (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2-m-1))))


;;; Properply doesn't work.

(list
 (not
  (equal
   (se 'sim 'tst-2     (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2))
   (se 'sim 'tst-2-m-0 (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2-m-0))))

 (not
  (equal
   (se 'sim 'tst-2     (@ tst-ins-2-m-nil) nil (@ net-test-circuit-2))
   (se 'sim 'tst-2-m-1 (@ tst-ins-2-m-t  ) nil (@ net-test-circuit-2-m-1)))))



;;; Shouldn't work, but does.



;;; Should work, but doesn't.
```

```
;  net-74182.lisp                          Warren A. Hunt, Jr.

; Netlist definition of the SN74182, look-ahead carry generator.

(bvl sn74182
     '((SN74182
        (type . module)
        (ins |\c~| |x0| |x1| |x2| |x3| |y0| |y1| |y2| |y3|)
        (sts )
        (outs |\cn+x~| |\cn+y~| |\cn+z~| |x| |y|)
        (wires  |w0|  |w1|  |w2|  |w3|  |w4|  |w5|  |w6|  |w7|  |w8|  |w9|
               |w10| |w11| |w12| |w13| |w14| |w15| |w16| |w17| |w18|)
        (occs
         ( |occ_0|  (|w0|)     INVERT (|\c~|))
         ( |occ_1|  (|w1|)     AND2   ( |y0|  |x0|))
         ( |occ_2|  (|w2|)     AND2   ( |w0|  |y0|))
         ( |occ_3|  (|w3|)     AND2   ( |y1|  |x1|))
         ( |occ_4|  (|w4|)     AND3   ( |y0|  |y1|  |x0|))
         ( |occ_5|  (|w5|)     AND3   ( |w0|  |y0|  |y1|))
         ( |occ_6|  (|w6|)     AND2   ( |y2|  |x2|))
         ( |occ_7|  (|w7|)     AND3   ( |y1|  |y2|  |x1|))
         ( |occ_8|  (|w8|)     AND4   ( |y0|  |y1|  |y2|  |x0|))
         ( |occ_9|  (|w9|)     AND4   ( |w0|  |y0|  |y1|  |y2|))
         (|occ_10| (|w10|)     AND2   ( |y3|  |x3|))
         (|occ_11| (|w11|)     AND3   ( |y2|  |y3|  |x2|))
         (|occ_12| (|w12|)     AND4   ( |y1|  |y2|  |y3|  |x1|))
         (|occ_13| (|w13|)     AND4   ( |y3|  |y2|  |y1|  |y0|))
         (|occ_14| (|w14|)     OR4    ( |x0|  |x1|  |x2|  |x3|))
         (|occ_15| (|w15|)     NOR2   ( |w1|  |w2|))
         (|occ_16| (|w16|)     NOR3   ( |w3|  |w4|  |w5|))
         (|occ_17| (|w17|)     NOR4   ( |w6|  |w7|  |w8|  |w9|))
         (|occ_18| (|w18|)     OR4    (|w10| |w11| |w12| |w13|))
         (|renm_0| (|\cn+x~|)  RENAME (|w15|))
         (|renm_1| (|\cn+y~|)  RENAME (|w16|))
         (|renm_2| (|\cn+z~|)  RENAME (|w17|))
         (|renm_3| (|y|)       RENAME (|w18|))
         (|renm_4| (|x|)       RENAME (|w14|))))
       ))

(bvl net-74182 (module-union (@ sn74182) (@ all-primitives)))

(sx-netp (@ net-74182))
(ar-netp (@ net-74182))
```

```
;  net-74182-tests.lisp                    Warren A. Hunt, Jr.

;  We will now define some functions that specify the 74182 look-ahead
;  carry generator.

(defun sn74182 (args)
  (declare (xargs :guard (true-listp args)))
  (let ((c~ (nth 0 args))
        (x0 (nth 1 args))
        (x1 (nth 2 args))
        (x2 (nth 3 args))
        (x3 (nth 4 args))
        (y0 (nth 5 args))
        (y1 (nth 6 args))
        (y2 (nth 7 args))
        (y3 (nth 8 args)))

    (list (nor (and y0 (not c~))
               (and x0 y0))
          (nor (and y0 y1 (not c~))
               (and x0 y0 y1)
               (and x1 y1))
          (nor (and y0 y1 y2 (not c~))
               (and x0 y0 y1 y2)
               (and x1 y1 y2)
               (and x2 y2))
          (or x0 x1 x2 x3)
          (or  (and y0 y1 y2 y3)
               (and x1 y1 y2 y3)
               (and x2 y2 y3)
               (and x3 y3)))))

(defun test-sn74182 (inputs netlist)
  (equal (se 'sim 'sn74182 inputs nil netlist)
         (sn74182 inputs)))

(defun test-sn74182-n (n netlist)
  (if (zp n)
      (test-sn74182 (nat-to-v n 9) netlist)
    (and (test-sn74182 (nat-to-v n 9) netlist)
         (test-sn74182-n (1- n) netlist))))

:comp t

; (time (test-sn74182-n 511 (@ net-74182)))


#|

(se 'sim 'sn74182
    (list t t t t t t t t t)
    nil
    (@ net-74182))

(se 'sim 'sn74182
    (list nil nil nil nil nil nil nil nil nil)
    nil
    (@ net-74182))

(se 'dep 'sn74182
     (listify-elements (m-ins (m-body (assoc-eq 'sn74182 (@ net-74182)))))
     nil
     (@ net-74182))

(de 'dep 'sn74182
     (listify-elements (m-ins (m-body (assoc-eq 'sn74182 (@ net-74182)))))
```

```
     nil
     (@ net-74182))

(se 'simplify 'sn74182
    '((t)
      (nil)
      (|x1|)
      (|x2|)
      (|x3|)
      (|y0|)
      (|y1|)
      (|y2|)
      (|y3|))
    nil
    (@ net-74182))

(se 'simp-with-x 'sn74182
    '(t
      nil
      (|x1|)
      (|x2|)
      (|x3|)
      (|y0|)
      (|y1|)
      (|y2|)
      (|y3|))
    nil
    (@ net-74182))

(test-sn74182-n 511 (@ net-74182))

|#
```

```
;  net-74175.lisp                              Warren A. Hunt, Jr.

; Netlist definition of a four-bit register -- a simplified version
; of the SN74175.

(bvl sn74175
     '((SN74175
        (type . module)
        (ins   |x0|  |x1|  |x2|  |x3|)
        (sts   |s0|  |s1|  |s2|  |s3|)
        (outs  |y0|  |y1|  |y2|  |y3|)
        (wires  )
        (occs
         ( |s0|  (|y0|)     F1     ( |x0| ))
         ( |s1|  (|y1|)     F1     ( |x1| ))
         ( |s2|  (|y2|)     F1     ( |x2| ))
         ( |s3|  (|y3|)     F1     ( |x3| ))))))

(bvl net-74175
     (module-union (@ sn74175)
                   (@ all-primitives)))

(sx-netp (@ net-74175))
(ar-netp (@ net-74175))

(list (equal (se 'sim 'SN74175
                 (list t t t t)
                 (list (list nil) (list nil) (list nil) (list nil))
                 (@ net-74175))
             (list nil nil nil nil))

      (equal (se 'sim 'SN74175
                 (list nil nil nil nil)
                 (list (list t) (list t) (list t) (list t))
                 (@ net-74175))
             (list t t t t))

      (equal (de 'sim 'SN74175
                 (list t t t t)
                 (list (list nil) (list nil) (list nil) (list nil))
                 (@ net-74175))
             (list (list t) (list t) (list t) (list t)))

      (equal (de 'sim 'SN74175
                 (list nil nil nil nil)
                 (list (list t) (list t) (list t) (list t))
                 (@ net-74175))
             (list (list nil) (list nil) (list nil) (list nil))))

(se 'dep 'SN74175
    (listify-elements '(a b c d))
    (listify-elements '((w) (x) (y) (z)))
    (@ net-74175))

(de 'dep 'SN74175
    (listify-elements '(a b c d))
    (listify-elements '((w) (x) (y) (z)))
    (@ net-74175))
```

```
;  net-circuit.lisp                          Warren A. Hunt, Jr.


