BDSYN: LOGIC DESCRIPTION TRANSLATOR
BDSIM: SWITCH-LEVEL SIMULATOR

by

Russell B. Segal

Memorandum No. UCB/ERL M87/33

21 May 1987

COVER PAGE

BDSYN: LOGIC DESCRIPTION TRANSLATOR
BDSIM: SWITCH-LEVEL SIMULATOR

by

Russell B. Segal

Memorandum No. UCB/ERL M87/33

21 May 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

BDSYN: LOGIC DESCRIPTION TRANSLATOR
BDSIM: SWITCH-LEVEL SIMULATOR

by

Russell B. Segal

Memorandum No. UCB/ERL M87/33

21 May 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

The two programs discussed in this report, BDSYN and BDSIM, are front end tools that are intended to aid a chip designer in creating new systems. BDSYN is a tool which takes functional description of combinational logic, and create appropriate logic equations to implement the described logic. BDSIM is a zero-one switch-level simulator which provides the facility for a designer to quickly verify the functionality of or locate errors in a digital circuit.

Acknowledgments

I would like to thank Richard Rudell for his help and advice on almost every project which I worked. Rick was responsible for impementing a large portion of BDSYN including the parser, inline routine expansion, and macro expansion. In addition, he supplied many of the algorithms for implementing the rest of BDSYN and provided direction for the whole project.

I would like to thank Albert Wang for his work on MIS on which BDSYN relies heavily. In addition, the BDSIM front end is based on Albert's MIS code. Also thanks to Peter Moore for patiently answering my never-ending stream of unix and 'C' questions.

I appreciate the work of all the students and industrial visitors who participated in the Spring 1986 synthesis class, who provided extensive testing for BDSYN. I also appreciate the efforts of Daebum Lee, Wook Koh, D. K. Jeong, David Wood, and Shau-Lim Chow for providing testing and many useful suggestions for BDSIM.

Finally, I would like to thank my advisor, Alberto Sangiovanni-Vincentelli, and Richard Newton for organizing the synthesis project and providing the framework in which I could work on such interesting projects.

I acknowledge funding for this research under NSF grant UCol-PO-454629.

Contents

Overview

The two programs discussed in this report, BDSYN and BDSIM, are front end tools that are intended to aid a chip designer in creating new systems. BDSYN is a tool which takes functional description of combinational logic, and create appropriate logic equations to implement the described logic. BDSIM is a zero-one switch-level simulator which provides the facility for a designer to quickly verify the functionality of or locate errors in a digital circuit.

The major portion of this report is devoted to the discussion of bdsyn. The documentation for BDSYN is split into two distinct parts. Appendix A of this paper contains the "Bdsyn Users' Guide." This guide is intended to be a self-contained document introducing the BDSYN program and its input format. Chapters 1 and 2 of this report deal with more in-depth details of BDSYN and its purpose. Chapter 1 addresses issues pertaining to the usefulness of BDSYN and how it compares to similar and related programs. Chapter 2 gives a detailed description of the algorithms that BDSYN uses. Both chapters assume that the reader has a working knowledge of BDSYN and its constructs. Readers who are unfamiliar with BDSYN should skim appendix A before reading chapters 1 and 2.

Chapter 3 of this paper is devoted to the discussion of BDSIM. Included is a description of the data structures, simulation algorithms, and optimizations performed by BDSIM. The user commands and directions for using BDSIM are documented in appendix F.

Chapter 1

Philosophy of Bdsyn

BDSYN is a translator which will generate a multiple level logic network from a user supplied functional description of combinational logic. Much effort has been invested to make BDSYN both flexible and powerful. BDSYN's input logic description language is a subset of the BDS simulation language.[1] The logic descriptions that are written in the BDS subset, resemble traditional procedural programming languages. Writing a logic description is similar to writing a software program, and, as in programming software, there are many ways to implement the same function.

BDSYN relies on the logic optimizer MIS to provide a good implementation for the multiple level logic it extracts. BDSYN and MIS combine to provide the logic designer the freedom to describe logic in a fashion which is suggestive of its intended function, rather than worrying about the details of a gate-level implementation. The implementation details are handled automatically.

1.1 Combinational versus Sequential Logic

One of the primary difficulties in interpreting functional hardware descriptions is discerning what sections of the description imply pure combinational logic, and which parts are intended to be sequential logic. Procedural programming languages

[1]BDS is part of Digital Equipment Corporation's proprietary multi-level simulator DECSIM.

are basically sequential in nature, and a typical software program executes one statement at a time. Such a language is not well suited to describing both combinational and sequential logic.

In a typical functional hardware description, any particular variable may be used for one of two very different purposes. It may be intended to hold the state of some logic variable, implying a latch or memory cell. Or it may be an intermediate variable in some larger combinational network. In the latter case, the variable represents a wire, and no physical gates are implied. One approach to solving this "combination versus sequential logic" problem has been to label explicitly which variables imply latches [11]. This is a reasonable solution to the problem, but does not yet address how a translator would interpret general flow-of-control statements (WHILE, FOR, GOTO, etc.) and their interaction with the latch variables.

BDSYN restricts the general problem of hardware description translation to dealing with the processing of combinational logic descriptions only. BDSYN requires that latches and other synchronizing circuit elements be integrated into a design by the logic designer. This task is not difficult. The designer need only combine latches and combinational logic using a netlist entry mechanism of some kind. This may be a graphical schematic entry system built on top of VEM [8] or a textual netlist entry program like bdnet.[2] Though this method of latch entry is not as convenient as simply declaring "latch variables," it accomplishes the exact same task. It also allows much greater flexibility in specifying the flow-of-control among the combinational modules.

There are significant advantages to restricting input descriptions to combinational logic. It greatly simplifies the problem of logic translation since there is no ambiguity in determining what is combinational. More importantly, however, is that it allows BDSYN to assign very specific meanings to procedural language constructs. A FOR loop, for example, could easily be interpreted in two different ways. In the

[2]See bdnet manual page in appendix F

sequential sense, a FOR loop could signify that a certain piece of logic is to evaluate repeatedly while a counter circuit counts from 1 to 10. In the combinational sense, the same FOR loop signifies ten identical pieces of logic that are connected together. BDSYN will always interpret a FOR as signifying combinational logic.

1.2 Gate-Level versus Behavioral Descriptions

Logic description formats can be lumped into two broad categories which I call gate-level descriptions and behavioral descriptions. Gate level descriptions are closer to the actual hardware implementation and are generally meant to serve as a very accurate model of the real hardware. Functional descriptions look more like software programs and are generally targeted for efficient simulation.

Systems which use gate-level descriptions of hardware usually give the logic designer a set of hardware primitives out of which to build his system [1] [3]. These primatives range in type from MOSFET's, to logic gates and register cells. The primatives are either selected out of a fixed library and/or described with a set a simple constructs. The designer creates systems by building up a hierarchy of cells. This is done by specifying a connection of lower level cells at each level of the hierarchy. For example, a one bit ALU might be composed of several logic gates, a datapath bit slice might be composed of the ALU bit and some registers, and a full data path could be composed of several bit slices. Notice that this way of describing logic closely parallels the actual implementation of a design.

Systems which take behavioral descriptions as input generally do not use physical cells as primitives [10] [12]. They instead rely on traditional computer language constructs, with a few constructs added for sequencing and synchronization. Designs are usually defined in terms of their intended function, rather than their construction. A "program" which performs the same function as the planned hardware is written for each main module of a design. The modules are then linked together. This method of describing logic more closely parallels the actual idea and function of the design. It is, however, farther from the actual hardware implementation.

It is becoming increasingly popular to integrate both gate-level descriptions and behavior descriptions together [2]. For a designer, this gives the best of both worlds. Early revisions of a design can be described behaviorally. Simulation can then be done on this description to verify architectural soundness of a design. Later, as an implementation of the design is fleshed out, parts of the behavioral description may be replaced by gate-level (or even transistor-level) descriptions. The new description can then be resimulated to check for consistency.

BDSYN, along with the MIS logic optimizer (discussed below), begins to bridge the gap between behavioral hardware descriptions and gate-level descriptions. BDSYN eliminates the traditionally manual task of translating behavioral descriptions to gate-level descriptions. Within the realm of combinational logic, BDSYN is quite general. Any logical behavior that can be described using BDSYN constructs, can be mapped to logic. MIS then insures that a good implementation of the logic is generated. This allows the designer the freedom to describe and simulate logic at a very high level, without worrying about implementation details. The gate-level description of the circuit is generated automatically by BDSYN and optimized by MIS. There should be no need to describe directly combinational logic at the gate level.

1.3 Two Level versus Multiple Level Logic

Today in many design VLSI design centers, the only logic implemented from a high level language is PLA's. PLA's implement what is called "two level logic," which is characterized by two stages of logic gates. Two level logic is quite useful in many common applications such as finite state machines, and PLA's are commonly used in many VLSI applications. However, two level logic and PLA implementations are not useful in all applications.

"Multiple level logic," is characterized by many stages of cascaded logic gates.

Multiple level logic is often implemented in using a standard cell technology, where a set of pre-created gates are placed and routed together, or a gate array technology, where gates are patterned onto a partially fabricated chip and routed together. Module generators for multiple level logic are also becoming popular. These systems, like PLA generators, generally rely on placing and routing transistors in a regular array.

Implementing logic in multiple level form has several advantages. By manipulating multiple level logic, one can optimize the logic for minimum delay or minimum area. As the complexity of logic increases, PLA implementations suffer the problem of declining performance *and* increasing area requirements. This is not necessarily the case for multiple level implementations, where tradeoffs between speed and area can be made. Another important consideration is that certain multiple level circuits do not map well into PLA structures. The process of collapsing multiple level logic into two levels may sometimes explode. For example, a PLA implementing an n bit adder circuit requires $O(2^n)$ product terms. This suggests that multiple level logic structures are preferable to PLA's in certain applications.

A typical PLA description may be specified in terms of logic equations [7]. This can be extended to further by allowing logic to be described in terms of IF and SELECT statements [17]. The conditions of the IF and SELECT statements are taken from among the inputs of the PLA, and the outputs of the PLA are set by assignment statements within the IF and SELECT structures. This set of constructs allows the designer to choose a value for each PLA output based directly on some combination of input values.

BDSYN can be used for describing PLA's as easily as for describing any other circuit. However, BDSYN allows for the description of multiple level logic. This is accomplished by allowing the general use of "intermediate variables," which can be thought of as intermediate results in a multiple level logic network. Intermediate variables are defined in terms of inputs and other intermediate variables. They, in turn, are used to define outputs. BDSYN will generate, as output, a multiple level gate description called BLIF.[3]

Most logic translators, YLL [6] being a notable exception, do not allow the use of intermediate variables in their input logic description. The reason for this can be found in the nature of PLA's. There appears to be little use in describing logic being mapped to a two level PLA structure in multiple level form. In reality, multiple level descriptions are quite useful for describing PLA's, as well as for multiple level implementations.

A major reason that PLA description languages have existed for some time is the availability of PLA logic minimizers. PLA minimizers such as ESPRESSO [14] [13] [5] make it possible to create PLA's that are as good or better than human designs. Since ESPRESSO can be trusted to implement a "good" PLA, the designer is able to think in terms of circuit function rather than worrying about circuit implementation. This paved the way for PLA description languages.

MIS [4] [16] is a multiple level logic minimizer written at Berkeley. MIS preserves the multiple level structure of logic during minimization, allowing it to avoid the problem of exponential "blow-up" associated with the collapsing to two level logic. It takes as input a multiple level gate description of a logic circuit in BLIF format and writes its output back to BLIF or into the OCT database [8]. It is important to note that if either BDSYN or MIS required logic to be represented in two level form, many logic circuits could not be processed.

BDSYN relies heavily on MIS to optimize the gate descriptions it creates. The quality of BDSYN's output depends heavily on the user supplied input description. Using MIS removes much of this dependence on user input, giving the designer far more flexibility in the way he describes his design. Without MIS to minimize multiple level logic, designers would essentially have to describe circuits at the gate-level in order to assure quality results, and BDSYN would be of very limited utility.

[3] A specification for BLIF is given in appendix D.

1.4 Software versus Hardware

BDSYN is unusual in that it draws an association between a software "program" and a hardware "schematic." The BDS language, upon which BDSYN input is based, is a programming language. These programs are intended to be compiled and run in a simulation environment. The BLIF output of BDSYN is a hardware description. It represents a group of gates connected together in a combinational network. The ways in which software and hardware behave are different, but both may imply similar functions. BDSYN's job is to create hardware that functions similarly to a software program.

One of the major differences between simulating a hardware description (running a program) and synthesizing gates is that a simulator is given values for the primary inputs to the network, while a logic synthesizer is not. A simulator can sequence through the given commands of a description and determine values for the intermediate variables based on the given inputs. When a branching statement such as an IF or SELECT is encountered, the simulator can evaluate the branching condition and act accordingly. When synthesizing logic from a description, the network inputs are not known. BDSYN has no way of determining the value of intermediate variables. More importantly, there is no way to know in advance which of the THEN or ELSE clause of an IF statement will apply.

BDSYN must accomplish "static evaluation" of the input description. Any possible path through the flow-of-control structures of the description must be represented by corresponding logic implementation. BDSYN must create hardware that covers all possible combinations of inputs. In a simulation, one of the THEN or ELSE clause of an IF statement need not be evaluated. BDSYN must consider both in order to produce logic that will be correct for any input. Static evaluation goes beyond just considering both the THEN and ELSE clauses. Correct static evaluation is even more difficult when processing the LEAVE statement. A LEAVE which is contained within an IF statement, implements a conditional branch. Care must be taken to insure that all possible sets of "branches taken" are covered in the hardware implementation.

There are certain software structures which can not be statically evaluated. Consider the general FOR loop. To do a correct static evaluation of FOR loops BDSYN chooses to duplicate the body of the loop an appropriate number of times. This is only possible if the extent of the FOR loop is bounded by constants, allowing a calculation of how many copies to make of the body of the loop. If the extent of a FOR loop is bounded by a variable which depends on the input to the logic network, BDSYN can not calculate, statically, a tight bound on how many copies should be created. For this reason, BDSYN only allows constant bounded FOR loops.

There are other structures which are not susceptible to static evaluation. These include WHILE loops, variable width bit selects, and recursive routine calls. In each case, the number of loops, bits, or depth of recursion may depend on a variable which can not be statically evaluated. Since BDSYN can not calculate, in general, how much logic is implied by these constructs, it does not support them. For similar reasons, the GOTO statement also can not be supported.

To say that there is no way for BDSYN to handle WHILE loops is a slight exaggeration. BDSYN could handle this construct if there were some declaration that told BDSYN the maximum number of times a while loop would execute. This way, BDSYN could create hardware that for the maximum number of WHILE iterations just as it does in FOR loops. If the conditions were such that fewer than the maximum number of loops should be executed, it could be handled in the same manner as LEAVE statements in FOR loops are handled. Recursive subroutine calls might also be handled if there were some constant bound on the depth of the recursion.

There is a side issue here which should be mentioned. Constructs such as WHILE and GOTO, while less useful for describing combinational logic, are potentially useful for describing sequential logic. The use of these statements could correspond directly to the flow-of-control among active combinational logic blocks in a sequential system. A WHILE loop would imply that the logic within the body of the loop should produce valid outputs for as many clocks as the while loop's condition is true. A GOTO implies that a different combinational logic block will be active in the next clock cycle. "Static evaluation" does not apply to sequential logic as it does for combinational logic. In a sequential circuit each combinational block evaluates on every clock phase. Intermediate *sequential* variables (latched signals) are continuously recalculated, and flow-of-control decisions can be made based on their value.

Chapter 2

Processing Steps in Bdsyn

This chapter covers the processing steps that BDSYN takes to reduce an input logic description to the corresponding logic equations. BDSYN is set up in several discrete phases of operation. All of the phases of operation manipulate the same simple parse tree data structure. The sequencing of BDSYN's major processing steps is summarized in figure 2.1. An example of the processing of an actual sample input file is given in appendix B.

