High Speed, Complete and Proper Logic Design

By David McFarland

ABSTRACT: Representing logic in a hierarchy of Karnaugh maps, where the hierarchy can be collapsed for don't care conditions, enables all transitions from all states for all input combinations to be specified, yet keeps the combinatorial explosion manageable.

Design automation must do more than just speed development. It must also find errors in the early, inexpensive design phase and ensure that the final specification is correct. The challenge is to do so with a large number of inputs or states. This paper describes the Logic Design Tool, which provides these functions.

The motivation behind design automation is well understood - find and stop human errors. The unwritten goal is to make logic design as easy as a video game. But design automation must do more than just speed development. It must find errors early in the inexpensive design phase and ensure that the final specification is correct.

LDT does this with a representation made of Karnaugh maps linked in a tree structure hierarchy. Areas in the structure that do not have bearing on its operation can be ignored by being collapsed, so the size of the representation is reduced.

Application. An error in a project's production phase is typically orders of magnitude more expensive to fix than finding that same error in the design phase. And because digital logic is now present everywhere, this expense can be manifest in time, money, litigation or even human life.

Designing logic with more than four or five variables is, however, prone to error, because the specification may not cover all desired conditions or may include unwanted actions.

Truth tables, Binary Decision Diagrams (BDD) and Karnaugh maps can show all conditions and actions, but are cumbersome and subject to combinatorial explosion.

A correct design means it is complete and proper. A complete description accounts for all the possible stimuli; a proper model is unambiguous and each condition only has one response.

Digital logic typically suffers from three kinds of errors: holes, conflicts and being just plain wrong. Holes occur when no response is specified for the input and state conditions. Conflicts occur when multiple responses are specified for the same condition. Being wrong occurs when a desired response is not generated for a given set of conditions. One cause of these errors is the inability to visualize the effect of all combinations when more than four or five variables are required.

Various methods with benefits and disadvantages are used to defined logic.

Text is inexact and typically only covers apparent conditions. Boolean equations are difficult to understand and reduce when the logic includes more than four or five input variables. Truth tables do not show patterns that would be evident in a Karnaugh map. Truth tables are also tedious and subject to combinatorial explosion of cases. If, then, else statements, as would be found in software code, may have hidden or lost conditions that are easily overlooked. Binary decision diagrams are bit oriented, so visualizing the relationship of a several output variables is difficult. State Charts are two dimensional and do not account for the combinations of inputs necessary for each transition. They can quickly become spaghetti charts. Entered variables used in conjuction with a Karnaugh map do show all conditions, but reducing the size of the map involves mathematical equations and techniques that can easily introduce human error.

However, LDT can specify all transitions from all states for all input combinations and do so with a large number of inputs and states without mathematical equations. If no states are required, LDT can specify combinatorial logic.

Statement of Need. In a EE Times article from July, 1995, Stephen L. Wasson raised the need for hierarchical modeling of logic. He outlined present methods of specifying logic such as compilable flowcharts, HDLs, and bubble diagrams, and then called for a future FSM generation tool. He said:

"Of course, all of these fine-grained manipulations should be done automatically, and some day they will be. Bubble diagram tools are an excellent start, but they need to be expanded to provide hierarchical modeling to facilitate more complex, hierarchical state machines and to accommodate more rigorous next-state analysis. Designers should be able to push through bubble trees down to leaf nodes that contain interactive Karnaugh maps. These maps should be automatically filled in from the initial next-state equation specifications, and all holes, conflicts, and else-assumptions should be clearly indicated. At this level, the designer should be able to perform map manipulations that automatically back-annotate the next-state equations."

The Logic Design Tool is a software executable that provides these functions. More importantly, it can do so in the presence of a large number of inputs and states.

LDT Advantages. LDT is based upon an underlying method which relates binary input variables to outputs and (in the case of sequential logic) next state variables through a hierarchy of Karnaugh maps. The method extends the number of variables that can be used to implement combinational logic, and increases the number of state transitions that can be practically included in sequential logic. This method allows LDT to generate a control specification that is complete (all conditions are specified) and unambiguous (only one action is possible for any combination of variables). All actions the logic will take, under all conditions, are specified without the need for mathematical relationships or logic equations, which are prone to human error.

Because the specification is complete, certain analysis is possible, such as an exhaustive search for worst and best-case performance paths or reduction of logic minterms. LDT handles the combinatorial explosion of variables by collapsing regions of the logic space where variables are don't cares.

LDT is especially useful in applications where system behavior must be proven correct, such as fault tolerant or data secure logic. At the option of the operator, LDT can perform reduction of specified logic and produce source files for automatic implementation of a state machine in hardware or software.

LDT displays inputs, present states, next states and outputs in both graphical and Boolean format. The graphical format presents a visual plot of a binary or Boolean equation and shows the behavior of the state machine under all conditions. With

multiple views of the specification, design errors are less likely. Where the state machine is to be implemented with Boolean logic, the number of minterms can be reduced. Reduction in logic can result in less complex, more reliable, and cheaper systems with faster execution.

Although LDT can easily specify simple systems, it is of most benefit to developers of software or hardware that is logic intensive, has many modes, many sequences of operation, or where operation must be very fast. It can reduce the specification to a sum of product output for fewer gate delay hardware implementation. Multiple simple systems can sometimes be incorporated into a single larger system with the benefit of lower overall system cost and complexity. LDT makes this incorporation easier (and in some cases possible) due to its ability to manage a larger number of input variables and states.

LDT will aid the user in specifying system behavior under differing conditions for both synchronous and asynchronous processes. Since all actions of the state machine are known under all conditions, and generation of the implementation is via a known algorithm, LDT is useful in applications where the behavior of the system must be proven correct, such as highly reliable, fault tolerant or secure data systems. This is in comparison to software built with nested if-then-else structures, where system modes are not as easily viewed.

If software execution speed of an algorithm is important, the state machine can be implemented with a software array, such that the steps for execution are simply a single memory lookup and decode. This becomes useful in software systems that are time critical, such as real-time operating systems.

To prove the utility of the tool, both combinational and sequential control logic of some LDT options have been implemented with state machines and combinational logic generated by LDT itself.

Either hardware or software behavior can be specified with LDT, so that the decision to implement functions in hardware or software can be made after specification with LDT and according to throughput estimates generated by the tool. Specification of the state machine during analysis is done at an abstract level, so it does not need to influence the ultimate design.

LDT will generate software source code for C, Ada, Pascal, or 80386 assembly. LDT will also generate source files in VHDL.

Within LDT, arbitrary inputs or output conditions can be labeled as don't cares in order to minimize control specifications. LDT graphics are very simple, but the simplicity does not detract from its utility. LDT supports timing analysis whereas most other tools can not offer this option because the resulting specification is not complete.

Most digital control systems using more than 4 or 5 variables will contain errors. Where LDT is used to examine that control, it is likely to find an error, find a case not considered, or reduce the logic further. Discussions about a digital control system often include some "what-ifs", but all the other "what-if" cases are left unexamined. LDT allows each case to be examined and discussed, separate from all other cases.