;;;  NOTES:  ar-netp did not catch that I was missing the correct
;;;          arguments for the STS argument.

(bvl test-circuit
     '((tst
        (type . module)
        (ins  |b0| |b1| |b2| |b3|
              |m|
              |s0| |s1| |s2| |s3| )
        (sts  |reg0| |reg1| |ff-1| )
        (outs |sf0| |sf1| |sf2| |sf3|
              |\scout~|
              |sx| |sy|
              |\sa=b| )
        (wires |f0| |f1| |f2| |f3|
               |\cout~|
               |x| |y|
               |\a=b| )
        (occs
         (|reg0|  (|sf0| |sf1| |sf2| |sf3|)
                  sn74175
                  (|f0|  |f1| |f2| |f3|))

         (|reg1|  (|\scout~| |sx| |sy| |\sa=b|)
                  sn74175
                  (|\cout~| |x| |y| |\a=b|))

         (|ff-1|  (|\c~|) f1 (|f3|))

         (|alu|   (|f0| |f1| |f2| |f3| |\cout~| |x| |y| |\a=b|)
                  sn74181
                  (|\c~|
                   |sf0| |sf1| |sf2| |sf3|
                   |b0| |b1| |b2| |b3|
                   |m|
                   |s0| |s1| |s2| |s3| ))))))

(bvl net-test-circuit
     (module-union-list
      (list (@ test-circuit)
            (@ sn74175)
            (@ sn74181)
            (@ sn74182)
            (@ all-primitives))))


;;;  Need to check from here on...

(list (sx-netp (@ net-test-circuit))
      (ar-netp (@ net-test-circuit)))


(bvl tst-ins (list nil nil nil nil nil t t t t))




(bvl tst-ins
     ;;     b0  b1  b2  b3   m  s0  s1  s2  s3
     (list nil nil nil nil nil   t nil nil nil))

(bvl tst-ins-b-free
     ;;      b0    b1    b2    b3    m  s0  s1  s2  s3
```

```
     (list '(b0) '(b1) '(b2) '(b3) nil   t nil nil nil))


(bvl tst-sts
     (list
      ;;           sf0        sf1        sf2       sf3
      (list (list  'x) (list nil) (list nil) (list nil))
      ;;       \scout~       |sx|       |sy|    |\sa=b|
      (list (list nil) (list nil) (list nil) (list nil))
      ;;     c~
      (list nil)))

(bvl tst-sts-with-x
     (list
      ;;           sf0        sf1        sf2       sf3
      (list (list  'x) (list nil) (list nil) (list nil))
      ;;       \scout~       |sx|       |sy|    |\sa=b|
      (list (list nil) (list nil) (list nil) (list nil))
      ;;     c~
      (list 'x)))




(se 'sim 'tst (@ tst-ins) (@ tst-sts) (@ net-test-circuit))
(de 'sim 'tst (@ tst-ins) (@ tst-sts) (@ net-test-circuit))

(bvl tst-ins-list-1  (make-list  1 :initial-element (@ tst-ins)))
(bvl tst-ins-list-10 (make-list 10 :initial-element (@ tst-ins)))

(bvl tst-ins-b-free-list-10 (make-list 10 :initial-element (@ tst-ins-b-free)))


(de-sim-list 'sim 'tst (@  tst-ins-list-1) (@ tst-sts) (@ net-test-circuit))
(de-sim-list 'sim 'tst (@ tst-ins-list-10) (@ tst-sts) (@ net-test-circuit))

(de-sim-sts-outs 'sim 'tst (@  tst-ins-list-1) (@ tst-sts) (@ net-test-circuit))
(de-sim-sts-outs 'sim 'tst (@ tst-ins-list-10) (@ tst-sts) (@ net-test-circuit))


(equal
 (de-sim-sts-outs 'simp-with-x 'tst
                  (@ tst-ins-list-10)
                  (@ tst-sts)
                  (@ net-test-circuit))

 (de-sim-sts-outs 'simp-with-x 'tst
                  (@ tst-ins-b-free-list-10)
                  (@ tst-sts)
                  (@ net-test-circuit)))

(de-sim-list 'simp-with-x 'tst
             (@ tst-ins-list-10)
             (@ tst-sts-with-x)
             (@ net-test-circuit))


(bvl stst-ins
     (append (listify-elements
              (list '|a0| '|a1| '|a2| '|a3|))
             (list nil t nil nil nil)))

(bvl stst-sts (list (list (list nil)
                          (list nil)
                          (list nil)
                          (list nil))
                    (list (list nil)
                          (list nil)
```

```
                        (list nil)
                        (list nil))
                   (list nil)))

(bvl stst-ins-list-10 (make-list 10 :initial-element (@ stst-ins)))


(de-sim-list 'simp-with-x 'tst
             (@ stst-ins-list-10) (@ stst-sts) (@ net-test-circuit))

(sec 'two-input-gates 'sn74182 (@ net-test-circuit))
(sec 'two-input-gates 'sn74181 (@ net-test-circuit))
(sec 'two-input-gates 'tst (@ net-test-circuit))
```

```
;;;  raw-accessors.lisp                    Warren A. Hunt, Jr.

; These macros are just accessors for separating the raw syntax in
; pieces.

(in-package "ACL2")

;(include-book "help")

(deflabel raw-accessors-defuns-section)


;;;  Accessors for a module and its pieces...


(defmacro mod-id        (x)  (declare (xargs :guard t))  `(car ,x))
(defmacro mod-name      (x)  (declare (xargs :guard t))  `(cadr ,x))
(defmacro mod-ports     (x)  (declare (xargs :guard t))  `(caddr ,x))
(defmacro mod-items     (x)  (declare (xargs :guard t))  `(cadddr ,x))

(defmacro item-type     (x)  (declare (xargs :guard t))  `(car ,x))

(defmacro net-type      (x)  (declare (xargs :guard t))  `(car ,x))
(defmacro net-strength  (x)  (declare (xargs :guard t))  `(cadr ,x))
(defmacro net-range     (x)  (declare (xargs :guard t))  `(caddr ,x))
(defmacro net-ids       (x)  (declare (xargs :guard t))  `(cdddr ,x))

(defmacro inst-type     (x)  (declare (xargs :guard t))  `(cadr ,x))
(defmacro inst-refs     (x)  (declare (xargs :guard t))  `(cddr ,x))
(defmacro inst-occ-name (x)  (declare (xargs :guard t))  `(car ,x))
(defmacro inst-bindings (x)  (declare (xargs :guard t))  `(cdr ,x))


;;;  Identify a set of symbols for this book.