The major phases are:

Parse Each routine of the input file is translated to an internal parse tree. Macro definitions are expanded, and symbol tables are generated.

Inline Subroutines and main routines are identified. The subroutines are then copied and inserted into the main routines, and parameter passing is handled.

Meta The value of all meta-variables are statically evaluated and the meta-variables are replaced by constants. As part of this processing, FOR loops are unrolled. That is, the body of the loop is duplicated the necessary number of times.

Complex Operators which are considered too complex for simple processing are replaced with calls to BDSYN library routines. The flow of control may loop back to Inline at this point.

```
/* BDSYN -- main routine
        "T" refers to a set of parse trees
        "tree" refers to the parse tree for a single routine
        "T'" refers to a tree for a single main routine plus trees
             for selected library subroutines.
*/

bdsyn(input, output)
{
    T = PARSE(input);           /* parse all routines in input */
    T = INLINE(T);              /* inline expand all subroutines *
                                 * leaving only main routines    */

    foreach tree in T {
        do {
            /* process meta-variables and FOR loops */
            tree = META(tree);

            /* get library routines for complex operators */
            T' = COMPLEX(tree);

            if (complex inserted a routine) {
                tree = INLINE(T');   /* expand library routines */
            }
        } while (complex inserted a routine);

        tree = EVAL(tree);       /* evaluate simple expressions */
        tree = LEAVE(tree);      /* process LEAVE statements */
        tree = PREMULT(tree);    /* prepare for multiple assignment */
        tree = MULT(tree);       /* Do multiple assignment */
        tree = CLEANUP(tree);    /* evaluate using info from MULT */
        output = LIFIFY(tree);   /* convert to output format */
    }
```

Figure 2.1: Sequencing of Bdsyn's Major Phases

Eval BDSYN attempts to evaluate and simplify expressions in the parse tree.

Leave BDSYN processes all LEAVE instructions, replacing them with one or more equivalent IF statements.

Premult This preprocessing step to multiple assignment handles three tasks. The nesting structure of IF and SELECT statements is stored in a simple data structure and each "clause" of each statement is assigned a unique number. Complicated condition expressions in IF and SELECT statements are assigned to temporary BDSYN variables as an optimization. The last use of each variable in the parse tree is identified and marked.

Mult This is the multiple assignment step. It performs two interrelated functions. It makes sequential assignments to a same variable possible by assigning a unique "revision number" for each assignment to that variable. The multiple assignment step also reduces IF and SELECT statements to a series of assignment statements.

Cleanup This repeats the earlier Eval phase using information provided by multiple assignment. This eliminates much of the meaningless logic that may be created from a redundant or loosely specified initial description.

Lifify This coverts the contents of the parse tree to BLIF output. As part of this procedure, MIS may be invoked to collapse the logic of many related BLIF tables into one. This is used to decrease the size of the output.

2.1 Parsing

The first phase of BDSYN processing is, of course, the parsing of the input description. This is accomplished through the use of a YACC [9] grammar and a custom lexical analyzer. The lexical analyzer handles much of the input processing, before passing tokens to the YACC generated program. The lexical analyzer detects where separate files should be inserted into the main text includes those files. It also expands any macro definitions in the input file. Both REQUIRE files and the macro definitions may be nested.

The actual grammar to parse the input was written in YACC. The YACC grammar was written with the objective of translating our subset of the BDS description language, while properly handling (ignoring) the BDS constructs that are not supported. While the input file is being parsed, several data structures are built. First, a symbol table is constructed that contains global variable, input and output port, global constant, and global synonym declarations. A symbol table for each routine of the input file is also built. It contains locally declared constructs, as well as routine parameter variables and statement label declarations.

The YACC grammar then produces a parse tree for each routine in the input. The nodes of the parse tree are of one of two major groups. The first, expression nodes, are for manipulating and representing data. These include boolean and arithmetic operations, shifts, comparisons, bit select operations, routine calls, bit vectors, and constants. The second major group of nodes in the parse tree are statement nodes. These include IF, SELECT, FOR, LEAVE, RETURN, and assignment statements. IF, SELECT, and FOR nodes point to linked lists of statement nodes which represent the clauses (e.g. THEN or ELSE) or loop body associated with the statement. Expression nodes may not contain statement nodes.

2.2 Inline Routine Expansion

The second processing step in BDSYN is the inline routine expansion. For each call to a subroutine, a copy of that subroutine's parse tree is inserted into the the middle of the calling routine's parse tree. The local variables of the subroutine are copied into the global symbol table, and their names are given a unique prefix in order to avoid name clashes with the global variables.

Passing of routine parameters and return values is handled by creating assignment statements before and after the inserted routine, respectively. RETURN statements are replaced with LEAVE statements which implement jumping out of routine blocks. Once a subroutine has been expanded, inline is then called recursively in order to handle nested subroutines. Recursive routine calls in the BDSYN input description are disallowed.

The BDS language has no concept of a single main routine. In fact, many routines in a BDS input file may be main routines which can be thought of as running in parallel. This presents a problem for BDSYN. BDSYN must decide which routines in the input file are main routines and which are subroutines, since only main routines should be directly represented in the output logic equations. The main routines are identified by being those which are not called by other routines.

As a result of the inline expansion step, many copies of the same routine may be created. This duplication, while not absolutely necessary, is useful due to the nature of the problem BDSYN is addressing. When hardware is being described, every subroutine call corresponds directly to a physical piece of logic. So, in order to represent that logic, BDSYN copies a parse tree for each subroutine call.

2.3 Meta-variable and FOR Loop Processing

A major reason for the existence of meta-variables in BDSYN is their property of being able to be statically evaluated. By definition, meta-variables in BDSYN may not depend on any logic variable. This means that the value of a meta-variable will always be the same regardless of the physical inputs to the combinational logic network. It is a fairly easy task for BDSYN to calculate the meta-variable values, and that is a major purpose of this stage of processing.

The processing of meta-variables is done in one sequential pass over the parse tree. Whenever an assignment to a meta-variable is encountered, BDSYN calls its expression evaluation routine to reduce the right hand side of that assignment. Since the right hand side must evaluate to a constant (otherwise an error is generated), the constant value of the meta-variable can be stored in the symbol table. From that point on, any use of the meta-variable is replaced with the stored constant.

A potential problem exists in the meta-variable replacement. The stored value of a meta-variable is overwritten whenever a second assignment to that meta-variable is encountered. If the same meta-variable is assigned to in both clauses of an IF statement, its final value will always be the value of the last assignment. For example, in the following:

```
IF cond THEN
    meta = 1
ELSE
    meta = 2;
```

meta will have the value 2 upon exit from the IF statement, regardless of the value of *cond*. This is because the assignment `meta = 2;` comes last in sequential order. Unfortunately, if *cond* is not a meta-variable, there is no way to determine the correct way to handle the value of *meta*. *meta* can not be statically evaluated.

If *cond* in the above example is a meta-variable, the situation can be handled in a consistent fashion. A meta-variable or constant appearing in an IF or SELECT condition implies that one or more clauses of that structure can never be executed. For this reason, BDSYN calls its evaluation routine to reduce all IF and SELECT conditions. If the condition can be evaluated statically, without using the value of any logic variable, the clause or clauses which are known to never occur are deleted. In the above example, the ELSE clause would be eliminated if *cond* was a meta-variable with value *1*.

FOR loop processing is closely tied to meta-variable processing. The extent of the loop (the FROM, TO, and BY expressions) are evaluated. Since the extents may only depend on meta-variables and constants, a constant result can be found. Once the extents are calculated, an appropriate number of copies of the loop's body are generated. For each copy of the FOR loop body, any use of the loop index variable is replaced with the appropriate constant value, as shown:

```
FOR i FROM 0 TO 3 DO
    y<i> = x<3 - i>;
```

becomes

```
y<0> = x<3 - 0>;
y<1> = x<3 - 1>;
y<2> = x<3 - 2>;
y<3> = x<3 - 3>;
```

The actual replacement of the index variable is accomplished using the same code as for the regular meta-variable replacement.

2.4 Complex Operator Processing

BDSYN is capable of handling a variety of complex operators. Complex operators are defined as those operators for which it is not "easy" to generate logic. They are as follows: addition, subtraction, and multiplication of logic variables; greater than or less than comparison of logic variables; shifts by a variable amount; and selection of a variable bit from a bit vector.[1] Rather than hard-wiring these rather messy functions into BDSYN, a different approach was taken. BDSYN relies on a small set of library macro definitions to support the complex operators.

During this stage of processing, BDSYN scans the parse tree for any use of a complex operation. When one of these operators is found in the parse tree data structure, it is replaced by a node which implements a BDSYN subroutine call. The actual BDSYN subroutine is then created using one of the library macro definitions. Once all uses of complex operations have been replaced with subroutine calls, BDSYN loops back to the inline expansion phase of processing to expand the subroutines. The looping continues until the inserted subroutines contain no complex operators.

The actual creation of complex operator subroutines deserves a comment. The BDSYN library contains only macro definitions for subroutines. These definitions differ from actual subroutines in that the width of the routine parameters are not fixed. They are, instead, defined as sub-macro definitions. By defining these sub-macro definitions, BDSYN can create custom subroutines of the any size. When

[1]Notice: These operators imply adders, multipliers, comparators, shifters, and multiplexors.

BDSYN wishes to create a 4 bit adder, for example, it defines a sub-macro definition to be 4, and then instantiates the subroutine macro into a subroutine with that width. The BDSYN library, which is shown in appendix C, lists the default set of sub-macro definitions.

2.5 Expression Evaluation

Expression evaluation in BDSYN is used to reduce trees of boolean, arithmetic, and bit vector operations to the simplest form possible. Expressions are evaluated in a recursive fashion. For any operator being processed, its operands are first evaluated, then the operator is applied to those operands.

There are two main classes of evaluation that can occur. If all of the operands of a particular operator are determined to be constant valued, then the given operation is performed using basic 'C' constructs. This is direct to accomplish since BDSYN constants are stored as 'C' unsigned integers. Unfortunately, constants of greater than 32 bits can not be accommodated using 'C' integers. Long BDSYN constants must be represented as a concatenation of shorter constants. Long constants in BDSYN are treated in the same manner as variable arguments.

The other main class of evaluation that can occur is simplification of functions with variable arguments. Because of the complex operator processing step, it is guaranteed that there will be only "simple" operators to process. These include boolean operations, equal and not equal comparison, shifts by a constant amount, bit extraction of constant extend fields, and extensions (e.g. sign extend). The boolean operators are applied bitwise to their operands. BDSYN will reduce simple boolean identities such as "`NOT(NOT(a))`," "`x AND 0`," "`y XOR 1`," etc. The equal and not equal comparisons are reduced to the equivalent boolean operations. The extension, bit extraction, and shifts simply involve reshaping of bit vectors.

Upon completion of the expression evaluation, most of the operator types have been simplified out. The only operators remaining are AND, OR, NOT, NAND, NOR, EQV, and XOR. In addition, BDSYN creates two internal operations ANDR, and ORR which are multiple operand versions of AND, and OR. ANDR, and ORR are created when reducing multiple bit equal and not equal comparisons.

Expression evaluation serves one other purpose. It implements the assignment rules of BDS. If the width of the right hand side of an assignment exceeds that of the left, the high order bits are truncated. If the width of the left hand side exceeds that of the right, zeros are added to pad the right hand side.

2.6 LEAVE Processing

LEAVE processing deals with the reduction of trees containing LEAVE statements to the equivalent tree using only IF statements. The algorithm to process LEAVE statements is not as straight forward as the other BDSYN algorithms. For this reason, I will be somewhat formal in its definition. I will also step through the processing of a sample piece of code, which follows:

```
A: BEGIN                           ! start block A
     IF x THEN
          B: BEGIN                      ! start block B
               c = 1;
               IF y THEN
                    LEAVE B;
               d = 1;
               IF z THEN
                    LEAVE A;
               e = 1;
          END;
          f = 1;
     END;                               ! end block B
     g = 1;
END;                               ! end block A
```

In the following discussion I will use the concept of a "level." Statements in a parse tree can be grouped into levels corresponding to their depth in that tree.[2] For instance, in the above example the following statements occur at the same level:

```
          c = 1;
          IF x THEN ...
```

[2]This is equivalent to the level of indentation in nicely formated code.

```
        d = 1;
        IF y THEN ...
        e = 1;
```

The "LEAVE B" and "LEAVE A" statements that were originally in this section of code are omitted because they occur in the THEN clause of an IF statement and therefore occur at a lower level. I will call a lower level, that is contained by a given level, a "sub-level."

The LEAVE statement in BDS is followed by one argument which is the name of the labeled statement out of which it is supposed to break. The block of code that is contained by the labeled statement is the "leave block" corresponding to a particular LEAVE statement. When conditions exist that cause a LEAVE statement to take affect, any code following that statement in its leave block is skipped.

The problem is to create IF statements which cause the correct code to be skipped when a LEAVE is in affect. As a convenience in creating IF statements, BDSYN introduces a "leave condition" in the LEAVE statements. If the condition is true, the LEAVE takes affect. If the condition is false the LEAVE is ignored. By this definition "LEAVE (x) A" is equivalent to "IF x THEN LEAVE A."

The LEAVE processing is done in a bottom up fashion, starting at the lowest levels of the parse tree and working up. The basic idea is to "push" the LEAVE statements progressive upward through the levels, until they reach the level of their corresponding leave block. Each time a LEAVE is pushed up, an IF statement may be generated which implements the potential skipping of code.

At any particular level of the parse tree, processing occurs in two phases. In the first phase, all sub-levels are processed recursively. The result of this will be that LEAVE statements from the sub-levels will be pushed up to the current level. When a LEAVE statement is pushed up a level, its condition field is update appropriately to reflect the circumstances under which the leave occurs. As a simple illustration, the following results after the lowest level of the example is processed:

```
    A: BEGIN                    ! start block A
        IF x THEN
        B: BEGIN                ! start block B
            c = 1;
            LEAVE (y) B; ! pushed up LEAVE
            d = 1;
            LEAVE (z) A; ! pushed up LEAVE
            e = 1;
        END;
        f = 1;
    END;                        ! end block B
    g = 1;
END;                        ! end block A
```

The second phase of processing at a level involves replacing LEAVE statements with IF statements. For any particular LEAVE at a given level an IF statement is generated that "protects" the following code. Only if the leave condition is false should the following statements take affect. This suggests a series of nested IF statements which have complement conditions of the leave conditions:

```
A: BEGIN                        ! start block A
    IF x THEN
            c = 1;
            IF NOT y THEN BEGIN         ! generated by "LEAVE (y) B"
                d = 1;
                IF NOT z THEN BEGIN     ! generated by "LEAVE (z) A"
                    e = 1;
                END;
            END;
        LEAVE (z) A;
        f = 1;
    END;
    g = 1;
END;                            ! end block A
```

Notice that the statement "LEAVE (z) A" has been pushed to the next higher level of the parse tree. However, the statement "LEAVE (y) B" has been dropped. This is because it has been pushed to the level of its leave block. It therefore no longer applies and can be dropped. In addition, the "BEGIN" and "END" of *block B* itself have been dropped since *block B* has been fully processed.

Continuing the processing for the next higher level better illustrates the updating of leave conditions.

```
A: BEGIN                        ! start block A
    IF x THEN BEGIN
            c = 1;
            IF NOT y THEN BEGIN
                d = 1;
```

```
                    IF NOT z THEN BEGIN
                        e = 1;
                    END;
                END;
            IF NOT z THEN               ! generated by "LEAVE (z) A"
                f = 1;
        END;
        LEAVE (x AND z) A;
        g = 1;
    END;                                ! end block A
```

Notice that when the statement "LEAVE (z) A" is pushed out of the IF statement, the condition of that IF statement is logically ANDed into the leave condition to produce "LEAVE (x AND z) A."

