NASA Technical Memorandum 100669

AVSCOM Technical Memorandum 88-B-017

HARDWARE PROOFS USING EHDM AND THE RSRE VERIFICATION METHODOLOGY

Ricky W. Butler
Jon A. Sjogren

December 1988

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

US ARMY AVIATION SYSTEMS COMMAND
AVIATION R&T ACTIVITY

TABLE OF CONTENTS

INTRODUCTION

Recently NASA, Langley Research Center and the Royal Signals and Radar Establishment (RSRE) have initiated a joint research program in formal verification of life-critical systems. The first phase of this work involves a critical assessment of the RSRE work on the VIPER microprocessor. The VIPER was designed by RSRE researchers specifically for life-critical applications and was subjected to a formal proof of correctness. The proof methodology is based on a hierarchical specification of the system design. This methodology was first illustrated on a 6-bit counter by RSRE in the RSRE Memorandum 3832 entitled "Hardware Proofs using LCF-LSM and ELLA" by W. J. Cullyer and C. H. Pygott (ref. 1.) In this paper, the RSRE approach to hardware verification is studied in the context of a different specification language -- Revised Special developed by SRI International (ref. 2). The reason the methodology is explored via a different specification language is twofold: (1) to expose any weaknesses in the methodology due to the specification language LCF-LSM, and (2) to explore the feasibility of using EHDM (Enhanced Hierarchical Design Methodology) for hardware verification using the RSRE methodology.

In this paper RSRE's 6-bit counter example is re-specified in Revised Special. In the RSRE report, the proofs between the levels of the hierarchical specification were accomplished by hand. In this report, the proofs are performed using the EHDM (Enhanced Hierarchical Design Methodology) theorem proving system. The paper makes a comparison between the LCF-LSM and Revised Special languages. The viability of the RSRE methodology is discussed. Particular attention is given to the feasibility of using their methodology in concert with the EHDM tools.

SUMMARY OF RSRE HARDWARE VERIFICATION METHODOLOGY

The RSRE approach to verification is based on the use of hierarchical specification. The formal hierarchy consists of the following four levels:

(1) Functional
(2) Finite-state automata
(3) Block model
(4) Circuit model

The proof between level (1) and (2) establish that the finite automata of level (2) implements all of the functions of the top level. The top level consists of axioms which define the output of the circuitry in response to inputs without any details of the steps that are performed to accomplish the computation. Thus, the top level is essentially the definition of a mathematical function. The second level decomposes the function into sequences of steps which can accomplish the overall functionality. The sequences of steps are defined by a finite-automata model. Proof that level (1) follows from (2) is accomplished by enumerating all possible sequences that the finite automata can perform and demonstrating that these accomplish the function of level (1).

At level (2) the computation performed by each transition of the finite automata is specified by a mathematical (sub)function. The details of how each of these subfunctions are computed is not specified until level (3). Level (3) specifies how each of the subfunctions of level (2) are accomplished in terms of an electronic block diagram. The proof between level (2) and (3) establishes that each of the subfunctions of level (2) are properly computed by the level (3) structure. Finally, the proof between level (3) and (4) establishes that each "block" in the level (3) model is correctly implemented with logic gates.

In the RSRE work the first three levels were specified in LCF-LSM. The last level was specified in ELLA (ref. 3.) The third level was also specified in ELLA in addition to the LCF-LSM specification. The properties at each level in the hierarchy must be proved to be theorems in the level below it. The proofs between levels (1) and (2) and between (2) and (3) are accomplished analytically. The proof between levels (3) and (4) are accomplished by the method of "intelligent exhaustion".

TOP-LEVEL SPECIFICATION OF SIX-BIT COUNTER

The example used by RSRE to illustrate their specification/verification methodology was a six-bit counter. The counter holds a value "count" which is either retained at its current value, loaded with a new value from an external source, or incremented once or twice depending on the value of "func", a two-bit control signal. The informal specification for the counter is

func = 0 Do nothing: "count" unchanged

func = 1 Load "count" from a 6-bit parallel input, "loadin"

func = 2 Increment "count": i.e. count := count + 1

func = 3 Increment "count" twice: i.e. count := count + 2

In the RSRE report this informal specification is translated into a formal specification written in LCF-LSM. The design of the counter is documented by a hierarchical specification where each successive level in the hierarchy introduces more detail as the result of design decisions. In this paper these specifications are presented in Revised Special.

The formal Specification of the counter in Revised Special is:

```
cnt6: MODULE
USING words
THEORY

  states: TYPE
  word6: TYPE is word[6]
  word2: TYPE is word[2]

  state: VAR states
  loadin,w: VAR word6
  func: VAR word2

  val2: function[word2 -> int] is val[2]
  val6: function[word6 -> int] is val[6]
  mw6: function[int -> word6]  is mw[6]

  cnt: function[states -> word6]
  exec_cnt: function[states,word6,word2 -> states]
  ready: function[states -> bool]

  add1_mod64: function[word6 -> word6] ==
              ( LAMBDA w -> word6:
                       IF val6(w) = 63 THEN mw6(0)
                       ELSE mw6(val6(w)+1)
                       END )

  ready_ax: AXIOM ready(state) IMPLIES ready( exec_cnt(state,loadin,func) )

  counter_ax: AXIOM ready(state) IMPLIES cnt(exec_cnt(state,loadin,func)) =
                    IF val2(func) = 0 THEN cnt(state)
                    ELSIF val2(func) = 1 THEN loadin
                    ELSIF val2(func) = 2 THEN
                       add1_mod64(cnt(state))
                    ELSE
                       add1_mod64(add1_mod64(cnt(state)))
                    END

END cnt6
```

It is unnecessary to present details about Revised Special, since the above specification can be understood with a little explanation. (This is probably the best way to be introduced to a formal specification language -- by way of example.) The first line assigns the name "cnt6" to the specification. The second line indicates that an external module "words" will be used in this module. This will be explained in more detail in the following discussion. The next three lines which follow the THEORY keyword define three "types" -- "states", "word6", and "word2". These types will be used to distinguish logic variables which represent the state (or "count") of the machine, 6-bit words, and 2-bit words, respectively. Types serve the same function in the formal specification language as in a programming language -- they enable the automatic system to detect user errors.

The first type "state" is uninterpreted, that is, there is no domain of values or any meaning associated with it. At this level of abstraction it represents the state of the machine, but details about what constitutes the state of the machine are not specified. The two types, "word6" and "word2" are equated to word[6] and word[2] by the "is" clause. They represent the domain of 6-bit words and 2-bit words. N-bit words are used to specify an N-bit transmission lines which can be interpreted as integers or as ordered sets of boolean values. The properties of the generic type word[N] are defined in the module "words" which is discussed in detail in the section entitled "SPECIFICATION OF N-BIT WORDS". The next three lines of the specification define four logic variables - "state", "loadin", "w" and "func". (Note. These are mathematical variables, not program variables) Next, seven functions are defined. The meaning of these functions are:

val2: maps an unsigned 2-bit word to its positive integer equivalent

val6: maps an unsigned 6-bit word to its positive integer equivalent

mw6: maps a positive integer into an unsigned 6-bit word

cnt: returns the value of the 6-bit counter when applied to the "state" of the machine

exec_cnt: maps the state of the machine to its new state when the counter is "executed"

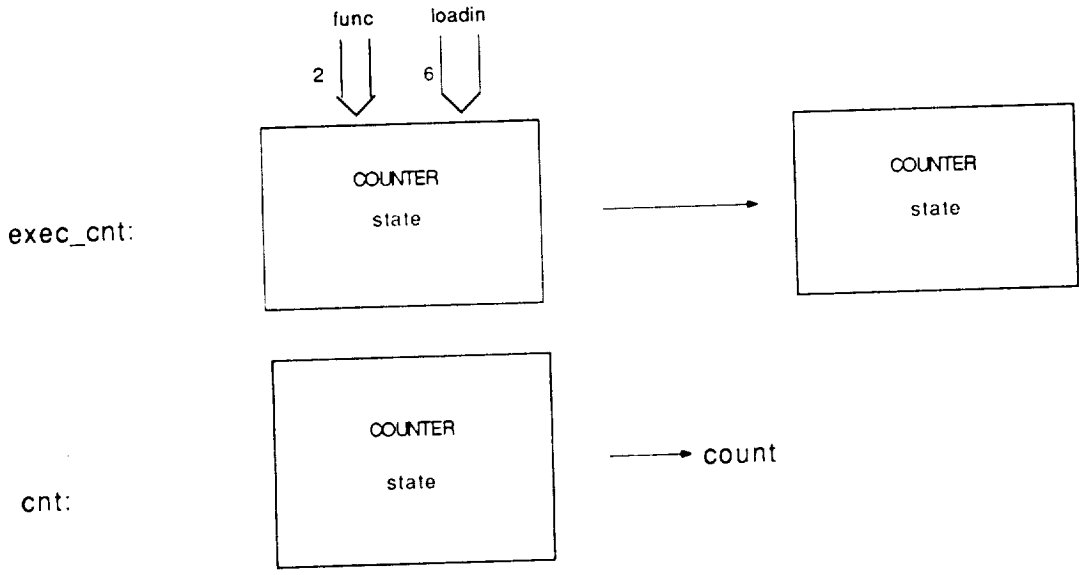
ready: when applied to the state of the machine returns "true" if and only if the machine is in the ready state, i.e. ready to receive the next "func". This function is necessary since the execution

of the counter is not instantaneous. There are intermediate states in the machine. It is important to prove that when the counter is executed (i.e. via exec_cnt) the machine is returned to a state where it is ready to receive the next input. (Note. This property was not considered in the RSRE work).

add1_mod64: adds 1 (modulo-64) to a 6-bit word

The first three functions are equated to functions defined in the module "words". The last three functions are elaborated in more detail (formally) at lower levels of the specification hierarchy.

The last part of the specification provides two axioms which define the behavior of the counter. The first axiom expresses the most important properties of the counter in terms of the function "exec_cnt". The function "exec_cnt" transforms the state of the counter in response to the inputs "state", "loadin", and "func" and thus defines the "execution" of the counter:



At this level of abstraction, the state of the machine has only two properties -- the value of the counter, an integer between 0 and 63, and whether the machine is ready. The value of the counter is returned by the function "cnt". The function "ready" returns a boolean variable indicating whether the machine is ready for input. The execution of the counter is defined by "exec_cnt". If the machine is ready, then the counter will operate correctly and the state of

the counter will be updated according to the "counter_ax" axiom. If the value of func is 0 (i.e. val2(func) is 0), then the value of the counter remains the same (i.e. cnt(state) -- the original state). If the value of "func" is 1, then the value of the counter is changed to the value "loadin". If the value of "func" is 2, then the value of the counter is incremented by 1. Of course, one must consider the case where the counter "turns over" (i.e. if the counter is currently at $63 = 111111_2$, it next becomes 0). This concept is captured by modulo-64 arithmetic. The "add1_mod64" function adds 1 to a word6 variable in modulo-64 arithmetic. The meaning of the "add1_mod64" function specification is simple. If the value of the current value of the state (i.e. val6(cnt(state))) is 63, then a single increment will turn-over the counter to 0 (i.e. mw(6) -- the 6-bit word whose value is 0). Otherwise, it is just the value of the counter plus one (i.e. mw6(val6(cnt(state))+1)).[1]

Thus, for "func" = 2 the value of the counter becomes

add1_mod64(cnt(state))

If the value of "func" is 3, then a double increment is performed:

add1_mod64(add1_mod64(cnt(state)))

The second axiom, "ready_ax", expresses the concept that a complete execution of the counter returns the counter to a ready state, if it was originally ready.

THE FINITE-STATE AUTOMATA SPECIFICATION

The six-bit counter is defined as a finite-state automata at this level of abstraction. The second level in the hierarchy is called the major-state specification in later RSRE documents, but is referred to as the finite-state automata specification throughout this paper. The finite automata consists of 4 states named "fetch", "inc1", "inc2", and "load" shown in figure 1. The machine is assumed to start in the "fetch" state.

---

1 The "add1_mod64" function is defined using EHDM's LAMBDA notation. The concept is intuitive. The keyword "LAMBDA" indicates that a function is being defined. This keyword is followed by the formal arguments to the function. The symbol -> is followed by the type of the function result. Finally, the function's body is provided. For example, the integer function $f(x) = x^2+1$ would be formally defined by f: function[int -> int] = (LAMBDA x -> int: x*x+1).

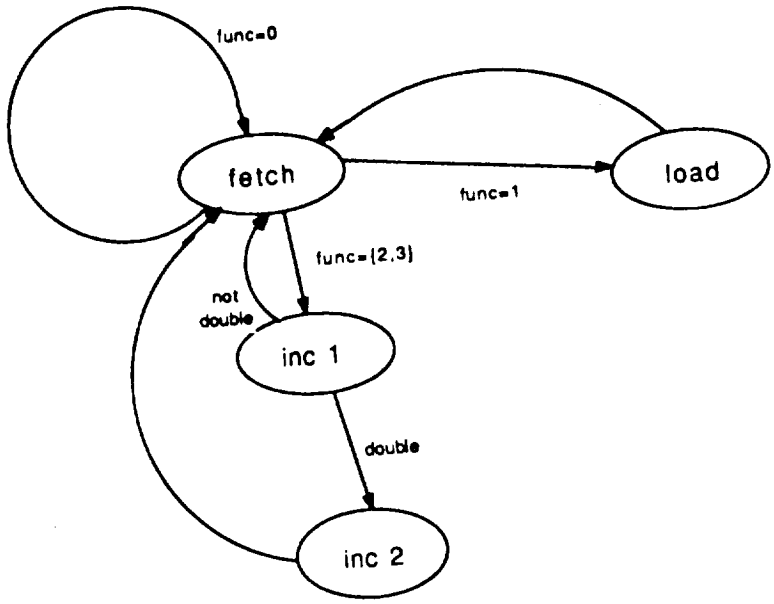


Figure 1. Diagram of finite-state automata

It is necessary not only to describe this finite-state automata formally, but also mathematically map the more abstract specification onto this specification. These issues are addressed in the next two sections.

Definition of Finite-state Automata

The finite state model is defined using two external modules -- "words" and "triples".[2] The first module "words" provides a formal definition of what constitutes an N-bit word. This specification module is generic and can be used for other hardware designs. The module "words" is described in detail in the section called "FORMAL SPECIFICATION OF N-BIT WORDS". The theory of N-bit

2 Modules serve the same purpose in Revised Special as they do in a programming language -- they facilitate the definition and re-use of specifications.

words can be used by "importing" the "words" module via a USING clause or the similar MAPPING clause. This module defines the following functions:

val: maps an N-bit word to an unsigned integer
mw: maps unsigned integer to an N-bit word
bit: returns the contents of a specified bit in a word

These functions are defined in a "parameterized" module (i.e. parameterized by N, the number of bits in a word). Thus, for each "instantiation" of the module (i.e. declaration in a USING clause), there are three different functions. For example,

```
USING words[2], words[6]
```

defines two types -- word[2] and word[6]-- and six functions -- val[2], mw[2], bit[2], val[6], mw[6], bit[6].

The module "triples" defines the concept of an "ordered triple".[3] The ordered triple has three components which can be accessed with the functions "first", "second", and "third". An ordered triple is created from individual components via the function make_triple. The relationship between these functions is described by the following axiom in the triples module:

```
Make_triple_ax: AXIOM
  x = first(make_triple(x, y, z))
    AND y = second(make_triple(x, y, z))
      AND z = third(make_triple(x, y, z))
```

The "state" of the finite-state machine is designated by an ordered triple:

(count, double, node)

where

count = the current value of the counter, a number between 0 and 63 inclusive
double= a boolean variable which is true if and only if a double increment is to be performed
node = indicates at which node the machine is currently located

---

3 Mathematicians typically use the notation (x,y,z) to define a triple.

This triple is defined formally as follows:

```
USING triples[word[6],bool,word[2]]

  statevector: TYPE is triple

  count:  function[ statevector -> word6 ] is first
  double: function[ statevector -> bool ] is second
  node:   function[ statevector -> word2 ] is third[4]
```

The first line imports the generic theory of triples. The module was parameterized by "word[6], bool, word[2]". This results in a theory of triples where the first component is of type word[6], the second component is of type bool and the third component is word[2]. To enhance readability of the specification, alternate names are given to the names which are exported from the "triples" module in the next 4 lines. Thus statevector is an alternate name for the type "triple" with the following components -- a 6-bit word, a boolean, and a 2-bit word. The individual components of the triple can be accessed via the functions -- "count", "double", and "node".

The allowed transitions of the finite automata are defined by the NEXT function. The NEXT function maps the "statevector" to the new "statevector" in response to the 2-bit function code "func" and "loadin":

```
  NEXT: function[statevector,word6,word2 -> statevector]
```

This function was defined as follows in the original version:

```
  NEXT: function[statevector,word6,word2 -> statevector] =
        (LAMBDA stv,ldn,fn -> statevector:
                    IF val2(node(stv)) = 0 THEN
                       FETCH(count(stv),double(stv),ldn,fn)
                    ELSIF val2(node(stv)) = 1 THEN
                       INC1(count(stv),double(stv),ldn,fn)
                    ELSIF val2(node(stv)) = 2 THEN
                       INC2(count(stv),double(stv),ldn,fn)
                    ELSE
                       LOAD(count(stv),double(stv),ldn,fn)
                    END
```

Unfortunately, when the function was defined in this manner, the EHDM theorem

---

4 In EHDM synonyms are defined using the keyword "is".

prover required excessive amounts of time. The properties of "NEXT" were consequently defined using 4 separate axioms:

```
NEXT0_ax: AXIOM val2(node(stv)) = 0 IMPLIES
                NEXT(stv,ldn,fn) = FETCH(count(stv),double(stv),ldn,fn)

NEXT1_ax: AXIOM val2(node(stv)) = 1 IMPLIES
                NEXT(stv,ldn,fn) = INC1(count(stv),double(stv),ldn,fn)

NEXT2_ax: AXIOM val2(node(stv)) = 2 IMPLIES
                NEXT(stv,ldn,fn) = INC2(count(stv),double(stv),ldn,fn)

NEXT3_ax: AXIOM val2(node(stv)) = 3 IMPLIES
                NEXT(stv,ldn,fn) = LOAD(count(stv),double(stv),ldn,fn)
```

By defining the properties of NEXT using four axioms, the theorem prover could be directed to find the proof in a more efficient manner.[5] The function NEXT is defined in terms of the subfunctions INC1, INC2, LOAD, and FETCH to enhance the readability of the specification. Originally all of the subfunctions were defined using the LAMBDA syntax. However, in this form the formal proofs (i.e., proving that this level implements the top_Level spec) required exhorbitant amounts of CPU time. These functions were redefined using "axiomatic" definitions and the proofs required only a few minutes to complete.[6]

The total functionality of the counter is captured in the function "Finite_automata":

```
Finite_automata: function[statevector,word6,word2 -> statevector] =
                 (LAMBDA svt, ldn, fn -> statevector:
                      IF val2(fn) = 0 THEN
                         NEXT(svt,ldn,fn)
                      ELSIF val2(fn) = 3 THEN
                         NEXT(NEXT( NEXT(svt,ldn,fn), ldn,fn ),ldn,fn )
                      ELSE
                         NEXT( NEXT(svt,ldn,fn), ldn,fn )
                      END )
```

---

5 One could easily prove that the former specification defines a function which is equivalent to the function defined by these four axioms.

6 Although defining the subfunctions with axioms increases the work of the "human" prover -- i.e. one must explicitly cite the axiom whenever the function is used in a formula being proved --, the amount of proving time can be drasticly reduced. The reduction in proving time comes by only citing the functions whose expansion is relevant to the proof.

This function defines the sequence of calls of "NEXT" which accomplish each function. This function represents a "spanning tree" of the graph shown in figure 2.

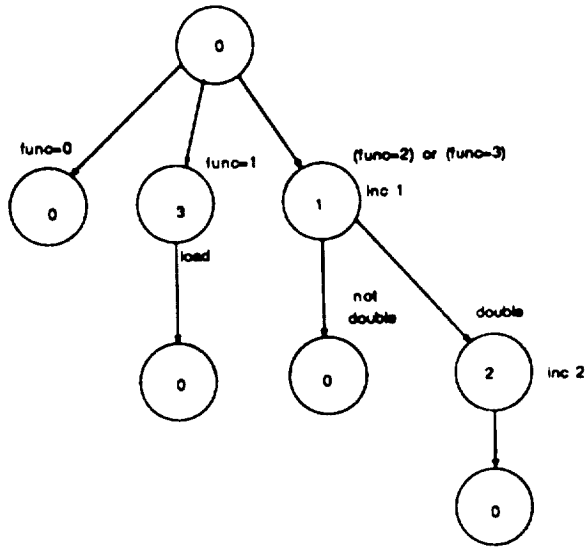


Figure 2. - Spanning Tree for Finite-Automata

If "Finite_Automata" is defined properly, the counter will be returned to the "fetch" node at the completion of the function as well as performing the specified function.

Mapping to the Top-Level Specification

The mappings to the higher level of abstraction are made using EHDM mapping statements. (In the RSRE report the connections between models were informal). EHDM requires that a mapping be provided for every uninterpreted type and every constant of the module being mapped (i.e. the higher level specification). In "cnt6" the following uninterpreted type was defined:

states

The following functions were defined:

cnt, exec_cnt, ready:

These are "mapped" in module "cnt6_fa" as follows:

```
cnt6.states: TYPE FROM statevector

cnt6.cnt: function[statevector -> word6] is count

cnt6.exec_cnt: function[statevector,word6,word2 -> statevector]
                is Finite_automata

cnt6.ready: function[statevector -> bool] =
             (LAMBDA stv -> bool: node(stv) = fetchnode )
```

The "cnt6." prefix indicates that "cnt6" functions are being mapped. The "cnt6" function "cnt" is mapped to "count" which is an abbreviation for the first component accessor function "first" of type "triples".