(deftheory raw-accessors-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'raw-accessors-defuns-section)))
```

```
;;;  raw-syntax.lisp                          Warren A. Hunt, Jr.

;;;  This book defines the acceptable syntax for the Verilog input.

;;;  We now write a recognizer for a well-formed, parsed Verilog
;;;  modules in the subset used by IBM to specify their ASIC library
;;;  elements.  Here we check obvious syntatic properties.

;;;  The syntax recognized by this set of recognizers will later be
;;;  transformed slightly to permit constraints to be associated with
;;;  the netlist.  Before making the transformation, we check the
;;;  "raw" syntax of the netlist as produced by the Verilog-to-Lisp
;;;  (vl2lisp) parser.

;;;  After checking the raw syntax, we will transform the netlist into
;;;  similar form where is suitable for our constraint propagation
;;;  system.  This form is very similar to the original form, but
;;;  permits the inclusion of constraints.  After making this
;;;  transformation, we conduct additional semantic checks, e.g., for
;;;  connectivity.

(in-package "ACL2")
; (include-book "raw-accessors")

(deflabel raw-syntax-defuns-section)

;;;  A Verilog module is used as a representation of a non-primitive
;;;  FSM.  It is composed of a name and a number of module items.  Our
;;;  recognizer does not recognize the complete Verilog language, but
;;;  a subset of the Verilog used by IBM ASIC library elements.

;;;  Below we make clear what we allow for Verilog identifiers.

(defun vrsx-identifier-char-okp (chr)
  (declare (xargs :guard t))
  (and (characterp chr)
       (member-eql chr *standard-chars*)
       (not (member-eql chr '(#\Newline #\Space)))))

(defun vrsx-identifier-first-char-okp (chr)
  (declare (xargs :guard t))
  (and (characterp chr)
       (alpha-char-p chr)))

(defun vrsx-identifier-char-str-okp (str pos)
  (declare (xargs :guard (and (stringp str)
                              (natp pos)
                              (< pos (length str)))))
  (if (zp pos)
      t
    (and (member-eql (char str pos) *standard-chars*)
         (not (member-eql (char str pos) '(#\Newline #\Space)))
         (vrsx-identifier-char-str-okp str (1- pos)))))

(defun vrsx-identifier-chars-okp (sym)
  (declare (xargs :guard (symbolp sym)))
  (let* ((char-str     (symbol-name sym))
         (len-char-str (length char-str)))
    (and (< 0 len-char-str)
         (if (eql (char char-str 0) #\\)
             (vrsx-identifier-first-char-okp (char char-str 0))
           t)
         (vrsx-identifier-char-str-okp char-str (1- len-char-str)))))

(defun vrsx-identifier-okp (name)
  (declare (xargs :guard t))
```

```
  (and (symbolp name)
       (not (eq name nil))
       (not (eq name t))
       (vrsx-identifier-chars-okp name)))

(defthm vrsx-identifier-okp-forward-to-symbolp
  (implies (vrsx-identifier-okp name)
           (symbolp name))
  :rule-classes :forward-chaining)

(defun vrsx-identifier-list-okp (names)
  (declare (xargs :guard t))
  (if (atom names)
      (eq names nil)
    (and (vrsx-identifier-okp (car names))
         (vrsx-identifier-list-okp (cdr names)))))

(defthm vrsx-identifier-list-okp-forward-to-symbol-listp
  (implies (vrsx-identifier-list-okp names)
           (symbol-listp names))
  :rule-classes :forward-chaining)

(in-theory (disable vrsx-identifier-okp))

#|

;;;  Below is a more generalized from of an identifier that allows
;;;  an identifier to be of the form (<id> <natural-number>).  Our
;;;  Verilog parser can produce this, but we don't currently allow it.

;;;  NOTE:  This is a serious issue.  There are a number of circuits
;;;  that contain busses that our current system does not recognize.
;;;  This will need to be corrected, but this will cause a number of
;;;  changes.

(defun vrsx-id-okp (name)
  (declare (xargs :guard t))
  (if (atom name)
      (vrsx-identifier-okp name)
    (and (consp (cdr name))
         (eq (cddr name) nil)
         (vrsx-identifier-okp name)
         (natp (cadr name)))))

(defun vrsx-id-list-okp (names)
  (declare (xargs :guard t))
  (if (atom names)
      (eq names nil)
    (and (vrsx-id-okp (car names))
         (vrsx-id-list-okp (cdr names)))))

|#

;;;  Currently, an expression is either a Verilog identifier or a
;;;  numeric constant.

;;;  NOTE:  This will need to be generalized.

(defun vrsx-expr-okp (expr)
  (declare (xargs :guard t))
  (or (vrsx-identifier-okp expr)
      (and (natp expr)
           (or (= expr 0)
               (= expr 1)))))

(defun vrsx-expr-list-okp (exprs)
  (declare (xargs :guard t))
  (if (atom exprs)
```

```
      (eq exprs nil)
    (and (vrsx-expr-okp (car exprs))
         (vrsx-expr-list-okp (cdr exprs)))))


;;;  We now have a collection of syntactic recognizers for different
;;;  syntatic patterns of identifiers and expressions

(defun vrsx-identifier-expr-okp (id-expr)
  (declare (xargs :guard t))
  (and (consp id-expr)
       (consp (cdr id-expr))
       (eq (cddr id-expr) nil)
       (vrsx-identifier-okp (car id-expr))
       (vrsx-expr-okp (cadr id-expr))))

(defun vrsx-identifier-expr-list-okp (lst-of-id-expr)
  (declare (xargs :guard t))
  (if (atom lst-of-id-expr)
      (eq lst-of-id-expr nil)
    (and (vrsx-identifier-expr-okp (car lst-of-id-expr))
         (vrsx-identifier-expr-list-okp (cdr lst-of-id-expr)))))


(defun vrsx-occ-name-and-args-okp (module-reference)
  (declare (xargs :guard t))
  (and (consp module-reference)
       (symbolp (inst-occ-name module-reference))
       (vrsx-identifier-expr-list-okp (inst-bindings module-reference))))


(defun vrsx-occ-name-and-args-list-okp (module-references)
  (declare (xargs :guard t))
  (if (atom module-references)
      (eq module-references nil)
    (and (consp (car module-references))
         (vrsx-occ-name-and-args-okp (car module-references))
         (vrsx-occ-name-and-args-list-okp (cdr module-references)))))


;;;  For each module item we have a recognizer.  We first define
;;;  some helper functions for recognizing net declarations.