After one more level of processing, the example has been completely converted:

```
IF x THEN BEGIN
        c = 1;
        IF NOT y THEN BEGIN
            d = 1;
            IF NOT z THEN BEGIN
                e = 1;
            END;
        END;
    IF NOT z THEN
        f = 1;
END;
IF NOT (x AND z) THEN    ! generated by "LEAVE (x AND z) A"
    g = 1;
```

The only possible flow-of-control statements remaining in the parse tree at this point are IF and SELECT statements.

2.7 Multiple Assignment Preprocessing

BDSYN requires three tasks to be performed prior to its multiple assignment phase. The first task is to define a necessary data structure. The data structure created represents the nesting of the IF and SELECT statements. THEN or ELSE blocks in IF statements, or cases in SELECT statements are considered to be "statement blocks." A simple tree is created, where each node represents a statement block. Each statement block in given a unique identifying number, the "block number."

The other tasks performed in this processing step are for optimization of multiple assignment. The first task is to identify and mark the last use of every variable.

This is used by multiple assignment to avoid the creation of unnecessary logic. The second task involves creation of intermediate assignments for IF and SELECT conditions which are complex. For example:

```
    IF x AND (y OR z) THEN ...
```

becomes

```
    $$COND = x AND (y OR z);
    IF $$COND THEN ...
```

This has two main uses. It removes considerable overhead in the multiple assignment routine. More importantly, however, this prevents multiple assignment from combining these complex conditions into assignment statements which are within the scope of the IF or SELECT structure. Conditions in IF and SELECT statements tend to affect many assignment statements. If conditions were to be combined into all of those statements, MIS would be forced to extract them out again.

2.8 Multiple Assignment Processing

The multiple assignment processing is at the heart of BDSYN. It is this processing that allows BDSYN descriptions to be sequential in nature. Successive assignments to the same variable are handled in a consistent manner, in order to create correct combinational logic. All logic variables are handled one bit at a time. Each time a bit of a variable is assigned to, a "revision" of that bit is generated. Each revision has a unique "revision number" which is based on the block number of the assignment statement and the number of revisions to the same variable that precede the current one.

Multiple assignment for the basic case can be accomplished by creating revision numbers for each revision to each variable. Whenever a variable is used on the left hand side of an assignment, a new revision is generated. Whenever a variable is used on the right hand side of an assignment, a reference is made to the most recently generated revision. For example:

```
x = a;
x = x AND b;
```

becomes

```
x.0.1 = a;
x.0.2 = x.0.1 AND b;
```

Handling multiple assignment gets slightly more tricky when IF or SELECT statements are involved. The problem is to make a set of assignment statements which imply the same logic as the IF or SELECT. For example, after the assignment of revision numbers a piece of code may look like the following:

```
IF cond THEN
    x.1.1 = a
ELSE
    x.2.1 = b;
```

Here, the IF statement can be replaced by adding a single assignment statement. The revision *x.0.1* represents the final effect of the IF statement.

```
x.1.1 = a;
x.2.1 = b;
x.0.1 = (cond AND x.1.1) OR (NOT cond AND x.2.1);
```

When an assignment statement to a particular variable does not occur in all clauses of an IF or SELECT statement, it is necessary to reference a previous revision of the variable in the created assignment statement. This preserves the the sequential nature of the code. In the following example, *x.0.1* is used when *state* is either 1 or 2:

```
x.0.1 = a;
SELECT state FROM
    [0]: x.1.1 = b;
    [3]: x.1.2 = c;
ENDSELECT;
```

becomes

```
x.0.1 = a;
x.1.1 = b;
x.1.2 = c;
x.0.2 = ((state EQL 0) AND x.1.1) OR
        ((state EQL 3) AND x.1.2) OR
        (((state EQL 1) OR (state EQL 2)) AND x.0.1)
```

When processing a nested structure of IF and SELECT structure, many assignments must be generated. For every assigned variable that BDSYN encounters, an assignment statement must be generated for each level of nesting. This can be potentially wasteful. If there is no use of a variable after a particular IF or SELECT there is no need to create an extra assignment for it. This is the reason for marking the last occurrence of every variable during the preprocessing to multiple assignment. BDSYN is able to avoid creating superfluous assignment statements, as shown:

```
y = 0;
z = 0;
IF cond1 THEN BEGIN
    IF cond2 THEN
        x = a
    ELSE BEGIN
        x = b;
        z = c;
    END;
    y = x AND d;      ! marked as last use of x
END;
```

becomes

```
y.0.1 = 0;
z.0.1 = 0;
x.2.1 = a;
x.3.1 = b;
z.3.1 = c;

! created assignments for inner IF
x.1.1 = (cond2 AND x.2.1) OR (NOT cond2 AND x.3.1);
z.1.1 = (cond2 AND z.0.1) OR (NOT cond2 AND z.3.1);

y.1.1 = x.1.1 AND d;

! created assignments for outer IF
! no revision for x.0.2 is created here
y.0.2 = (cond1 AND y.1.1) OR (NOT cond1 AND y.0.1);
z.0.2 = (cond1 AND z.1.1) OR (NOT cond1 AND z.0.1);
```

By creating an assignment statement for every variable that is assigned to within an IF or SELECT the input parse tree can be reduced to contain only assignments of boolean expressions. One should also note that sequential nature of the original input is eliminated. The set of assignments remaining in BDSYN can be viewed as a combinational logic network, with each assignment statement corresponding to a complex boolean gate. The remaining phases of BDSYN processing, merely simplify the network slightly.

2.9 Cleanup Evaluation

One might notice that some of the equations generated by the multiple assignment processing are trivial. That is, the right hand side of an assignment contains only a simple variable or a constant. In terms of combinational logic, such an assignment statement corresponds to nothing more than a wire. Unless told not to (by the -b option), cleanup evaluation will eliminate the trivial assignment statements in the parse tree.

At this point in its processing, BDSYN has complete information concerning the boolean equations of the combinational logic and the interconnection of the logic gates. This makes it possible for BDSYN to combine and collapse assignment statements together. Before revision numbers are assigned, BDSYN does not know where a particular revision of a variable fans out. With revision numbers, it is a simple matter to "forward substitute" trivial assignments. For example:

```
x.0.1 = a;
x.1.1 = b;
x.1.2 = c;
x.0.2 = ((state EQL 0) AND x.1.1) OR
        ((state EQL 3) AND x.1.2) OR
        (((state EQL 1) OR (state EQL 2)) AND x.0.1)
```

becomes

```
x.0.2 = ((state EQL 0) AND b) OR
        ((state EQL 3) AND c) OR
        (((state EQL 1) OR (state EQL 2)) AND a)
```

In addition to performing forward substitutions, cleanup evaluation also calls the evaluation routines that were used in the earlier expression evaluation phase of processing. This performs a very simple but effective minimization. If any of the forwarded variables had constant values, entire blocks of logic may be simplified away. If in the above example the variable a were instead the constant value zero, the last term of the final equation would just drop out.

In actual BDSYN descriptions, there is a tendency to describe desired functions in a loose (but more convenient) manner. Usually this results in a set of very complicated equations which have predominantly constant fanin. The combination of forward substitutions and expression evaluation reduce these equations to a much simpler form. Cleanup evaluation typically reduces the number of logic nodes by 20 to 30 percent.

2.10 Output Processing

The final phase of processing is to convert the internal boolean equations to BLIF format. It is a simple matter to make a series of small BLIF tables to represent a boolean equation. Unfortunately, BDSYN only knows how to create tables for simple boolean functions (AND, OR, etc.). For each complex boolean equation, there may be many BLIF tables. In order to avoid creating very large numbers of very small tables, BDSYN calls MIS to collapse many small BLIF tables into fewer large tables.

BDSYN has some latitude in deciding which and how many boolean equations to collapse into each final BLIF table. There are many considerations in this making this decision. There should be enough collapsing done to reduce the output to a manageable size. The collapsing should not cause the removal of a variable that the user may have considered important. Most importantly, the intermediate variables which fan out to several destinations should not be collapsed out. These intermediate variables could be seen as common factors to gates that follow them. If they were to be collapsed into their fanout gates, MIS would have to extract them again (a potentially costly operation). Instead by preserving these factors, MIS can choose to reatin these factors or delete them.

In order to preserve important intermediate variables, BDSYN performs a preprocessing step to the output generation. This step marks which equations should not be collapsed by MIS. The algorithm is simple. For each primary output of the combinational network, a recursive search is made of the transitive fanin to those outputs.[3] Whenever a revision of a variable fans out to revision of a revision of a different variable, the former is marked to be preserved. This way the revisions that are collapsed out are those which fan out only to revisions of the same variable.

BDSYN's default algorithm for choosing equations to collapse out is not perfect. It is possible to describe a circuit in which BDSYN will try to collapse too much logic. This may result in an exponential blow-up in the size of a BLIF table. It is extremely difficult for BDSYN to detect when this behavior will be insighted. As a secondary solution, BDSYN provides an option (*-c1*) that will change the collapsing algorithm. The option causes each equation to be collapsed separately with no equations being collapsed out. This insures that the biggest BLIF table created is only as complex as a single equation. If a single user equation proves too complex to collapse, the option *-c0* will turn off collapsing altogether.

As mentioned before, BDSYN calls MIS to do its logic collapsing. This is accomplished through the use of UNIX[4] pipes. The output of BDSYN is "piped" to the input of MIS and the output of MIS is piped to the input of BDSYN. The two programs communicate with each other in standard BLIF format. On non-UNIX systems, it is not possible to use pipes, and BDSYN can not collapse logic. Although this causes the amount of BDSYN's output to increase dramatically, the logic can be reduced in a separate processing step by MIS.

[3]Transitive fanin is the set of signals which affect a node.
[4]Unix is a trademark of AT&T Bell Laboratories.

Chapter 3

Bdsim Simulator

3.1 Introduction

BDSIM is a switch-level event driven simulator written for use with the OCT data manager. It has three primary goals: loading and executing simulations quickly; using the circuit hierarchy information provided by OCT to aid the user in accessing data; and providing a powerful user interface for quick circuit debugging. The first goal is accomplished, in part, through the use of very simplified circuit models. Transistors are treated as voltage-controlled switches, and capacitance is completely ignored with all nodes being considered as storage nodes. BDSIM also speeds simulation by using a unique transistor simulation algorithm and performing preprocessing optimizations on the input circuit.

BDSIM meets its second goal of using input hierarchy information, by providing the user with a special interface to the simulation data. This interface makes the data within BDSIM appear to be stored in a hierarchical fashion. The manual page for BDSIM in appendix F contains descriptions of commands which access data using hierarchy.

To meet its third goal, BDSIM supplies a wide range of user functions that are intended to make it powerful and easy to use. Users may create their own commands through the use of parameterized macros. Users may create source files of commands which they can run in batch mode or single step mode. There are commands to create and manipulate groups of signals as bit vectors. Output data is easy to format through the use of 'C-like' format strings. BDSIM has two commands that allow the user to access low level data about nodes and active instances in the simulation database. BDSIM also has a particularly useful command that lets the user trace the origin of particular signals in a circuit network. The commands to perform these functions are discussed in detail in the BDSIM manual page.

3.2 Definitions and Data Structures

BDSIM does most of its processing on two classes of data elements. The first is the "instance." An instance represents one or more transistors in the logic network, logic gates, latches, or user described device. Instances predomently represent the active elements of the circuit, but are also used in representing hierarchy (as described below). The other major data element is the "node." Nodes represent the connecting wires between instances and have a voltage level associated with them. Each "terminal" of an instance may be attached to a node. Each node might fan out to the terminals of many instances. An instance terminal might drive a node or might be driven by a node.

BDSIM is set up to handle input from a hierarchical circuit representations such as the OCT data manager. To the BDSIM simulator, the circuit appears to be flat, but enough information about the initial hierarchy is kept to allow the user to view the circuit as a hierarchical structure. The data structure can be described as a pyramid as illustrated in figure 3.1. At the base of the pyramid are the nodes and instances on which the simulation is actually done. Above this base the hierarchy information is annotated. Special "module instances" which do not imply any actual circuit devices are used as internal nodes for a hierarchical tree structure. Each module instance may contain other module instances or "leaf instances" which are the actual simulatable devices. The user view of the simulation data is through

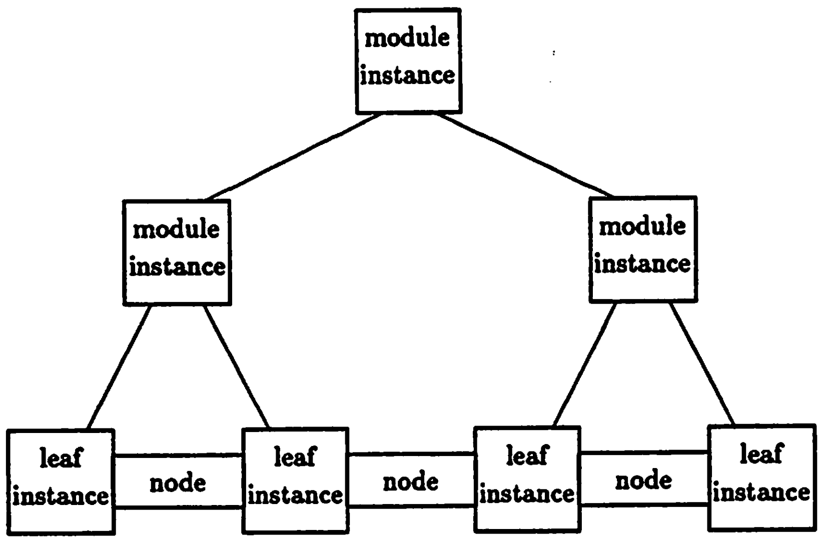


Figure 3.1: Storing Hierarchy Information in Bdsim

the annotated data which constitutes the top of the pyramid. The simulator itself only uses information at the base of the pyramid since it needs only the flat interconnection information.

The simulation nodes in BDSIM are also accessed through the hierarchy structure. Each module instance contains a set of "elements" which contain the names of connections that exist in the original hierarchy.[1] Each element points to the node which implements its connection. It is possible for a connection to extend across several levels of hierarchy, and in that case there may be several elements which connect to the same node. This is illustrated in figure 3.2.

3.3 Simulation of Transistor Networks

The algorithm to find the voltage on any particular node in a circuit involves keeping two levelized graphs. The nodes of the graph are the BDSIM nodes. The arcs of

[1]For Oct input, elements represent Oct nets.

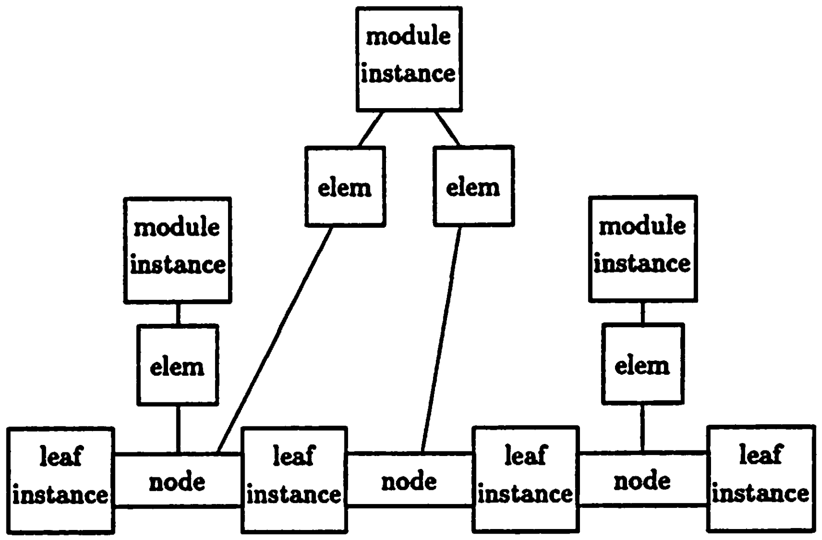


Figure 3.2: Storing Element Names in Bdsim

the graph are source to drain connections of conducting (active) transistors in the circuit. The nodes of the respective graphs are given level numbers based on the length of the shortest conducting paths to power or ground. As transistors switch off and on, the graphs are updated to reflect the change as shown in figure 3.3. If at any time a node is connected by a path to ground and not connected to power, the voltage on the node would be low. If a node is connected by a path to power and not to ground, the voltage on the node would be high. If a node is connected to both, its voltage is unknown. And if a node is connected to neither rail, it will be charged to the voltage level at which it was last driven.