The "cnt6" function "exec_cnt" is mapped to the function "Finite_automata". The "cnt6" function "ready" is mapped by a function that returns true if and only if the automata is currently located at "fetchnode":

```
cnt6.ready: function[statevector -> bool] =
              (LAMBDA stv -> bool: node(stv) = fetchnode )
```

The need for this function now becomes clear. It is possible that an improperly designed counter could return the correct "count" but not correctly return the machine to the proper state, namely "fetchnode", where it is ready for the next input. This captures the "sequential" nature of the circuit. This property was not captured in the RSRE LCF-LSM specification.

External Interface and Timing Issues

The RSRE report does not formally define the interaction of the counter with respect to asynchronous changes in "func" and "loadin". The report examined the impact of changes in "func" and "loadin" while the finite automata is executing by a method called "hoisting the exit conditions". The method is built on the concept that the finite automata samples from lists of "func" and "loadin" values. These lists contain countable sequences of values which the "func" and "loadin" lines contain at the time points which the finite automata samples them. The finite automata is assumed to be driven by a synchronous clock -- one clock tick per transition of the finite automata. Thus, the calls

to "NEXT" are triggered by the synchronous clock. The analysis given in the RSRE report indicates how to match the values in the list with the execution of the counter.

Since it is possible that the value of "func" or "loadin" can change over time, this must be accounted for in the specification. In the "cnt6_fa" specification above it is implicitly assumed that the values do not change until the counter has returned to the "fetchnode" state. This is implied by the fact that all of the calls to "NEXT" in the function "Finite_automata", use the same values of "ldn" and "fn" (i.e. the parameters which correspond to "loadin" and "func" in the top spec). For example,

```
NEXT(NEXT( NEXT(svt,ldn,fn), ldn,fn ),ldn,fn )
```

If this assumption is not valid, the specification could be generalized by defining a list of "func" and "loadin" signal values:

```
clocktime: TYPE is int

funcsigs: function[clocktime -> word2]
loadinsigs: function[clocktime -> word6]
```

These functions "map" the synchronous clock time to the values of "func" and "loadin" at those times. It is necessary to assume that the values of "func" and "loadin" are "stable" at the time that the finite automata samples them.

In order to relate the behavior of the finite automata over time to these input values it is necessary to extend the definition of state to include time:

```
(count, double, node, clk)
```

The first three components are as before. The fourth component indicates the current time, i.e. the number of clock pulses which have been sent to the automata thus far. Formally, we would have:

```
USING quads[word6,bool,word2,nat]

  statevector: TYPE is quad
```

```
count:  function[ statevector -> word6 ] is first
double: function[ statevector -> bool ] is second
node:   function[ statevector -> word2 ] is third
clk:   function[ statevector -> word2 ] ] is fourth
```

Function "NEXT" and its subfunctions would have to be modified to increment the value of clk. For example,

```
NEXT0_ax: AXIOM val2(node(stv)) = 0 IMPLIES
          NEXT(stv,ldn,fn) = FETCH(count(stv),double(stv),clk(stv),ldn,fn)
           = IF val2(fn) = 0 THEN
                make_quad(count(stv),BOOLF(bit2(0,fn)),fetchnode,clk(stv)+1)
             ELSIF val2(fn) = 1 THEN
                make_quad(count(stv),BOOLF(bit2(0,fn)),loadnode,clk(stv)+1)
             ELSE
                make_quad(count(stv),BOOLF(bit2(0,fn)),inclnode,clk(stv)+1)
             END
```

The net result would be to formally connect the arguments of "NEXT" in the definition of "Finite_automata" to the sequence of func and loadin values over clock time:

```
NEXT(NEXT( NEXT(svt,loadinsigs[1], funcsigs[1]),
           loadinsigs[2], funcsigs[2]),
     loadinsigs[3], funcsigs[3])
```

The cnt6_fa Specification

The cnt6_fa specification excluding the proofs is:

```
cnt6_fa: MODULE
MAPPING cnt6 ONTO words, triples[word[6],bool,word[2]],bsignal

THEORY
  (* ---------------------- create some abbreviations ---------------------- *)

  word2: TYPE is word[2]
  word6: TYPE is word[6]

  mw2: function[int -> word2] is mw[2]
  mw6: function[int -> word6] is mw[6]
  val2: function[word2 -> int] is val[2]
  val6: function[word6 -> int] is val[6]
  bit2: function[int, word2 -> signalval] is bit[2]
```

```
statevector: TYPE is triple

count: function[statevector -> word6] is first
double: function[statevector -> bool] is second
node: function[statevector -> word2] is third

BOOLF: function[signalval -> bool] is signal_to_bool

(* ---------------------- define logic constants ---------------------- *)

fetchnode: word2 = mw2(0)
inc1node: word2 = mw2(1)
inc2node: word2 = mw2(2)
loadnode: word2 = mw2(3)
undef_svt: statevector

(* ---------------------- define logic variables ---------------------- *)

svt: VAR statevector
ct, ldn, w: VAR word6
fn: VAR word2
dbl,b: VAR bool

(* ---------------------- define functions ---------------------- *)

ADD1: function[word6 -> word6] ==
        (LAMBDA w -> word6:
           IF val6(w) = 63 THEN mw6(0) ELSE mw6(val6(w)+1)
           END )

INC1: function[word6,bool,word6,word2 -> statevector]
INC1_ax: AXIOM INC1(ct, dbl, ldn, fn) =
             IF dbl THEN
                make_triple(ADD1(ct),BOOLF(bit2(0,fn)),inc2node)
             ELSE
                make_triple(ADD1(ct),BOOLF(bit2(0,fn)),fetchnode)
             END

INC2: function[word6,bool,word6,word2 -> statevector]
INC2_ax: AXIOM INC2(ct, dbl, ldn, fn) =
                make_triple( ADD1(ct),BOOLF(bit2(0,fn)),fetchnode)

LOAD: function[word6,bool,word6,word2 -> statevector]
LOAD_ax: AXIOM LOAD(ct, dbl, ldn, fn) =
                make_triple(ldn,BOOLF(bit2(0,fn)),fetchnode)

FETCH: function[word6,bool,word6,word2 -> statevector]
FETCH_ax: AXIOM FETCH(ct, dbl, ldn, fn)=
             IF val2(fn) = 0 THEN
                make_triple(ct,BOOLF(bit2(0,fn)),fetchnode)
             ELSIF val2(fn) = 1 THEN
                make_triple(ct,BOOLF(bit2(0,fn)),loadnode)
             ELSE
                make_triple(ct,BOOLF(bit2(0,fn)),inc1node)
             END
```

```
    NEXT: function[statevector,word6,word2 -> statevector]
    (* ------------------------------------------------------------
    NEXT_ax: AXIOM NEXT(svt,ldn,fn) =
                     IF val2(node(svt)) = 0 THEN
                        FETCH(count(svt),double(svt),ldn,fn)
                     ELSIF val2(node(svt)) = 1 THEN
                        INC1(count(svt),double(svt),ldn,fn)
                     ELSIF val2(node(svt)) = 2 THEN
                        INC2(count(svt),double(svt),ldn,fn)
                     ELSIF val2(node(svt)) = 3 THEN
                        LOAD(count(svt),double(svt),ldn,fn)
                     ELSE
                        undef_svt
                     END
    ------------------------------------------------------------ *)

    NEXT0_ax: AXIOM val2(node(svt)) = 0 IMPLIES
                    NEXT(svt,ldn,fn) = FETCH(count(svt),double(svt),ldn,fn)

    NEXT1_ax: AXIOM val2(node(svt)) = 1 IMPLIES
                    NEXT(svt,ldn,fn) = INC1(count(svt),double(svt),ldn,fn)

    NEXT2_ax: AXIOM val2(node(svt)) = 2 IMPLIES
                    NEXT(svt,ldn,fn) = INC2(count(svt),double(svt),ldn,fn)

    NEXT3_ax: AXIOM val2(node(svt)) = 3 IMPLIES
                    NEXT(svt,ldn,fn) = LOAD(count(svt),double(svt),ldn,fn)


    Finite_automata: function[statevector,word6,word2 -> statevector] =
                     (LAMBDA svt, ldn, fn -> statevector:
                          IF val2(fn) = 0 THEN
                             NEXT(svt,ldn,fn)
                          ELSIF val2(fn) = 3 THEN
                             NEXT(NEXT( NEXT(svt,ldn,fn), ldn,fn ),
                                                ldn,fn )
                          ELSE
                             NEXT( NEXT(svt,ldn,fn), ldn,fn )
                          END )

   (* --------- Mapping to Top Level Spec in Module cnt6 ------------ *)

   cnt6.states: TYPE FROM statevector

   cnt6.cnt: function[statevector -> word6] is count

   cnt6.exec_cnt: function[statevector,word6,word2 -> statevector]
                  is Finite_automata

   cnt6.ready: function[statevector -> bool] =
                (LAMBDA svt -> bool: node(svt) = fetchnode )

END cnt6_fa
```

Strengthening the Top-Level Specification

The top-level specification defines the operation of the counter when "ready" is true, i.e. when it is ready for input. But, an implementation that is never ready satisfies the specification above. The following mappings would satisfy the specification:

```
cnt6.ready: function[statevector -> bool] = FALSE

cnt6.states: TYPE
cnt6.cnt: function[statevector -> word6]
cnt6.exec_cnt: function[statevector,word6,word2 -> statevector]
```

The following property would preclude trivial implementations:

```
reset_ready_ax: AXIOM NOT ready(state) and func = 0 IMPLIES
                                ready( exec_cnt(state,loadin,func) )
```

This property would define "func=0" as a reset, i.e., regardless of which state the counter is currently in, if the counter is "executed" with "func=0" it will be returned to "fetchnode". Unfortunately, the RSRE implementation does not satisfy this property. If the counter is located at state "inc1node" and "double(state)" is true, then the counter would transition to "inc2node" in response to a "func=0" command. The following more complicated property also precludes trivial solutions and is satisfied by the RSRE implementation:

```
eventually_ready_ax: AXIOM
     NOT ready(state) and func = 0 IMPLIES
         ready( exec_cnt(exec_cnt(state,loadin,func),loadin,func) )
```

BLOCK DIAGRAM SPECIFICATION

This section describes the third level in the hierarchy -- the block diagram specification. In the RSRE work the block-diagram spec was the lowest level of the system specified in the formal language LCF-LSM. They created a second description of the block diagram in the hardware design language ELLA (ref 3.) The connection between these two theoretically equivalent specifications was informal. The connection between the ELLA specification and the lower level circuit description was done using the method of "Intelligent Exhaustion" (ref. 4.)

This specification describes the system as a block diagram illustrated in figure 3.

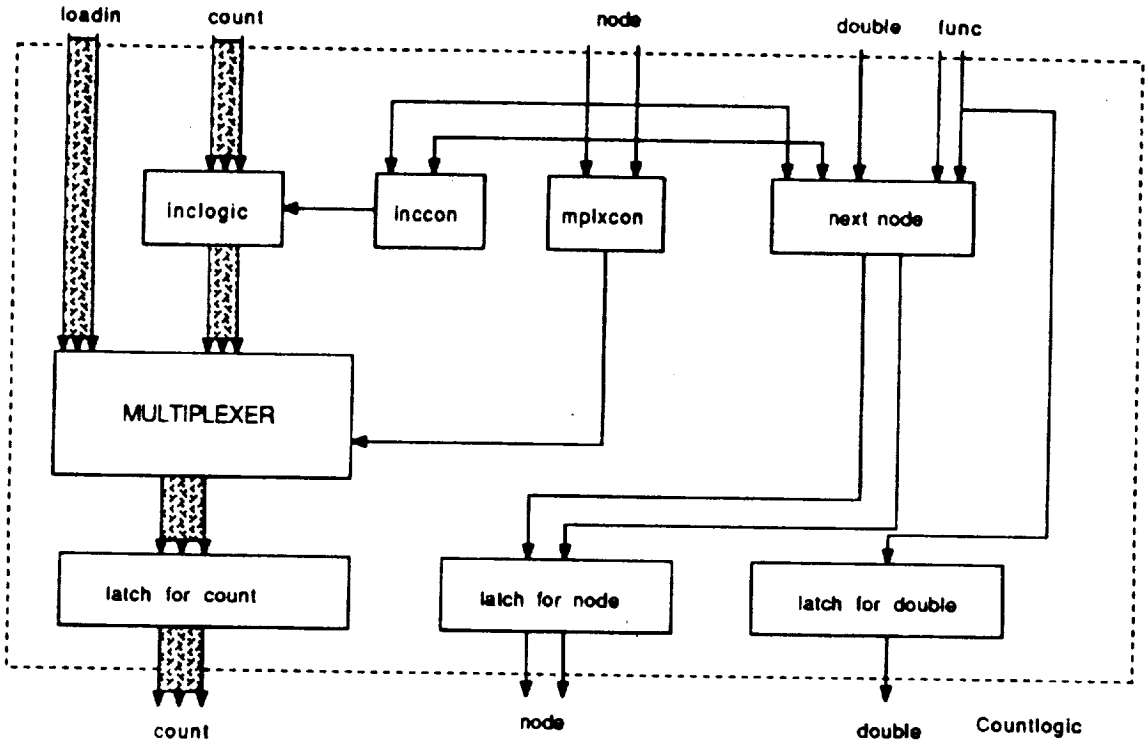


Figure 3. - Block Diagram Specification

The finite automata is implemented by the following blocks (or subcircuits):

INCLOGIC MULTIPLEX MPLXCON INCCON NEXTNODE

The internal state variables (i.e. count, node and double) are assumed to be stored by latches which maintain their values between clock ticks. This is not explicitly formalized in the RSRE methodology. Consequently, it is possible that an implementation which failed to store these variables in latches would not be detected as erroneous by the RSRE methodology. Although informally this could be checked, it is not clear how this could be detected by an automatic theorem prover.

The cnt6_blk specification without proofs is:

```
cnt6_blk: MODULE
MAPPING cnt6_fa ONTO words,triples[word[6],bool,word[2]],bsignal

THEORY

  (* ------------ define abbreviations for 'words' ------------- *)

  word2: TYPE is word[2]
  word6: TYPE is word[6]

  mw2: function[int -> word2] is mw[2]
  val2: function[word2 -> int] is val[2]
  bit2: function[int, word2 -> signalval] is bit[2]
  mw6: function[int -> word6] is mw[6]
  val6: function[word6 -> int] is val[6]
  bit6: function[int, word6 -> signalval] is bit[6]

  BOOLF: function[signalval -> bool] is signal_to_bool

  statevector: TYPE is triple

  (* ----------  logic constants defined in cnt6_fa ------------

  fetchnode: word2 = mw2(0)
  inc1node: word2 = mw2(1)
  inc2node: word2 = mw2(2)
  loadnode: word2 = mw2(3)

   ---------------- define logic variables ----------------- *)

  stv: VAR statevector
  ct,incout,loadin: VAR word6
  noinc: VAR bool
  nd,func: VAR word2
  dbl: VAR bool
  mplxsel: VAR bool

  (* ---------------- define functions  ----------------- *)

  INCLOGIC: function[word6,bool -> word6]
  INCLOGIC_ax: AXIOM INCLOGIC(ct,noinc) =
            IF noinc THEN ct
            ELSE ADD1(ct)
            END


  MULTIPLEX: function[word6,word6,bool -> word6]
  MULTIPLEX_ax: AXIOM MULTIPLEX(incout, loadin, mplxsel) =
                    IF mplxsel THEN incout
                    ELSE loadin
                    END
```

```
    MPLXCON: function[word2 -> bool] =
             (LAMBDA nd -> bool:  NOT (val2(nd) = 3) )

    INCCON: function[word2 -> bool] =
            (LAMBDA nd -> bool:  (val2(nd) = 0) )


    NEXTNODE: function[word2,word2,bool -> word2]
    (* ------------------------------------------------------------------
    NEXTNODE_ax: AXIOM NEXTNODE(nd,func,dbl) =
                   IF val2(nd) = 0 THEN
                      IF val2(func) = 0 THEN fetchnode
                      ELSIF val2(func) = 1 THEN loadnode
                      ELSE inc1node
                      END
                   ELSIF val2(nd) = 1 THEN
                      IF dbl THEN inc2node
                             ELSE fetchnode
                      END
                ELSE
                    fetchnode
                END
    ------------------------------------------------------------------ *)

    NEXTNODE0_ax: AXIOM val2(nd) = 0 IMPLIES
                     NEXTNODE(nd,func,dbl) =
                        IF val2(func) = 0 THEN fetchnode
                        ELSIF val2(func) = 1 THEN loadnode
                        ELSE inc1node
                        END

    NEXTNODE1_ax: AXIOM val2(nd) = 1 IMPLIES
                     NEXTNODE(nd,func,dbl) = IF dbl THEN inc2node
                                             ELSE fetchnode
                                             END

    NEXTNODE2a3_ax: AXIOM val2(nd) = 2 or val2(nd) = 3 IMPLIES
                       NEXTNODE(nd,func,dbl) = fetchnode

    COUNTLOGIC: function[statevector,word6,word2 -> statevector] =
                   (LAMBDA stv, loadin, func -> statevector:
                   make_triple( MULTIPLEX(INCLOGIC(count(stv),
                                                   INCCON(node(stv))    ),
                                          loadin,
                                          MPLXCON(node(stv)) ),
                                BOOLF(bit2(0,func)),
                                NEXTNODE(node(stv),func,double(stv)) )
                    )


  cnt6_fa.NEXT: function[statevector,word6,word2 -> statevector] = COUNTLOGIC

END cnt6_blk
```

SPECIFICATION OF CIRCUIT

In this section the 6-bit counter is expressed in terms of low-level circuit elements -- NAND2, INV, XNOR, etc. In the RSRE paper, this level was only defined in the ELLA language. Although the MAPPINGS to "cnt6_blk" have been included, none of the proofs between this level and the block model have yet been attempted.

Listing of Cnt6_cir

```
cnt6_cir: MODULE

MAPPING cnt6_blk ONTO words, triples, bsignal

THEORY

  (* ---------------------------- abbreviations ---------------------------- *)

  word2: TYPE is word[2]
  word6: TYPE is word[6]
  cntrlsigs: TYPE is triple[bool,bool,word[2]]

  bit2: function[int, word2 -> bool] is bit[2]
  bit6: function[int, word6 -> bool] is bit[6]
  assign2: function[int,bool,word2 -> word2] is assign[2]
  assign6: function[int,bool,word6 -> word6] is assign[6]

  (* ---------------------- circuit elements ------------------------------- *)

  b,b1,b2,b3,b4: VAR bool

  INV: function [bool -> bool] = (LAMBDA b -> bool: not b)
  NAND2: function [bool, bool -> bool] =
         (LAMBDA b1,b2 -> bool: not (b1 and b2))
  NAND3: function [bool, bool, bool -> bool] =
         (LAMBDA b1,b2,b3 -> bool: not (b1 and b2 and b3))
  NAND4: function [bool, bool, bool, bool -> bool] =
         (LAMBDA b1,b2,b3,b4 -> bool: not (b1 and b2 and b3 and b4))
  XNOR: function [bool, bool -> bool] =
         (LAMBDA b1,b2 -> bool: not (not b1 and b2 or b1 and not b2))
  NOR2: function [bool, bool -> bool] =
         (LAMBDA b1,b2 -> bool: not (b1 or b2))

  (* ---------------------- logic variables -------------------------------- *)

  i0,i1,i2,i3,i4,i5: VAR bool
  lbit,lsel,incbit,incsel: VAR bool
  incout,loadin,cntr: VAR word6
```