(defun vrsx-net-strength-okp (net-strength)
  (declare (xargs :guard t))
  (natp net-strength))

(defun vrsx-net-range-okp (range)
  (declare (xargs :guard t))
  (or (eq range nil)
      (and (consp range)
           (consp (cdr range))
           (eq (cddr range) nil)
           (natp (car range))
           (natp (cadr range)))))

(defun vrsx-net-args-okp (arguments)
  (declare (xargs :guard t))
  (and (consp arguments)
       (consp (cdr arguments))
       (consp (cddr arguments))
       (vrsx-net-range-okp (car arguments))
       (vrsx-net-strength-okp (cadr arguments))
       (vrsx-identifier-list-okp (cddr arguments))))

(defun vrsx-mod-item-input-declaration-okp (module-input-item)
  (declare (xargs :guard t))
  (and (consp module-input-item)
```

```
       (vrsx-net-args-okp (cdr module-input-item))))

(defun vrsx-mod-item-output-declaration-okp (module-output-item)
  (declare (xargs :guard t))
  (and (consp module-output-item)
       (vrsx-net-args-okp (cdr module-output-item))))

(defun vrsx-mod-item-wire-declaration-okp (module-wire-item)
  (declare (xargs :guard t))
  (and (consp module-wire-item)
       (vrsx-net-args-okp (cdr module-wire-item))))

(defun vrsx-mod-item-parameter-declaration-okp (module-parameter-item)
  (declare (xargs :guard t))
  (and (consp module-parameter-item)
       (vrsx-identifier-expr-list-okp (cdr module-parameter-item))))

(defun vrsx-mod-item-declaration-okp (mod-item)
  (declare (xargs :guard t))
  (and (consp mod-item)
       (symbolp (car mod-item))
       (case (car mod-item)
         (Input     (vrsx-mod-item-input-declaration-okp     mod-item))
         (Output    (vrsx-mod-item-output-declaration-okp    mod-item))
         (Wire      (vrsx-mod-item-wire-declaration-okp      mod-item))
         (ParamDecl (vrsx-mod-item-parameter-declaration-okp mod-item))
         (t         nil))))

(defun vrsx-mod-item-continuous-assign-okp (mod-item)
  (declare (xargs :guard t))
   (and (consp mod-item)
        (symbolp (car mod-item))
        (vrsx-identifier-expr-list-okp (cdr mod-item))))

(defun vrsx-module-instantiation-item-okp (mod-item)
  (declare (xargs :guard t))
  (and (consp mod-item)
       (symbolp (car mod-item))
       (consp (cdr mod-item))
       (symbolp (cadr mod-item))
       (consp (cddr mod-item))  ;;  At least one occurrence with gate.
       (vrsx-occ-name-and-args-list-okp (cddr mod-item))))

(defun vrsx-module-instantiation-item-lst-okp (mod-items)
  (declare (xargs :guard t))
  (if (atom mod-items)
      t
    (and (consp (car mod-items))
         (symbolp (car (car mod-items)))
         (vrsx-module-instantiation-item-okp (car mod-items))
         (vrsx-module-instantiation-item-lst-okp (cdr mod-items)))))

(defun vrsx-mod-item-okp (mod-item)
  (declare (xargs :guard t))
  (and
   (consp mod-item)
   (let ((module-type  (car mod-item)))
     (and
      (symbolp module-type)
      (cond ((member module-type '(Input Output Wire ParamDecl))
             (vrsx-mod-item-declaration-okp          mod-item))

            ((eq module-type 'ModInst)
             (vrsx-module-instantiation-item-okp     mod-item))

            ((eq module-type 'Cont-Assign)
             (vrsx-mod-item-continuous-assign-okp    mod-item))
```

```
           (t nil))))))

;;;  We are leaving space for annotations.  This is likely to be
;;;  important as we look at more complex circuits.

(defun vrsx-mod-items-okp (mod-items)
  (declare (xargs :guard t))
  (if (atom mod-items)
      (eq mod-items nil)
    (and (consp (car mod-items))
         (vrsx-mod-item-okp (car mod-items))
         (vrsx-mod-items-okp (cdr mod-items)))))

(defthm symbol-alistp-vrsx-mod-items-okp
  (implies (vrsx-mod-items-okp items)
           (symbol-alistp items)))

;;;  We define the syntactic restrictions for occurrences and modules.

(defun vrsx-module-ports-match-inouts (ins outs ports)
  (declare (xargs :guard (and (symbol-listp ins)
                              (symbol-listp outs)
                              (symbol-listp ports))))
  (set-equal-eq ports (append ins outs)))

(defun vrsx-wires-disjoint-from-inouts (ins outs wires)
  (declare (xargs :guard (and (symbol-listp ins)
                              (symbol-listp outs)
                              (symbol-listp wires))))
  (and (disjoint-eq ins wires)
       (disjoint-eq outs wires)))

(defun vrsx-continuous-assignments-okp (cont-assigns)
  (declare (xargs :guard (all-true-listp cont-assigns)))
  (vrsx-identifier-expr-list-okp cont-assigns))


;;;  We define the syntactic restrictions for occurrences and modules.

(defun vrsx-module-basic-okp (module)
  (declare (xargs :guard t))
  (and (all-true-listp module)
       (all-atoms-are-symbolp-or-natp module)
       (true-listp-consp-at-least-n module 4)))

(defun vrsx-module-basic-error-msg (module)
  (declare (xargs :guard t))
  (or (if (all-true-listp module) nil
        "Not a well-formed lisp expression.")

      (if (all-atoms-are-symbolp-or-natp module) nil
        "Not all atoms are symbols or natural numbers.")

      (if (true-listp-consp-at-least-n module 4) nil
        "Module not four elements long.")

      nil))

(defthm not-vrsx-module-basic-error-msg-is-vrsx-module-basic-okp
  (equal (not (vrsx-module-basic-error-msg module))
         (vrsx-module-basic-okp module)))

(defun vrsx-module-pieces-okp (module)
  (declare (xargs :guard (vrsx-module-basic-okp module)))
  (let ((mod-id     (mod-id     module))
        (mod-name   (mod-name   module))
        (mod-ports  (mod-ports  module))
        (mod-items  (mod-items  module)))
```

```
    (and (eq mod-id 'Module)
         (vrsx-identifier-okp      mod-name)
         (vrsx-identifier-list-okp mod-ports)
         (vrsx-mod-items-okp       mod-items))))


(defun vrsx-module-pieces-error-msg (module)
  (declare (xargs :guard (vrsx-module-basic-okp module)))
  (let ((mod-id     (mod-id     module))
        (mod-name   (mod-name   module))
        (mod-ports  (mod-ports  module))
        (mod-items  (mod-items  module)))

    (or (if (eq mod-id 'Module) nil
          "Module does not begin with `Module' keyword")

        (if (vrsx-identifier-okp mod-name) nil
          "Module name is not valid.")

        (if (vrsx-identifier-list-okp mod-ports) nil
          "Module port list should only contains names.")