Description. Karnaugh Maps are a graphical representation of a truth table and are often used to descibe problems with a small number of variables. (See figure 1) Karnaugh maps are used to visualize, specify and reduce a binary function because they give a straightforward way to see minterms in a function and to help the designer minimize them.

Karnaugh map fundamentals are taught in most standard references. An example is Thomas McCalla, Digital Logic and Computer Design: New York, Macmillan Publishing, 1992

Karnaugh map displayed in figure 2 has nine input variables, and to be displayed, the map must be replicated again for h and again for i, in different directions. But to do so, the map would overflow the display page.

As shown, two dimensional Karnaugh map representation becomes cumbersome for large or complex functions. (e.g. see column 1, lines 32 43 of U.S. Patent 4,583,169; and Hoerner and Heilweil, Introduction to Boolean Algebra and Logic Design: New York, McGraw Hill, 1964, p.168). So there is a need for a method that can relate a larger number of variables.

Entered variables can extend the map input size, but involve equations and steps that can easily introduce human error. See “Engineering Digital Design” by Richard F. Tinder, Academic Press, San Diego, 2000, ISBN 0-12-691295-5, pp. 158.

LDT uses an alternate representation of logic as seen in figure 3. It starts with a blackbox approach to the design. Inputs, outputs and state bits are identified and then grouped into fields, where each field is displayed in one window at a time. (For combinatorial logic, there are no state bits and no state storage.) At present, the maximum number of combinations LDT can display at a time is six, where 2**6 or 64 combinations are shown in a map.

In this example, the first field, FIELD_0, is the state bit field, which with two state bits will allow four states to be described. If more states were needed, another bit could be added so that eight states could be used, or with four bits, 16 states, and so on. The next field, FIELD_1, is an intermediate set of inputs, and the intermediate field will display 16 combinations as is shown in the next slide. FIELD_2 is the final leaf or transition field, which with 3 bits will have eight combinations displayed. The desired transform output or next state bit condition is entered in the transition field.

Output and next state equations are a function of the inputs. The next state bit a_next is a function of a,b,c,d,e,f,g,h and I, as is also true of b_next and the output variable x.

Figure 4 is a transform made of the map hierarchy that relates a binary output variable to a set of binary input variables. The transform's input variables are grouped into the successive fields assigned in figure 3. Each field is then given a map having cells which are each a combination of that field's variables. A different field combination map is assigned to each successive field for each preceding field cell, until it reaches the last field cell of each cell of each preceding field. This process forms a field cell chain associated with the last field cell and that combination of inputs. Finally, binary values are assigned to all field cell chains according to the desired transform.

LDT can relate a large number of inputs to output variables because only a part of the hierarchy, the Karnaugh map in that field, needs to be viewed at any one time. This visual hierarchy aids the design and enhances a user's understanding of the logic transforms. This is especially true for transforms involving large numbers of variables, partly because patterns are easily identified and compared in the map.

LDT also insures all input combinations have been considered and that only one transform value has been assigned to each of the combinations (complete and unambiguous).

Although the method used by LDT allows a large number of variables to be examined, the combinatorial explosion can still make the assignment of each combination unwieldy.

To solve this problem, the number of combinations that are displayed can be reduced. In some areas of the hierarchy, input variables are don't cares and have no bearing on the output values. See figure 5. So, to reduce the hierarchy size and the number of combinations that must be specified, LDT enables the display of only those inputs that do affect the output. The identification of don't care inputs allows the hierarchy in that region to be collapsed.

In this example, the combination map of FIELD_1 related to FIELD_0's combination a'b are all don't cares. That field is collapsed to a single combination cell in FIELD_1. The three inputs in the map at the final field, FIELD_2, are not don't cares, so all eight combination cells in that field must be displayed.

Illustrative Examples. Three examples are given to further illustrate the use and utility of LDT.

While both SAFEMSL and SUREMSL examples have the same number of inputs, outputs and states, their behavior is very different.

SAFEMSL is a hypothetical ballistic missile which must fire only when the correct sequence of commands is entered, otherwise it must halt. If a hardware failure should occur, the system must fail in a safe or halted state. Inputs presented to the controller must follow a given enforced sequence of combinations or else the system will halt in an invalid state.

SUREMSL is a hypothetical tactical missile controller that is only enabled when a jet is in a dogfight. The controller must fail active, so that if a hardware failure occurs, the missile will fire, even if a missed target is likely. The system has a preferred sequence of input conbinations that cause the missile to fire, but a number of other sequences are allowed.

CPLXMSL is a actual controller for a fielded battleship defense missile. It is shown here briefly, but explained in detail in the user manual.

SAFEMSL Example. In figure 6, SAFEMSL inputs are grouped in two fields, FIELD_0, which defines the four present states, and FIELD_1, which specifies the next state transitions for each of the 16 combinations of inputs. SAFEMSL present states are listed in the 4-combination state map of FIELD_0. The SAFEMSL controller's desired behavior, shown on the enforced path, is restricted by the four 16-combination transition maps in FIELD_1 above the state map of FIELD_0. The starting state is READY (0). If FUEL (w) and COMPTR (x) become true, the machine will transition to AIM (1).

FUEL must be true before CPMTR or else the machine will transition to the INVALD (2) state. If COMPTR becomes true, FUEL must stay true or the machine will transiltion to INVALD. The sequence of input combinations is also restricted in the AIM state such that AIMED must become true after FUEL, then COMPTR must become true, or the machine will transition to the INVALD state.

Once again, BUTTON must come true after FUEL then COMPTR then AIMED to cause a missile to FIRE. Otherwise, the machine will halt in the INVALD state. Enforcing this sequence would be difficult without the ability to see this path in the Karnaugh maps.

In figure 15 are the next state equations for the SAFEMSL specification. These equations include an error in the INVALD transition map. This INVALD transition map has an erroneous transition in INVALD state 2 to FIRE state 3 at combination wx'y'z'.

The error is not aparent from these equations, but can be easily identified in the slide as a '3' surrounded by a group of '2's. This case is meant to show the need for visualization of the logic.

In figure 16 are the next state equations for the SAFEMSL specification shown in figure 6, however, in this case SAFEMSL has no '3' transition error in the INVALD map. Again, the equations do not easily show the difference between the specification with and without the error, and yet the error could have drastic consequences (Oops, we just took out Vladislovstok. Sorry about that.)

SUREMSL Example. In figure 7, SUREMSL present states are, like SAFEMSL, listed in the 4-combination state map of FIELD_0. The SUREMSL controller's behavior, shown as the preferred path, is enabled by the four 16-combination

transition maps shown above the state map in FIELD_1. The starting state is READY (0). If FUEL (w) and COMPTR (x) become true in any order, the machine will transition to state AIM (1).

Any time FUEL and COMPTR are both true, the machine will transition to the INVALD (2) state. Unlike SAFEMSL, any sequence of their becoming true will cause the machine to transition to AIM. The sequence of input combinations also does not affect the transition to the FIRE (3) state because anytime COMPTR, FUEL and AIMED are all true, the machine will transition to the FIRE state.

Because the INVALD state will never be reached, all transitions in that state can be designated as don't cares. The don't cares can be used at the discretion of the reduction algorithm to minimize the number of minterms needed to implement SUREMSL.