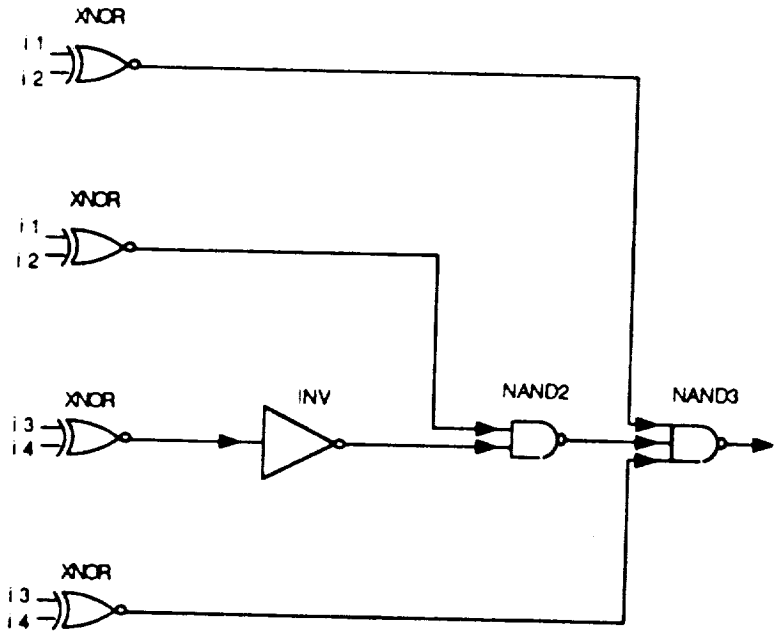
```
mplxsel,noinc,Double: VAR bool
Node,Func: VAR word2

(* ---------------------- circuit definition ---------------------------- *)

output: function [bool,bool,bool,bool,bool,bool -> word6] =
        (LAMBDA i0,i1,i2,i3,i4,i5 -> word6:
                assign6(0,i0,
                 assign6(1,i1,
                  assign6(2,i2,
                   assign6(3,i3,
                    assign6(4,i4,
                     assign6(5,i5,newword[6])))))))


bitsel: function[bool,bool,bool,bool -> bool] =
        (LAMBDA 1bit,1sel,incbit,incsel -> bool:
         NAND2( NAND2(1bit,1sel), NAND2(incbit,incsel)) )


MPLEXCIRC: function[word6,word6,bool -> word6]
MPLEXCIRC_ax: AXIOM MPLEXCIRC(incout, loadin, mplxsel) =
            output(
          bitsel(bit6(0,loadin),INV(mplxsel),bit6(0,incout),mplxsel),
          bitsel(bit6(1,loadin),INV(mplxsel),bit6(1,incout),mplxsel),
          bitsel(bit6(2,loadin),INV(mplxsel),bit6(2,incout),mplxsel),
          bitsel(bit6(3,loadin),INV(mplxsel),bit6(3,incout),mplxsel),
          bitsel(bit6(4,loadin),INV(mplxsel),bit6(4,incout),mplxsel),
          bitsel(bit6(5,loadin),INV(mplxsel),bit6(5,incout),mplxsel)
                    )

carry4bar: function[word6,bool -> bool] =
            (LAMBDA cntr,noinc -> bool:
               NAND4(INV(noinc),bit6(0,cntr),bit6(1,cntr),bit6(1,cntr))
            )


INCCIRC: function[word6,bool -> word6] =
         (LAMBDA cntr,noinc -> word6:
            output(
              XNOR(bit6(0,cntr), noinc),
              XNOR(bit6(1,cntr), NAND2(INV(noinc),bit6(0,cntr)) ),
              XNOR(bit6(2,cntr),
                   NAND3( INV(noinc), bit6(0,cntr), bit6(1,cntr) ) ),
              XNOR(bit6(3,cntr),carry4bar(cntr,noinc) ),
              XNOR(bit6(4,cntr),
                   NAND2(INV(carry4bar(cntr,noinc)), bit6(3,cntr)  )
                   ),

              XNOR(bit6(5,cntr),
                   NAND3(INV(carry4bar(cntr,noinc)),
                         bit6(3,cntr) ,
                         bit6(4,cntr)  )         )
                  )
           )
```

```
inccon: function[word2 -> bool] =
          (LAMBDA Node -> bool:
             NOR2(bit2(0,Node),bit2(1,Node))    )

common: function[word2,word2 -> bool] =
          (LAMBDA Node,Func -> bool:
             NAND3(inccon(Node),INV(bit2(1,Func)),bit2(0,Func))   )

CONTROLCIR: function[word2,word2,bool -> cntrlsigs] =
            (LAMBDA Node,Func,Double -> cntrlsigs:
               make_triple( inccon(Node),
                            NAND2(bit2(0,Node),bit2(1,Node)),
                            assign2(0, NAND2(common(Node,Func),
                                             NAND2(inccon(Node),bit2(1,Func))
                                            ),
                              assign2(1,NAND2(common(Node,Func),
                                              NAND3(Double,
                                                    bit2(0,Node),
                                                    INV(bit2(1,Node) ) )),
                                      newword[2])
                                     )
                           )
            )

(* ------------------- Mappings to "cnt6_blk" ------------------------- *)

cnt6_blk.INCLOGIC: function[word6,bool -> word6] = INCCIRC

cnt6_blk.MULTIPLEX: function[word6,word6,bool -> word6] = MPLEXCIRC

cnt6_blk.INCCON: function[word2,word2,bool -> bool] =
                   (LAMBDA Node,Func,Double -> bool:
                       first(CONTROLCIR(Node,Func,Double))
                   )

cnt6_blk.MPLXCON: function[word2,word2,bool -> bool] =
                   (LAMBDA Node,Func,Double -> bool:
                       second(CONTROLCIR(Node,Func,Double))
                   )

cnt6_blk.NEXTNODE: function[word2,word2,bool -> word2] =
                   (LAMBDA Node,Func,Double -> word2:
                       third(CONTROLCIR(Node,Func,Double))
                   )

END cnt6_cir
```

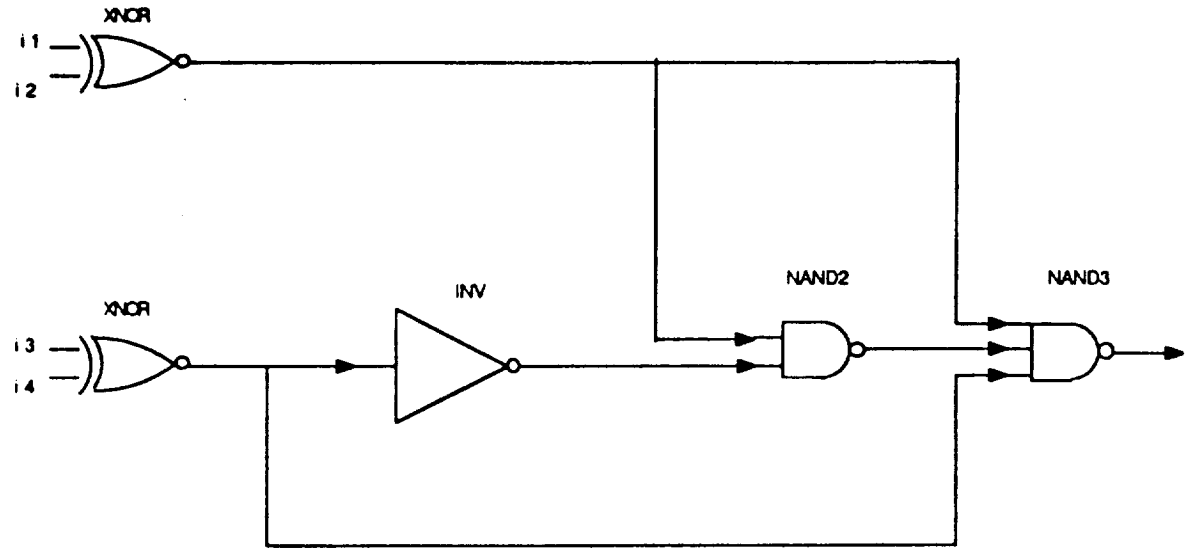
Although the circuit-level description is defined in terms of only low-level circuit elements, this level does not explicitly specify the layout of the circuit. There are many problems to be addressed here. The first is uncovering the basic element interconnections from the functional description. For example, suppose we have the following circuit specification

```
BLACK_BOX: function[bool,bool,bool,bool -> bool]
           (LAMBDA i1,i2,i3,i4 -> bool:
            NAND3( XNOR(i1,i2),
                   NAND2(XNOR(i1,i2), INV(XNOR(i3,i4)))
                   XNOR(i3,i4)
                 )
           )
```

A brute-force implementation of this function would yield:

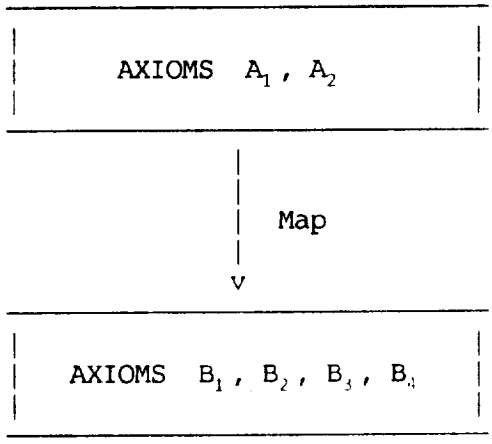


Of course, by recognizing common sub-expressions, this could be implemented as follows:



INFORMAL PROOFS

The concept of hierarchical specification depends on the idea of proving the axioms of a specification level as theorems in the level below it. One first maps the uninterpreted types and constants of the high level theory into more concrete objects in the lower level. The axioms of the high level specification are mapped down (using the mappings) to the objects of the lower level and proved as theorems there:



One must then prove that Map(A_1) and Map(A_2) follow from B_1 , B_2 , B_3 , B_4 .

In the next two sections, the proofs which establish the connection from the block model specification up to the top specification are presented informally.

Proof Between Top Level Spec and Major State Machine Spec

There were two axioms of the top level spec "cnt6": "counter_ax" and "ready_ax".

```
counter_ax: AXIOM ready(state) IMPLIES cnt(exec_cnt(state,loadin,func)) =
                  IF val2(func) = 0 THEN cnt(state)
                  ELSIF val2(func) = 1 THEN loadin
                  ELSIF val2(func) = 2 THEN
                     add1_mod64(cnt(state))
                  ELSE
                     add1_mod64(add1_mod64(cnt(state)))
                  END
```

When "counter_ax" is mapped down to the next level, the functions "ready", "cnt", and "exec_cnt", are interpreted in terms of their mapping definitions. Thus, in the lower level, the "counter_ax" is:

```
counter_ax: AXIOM node(state) = fetchnode  IMPLIES
count(Finite_automata(state,loadin,func)) =
                  IF val2(func) = 0 THEN count(state)
                  ELSIF val2(func) = 1 THEN loadin
                  ELSIF val2(func) = 2 THEN
                     add1_mod64(count(state))
                  ELSE
                     add1_mod64(add1_mod64(count(state)))
                  END
```

This must be proved as a theorem in terms of the axioms of "cnt6_fa". The basic strategy is to decompose this theorem into four cases:

Case 1: val2(func) = 0
Case 2: val2(func) = 1
Case 3: val2(func) = 2
Case 4: val2(func) = 3

First, one lemma is proved which simplifies the proof of the four cases. Next, each case is proved separately. Finally, "counter_ax" is proved from these four cases:

Proof of a Lemma

```
stb1: LEMMA ready(st) IMPLIES val2(node(st)) = 0

Proof:  By definition,  ready(st) => ( node(st) ≠ fetchnode )
                                  => ( node(st) = mw2(0) ).
 Thus,

      ready(st) ==> val2(node(st)) = val2(mw2(0))

By the "val_mw_thm" theorem of words[2] we have:

      ready(st) ==> val2(node(st)) = val2(mw2(0)) = 0

Endproof.
```

Proof of cnt_0

```
cnt_0: LEMMA ready(state) and val2(func) = 0
              IMPLIES cnt(exec_cnt(state,loadin,func)) = cnt(state)

Proof:  From the definition of "exec_cnt" and val2(func)=0 we have:

  cnt(exec_cnt(state,loadin,func)) =
          cnt( NEXT(state,loadin,func), loadin,func )

Using "NEXT_ax", the preceeding lemma, and "FETCH_ax" we have:

  cnt(exec_cnt(state,loadin,func)) =
     cnt( FETCH(cnt(state) ,doublef(state),loadin,func) ) =
     cnt( make_triple(cnt(state),BOOLF(bit2(0,func)),fetchnode) )

Finally by definition of "cnt" and the "make_triple_ax" we have:

  cnt(exec_cnt(state,loadin,func)) =
     cnt(state)

Endproof.
```

Proof of cnt_1

```
cnt_1: LEMMA ready(state) and val2(func) = 1
               IMPLIES cnt(exec_cnt(state,loadin,func)) = loadin

Proof:
  cnt(exec_cnt(state,loadin,func)) =                    (* NEXT_ax *)
  cnt( NEXT( NEXT(state,loadin,func), loadin,func )) =
  cnt( NEXT( FETCH(cnt(state) ,doublef(state),loadin,func) )) =
  cnt( NEXT( make_triple(cnt(state),BOOLF(bit2(0,func)),loadnode)
             loadin,func  )) =
(* --- Since VAL2(loadnode) = 3 --- *)
  cnt(LOAD(cnt(make_triple(cnt(state),BOOLF(bit2(0,func)),loadnode),
           doublef(make_triple(cnt(state),BOOLF(bit2(0,func)),loadnode),
           loadin,func) ) =
  cnt(LOAD(cnt(state),
           BOOLF(bit2(0,func)),
           loadin,func) ) =
  cnt( make_triple(loadin,BOOLF(bit2(0,func)),fetchnode)  ) =
  loadin
```

Proof of cnt_2

```
cnt_2: LEMMA ready(state) and val2(func) = 2
            IMPLIES cnt(exec_cnt(state,loadin,func)) =
               IF val6(cnt(state)) = 63 THEN mw6(0)
                  ELSE mw6(val6(cnt(state))+1)
                    END

Proof:
  cnt(exec_cnt(state,loadin,func)) =                          (* NEXT_ax *)
  cnt( NEXT( NEXT(state,loadin,func), loadin,func )) =
  cnt( NEXT( FETCH(cnt(state) ,doublef(state),loadin,func) )) =
  cnt( NEXT( make_triple(cnt(state),BOOLF(bit2(0,func)),inc1node)
              loadin,func  )) =
(* --- Since VAL2(inc1node) = 1 --- *)
  cnt(INC1(cnt(make_triple(cnt(state),BOOLF(bit2(0,func)),inc1node),
           doublef(make_triple(cnt(state),BOOLF(bit2(0,func)),inc1node),
           loadin,func) ) =
  cnt(INC1( cnt(state),
            BOOLF(bit2(0,func),
            loadin,func) ) =
(* --- Since bit2(0,func) = 0 ==> BOOLF(bit2(0,func)) = false --- *)
 cnt(make_triple(ADD1(cnt(state)),BOOLF(bit2(0,func)),fetchnode)) =
 ADD1(cnt(state)) =
 IF val6(cnt(state)) = 63 THEN mw6(0) ELSE mw6(val6(cnt(state))+1)
 END
```

Proof of cnt_3

```
cnt_3: LEMMA ready(state) and val2(func) = 3
               IMPLIES cnt(exec_cnt(state,loadin,func)) =
                       IF val6(cnt(state)) = 63 THEN mw6(1)
                       ELSIF val6(cnt(state)) = 62 THEN mw6(0)
                       ELSE mw6(val6(cnt(state))+2)
                       END
```

```
Proof:
  cnt(exec_cnt(state,loadin,func)) =                          (* NEXT_ax *)
  cnt( NEXT( NEXT( NEXT(state,loadin,func), loadin,func ), loadin,func)) =
  cnt( NEXT( NEXT( FETCH(cnt(state) ,doublef(state),loadin,func)),
             loadin,func) =
  cnt( NEXT( NEXT( make_triple(cnt(state),BOOLF(bit2(0,func)),inc1node)
              loadin,func  ), loadin,func )) =
(* --- Since VAL2(inc1node) = 1 --- *)
  cnt(NEXT(
      INC1(cnt(make_triple(cnt(state),BOOLF(bit2(0,func)),inc1node),
           doublef(make_triple(cnt(state),BOOLF(bit2(0,func)),inc1node),
           loadin,func),loadin,func ) ) =
  cnt(NEXT( INC1( cnt(state),
                  BOOLF(bit2(0,func),
                  loadin,func),loadin,func) ) =
(* --- Since bit2(0,func) = 1 ==> BOOLF(bit2(0,func)) = true --- *)
 cnt(NEXT(make_triple(ADD1(cnt(state)),BOOLF(bit2(0,func)),inc2node),
         loadin,func) ) =
 cnt(INC2(cnt(make_triple(ADD1(cnt(state)),BOOLF(bit2(0,func)) ),
          doublef( make_triple(ADD1(cnt(state)),BOOLF(bit2(0,func)) ),
          loadin,func)) =
 cnt(INC2(ADD1(cnt(state)),
          BOOLF(bit2(0,func)),
          loadin,func ) ) =
 cnt( make_triple(ADD1((ADD1(cnt(state))),BOOLF(bit2(0,func)),fetchnode) ) =
 ADD1((ADD1(cnt(state))) =
 ADD1(IF val6(ADD1(cnt(state))) = 63 THEN mw6(0)
      ELSE mw6(val6(ADD1(cnt(state)))+1) END )

 IF val6(IF val6(ADD1(cnt(state))) = 63 THEN mw6(0)
         ELSE mw6(val6(ADD1(cnt(state)))+1) END) = 63 THEN mw6(0)
 ELSE mw6(val6(IF val6(ADD1(cnt(state))) = 63 THEN mw6(0)
               ELSE mw6(val6(ADD1(cnt(state)))+1) END)+1)
 END
```

```
IF val6(cnt(state)) = 63 THEN mw6(1)
ELSIF val6(cnt(state)) = 62 THEN mw6(0)
ELSE mw6(val6(cnt(state))+2)
END
```

Proof of the cnt6 axioms

```
ready_ax: AXIOM ready(state) IMPLIES ready( exec_cnt(state,loadin,func) )

counter_ax: AXIOM ready(state) IMPLIES cnt(exec_cnt(state,loadin,func)) =
                  IF val2(func) = 0 THEN cnt(state)
                  ELSIF val2(func) = 1 THEN loadin
                  ELSIF val2(func) = 2 THEN
                     add1_mod64(cnt(state))
                  ELSE
                     add1_mod64(add1_mod64(cnt(state)))
                  END
```

The "counter_ax" follows from "cnt_0", "cnt_1", "cnt_2", and "cnt_3" and the "val_range_thm" applied to "func". The "val_range_thm" is needed to establish that "func" can only be equal to 0, 1, 2, or 3. Thus the "ELSE" clause in "counter_ax" applies to "func" = 3 only. Thus, counter_ax follows directly from cnt_0, cnt_1, cnt_2, cnt_3, and val_range_thm[2]. The axiom "ready_ax" is proved from the same lemmas.

Proof Between Major State Machine Spec and Block Model Spec

In this section the connection between the major state machine model and the block diagram spec is demonstrated via informal proof. The following axioms of the major state machine model must be proved as theorems in the Block Model:

```
NEXT0_ax: AXIOM val2(node(stv)) = 0 IMPLIES
                NEXT(stv,ldn,fn) = FETCH(count(stv),double(stv),ldn,fn)

NEXT1_ax: AXIOM val2(node(stv)) = 1 IMPLIES
                NEXT(stv,ldn,fn) = INC1(count(stv),double(stv),ldn,fn)

NEXT2_ax: AXIOM val2(node(stv)) = 2 IMPLIES
                NEXT(stv,ldn,fn) = INC2(count(stv),double(stv),ldn,fn)

NEXT3_ax: AXIOM val2(node(stv)) = 3 IMPLIES
                NEXT(stv,ldn,fn) = LOAD(count(stv),double(stv),ldn,fn)
```

The function "NEXT" is mapped onto "COUNTLOGIC" at this level, so each of these axioms must be proved with respect to the "COUNTLOGIC" implementation:

```
-------------------------------------------------------------------------
  NEXT0_ax: AXIOM val2(node(stv)) = 0 IMPLIES
                  COUNTLOGIC(stv,ldn,fn) = FETCH(count(stv),double(stv),ldn,fn)
-------------------------------------------------------------------------
```

Proof:

```
  COUNTLOGIC(stv, loadin, func) =
          make_triple(MULTIPLEX(INCLOGIC(count(stv),INCCON(node(stv))),
                                         loadin,
                                         MPLXCON(node(stv))),
                                BOOLF(bit2(0,func)),
                                NEXTNODE(node(stv),func,double(node)) ) =

{ by definition of INCCON and MPLXCON: }

                  make_triple( MULTIPLEX(INCLOGIC(count(stv),
                                                  (val2(node(stv)) = 0) ),
                                         loadin,
                                         NOT (val2(node(stv)) = 3)),
                               BOOLF(bit2(0,func)),
                               NEXTNODE(node(stv),func,double(node)) ) =

                  make_triple( MULTIPLEX(INCLOGIC(count(stv),
                                                  true ),
                                         loadin,
                                         true ),
                               BOOLF(bit2(0,func)),
                               NEXTNODE(node(stv),func,double(node)) ) =

{ by INCLOGIC_ax: }

                  make_triple( MULTIPLEX(count(stv),
                                         loadin,
                                         true ),
                               BOOLF(bit2(0,func)),
                               NEXTNODE(node(stv),func,double(node)) ) =

{ by MULTIPLEX_ax: }

                  make_triple( count(stv),
                               BOOLF(bit2(0,func)),
                               NEXTNODE(node(stv),func,double(node)) ) =

{ by FETCH_ax: }

                 FETCH(cnt(stv),doublef(stv),loadin,func) =
```

```
( The last step follows from the fact that val2(node(stv) = 0 IMPLIES that

    NEXTNODE(node(stv),func,double(node)) ) is an element of
              {fetchnode,loadnode,inclnode}

Endproof

-------------------------------------------------------------------------

  NEXT1_ax: AXIOM val2(node(stv)) = 1 IMPLIES
                  COUNTLOGIC(stv,ldn,fn) = INC1(count(stv),double(stv),ldn,fn)
-------------------------------------------------------------------------

Proof:

  COUNTLOGIC(stv, loadin, func) =
                  make_triple( MULTIPLEX(INCLOGIC(count(stv),INCCON(node(stv))),
                                         loadin,
                                         MPLXCON(node(stv))),
                               BOOLF(bit2(0,func)),
                               NEXTNODE(node(stv),func,double(node)) ) =

{ by definition of MPLXCON: }

                  make_triple( MULTIPLEX(INCLOGIC(count(stv),INCCON(node(stv))),
                                         loadin,
                                         NOT(val2(node(stv))=3) ),
                               BOOLF(bit2(0,func)),
                               NEXTNODE(node(stv),func,double(node)) ) =

{ by INCLOGIC_ax: }

                  make_triple( MULTIPLEX(ADD1(count(stv),
                                         loadin,
                                         true),
                               BOOLF(bit2(0,func)),
                               NEXTNODE(node(stv),func,double(node)) ) =

{ by MULTIPLEX_ax: }

                   make_triple( ADD1(count(stv),
                                BOOLF(bit2(0,func)),
                                NEXTNODE(node(stv),func,double(node)) ) =

{ by INC1_ax: }

                  INC1(cnt(stv),double(stv),loadin,func) =

( The last step follows from the fact that val2(node(stv) = 1 IMPLIES that

    NEXTNODE(node(stv),func,double(node)) ) is an element of
              {fetchnode,inc2node}

Endproof
```

```
NEXT2_ax: AXIOM val2(node(stv)) = 2 IMPLIES
                COUNTLOGIC(stv,ldn,fn) = INC2(count(stv),double(stv),ldn,fn)
```

Proof:

```
  COUNTLOGIC(stv, loadin, func) =
                make_triple( MULTIPLEX(INCLOGIC(count(stv),INCCON(node(stv))),
                                       loadin,
                                       MPLXCON(node(stv))),
                             BOOLF(bit2(0,func)),
                             NEXTNODE(node(stv),func,double(node)) ) =

                make_triple( MULTIPLEX(INCLOGIC(count(stv),INCCON(node(stv))),
                                       loadin,
                                       NOT(val2(node(stv))=3) ),
                             BOOLF(bit2(0,func)),
                             NEXTNODE(node(stv),func,double(node)) ) =
[ by INCLOGIC_ax: ]

                make_triple( MULTIPLEX(ADD1(count(stv),
                                       loadin,
                                       true),
                             BOOLF(bit2(0,func)),
                             NEXTNODE(node(stv),func,double(node)) ) =
[ by MULTIPLEX_ax: ]

                make_triple( ADD1(count(stv),
                             BOOLF(bit2(0,func)),
                             NEXTNODE(node(stv),func,double(node)) ) =
[ by NEXTNODE2a3_ax: ]

                make_triple( ADD1(count(stv)),
                             BOOLF(bit2(0,func)),
                             fetchnode ) =
{ by INC2_ax: }

               INC2(cnt(stv),double(stv),loadin,func) =
```

```
NEXT3_ax: AXIOM val2(node(stv)) = 3 IMPLIES
               COUNTLOGIC(stv,ldn,fn) = LOAD(count(stv),double(stv),ldn,fn)
```

```
COUNTLOGIC(stv, loadin, func) =
                make_triple(MULTIPLEX(INCLOGIC(count(stv),INCCON(node(stv))),
                                      loadin,
                                      MPLXCON(node(stv))),
                            BOOLF(bit2(0,func)),
                            NEXTNODE(node(stv),func,double(node)) ) =

                make_triple(MULTIPLEX(INCLOGIC(count(stv),val2(node(stv))=0),
                                      loadin,
                                      NOT(val2(node(stv))=3) ),
                            BOOLF(bit2(0,func)),
                            NEXTNODE(node(stv),func,double(node)) ) =
[ by INCLOGIC_ax: ]

                make_triple(MULTIPLEX((count(stv)
                                      loadin,
                                      NOT(val2(node(stv))=3) ),
                            BOOLF(bit2(0,func)),
                            NEXTNODE(node(stv),func,double(node)) ) =
[ by MULTIPLEX_ax: ]

                make_triple( loadin,
                            BOOLF(bit2(0,func)),
                            NEXTNODE(node(stv),func,double(node)) ) =

[ by NEXTNODE2a3_ax: ]

                make_triple( loadin,
                            BOOLF(bit2(0,func)),
                            fetchnode ) =

{ by LOAD_ax: }

                  LOAD(cnt(stv),double(stv),loadin,func) =
```

Proof Between Block Diagram Spec and Circuit-Level Spec

In the RSRE methodology, the proof between the block-diagram specification and the circuit-level specification is accomplished by the method of intelligent exhaustion (ref. 4) or the more recent "NODEN" method (ref 5). The proofs at this level have not yet been attempted. Future work will investigate the advantages and disadvantages of "NODEN" versus EHDM proof.