        (if (vrsx-mod-items-okp mod-items) nil
          "The syntax of an item in the module is incorrect.")

        nil
        )))

(defthm not-vrsx-module-pieces-error-msg-is-vrsx-module-pieces-okp
  (equal (not (vrsx-module-pieces-error-msg module))
         (vrsx-module-pieces-okp module)))

(defun vrsx-module-okp (module)
  (declare (xargs :guard t))
  (and
   (vrsx-module-basic-okp module)
   (vrsx-module-pieces-okp module)

   (let* ((mod-ports  (mod-ports module))
          (mod-items  (mod-items module))

          (in-items          (collect-if-car-eq-to-symbol 'Input       mod-items))
          (out-items         (collect-if-car-eq-to-symbol 'Output      mod-items))
          (wire-items        (collect-if-car-eq-to-symbol 'Wire        mod-items))
          (mod-inst-items    (collect-if-car-eq-to-symbol 'ModInst     mod-items))
          (cont-assign-items (collect-if-car-eq-to-symbol 'Cont-Assign mod-items)))

     (and
      (list-of-true-listp in-items)
      (list-of-true-listp out-items)
      (list-of-true-listp wire-items)
      (list-of-true-listp mod-inst-items)
      (list-of-true-listp cont-assign-items)

      (let ((in-names        (collect-all-nthcdr 3 in-items))
            (out-names       (collect-all-nthcdr 3 out-items))
            (wire-names      (collect-all-nthcdr 3 wire-items))
            (cont-assigns    (collect-all-nthcdr 1 cont-assign-items))
            (mod-insts       (get-all-nthcdr     1 mod-inst-items)))

        (and
         (list-of-true-listp mod-insts)

         (let ((mod-insts-without-type (collect-all-nthcdr 1 mod-insts)))

           (and
            (list-of-true-listp mod-insts-without-type)
```

```
            (let* ((mod-occ-names      (get-all-nth 0 mod-insts-without-type))
                   (wire-and-occ-names (append wire-names mod-occ-names)))

              (and
               (symbol-listp in-names)  ;  Could be established
               (symbol-listp out-names) ;  from previous guards.
               (symbol-listp wire-names)
               (symbol-listp mod-occ-names)

               (list-of-true-listp cont-assigns)
               (list-of-true-listp mod-insts)

               (all-true-listp cont-assigns)
               (symbol-listp wire-and-occ-names)

               (vrsx-module-ports-match-inouts in-names out-names mod-ports)
               (vrsx-wires-disjoint-from-inouts in-names out-names wire-names)
               ;;  Ensure that the wire and occurrence names are distinct.
               (tree-for-set wire-and-occ-names)
               ;; (fast-no-duplicatesp wire-and-occ-names)
               (vrsx-continuous-assignments-okp cont-assigns)))))))))))


(defun vrsx-module-error-msg (module)
  (declare (xargs :guard t))
  (or
   (vrsx-module-basic-error-msg module)
   (vrsx-module-pieces-error-msg module)

   (let* ((mod-ports  (mod-ports module))
          (mod-items  (mod-items module))

          (in-items          (collect-if-car-eq-to-symbol 'Input       mod-items))
          (out-items         (collect-if-car-eq-to-symbol 'Output      mod-items))
          (wire-items        (collect-if-car-eq-to-symbol 'Wire        mod-items))
          (mod-inst-items    (collect-if-car-eq-to-symbol 'ModInst     mod-items))
          (cont-assign-items (collect-if-car-eq-to-symbol 'Cont-Assign mod-items)))

     (and
      (list-of-true-listp in-items)
      (list-of-true-listp out-items)
      (list-of-true-listp wire-items)
      (list-of-true-listp mod-inst-items)
      (list-of-true-listp cont-assign-items)

      (let ((in-names        (collect-all-nthcdr 3 in-items))
            (out-names       (collect-all-nthcdr 3 out-items))
            (wire-names      (collect-all-nthcdr 3 wire-items))
            (cont-assigns    (collect-all-nthcdr 1 cont-assign-items))
            (mod-insts       (collect-all-nthcdr 1 mod-inst-items)))

        (and
         (alistp mod-insts)

         (let ((mod-insts-without-type (strip-cdrs mod-insts)))

           (and
            (list-of-true-listp mod-insts-without-type)

            (let* ((mod-occ-names   (get-all-nth 0 mod-insts-without-type))
                   (wire-and-occ-names (append wire-names mod-occ-names)))

              (and
               (symbol-listp in-names)  ;  Could be established
               (symbol-listp out-names) ;  from previous guards.
               (symbol-listp wire-names)
               (symbol-listp mod-occ-names)
```

```
               (list-of-true-listp cont-assigns)
               (list-of-true-listp mod-insts)

               (all-true-listp cont-assigns)
               (all-true-listp mod-insts)

               (symbol-listp wire-and-occ-names)

          (if (vrsx-module-ports-match-inouts in-names out-names mod-ports) nil
            "The input and output names that do not match the ports.")

          (if (vrsx-wires-disjoint-from-inouts in-names out-names wire-names) nil
            "The module wires are not disjoint from the ports.")

          (if ;; (fast-no-duplicatesp (append wire-names mod-occ-names))
              (tree-for-set (append wire-names mod-occ-names))
              nil
            "Duplicate wire names and/or occurrence names.")

          (if (vrsx-continuous-assignments-okp cont-assigns) nil
            "The continuous assignments are not well formed.")

          nil
          ))))))))))

;(defthm not-vrsx-module-error-msg-is-vrsx-module-okp
;  (equal (not (vrsx-module-error-msg module))
;         (vrsx-module-okp module)))


(defun vrsx-netlist-okp (netlist)
  (declare (xargs :guard t))
  (if (atom netlist)
      (eq netlist nil)
    (and (vrsx-module-okp (car netlist))
         (vrsx-netlist-okp (cdr netlist)))))



;;;  For debugging.

#|

(bvl mod-ports  (mod-ports (@ module)))
(bvl mod-items  (mod-items (@ module)))

(bvl in-items          (collect-if-car-eq-to-symbol 'Input       (@ mod-items)))
(bvl out-items         (collect-if-car-eq-to-symbol 'Output      (@ mod-items)))
(bvl wire-items        (collect-if-car-eq-to-symbol 'Wire        (@ mod-items)))
(bvl mod-inst-items    (collect-if-car-eq-to-symbol 'ModInst     (@ mod-items)))
(bvl cont-assign-items (collect-if-car-eq-to-symbol 'Cont-Assign (@ mod-items))))

(bvl in-names        (collect-all-nthcdr 3 (@ in-items)))
(bvl out-names       (collect-all-nthcdr 3 (@ out-items)))
(bvl wire-names      (collect-all-nthcdr 3 (@ wire-items)))
(bvl cont-assigns    (collect-all-nthcdr 1 (@ cont-assign-items)))
(bvl mod-insts       (get-all-nthcdr     1 (@ mod-inst-items))))

|#

(defun vrsx-mod-items-listp (mod-items)
  (declare (xargs :guard t))
  (if (atom mod-items)
      nil
    (cons (and (consp (car mod-items))
               (vrsx-mod-item-okp (car mod-items)))
          (vrsx-mod-items-listp (cdr mod-items)))))
```

```
;;  (vrsx-mod-items-listp (mod-items (@ module)))

(defun vrsx-netlist-listp (netlist)
  (declare (xargs :guard t))
  (if (atom netlist)
      nil
    (cons (vrsx-module-okp (car netlist))
          (vrsx-netlist-listp (cdr netlist)))))


;;;  Identify a set of symbols for this book.

(deftheory raw-syntax-section
  (set-difference-theories (current-theory :here)
                           (current-theory 'raw-syntax-defuns-section)))
```

```
;;;  raw-arity.lisp                          Warren A. Hunt, Jr.

;  This book continues what is acceptable syntax for the Verilog
;  input by checking the arity of the netlist in preparation for it
;  translation to the DE format.

;  NOTE:  Currently, this check is weak.  It's not clear what the
;         value of this is other than to make sure the arity of
;         called to definied primitives is correct.


(in-package "ACL2")
;(include-book "raw-syntax")

(deflabel raw-arity-defuns-section)


;;;  Some helper functions...