Keeping a levelization of nodes offers several advantages for speeding up processing. MOS transistors are bidirectional in nature and can conduct from source to drain or from drain to source. By using the level information of the nodes, every conducting transistor can be given a direction. When a transistor switches off, it is easy to determine which nodes will be cut off from the power or ground rail and which nodes will not. The nodes that are closer to a rail than the given transistor

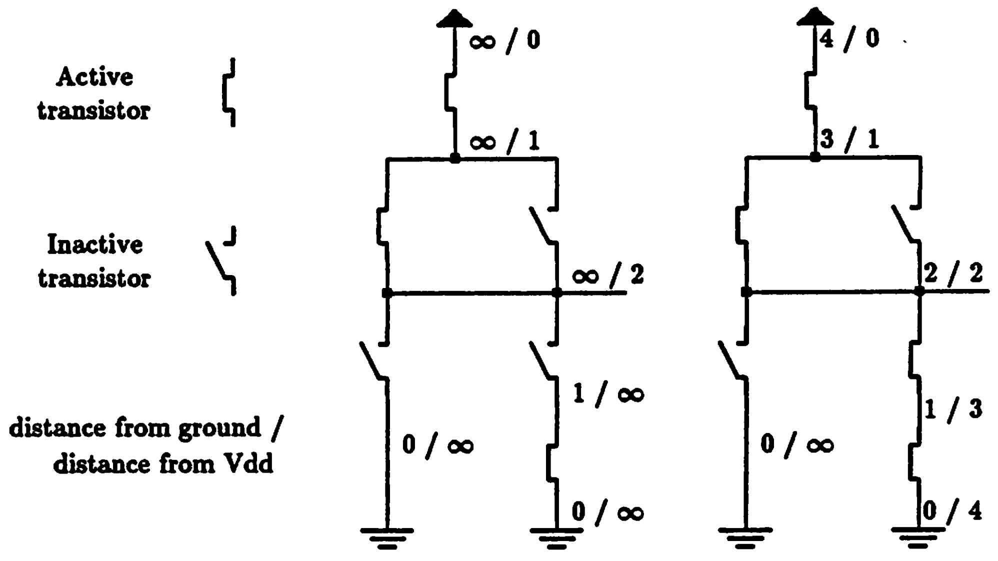


Figure 3.3: Levelization of Transistor Networks

need not be re-evaluated. Conversely, when a transistor switches on, it is easy to determine in which direction it will drive. Only nodes which "follow" the transistor need to be considered.

A further optimization is also done. For each node a count is kept of how many transistors are driving it at the current level. This is quite useful in determining whether a transistor affects the node to which it fans out. For instance, if two transistors drive a node at a certain level and one of the transistors switches off, the levelization of the graph does not change. In general this scheme may prevent a large majority of transistors and nodes from being evaluated when a transistor switches. Figure 3.4 gives an example where only two of seven node are affected by a switching transistor.

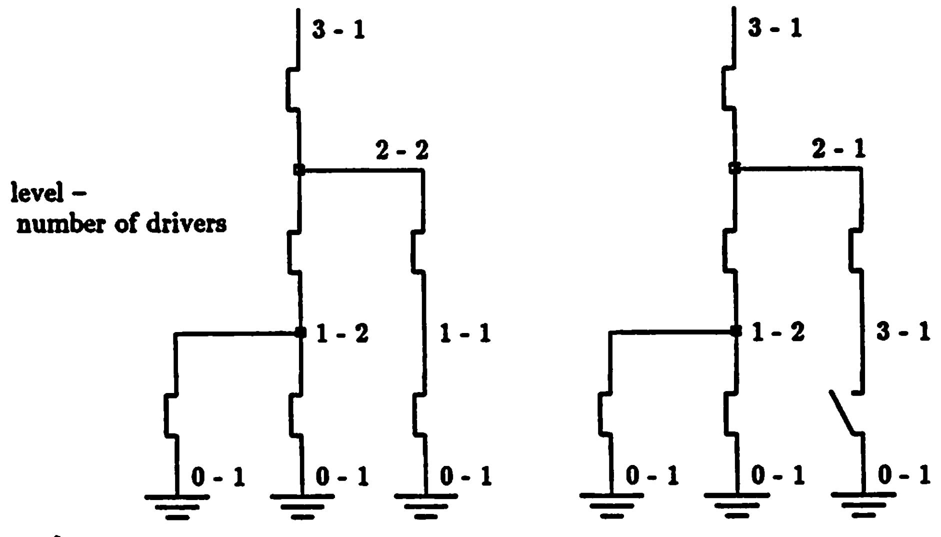


Figure 3.4: Using Driver Counts on Nodes

3.4 Capacitance and Charge Sharing

In a physical circuit, two oppositely charged nodes that are electrically connected affect each other by "charge sharing." Depending on the relative magnitudes of the capacitance on the two nodes, both nodes may be charged high, low, or unknown after charge sharing occurs. Some existing switch level simulators attempt to model charge sharing by doing pairwise comparison of node capacitances [15]. If one node's capacitance is sufficiently larger than the others, then both nodes receive the voltage of former node. If the nodes have comparable capacitance, they both receive an "undefined" voltage. The problem with this approach is that it may give an incorrect result if more than two nodes contribute to the charge sharing. The correct, but very expensive, solution to this problem is to determine the result of charge sharing by taking the capacitance weighted average of voltages over all nodes which contribute to the charge sharing.

BDSIM assumes that every node in a circuit has infinite capacitance. That is, when a driver to a node is removed, that node will retain its original voltage until it is driven in the opposite direction. When two charged nodes are electrically connected to each other, BDSIM assumes that they do not affect each other. In effect, BDSIM ignores charge sharing altogether. This is a disadvantage, but it does allow BDSIM to run faster. Handling charge sharing correctly can be very expensive and, indeed, may not be possible in a zero-one simulator. BDSIM defers the simulation of charge sharing to systems that can better model electrical effects.

3.5 Getting Real Circuits to Simulate

There are several classic circuits which switch level simulators have trouble simulating. Some of them are difficult to handle because they rely on analog effects and voltages that are between high and low for their operation. Some circuits rely on certain timing characteristics that a simple event driven simulator can not handle. BDSIM supports two constructs which help to deal with difficult circuits. These constructs are indicated by the user to give BDSIM hints on how to simulate.

The most common situation that is difficult to simulate is when one transistor is expected to "over-drive" another. That is, when two transistors are driving the same node in opposite directions, one is expected to win. The usual result when two transistor are fighting is that the node receives a value of "unknown." BDSIM resolves the problem of fighting transistor by allowing the specification of "weak" transistors. A signal which is passed through a weak transistor can always be overcome by a signal from a regular transistor. Weak transistor are selected in one of two ways. A weak transistor is created if a transistor's W/L ratio is less than some user specified threshold. Alternatively, the user may label individual transistor as weak.[2]

Figure 3.5 illustrates two examples of where weak transistors are required. The example on the left is a NMOS inverter. Transistor 2 must be weak so that tran-

[2]This is done with properties in Oct, or gate attributes in Magic.

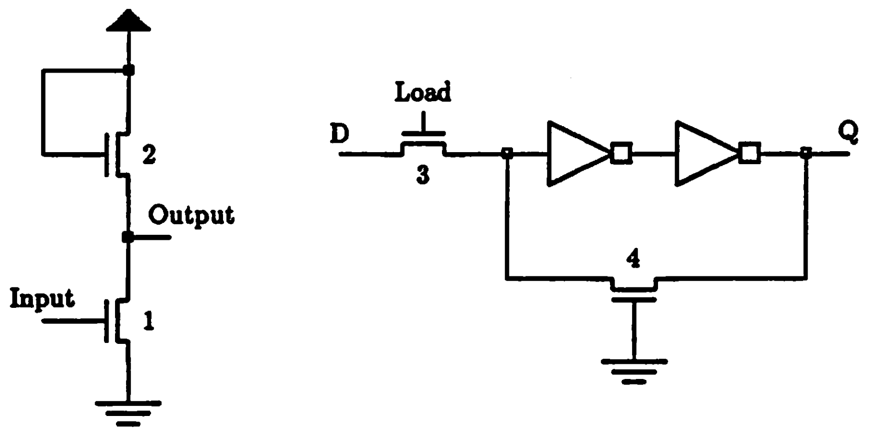


Figure 3.5: Using Weak Transistors

sistor 1 can pull *output* low. The example on the right is a static transparent latch. The feedback transistor 4 must be weak so that transistor 3 can place a new value in the latch.

A classic circuit that is difficult for switch level simulators is the exclusive OR gate shown in figure 3.6. The problem in simulating this circuit occurs when the inputs change. It is possible for some of the internal nodes to take on undefined values during the switching time and get stuck at undefined. As a concrete example assume that the B input of the exclusive OR gate is set high and the A input makes a low to high transition. During transition, node $\bar{A}$ is driven low by the input transistor, but is also driven high by a signal propagating downward through transistor 1. This causes node $\bar{A}$ to take on a value of undefined, and transistor 2 to be stuck on rather than turning off as it should.

BDSIM allows the user to label transistors as being unidirectional. This can be used to fix the particular problem described by telling BDSIM that transistor 1 is intended to propagate signals in the upward direction only. This breaks the feedback loop described in the last paragraph and guarantees node $\bar{A}$ can only be affected

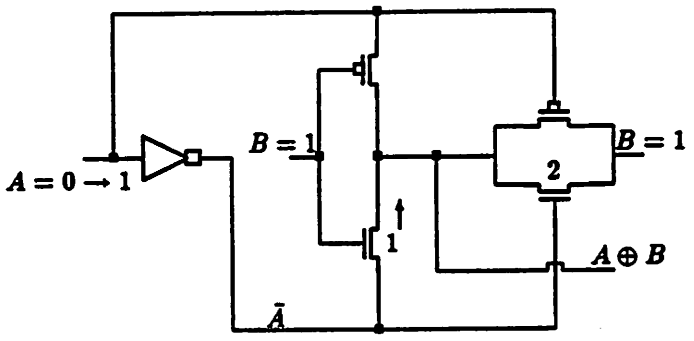


Figure 3.6: Using Directions on Transistors

by the input inverter. Of course the user should do a thorough circuit simulation to assure that the gate really does behave in the expected manner.

3.6 Preprocessing Optimizations

As part of its read-in phase, BDSIM performs two types of optimization on the topology of given circuits. Both of these optimizations simplify the circuit and try to decrease the work that the simulator has to do during simulation. While these optimizations do change the leaf instances in the given circuit, they do not change the behavior of the circuit or the results of the simulation.

3.6.1 Pullups and Pulldowns

The first preprocessing optimization that BDSIM makes is a simple one. All transistors whose sources or drains are attached directly to power or ground are replaced with "pullups" or "pulldowns" respectively. Pullups and pulldowns, as defined by BDSIM, are two terminal devices with a gate and an output. When the voltage on the gate is such that the pullup (pulldown) is active, a high (low) voltage will be driven at the output. When the voltage on the gate is such that the device is inactive, the output will be high impedance. Pullups and pulldowns act exactly as the transistors they replaced would have, with one important exception. The pullups and pulldowns are unidirectional devices which only propagate signals away from the rails. They never attempt to drive the power and ground rails. MOS transistors are bidirectional devices to BDSIM. If the transistor replacement is not done, BDSIM spends a significant amount of effort trying to propagate signals through the transistors toward the rails. This effort is serves no purpose and is avoided by the transistor replacement.

3.6.2 Merging Transistor Networks

BDSIM performs a second preprocessing optimization step on the input circuit. This step involves identify and merging together series-parallel connections of similar transistors (all NMOS or all PMOS) into one simulatable instance. These "transistor networks" are a multi-terminal devices. They have one source terminal, one drain terminal, and one gate terminal for each transistor that comprises it. When the gate terminals have voltages on them such that a certain boolean equation is satisfied, the network conducts between the source and drain. Otherwise, the network is off. Transistor networks are very similar to simple transistors and, in fact, are scheduled and simulated in much the same way.

The advantage of creating transistor networks is fairly obvious. Several transistors can be compressed into one BDSIM instance. This eliminates the overhead of scheduling many instances. The merging process also eliminates some of the circuit nodes completely, again decreasing overhead. The actual evaluation of boolean expressions in transistor networks is done by table lookup. This makes simulating a transistor network virtually as fast as simulating a simple transistor. In fact, the percentage of simulation time saved by this optimization is potentially as great as the percentage of transistors merged away.

The algorithm for generating transistor networks involves a recursive merging of small transistor networks into larger ones. In the following discussion I will consider simple transistor to be the same as a transistor network with one transistor.[3] The algorithm incorporates two types of merging – series and parallel. A parallel merge is indicated when two transistor networks share the same source and drain nodes. To perform a parallel merge, a new network is created with the correct number of gates and an appropriate boolean function. The two original networks are then eliminated and the new network is inserted with source and drain terminals connected to the original source and drain nodes.

A series merge of networks is slightly more tricky. When two networks are merged in series, the node that originally joined the two networks is eliminated. Care must be taken to avoid merging away important nodes. The criteria for performing a series merge is that two networks must have a common source or drain on a node. The node may have no other connections to any other instance. The series merge is performed by creating a new network and replacing the two original networks. The node that formerly joined the two discarded instances then has no connections and is also discarded by BDSIM.

Notice that eliminating nodes from a logic circuit is not necessarily a safe practice for circuit simulators. When a node is eliminated so is its associated capacitance. If BDSIM performed calculations involving capacitance, some amount of processing would be required to accommodate for eliminated capacitance. As BDSIM stands, eliminating nodes does not present a problem.

A key problem in defining a strategy for merging transistors is that the number of permutations for constructing series-parallel networks grows extremely quickly as more merging is done. Since it was desired that lookup tables be used for the evaluation of the networks' boolean equations, it was necessary to limit the number of possible networks that could be created. The number of permutations are limited somewhat by the fact that BDSIM will only create networks of five transistors or

[3]In fact, this is what bdsim does internally.

fewer. The number of permutations is limited further by applying an ordering to the merging operation. The smaller of two networks is merged *into* the larger. With a maximum network size of five transistors, the smaller network in a merge can contain only one or two transistors. This scheme results in a fairly small table of possible merge operations which are outlined in appendix E.

3.7 Gate-level Instances

BDSIM is capable of handling gate-level constructs as well as transistors. These gate-level constructs may range in complexity from inverters and buffers to latches, sense amps, and memory cells. Simulating higher level gates in BDSIM is quite simple. Gate constructs are treated as BDSIM instances, just like transistors and transistor networks. Gate instances, like transistor instances, are connected to a number of nodes. Gates typically sense the voltages on the nodes at some of its terminals and drive a voltage onto other nodes.

Gate constructs are read into BDSIM from OCT and are identified by properties attached to the OCT instances. When BDSIM is unable to find identifying properties on an instance, it assumes that it is a "module instance" and looks at lower levels of the hierarchy searching for construct it recognizes. Currently, there is no way to read gate structures from the *sim* files produced by Magic. BDSIM has only one built-in type of gate construct – the logic gate. The logic gate is special in that it has a variable number of terminals associated with it (depending on the gate's function).

All gate-level structures other than logic gates are integrated into BDSIM by linking user supply subroutines with the BDSIM code. BDSIM expects a set of routines for each gate type. These include routines for reading the gate from Oct, initializing the gate, simulating the gate, saving the state of the gate (for the BDSIM "save state" command), loading the state of the gate, and printing information about the gate (for the "instanceinfo" command). BDSIM supplies a set of interface routines to aid the user in defining gates.