SPECIFICATION OF N-BIT WORDS

A physical row of input or output lines are often interpreted as integers in some hardware devices. Of course the range of integer values which can be presented on N wires is finite, usually taken to be 0 to 2^N-1. It is necessary to build a theory which enables one to reason about such rows of signal values as integers, in a simple manner. This section describes such a theory. In this theory a row of N inputs is referred to as a "N-bit word". This theory has been defined in a separate module "words" to facilitate its reuse. This module should be usable by most hardware verification projects without modification.

It is necessary to first define the possible signal values which can appear on a single line. For simplicity, the set of boolean values: {true, false} are used to represent signal values in "words". In the RSRE reports, the domain of signal values range over t, f, x, z, q and i, which stand for true, false, don't care, tri-state high impedance, unaltered memory element and indeterminate, respectively. The x, z, q, and i values were not needed to verify the counter in EHDM so the simpler boolean domain was used. Appendix A shows how the theory of words can be generalized to include these other values.

Conceptually, a word consists of N bits which are indexed by an integer between 0 and N-1 inclusive. Thus, the module is defined in terms of a generic parameter "N" which is the number of bits in the word. This module exports the type "word[N]". If only one type of word will be used in a specification, the user can declare the size of the words in a USING clause, e.g.

```
USING words[32]
```

and the identifier "word" can be used instead of word[32]. If more than one word type is needed, then the following using clause is used:

```
USING words
```

and the user must cite the length of the word explicitly, e.g. word[32], word[16], etc. This module also defines the following functions:

val(w): returns the unsigned integer value of the N-bit word "w"

mw(i): returns an N-bit word containing the binary representation of the unsigned integer "i".

bit(i,w): returns the contents of the "ith" bit of word "w"

assign(i,b,w): assigns the boolean value "b" to the "ith" bit of word "w"

If there is only one declaration of the "words" module, then the above functions cab be abbreviated as "val", "mw", "bit" and "assign". If there are multiple declarations of the "words" module, then the function names cannot be abbreviated, e.g.:

val[32](w): returns the unsigned integer value of the 32-bit word "w"

val[12](w): returns the unsigned integer value of the 12-bit word "w"

bit[16](i,w): returns the contents of the "ith" bit of the 16-bit word "w"

assign[12](i,b,w): assigns "b" to the "ith" bit of the 12-bit word "w"

The "bit" and "assign" functions enable the access and modification of individual bits of a N-bit word. These functions are defined formally as follows:

```
bitassign: AXIOM (i >= 0 and i < N) IMPLIES
                      bit(i,assign(k,b,w)) =
                         ( IF k = i THEN b ELSE bit(i,w) END )
```

Thus, bit and assign are defined in terms of each other. The axiom defines the effect of retrieving a bit from a word which has been modified by assigning a new value to one of its bits. If the bit being retrieved is the same as the one just assigned, the new value is retrieved. Otherwise, the value retrieved is the same as before the assignment. Thus, assigning a bit in a word does not affect any other bit in the word. It should be noted that these functions are not defined in a "constructional" manner; that is, they have not been defined separately in terms of previously defined primitives. Their properties have been defined axiomatically in terms of each other. Such axioms must be carefully scrutinized to insure that inconsistencies are not introduced into the specification.

This method of defining a word differs considerably from the way they were defined in the RSRE report using LCF-LSM. In LCF-LSM there are specific built-in functions that manipulate lists of objects. In the RSRE work, a word is

represented by a list of objects. Thus, they "construct" a word using more primitive functions.

In the top-level specification of the 6-bit counter, its behavior is defined in terms of modulo-64 arithmetic over integers. Thus, it is necessary to define functions which "interpret" an N-bit word as an integer. The "val" and "mw" functions perform this duty. The "val" function is defined recursively as follows:

```
valm: function[word,int,int -> int]
valm_def: AXIOM valm(w,m,n) =  IF m = 0 THEN 0
                               ELSE 2*valm(w,m-1,n) + BOOLVAL(bit(n-m,w))
                               END

val: function[word -> int]
val_def: AXIOM val(w) = valm(w,N,N)
```

Similarly, the "mw" function is defined recursively:

```
mw: function[int -> word]
mwm: function[int,int,int -> word]

mwm_def: AXIOM mwm(v,m,n) = IF m = 0 THEN newword
                            ELSE
                               assign(n-m,BOOLVAR(MOD2(v)),
                                      mwm(DIVBY2(v),m-1,n) )
                            END
```

The constant "newword" used in "mwm_def" above represents an undefined word. It is defined formally as:

```
accessnew: AXIOM bit(k, newword) = f
```

The major theorem of this module establishes that "val" and "mw" are inverse functions:

```
( ii >= 0 AND ii < power2(N) ) IMPLIES val(mw(ii)) = ii
```

The specification of "words" is:

```
words: MODULE[N: int]

USING power2_th,divby2_th

EXPORTING word, newword, bit, assign, val, mw, mwm, valm, bool_to_int
  WITH power2_th,divby2_th
```

```
ASSUMING
  N_pos: FORMULA N>0

THEORY
  word: TYPE
  k,i,ii,m,v,n: VAR int
  w,w1,w2: VAR word
  b: VAR bool
  newword: word

  assign: function[int, bool, word -> word]
  bit: function[int, word -> bool]

  bitassign: AXIOM (i >= 0 and i < N) IMPLIES
                     bit(i,assign(k,b,w)) =
                          ( IF k = i THEN b ELSE bit(i,w) END )

  mw: function[int -> word]
  val: function[word -> int]


  mwm: function[int,int,int -> word]
  mwm_def: AXIOM mwm(v,m,n) =
                 IF m = 0 THEN newword
                 ELSE
                    assign(n-m,BMOD2(v),mwm(DIVBY2(v),m-1,n) )
                 END

  mw_def: AXIOM mw(ii) = mwm(ii,N,N)

  bool_to_int: function[bool -> int] =
                (LAMBDA b -> int: IF b THEN 1 ELSE 0 END )

  valm: function[word,int,int -> int]
  valm_def: AXIOM valm(w,m,n) = IF m = 0 THEN 0
                                ELSE 2*valm(w,m-1,n) + bool_to_int(bit(n-m,w))
                                END

  val_def: AXIOM val(w) = valm(w,N,N)

  (* ------------------------- Big Theorems ----------------------------- *)

  val_mw_thm: THEOREM ( ii >= 0 AND ii < power2(N) )
                    IMPLIES val(mw(ii)) = ii

  val_range_thm: THEOREM val(w) >= 0 and val(w) < power2(N)

  val_bits_thm: THEOREM val(w1) = val(w2) IMPLIES
                    (FORALL m: m>=0 AND m<N IMPLIES bit(m,w1)=bit(m,w2))

END words
```

The proofs of the theorems are listed in Appendix C.

FORMAL PROOFS

In this section a brief overview is given of the formal proofs performed using the EHDM theorem prover. The automatic theorem prover is used to guarantee that no errors have been made in the proofs themselves.

Introduction to Proving in EHDM

Proving is accomplished in EHDM by reducing the problem to the decidable domain of the theorem prover by citing all premises which the theorem depends upon and "instantiating" the variables of the theorem and premises. To illustrate this process, the cnt_0 proof will be examined. From the informal proof (see section entitled "INFORMAL PROOFS") we can see that the theorem follows from NEXT_ax, FETCH_ax, stb1, and make_triple_ax. The first step to proving the theorem is to list these premises. Next, the variable names in the premises must be "matched". This is accomplished by "instantiating" (i.e. substituting) the variables in the premises with the same names as in the conclusion. The first premise used was NEXT_ax. NEXT_ax has three variables which must be "instantiated", namely maj, loadin, and func.[7] The cnt_0 formula calls the "NEXT" function with arguments "state", "loadin", and "func". Therefore, these are the values that must be assigned to the "NEXT_ax" variables. This is done as follows:

```
maj <- state@C,
loadin <- loadin@C,
func <- func@C
```

The @C is used to indicate that the names come from the conclusion. (Names from premise one are designated by @P1, premise two by @P2 and so forth.) This matching process is done for all of the premises. The formal proof is:

---

7 Substitutable variables are those that are universally quantified in the premises and existentially quantified in the conclusion.

```
p_cnt_0: PROVE cnt_0 FROM NEXT_ax{maj <- state@C,
                                  loadin <- loadin@C,
                                  func <- func@C},
                         FETCH_ax{count <- cnt(state@C),
                                  double <- doublef(state@C),
                                  loadin <- loadin@C,
                                  func <- func@C},
                          make_triple_ax{x <- cnt(state@C),
                                         y <- BOOLF(bit2(0,func@C)),
                                         z <- fetchnode},
                         stbl{st <- state@C}
```

Proof statements are included in the specification module in the last section which follows the reserved word "PROOF". The module is subjected to the theorem prover. The prover attempts to prove each of the theorems referenced by a "PROVE" statement. The prover returns either "PROVED" or "UNPROVED". A proof trace is supplied by the theorem prover to aid the user in accomplishing a proof.

Some suggestions for improving the EHDM Theorem Prover are given in Appendix B.

Status of Proofs

All of the proofs between the Top Level and the Major-State Level and Between the Major-state level and the Block Model level have been completed. A complete listing of the specifications and proofs are given in Appendix C. The status reports generated by EHDM are listed below:

Proofs Between Top-level and Major-state level

The proof chain is complete

The following formulas were justified only as specific instances

```
words[&1].N_pos
```

The axioms and assumptions at the base are:

```
cnt6_fa.INC2_ax*
cnt6_fa.NEXT2_ax*
cnt6_fa.INC1_ax*
cnt6_fa.NEXT1_ax*
cnt6_fa.LOAD_ax*
cnt6_fa.NEXT3_ax*
triples[&1, &2, &3].make_triple_ax
```

```
  cnt6_fa.FETCH_ax*
  cnt6_fa.NEXT0_ax*
  divby2_th.alt_ax*
  divby2_th.kill_ax*
  words[&1].valpos_ax*
  words[&1].twen_ax*
  words[&1].mw_def
  words[&1].weir_ax*
  divby2_th.tfun_ax*
  int_inductions.int_induct_by_2
  divby2_th.ifun_ax*
  words[&1].qfun_ax*
  words[&1].vfun_ax*
  divby2_th.DIV_ax
  divby2_th.BMOD2_ax
  divby2_th.MOD2_ax
  words[&1].bitassign
  words[&1].mwm_def
  words[&1].zfun_ax*
  words[&1].val_def
  int_inductions.int_induction
  words[&1].valm_def
  words[&1].rang_ax*
  power2_th.power2_ax
```

* denotes the axioms which are merely name definitions (see section entitled "Definitional Axioms" in APPENDIX B. The critical axioms are

```
  triples[&1, &2, &3].make_triple_ax
  words[&1].mw_def
  int_inductions.int_induct_by_2
  divby2_th.DIV_ax
  divby2_th.BMOD2_ax
  divby2_th.MOD2_ax
  words[&1].bitassign
  words[&1].mwm_def
  words[&1].val_def
  int_inductions.int_induction
  words[&1].valm_def
  power2_th.power2_ax
```

Proofs Between Major-state level and Block-model level

The proof chain is complete

The axioms and assumptions at the base are:

```
  cnt6_fa.LOAD_ax*
  cnt6_blk.NEXTNODE2a3_ax*
  cnt6_fa.INC2_ax*
  cnt6_blk.NEXTNODE1_ax*
  cnt6_fa.INC1_ax*
  cnt6_blk.NEXTNODE0_ax*
  cnt6_fa.FETCH_ax*
```

cnt6_blk.MULTIPLEX_ax*
cnt6_blk.INCLOGIC_ax*

* denotes the axioms which are merely name definitions (see section entitled "Definitional Axioms" in APPENDIX B.

Status of modules in context

cnt6_cir: Parsed and Typechecked
signal: Parsed and Typechecked
cnt6: Parsed and Typechecked
cnt6_fa: Parsed and Typechecked, 31 proofs, 31 attempted, 31 succeeded
ineq_cases: Parsed and Typechecked, 5 proofs, 5 attempted, 5 succeeded
divby2_th: Parsed and Typechecked, 29 proofs, 29 attempted, 29 succeeded
power2_th: Parsed and Typechecked, 7 proofs, 7 attempted, 7 succeeded
words_prf: Parsed
cnt6_blk: Parsed and Typechecked, 14 proofs, 14 attempted, 14 succeeded
bsignal: Parsed and Typechecked
words: Parsed and Typechecked, 58 proofs, 58 attempted, 58 succeeded
int_inductions: Parsed and Typechecked
triples: Parsed and Typechecked

CONCLUSIONS

The RSRE methodology appears to be a practical approach to designing and verifying digital hardware. No major problems have been discovered in the methodology thus far. The work of this paper has focused on the hierarchical specification method. Future work will concentrate on the method of intelligent exhaustion.

There was one property which should be demonstrated about sequential circuits that was not explicitly dealt with in the RSRE papers -- "readiness" of the finite state automata. The "readiness" property refers to whether the finite automata always returns to the fetch state after executing a function. Although RSRE's hand proofs clearly established this property, the property was not specified formally.

The RSRE report did not formalize in LCF-LSM the timing behavior of the counter. Although some timing diagrams were provided and some informal remarks were made about "hoisting exiting conditions", it is not clear how this material could be formalized. Although omitting timing details from the LCF-LSM specifications significantly reduces the complexity of the specifications, it raises the possibility that certain design errors could go undetected. For example, if the state variables were not stored in latches (e.g., connected via direct feedback line), the automatic theorem prover would not report this

deficiency. This appears to be an implicit assumption of the overall methodology which should be more carefully documented.[8]

The EHDM system was found to be fully capable of supporting the RSRE methodology for hardware verification. Two features of the EHDM system were found to be especially useful -- generic modules and the MAPPING constructs. The generic module capability provided a convenient method of defining the theory of words. This stands in contrast to LCF_LSM where the "almost identical" text must be repeated which define "val2", "val6", "val32", etc. The MAPPING constructs enabled the user to formally connect the levels of the system hierarchy. This was only accomplished informally in the RSRE report.

REFERENCES

1. Cullyer, W. J.; and Pygott, C. H.: Hardware Proofs Using LCF-LSM and ELLA, RSRE Memorandum No. 3832, Royal Signals and Radar Establishment, Sept. 1985

2. F. W. von Henke; J.S. Crow; R. Lee; J.M. Rushby; R.A. Whitehurst: The EHDM verification environment: an overview. Proc. of the 11th NBS/NCSC National Computer Security Conference, Baltimore, October 1988.

3. Morison, J. D.; Peeling N. E.; Thorp T. L. : ELLA: Hardware Description Or Specification?, Proc. IEEE International Conference CAD-84, Santa Clara, Nov. 12-15, 1984.

4. Pygott, C. H.: Formal Proof of Correspondence between the specification of a hardware module and its gate level implementation, RSRE Report No. 85012, Royal Signals and Radar Establishment, Sept. 1985

5. Pygott, C. H.: NODEN: An Engineering Approach to Hardware Verification, IFIPS Hardware Verification Conference, July 1988.

---

8 A simple program could be writen which examines the circuit-level specification to insure that all feedback paths contain a one clock delay device (e.g. a latch).

APPENDIX A

THEORY OF GENERAL WORDS

In this section, a general theory of words is developed. This theory defines the concept of a N-bit word where each bit can take values from a more general domain than the booleans. Although the theory developed does not depend upon the specific domain, the following values are of typical interest:

t -- true
f -- false
x -- don't care
q -- unaltered memory element
z -- tristate impedance high
i -- indeterminate

The following specification defines these values

```
signalval: TYPE
t,f,x,q,z,i: signalval

unique: AXIOM t ~= f and t ~= x and t ~= q and t ~= z and t ~= i and
              f ~= x and f ~= q and f ~= z and f ~= i and
              x ~= q and x ~= z and x ~= i and
              q ~= z and q ~= i and
              z ~= i and

ss: VAR signalval
exhaust:   AXIOM x = t or x = f or x = x or x = q or x = z or x = i
```

Since EHDM has no method of defining a new domain of values automatically, the user must manually define its value and explicitly state the property of uniqueness and completeness.

The properties of words over the domain of "signalval" are defined in the same manner as words defined over booleans.

```
BitAssign: AXIOM (i < N and i >= 0) IMPLIES
                    ( IF k = i THEN Bit(i,Assign(k,s,gw))= s
                      ELSE Bit(i,Assign(k,s,gw))= Bit(i,gw) END )
```

This is essentially the same as "bitassign" in the boolean words theory. The function names are capitalized to distinguish them from the boolean word

functions since EHDM does not support overloading of function names. The axiom defines the effect of retrieving a bit from a word which has been modified by assigning a new value to one of its bits. If the bit being retrieved is the same as the one just assigned, the new value is retrieved. Otherwise, the value retrieved is the same as before the assignment. It should be noted that these functions are not defined in a "constructional" manner, that is, they have not been defined separately in terms of previously defined primitives. Their properties have been defined axiomatically in terms of each other. Such axioms must be carefully scrutinized to insure that inconsistencies are not introduced into the specification.

Next, the functions "val" and "mw" are defined in the general theory of words. The "val" and "mw" functions "interpret" the N bits of boolean values as an integer. Consequently they are only defined for general words that contain only values of "t" and "f". They must be defined as partial functions. This is accomplished by defining a function which embeds the boolean words in the set of general words:

```
embed: function[word -> gword]
embed_ax: AXIOM (i < N and i >= 0) IMPLIES
                Bit(i,embed(w)) = bool_to_signal(bit(i,w))
```

There are now two distinct types -- word and gword -- which represent boolean words and general words respectively. The function bool_to_signal associates the boolean values with "t" and "f" of "signalval":

```
bool_to_signal: function[bool -> signalval] =
        (LAMBDA bb -> signalval:
            IF bb THEN t
            ELSE f
            END)
```

Thus, the function "embed" maps boolean words to the corresponding general word which consists of only "t" and "f" values. Using the "embed" function, the partial functions "Val" and "Mw" can be defined:

```
Val: function[gword -> int]
Val_ax: AXIOM Val(embed(w)) = val(w)

Mw: function[int -> gword]
Mw_ax: AXIOM Mw(ii) = embed(mw(ii))
```

The theorems of the "words" module are easily established in the theory of general words. For example:

```
Val_Mw_thm: THEOREM ( ii >= 0 AND ii < power2(N) )
                IMPLIES Val(Mw(ii)) = ii

Proof:  Val(Mw(ii)) = Val(embed(mw(ii))
                    = val(mw(ii)
                    = ii
```

The last step follows from the "val_mw_thm" theorem of "words", the boolean word theory.