(defun collect-module-inst-types (module)
  (declare (xargs :guard (vrsx-module-okp module)))
  (let* ((items           (mod-items module))
         (mod-inst-items  (collect-if-car-eq-to-symbol 'ModInst items))
         (mod-names       (get-all-nth 1 mod-inst-items)))
    (and (symbol-listp mod-names)
         (sort-symbols (remove-duplicates-eq mod-names)))))

(defthm symbol-list-collect-module-inst-types
  (implies (vrsx-module-okp module)
           (symbol-listp (collect-module-inst-types module)))
  :hints (("Goal" :in-theory (en-dis nil (vrsx-module-okp)))))

(defun collect-module-inst-root-types (module)
  (declare (xargs :guard (vrsx-module-okp module)
                  :guard-hints
                  (("Goal" :in-theory (en-dis nil (vrsx-module-okp))))))
  (remove-duplicates-eq
   (truncate-symbol-at-underbar-list
    (collect-module-inst-types module))))

(defun collect-module-instance-occurrence-names (module)
  (declare (xargs :guard (vrsx-module-okp module)))
  (let* ((mod-items      (mod-items module))
         (mod-inst-items (collect-if-car-eq-to-symbol 'ModInst mod-items))
         (mod-insts      (get-all-nthcdr 1 mod-inst-items))
         (mod-insts-without-type (collect-all-nthcdr 1 mod-insts))
         (mod-occ-names  (get-all-nth 0 mod-insts-without-type)))
    mod-occ-names))



;;;  The arity recognizer is a litany of events.  Here, like with the
;;;  raw syntax functions, we check for basic syntactic rationality.


;;;  NOTE:  This is not sufficient for arity checks.  The arity of
;;;  non-primitive module references is not checked.  This will need
;;;  to be improved!!!

(defun primp (fn primitives)
  (declare (xargs :guard (if (symbolp fn)
                             (alistp primitives)
                           (symbol-alistp primitives))))
  (cdr (assoc-eq fn primitives)))

(defthm symbol-alistp-primp
```

```
  ;;  I seem to need this conjunct in spite
  ;;  of forward chaining rules to this one.
  (implies (symbol-alistp-symbol-alistp primitives)
           (and (alistp (primp fn primitives))
                (symbol-alistp (primp fn primitives)))))


(defun primp-type (fn primitives)
  (declare (xargs :guard (and (symbolp fn)
                              (symbol-alistp-symbol-alistp primitives))))
  (cdr (assoc-eq 'type  (primp fn primitives))))

(defun primp-ins  (fn primitives)
  (declare (xargs :guard (and (symbolp fn)
                              (symbol-alistp-symbol-alistp primitives))))
  (cdr (assoc-eq 'ins  (primp fn primitives))))

(defun primp-outs (fn primitives)
  (declare (xargs :guard (and (symbolp fn)
                              (symbol-alistp-symbol-alistp primitives))))
  (cdr (assoc-eq 'outs (primp fn primitives))))

(defun primp-sts  (fn primitives)
  (declare (xargs :guard (and (symbolp fn)
                              (symbol-alistp-symbol-alistp primitives))))
  (cdr (assoc-eq 'sts  (primp fn primitives))))

(in-theory (disable primp))


(defun vrar-mod-occ-okp (sym occ primitives)
  (declare (xargs :guard (and (symbolp sym)
                              (vrsx-occ-name-and-args-okp occ)
                              (symbol-alistp-symbol-alistp primitives))))
  ;;  Checks a single reference
  (if (primp sym primitives)
      (let ((formal-ins  (primp-ins sym primitives))
            (formal-outs (primp-outs sym primitives)))
        (and (alistp (cdr occ))
             (symbol-listp (strip-cars (cdr occ)))
             (symbol-listp formal-ins)
             (symbol-listp formal-outs)
;             (eq (primp-type sym primitives) 'defined-primitives)
             ;; This next s-exp checks the arity as well as the arguments.
             (set-equal-eq (append formal-ins formal-outs)
                           (strip-cars (cdr occ)))))
    t))

(defun vrar-mod-occs-okp (sym lst primitives)
  (declare (xargs :guard (and (symbolp sym)
                              (vrsx-occ-name-and-args-list-okp lst)
                              (symbol-alistp-symbol-alistp primitives))))
  (if (atom lst)
      t
    (and (vrar-mod-occ-okp sym (car lst) primitives)
         (vrar-mod-occs-okp sym (cdr lst) primitives))))


(defun vrar-mod-inst-okp (mod-inst-item primitives)
  (declare (xargs :guard (and (consp mod-inst-item)
                              (symbolp (car mod-inst-item))
                              (vrsx-module-instantiation-item-okp mod-inst-item)
                              (symbol-alistp-symbol-alistp primitives))))
  (let ((type (cadr mod-inst-item))
        (occs (cddr mod-inst-item)))
    (vrar-mod-occs-okp (truncate-symbol-at-underbar type) occs primitives)))
```

```
(defun vrar-mod-inst-lst-okp (mod-inst-items primitives)
  (declare (xargs
            :guard (and (vrsx-module-instantiation-item-lst-okp mod-inst-items)
                        (symbol-alistp-symbol-alistp primitives))))
  (if (atom mod-inst-items)
      t
    (and (vrar-mod-inst-okp (car mod-inst-items) primitives)
         (vrar-mod-inst-lst-okp (cdr mod-inst-items) primitives))))

(defun vrar-module-items (module primitives)
  (declare (xargs :guard (and (vrsx-module-okp module)
                              (symbol-alistp-symbol-alistp primitives))
                  :guard-hints
                  (("Goal" :in-theory
                    (en-dis nil (vrsx-module-ports-match-inouts
                                 vrsx-wires-disjoint-from-inouts
                                 tree-for-set
                                 vrsx-continuous-assignments-okp
                                 collect-if-car-eq-to-symbol
                                 vrsx-module-instantiation-item-lst-okp
                                 vrar-mod-inst-lst-okp
                                 ))))))
  (let* ((mod-items      (mod-items module))
         (mod-inst-items (collect-if-car-eq-to-symbol 'ModInst mod-items)))
    (and (vrsx-module-instantiation-item-lst-okp mod-inst-items)
         (vrar-mod-inst-lst-okp mod-inst-items primitives))))

(defun vrar-netlist-okp (netlist primitives)
  (declare (xargs :guard (and (vrsx-netlist-okp netlist)
                              (symbol-alistp-symbol-alistp primitives))
                  :guard-hints
                  (("Goal" :in-theory (en-dis nil (vrsx-module-okp))))))
  (if (atom netlist)
      t
    (and (vrar-module-items (car netlist) primitives)
         (vrar-netlist-okp (cdr netlist) primitives))))


;;;  Identify a set of symbols for this book.

(deftheory raw-arity-section
  (set-difference-theories (current-theory :here)
   (current-theory 'raw-arity-defuns-section)))
```

```
;;;  raw-translate.lisp                          Warren A. Hunt, Jr.

;  This book defines a translator that transforms a parsed Verilog
;  netlist into the internal form used by the constraint propagation
;  system.

;  I have decided to use the DE format for the constraint propagation
;  system.  It is simple and effective.  This will require additional
;  work in the translator, but the simplicity of the format seems
;  worth it.  This means that the formal parameter names from the
;  parsed Verilog will be eliminated in favor of positional arguments.
;  This means the actual parameter names need to be kept in a
;  particular order, i.e., the order given by the definition of the
;  model being referenced.  In the case of primitives, this order can
;  be found in the database, and in the case of defined modules by
;  looking at the definition of the module.