3.8 Scheduling Algorithm

BDSIM uses an "event driven" scheme to accomplish simulation. Events, such as changes in node voltages cause, cause other events, such as re-evaluation of gates, to be scheduled. When an event is scheduled, it is put in a queue with other events. These events are processed in a first in first out (FIFO) manner. Events are scheduled and processed until there are no more events on the queue.

BDSIM has three distinct event queues. The first queue must always be empty before the second queue can be processed. The second queue must be empty before the third queue is processed. This results in a three tier priority system, with events being processed in a FIFO manner on each tier. The three queues, in order of priority, operate on levelized graphs described in section 3.3, BDSIM nodes, and BDSIM instances.

The highest priority queue is used to keep the levelized graphs of transistors up to date. Whenever a transistor is switched on or off, it is necessary to assure that the levelized graphs are still correct. This is done by scheduling the transistors that are directly affected by the switching transistor. As the scheduled transistors are processed, changes in the levelization of the graph are propagated. Processing associated with the first scheduling queue can be viewed as a separate mini-simulation that deals only with small networks of transistors. Once these networks are processed, the second and third queues are used to simulate the entire circuit.

The second queue contains nodes which were touched as graph levelization was done. The nodes are processed to ascertain if the graph relevelization caused the voltage on the node to change. If the voltage did indeed change, any instance affected by the change is added to the third queue. When instances on the third queue are evaluated, their outputs may change causing further scheduling in the first two queues.

BDSIM's scheme for identifying instances to be scheduled is based on "terminal types." Each instance has several terminals which attach to nodes. These terminals are given types which suggest their function and how their associated instance should be scheduled. The most inactive terminal is the OUTPUT. These are terminals that are not affected by a change in node voltage. When a node voltage changes, instances whose connection to that node is an OUTPUT need not be scheduled. The next type of terminal type is the voltage sensitive input, VSI. Instances connected to a node by one of these terminals are scheduled whenever the node voltage changes. Transistor gate terminals are VSI's.

Instances with drive sensitive inputs, DSI's, are scheduled whenever the node associated with the terminal changes state in any way. For instance, a node may change from being driven high to being charged high. This change in a node would schedule DSI's but not VSI's. DSI's are useful in implementing bidirectional terminals which output signals when they are not being driven.

The final type of terminal is the TRANSISTOR input. These are used exclusively for transistor source and drain terminals. They are used to indicate which instances should be considered when processing the first scheduling queue. Instances which are scheduled in association with TRANSISTOR inputs are scheduled on the first queue. All other instances, including transistors that were triggered by their gate terminals, are scheduled on the third queue.

Chapter 4

Conclusion

I have described two programs which can be used as part of a VLSI design system. BDSYN can be used to describe and implement logic quickly and easily. It allows a designer to worry about the high level issues of design. Many of the low-level details are abstracted away. BDSIM allows a designer to test the soundness of a design, and attempt to convince himself of a design's correctness. It is hoped that the powerful user commands and speed optimizations in BDSIM provide the designer with an environment where he can quickly locate errors in a design.

There are a few projects in the realm of these two programs that would be useful to attack in the future. BDSIM could be extended to incorporate functional descriptions. High-level, gate-level, and transistor-level descriptions could be integrated together in one simulation to allow testing of partially implemented circuits. Also, large sections of a design could be described at a high level to afford faster simulation. It would be particularly useful to support the simulation of bdsyn descriptions, allowing a designer to simulate and synthesize from a common source.

Probably, the most useful work that could be done in the realm of these programs (and certainly the most interesting) would be to extend BDSYN (or rewrite it) to allow combinational *and* sequential descriptions. In my opinion, this could best be done by defining a language which used combinational logic blocks as basic units. Each block could look identical to current BDSYN syntax. These blocks could then be connected together by flow-of-control structures which indicated which blocks were active at any one time. The program that read such a description could infer the logic was needed to implement sequencing between logic blocks, the logic needed to arbitrate the outputs of the logic blocks, and where latches would have to be inserted to make the circuit work correctly.

Appendix A

Bdsyn Users' Guide

A.1 Introduction

BDSYN is a hardware description translator. It takes as input a textual description of a block of combinational logic and produces a collection of logic functions which implement the described function. BDSYN was written to be a front end to the Berkeley Synthesis System. Work on BDSYN was begun in the spring of 1986 as a class project for Berkeley's synthesis class. For the synthesis class, we were given permission to use Digital Equipment Corporation's proprietary multi-level simulator DECSIM. The functional simulation language BDS is used for high level simulation in DECSIM. We have chosen to use a subset of the BDS language as the input language to BDSYN. The output of BDSYN is a multiple-level logic representation of the described function. The output created is in BLIF (Berkeley Logic Interchange Format). This output is suitable input for MIS which is the multiple level logic minimizer in the Berkeley Synthesis System.

A.1.1 What Bdsyn Is

BDSYN is a tool for quickly describing and implementing combinational logic. It allows the user to describe a circuit function and produce an implementation of that circuit automatically. BDSYN's output is given in a "multiple-level" format.

This means that BDSYN is capable of translating many types of logic configurations that would be impractical to represent in a two-level PLA form. BDSYN supports many high-level language constructs: subroutines, IF–THEN–ELSE, SELECT, constant bounded FOR loops, multiple assignment to variable, and multiple returns from a single routine. In addition, BDSYN supports complex operators such as addition, and shifts by a variable amount. (Note that complex operators may imply a large number of gates.)

When BDSYN is used in conjunction with MIS, it can be used to describe logic in standard cell, PLA, or a more complicated module generated form.

A.1.2 What Bdsyn Is Not

It is very important to keep in mind that BDSYN can be used for describing combinational logic blocks only. We define combinational logic to be any block of logic which does not use system clocks or signal latching of any kind. Combinational logic does not hold state information. In particular, describing a finite state machine in BDSYN requires the logic designer (or a program) to add external latches to the described combinational logic block that hold the current state of the machine.

The limitation of describing only combinational logic is common to PLA description programs as well as BDSYN. (Although there may be provisions for automatically adding the required external latches.) The reason we find it necessary to describe this limitation in such detail here, however, is that some of the BDSYN constructs (e.g. FOR loops and multiple assignment) appear very sequential in nature. In reality, they are merely interpreted as a short hand for describing parallel structure. There are many examples of the use of sequential-like descriptions in the last section of this guide.

A.2 Using Bdsyn

A.2.1 Describing Logic in Bdsyn

BDSYN descriptions are very similar to standard procedural programming languages (Pascal, C, etc.) Describing logic in BDSYN amounts to writing a program that accomplishes the desired function. Each flow-of-control structure and operation corresponds roughly to a small block of logic. Each variable corresponds to "bit vector," which is a group of single bit signals in the hardware.

At the top of the BDSYN description it is necessary to define input and output "ports." In a programming point of view, these ports can be viewed as external variable declarations. In a hardware point of view, these ports represent primary input and outputs of the combinational logic block. Once the ports are declared, the logic designer should define a set of values for the input ports for which he wishes the hardware to operate. If he then writes a program that will generate desired output values for each of the interesting input combinations, this constitutes a legal description.

To say it in a different way, suppose a BDSYN "program" has been written. BDSYN will create logic that will act in an identical manner to that program. If a set of inputs were to be introduced to the BDSYN program and the program were executed (run in a software sense), a particular set of outputs would be generated. If these same inputs were introduced to the combinational logic block created by BDSYN, the same set of outputs would result.

A.2.2 Running Bdsyn

Once a hardware description has been written, it can be converted to logic by running BDSYN. The input description is read from the file given on the command line. The BLIF output is returned to standard output. There are several command line options for BDSYN which are outlined in the BDSYN manual page. Two of the options, however, will be discussed here.

The -b option tells BDSYN not to do its cleanup evaluation. This cleanup evaluation process in BDSYN eliminates redundant and unnecessary logic. Unfortunately, it may also have the undesired effect of eliminating user specified intermediate variables that BDSYN considers unimportant. The -b makes sure that the variable are preserved, at the price of extra logic.[1]

The -c option is followed by a number which tells BDSYN how much "collapsing" of logic that it should do. Collapsing is done in order to decrease the amount of output which BDSYN creates. Although quite rare, it is possible to create an input description which causes BDSYN to collapse too much logic and run for inordinate amounts of time. The option -c1 will limit the amount of collapsing that is done, and will alleviate the problem. BDSYN uses MIS do to logic collapsing for it. The actual communication with MIS is accomplished through the use of UNIX pipes. On non-UNIX systems or on a system where MIS is not available, the option -c0 should be used to turn off collapsing altogether.

A.2.3 What is New in Version 1.1

There are two main changes in BDSYN since version 1.0 that users may notice. The first change is that the intermediate variable names created by BDSYN are shorter. Each variable is followed by two numbers. The first number is the block in the description in which the variable is set. The second number represents how many times the variable has been revised in the block.

The second change is that BDSYN will create its own intermediate variables corresponding to routine return values and condition expressions from IF and SELECT statements. These variables are good candidates for having a large fanout. They are created in the hope that it will save MIS some work in extracting common factors. On occasion, BDSYN may create an internal variable which does not fan out. MIS will give a warning when such a variable is created. In general, it is safe to ignore

[1]The extra logic could easily be reduced by MIS at a later time.

these warnings for variables whose name begin with "$$COND."

A.3 Language Constructs

A.3.1 Names and Numbers

BDSYN accepts a subset of the BDS language as input. BDS is a Pascal-like language. The input consists of names, numbers, and reserved keywords separated by white space. White space is spaces, tabs, and carriage returns. Any characters following a '!' (exclamation point) are interpreted as comments by BDSYN. Legal names may contain the characters a-z, A-Z, 0-9, '_' (underscore), and '%'. Other characters are legal in the name as long as the name is quoted with double quotes ("). Names may not be any of the reserved keywords (given below) unless they are quoted. Names that begin with '$' are illegal. BDSYN, like DECSIM, is case insensitive, so the name "signal" is the same as the name "SIGNAL."

Numbers in BDSYN must be positive and may be given in any one of base 2, 4, 8, 10, or 16. Decimal numbers are represented as a simple integer (e.g. 511). Other base numbers are represented as an integer followed by a '#' and the base (e.g. 1FF#16). The width, in bits, of a decimal number is the minimum number of bits required to represent that number. The width of a number with a '#' is equal to the number of bits required to represent a number in that base with the given number of digits. For example 001#2 has width three, 123#8 has width nine, and 12345678#16 has width thirty-two. BDSYN has the limitation that numbers in the input must be thirty-two bits or fewer. This limitation may be overcome by the use of the concatenation operator. (e.g. 12#16 & 34567890#16 is a forty bit constant.)

A.3.2 Reserved Keywords

Table A.1 gives a list of reserved words for BDSYN. These may not be used as names unless they are quoted. In addition to the reserved words there are several keywords

AND	BEGIN	BUF	BY	CONSTANT
DO	DOWNTO	ELSE	END	ENDMODEL
ENDROUTINE	ENDSELECT	ENDSELECTALL	ENDSELECTONE	EQL
EQV	FOR	FROM	GEQ	GTR
IF	LEAVE	LEQ	LSS	MACRO
MOD	MODEL	NAND	NEQ	NOR
NOT	OR	OTHERWISE	OXT	REQUIRE
RETURN	ROUTINE	SELECT	SELECTALL	SELECTONE
SL0	SL1	SLR	SR0	SR1
SRR	STATE	SXT	SYNONYM	THEN
TO	WIDTH	XOR	ZXT	

Table A.1: Bdsyn Reserved Words

BEHAVIOR	ENDBEHAVIOR	ENDMODULE	ENTRY	FORWARD
FOURSTATE	HALT	INFORM	INPUT	INVALID
MODULE	NOENTRY	OUTPUT	PORT	REPEAT
REVISION	STATIC	TWOSTATE	UNTIL	WHILE

Table A.2: Keywords Ignored by Bdsyn

which BDSYN explicitly ignores but are included so that complete BDS descriptions (not just our subset) may be parsed. They are listed in table A.2

A.3.3 Variables

BDSYN has two types of variables which it understands. The first, "logic variables," imply actual logic and can be thought of as wires in the actual layout. Logic variables are declared with bit subscripts that define their width. The lower bit numbers correspond to the less significant bits, and BDSYN requires that the high bit be first.

```
STATE name1<7:0>;
```

If the bit range of a variable is omitted in its declaration,

```
STATE name2;
```

BDSYN will assume a single bit vector (name2<0>).

When using a logic variable (as opposed to specifying one), the bit subscript is used to choose certain bits from the bit vector. If the bit vector is left out, the the entire bit vector is used.

In addition to logic variables, BDSYN defines "meta-variables." Their main application is as the index variable for FOR loops. Loop index variables are required to be meta-variables. Meta-variables are declared with null bit subscripts:

```
STATE name3<>;
```

Meta-variables do not imply any logic, and are used for temporary storage of constants. When meta-variables appear on the left hand side of an assignment, the right hand side is evaluated and assigned to the meta-variable. The right hand side of this assignment must be a meta-variable expression (i.e., an expression written from meta-variables and constants). When meta-variables appear on the right hand of an assignment, they are treated exactly like constants. This means that meta-variables may be used in places where logic variables are disallowed by BDSYN. Section A.6 covers more about meta-variables.

A.3.4 Input Format

The following is a BNF description of a typical input program for BDSYN. Keywords are given in capital letters, and user names and productions are given in small letters. Optional phrases are given in square brackets '[ ]'. Phrases in curly braces '{ }' many be repeated zero or more times in the input. The character '|' (vertical bar) represents a choice of many lines.

The basic BDSYN input file consists of a "MODEL":

```
MODEL model_name [output_var {, output_var}] =
                              [input_var {, input_var}] ;
    {global_declaration ;}
    {routine}
ENDMODEL [model_name];
```

A "global_declaration" is defined as the following:

```
|   STATE global_var {, global_var}
|   CONSTANT constant_name = number {, constant_name = number}
|   SYNONYM var_name = var_name {, var_name = var_name}
```

A "routine" is defined as the following:

```
ROUTINE routine_name [bit_subscripts] [( var_param {, var_param} )] ;
    { local_declaration ;}
    { [ label_name: ] statement ;}
ENDROUTINE [routine_name];
```

A "local_declaration" is defined as the following:

```
|   STATE local_var {, local_var}
|   CONSTANT constant_name = number {, constant_name = number}
|   SYNONYM var_name = var_name {, var_name = var_name}
```

A statement of a routine is defined below. Note: The square brackets in the SELECT statements represent actual brackets in the input.

```
|   BEGIN {statement ;} END
|   variable = expression
|   IF expression THEN statement [ELSE statement]

|   FOR meta_var FROM const_expr TO const_expr
                        [BY const_expr] DO statement

|   FOR meta_var FROM const_expr DOWNTO const_expr
                        [BY const_expr] DO statement

|   SELECT expression FROM {[expression {, expression}]: statement;}
                        {[OTHERWISE]: statement ;} ENDSELECT

|   SELECTALL expression FROM {[expression {, expression}]: statement;}
                     {[OTHERWISE]: statement ;} ENDSELECTALL

|   SELECTONE expression FROM {[expression {, expression}]: statement;}
                        {[OTHERWISE]: statement ;} ENDSELECTONE

|   RETURN [expression]
|   LEAVE label_name
```

Note that these tables imply that a semicolon ';' separates every statement. As in Pascal, there is no semicolon before ELSE, but unlike Pascal there is a semicolon before END. Expressions and const_expr's (constant expressions) will be described in later sections.

A.3.5 Declaration Statements

The MODEL statement defines the primary inputs and outputs to the combinational logic block being described. The declared "ports" may not be meta-variables.

The STATE, CONSTANT, and SYNONYM may be used as either global declarations, applying to all routines, or local declarations, applying to only one routine. The STATE statement is for declaration of variables. All variables must be declared prior to use. Local variables can be used only within the scope of a routine and their values are lost between consecutive calls to a routine. Constant declarations are used to assign a name to a constant. Declared constants can be used anywhere a constant can. Synonyms are used to create an alternate name for a previously defined variable. They are particularly useful for defining subfields of a variable:

```
SYNONYM opcode<6:0> = instruction<31:25>;
```

A.3.6 Flow-of-control Statements

BDSYN offers a rich selection of flow-of-control statements. They are IF, SELECT, SELECTONE, SELECTALL, FOR, RETURN, and LEAVE. The IF statement implements optional execution of a statement. If the least significant bit of the IF expression evaluates to one, the THEN clause is used. If the low bit evaluates to zero, the ELSE clause (if present) is used. The SELECT statement is used to execute one statement out of many. If the expression following the SELECT keyword is equal to the expression appearing in the square brackets, the corresponding statement is executed.