The full specification of "gwords" follows:

```
gwords: MODULE[N: int]

USING words,power2_th,signal,divby2_th

EXPORTING gword, newgword, Valuable, Assign, Bit
  WITH power2_th,signal,divby2_th

ASSUMING
  N_pos: FORMULA N>0

THEORY
 (* ---------------------- abbreviations for words items ------------------- *)

  word: TYPE is word[N]

  assign: function[int, bool, word -> word] is assign[N]
  bit: function[int, word -> bool] is bit[N]
  mw: function[int -> word] is mw[N]
  val: function[word -> int] is val[N]

  (* ---------------- Theory needed to define words functions ------------- *)

  gword: TYPE
  newgword: gword
  k,i,ii: VAR int
  w: VAR word
  gw, gw2: VAR gword
  s: VAR signalval
  a,b: VAR bool

  Assign: function[int, signalval, gword -> gword]
  Bit: function[int, gword -> signalval]

  Bit_Assign_ax: AXIOM (i < N and i >= 0) IMPLIES
                    ( IF k = i THEN Bit(i,Assign(k,s,gw))= s
                      ELSE Bit(i,Assign(k,s,gw))= Bit(i,gw) END )
```

```
    accessnew: AXIOM Bit(k,newgword) = x

    (* ------- Concepts related to interpretation of words as integers ------ *)

    embed: function[word -> gword]
    embed_ax: AXIOM (i < N and i >= 0) IMPLIES
                    Bit(i,embed(w)) = bool_to_signal(bit(i,w))

    Val: function[gword -> int]
    Val_ax: AXIOM Val(embed(w)) = val(w)

    Mw: function[int -> gword]
    Mw_ax: AXIOM Mw(ii) = embed(mw(ii))

    Valuable: function[gword -> bool]
    Valuable_def: AXIOM Valuable(gw2) = (EXISTS w: gw2 = embed(w))

    Val_Mw_thm: AXIOM ( ii >= 0 AND ii < power2(N) )
                      IMPLIES Val(Mw(ii)) = ii

    Bit_bit_thm: THEOREM (i < N and i >= 0) IMPLIES
                           bit(i,w) = signal_to_bool(Bit(i,embed(w)))

    Assign_assign_thm: THEOREM (i < N and i >= 0) IMPLIES
                      embed(assign(i,b,w)) = Assign(i,bool_to_signal(b),embed(w))

    Valuable_mw: THEOREM Valuable(embed(mw(ii)))

    Valuable_thm: THEOREM Valuable(gw) = ((i>=0) and (i < N) IMPLIES
                                          (Bit(i,gw) = t or Bit(i,gw) = f))

    Val_range_thm: AXIOM Valuable(gw) IMPLIES
                         Val(gw) >= 0 and Val(gw) < power2(N)

  PROOF

    p_Val_Mw_thm: PROVE Val_Mw_thm FROM Mw_ax,
                                        Val_ax{w <- mw(ii)},
                                        val_mw_thm[N]

    p_Val_range_thm: PROVE Val_range_thm{gw <- embed(w@p3)} FROM
                                        Valuable_def{gw2 <- embed(w@P3)},
                                        Val_ax{w <- w@P3},
                                        val_range_thm[N]

    p_Valuable_mw: PROVE Valuable_mw FROM Valuable_def

    p_Bit_bit_thm: PROVE Bit_bit_thm FROM embed_ax,
                                          signal_to_bool_ax

    p_Assign_assign_thm: PROVE Assign_assign_thm FROM
                                       Bit_Assign_ax,
                                       embed_ax{w <- assign(i,b,w)}
END gwords
```

```
signal: MODULE

EXPORTING signalval, t, f, x, (* EQsig, AND3, OR3, NOT3, EQUIV *)
          bool_to_signal, signal_to_bool, bool_to_int, BOOLF
          (* , int_to_signal , signal_to_int *)

THEORY

   signalval: TYPE
   t,f,x: signalval
   a,b,c: VAR signalval
   bb: VAR bool
   i: VAR int

   unique: AXIOM (t ~= f) and (t ~= x) and (f ~= x)

   exhaust: AXIOM a=t OR a=f OR a=x

   signal_to_bool: function[signalval -> bool]
   signal_to_bool_ax: AXIOM signal_to_bool(t) = true and
                            signal_to_bool(f) = false

   bool_to_signal: function[bool -> signalval] =
           (LAMBDA bb -> signalval:
               IF bb THEN t
               ELSE f
               END)

   BOOLF: function[signalval -> bool] is signal_to_bool

   bool_to_int: function[bool -> int] =
                (LAMBDA bb -> int: IF bb THEN 1 ELSE 0 END )


END signal
```

APPENDIX B

SUGGESTIONS FOR IMPROVING EHDM

In the following subsections, several suggestions are made for improving the EHDM verification system.

Definition of the Values of a Type

Frequently it is necessary to define a type which takes on a finite number of distinct values. This is accomplished in LCF-LSM as follows:

```
type signalval = NEW( t | f | x)
```

In EHDM this must be done via a laborious detailed specification of all of the properties needed:

```
signalval: TYPE
t,f,x: signalval
s: VAR signalval

unique:  AXIOM t ~= f and t ~= x and f ~= x
exhaust: AXIOM s = t or s = f or s = x
```

Definitional Axioms

There is a need for another kind of "axiom" in EHDM. In order to facilitate the proof process it is often necessary to rewrite LAMBDA definitions as axioms. Instead of

```
F: function[int -> int] = (LAMBDA x -> int: x*x)
```

one writes:

```
F: function[int -> int]
F_ax: AXIOM F(x) = x*x
```

Thus, the theorem prover only expands the definition of F when specifically stated as a premise. This is desirable when the definition is complicated and a proof does not depend upon the particulars of the definition. However, there is a unhappy side-effect of this procedure. There is now an additional axiom which appears at the base of the theory. (See section entitled "Status of Proofs".) In other words, when one performs a proof analysis, the big theorems you have proved are reported to depend upon a set of axioms. This set now includes all of these "axioms" which are merely definitions of temporary "names". The big result in no sense depends upon these "names". They were used as a convenience. If one would rewrite the module using LAMBDA definitions, then the same big theorems could be proved and the set of axioms it depended upon would not include these name definitions. Of course, the theorem prover may take weeks rather than minutes to prove the results. Perhaps EHDM could be extended with a new construct, say DEFINITION:

```
F-ax: DEFINITION f(x) = x*x
```

which must be of a particular restricted form.

In the User's Manual it states " EHDM currently requires a mapping for every uninterpreted type and every constant of the module being mapped." If one defines a function name to simplify the statement of a big theorem, one should not have to have to map this "temporary" function. For example, suppose the big theorem is:

```
big_theorem: AXIOM x*x + x = f(x*x+x)g(x)
```

Suppose that f and g are mapped into some concrete form in a mapping module and big-theorem is proved there. What if for convenience the above axiom was written as:

```
h: function[int -> int]
h_ax: AXIOM h(x) = x*x + x
big_theorem: AXIOM h(x) = f(h(x))g(x)
```

Would it be necessary to specifically "map" h down to the next level? Would "h_ax" have to be proved as a theorem in the mapping module? It is not clear

from the User's manual exactly what must be proved when doing hierarchical mappings.

Also the fact that axioms of one level are proved as theorems in the lower level leads to a terminology nightmare. It would be nice if a new keyword could be invented which conveyed this concept.

Improvement to Proof Instantiator

The Proof Instantiator "overlooks" some very obvious substitutions. Consider the function COUNTLOGIC (from module cnt6_blk) listed below in full:

```
COUNTLOGIC: function[statevector,word6,word2 -> statevector] =
              (LAMBDA stv, loadin, func -> statevector:
              make_triple( MULTIPLEX(INCLOGIC(count(stv),
                                              INCCON(node(stv))    ),
                                     loadin,
                                     MPLXCON(node(stv)) ),
                           BOOLF(bit2(0,func)),
                           NEXTNODE(node(stv),func,double(stv)) )
               )
```

There is an axiom NEXTNODE0_ax which will be cited as a premise:

```
NEXTNODE0_ax: AXIOM val2(nd) = 0 IMPLIES
                NEXTNODE(nd,func,dbl) =
                   IF val2(func) = 0 THEN fetchnode
                   ELSIF val2(func) = 1 THEN loadnode
                   ELSE inc1node
                   END
```

The following lemma is to be proved:

```
claa: LEMMA  val2(node(stv)) = 0  and val2(func) = 0 IMPLIES
        COUNTLOGIC(stv,loadin,func) =
            make_triple( MULTIPLEX(INCLOGIC(count(stv),
                                            INCCON(node(stv))    ),
                                   loadin,
                                   MPLXCON(node(stv)) ),
                         BOOLF(bit2(0,func)),
                         fetchnode)
```

The following proof statement does the job:

```
p_claa: PROVE claa FROM NEXTNODE0_ax{nd <- node(stv),
                                     func <- func,
                                     dbl <- double(stv) }
```

But the Instantiator will not find the instantiations! But, starting with the conclusion it is obvious that NEXTNODE is called:

```
NEXTNODE(node(stv),func,double(stv)) )
```

The only premise stated NEXTNODE0 is of the form:

```
NEXTNODE(nd,func,dbl) = ...
```

The required matchings are obvious, and would seem to be easily automated!