;  It is going to be necessary for translate to get the modules in an
;  order that will permit the elements with state to be identified.
;  In other words, it is necessary for modules to be translated in an
;  order that allows a module being translated to refer to previously
;  translated modules.

;  The issue of permitting continueous assignments seems best handled
;  by permitting occurrences with no occurrence names; however,
;  because of other checks on the DE format, we will, in some cases,
;  generate dummy occurrence names.

;  The syntax generated by the Verilog parser was designed to be easy
;  to generate and easy to read.

(in-package "ACL2")
; (include-book "arity")

(deflabel raw-translate-defuns-section)

(defun make-pairs (alst-lst)
  (declare (xargs :guard t))
  (if (atom alst-lst)
      nil
    (let ((pair-lst (car alst-lst)))
      (if (and (consp pair-lst)
               (consp (cdr pair-lst)))
          (let ((the-car  (car  pair-lst))
                (the-cdr  (cadr pair-lst)))
            (cons (cons the-car the-cdr)
                  (make-pairs (cdr alst-lst))))
        'error??!!!))))

(defun lst-cons-pairs (x lst)
  (declare (xargs :guard t))
  (if (atom lst)
      nil
    (if (consp (car lst))
        (cons (cons x
                    (cons (caar lst)
                          (make-pairs (cdar lst))))
              (lst-cons-pairs x (cdr lst)))
      nil)))

(defthm alistp-listp-lst-cons-pairs
  (and (true-listp (lst-cons-pairs x lst))
       (alistp     (lst-cons-pairs x lst))))


(defthm symbol-listp-lst-cons-pairs
```

```
  (implies (symbolp x)
           (symbol-alistp (lst-cons-pairs x lst))))

(defun flatten-single-type-mod-occs (mod-inst-items)
  (declare
   (xargs :guard
          (vrsx-module-instantiation-item-lst-okp mod-inst-items)))
  (if (atom mod-inst-items)
      nil
    (let* ((module-instance (car mod-inst-items))
           (type     (cadr module-instance))
           (occ-ref  (cddr module-instance)))
      (append (lst-cons-pairs type occ-ref)
              (flatten-single-type-mod-occs (cdr mod-inst-items))))))

(defthm symbol-alistp-flatten-single-type-mod-occs
  (implies
   (vrsx-module-instantiation-item-lst-okp mod-inst-items)
   (and (true-listp    (flatten-single-type-mod-occs mod-inst-items))
        (alistp        (flatten-single-type-mod-occs mod-inst-items))
        (symbol-alistp (flatten-single-type-mod-occs mod-inst-items))
        )))

; The next function converts a list of continueous assignments
; into the same form as a module RENAME instance.

(defun add-z-a-to-cont-assigns (cont-assigns)
  (declare (xargs :guard t))
  (if (atom cont-assigns)
      nil
    (let ((element (car cont-assigns)))
      (cons (if (and (consp element)
                     (consp (cdr element)))
                (list (list 'a (cadr element))
                      (list 'z (car  element)))
              'error??!!)
            (add-z-a-to-cont-assigns (cdr cont-assigns))))))

; We add an occurrence names for each continuous assignment.

(defun add-occ-name-to-cont-assigns (cont-assign-bind n)
  (declare (xargs :guard (natp n)))
  (if (atom cont-assign-bind)
      nil
    (cons (cons (make-symbol-with-number 'continuous-assign n)
                (car cont-assign-bind))
          (add-occ-name-to-cont-assigns
           (cdr cont-assign-bind) (1+ n)))))

; Convert just the continueous assignments.

(defun convert-cont-assigns-to-rename-mod-insts (items n)
  (declare (xargs :guard (and (symbol-alistp items)
                              (natp n))))
  (if (atom items)
      nil
    (let* ((item (car items))
           (type (item-type item)))
      (if (eq type 'cont-assign)
          (cons (list* 'ModInst 'RENAME
                       (add-occ-name-to-cont-assigns
                        (add-z-a-to-cont-assigns (cdr item))
                        n))
                (convert-cont-assigns-to-rename-mod-insts
                 (cdr items) (+ n (len (cdr item)))))
        (cons item
              (convert-cont-assigns-to-rename-mod-insts
               (cdr items) n))))))
```

```
(defun process-mod-inst-items (items)
  (declare (xargs :guard (symbol-alistp items)))
  (convert-cont-assigns-to-rename-mod-insts items 0))

(defun remove-formal-parameter-names-from-a-item
  (mod-inst-item current-netlist)
  (declare (xargs :guard (sx-netp current-netlist)))
  (if (not (and (consp mod-inst-item)
                (consp (cdr mod-inst-item))))
      'remove-formal-parameter-names-from-a-item-error!!!
    (let* ((reference-name (car mod-inst-item))
           (occ-name       (cadr mod-inst-item))
           (args           (cddr mod-inst-item)))
      (if (not (and (symbolp reference-name)
                    (symbolp occ-name)
                    (symbol-alistp args)))
          'remove-formal-parameter-names-from-a-item-name-error!!!
        (let* ((module-ref (assoc-eq reference-name current-netlist))
               (module-body (and (consp module-ref)
                                 (m-body module-ref)))
               (m-ins  (m-ins  module-body))
               (m-outs (m-outs module-body)))
          (if (not (and (symbol-listp m-ins)
                        (symbol-listp m-outs)))
              'remove-formal-parameter-names-from-a-item-module-ref-error!!!
            (let* ((new-outs (assoc-eq-values m-outs args))
                   (new-ins  (assoc-eq-values m-ins  args)))
              (list occ-name new-outs reference-name new-ins)))))))))

(defun remove-formal-parameter-names-from-items
  (mod-inst-items current-netlist)
  (declare (xargs :guard (sx-netp current-netlist)))
  (if (atom mod-inst-items)
      nil
    (cons (remove-formal-parameter-names-from-a-item
           (car mod-inst-items)
           current-netlist)
          (remove-formal-parameter-names-from-items
           (cdr mod-inst-items)
           current-netlist))))

(defun collect-symbols-from-eqlable-listp (lst)
  (declare (xargs :guard (eqlable-listp lst)))
  (if (atom lst)
      nil
    (if (symbolp (car lst))
        (cons (car lst)
              (collect-symbols-from-eqlable-listp (cdr lst)))
      (collect-symbols-from-eqlable-listp (cdr lst)))))

(defthm symbol-listp-collect-symbols-from-eqlable-listp
  (symbol-listp (collect-symbols-from-eqlable-listp lst)))

(defun collect-inputs-from-occs (occs)
  (declare (xargs :guard (sx-occsp occs)
                  :verify-guards nil))
  (if (atom occs)
      nil
    (let* ((occ   (car occs))
           (o-ins (o-ins occ))
           (v-ins (collect-symbols-from-eqlable-listp o-ins)))
      (set-union v-ins
                 (collect-inputs-from-occs (cdr occs))))))

(defthm symbol-listp-collect-inputs-from-occs
  (symbol-listp (collect-inputs-from-occs occs)))
```

```
(verify-guards collect-inputs-from-occs)

; Find the occurrence names for the occurrences that contain state.