Collapsed SUREMSL. Transition patterns are repeated in the SUREMSL transition map, so the size of the transition maps can be reduced as follows. See figure 8.

In the READY state, the transition pattern is repeated such that the FIRE input is a don't care and its transition map is reduced to eight combinations.

In the AIM map, FUEL, COMPTR and AIMED are don't cares and the AIM transition map is reduced to two combinations of BUTTON. All inputs in the INVALD transition map are don't cares, and the map is reduced to one combination where all combinations are don't cares. All inputs to the FIRE state result in FIRE output, and the map can be reduced to one combination that stays in the FIRE state.

The SUREMSL is then collapsed from a total of 64 transition map combinations to 12, for much easier transition entry.

CPLXMSL Example. In figure 9, CPLXMSL indicates how involved real world problems can become. CPLXMSL has 16 states and 29 transitions. The number of transitions makes the state transition diagram into a spagetti chart that is difficult to follow.

However, even though this controller is complex. LDT was able to implement and test the controller in 3 hours. The actual implementation of this controller, done with normal coding, is claimed to have taken 2 months to write and install.

Figure 10 is a state analysis of the CPLXMSL example, where dead and hanging states have been found. Dead states have no transition to another state. Hanging states have no transition to that state from any other state.

No decision states, where there is an automatic transition to another state, can also be shown.

Because the specification is complete, certain analysis is possible, such as an exhaustive search for worst and best-case performance paths or for reduction of logic minterms.

Figure 11 shows the booean equation of next state bit A for the real controller CPLXMSL.

This equation is intractable and would be difficult to correctly generate without a design aid such as LDT.

Specification Entry, Views, Analysis and Implementation. See figure 13. The likelihood of finding design errors increases when mutliple means are available to enter the specification, view it, analyze it, and implement it. Seeing the error from 'different angles' gives more opportunity to recognize and fix the error before the error becomes latent in the target system. Also, rather than entering each transition in each individual combination in the transition map, the transitions can be specifiied via a Boolean equation that describes when the present state should transition to the given next state. This alternate entry eases the specification burden. For even less rigor and faster entry, the output bit equations can be specified at the next state and output bit level, then viewed. Don't cares and a collapsable hierarchy can also reduce entry effort.

The final specification can be stepped through states with an interactive debugger. It can also be shown as a state transition diagram or as a truth table. Analysis can be used to verify that only the intended dead, hanging and no-decision states are present. The logic reduction report can show that reduction was as expected and the result matches the required behavior.

LDT generates a variety of software and hardware source code and code drivers in several types of implementations. The choices enable the specification to be made at an abstract level without tying the design to a particular implementation in hardware or software.

Range of Rigor. There are several ways to enter a specification into LDT. Each insures a higher degree of certainty that each condition was examined. This ranges from single entry for each combination, where there are no defaults and each entry must be actively specified, to entry with a next state bit level boolean equation, where only the result is viewed.

So the entry method should be chosen based upon the certainty of correctness that is required for a particular project.

Conclusion. LDT can specify all transitions from all states for all input combinations.

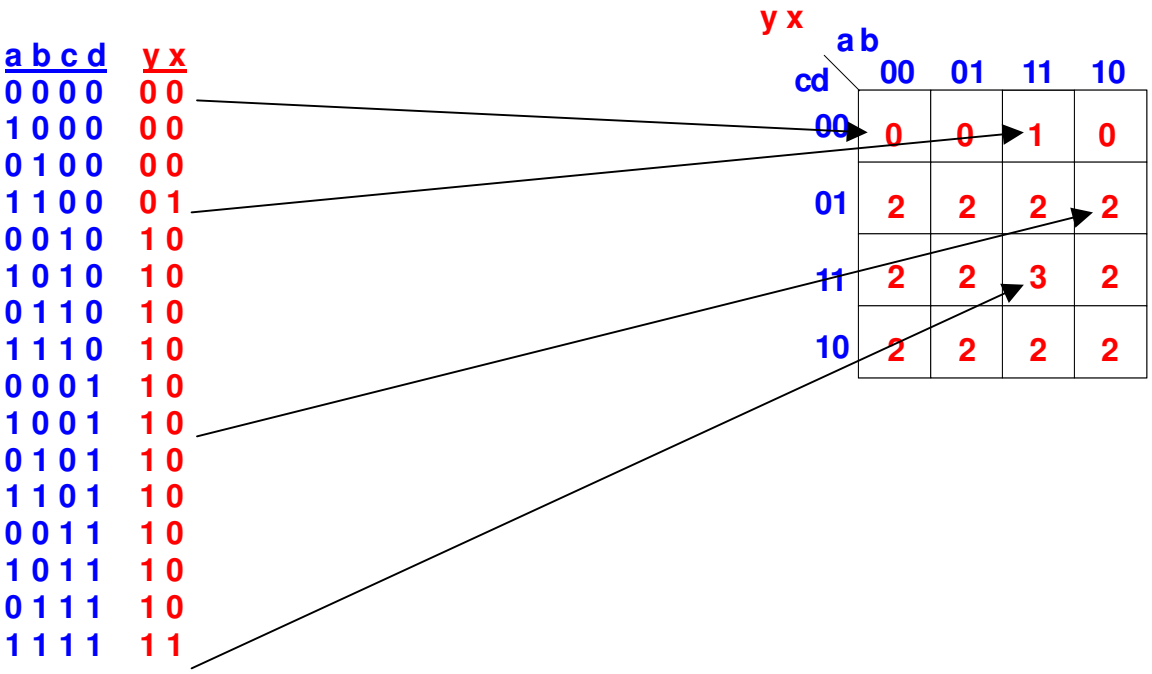
Because of LDT's higher degree of rigor, and its additional visibility into the specification, it increases the chance of finding logic faults early in the inexpensive design phase.

Specifications are debugged interactively and implementation is automated, reducing the chance for human error and speeding development time.

The amount of rigor and the size of collapsible don't care space is user selectable and can be tailored to meet system needs.

LDT has many options that make it a valuable fit for a broad set of applications.

4 INPUT 2 OUTPUT KARNAUGH MAP



x = a and b and not c and not d or a and b and c and d
y = c or d

Figure 1 – Karnaugh Map Example.