APPENDIX C

FULL LISTING OF SPECIFICATIONS INCLUDING PROOFS

```
cnt6: MODULE
USING words
THEORY

  (* ----- define abbreviations for 'words' types and functions ----- *)

  word6: TYPE is word[6]
  word2: TYPE is word[2]

  val2: function[word2 -> int] is val[2]
  val6: function[word6 -> int] is val[6]
  mw6: function[int -> word6]  is mw[6]

  (* ---------- define TYPE to represent state of machine ---------- *)

  states: TYPE

  (* -------------------- define logic variables -------------------- *)

  state: VAR states
  loadin,w: VAR word6
  func: VAR word2

  (* ---------- define properties of 6-bit counter ---------- *)

  cnt: function[states -> word6]
  exec_cnt: function[states,word6,word2 -> states]
  ready: function[states -> bool]

  add1_mod64: function[word6 -> word6] ==
              ( LAMBDA w -> word6:
                        IF val6(w) = 63 THEN mw6(0)
                        ELSE mw6(val6(w)+1)
                        END )

  ready_ax: AXIOM ready(state) IMPLIES ready( exec_cnt(state,loadin,func) )

  counter_ax: AXIOM ready(state) IMPLIES cnt(exec_cnt(state,loadin,func)) =
                    IF val2(func) = 0 THEN cnt(state)
                    ELSIF val2(func) = 1 THEN loadin
                    ELSIF val2(func) = 2 THEN
                       add1_mod64(cnt(state))
                    ELSE
                       add1_mod64(add1_mod64(cnt(state)))
                    END

END cnt6
```

```
words: MODULE[N: int]

USING power2_th,divby2_th

EXPORTING word, newword, bit, assign, val, mw, mwm, valm, bool_to_int
  WITH power2_th,divby2_th

ASSUMING
  N_pos: FORMULA N>0

THEORY
  word: TYPE

  k,i,ii,m,v,n,h,jj,y: VAR int
  w,w1,w2: VAR word
  a,b: VAR bool
  newword: word

  assign: function[int, bool, word -> word]
  bit: function[int, word -> bool]

  bitassign: AXIOM (i >= 0 and i < N) IMPLIES
                    bit(i,assign(k,b,w)) =
                         ( IF k = i THEN b ELSE bit(i,w) END )

  mw: function[int -> word]
  val: function[word -> int]

  mwm: function[int,int,int -> word]
  mwm_def: AXIOM mwm(v,m,n) =
                 IF m = 0 THEN newword
                 ELSE
                    assign(n-m,BMOD2(v),mwm(DIVBY2(v),m-1,n) )
                 END

  mw_def: AXIOM mw(ii) = mwm(ii,N,N)

  bool_to_int: function[bool -> int] =
               (LAMBDA b -> int: IF b THEN 1 ELSE 0 END )

  valm: function[word,int,int -> int]
  valm_def: AXIOM valm(w,m,n) = IF m = 0 THEN 0
                                ELSE 2*valm(w,m-1,n) + bool_to_int(bit(n-m,w))
                                END

  val_def: AXIOM val(w) = valm(w,N,N)

  (* ------------------------- Big Theorems ---------------------------- *)

  val_mw_thm: THEOREM ( ii >= 0 AND ii < power2(N) )
                      IMPLIES val(mw(ii)) = ii

  val_range_thm: THEOREM val(w) >= 0 and val(w) < power2(N)
```

```
val_bits_thm: THEOREM val(w1) = val(w2) IMPLIES
                  (FORALL m: m>=0 AND m<N IMPLIES bit(m,w1)=bit(m,w2))

(* --------- Definition Axioms That Should Be in Proof Section ---------- *)

zfun: function[int -> bool]
zfun_ax: AXIOM zfun(n) = ( (n > 0 AND n <= N AND ii >= 0 AND ii < power2(n) )
                            IMPLIES  valm(mwm(ii,n,n),n,n) = ii )

vfun: function[int -> bool]
vfun_ax: AXIOM vfun(m) = ( (m<n and m>=0 and n<=N ) IMPLIES
                             (valm(w,m,n) = valm(assign(0,b,w),m,n)) )

qfun: function[int -> bool]
qfun_ax: AXIOM qfun(m) =
              ( (m+1>0 and ii>=0 and m<n and ii < power2(m) and n<=N) IMPLIES
                   valm( mwm(ii,m,n) ,m,n)  = valm( mwm(ii,m,n-1) ,m,n-1)  )

twen: function[int->bool]
twen_ax: AXIOM twen(k) = (k>=0 AND k<N AND valm(w1,N,N) = valm(w2,N,N)
               IMPLIES valm(w1,N-k,N)=valm(w2,N-k,N))

weir: function[int -> bool]
weir_ax: AXIOM  weir(m) = (
                     (m<n and k>0 and m>=0 and n<=N) IMPLIES
                           (valm(w,m,n) = valm(assign(n-m-k,b,w),m,n)) )

valpos:function[int -> bool]
valpos_ax: AXIOM valpos(m) = (m>=0 IMPLIES valm(w,m,n) >=0)

build: function[int->bool]
build_ax: AXIOM build(k) = (k>=0 AND (FORALL m: m>=N-k AND m<N IMPLIES
            bit(m,w1)=bit(m,w2)) IMPLIES valm(w1,k,N) = valm(w2,k,N))

copy_m_bits: function[int,word,word -> word]
copy_m_bits_ax: AXIOM copy_m_bits(m,w1,w2) =
           ( IF m=0 THEN w2
             ELSE assign(N-m,bit(N-m,w1),copy_m_bits(m-1,w1,w2)) END)

gnu: function[int -> bool]
gnu_ax: AXIOM gnu(k) = (k>=0 AND k<m AND k+N-m>=0 IMPLIES
              bit(k+N-m,copy_m_bits(m,w1,w2)) = bit(k+N-m,w1))


rang: function[int -> bool]
rang_ax: AXIOM rang(m) = (m>=0 and m <= N IMPLIES valm(w,m,N)>=0 AND
                            valm(w,m,N) < power2(m)    )

PROOF

(* ----------------------- val_mw_thm THEOREM ---------------------------

val_mw_thm: AXIOM ( ii >= 0 AND ii < power2(N) ) IMPLIES val(mw(ii)) = ii *)
```

```
  inv_axiom: LEMMA ( n > 0 AND n <= N AND ii >= 0 AND ii < power2(n) )
                   IMPLIES valm(mwm(ii,n,n),n,n) = ii

(*zfun: function[int -> bool]
  zfun_ax: AXIOM zfun(n) = ( (n > 0 AND n <= N AND ii >= 0 AND ii < power2(n) )
                             IMPLIES  valm(mwm(ii,n,n),n,n) = ii )        *)


  L0: LEMMA zfun(0)

  L1: LEMMA zfun(1)

    L1a: LEMMA ( ii >= 0 AND ii < power2(1) ) IMPLIES MOD2(ii) = ii

      L1a_a: LEMMA MOD2(0) = 0
      L1a_b: LEMMA MOD2(1) = 1

  L2: LEMMA zfun(m) IMPLIES zfun(m+1)

    L2a: LEMMA  (m>=0 and ii>=0) IMPLIES valm(mwm(ii,m+1,m+1),m+1,m+1) =
                 2*valm(assign(0,BMOD2(ii),
                                mwm(DIVBY2(ii),m,m+1)),m,m+1)+
                 bool_to_int(bit(0,assign(0,BMOD2(ii),
                                    mwm(DIVBY2(ii),m,m+1))))

    L2b: LEMMA (m>=0 and m+1<=N and ii>=0) IMPLIES
        valm( assign( 0, BMOD2(ii), mwm(DIVBY2(ii),m,m+1) ) ,m,m+1) =
               valm( mwm(DIVBY2(ii),m,m+1) ,m,m+1)

(*    vfun: function[int -> bool]
      vfun_ax: AXIOM vfun(m) = ( (m<n and m>=0 and n<=N ) IMPLIES
                                 (valm(w,m,n) = valm(assign(0,b,w),m,n)) )    *)

      b20: LEMMA vfun(0)
      b21: LEMMA vfun(1)

      b2m: LEMMA(vfun(m) IMPLIES vfun(m+1))

      b2h: LEMMA (h>=0) IMPLIES vfun(h)

    L2c: LEMMA (m+1>0 and m+1<=N and ii>=0) IMPLIES
           valm(mwm(ii,m+1,m+1),m+1,m+1) =
                2*valm( mwm(DIVBY2(ii),m,m+1) ,m,m+1) + MOD2(ii)

    L2d: LEMMA (m+1>0 and ii>=0 and ii < power2(m+1) and m+1<=N) IMPLIES
                          valm( mwm(DIVBY2(ii),m,m+1) ,m,m+1)  =
                          valm( mwm(DIVBY2(ii),m,m) ,m,m)

(*    qfun: function[int -> bool]
      qfun_ax: AXIOM qfun(m) =
                ( (m+1>0 and ii>=0 and m<n and ii < power2(m) and n<=N) IMPLIES
                     valm( mwm(ii,m,n) ,m,n)  = valm( mwm(ii,m,n-1) ,m,n-1)  )
*)
      d20: LEMMA qfun(0)

        d20_a: LEMMA valm(mwm(ii,0,n),0,n) = valm(mwm(ii,0,n-1),0,n-1)
```

```
      d2m: LEMMA(qfun(m) IMPLIES qfun(m+1))

(*          weir: function[int -> bool]
            weir_ax: AXIOM  weir(m) = (
                       (m<n and k>0 and m>=0 and n<=N) IMPLIES
                             (valm(w,m,n) = valm(assign(n-m-k,b,w),m,n)) ) *)

            d2m_1: LEMMA weir(0)
            d2m_2: LEMMA weir(m) IMPLIES weir(m+1)
            d2m_3: LEMMA h>=0 IMPLIES weir(h)


            d2m_4: LEMMA (m<n AND m>=0 AND n<=N) IMPLIES
                   (valm(w,m,n) = valm(assign(n-m-1,b,w),m,n))

            d2m_5: LEMMA n - m - 1 > 0 IMPLIES n - m - 2 >= 0

      d2h: LEMMA (h>=0) IMPLIES qfun(h)


   L2e: LEMMA (m+1>0 and ii>=0 and m+1<=N and ii < power2(m+1))
              IMPLIES valm(mwm(ii,m+1,m+1),m+1,m+1) =
                2*valm( mwm(DIVBY2(ii),m,m) ,m,m) +
                MOD2(ii)

L2h: LEMMA (ii>=0 AND m+1 > 0 AND ii < power2(m+1)) IMPLIES
             (DIVBY2(ii) < power2(m))

L2i: LEMMA (m>0 AND zfun(m)) IMPLIES zfun(m+1)

L3: LEMMA (m>=0) IMPLIES zfun(m)

(* ------------------- PROVE Statements for Val_mw_thm ------------------- *)

p_val_mw_thm: PROVE val_mw_thm FROM inv_axiom{n <- N},
                                   val_def{w <- mwm(ii,N,N)},
                                   mw_def,
                                   N_pos

pinv: PROVE inv_axiom FROM L3{m <- n},
                           zfun_ax,
                           val_def{w <- mwm(ii,n,n)},
                           mw_def

p_L0: PROVE L0 FROM zfun_ax{n <- 0}

p_L1: PROVE L1 FROM zfun_ax{n <- 1},
                    valm_def{m <- 1, n <- 1, w <- mwm(ii@P1,1,1) },
                    valm_def{m <- 0, n <- 1, w <- mwm(ii@P1,1,1) },
                    mwm_def{v <- ii@P1, m <- 1, n <- 1},
                    bitassign{i <- 0,k <- 0,
                              b <- BMOD2(ii@P1),
                              w <- mwm(DIVBY2(ii@P1),0,1)},
                    MOD2_ax{i <- ii@P1},
                    N_pos, L1a{ii <- ii@P1}
```

```
p_L1a: PROVE L1a FROM L1a_a,L1a_b,
                      MOD2_ax{i <- 0},
                      power2_ax{i<-1},power2_ax{i<-0},
                      Y_1{y<-ii}

  p_L1aa: PROVE L1a_a FROM BMOD2_ax {i <- 0},
                           MOD2_ax{i <- 0},
                           DIV_ax{i <- 0}

  p_L1ab: PROVE L1a_b FROM BMOD2_ax {i <- 1},MOD2_ax{i<-1},
                           DIV_ax{i <- 1}

p_L2: PROVE L2 FROM zfun_ax{n<-m},zfun_ax{n <- m+1},L1,L2i,zfun_ax{n<-1}

p_L2a: PROVE L2a FROM valm_def{w <- mwm(ii,m+1,m+1), m <- m+1, n <- m+1},
                      mwm_def{v <- ii, m <- m+1, n <- m+1},
                      bitassign{i <- 0, k <- 0, b <- BMOD2(ii),
                                w <- mwm(DIVBY2(ii),0,1)},
                      N_pos

p_L2b: PROVE L2b FROM b2h{h <- m},
                      vfun_ax{m <- m, n <- m+1, b <- BMOD2(ii),
                              w <- mwm(DIVBY2(ii),m,m+1) }

   p_b20: PROVE b20 FROM
        vfun_ax {m <- 0},
        valm_def{m <- 0},
        valm_def{m <- 0,w <- assign(0,b@p1,w)}

  p_b21: PROVE b21 FROM
        vfun_ax {m <- 1},
        valm_def{m <- 1},
        valm_def{m <- 1,w <- assign(0,b@p1,w)},
        valm_def{m <- 0},
        valm_def{m <- 0,w <- assign(0,b@p1,w)},
        bitassign{i <- (n@p1-1),k <- 0}

  p_b2m: PROVE b2m FROM
        vfun_ax,
        vfun_ax{m <- m+1},
        valm_def{m <- m+1},
        valm_def{m <- m+1, w <- assign(0,b@p1,w)},
        bitassign{i <- n@p1-m-1, k<- 0}

  p_b2h: PROVE b2h FROM b20,
                        b2m{m<-d1@p3},
                        int_induction{p <- vfun,
                                      d2 <- h@C}

 p_L2c: PROVE L2c FROM L2a,L2b,
                    bitassign{i <- 0, k <- 0,b <- BMOD2(ii),
                              w <- mwm(DIVBY2(ii),m,m+1)},
                    MOD2_ax{i <- ii},
                    N_pos
```

```
p_L2d: PROVE L2d FROM d2h{h <- m},
                      qfun_ax{ii <- DIVBY2(ii@C),n<-m+1},
                      L2h{ii<-ii@C},DIVBY2g0{ii<-ii@C}

   p_d20_a: PROVE d20_a FROM  valm_def{w<- mwm(ii,0,n),m<-0},
                              valm_def{w<- mwm(ii,0,n-1),m<-0,n<-n-1}


   p_d20: PROVE d20 FROM power2_ax{i<-0},
         qfun_ax {m <- 0},
         d20_a

   p_d2m_5: PROVE d2m_5

   p_d2m: PROVE d2m FROM
         qfun_ax,
         qfun_ax{m <- m+1},
         valm_def{w<-mwm(ii@P1,m+1,n@P1),m <- m+1},
         valm_def{w<-mwm(ii@P1,m+1,n@P1-1),m <- m+1,n<-n@P1-1},
         mwm_def{v<-ii@P1,m<-m+1},
         mwm_def{v<-ii@P1,m<-m+1,n<-n@P1-1},
         L2h{ii<-ii@P1},DIVBY2g0{ii<-ii@P1},
         d2m_4{w<-mwm(DIVBY2(ii@P1),m,n),b<-BMOD2(ii@P1)},
         qfun_ax{ii<-DIVBY2(ii@P1)},
         d2m_4{w<-mwm(DIVBY2(ii@P1),m,n),b<-BMOD2(ii@P1),
             n<-n@P1-1},
         bitassign{i<-n@P1-m-1,k<-n@P1-m-1,b<- BMOD2(ii@P1),
                   w<- mwm(DIVBY2(ii@P1),m,n@P1)},
         bitassign{i<-n@P1-m-2,k<-n@P1-m-2,b<- BMOD2(ii@P1),
                   w<- mwm(DIVBY2(ii@P1),m,n@P1-1)     },
         d2m_5

    p_d2m_1: PROVE d2m_1 FROM weir_ax{m<-0},valm_def{m<-0},
                   valm_def{w<-assign(n-m-k@P1,b@P1,w),m<-0}


    p_d2m_2: PROVE d2m_2 FROM weir_ax,
                              valm_def{w<-assign(n-m-1-k@P1,b@P1,w),
                                        m<-m+1},
                              weir_ax{m<-m+1,k<-k@P1,b<-b@P1},
                              valm_def{m<-m+1},
                              weir_ax{k<-k@P1+1,b<-b@P1},
                              bitassign{i<-n@P1-m-1,
                                        k<-n@P1-m-1-k@P1,b<-b@P1}

   p_d2m_3:PROVE d2m_3 FROM d2m_1,
                            d2m_2{m<-d1@p3},
                            int_induction{p <- weir,
                                          d2 <- h@C}

   p_d2m_4: PROVE d2m_4 FROM d2m_3{h<-m},weir_ax{m<-m@C,k<-1}
```

```
    p_d2h: PROVE d2h FROM d20,
                          d2m{m<-d1@p3},
                          int_induction{p <- qfun,
                                        d2 <- h@C}

    p_L2e: PROVE L2e FROM L2c,L2d

    p_L2h: PROVE L2h FROM power2_ax{i <- m+1},
                          DIVBY2x2

    p_L2i: PROVE L2i FROM zfun_ax {n <- m + 1, ii <- power2(m@CS + 1)},
                          L2e {ii <- ii@P1},
                          zfun_ax {n <- m, ii <- DIVBY2(ii@P1)},
                          DIV_MOD_thm,
                          DIVBY2g0 {ii <- ii@P1},
                          L2h {ii <- ii@P1}

  p_L3: PROVE L3 FROM  L0,
                       L2{m<-d1@p3},
                       int_induction{p <- zfun,
                                     d2 <- m@C},
                       zfun_ax{n <- m+1},
                       zfun_ax{n <- m}


  (* ------------------------ val_bits_thm THEOREM --------------------------

  val_bits_thm: THEOREM val(w1) = val(w2) IMPLIES
                (FORALL m: m>=0 AND m<N IMPLIES bit(m,w1)=bit(m,w2))        *)

    Subwords: THEOREM val(w1) = val(w2) IMPLIES
              (FORALL m: 0<m AND m<=N IMPLIES valm(w1,m,N) = valm(w2,m,N))

(*     twen: function[int->bool]
     twen_ax: AXIOM twen(k) = (k>=0 AND k<N AND valm(w1,N,N) = valm(w2,N,N)
                      IMPLIES valm(w1,N-k,N)=valm(w2,N-k,N))                *)

       tw0: LEMMA twen(0)
       twk: LEMMA twen(k) IMPLIES twen(k+1)
       twh: LEMMA h>=0 IMPLIES twen(h)

    T1: LEMMA m>0 AND valm(w1,m,N) = valm(w2,m,N) IMPLIES
                         valm(w1,m-1,N) = valm(w2,m-1,N)

    T1corol: LEMMA m>0 AND valm(w1,m,N) = valm(w2,m,N) IMPLIES
                bool_to_int(bit(N-m,w1))=bool_to_int(bit(N-m,w2))

     V1: LEMMA m>0 IMPLIES DIVBY2(valm(w,m,N))= valm(w,m-1,N)

(*       valpos:function[int -> bool]
         valpos_ax: AXIOM valpos(m) = (m>=0 IMPLIES valm(w,m,n) >=0)    *)

         val0: LEMMA valpos(0)
         valm_step: LEMMA valpos(m) IMPLIES valpos(m+1)
         valh: LEMMA h>=0 IMPLIES valpos(h)
```

```
        valpos_result: LEMMA m>=0 IMPLIES valm(w,m,n)>=0

    (* ------------------- Proof of val_bits_thm Theorem ---------------------- *)

    p_val_bits_thm: PROVE val_bits_thm FROM Subwords{m<-N-m@C},
                                            T1corol{m<-N-m@C}

    p_Subwords: PROVE Subwords FROM twh{h<-N-m@C},twen_ax{k<-N-m@C},
                                    val_def{w<-w1},val_def{w<-w2}

      p_val0: PROVE val0 FROM valm_def{m<-0},valpos_ax{m<-0}

      p_valm: PROVE valm_step FROM valm_def{m<-m@C+1},
                                  val0,valpos_ax{m<-m@C},valpos_ax{m<-m@C+1}

      p_valh: PROVE valh FROM val0,valm_step{m<-d1@P3},
                              int_induction{p<-valpos,d2<-h@C}

      p_valpos:  PROVE valpos_result FROM valh{h<-m@C},valpos_ax{m<-m@C}

      p_V1: PROVE V1 FROM valm_def{w<-w@C,m<-m@C,n<-N},
                          valpos_result{m <- m-1, n<-N},
                          valm_def{w<-w@C,m<-0,n<-N},
                          DIV_doub{ii <- valm(w,m-1,N)}

      p_T1: PROVE T1 FROM V1{w<-w1@C},V1{w<-w2@C}

      p_T1corol: PROVE T1corol FROM T1,valm_def{w<-w1,n<-N},
                                    valm_def{w<-w2,n<-N}

      p_tw0: PROVE tw0 FROM twen_ax{k<-0}

      p_twk: PROVE twk FROM tw0,twen_ax,twen_ax{k<-k+1},T1{m<-N-k}

      p_twh: PROVE twh FROM tw0,twk{k<-d1@P3},int_induction{p<-twen,d2<-h@C}


    (* ---------------------- Bits_enuf THEOREM ---------------------------- *)

    Bits_enuf: THEOREM (FORALL m: m>=0 AND m<N IMPLIES bit(m,w1)=bit(m,w2))
                    IMPLIES val(w1)=val(w2)

      Build: LEMMA k>=0 AND (FORALL m: m>=N-k AND m<N IMPLIES
                  bit(m,w1)=bit(m,w2)) IMPLIES valm(w1,k,N) = valm(w2,k,N)

(*      build: function[int->bool]
        build_ax: AXIOM build(k) = (k>=0 AND (FORALL m: m>=N-k AND m<N IMPLIES
                bit(m,w1)=bit(m,w2)) IMPLIES valm(w1,k,N) = valm(w2,k,N))    *)

        bld0: LEMMA build(0)

        bldk: LEMMA build(k) IMPLIES build(k+1)
```

```
    bldh: LEMMA h>=0 IMPLIES build(h)


 (* ------------  Proof of Bits_enuf Theorem ---------------------- *)

 p_Bits: PROVE Bits_enuf{m<-m@P1} FROM Build{k<-N},val_def{w<-w1},
                                       val_def{w<-w2},N_pos

 p_Build: PROVE Build{m<-m@P2} FROM bldh{h<-k@C},build_ax{k<-k@C}

   p_bld0: PROVE bld0 FROM build_ax{k<-0},valm_def{w<-w1@P1,m<-0,n<-N},
                           valm_def{w<-w2@P1,m<-0,n<-N}

   p_bldk: PROVE bldk FROM bld0,build_ax,build_ax{k<-k@P2+1,m<-m@P2},
                           valm_def{w<-w1@P2,m<-k@P2+1,n<-N},
                           valm_def{w<-w2@P2,m<-k@P2+1,n<-N},
                           build_ax{k<-k@P2+1,m<-N-k@P2-1}

   p_bldh: PROVE bldh FROM bld0,bldk{k<-d1@P3},
                           int_induction{p<-build,d2<-h@C}


 (* ----------------------- Copy_word_thm THEOREM ------------------------ *)

(* copy_m_bits: function[int,word,word -> word]
   copy_m_bits_ax: AXIOM copy_m_bits(m,w1,w2) =
            ( IF m=0 THEN w2
              ELSE assign(N-m,bit(N-m,w1),copy_m_bits(m-1,w1,w2)) END)  *)


   copy: function[word -> word] =
          (LAMBDA w1 -> word: copy_m_bits(N,w1,newword) )

   Copy_word_thm: THEOREM k>=0 AND k<N IMPLIES
                          bit(k,copy(w1)) = bit(k,w1)

(*   gnu: function[int -> bool]
     gnu_ax: AXIOM gnu(k) = (k>=0 AND k<m AND k+N-m>=0 IMPLIES
                bit(k+N-m,copy_m_bits(m,w1,w2)) = bit(k+N-m,w1))      *)

     gnu0: LEMMA gnu(0)

     gnuk: LEMMA gnu(k) IMPLIES gnu(k+1)

     gnuh: LEMMA h>=0 IMPLIES gnu(h)

     gnu_lemma: LEMMA k>=0 AND k<m AND k+N-m>=0 IMPLIES
                      bit(k+N-m,copy_m_bits(m,w1,w2)) = bit(k+N-m,w1)


   (* ----------------- Proof of Copy_word_thm ---------------------------- *)

   p_gnu_lemma: PROVE gnu_lemma FROM gnuh{h<-k@C},gnu_ax
```

```
p_Copy_word_thm: PROVE Copy_word_thm FROM gnu_lemma{m<-N, w2 <- newword}

  p_gnu0: PROVE gnu0 FROM gnu_ax{k<-0},copy_m_bits_ax,
                          bitassign{i<-N-m@P1,
                                    k<-N-m@P1,b<-bit(N-m@P1,w1@P1),
                                    w<-copy_m_bits(m@P1-1,w1@P1,w2@P1)}

  p_gnuk: PROVE gnuk FROM gnu0,gnu_ax,gnu_ax{k<-k+1},gnu_ax{m<-m@P2-1},
                          copy_m_bits_ax,
                          bitassign{i<-k+1+N-m@P2,k<-N-m@P2,
                                    b<-bit(N-m@P2,w1@P2),
                                    w<-copy_m_bits(m@P2-1,w1@P2,w2@P2)}

  p_gnuh: PROVE gnuh FROM gnu0,gnuk{k<-d1@P3},
                          int_induction{p<-gnu,d2<-h@C}


(* ---------------------- val_range_thm THEOREM ------------------------

  val_range_thm: THEOREM val(w) >= 0 and val(w) < power2(N)                *)

(*  rang: function[int -> bool]
    rang_ax: AXIOM rang(m) = (m>=0 and m <= N IMPLIES valm(w,m,N)>=0 AND
                              valm(w,m,N) < power2(m)    )                *)

    rang0: LEMMA rang(0)

    lrana: LEMMA m>=0 AND  valm(w,m,N)  < power2(m) IMPLIES
                   2*valm(w,m,N) + bool_to_int(bit(N-m-1,w)) < 2*power2(m)

    lranb: LEMMA m>=0 and valm(w,m,N) >=0 IMPLIES
                 2*valm(w,m,N) + bool_to_int(bit(N-m-1,w))>=0

    rangm: LEMMA rang(m) IMPLIES rang(m+1)

    rangh: LEMMA h>=0 IMPLIES rang(h)

    valm_range: THEOREM m>=0 and m<=N IMPLIES valm(w,m,N)>=0 AND
                              valm(w,m,N) < power2(m)

(* ----------------- Proof of Val_range_thm ----------------------------- *)

  p_val_range: PROVE val_range_thm FROM valm_range{m<-N},
                                       val_def,N_pos


    p_rang0: PROVE rang0 FROM rang_ax{m<-0},valm_def{w<-w@P1,m<-0,n<-N},
                              N_pos, power2_ax{i<-0}


    p_lrana: PROVE lrana FROM N_pos
    p_lranb: PROVE lranb
```

```
    p_rangm: PROVE rangm FROM rang0,rang_ax,rang_ax{m<-m+1},
                              valm_def{w<-w@P2,m<-m@C+1,n<-N},
                              power2_ax{i<-m@C+1},lrana,lranb

    p_rangh: PROVE rangh FROM rang0,rangm{m<-d1@P3},
                              int_induction{p<-rang,d2<-h@C}

    p_valm_range: PROVE valm_range FROM rangh{h<-m@C},rang_ax

END words


power2_th: MODULE

USING int_inductions

EXPORTING power2

THEORY
   x: VAR bool
   y,m,i,ii,h: VAR int

   power2: function[int -> int]
   power2_ax: AXIOM power2(i) = IF i=0  THEN 1 ELSE 2*power2(i-1) END


   pow_eg: THEOREM (y>=0) IMPLIES (power2(y) >= 0)

   pow_gr: THEOREM (y>=0) IMPLIES (power2(y+1) >= power2(y))

     xpow: function[int -> bool]

     xpow_ax: AXIOM xpow(m) = (power2(m) >= 0)

     xp0: LEMMA xpow(0)

     xpm: LEMMA (xpow(m) IMPLIES xpow(m+1))

     xph: LEMMA (FORALL h: (h>=0 IMPLIES xpow(h)))

    G_0: THEOREM y>=0 IMPLIES 2*y >= 0

    G_1: THEOREM y>=0 IMPLIES 2*y+1 >= 0

PROOF
  p_pow_eg: PROVE pow_eg FROM xpow_ax{m<-y},
                              xph{h<-y}

  p_pow_gr: PROVE pow_gr FROM
                        G_0, pow_eg, power2_ax{i <- y+1}
    p_xp0: PROVE xp0 FROM power2_ax{i<- 0},xpow_ax{m<-0}
    p_xpm: PROVE xpm FROM
                xpow_ax,
                xpow_ax{m<-m+1},
                power2_ax{i<-m+1},
                G_0{y <- power2(m)}
```

```
    p_xph: PROVE xph FROM xp0,
                          xpm{m<-d1@p3},
                          int_induction{p <- xpow,
                                        d2<- h@C}

  p_G0: PROVE G_0

  p_G1: PROVE G_1

END power2_th

divby2_th: MODULE

USING int_inductions,power2_th,ineq_cases

EXPORTING DIVBY2,MOD2,BMOD2

THEORY
  b: VAR bool
  y,m,i,ii,h: VAR int

  DIVBY2: function[int -> int]
  DIV_ax: AXIOM DIVBY2(i) = IF i >= 2 THEN 1 + DIVBY2(i-2)
                            ELSE IF i <= -2 THEN -DIVBY2(-i)
                            ELSE 0 END END

  BMOD2: function [int -> bool]
  BMOD2_ax: AXIOM BMOD2(i) = (2*DIVBY2(i) ~= i)

  MOD2: function [int -> int]
  MOD2_ax: AXIOM MOD2(i) = IF BMOD2(i) THEN 1 ELSE 0 END

  B0: LEMMA BMOD2(0) = false
  B1: LEMMA BMOD2(1) = true

  Balt: THEOREM h>=0 IMPLIES BMOD2(h) = NOT BMOD2(h+1)

    alt: function[int -> bool]

    alt_ax: AXIOM alt(ii) = (ii >= 0 IMPLIES BMOD2(ii) = NOT BMOD2(ii+1))
    alt0: LEMMA alt(0)
    alt1: LEMMA alt(1)
    altm: LEMMA alt(m) IMPLIES alt(m+2)
    alth: LEMMA h>=0 IMPLIES alt(h)


  Even: LEMMA ii>=0 IMPLIES NOT BMOD2(2*ii)

  Even_MOD: LEMMA ii>=0 IMPLIES MOD2(2*ii) = 0

    kill: function[int -> bool]

    kill_ax: AXIOM  kill(ii) =( ii>=0 IMPLIES NOT BMOD2(2*ii))
    kill0: LEMMA kill(0)
```

```
    killm: LEMMA kill(m) IMPLIES kill(m+1)
    killh: LEMMA h>=0 IMPLIES kill(h)

 Odd_MOD:  LEMMA ii>=0 IMPLIES MOD2(2*ii+1) = 1

  DIVBY2x2: LEMMA (ii >= 0) IMPLIES 2*DIVBY2(ii) <= ii

       ifun: function[int -> bool]
       ifun_ax: AXIOM ifun(ii) =  (ii>=0 IMPLIES 2*DIVBY2(ii) <= ii)
       if0: LEMMA ifun(0)
       if1: LEMMA ifun(1)
       ifm: LEMMA ifun(m) IMPLIES ifun(m+2)
       ifh: LEMMA (h>= 0 ) IMPLIES ifun(h)


  DIVBY2x2p1: LEMMA (ii >= 0) IMPLIES 2*DIVBY2(ii)+1 >= ii

       tfun: function[int -> bool]
       tfun_ax: AXIOM tfun(ii) =  (ii>=0 IMPLIES 2*DIVBY2(ii)+1 >= ii)
       tf0: LEMMA tfun(0)
       tf1: LEMMA tfun(1)
       tfm: LEMMA tfun(m) IMPLIES tfun(m+2)
       tfh: LEMMA (h>= 0 ) IMPLIES tfun(h)


  DIV_MOD_thm: LEMMA ii>=0 IMPLIES 2*DIVBY2(ii) + MOD2(ii) = ii

      Pre2f: LEMMA ii>=0 IMPLIES (2*DIVBY2(ii) = ii OR 2*DIVBY2(ii) + 1 = ii)


  DIVBY2g0: LEMMA (ii >= 0) IMPLIES (DIVBY2(ii) >= 0)

  DIV_doub: LEMMA (ii >= 0 IMPLIES DIVBY2(2*ii) = ii) AND
                  (ii >= 0 IMPLIES DIVBY2(2*ii+1) = ii)


  MOD0_0: LEMMA MOD2(0) = 0
  MOD1_1: LEMMA MOD2(1) = 1

PROOF

  p_MOD0_0: PROVE MOD0_0 FROM BMOD2_ax {i <- 0},
                              MOD2_ax{i <- 0},
                               DIV_ax{i <- 0}

  p_MOD1_1: PROVE MOD1_1 FROM BMOD2_ax {i <- 1},MOD2_ax{i<-1},
                               DIV_ax{i <- 1}


  p_B0: PROVE B0 FROM BMOD2_ax{i <- 0}, DIV_ax{i<- 0}

  p_B1: PROVE B1 FROM BMOD2_ax{i <- 1}, DIV_ax{i<- 1}

  p_Balt: PROVE Balt FROM alth{h<-h@C},alt_ax{ii<-h@C}
```

```
    palt0: PROVE alt0 FROM alt_ax{ii<-0},B0,B1

    palt1: PROVE alt1 FROM alt_ax{ii<-1},B0,B1,
                           BMOD2_ax{i<-2},BMOD2_ax{i<-1},DIV_ax{i<-1},
                           DIV_ax{i<-2},DIV_ax{i<-0}

    paltm: PROVE altm FROM alt_ax{ii<-m@C},alt_ax{ii<-m@C+2},BMOD2_ax{i<-m},
                           alt0,alt1,BMOD2_ax{i<-m+1},BMOD2_ax{i<-m+2},
                           BMOD2_ax{i<-m+3},DIV_ax{i<-m+2},DIV_ax{i<-m+3}

    palth: PROVE alth FROM alt0,alt1,altm{m<-d1@P4},
                            int_induct_by_2{p<-alt,d2<-h@C}


  p_Even: PROVE Even FROM killh{h<-ii@C},kill_ax{ii<-ii@C}

  p_Even_MOD: PROVE Even_MOD FROM Even{ii<-ii@C},MOD2_ax{i<-2*(ii@C)}

  p_Odd_MOD: PROVE Odd_MOD FROM MOD2_ax{i<-2*ii@C+1},Balt{h<-2*ii@C},
                                Even{ii<-ii@C}
  pDIVBY2x2: PROVE DIVBY2x2 FROM ifh{h<-ii},ifun_ax
     pif0:  PROVE if0 FROM  ifun_ax{ii<-0}, DIV_ax{i<-0}
     pif1:  PROVE if1 FROM  ifun_ax{ii<-1}, DIV_ax{i<-1}
     pifm:  PROVE ifm FROM  ifun_ax{ii<-m}, ifun_ax{ii<- (m+2)},
              DIV_ax{i<- (m+2)}
     pifh:  PROVE ifh FROM if0,if1,ifm{m<-d1@P4},
                           int_induct_by_2{p<-ifun,d2<-h@C}


  pDIVBY2x2p1: PROVE DIVBY2x2p1 FROM tfh{h<-ii},tfun_ax
     ptf0:  PROVE tf0 FROM  tfun_ax{ii<-0}, DIV_ax{i<-0}
     ptf1:  PROVE tf1 FROM  tfun_ax{ii<-1}, DIV_ax{i<-1}
     ptfm:  PROVE tfm FROM  tfun_ax{ii<-m}, tfun_ax{ii<- (m+2)},
                            DIV_ax{i<- (m+2)}
     ptfh:  PROVE tfh FROM tf0,tf1,tfm{m<-d1@P4},
                           int_induct_by_2{p<-tfun,d2<-h@C}


  p_DIVBY2g0: PROVE DIVBY2g0 FROM DIVBY2x2p1

  p_doub: PROVE DIV_doub FROM Even_MOD{ii<-ii@C}, Odd_MOD{ii<-ii@C},
                              G_0{y<-ii@C}, G_1{y<-ii@C},
                              DIV_MOD_thm{ii<-2*ii@C},DIV_MOD_thm{ii<-2*ii@C+1}

  p_DIV_MOD_thm: PROVE DIV_MOD_thm FROM Pre2f,MOD2_ax{i<-ii},
                          BMOD2_ax{i<-ii}

  p_pre2f: PROVE Pre2f FROM DIVBY2x2,DIVBY2x2p1,
                           Y_1{y<-2*DIVBY2(ii) - ii +1}

END divby2_th
```

```
cnt6_fa: MODULE

(* ------------------------------------------------------------------------

   This module provides a more detailed view of the 6-bit counter function
  *counter* defined in the module cnt6.  These module defines the counter
   as a finite state automata with the following states:

      fetchnode  inc1node  inc2node  loadnode

   The state transitions are performed by the function NEXT.

 ------------------------------------------------------------------------*)
MAPPING cnt6 ONTO words, triples[word[6],bool,word[2]],bsignal

THEORY
  (* ---------------------- create some abbreviations ---------------------- *)

  word2: TYPE is word[2]
  word6: TYPE is word[6]

  mw2: function[int -> word2] is mw[2]
  mw6: function[int -> word6] is mw[6]
  val2: function[word2 -> int] is val[2]
  val6: function[word6 -> int] is val[6]
  bit2: function[int, word2 -> signalval] is bit[2]

  statevector: TYPE is triple

  count: function[statevector -> word6] is first
  double: function[statevector -> bool] is second
  node: function[statevector -> word2] is third

  BOOLF: function[signalval -> bool] is signal_to_bool

  (* ---------------------- define logic constants ---------------------- *)

  fetchnode: word2 = mw2(0)
  inc1node: word2 = mw2(1)
  inc2node: word2 = mw2(2)
  loadnode: word2 = mw2(3)
  undef_svt: statevector

  (* ---------------------- define logic variables ---------------------- *)

  svt: VAR statevector
  ct, ldn, w: VAR word6
  fn: VAR word2
  dbl,b: VAR bool

  (* ---------------------- define functions ---------------------- *)
```

```
ADD1: function[word6 -> word6] ==
        (LAMBDA w -> word6:
           IF val6(w) = 63 THEN mw6(0) ELSE mw6(val6(w)+1)
           END )

INC1: function[word6,bool,word6,word2 -> statevector]
INC1_ax: AXIOM INC1(ct, dbl, ldn, fn) =
             IF dbl THEN
                make_triple(ADD1(ct),BOOLF(bit2(0,fn)),inc2node)
             ELSE
                make_triple(ADD1(ct),BOOLF(bit2(0,fn)),fetchnode)
             END


INC2: function[word6,bool,word6,word2 -> statevector]
INC2_ax: AXIOM INC2(ct, dbl, ldn, fn) =
                make_triple( ADD1(ct),BOOLF(bit2(0,fn)),fetchnode)

LOAD: function[word6,bool,word6,word2 -> statevector]
LOAD_ax: AXIOM LOAD(ct, dbl, ldn, fn) =
                make_triple(ldn,BOOLF(bit2(0,fn)),fetchnode)

FETCH: function[word6,bool,word6,word2 -> statevector]
FETCH_ax: AXIOM FETCH(ct, dbl, ldn, fn)=
            IF val2(fn) = 0 THEN
               make_triple(ct,BOOLF(bit2(0,fn)),fetchnode)
            ELSIF val2(fn) = 1 THEN
               make_triple(ct,BOOLF(bit2(0,fn)),loadnode)
            ELSE
               make_triple(ct,BOOLF(bit2(0,fn)),inc1node)
            END

NEXT: function[statevector,word6,word2 -> statevector]
(* ------------------------------------------------------------
NEXT_ax: AXIOM NEXT(svt,ldn,fn) =
                 IF val2(node(svt)) = 0 THEN
                    FETCH(count(svt),double(svt),ldn,fn)
                 ELSIF val2(node(svt)) = 1 THEN
                    INC1(count(svt),double(svt),ldn,fn)
                 ELSIF val2(node(svt)) = 2 THEN
                    INC2(count(svt),double(svt),ldn,fn)
                 ELSIF val2(node(svt)) = 3 THEN
                    LOAD(count(svt),double(svt),ldn,fn)
                 ELSE
                    undef_svt
                 END
------------------------------------------------------------ *)

NEXT0_ax: AXIOM val2(node(svt)) = 0 IMPLIES
            NEXT(svt,ldn,fn) = FETCH(count(svt),double(svt),ldn,fn)

NEXT1_ax: AXIOM val2(node(svt)) = 1 IMPLIES
            NEXT(svt,ldn,fn) = INC1(count(svt),double(svt),ldn,fn)
```

```
NEXT2_ax: AXIOM val2(node(svt)) = 2 IMPLIES
              NEXT(svt,ldn,fn) = INC2(count(svt),double(svt),ldn,fn)

NEXT3_ax: AXIOM val2(node(svt)) = 3 IMPLIES
              NEXT(svt,ldn,fn) = LOAD(count(svt),double(svt),ldn,fn)


Finite_automata: function[statevector,word6,word2 -> statevector] =
                 (LAMBDA svt, ldn, fn -> statevector:
                      IF val2(fn) = 0 THEN
                         NEXT(svt,ldn,fn)
                      ELSIF val2(fn) = 3 THEN
                         NEXT(NEXT( NEXT(svt,ldn,fn), ldn,fn ),
                                    ldn,fn )
                      ELSE