SELECTONE and SELECTALL are simply a shorthand for several IF statements. SELECTONE implies nested IF statements:

```
SELECTONE expr FROM
    [expr1, expr2]: statement1;
    [expr3]: statement2;
    [OTHERWISE]: statement3;
ENDSELECTONE;
```

is equivalent to:

```
IF (expr EQL expr1) OR (expr EQL expr2) THEN
    statement1
ELSE IF expr EQL expr3 THEN
    statement2
ELSE
    statement3;
```

SELECTALL is equivalent to sequential IF statements:

```
SELECTALL expr FROM
    [expr1, expr2]: statement1;
    [expr3]: statement2;
    [OTHERWISE]: statement3;
ENDSELECTALL;
```

is equivalent to:

```
IF (expr EQL expr1) OR (expr EQL expr2) THEN
    statement1;
IF expr EQL expr3 THEN
    statement2;
IF (expr NEQ expr1) AND (expr NEQ expr2) AND (expr NEQ expr3) THEN
    statement3;
```

SELECTALL is different than SELECT in that SELECTALL implies sequentiality in its case evaluation. The SELECT statement can be viewed as having all of its cases evaluated in parallel. A SELECTALL allows two cases to evaluate to true. If two cases are true in a SELECT, incorrect logic may be produced. SELECT generally implies much less logic, however, because there is no implied ordering of the cases.

FOR statements are used to iterate a single piece of code several times. As a hardware description language, BDSYN must enforce several rules on the FOR statement. The expressions which specify the start, end, and step values for the loop index must be either constants, or constant expressions. This is because the number of iterations must be fixed. BDSYN also requires that the index variable of the FOR statement is a meta-variable. FOR loops may be nested, and since meta-variables are treated as constants by BDSYN, the inner loop may depend on the outer loop.

```
FOR i FROM 0 TO 7 DO
    FOR j FROM i TO 7 DO
        statement;
```

The RETURN statement is used to break out of a subroutine and return to the calling routine. If the subroutine returns a value (as described below) the RETURN statement must be followed by an expression which is the return value. The LEAVE statement is similar to RETURN in that can be used to break out of a block of statements. When a LEAVE statement is encountered, control will break out of a labeled statement. The LEAVE must occur within the labeled statement to which it points. For example:

```
loop: FOR i FROM 1 TO 10 DO BEGIN
    statement1;
    IF condition THEN LEAVE loop;
    statement2;
END;
```

Notice that by using the LEAVE it is possible to describe a loop whose number of iterations depends on a logic variable (*condition* in this case). It is only important that the FOR statement itself have extents which are constants or meta-variables.

A.3.7 Routines and Routine Calls

BDSYN has no concept of a main routine, per se. At runtime, BDSYN routines are classified as main routines or subroutines depending on whether or not they called by another routine. BDSYN allows many main routines in the same model, and these routines can be viewed as running in parallel. It is an error in BDSYN for two disjoint routines (one is not called by the other) to set the same global variable or primary output of the block. Two disjoint routines may call the same subroutine, however. It is illegal to have recursive routine calls.

Subroutines may return a single variable (regular or meta) to the calling routine. A subroutine that returns a value is declared by placing a bit subscript after the routine name in the ROUTINE declaration. The bit subscript specifies the width of the return value or if the value is a meta-variable. If the subscript is missing, the routine returns no value. Parameters passed to subroutines must be declared as part of the ROUTINE declaration. Parameter passing has the same semantics as assignment statements (described below).

A common error in BDSYN is to have a subroutine which is not called by another routine. This causes BDSYN to treat the subroutine as a main routine.

A.3.8 Assignment Statements

Assignment statements are written in the form:

```
    variable = expression
```

They result is that the variable receives the value of the expression. If the width of the variable is wider than the width of the evaluated expression, extra zeros will be added to pad the high bits of the variable. If the expression is wider than the variable, the high bits are truncated. If the variable is a meta-variable the expression may contain only constants and other meta-variables.

It is possible to assign a value to the same variable twice in the same program. BDSYN will treat this situation in the same manner as any sequential programming language. In particular the following works correctly.

```
    x = 1;   y = x;   x = 0;
    IF x NEQ y THEN is_ok = 1;
```

Section A.4 contains further details on multiple assignment.

A.3.9 Expressions

Legal expressions are of the following form. They are listed in order of precedence, and expressions higher on the list are evaluated first. The curly braces in the ZXT, OXT, and SXT expressions represent actual characters in the input, not a repeatable phrase.

```
    |   number                            ! number as described above
    |   constant                          ! a declared constant
    |   meta_variable
    |   regular_variable
    |   (expression)                      ! parenthesized expression
    |   expression<expression>            ! choose a bit
    |   expression<const_expr:const_expr>         ! choose several bits
    |   expression & expression           ! concatenation
    |   expression SRO expression         ! shift right filling with zeros
```

```
|   expression SR1 expression        ! shift right filling with ones
|   expression SRR expression        ! rotate right
|   expression SL0 expression        ! shift left filling with zeros
|   expression SL1 expression        ! shift left filling with ones
|   expression SLR expression        ! rotate left
|   expression * expression          ! multiplication
|   const_expr / const_expr          ! integer division
|   const_expr MOD const_expr        ! mod operation
|   expression + expression          ! addition
|   expression - expression          ! subtraction
|   expression EQL expression        ! Equal comparison
|   expression NEQ expression        ! Not equal comparison
|   expression LSS expression        ! Less than comparison
|   expression LEQ expression        ! Less than equal comparison
|   expression GTR expression        ! Greater than comparison
|   expression GEQ expression        ! Greater than equal comparison
|   BUF expression                   ! null operation
|   NOT expression                   ! bitwise NOT
|   expression AND expression        ! bitwise AND
|   expression NAND expression       ! bitwise NAND
|   expression OR expression         ! bitwise OR
|   expression NOR expression        ! bitwise NOR
|   expression XOR expression        ! bitwise XOR
|   expression EQV expression        ! bitwise XNOR
|   WIDTH expression                 ! return number of bits
|   ZXT {WIDTH = const_expr} expression      ! zero extend
|   OXT {WIDTH = const_expr} expression      ! one extend
|   SXT {WIDTH = const_expr} expression      ! sign extend
```

In the above "const_expr" represents a constant expression. A constant expression is an expression that contains only numbers, declared constants, and meta-variables. It may not contain regular variables.

The single bit selection operator, <0>, may take an arbitrary expression as the bit selector. When the bit selector is not a constant, a multiplexor is implied. The multiple bit selection operator, <31:0>, requires that both expressions be a constant expression and that the first number is larger than the second. The concatenation operator '&' joins two bit vectors into one. The expression before the '&' becomes the high order bits, the trailing expression becomes the low order bits.

The shift operations shift the first expression by an amount given by the second expression. When the shift amount is a variable expression, a barrel shifter is implied. Note: the shift operation does not change the width of the shifted expression. This means it is possible to shift bits off the end of the expression accidentally. Care should be taken to extend the expression (using ZXT) so no bits are lost.

'+', '-', '*', '/', and MOD implement integer arithmetic and are supported for constant expressions. In addition, '+', '-', and '*' are supported for variable expressions. '+' and '-' imply a combinational adder in this case, and '*' implies a full combinational multiplier. The '*' operation should be used sparingly on variable expressions since it implies so much logic. In particular, '*' should not be used where a shift would be sufficient.

The comparison operators evaluate to a single bit 1 if the condition is true and a single bit 0 if the condition is false. If an expression in the comparison is a variable, a comparator is implied. The bitwise boolean operators take two bit vectors and perform the desired operation bit by bit. If one of the expressions is smaller than the other, it will be automatically zero extended. The WIDTH operator returns a constant which is the number of bits in the given expression. ZXT, OXT, and SXT will add more significant bits to the given expression to make it the given width. It is an error to specify a width that is smaller than the given expression.

A.3.10 Macro Definitions and Required Files

BDSYN has the facility for defining macro definitions in the input description. Macro definitions may appear anywhere in the input description and have the following format:

```
MACRO macro_name = macro_body $ENDMACRO
```

Following a macro definition, any use of *macro_name* will be replaced by the *macro_body*. The *macro_body* is expanded exactly as typed (including comments). It may span more than one line and may contain uses of other macro definitions.

BDSYN also allows distinct input files to be included within other files. This is done using the REQUIRE command. A REQUIRE may occur anywhere in a BDSYN file and has the syntax:

```
REQUIRE 'filename.ext';
```

The text of the required file will be inserted in place of the REQUIRE command.

A.4 Multiple Assignment

As mentioned in the introduction, BDSYN's input descriptions appear very sequential in nature yet are intended to describe combinational logic. We feel that our form of sequential description gives the user a great deal of power and flexibility in specifying logic. The translation of sequential description to combinational logic is accomplished by correctly handling "multiple assignment."

Our definition of multiple assignment is the correct and consistent handling of many sequential assignments to the same variable. For example:

```
x = default_value;
IF condition THEN x = new_value;
```

describes a multiplexer which is controlled by "condition." The above example could be rewritten in a more conventional way:

```
IF condition THEN
    x = default_value
ELSE
    x = new_value;
```

At first glance, one might think that multiple assignment is not particularly useful. In fact, multiple assignment is the heart of BDSYN's ability to process sequential descriptions. Several examples of this can be found in the "Examples" section at the end of this guide.

A.5 Unspecified Variables

In BDSYN input descriptions, as in any programming language, it is possible to have variables which are not always defined. For instance, in the following code:

```
IF cond THEN
    x = expr;
```

the variable x, if it has not previously been assigned to, is unspecified when *cond* is false.

Under normal circumstances BDSYN will assume that when variables are unspecified, they should have the value zero. This assumption is probably not a good one. In some cases, the unspecified variables could be caused by a mistake in the user specification. In other cases the user may not care about a variable value under certain conditions. In this case the user should probably specify DONT_CARE conditions as discussed in section A.8.

In general, it is very difficult for BDSYN to detect the use of unspecified variables. However, unspecified variables may be detected by using MIS. First, BDSYN should be run using the -n flag. This will cause BDSYN to create a group of variables that end with the characters "**". (Also, the ".inputs" line in the BLIF output will not be printed.) The BLIF file containing the "**" variables should then be read into MIS and minimized. After minimization, if any of the "**" variables fan out to any gates, this corresponds to the use of an unspecified variable.

A.6 Meta-variables

Meta-variables have a very separate use from logic variables. They can be used within constant expressions because they represent constant values. Unlike logic variables, they are guaranteed to always assume the same particular value. The value of a logic variable may change based on the primary inputs to the BDSYN combinational block.

Because BDSYN is able to statically evaluate the value of meta-variables, it is able to perform some simple logic minimization when meta-variables are involved. In the following example, function1 and function2 are complex functions. (*i* is a meta-variable.)

```
FOR i FOR 0 TO 5 DO
    IF i MOD 2 EQL 0 THEN
        x = function1(x)
    ELSE
        x = function2(x);
```

After BDSYN minimization this becomes:

```
x = function1(x);
x = function2(x);
x = function1(x);
x = function2(x);
x = function1(x);
x = function2(x);
```

Had the condition in the IF statement above depended on a logic variable, no such minimization would have been possible inside of BDSYN. This is because the value of a logic variable is generally unpredictable.

One non-obvious problem with meta-variables is the following. A meta-variable is defined within a branching structure (i.e. IF, SELECT, etc.). If the condition of the branching structure depends on a logic variable, the meta-variable is not correctly defined upon exit from that structure. For example:

```
meta = 3;
IF x EQL 1 THEN BEGIN
    meta = 4;
    a<0> = w<meta>;       ! This works correctly
END
a<1> = w<meta>;           ! This uses meta = 4 even if ( x NEQ 1 )
```

This problem stems from indirectly trying to define the meta-variable *meta* in terms of the logic variable *x*. For similar reasons, meta-variables that are used for loop counts are not necessarily correct upon exit from the loop. For example:

```
loop: FOR i FROM 1 TO 10 DO BEGIN
    x = function(x);
    IF x EQL 1 THEN LEAVE loop;
END;
y = i;                 ! y gets the value 10 regardless of the function
```

A.7 Complex Operators

BDSYN supports several operations which can not be implemented by just one or two gates. These include '+', '-', '*', LEQ, GEQ, LSS, GTR, shifts, and single bit selection when they are operating on non-constants. The operations imply complex structures such as adders, multipliers, comparators, barrel shifters, and multiplexors. The complex operators are implemented through a set of library routines found in the file "bdsyn.lib". The macros in the library are used to create routines of an appropriate size (a 3 bit adder, or a 4 bit comparator, etc) which are then inserted into the code.

A.8 Don't Care Conditions

It is often the case that it makes no difference what the value of an output is. For example, in a finite state machine with seven states encoded in three state bits, state 7 never occurs:

```
SELECT state<2:0> FROM
    [0,1,2,3]: output = 0;
    [4,5,6]: output = 1;
ENDSELECT;
```

Here, the output for state 7 would be unspecified. To resolve this, a default value for output might be added.

```
output = 0;
SELECT state<2:0> FROM
    [0,1,2,3]: output = 0;
    [4,5,6]: output = 1;
ENDSELECT;
```

The problem is that this is unnecessarily restrictive, and does not allow for all of the potential logic simplification. In actuality, it may make no difference whether "output = 0" or "output = 1" in state 7, because state 7 never occurs. For this reason, BDSYN has the special variable DONT_CARE, which will allow logic optimization tools to choose the optimal value for output. DONT_CARE is used as any other variable:

```
output = DONT_CARE;
SELECT state<2:0> FROM
    [0,1,2,3]: output = 0;
    [4,5,6]: output = 1;
ENDSELECT;
```

Inside of BDSYN, assignments from this variable will automatically be extended to fit the size of the destination. When DONT_CARE is used, it will automatically be added to the input port list in BDSYN's BLIF output. There is no need to declare the DONT_CARE variable.

The current version of MIS can not process the don't care conditions produced by BDSYN. This is because don't care specifications in multiple level descriptions can be ambiguous in their meaning. Until a uniform method for handling multiple level don't cares is implemented, it is still possible to use ESPRESSO to make partial use of the don't care specification. This can be accomplished by the following (in UNIX):

```
bdsyn input_descr | mis -c clp -T pla | \
        espresso -Dmapdc | espresso > output.pla
```

The resulting PLA can then be read into MIS.

If a description has been written that includes a don't care specification and the use of the don't care specification is not desired, the -z option should be used. The -z option maps every use of DONT_CARE to zero. The effect will be that in places where DONT_CARE is specified, the result will be forced to the value zero.

A.9 Examples

Figures A.1 through A.5 contain five examples of various BDSYN descriptions. The first is a generic finite state machine. It merely generates the next state and control signals based on the present state. The next three descriptions implement the exact same piece of logic. They are included to demonstrate the flexibility of BDSYN. Note that the second implementation (COUNT2) uses intermediate variables *seenZero* and *seenTrailing* as state holders. The variables do not really imply latches to hold the state. They are simply a convenient way to describe an idea.