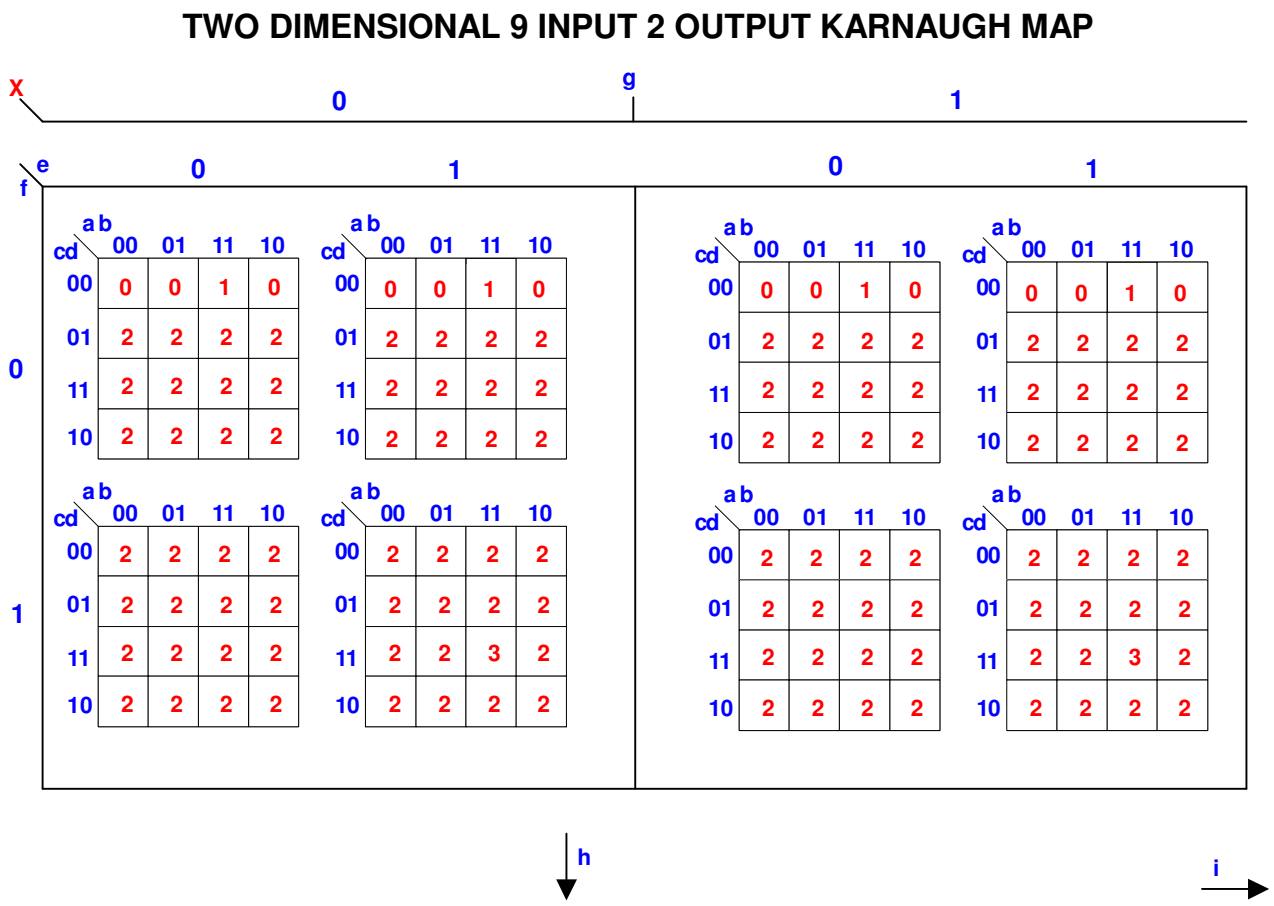


Figure 2 – Karnaugh Map Larger Than Display Area.

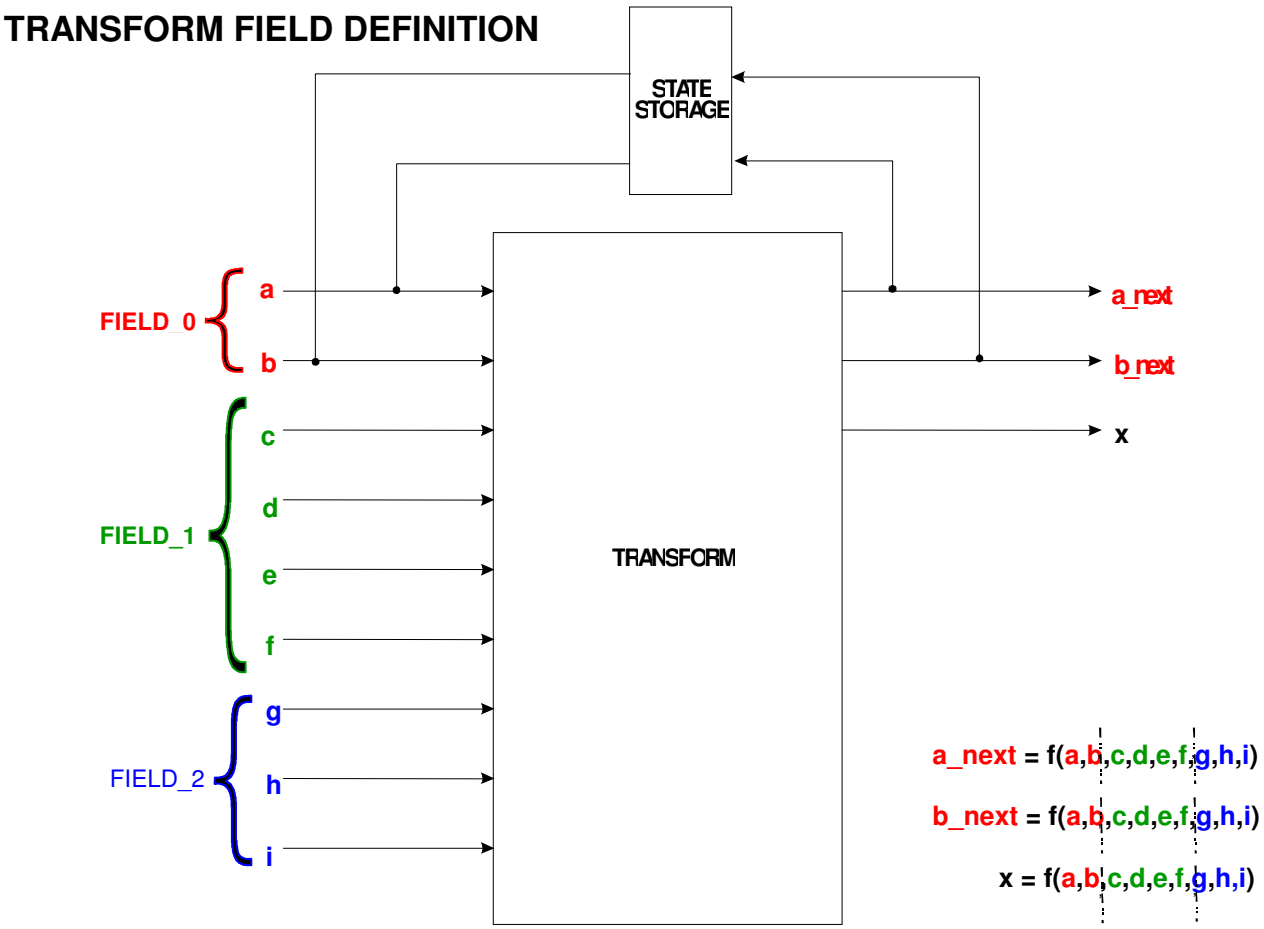


Figure 3 – Sequential Logic Transform and Field Definition.