                         NEXT( NEXT(svt,ldn,fn), ldn,fn )
                      END )

(* --------- Mapping to Top Level Spec in Module cnt6 ------------ *)

cnt6.states: TYPE FROM statevector

cnt6.cnt: function[statevector -> word6] is count

cnt6.exec_cnt: function[statevector,word6,word2 -> statevector]
                is Finite_automata

cnt6.ready: function[statevector -> bool] =
               (LAMBDA svt -> bool: node(svt) = fetchnode )


(* -------------------- LEMMAs -------------------------*)
st,state: VAR states
loadin,ld: VAR word6
func: VAR word2
y,m: VAR int

(* --------- LEMMAs needed to prove counter_ax ---------- *)


g1: LEMMA power2(2) = 4

g2: LEMMA val2(fn) = 0 or val2(fn) = 1 or
          val2(fn) = 2 or val2(fn) = 3

 g2a: THEOREM (y >= 0 AND y < m) IMPLIES ((y >= 0 AND y < m-1) OR (y=m-1))

g3: LEMMA bit2(0,fn) = BMOD2(val2(fn))


stb1: LEMMA ready(st) IMPLIES val2(node(st)) = 0

cnt_0: LEMMA ready(state) and val2(func) = 0
             IMPLIES cnt(exec_cnt(state,loadin,func)) = cnt(state)
                AND ready(exec_cnt(state,loadin,func))
```

```
cnt_1: LEMMA ready(state) and val2(func) = 1
             IMPLIES cnt(exec_cnt(state,loadin,func)) = loadin
                AND ready(exec_cnt(state,loadin,func))

cnt_2: LEMMA ready(state) and val2(func) = 2
             IMPLIES cnt(exec_cnt(state,loadin,func)) =
                     add1_mod64(cnt(state))
                AND ready(exec_cnt(state,loadin,func))

cnt_3: LEMMA ready(state) and val2(func) = 3
             IMPLIES cnt(exec_cnt(state,loadin,func)) =
                     add1_mod64(add1_mod64(cnt(state)))
                AND ready(exec_cnt(state,loadin,func))

(* --------- LEMMAs needed to prove cnt_1 ---------- *)

c1a: LEMMA ready(st) and val2(fn) = 1
          IMPLIES exec_cnt(st,ld,fn) =
              NEXT( FETCH(cnt(st),double(st),ld,fn), ld,fn )

c1b: LEMMA val2(fn) = 1 IMPLIES
             NEXT( FETCH(cnt(st),double(st),ld,fn), ld,fn ) =
               LOAD( cnt(st), BOOLF(bit2(0,fn)), ld,fn)

(* --------- LEMMAs needed to prove cnt_2 ---------- *)


c2a: LEMMA ready(st) and val2(fn) = 2 IMPLIES
            exec_cnt(st,ld,fn) =
             NEXT( FETCH(cnt(st),double(st),ld,fn), ld,fn)

c2b: LEMMA ready(st) and val2(fn) = 2 IMPLIES
          NEXT( FETCH(cnt(st),double(st),ld,fn),  ld,fn) =
          NEXT( make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node),  ld,fn)

c2c: LEMMA ready(st) and val2(fn) = 2 IMPLIES
          NEXT( make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node),  ld,fn) =
          INC1(cnt(make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node) ),
               double(make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node)),
               ld,fn)

c2d: LEMMA ready(st) and val2(fn) = 2 IMPLIES
          INC1(cnt(make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node) ),
               double(make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node)),
               ld,fn)  =
          INC1(cnt(st),BOOLF(bit2(0,fn)),ld,fn)

c2e: LEMMA val2(fn) = 2 IMPLIES NOT BOOLF(bit2(0,fn))

c2f: LEMMA  ready(st) and val2(fn) = 2 IMPLIES
             INC1(cnt(st),BOOLF(bit2(0,fn)),ld,fn)  =
               make_triple(ADD1(cnt(st)),BOOLF(bit2(0,fn)),fetchnode)

(* --------- LEMMAs needed to prove cnt_3 ----------  *)
```

```
c3a: LEMMA ready(st) and val2(fn) = 3 IMPLIES
              exec_cnt(st,ld,fn) =
                 NEXT(NEXT(NEXT(st,ld,fn),ld,fn),ld,fn)

c3b: LEMMA ready(st) and val2(fn) = 3 IMPLIES
           NEXT(NEXT(NEXT(st,ld,fn),ld,fn),ld,fn) =
              NEXT(NEXT(FETCH(cnt(st),double(st),ld,fn),ld,fn),ld,fn)

c3c: LEMMA ready(st) and val2(fn)=3 IMPLIES
           NEXT(NEXT(FETCH(cnt(st),double(st),ld,fn),ld,fn),ld,fn) =
              NEXT(NEXT(make_triple(cnt(st),
                                    BOOLF(bit2(0,fn)),inc1node),
                        ld,fn),ld,fn)

c3d: LEMMA ready(st) and val2(fn) = 3 IMPLIES
           NEXT(NEXT(make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node),
                     ld,fn),ld,fn)=
             NEXT(INC1(cnt(make_triple(cnt(st),
                                       BOOLF(bit2(0,fn)),inc1node)),
                          double(make_triple(cnt(st),
                          BOOLF(bit2(0,fn)),
                          inc1node)),
                  ld,fn),ld,fn)

c3e: LEMMA ready(st) and val2(fn) = 3 IMPLIES
       NEXT(INC1(cnt(make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node)),
          double(make_triple(cnt(st),BOOLF(bit2(0,fn)),inc1node)),
                            ld,fn),ld,fn) =
          NEXT(make_triple(ADD1(cnt(st)),BOOLF(bit2(0,fn)),inc2node),
                   ld,fn)

c3f: LEMMA ready(st) and val2(fn) = 3 IMPLIES
      NEXT(make_triple(ADD1(cnt(st)),
                       BOOLF(bit2(0,fn)),inc2node),ld,fn) =
        INC2(cnt(make_triple(ADD1(cnt(st)),
                             BOOLF(bit2(0,fn)),
                             inc2node)),
                double(make_triple(ADD1(cnt(st)),BOOLF(bit2(0,fn)),
                                   inc2node)),ld,fn)

c3g: LEMMA ready(st) and val2(fn) = 3 IMPLIES
         INC2(cnt(make_triple(ADD1(cnt(st)),BOOLF(bit2(0,fn)),inc2node)),
                  double(make_triple(ADD1(cnt(st)),BOOLF(bit2(0,fn)),
                                     inc2node)),ld,fn) =
         INC2(ADD1(cnt(st)),BOOLF(bit2(0,fn)),ld,fn)

c3h: LEMMA ready(st) and val2(fn) = 3 IMPLIES
           INC2(ADD1(cnt(st)),BOOLF(bit2(0,fn)),ld,fn)=
            make_triple(ADD1(ADD1(cnt(st))),BOOLF(bit2(0,fn)),fetchnode)

c3n: LEMMA val2(inc1node) = 1

c3p: LEMMA val2(fn) = 3 IMPLIES BOOLF(bit2(0,fn))
```

```
PROOF

   p_assuming1: PROVE words[2].N_pos

   p_counter_ax: PROVE counter_ax
                      FROM cnt_0{func <- func@C,loadin <- loadin@C},
                           cnt_1{func <- func@C,loadin <- loadin@C},
                           cnt_2{func <- func@C,loadin <- loadin@C},
                           cnt_3{func <- func@C,loadin <- loadin@C},
                           g2{fn <- func@c},
                           val_range_thm[2]{w <- func@C}

   p_ready_ax: PROVE ready_ax
                      FROM cnt_0{func <- func@C,loadin <- loadin@C},
                           cnt_1{func <- func@C,loadin <- loadin@C},
                           cnt_2{func <- func@C,loadin <- loadin@C},
                           cnt_3{func <- func@C,loadin <- loadin@C},
                           g2{fn <- func@c},
                           val_range_thm[2]{w <- func@C}


   p_g1: PROVE g1 FROM   power2_ax{i <- 2},
                         power2_ax{i <- 1},
                         power2_ax{i <- 0}

   p_g2: PROVE g2 FROM g1, val_range_thm[2]{w <- fn@C},
                       g2a{y <- val2(fn),m <- 4},
                       g2a{y <- val2(fn),m <- 3},
                       g2a{y <- val2(fn),m <- 2},
                       g2a{y <- val2(fn),m <- 1}

     p_g2a: PROVE g2a

   p_g3: PROVE g3 FROM val_mw_thm[2]{ii <- val2(fn)}, g1,
                       val_range_thm[2]{w <- fn},
                       val_bits_thm[2]{w1 <- fn, m <- 0,
                                       w2 <- mw2(val2(fn))},
                       mw_def[2]{ii <- val2(fn)},
                       mwm_def[2]{v <- val2(fn), m <- 2, n <- 2},
                       bitassign[2]{i <- 0, k <- 0,b <- BMOD2(val2(fn)),
                                    w <- mwm[2](DIVBY2(val2(fn)),1,2)}


   p_stb1: PROVE stb1 FROM val_mw_thm[2]{ii <- 0}, g1
```

```
(* ---------------- PROVE cnt_0 ------------------ *)

p_cnt_0: PROVE cnt_0 FROM NEXT0_ax{svt <- state@C,
                                   ldn <- loadin@C,
                                   fn <- func@C},
                          FETCH_ax{ct <- cnt(state@C),
                                   dbl <- double(state@C),
                                   ldn <- loadin@C,
                                   fn <- func@C},
                           make_triple_ax{x <- cnt(state@C),
                                          y <- BOOLF(bit2(0,func@C)),
                                          z <- fetchnode},
                          stb1{st <- state@C}


(* ---------------- PROVE cnt_1 ------------------ *)

p_cnt_1: PROVE cnt_1 FROM c1a{st <- state@C, fn <- func@C, ld <- loadin@C},
                          c1b{st <- state@C, fn <- func@C, ld <- loadin@C},
                          LOAD_ax{ct <- cnt(state@C),
                                  dbl <- BOOLF(bit2(0,func@C)),
                                  ldn <- loadin@C,
                                  fn <- func@C},
                          make_triple_ax{x <- loadin@C,
                                         y <- BOOLF(bit2(0,func@C)),
                                         z <- fetchnode}


p_c1a: PROVE c1a FROM NEXT0_ax{svt <- st@C,
                               ldn <- ld@C,
                               fn <- fn@C},
                              stb1{st <- st@C}

p_c1b: PROVE c1b FROM FETCH_ax{ct <- cnt(st@C),
                               dbl <- double(st@C),
                               ldn <- ld@C,
                               fn <- fn@C},
                      NEXT3_ax{svt <- make_triple(cnt(st@C),
                                                  BOOLF(bit2(0,fn@C)),
                                                  loadnode),
                               ldn <- ld@C,
                               fn <- fn@C},
                      make_triple_ax{x <- cnt(st@C ),
                                     y <- BOOLF(bit2(0,fn@C)),
                                     z <- loadnode},
                      val_mw_thm[2]{ii <- 3},
                      power2_ax{i <- 2},
                      power2_ax{i <- 1},
                      power2_ax{i <- 0}
```

```
(* ---------------- PROVE cnt_2 ------------------ *)

p_cnt_2: PROVE cnt_2 FROM c2a{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c2b{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c2c{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c2d{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c2e{fn<-func@C},
                          c2f{st<-state@C,fn<-func@C,ld<-loadin@C},
                          make_triple_ax{x<-ADD1(cnt(state@C)),
                 y<-BOOLF(bit2(0,func@C)),
                 z<-fetchnode}


p_c2a: PROVE c2a FROM stbl,
                      NEXT0_ax{svt <- st,
                               ldn <- ld,
                               fn <- fn}

p_c2b: PROVE c2b FROM FETCH_ax{ct <- cnt(st),
                               dbl <- double(st),
                               ldn <- ld,
                               fn <- fn}

p_c2c: PROVE c2c FROM NEXT1_ax{svt <- make_triple(cnt(st),
                                                  BOOLF(bit2(0,fn)),
                                                  inclnode),
                               ldn <- ld,
                               fn <- fn},
                      val_mw_thm[2]{ii <- 1},g1,
                      make_triple_ax{x <- cnt(st),
                                     y <- BOOLF(bit2(0,fn)),
                                     z <- inclnode}

p_c2d: PROVE c2d FROM  make_triple_ax{x <- cnt(st ),
                                      y <- BOOLF(bit2(0,fn)),
                                      z <- inclnode}


p_c2e:  PROVE c2e FROM g3,
                      BMOD2_ax{i <- 2},BMOD2_ax{i <- 0},
                      DIV_ax{i <- 2},DIV_ax{i <- 0}

p_c2f: PROVE c2f FROM INC1_ax{ct <- cnt(st@C),
                              dbl <- BOOLF(bit2(0,fn)),
                              ldn <- ld@C,
                              fn <- fn@C},
                     c2e
```

```
(* ---------------- PROVE cnt_3 ------------------- *)

p_cnt_3: PROVE cnt_3 FROM c3a{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c3b{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c3c{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c3d{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c3e{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c3f{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c3p{fn<-func@C},c3n,
                          c3g{st<-state@C,fn<-func@C,ld<-loadin@C},
                          c3h{st<-state@C,fn<-func@C,ld<-loadin@C},
                          make_triple_ax{x<-cnt(state@C),
                                         y<-BOOLF(bit2(0,func@C)),
                                         z<-inc1node},
                          make_triple_ax{x<-ADD1(cnt(state@C)),
                                         y<-BOOLF(bit2(0,func@C)),
                                         z<-inc2node},
                          make_triple_ax{x<-ADD1(ADD1(cnt(state@C))),
                                         y<-BOOLF(bit2(0,func@C)),
                                         z<-fetchnode}

p_c3a:  PROVE c3a

p_c3b:  PROVE c3b FROM stb1,NEXT0_ax{svt<-st@C,ldn<-ld@C,fn<-fn@C}

p_c3c:  PROVE c3c FROM FETCH_ax{ct<-cnt(st@C),dbl<-double(st@C),ldn<-ld@C,
                                fn<-fn@C}

p_c3d:  PROVE c3d FROM NEXT1_ax{svt<-make_triple(cnt(st@C),
                        BOOLF(bit2(0,fn@C)),inc1node),ldn<-ld@C,fn<- fn@C},
                        g1, val_mw_thm[2]{ii<-1},
                        make_triple_ax{x<-cnt(st@C),
                                       y<-BOOLF(bit2(0,fn@C)),z<-inc1node}

p_c3e:  PROVE c3e FROM INC1_ax{ct<-cnt(make_triple(cnt(st@C),
                                BOOLF(bit2(0,fn@C)),inc1node)),
                                dbl<-double(make_triple(cnt(st@C),
                        BOOLF(bit2(0,fn@C)),inc1node)),ldn<-ld@C,fn<-fn@C},
                        c3p,  make_triple_ax{x <- cnt(st@C),
                                             y <- BOOLF(bit2(0,fn@C)),
                                             z <- inc1node}


p_c3f:  PROVE c3f FROM NEXT2_ax{svt<-make_triple(ADD1(cnt(st@C)),
                        BOOLF(bit2(0,fn@C)),inc2node),ldn<-ld@C,fn<-fn@C},g1,
                        val_mw_thm[2]{ii<-2},
                        make_triple_ax{x<-ADD1(cnt(st@C)),
                                       y<-BOOLF(bit2(0,fn@C)),z<-inc2node}

p_c3n:  PROVE c3n FROM power2_ax{i<-1},power2_ax{i<-0},
                        val_mw_thm[2]{ii<-1},g1
```

```
   p_c3g:  PROVE c3g FROM make_triple_ax{x<-ADD1(cnt(st@C)),
                                        y<-BOOLF(bit2(0,fn)),z<-inc2node}

   p_c3h:  PROVE c3h FROM INC2_ax{ct<-ADD1(cnt(st@C)),dbl<-BOOLF(bit2(0,fn@C)),
                                  ldn<-ld@C,fn<-fn@C}

   p_c3p:  PROVE c3p FROM g3,
                          BMOD2_ax{i <- 3},BMOD2_ax{i <- 3},
                          DIV_ax{i <- 3},DIV_ax{i <- 1}


END cnt6_fa

triples: MODULE [firsttype, secondtype, thirdtype: TYPE]

EXPORTING triple, first, second, third, make_triple

THEORY

  triple: TYPE
  first: function[triple -> firsttype]
  second: function[triple -> secondtype]
  third: function[triple -> thirdtype]
  make_triple: function[firsttype, secondtype, thirdtype -> triple]

  x: VAR firsttype
  y: VAR secondtype
  z: VAR thirdtype
  t: VAR triple

  make_triple_ax: AXIOM
    x = first(make_triple(x, y, z))
      AND y = second(make_triple(x, y, z))
        AND z = third(make_triple(x, y, z))

(*  exists_triple_ax: AXIOM
    (FORALL t : (EXISTS x, y, z : t = make_triple(x, y, z)))
*)
END triples


bsignal: MODULE

EXPORTING signalval, signal_to_bool

THEORY
   b: VAR bool

   signalval: TYPE is bool

   signal_to_bool: function[signalval -> bool] = (LAMBDA b -> bool: b)

END bsignal
```

```
cnt6_blk: MODULE
MAPPING cnt6_fa ONTO words,triples[word[6],bool,word[2]],bsignal

THEORY

  (* ------------ define abbreviations for 'words' --------------- *)

  word2: TYPE is word[2]
  word6: TYPE is word[6]

  mw2: function[int -> word2] is mw[2]
  val2: function[word2 -> int] is val[2]
  bit2: function[int, word2 -> signalval] is bit[2]
  mw6: function[int -> word6] is mw[6]
  val6: function[word6 -> int] is val[6]
  bit6: function[int, word6 -> signalval] is bit[6]

  BOOLF: function[signalval -> bool] is signal_to_bool

  statevector: TYPE is triple

  (* ----------  logic constants defined in cnt6_fa -------------

  fetchnode: word2 = mw2(0)
  inc1node: word2 = mw2(1)
  inc2node: word2 = mw2(2)
  loadnode: word2 = mw2(3)

   ----------------- define logic variables ------------------ *)

  stv: VAR statevector
  ct,incout,loadin: VAR word6
  noinc: VAR bool
  nd,func: VAR word2
  dbl: VAR bool
  mplxsel: VAR bool

  (* ----------------- define functions  ------------------ *)

  INCLOGIC: function[word6,bool -> word6]
  INCLOGIC_ax: AXIOM INCLOGIC(ct,noinc) =
             IF noinc THEN ct
             ELSE ADD1(ct)
             END


  MULTIPLEX: function[word6,word6,bool -> word6]
  MULTIPLEX_ax: AXIOM MULTIPLEX(incout, loadin, mplxsel) =
                     IF mplxsel THEN incout
                     ELSE loadin
                     END

  MPLXCON: function[word2 -> bool] =
           (LAMBDA nd -> bool:  NOT (val2(nd) = 3) )
```

```
INCCON: function[word2 -> bool] =
        (LAMBDA nd -> bool:  (val2(nd) = 0) )


NEXTNODE: function[word2,word2,bool -> word2]
(* ------------------------------------------------------------------------
NEXTNODE_ax: AXIOM NEXTNODE(nd,func,dbl) =
               IF val2(nd) = 0 THEN
                  IF val2(func) = 0 THEN fetchnode
                  ELSIF val2(func) = 1 THEN loadnode
                  ELSE inclnode
                  END
               ELSIF val2(nd) = 1 THEN
                  IF dbl THEN inc2node
                         ELSE fetchnode
                  END
             ELSE
                  fetchnode
             END
 ------------------------------------------------------------------------ *)

NEXTNODE0_ax: AXIOM val2(nd) = 0 IMPLIES
                 NEXTNODE(nd,func,dbl) =
                    IF val2(func) = 0 THEN fetchnode
                    ELSIF val2(func) = 1 THEN loadnode
                    ELSE inclnode
                    END

NEXTNODE1_ax: AXIOM val2(nd) = 1 IMPLIES
                 NEXTNODE(nd,func,dbl) = IF dbl THEN inc2node
                                         ELSE fetchnode
                                         END

NEXTNODE2a3_ax: AXIOM val2(nd) = 2 or val2(nd) = 3 IMPLIES
                   NEXTNODE(nd,func,dbl) = fetchnode

COUNTLOGIC: function[statevector,word6,word2 -> statevector] =
               (LAMBDA stv, loadin, func -> statevector:
               make_triple( MULTIPLEX(INCLOGIC(count(stv),
                                               INCCON(node(stv))   ),
                                      loadin,
                                      MPLXCON(node(stv)) ),
                            BOOLF(bit2(0,func)),
                            NEXTNODE(node(stv),func,double(stv)) )
                )


cnt6_fa.NEXT: function[statevector,word6,word2 -> statevector] = COUNTLOGIC

 (* ------- LEMMAs needed to prove NEXT0_ax ---------- *)

 ldn: VAR word6
 fn: VAR word2
```

```
case_0:  LEMMA val2(node(stv)) = 0 IMPLIES
          COUNTLOGIC(stv,ldn,fn) = FETCH(count(stv),double(stv),ldn,fn)

  cs0a: LEMMA val2(node(stv)) = 0 IMPLIES
          COUNTLOGIC(stv,ldn,fn) =
               make_triple( count(stv),
                            BOOLF(bit2(0,fn)),
                            NEXTNODE(node(stv),fn,double(stv)) )

  cs0b: LEMMA val2(node(stv)) = 0 IMPLIES
                make_triple( count(stv),
                             BOOLF(bit2(0,fn)),
                             NEXTNODE(node(stv),fn,double(stv)) ) =
                FETCH(count(stv),double(stv),ldn,fn)

(* ------- LEMMAs needed to prove NEXT1_ax ---------- *)

case_1:  LEMMA val2(node(stv)) = 1 IMPLIES
          COUNTLOGIC(stv,ldn,fn) = INC1(count(stv),double(stv),ldn,fn)

  cs1a: LEMMA val2(node(stv)) = 1 IMPLIES
          COUNTLOGIC(stv,ldn,fn) =
               make_triple( ADD1(count(stv)),
                            BOOLF(bit2(0,fn)),
                            NEXTNODE(node(stv),fn,double(stv)) )

  cs1b: LEMMA val2(node(stv)) = 1 IMPLIES
                make_triple( ADD1(count(stv)),
                             BOOLF(bit2(0,fn)),
                             NEXTNODE(node(stv),fn,double(stv)) ) =
                INC1(count(stv),double(stv),ldn,fn)

(* ------- LEMMAs needed to prove NEXT2_ax ---------- *)

case_2:  LEMMA val2(node(stv)) = 2 IMPLIES
          COUNTLOGIC(stv,ldn,fn) = INC2(count(stv),double(stv),ldn,fn)

  cs2a: LEMMA val2(node(stv)) = 2 IMPLIES
          COUNTLOGIC(stv,ldn,fn) =
               make_triple( ADD1(count(stv)),
                            BOOLF(bit2(0,fn)),
                            NEXTNODE(node(stv),fn,double(stv)) )

  cs2b: LEMMA val2(node(stv)) = 2 IMPLIES
                make_triple( ADD1(count(stv)),
                             BOOLF(bit2(0,fn)),
                             NEXTNODE(node(stv),fn,double(stv)) ) =
                INC2(count(stv),double(stv),ldn,fn)

(* ------- LEMMAs needed to prove NEXT3_ax ---------- *)

case_3:  LEMMA val2(node(stv)) = 3 IMPLIES
          COUNTLOGIC(stv,ldn,fn) = LOAD(count(stv),double(stv),ldn,fn)
```

```
    cs3a: LEMMA val2(node(stv)) = 3 IMPLIES
            COUNTLOGIC(stv,ldn,fn) =
                make_triple( ldn,
                             BOOLF(bit2(0,fn)),
                             NEXTNODE(node(stv),fn,double(stv)) )

    cs3b: LEMMA val2(node(stv)) = 3 IMPLIES
                make_triple( ldn,
                             BOOLF(bit2(0,fn)),
                             NEXTNODE(node(stv),fn,double(stv)) ) =
                LOAD(count(stv),double(stv),ldn,fn)

PROOF

  p_case_0: PROVE case_0 FROM cs0a,cs0b


    p_cs0a: PROVE cs0a FROM INCLOGIC_ax{ct <- count(stv),
                                        noinc <- INCCON(node(stv))},
                           MULTIPLEX_ax{incout <- count(stv),
                                        loadin <- ldn,
                                        mplxsel <- MPLXCON(node(stv))}

    P_cs0b: PROVE cs0b FROM FETCH_ax{ct <- count(stv),
                                     dbl <- double(stv),
                                     ldn <- ldn, fn <- fn},
                           NEXTNODE0_ax{nd <- node(stv),
                                        func <- fn,
                                        dbl <- double(stv)}


  p_case_1: PROVE case_1 FROM cs1a,cs1b


    p_cs1a: PROVE cs1a FROM INCLOGIC_ax{ct <- count(stv),
                                       noinc <- INCCON(node(stv))},
                           MULTIPLEX_ax{incout <- ADD1(count(stv)),
                                        loadin <- ldn,
                                        mplxsel <- MPLXCON(node(stv))}


    P_cs1b: PROVE cs1b FROM INC1_ax {ct <- count(stv),
                                     dbl <- double(stv),
                                     ldn <- ldn, fn <- fn},
                           NEXTNODE1_ax{nd <- node(stv),
                                        func <- fn,
                                        dbl <- double(stv)}


  p_case_2: PROVE case_2 FROM cs2a,cs2b
```

```
    p_cs2a: PROVE cs2a FROM INCLOGIC_ax{ct <- count(stv),
                                        noinc <- INCCON(node(stv))},
                           MULTIPLEX_ax{incout <- ADD1(count(stv)),
                                        loadin <- ldn,
                                        mplxsel <- MPLXCON(node(stv))}

    P_cs2b: PROVE cs2b FROM INC2_ax{ct <- count(stv),
                                    dbl <- double(stv),
                                    ldn <- ldn, fn <- fn},
                           NEXTNODE2a3_ax{nd <- node(stv),
                                          func <- fn,
                                          dbl <- double(stv)}


  p_case_3: PROVE case_3 FROM cs3a,cs3b

  p_cs3a: PROVE cs3a FROM
                MULTIPLEX_ax{incout <-INCLOGIC(count(stv),INCCON(node(stv))) ,
                             loadin <- ldn,
                             mplxsel <- MPLXCON(node(stv))}

  p_cs3b: PROVE cs3b FROM LOAD_ax{ct <- count(stv),
                                  dbl <- double(stv),
                                    ldn <- ldn, fn <- fn},
                          NEXTNODE2a3_ax{nd <- node(stv),
                                         func <- fn,
                                         dbl <- double(stv)}

END cnt6_blk


cnt6_cir: MODULE

MAPPING cnt6_blk ONTO words, triples, bsignal

THEORY

  (* ---------------------------- abbreviations ---------------------------- *)

  word2: TYPE is word[2]
  word6: TYPE is word[6]
  cntrlsigs: TYPE is triple[bool,bool,word[2]]

  bit2: function[int, word2 -> bool] is bit[2]
  bit6: function[int, word6 -> bool] is bit[6]
  assign2: function[int,bool,word2 -> word2] is assign[2]
  assign6: function[int,bool,word6 -> word6] is assign[6]

  (* --------------------- circuit elements ------------------------------- *)

  b,b1,b2,b3,b4: VAR bool
```

```
INV: function [bool -> bool] = (LAMBDA b -> bool: not b)
NAND2: function [bool, bool -> bool] =
       (LAMBDA b1,b2 -> bool: not (b1 and b2))
NAND3: function [bool, bool, bool -> bool] =
       (LAMBDA b1,b2,b3 -> bool: not (b1 and b2 and b3))
NAND4: function [bool, bool, bool, bool -> bool] =
       (LAMBDA b1,b2,b3,b4 -> bool: not (b1 and b2 and b3 and b4))
XNOR: function [bool, bool -> bool] =
       (LAMBDA b1,b2 -> bool: not (not b1 and b2 or b1 and not b2))
NOR2: function [bool, bool -> bool] =
       (LAMBDA b1,b2 -> bool: not (b1 or b2))

(* ---------------------- logic variables ------------------------------ *)

i0,i1,i2,i3,i4,i5: VAR bool
lbit,lsel,incbit,incsel: VAR bool
incout,loadin,cntr: VAR word6
mplxsel,noinc,Double: VAR bool
Node,Func: VAR word2

(* ---------------------- circuit definition --------------------------- *)

output: function [bool,bool,bool,bool,bool,bool -> word6] =
        (LAMBDA i0,i1,i2,i3,i4,i5 -> word6:
                assign6(0,i0,
                 assign6(1,i1,
                  assign6(2,i2,
                   assign6(3,i3,
                    assign6(4,i4,
                     assign6(5,i5,newword[6])))))))


bitsel: function[bool,bool,bool,bool -> bool] =
        (LAMBDA lbit,lsel,incbit,incsel -> bool:
         NAND2( NAND2(lbit,lsel), NAND2(incbit,incsel)) )


MPLEXCIRC: function[word6,word6,bool -> word6]
MPLEXCIRC_ax: AXIOM MPLEXCIRC(incout, loadin, mplxsel) =
            output(
         bitsel(bit6(0,loadin),INV(mplxsel),bit6(0,incout),mplxsel),
         bitsel(bit6(1,loadin),INV(mplxsel),bit6(1,incout),mplxsel),
         bitsel(bit6(2,loadin),INV(mplxsel),bit6(2,incout),mplxsel),
         bitsel(bit6(3,loadin),INV(mplxsel),bit6(3,incout),mplxsel),
         bitsel(bit6(4,loadin),INV(mplxsel),bit6(4,incout),mplxsel),
         bitsel(bit6(5,loadin),INV(mplxsel),bit6(5,incout),mplxsel)
                  )

carry4bar: function[word6,bool -> bool] =
           (LAMBDA cntr,noinc -> bool:
              NAND4(INV(noinc),bit6(0,cntr),bit6(1,cntr),bit6(1,cntr))
           )
```

```
INCCIRC: function[word6,bool -> word6] =
        (LAMBDA cntr,noinc -> word6:
           output(
             XNOR(bit6(0,cntr), noinc),
             XNOR(bit6(1,cntr), NAND2(INV(noinc),bit6(0,cntr)) ),
             XNOR(bit6(2,cntr),
                  NAND3( INV(noinc), bit6(0,cntr), bit6(1,cntr) ) ),
             XNOR(bit6(3,cntr),carry4bar(cntr,noinc) ),
             XNOR(bit6(4,cntr),
                  NAND2(INV(carry4bar(cntr,noinc)), bit6(3,cntr)  )
                  ),

             XNOR(bit6(5,cntr),
                  NAND3(INV(carry4bar(cntr,noinc)),
                        bit6(3,cntr) ,
                        bit6(4,cntr)  )        )

                 )
          )


inccon: function[word2 -> bool] =
          (LAMBDA Node -> bool:
             NOR2(bit2(0,Node),bit2(1,Node))    )

common: function[word2,word2 -> bool] =
          (LAMBDA Node,Func -> bool:
             NAND3(inccon(Node),INV(bit2(1,Func)),bit2(0,Func))  )

CONTROLCIR: function[word2,word2,bool -> cntrlsigs] =
            (LAMBDA Node,Func,Double -> cntrlsigs:
               make_triple( inccon(Node),
                            NAND2(bit2(0,Node),bit2(1,Node)),
                            assign2(0, NAND2(common(Node,Func),
                                             NAND2(inccon(Node),bit2(1,Func))
                                            ),
                              assign2(1,NAND2(common(Node,Func),
                                              NAND3(Double,
                                                    bit2(0,Node),
                                                    INV(bit2(1,Node) ) )),
                                      newword[2])
                                     )
                           )
             )

 (* ------------------- Mappings to "cnt6_blk" ------------------------- *)

 cnt6_blk.INCLOGIC: function[word6,bool -> word6] = INCCIRC

 cnt6_blk.MULTIPLEX: function[word6,word6,bool -> word6] = MPLEXCIRC

(*
```

```
    cnt6_blk.INCCON: function[word2,word2,bool -> bool] =
                     (LAMBDA Node,Func,Double -> bool:
                         first(CONTROLCIR(Node,Func,Double))
                     )


    cnt6_blk.MPLXCON: function[word2,word2,bool -> bool] =
                     (LAMBDA Node,Func,Double -> bool:
                         second(CONTROLCIR(Node,Func,Double))
                     )

 *)
    cnt6_blk.NEXTNODE: function[word2,word2,bool -> word2] =
                     (LAMBDA Node,Func,Double -> word2:
                         third(CONTROLCIR(Node,Func,Double))
                     )

 END cnt6_cir


 ineq_cases: MODULE

 THEORY
    y,m: VAR int

    Y_0: THEOREM (y>=0 AND y<1) IMPLIES y=0
    Y_01: THEOREM (y>=1 AND y<2) IMPLIES y=1
    Y_S: THEOREM (y>=0 AND y<2) IMPLIES ((y>=0 AND y<1) OR (y>=1 AND y<2))

    Y_1: THEOREM (y>=0 AND y<2) IMPLIES (y=0 OR y=1)

    M_R: THEOREM (y>=0 AND y<=m) IMPLIES ((y>=0 AND y<=m-1) OR (y=m))

 PROOF

   pY0: PROVE Y_0
   pY01: PROVE Y_01

   pYS: PROVE Y_S

   pY1: PROVE Y_1 FROM
         Y_S, Y_0, Y_01

   pMR: PROVE M_R


 END ineq_cases

 int_inductions: MODULE

 EXPORTING next,pred,geq,bge

 THEORY

   i,j: VAR int
   d1, d2, d3, d4, de: VAR int
   x,y,z,s: VAR int
```

```
  First: int = 0
  next: function[int -> int] = (LAMBDA i -> int: i+1)

  pred: function[int -> int] = (LAMBDA i -> int: IF i > 0 THEN i-1 ELSE 0 END)

  geq: function[int,int -> bool] = (LAMBDA i,j -> bool: (i>=j))

  bge: function[int -> bool] =
         (LAMBDA i -> bool: IF i>=0 THEN true ELSE false END )

  p: VAR function[int -> bool]


  int_complete: THEOREM (FORALL d1: geq(d1,First) IMPLIES
             (FORALL d3: (geq(d3,First) AND geq(d1,d3) AND d3 ~= d1) IMPLIES
                 p(d3)) IMPLIES p(d1))
                   IMPLIES (FORALL d2: geq(d2,First) IMPLIES p(d2))


  int_induction: THEOREM (p(First) AND (FORALL d1: p(d1) IMPLIES p(next(d1))))
     IMPLIES (FORALL d2:  geq(d2,First) IMPLIES p(d2) )

  int_induct_by_2: THEOREM  (p(First) AND p(next(First)) AND (FORALL d1: p(d1)
                          IMPLIES p(next(next(d1)))))
                            IMPLIES (FORALL d2: ( geq(d2,First) IMPLIES p(d2) ))


END int_inductions
```