The fifth example is a complicated decoder circuit. Particular attention should be given to the last statement of the main routine. It illustrates an important aspect of BDSYN. The decoder is described as a barrel shifter which shifts a single bit 1 onto the correct word line. This is obviously a very inefficient way to build a decoder. It is important to note, however, that BDSYN and MIS are capable of reducing the hardware to a more sensible implementation. There is generally no penalty associated with a quick and dirty shortcut description. The logic that is generated is the same.

```
! The classic traffic light controller finite state machine from
! Mead and Conway, Introduction to VLSI Technology, page 85
MODEL traffic_light
    hl<1:0>,              ! control for highway light
    fl<1:0>,              ! control for farm light
    st<0>,                ! to start the interval timer
    nextState<1:0> =
    c<0>,                 ! indicating a car on the farm road
    ts<0>,                ! timeout of short interval timer
    tl<0>,                ! timeout of long interval timer
    presentState<1:0> ;
! state assignments
CONSTANT HG = 0, HY = 2, FG = 3, FY = 1;
! symbolic output assignments
CONSTANT GREEN = 0, RED = 1, YELLOW = 2;

ROUTINE traffic_light_controller;
    ! set up default outputs (use of multiple assignment)
    nextState = presentState;
    st = 0;

    SELECT presentState FROM
        [HG]: BEGIN
            hl = GREEN;   fl = RED;
            IF c AND tl THEN BEGIN
                nextState = HY;
                st = 1;
            END;
          END;
        [HY]: BEGIN
            hl = YELLOW;   fl = RED;
            IF ts THEN BEGIN
                nextState = FG;
                st = 1;
            END;
          END;
        [FG]: BEGIN
            hl = RED;   fl = GREEN;
            IF NOT c or tl THEN BEGIN
                nextState = FY;
                st = 1;
            END;
          END;
        [FY]: BEGIN
            hl = RED;   fl = YELLOW;
            IF ts THEN BEGIN
                nextState = HG;
                st = 1;
            END;
          END;
    ENDSELECT;
ENDROUTINE;
ENDMODEL;
```

Figure A.1: Traffic Light Controller

```
! COUNT1 (Count Zeros):

! The input is an 8 bit binary string expected to consist of
! a string of 1's, a string of 0's, and a string of 1's.  Any
! of these strings may be zero length.
! For example, '11000111', '10000001', and '11110000' are all
! valid strings, and '10001100' is a mal-formed string.

! The problem is to design a circuit which, given the 8 bit
! string, returns the count of number of consecutive zeros in
! the string, or returns an error condition if the string is
! mal-formed.  The output is undefined if the input string is
! mal-formed.

MODEL count1 error<0>, out<3:0> = in<7:0>;

! find first bit matching 'val' (return index of the bit)
ROUTINE ff<3:0>(x<7:0>, val<0>);
    STATE i<>;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL val THEN
            RETURN i;
    RETURN 8;
ENDROUTINE;

ROUTINE main;
    STATE x<7:0>;

    ! Shift off the first string of 1's
    x = in SR1 ff(in, 0);

    ! Count is where the first 1 is
    out = ff(x, 1);

    ! Check for a mal-formed string:
    ! shift off 0's, and check for any more 0's
    x = x SR1 out;
    error = ff(x, 0) NEQ 8;
    IF error THEN
        out = DONT_CARE;

ENDROUTINE;
ENDMODEL;
```

Figure A.2: Count1

```
! COUNT2 (Count Zeros)
! A different implementation for the last example.

MODEL count2 error<0>, out<3:0> = in<7:0>;
CONSTANT TRUE = 1, FALSE = 0;

ROUTINE legal<0>(x<7:0>);
    STATE i<>;
    STATE seenZero;          ! there has been a zero
    STATE seenTrailing;      ! we have hit the trailing ones field

    seenZero = FALSE;
    seenTrailing = FALSE;
    FOR i FROM 0 TO 7 DO
        IF seenTrailing AND (x<i> EQL 0) THEN
            RETURN FALSE                         ! Illegal string
        ELSE IF seenZero AND (x<i> EQL 1) THEN
            seenTrailing = TRUE
        ELSE IF x<i> EQL 0 THEN
            seenZero = TRUE;

    ! If we made it to here then the input is legal
    RETURN TRUE;
ENDROUTINE;

ROUTINE zeros<3:0>(x<7:0>);
    STATE i<>, count<3:0>;
    count = 0;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL 0 THEN
            count = count + 1;
    RETURN count;
ENDROUTINE;

ROUTINE main;
    IF legal(in) THEN BEGIN
        error = FALSE;
        out = zeros(in);
    END ELSE BEGIN
        error = TRUE;
        out = DONT_CARE;
    END;
ENDROUTINE;
ENDMODEL;
```

Figure A.3: Count2

```
! COUNT3 (Count Zeros)
! Yet another implementation

MODEL count3 error<0>, out<3:0> = in<7:0>;
CONSTANT TRUE = 1, FALSE = 0;

! check for a legal string
ROUTINE legal<0>(x<7:0>);
    STATE i<>, j<>, k<>;

    ! Look for a ...0...1...0...
    FOR i FROM 0 to 5 DO
        FOR j FROM i+1 to 6 DO
            FOR k FROM j+1 to 7 DO
                IF (x<i> EQL 0) AND (x<j> EQL 1) AND
                                    (x<k> EQL 0) THEN
                    ! The input is illegal
                    RETURN FALSE;

    ! If we made it to here then the input is legal
    RETURN TRUE;
ENDROUTINE;


ROUTINE zeros<3:0>(x<7:0>);
    STATE i<>, count<3:0>;
    count = 0;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL 0 THEN
            count = count + 1;
    RETURN count;
ENDROUTINE;

ROUTINE main;
    IF legal(in) THEN BEGIN
        error = FALSE;
        out = zeros(in);
    END ELSE BEGIN
        error = TRUE;
        out = DONT_CARE;
    END;
ENDROUTINE;
ENDMODEL;
```

Figure A.4: Count3

```
! Register decoder for the SPUR cpu chip.
! The register file of the SPUR cpu uses a Berkeley RISC
! type register windowing scheme.  The register file is
! divided into 8 overlapping register "windows".  Each
! register window can access 32 of SPUR's 138 registers.
! The first 10 registers of all of the windows point to
! the same 10 "global" registers.  The next 6 registers
! are shared with the top six registers of the previous
! window.  The 6 top registers of each window are similarly
! shared with the next window.  The last window wraps
! around and shares registers with the first window.
MODEL reg_decode
    addr<137:0>     ! one-hot decoded output (word lines)
=
    cwp<2:0>,       ! Current window pointer
    reg<4:0>;       ! Current register in the window

CONSTANT
    NUMREGS = 138,      ! total number of registers
    NUMGLOBALS = 10;    ! number of global registers

ROUTINE REG_DECODE();
    STATE
        sum<7:0>,       ! temporary variable
        decAddr<7:0>;   ! decoded address (number of the word line)

    ! calculate the correct register number
    IF (reg LSS NUMGLOBALS) THEN
        ! point to the global registers
        decAddr = regSpec

    ELSE BEGIN
        sum = (cwp & 0000#2) + regSpec;

        ! Check for wrap around.  If so, need to subtract an offset
        IF (sum GTR (NUMREGS - 1)) THEN
            sum = sum - (NUMREGS - NUMGLOBALS);

        decAddr = sum;
    END;

    ! Do the actual decoding (*** See main text for details ***)
    addr = (ZXT {width=138} 1) SLO decAddr;

ENDROUTINE;
ENDMODEL;
```

Figure A.5: Register file decoder

Appendix B

Processing of a Bdsyn Example

The following is an example of the processing done by BDSYN. While the actual input example is somewhat contrived, it illustrates almost all of the aspects of BDSYN. On the folling pages results of each BDSYN processing step are given in a BDS description format. This is not terribly misleading, since BDSYN actual does store its intermediate results in a BDS parse tree. The example program follows. The function of the logic is to take a four bit input vector *in* and rotate it until its low bit is a 1 (one). If none of the bits are 1, an the flag *error* is set.

```
MODEL rotate out<3:0>, error = in<3:0>;

ROUTINE main;
    STATE i<>, zeros<2:0>;

    ! find the number of trailing zeros
    block: BEGIN
        FOR i FROM 0 TO 3 DO
            IF in<i> EQL 1 THEN BEGIN
                zeros = i;
                LEAVE block;
            END;
        zeros = 4;
    END;

    ! rotate by correct amount
    out = in SRR zeros<1:0>;
    error = zeros EQL 4;
ENDROUTINE;
ENDMODEL;
```

After the parse stage of BDSYN a single parse tree has been generated (corresponding to the single routine called *main*).

```
block: BEGIN
    FOR i FROM 0 TO 3 DO
        IF in<i> EQL 1 THEN BEGIN
            zeros = i;
            LEAVE block;
        END;
    zeros = 4;
END;
out = in SRR zeros<1:0>;
error = zeros EQL 4;
```

The inline processing stage has very little to do, since there is only one routine. It does, however, give unique names to *main*'s local variables. This avoids name clashes with global variables (and local variables of other routines).

```
block: BEGIN
    FOR main_0/i FROM 0 TO 3 DO
        IF in<main_0/i> EQL 1 THEN BEGIN
            main_0/zeros = main_0/i;
            LEAVE block;
        END;
    main_0/zeros = 4;
END;
out = in SRR main_0/zeros<1:0>;
error = main_0/zeros EQL 4;
```

The meta processing stage unrolls the FOR loop and substitutes constants for the meta-variable *main_0/i*.

```
block: BEGIN
    IF in<0> EQL 1 THEN BEGIN
        main_0/zeros = 0;
        LEAVE block;
    END;
    IF in<1> EQL 1 THEN BEGIN
        main_0/zeros = 1;
        LEAVE block;
    END;
    IF in<2> EQL 1 THEN BEGIN
        main_0/zeros = 2;
        LEAVE block;
    END;
    IF in<3> EQL 1 THEN BEGIN
        main_0/zeros = 3;
        LEAVE block;
    END;
    main_0/zeros = 4;
END;
out = in SRR main_0/zeros<1:0>;
error = main_0/zeros EQL 4;
```

The next stage of processing, complex, detects that the function SRR is a complex operator and performs two tasks. It calls the parser to create a shifting routine called *$SRR_4_2* from the library macro *$SRR*.

```
ROUTINE $SRR_4_2( a<3:0>, b<1:0> );
    STATE pow<>, i<>;

    pow = 1
    FOR i FROM 0 TO 1 DO BEGIN
        IF b<i> THEN a = a SRR pow;
        pow = pow * 2;
    END;
    RETURN a;
ENDROUTINE;
```

It then replaces the second to last line of *main* with:

```
out = $SRR_4_2( in, zeros);
```

BDSYN, detecting that complex has inserted a subroutine, then goes back to the inline step. After inline and meta have been repeated, the following results. Notice that *pow* disappears because it is a meta-variable.

```
block: BEGIN
    IF in<0> EQL 1 THEN BEGIN
        main_0/zeros = 0;
        LEAVE block;
    END;
    IF in<1> EQL 1 THEN BEGIN
        main_0/zeros = 1;
        LEAVE block;
    END;
    IF in<2> EQL 1 THEN BEGIN
        main_0/zeros = 2;
        LEAVE block;
    END;
    IF in<3> EQL 1 THEN BEGIN
        main_0/zeros = 3;
        LEAVE block;
    END;
    main_0/zeros = 4;
END;
$SRR_4_2/a = in;
$SRR_4_2/b = main_0/zeros<1:0>;
IF $SRR_4_2/b<0> EQL 1 THEN
    $SRR_4_2/a = $SRR_4_2/a SRR 1;
IF $SRR_4_2/b<1> EQL 1 THEN
    $SRR_4_2/a = $SRR_4_2/a SRR 2;
$$SRR_4_2 = $SRR_4_2/a;
out = $$SRR_4_2;
error = main_0/zeros EQL 4;
```

The next phase of processing is evaluation. The only things in the given input that can be reduced at this stage are the SRR and EQL operations. From this point on, I will shorten the names of the variables to save space and improve readability.

```
block: BEGIN
    IF in<0> THEN BEGIN
        zeros = 0;
        LEAVE block;
    END;
    IF in<1> THEN BEGIN
        zeros = 1;
        LEAVE block;
    END;
    IF in<2> THEN BEGIN
        zeros = 2;
        LEAVE block;
    END;
    IF in<3> THEN BEGIN
        zeros = 3;
        LEAVE block;
    END;
    zeros = 4;
END;
a = in;
b = zeros<1:0>;
IF b<0> THEN
    a<3> = a<0>; a<0> = a<1>; a<1> = a<2>; a<2> = a<3>;
IF b<1> THEN
    a<3> = a<1>; a<0> = a<2>; a<1> = a<3>; a<2> = a<0>;
$$SRR_4_2 = a;
out = $$SRR_4_2;
error = zeros<2> AND NOT zeros<1> AND NOT zeros<0>;
```

After evaluation is complete, leave processing is done. Notice how consecutive LEAVEs become nested IFs.

```
IF in<0> THEN
    zeros = 0
ELSE IF in<1> THEN
    zeros = 1
ELSE IF in<2> THEN
    zeros = 2
ELSE IF in<3> THEN
    zeros = 3
ELSE
    zeros = 4;
a = in;
b = zeros;
IF b<0> THEN
    a<3> = a<0>; a<0> = a<1>; a<1> = a<2>; a<2> = a<3>;
IF b<1> THEN
    a<3> = a<1>; a<0> = a<2>; a<1> = a<3>; a<2> = a<0>;
$$SRR_4_2 = a;
out = $$SRR_4_2;
error = zeros<2> AND NOT zeros<1> AND NOT zeros<0>;
```

The next stage, premult does not change the parse tree. It merely records data. Folloing premult, multiple assignment assigns revision numbers and reduces the

IF statements. Multiple-bit vector assignments are also, by necessity, expanded to many single-bit assignments.

```
zeros<2>1.1 = 0; zeros<1>1.1 = 0; zeros<0>1.1 = 0;
zeros<2>3.1 = 0; zeros<1>3.1 = 0; zeros<0>3.1 = 1;
zeros<2>5.1 = 0; zeros<1>5.1 = 1; zeros<0>5.1 = 0;
zeros<2>7.1 = 0; zeros<1>7.1 = 1; zeros<0>7.1 = 1;
zeros<2>8.1 = 1; zeros<1>8.1 = 0; zeros<0>8.1 = 0;

zeros<2>6.1 = (in<3> AND zeros<2>7.1) OR (NOT in<3> AND zeros<2>8.1);
zeros<1>6.1 = (in<3> AND zeros<1>7.1) OR (NOT in<3> AND zeros<1>8.1);
zeros<0>6.1 = (in<3> AND zeros<0>7.1) OR (NOT in<3> AND zeros<0>8.1);

zeros<2>4.1 = (in<2> AND zeros<2>5.1) OR (NOT in<2> AND zeros<2>6.1);
zeros<1>4.1 = (in<2> AND zeros<1>5.1) OR (NOT in<2> AND zeros<1>6.1);
zeros<0>4.1 = (in<2> AND zeros<0>5.1) OR (NOT in<2> AND zeros<0>6.1);

zeros<2>2.1 = (in<1> AND zeros<2>3.1) OR (NOT in<1> AND zeros<2>4.1);
zeros<1>2.1 = (in<1> AND zeros<1>3.1) OR (NOT in<1> AND zeros<1>4.1);
zeros<0>2.1 = (in<1> AND zeros<0>3.1) OR (NOT in<1> AND zeros<0>4.1);

zeros<2>0.1 = (in<0> AND zeros<2>1.1) OR (NOT in<0> AND zeros<2>2.1);
zeros<1>0.1 = (in<0> AND zeros<1>1.1) OR (NOT in<0> AND zeros<1>2.1);
zeros<0>0.1 = (in<0> AND zeros<0>1.1) OR (NOT in<0> AND zeros<0>2.1);

a<3>0.1 = in<3>;  a<2>0.1 = in<2>;  a<1>0.1 = in<1>;  a<0>0.1 = in<0>;
b<1>0.1 = zeros<1>0.1;  b<0>0.1 = zeros<0>0.1;

a<3>9.1 = a<0>0.1;  a<0>9.1 = a<1>0.1;
a<1>9.1 = a<2>0.1;  a<2>9.1 = a<3>0.1;

a<3>0.2 = (b<0>0.1 AND a<3>9.1) OR (NOT b<0>0.1 AND a<3>0.1);
a<2>0.2 = (b<0>0.1 AND a<2>9.1) OR (NOT b<0>0.1 AND a<2>0.1);
a<1>0.2 = (b<0>0.1 AND a<1>9.1) OR (NOT b<0>0.1 AND a<1>0.1);
a<0>0.2 = (b<0>0.1 AND a<0>9.1) OR (NOT b<0>0.1 AND a<0>0.1);

a<3>11.1 = a<1>0.2;  a<0>11.1 = a<2>0.2;
a<1>11.1 = a<3>0.2;  a<2>11.1 = a<0>0.2;

a<3>0.3 = (b<1>0.1 AND a<3>11.1) OR (NOT b<1>0.1 AND a<3>0.2);
a<2>0.3 = (b<1>0.1 AND a<2>11.1) OR (NOT b<1>0.1 AND a<2>0.2);
a<1>0.3 = (b<1>0.1 AND a<1>11.1) OR (NOT b<1>0.1 AND a<1>0.2);
a<0>0.3 = (b<1>0.1 AND a<0>11.1) OR (NOT b<1>0.1 AND a<0>0.2);

$$SRR_4_2<3>0.1 = a<3>0.3;  $$SRR_4_2<2>0.1 = a<2>0.3;
$$SRR_4_2<1>0.1 = a<1>0.3;  $$SRR_4_2<0>0.1 = a<0>0.3;

out<3> = $$SRR_4_2<3>0.1;  out<2> = $$SRR_4_2<2>0.1;
out<1> = $$SRR_4_2<1>0.1;  out<0> = $$SRR_4_2<0>0.1;

error = zeros<2>0.1 AND NOT zeros<1>0.1 AND NOT zeros<0>0.1;
```