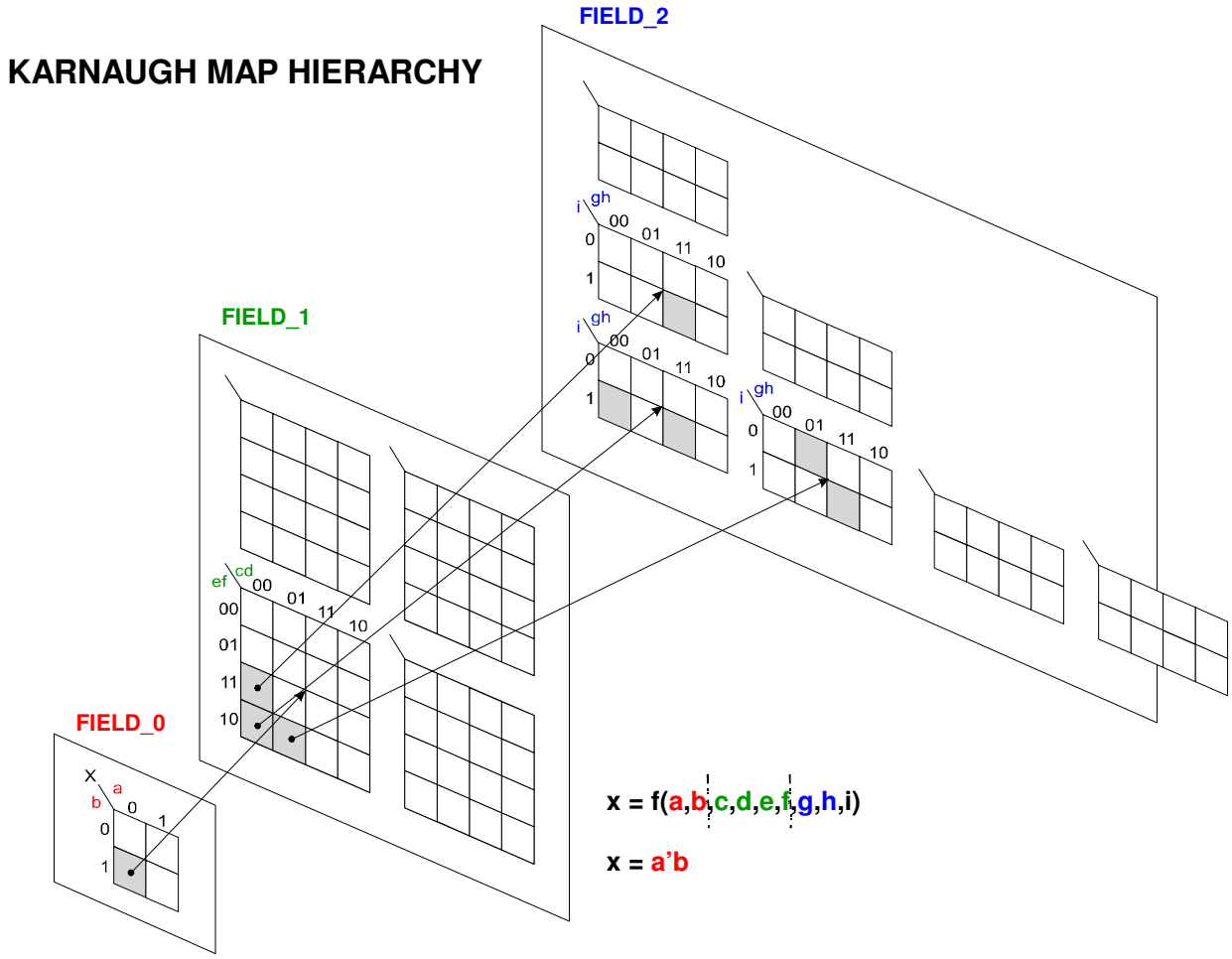


Figure 4 – Karnaugh Map Hierarchy with Example Functions.

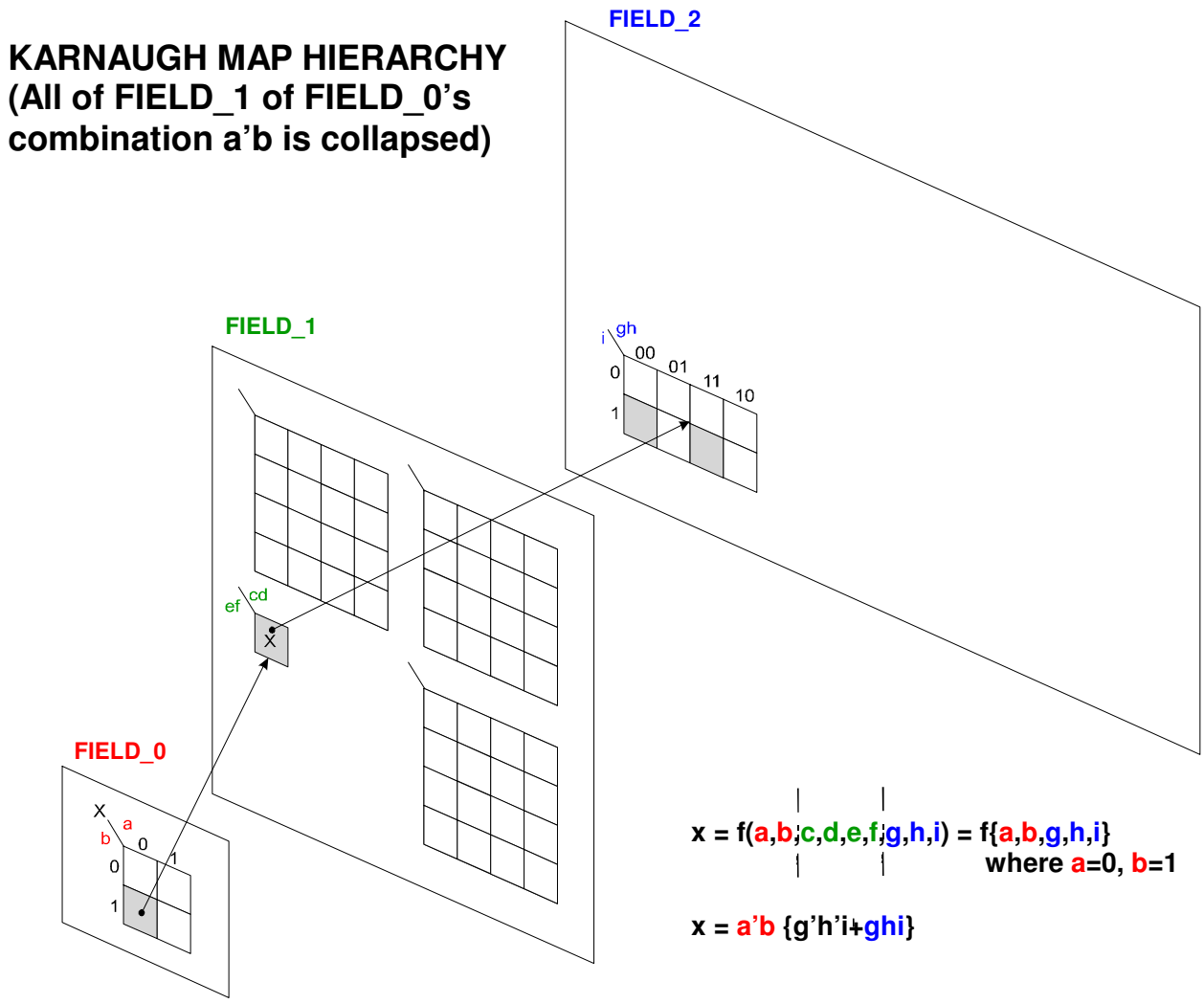


Figure 5 – Hierarchy Collapsed in FIELD_1 of a'b for All FIELD_1 Variables.