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-100669 AVSCOM TM-88-B-017		

4. Title and Subtitle
Hardware Proofs Using EHDM and the RSRE Verification Methodology

5. Report Date
December 1988

6. Performing Organization Code

7. Author(s)
Ricky W. Butler and Jon A. Sjogren

8. Performing Organization Report No.

9. Performing Organization Name and Address
NASA Langley Research Center
Hampton, VA 23665-5225 and
Joint Research Programs Office/AVRADA, Langley Research Center, Hampton, VA 23665-5225

10. Work Unit No.
505-66-21-01

11. Contract or Grant No.

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546-0001
and
U.S. Army Aviation Systems Command
St. Louis, MO 63120-1798

13. Type of Report and Period Covered
Technical Memorandum

14. Army Project No.
1L161102AH45E

15. Supplementary Notes
Ricky W. Butler: Langley Research Center, Hampton, Virginia
Jon A. Sjogren: Joint Research Programs Office, AVRADA-AVSCOM, Langley Research Center, Hampton, VA

16. Abstract

This paper examines a methodology for hardware verification developed by Royal Signals and Radar Establishment (RSRE) in the context of the SRI International's Enhanced Hierarchical Design Methodology (EHDM) specification/verification system. The methodology utilizes a four-level specification hierarchy with he following levels: functional level, finite automata model, block model, and circuit level. The properties of a level are proved as theorems in the level below it. In this paper, this methodology is applied to a 6-bit counter problem and is critically examined. The specifications are written in EHDM's specification language, Extended Special, and the proofs are improving both the RSRE methodology and the EHDM system.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
Verification Formal Proof Formal Specification Theorem Proving Hardware Verification	Unclassified - Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	89	A05

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171