(defun collect-sts-occ-names (mod-inst-items current-netlist)
  (declare (xargs :guard t))
  (declare (ignore mod-inst-items))
  (declare (ignore current-netlist))
  nil)

; Some helper functions for ordering occurrences.

(defun delete-equal (element lst)
  (declare (xargs :guard t))
  (if (atom lst)
      nil
    (if (equal element (car lst))
        (cdr lst)
      (cons (car lst)
            (delete-equal element (cdr lst))))))

(defthm size-smaller-delete-equal
   (implies (member-equal item lst)
            (< (acl2-count (delete-equal item lst))
               (acl2-count lst)))
   :rule-classes (:linear :rewrite))

(defthm true-listp-delete-equal
  (implies (true-listp lst)
           (true-listp (delete-equal element lst))))

(defun find-first-occ-with-defined-inputs (ins mod-occs)
  (declare (xargs :guard (symbol-listp ins)))
  (if (atom mod-occs)
      (list 'find-first-occ-with-defined-out-of-inputs-error!!!)
    (let ((occ (car mod-occs)))
      (if (not (sx-occp occ))
          (list 'find-first-occ-with-defined-syntax-bad-error!!!)
        (let ((o-ins (collect-symbols (o-ins  occ))))
          (if (subset-eq o-ins ins)
              occ
            (find-first-occ-with-defined-inputs ins (cdr mod-occs))))))))


; As necessary, level-order the occurrences.

(defun order-instances (ins mod-occs current-netlist)
  (declare
   (xargs :guard (and (symbol-listp ins)
                      (true-listp mod-occs)
                      (symbol-alistp current-netlist))
          :measure (acl2-count mod-occs)
          :guard-hints
          (("Goal"
            :in-theory (disable find-first-occ-with-defined-inputs)))))

  ;; We consider each occurrence to be well-formed at this point.
  ;; We assume that all states are defined upon entry, so we don't
  ;; check anything regarding the state.

  (if (atom mod-occs)
      nil
    (let ((occ (find-first-occ-with-defined-inputs ins mod-occs)))
      (if (atom occ)
          (list 'order-instance-could-not-find-occ-with-defined-inputs!!!)
        (if (or (not (sx-occp occ))
                (not (member-equal occ mod-occs)))
            (list (cons 'order-instance-syntax-problem!!!
```

```
                    occ))
          (let ((o-outs  (o-outs occ))
                (o-fn    (o-fn   occ)))
            (if (not (assoc-eq o-fn current-netlist))
                (list 'order-instance-module-reference-undefined!!!)
              (cons occ
                    (order-instances (append o-outs ins)
                                     (delete-equal occ mod-occs)
                                     current-netlist)))))))))


;;;  I am here!!!

; RAW-TRANSLATE-MODULE goes through a litany of steps to translate the
; output of the Verilog translator to the internal format used by the
; SeqSimp system.  This is definitely suspect software as there isn't
; even a good way to write a specification for this translator.  The
; idea to this procedure is produce a model that is recognized as
; valid, simplified with the SeqSimp system, and then translated back
; into Verilog.

(defun raw-translate-module (raw-module current-netlist)
  (declare (xargs :mode :program
                  :guard (and (vrsx-module-okp raw-module)
                              (sx-netp current-netlist))))
  (let*
      ((mod-name         (mod-name  raw-module))
       ;; (mod-ports        (mod-ports raw-module))
       (mod-items        (mod-items raw-module))

       ;; The inputs, outputs, and wires items are collected.

       (in-items         (collect-if-car-eq-to-symbol 'Input  mod-items))
       (out-items        (collect-if-car-eq-to-symbol 'Output mod-items))
       (wire-items       (collect-if-car-eq-to-symbol 'Wire   mod-items))

       ;; The input, output, and wire names are collected into lists.

       (in-names         (collect-all-nthcdr 3 in-items))
       (out-names        (collect-all-nthcdr 3 out-items))
       ;; (in-out-names     (append in-names out-names))
       (wire-names       (collect-all-nthcdr 3 wire-items))

       ;; The module instances and continuous assignments are collected.

       (mod-inst-items   (collect-if-car-eq-to-lst-syms
                          '(ModInst Cont-Assign) mod-items))

       ;; Continuous assignments are converted into rename occurrences
       ;; and for each rename occurrence an occurrence name is generated.
       ;; Note!  No check is made to see if there is a name clash.

       (mod-inst-items   (process-mod-inst-items mod-inst-items))

       ;; In Verilog, a reference to a module may contain multiple
       ;; binding; we separate these into separate module references.

       (mod-inst-items   (flatten-single-type-mod-occs mod-inst-items))

       ;; It appears, that all the netlists we will encounter have
       ;; named parameter references.  Our approach has positional
       ;; parameter references, which we collect here.

       (mod-inst-items   (remove-formal-parameter-names-from-items
                          mod-inst-items
                          current-netlist))
```

```
      ;; The representation of a module requires occurrences that
      ;; reference state-holding elements to be listed in the
      ;; module's STS argument.  We recursively collect these
      ;; references.  We assume that argument CURRENT-NETLIST has
      ;; this format.

      (sts-occ-names (collect-sts-occ-names mod-inst-items
                                            current-netlist))

      ;; We require occurrence to be in "level order" so we sort them
      ;; if necessary.  This can be a subtle operation as we must do
      ;; this hierarchically, meaning that access to the orderings of
      ;; the predefined modules and primitives is required.  We
      ;; assume that argument CURRENT-NETLIST has be so ordered.

      (occs          (order-instances in-names
                                      mod-inst-items current-netlist))
      )
    ;; Finally, we build a module.

    (list mod-name
          (cons 'type   'module)
          (cons 'ins    in-names)
          (cons 'sts    sts-occ-names)
          (cons 'outs   out-names)
          (cons 'wires  wire-names)
          (cons 'occs   occs))))


; We convert a "raw" netlist into one suitable for simplification.
; The input to this process is parsed Verilog, which is referred to as
; the "raw" systax.  We convert a raw netlist in the syntax required
; by our simplification system.

(defun raw-translate-netlist (raw-netlist evolving-netlist)
  (declare (xargs :mode :program
                  :guard (vrsx-netlist-okp raw-netlist)))
  (if (atom raw-netlist)
      evolving-netlist
    (raw-translate-netlist
     (cdr raw-netlist)
     (cons (raw-translate-module (car raw-netlist) evolving-netlist)
           evolving-netlist))))


;;;  Identify a set of symbols for this book.

(deftheory raw-translate-section
  (set-difference-theories (current-theory :here)
   (current-theory 'raw-translate-defuns-section)))
```

```
;;;  raw-test-events.lisp                    Warren A. Hunt, Jr.

;  A litany of events that read a parsed file, translate it into the
;  DE format, and the propagate the constraints.


;  Set NETLIST to the contents of the named file.

(bvl-file raw-netlist "raw-net-example.lisp")


;  Translate a raw netlist into the DE format.

(bvl netlist
     (raw-translate-netlist (@ raw-netlist)
                            (@ all-primitives)))


;  Check the translation for syntax and arity.

(list (sx-netp (@ netlist))
      (ar-netp (@ netlist)))


;  Exhaustive simulation tests.

; (test-sn74182-n 511 (@ netlist))
; (test-sn74181-n (1- (expt 2 14)) (@ netlist))
```