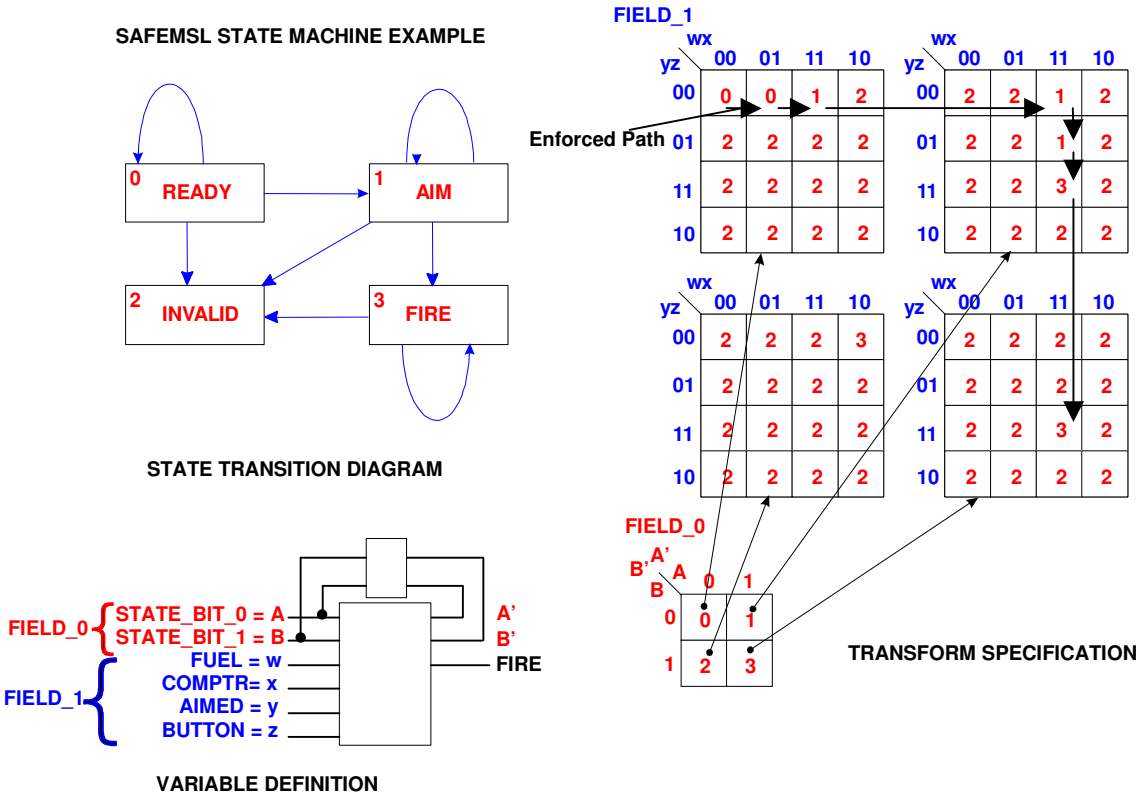


Figure 6 – SAFEMSL Example.

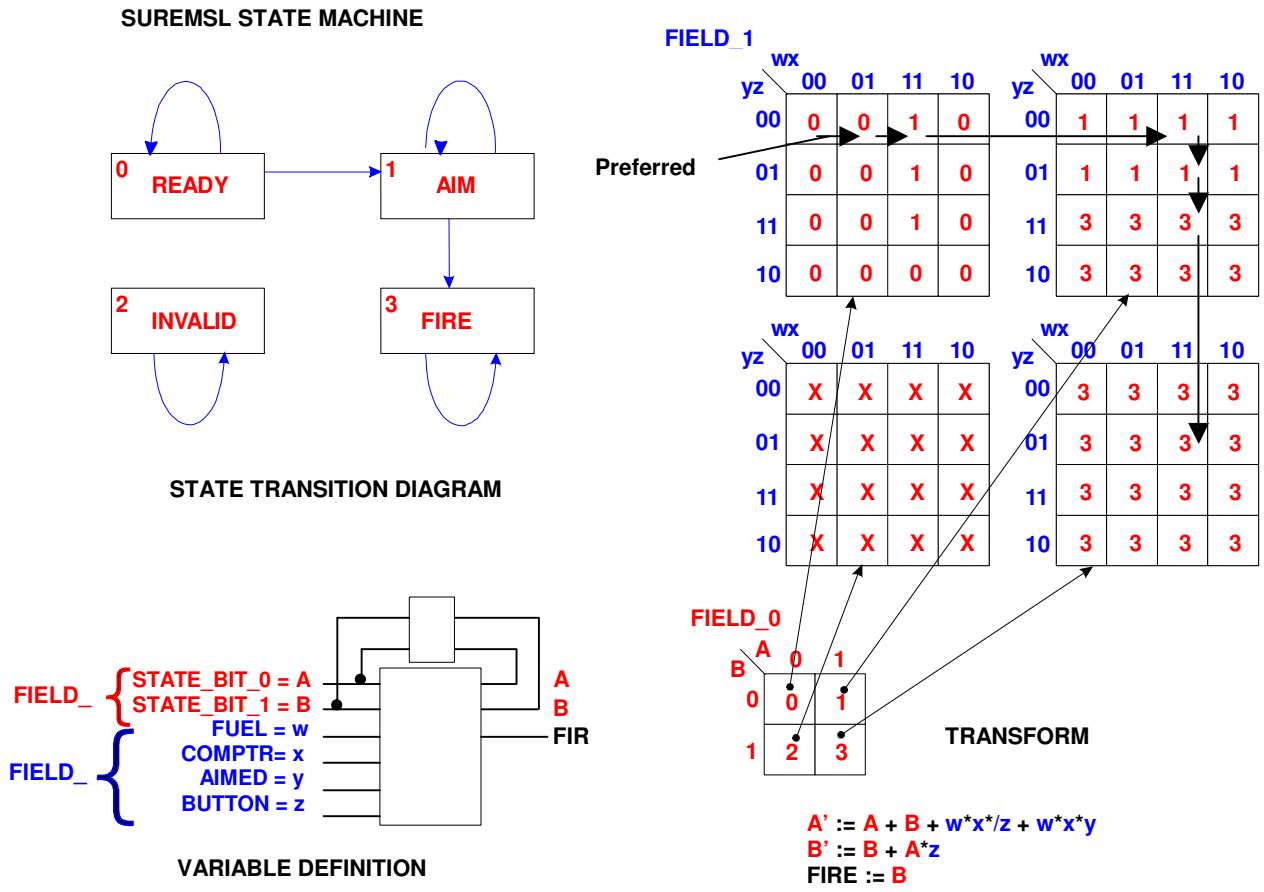


Figure 7 – SUREMSL Example.

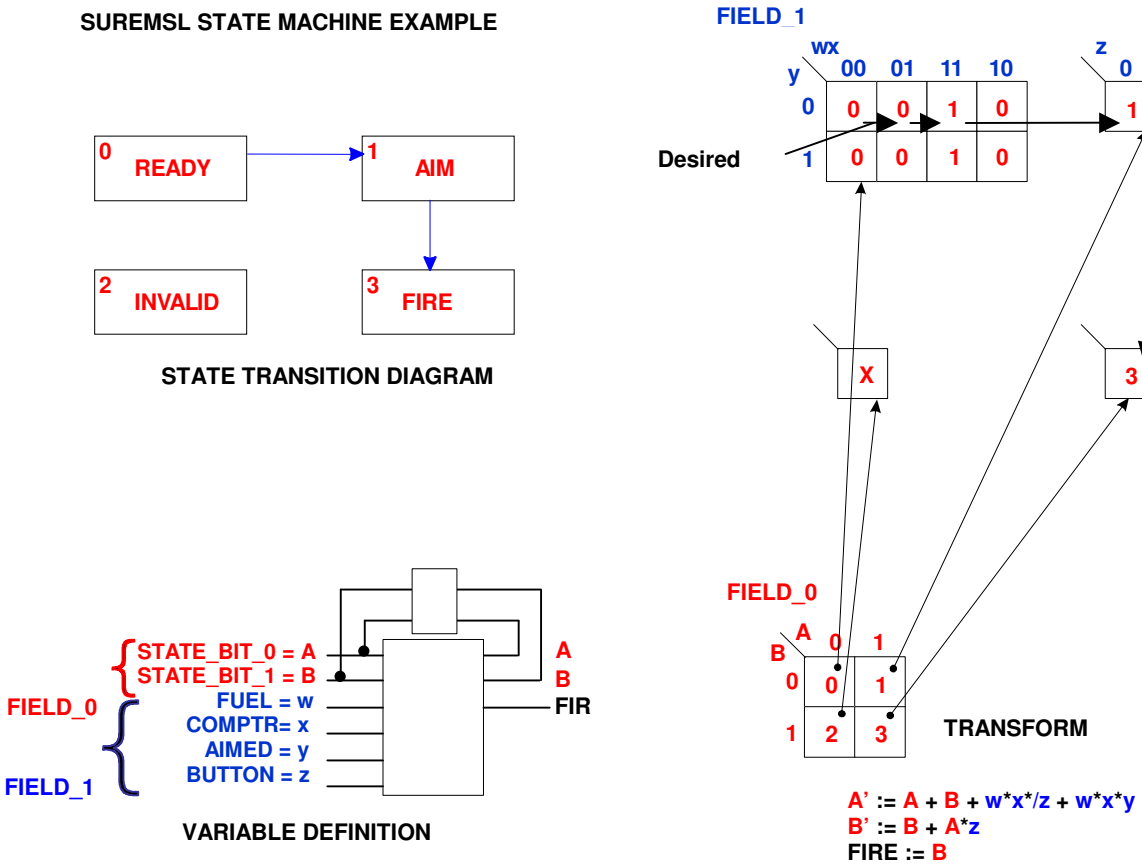


Figure 8 – SUREMSL Collapsed from 64 to 12 Entries via Don't Cares.

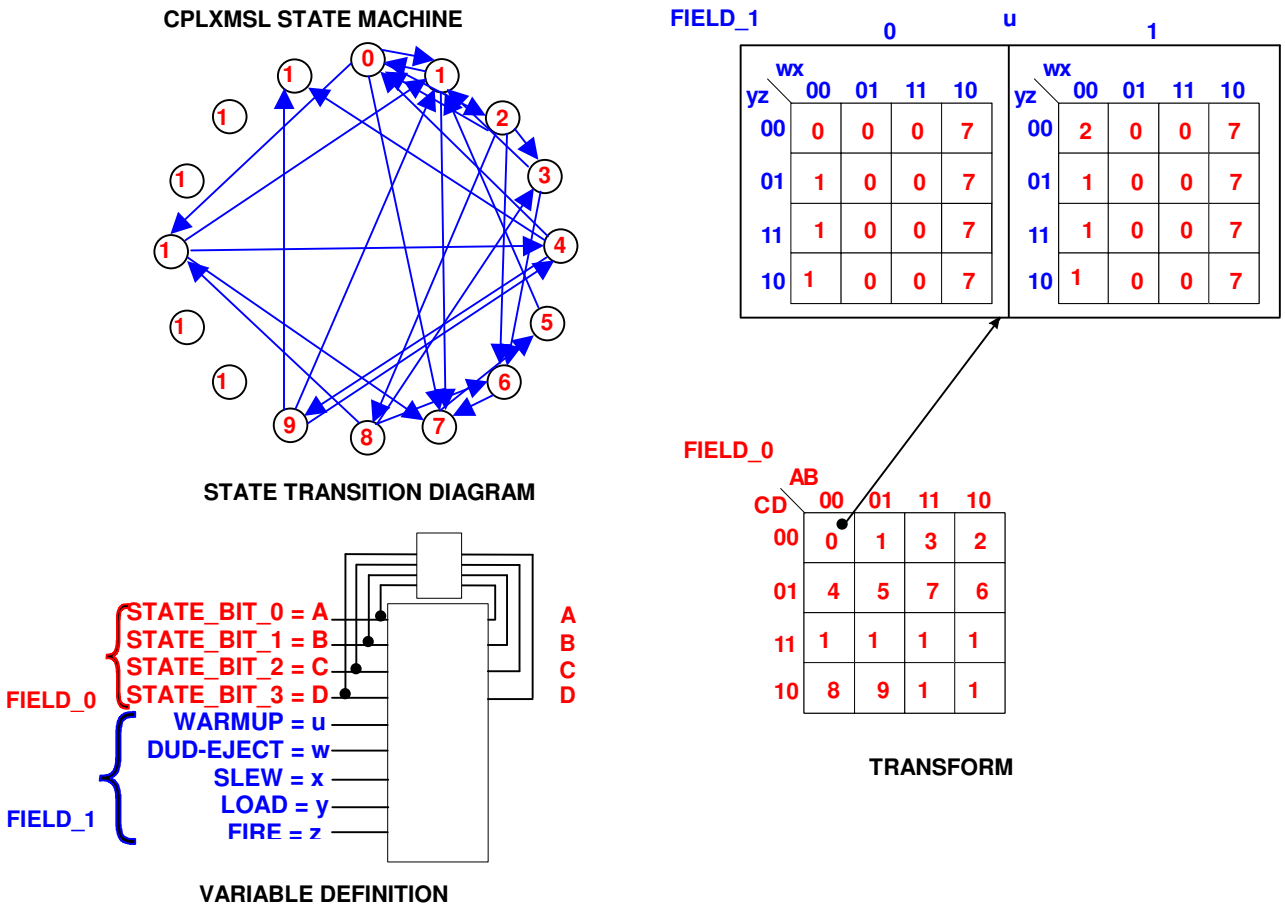


Figure 9 – CPLXMSL Example.

CPLXMSL STATE MACHINE EXAMPLE
Next State Analysis

Report of search for states which are
hanging (no states transition to this state) or
dead end (no states transition from this state).

State number 10 named State10 is hanging.
State number 10 named State10 is a dead end state.
State number 11 named State11 is hanging.
State number 11 named State11 is a dead end state.
State number 13 named State13 is hanging.
State number 13 named State13 is a dead end state.
State number 14 named State14 is hanging.
State number 14 named State14 is a dead end state.
State number 15 named State15 is a dead end state.

Figure 10 – CPLXMSL State Analysis Report.

CPLXMSL STATE MACHINE EXAMPLE
Next State Equation for Least Significant State Bit

```
A' := (
A and then B and then not D and then not u and then not v and then
not w and then not x and then not y )
 or else (A and then C and then not D and then not u and then not v
and then not w and then not x and then not y )
 or else (B and then C and then not D and then not u and then not v
and then not w and then not x and then not y )
 or else (A and then not B and then not C and then D and then not x )
 or else (A and then B and then C and then not u and then not v and
then not w and then not x and then not y )
 or else (A and then not B and then C and then not D )
 or else (A and then B and then C and then D )
 or else (not A and then not B and then not D and then not u and then
v )
 or else (not A and then not B and then C and then not u and then v )
 or else (not A and then not C and then not D and then not u and then
not v and then w )
 or else (B and then not C and then not D and then not u and then not
v and then w )
 or else (not B and then C and then not D and then not u and then w )
 or else (not B and then not C and then D and then not u and then not
v and then w )
 or else (not A and then not B and then not u and then not v and then
w )
 or else (A and then not B and then not C and then D and then u )
 or else (A and then not B and then not C and then D and then v )
 or else (A and then B and then not C and then not D and then not u
and then not v and then not x ) ;
```

Figure 11- Intractable Equation For CPLXMSL State Bit.

LDT CAPABILITIES NOT FOUND IN OTHER TOOLS

- **Specify logic that is complete and unambiguous without the use of equations.**

- **Represent both combinatorial and sequential logic.**

- **Find the worst and best case execution paths, accumulated time on given path.**

- **Generate exhaustive set of test vectors for all paths.**

- **Easily show a transition from many or all states to another single state.**

- **Display a large (greater than 16) number of states and transitions on a readable diagram without grouping substates.**

- **Make the software implemented state machine or combinational logic table driven so its behavior can be changed without recompilation of the code.**

Figure 12 – LDT Unique Functions.

USER SELECTIONS

Specification Entry Methods:

Single Entry, Output Bit Boolean, Transition Boolean, Default, Next State, Don't Care, Test Vector, Truth Table, espresso

Alternate Views:

Truth Table, If-Then-Else, Case, STD, Timing Diagrams, Boolean Equations, Karnaugh Map Patterns, Hierarchy

Analysis:

Interactive Debugger, Dead, Hanging, No Decision States; Worst, Best Case Execution Paths, Logic Reduction Report

Source Code Output:

C, Pascal, Ada, Assembly, VHDL, espresso, with test drivers and exhaustive test vector set

Transform Implementations:

Boolean Equation, Software Array, If-then-else, Case

Reverse Engineer:

VHDL, espresso

Figure 13 – LDT Options That Can Reduce Latent Errors.

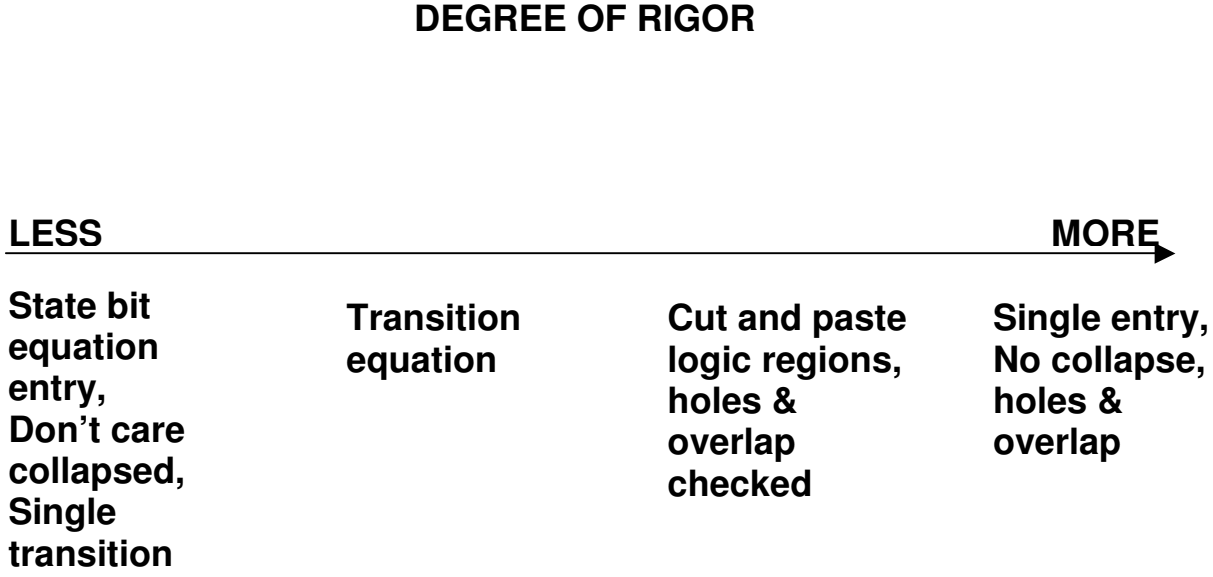


Figure 14 – Ease of Entry Versus Required Rigor

SAFEMSL NEXT STATE EQUATION WITH ERRONEOUS STATE 2 TO STATE 3 TRANSITION

```
State_bit_next(0) := (
not A and then B and then not w and then x and then not
y and then not z )
 or else (not B and then w and then x and then not y and
then not z )
 or else (A and then not B and then w and then x and
then not z )
 or else (A and then w and then x and then y and then z);

State_bit_next(1) := (
A and then not x )
 or else (B )
 or else (A and then not w )
 or else (not x and then y )
 or else (not w and then y )
 or else (not A and then y )
 or else (z );
```

Figure

Figure 15 – SAFEMSL Next State Bit Equation with INVALD (2) State Transition to FIRE (3).

SAFEMSL NEXT STATE EQUATION WITH NO ERROR

```
State_bit_next(0) := (
not B and then w and then x and then not y and then not
z )
 or else (A and then not B and then w and then x and
then not z )
 or else (A and then w and then x and then y and then z);

State_bit_next(1) := (
A and then not x )
 or else (B )
 or else (A and then not w )
 or else (not x and then y )
 or else (not w and then y )
 or else (not A and then y )
 or else (z) ;
```

Figure 16 – SAFEMSL Next State Bit Equation with No INVALD (2) Transition to FIRE (3).