Cleanup is the next step in the BDSYN processing. Because of forward substitution of simple assignments, many assignments statements are eliminated. Also, because of forward substitution of constant assignments, much simple boolean minimization can be done.

```
zeros<2>6.1 = NOT in<3>;

zeros<2>4.1 = NOT in<2> AND zeros<2>6.1;
zeros<1>4.1 = in<2> OR (NOT in<2> AND in<3>);
zeros<0>4.1 = NOT in<2> AND in<3>;

zeros<2>2.1 = (NOT in<1> AND zeros<2>4.1);
zeros<1>2.1 = (NOT in<1> AND zeros<1>4.1);
zeros<0>2.1 = in<1> OR (NOT in<1> AND zeros<0>4.1);

zeros<2>0.1 = (NOT in<0> AND zeros<2>2.1);
zeros<1>0.1 = (NOT in<0> AND zeros<1>2.1);
zeros<0>0.1 = (NOT in<0> AND zeros<0>2.1);

a<3>0.2 = (zeros<0>0.1 AND in<0>) OR (NOT zeros<0>0.1 AND in<3>);
a<2>0.2 = (zeros<0>0.1 AND in<3>) OR (NOT zeros<0>0.1 AND in<2>);
a<1>0.2 = (zeros<0>0.1 AND in<2>) OR (NOT zeros<0>0.1 AND in<1>);
a<0>0.2 = (zeros<0>0.1 AND in<1>) OR (NOT zeros<0>0.1 AND in<0>);

a<3>0.3 = (zeros<1>0.1 AND a<1>0.2) OR (NOT zeros<1>0.1 AND a<3>0.2);
a<2>0.3 = (zeros<1>0.1 AND a<0>0.2) OR (NOT zeros<1>0.1 AND a<2>0.2);
a<1>0.3 = (zeros<1>0.1 AND a<3>0.2) OR (NOT zeros<1>0.1 AND a<1>0.2);
a<0>0.3 = (zeros<1>0.1 AND a<2>0.2) OR (NOT zeros<1>0.1 AND a<0>0.2);

out<3> = a<3>0.3;  out<2> = a<2>0.3;  out<1> = a<1>0.3;  out<0> = a<0>0.3;
error = zeros<2>0.1 AND NOT zeros<1>0.1 AND NOT zeros<0>0.1;
```

The final stage of processing maps the internal equations to BLIF. Notice that several equations are collapsed out. The data is presented in BDS format for continuity.

```
zeros<0>0.2 = NOT in<0> AND NOT in<1> AND NOT in<2> AND NOT in<3>
zeros<1>0.2 = (NOT in<0> AND NOT in<1> AND in<2>) OR
              (NOT in<0> AND NOT in<1> AND NOT in<2> AND in<3>)
zeros<2>0.2 = (NOT in<0> AND in<1>) OR
              (NOT in<0> AND NOT in<1> AND NOT in<2> AND in<3>)

a<0>0.2 = (in<1> AND zeros<0>0.2) OR (in<0> AND NOT zeros<0>0.2)
a<1>0.2 = (in<2> AND zeros<0>0.2) OR (in<1> AND NOT zeros<0>0.2)
a<2>0.2 = (in<3> AND zeros<0>0.2) OR (in<2> AND NOT zeros<0>0.2)
a<3>0.2 = (in<0> AND zeros<0>0.2) OR (in<3> AND NOT zeros<0>0.2)

a<0>0.3 = (a<2>0.2 AND zeros<1>0.2) OR (a<0>0.2 AND NOT zeros<1>0.2)
a<1>0.3 = (a<3>0.2 AND zeros<1>0.2) OR (a<1>0.2 AND NOT zeros<1>0.2)
a<2>0.3 = (a<0>0.2 AND zeros<1>0.2) OR (a<2>0.2 AND NOT zeros<1>0.2)
a<3>0.3 = (a<1>0.2 AND zeros<1>0.2) OR (a<3>0.2 AND NOT zeros<1>0.2)

out<3> = a<3>0.3;  out<2> = a<2>0.3;  out<1> = a<1>0.3;  out<0> = a<0>0.3;
error = zeros<2>0.1 AND NOT zeros<1>0.1 AND NOT zeros<0>0.1;
```

Appendix C

Bdsyn Library

```
!------------------------------------------------------------
!
!  This library implements the macro expansions for complex
!  operations on variables.  The standard procedure is that
!  bdsyn creates the macro's
!
!       $a      the highest-bit number of the left operand
!       $b      the highest-bit number of the right operand
!       $max    the maximum of $a and $b
!       $pow    2 to the power $b - 1
!       $name   the name of the routine to be created
!
!  Then the complex operator macro is invoked which creates
!  a routine which implements the complex operation for
!  the given bit widths.
!
!------------------------------------------------------------


!-----------------------------------
!  + operator
!-----------------------------------
MACRO "$PLUS" =
ROUTINE $name<$max+1:0> ( a<$max:0>, b<$max:0> );
    STATE i<>, c<0>, sum<$max + 1:0>;

    c = 0;
    FOR i FROM 0 TO $max DO BEGIN
        sum<i> = a<i> XOR b<i> XOR c;
        c = (a<i> AND c) OR (b<i> AND c) OR (a<i> AND b<i>);
    END;
    sum<$max + 1> = c;
    RETURN sum;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  - operator
!-----------------------------------
```

```
MACRO "$MINUS" =
ROUTINE $name<$max+1:0> ( a<$max:0>, b<$max:0> );
    STATE i<>, c<0>, sum<$max+1:0>;

    b = NOT b;
    c = 1;
    FOR i FROM 0 TO $max DO BEGIN
        sum<i> = a<i> XOR b<i> XOR c;
        c = (a<i> AND c) OR (b<i> AND c) OR (a<i> AND b<i>);
    END;
    sum<$max+1> = c;
    RETURN sum;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  * operator
!-----------------------------------
MACRO "$TIMES" =
ROUTINE $name<$a+$b+1:0> ( a<$a:0>, b<$b:0> );
    STATE result<$a+$b+1:0>, i<>, pow<>;

    result = 0;
    FOR i FROM 0 TO $b DO BEGIN
        IF b<i> THEN
            result< i+$a+1 : i > = result< i+$a : i > + a;
    END;
    RETURN result;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  EQL operator (for comparison of two variables)
!-----------------------------------
MACRO "$EQL" =
ROUTINE $name<0> ( a<$max:0>, b<$max:0> );
    STATE i<>, test<0>;

    FOR i FROM $max DOWNTO 0 DO BEGIN
        test = a<i> xor b<i>;
        if test then
            return 0;
    END;
    RETURN 1;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  NEQ operator (for comparison of two variables)
!-----------------------------------
MACRO "$NEQ" =
ROUTINE $name<0> ( a<$max:0>, b<$max:0> );
    STATE i<>, test<0>;

    FOR i FROM $max DOWNTO 0 DO BEGIN
        test = a<i> xor b<i>;
        if test then
            return 1;
    END;
    RETURN 0;
ENDROUTINE $name;
$ENDMACRO;


!----------------------------------
! LEQ operator
!----------------------------------
MACRO "$LEQ" =
ROUTINE $name<0> ( a<$max:0>, b<$max:0> );
    STATE i<>, test<0>;

    FOR i FROM $max DOWNTO 0 DO BEGIN
        test = a<i> xor b<i>;
        if test then
            return b<i>;
    END;
    RETURN 1;
ENDROUTINE $name;
$ENDMACRO;


!----------------------------------
! GEQ operator
!----------------------------------
MACRO "$GEQ" =
ROUTINE $name<0> ( a<$max:0>, b<$max:0> );
    STATE i<>, test<0>;

    FOR i FROM $max DOWNTO 0 DO BEGIN
        test = a<i> xor b<i>;
        if test then
            return a<i>;
    END;
    RETURN 1;
ENDROUTINE $name;
$ENDMACRO;


!----------------------------------
! LSS operator
!----------------------------------
MACRO "$LSS" =
ROUTINE $name<0> ( a<$max:0>, b<$max:0> );
    STATE i<>, test<0>;

    FOR i FROM $max DOWNTO 0 DO BEGIN
        test = a<i> xor b<i>;
        if test then
            return b<i>;
    END;
    RETURN 0;
ENDROUTINE $name;
$ENDMACRO;
```

```
!-----------------------------------
!  GTR operator
!-----------------------------------
MACRO "$GTR" =
ROUTINE $name<0> ( a<$max:0>, b<$max:0> );
    STATE i<>, test<0>;

    FOR i FROM $max DOWNTO 0 DO BEGIN
        test = a<i> xor b<i>;
        if test then
            return a<i>;
    END;
    RETURN 0;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  SL0 operator
!-----------------------------------
MACRO "$SL0" =
ROUTINE $name<$a:0> ( a<$a:0>, b<$b:0> );
    STATE pow<>, i<>;

    pow = 1;
    FOR i FROM 0 TO $b DO BEGIN
        IF b<i> THEN a = a SL0 pow;
        pow = pow * 2;
    END;
    RETURN a;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  SL1 operator
!-----------------------------------
MACRO "$SL1" =
ROUTINE $name<$a:0> ( a<$a:0>, b<$b:0> );
    STATE pow<>, i<>;

    pow = 1;
    FOR i FROM 0 TO $b DO BEGIN
        IF b<i> THEN a = a SL1 pow;
        pow = pow * 2;
    END;
    RETURN a;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  SLR operator
!-----------------------------------
MACRO "$SLR" =
ROUTINE $name<$a:0> ( a<$a:0>, b<$b:0> );
    STATE pow<>, i<>;

    pow = 1;
    FOR i FROM 0 TO $b DO BEGIN
        IF b<i> THEN a = a SLR pow;
        pow = pow * 2;
    END;
    RETURN a;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  SR0 operator
!-----------------------------------
MACRO "$SR0" =
ROUTINE $name<$a:0> ( a<$a:0>, b<$b:0> );
    STATE pow<>, i<>;

    pow = 1;
    FOR i FROM 0 TO $b DO BEGIN
        IF b<i> THEN a = a SR0 pow;
        pow = pow * 2;
    END;
    RETURN a;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  SR1 operator
!-----------------------------------
MACRO "$SR1" =
ROUTINE $name<$a:0> ( a<$a:0>, b<$b:0> );
    STATE pow<>, i<>;

    pow = 1;
    FOR i FROM 0 TO $b DO BEGIN
        IF b<i> THEN a = a SR1 pow;
        pow = pow * 2;
    END;
    RETURN a;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  SRR operator
!-----------------------------------
MACRO "$SRR" =
ROUTINE $name<$a:0> ( a<$a:0>, b<$b:0> );
    STATE pow<>, i<>;

    pow = 1;
    FOR i FROM 0 TO $b DO BEGIN
        IF b<i> THEN a = a SRR pow;
        pow = pow * 2;
    END;
    RETURN a;
ENDROUTINE $name;
$ENDMACRO;
```

```
!-----------------------------------
!  <> single-bit select operator
!-----------------------------------
MACRO "$BIT" =
ROUTINE $name<0> ( a<$pow:0>, b<$b:0> );
    STATE i<>, pow<>;

    pow = $pow + 1;
    FOR i FROM $b DOWNTO 0 DO BEGIN
        IF b<i> THEN a< pow/2 - 1 : 0 > = a< pow - 1 : pow/2 >;
        pow = pow/2;
    END;
    RETURN a<0>;
ENDROUTINE $name;
$ENDMACRO;


!-----------------------------------
!  This is for anyone who needs %overflow, %carryin, %carryout
!  It is not currently called automatically.
!-----------------------------------
MACRO "$--" =
ROUTINE $name<$max + 2:0> ( a<$max:0>, b<$max:0>, carryin<0> );

    STATE i<>, c<$max + 1:0>, sum<$max + 2:0>;

    c<0> = carryin;
    FOR i FROM 0 TO $max DO BEGIN
        sum<i> = a<i> XOR b<i> XOR c<i>;
        c<i+1> = a<i> AND c<i> OR b<i> AND c<i> OR a<i> AND b<i>;
    END;
    sum<$max+1> = c<$max+1>;
    sum<$max+2> = c<$max+1> XOR c<$max>;

    RETURN sum;
ENDROUTINE $name;
$ENDMACRO;
```

Appendix D

Berkeley Logic Interchange Format (BLIF)

The goal of BLIF is to describe a Boolean network in textual form. A Boolean network is a representation of an arbitrary combinational logic network, and is an acyclic directed graph with a logic function attached to each node. Each node in this representation has a single output. Therefore, each net (or signal) has only a single driver, and we can therefore name either the signal or the gate which drives the signal without ambiguity.

In this section, angle-brackets surround nonterminals, and vertical bar separates choices in a pseudo-BNF style. Bdsyn uses only the *logic-gate* construct of BLIF. Bdsyn uses the the keywords *.module, .inputs, .outputs, .end,* and *.names.* The *module-reference* construct of BLIF is not implemented.

D.1 Modules

A module is declared by:

```
.module <decl-module-name>
.inputs <decl-input-list>
.outputs <decl-output-list>
<logic-gate> | <pla-reference> | <module-reference>
                .
                .
```

```
<logic-gate> | <pla-reference> | <module-reference>
.end
```

decl-module-name is a string giving the name of the module.

decl-input-list is a white-space-separated list of strings (terminated by the end of the line) giving the formal parameters for the module being declared. If this is the last or only module, then these signals can be identified as the primary inputs of the circuit. Multiple *.inputs* lines are allowed, and the lists of inputs are merely concatenated.

decl-output-list is a white-space-separated list of strings (terminated by the end of the line) giving the formal parameters for the module being declared. If this is the last or only module, then these signals can be identified as the primary outputs of the circuit. Multiple *.outputs* lines are allowed, and the lists of inputs are merely concatenated.

It is expected that a BLIF parser may allow the *.inputs* and *.outputs* statements to be optional. If they are not specified, the primary inputs can be inferred from the signals which are not the outputs of any other logic block, and the primary outputs can be inferred from the signals which are not the inputs to any other blocks. This makes the serious assumption that no primary output is used as an intermediate signal. Also, these assumptions are only useful for the "root" module. Subcircuits are invoked with positional arguments and hence the specified order for the inputs is significant in a subcircuit.

The keyword *.end* is optional, and is implied at the end of the file.

Anywhere in the file, a '\' (backslash) as the last character on a line indicates concatenation of the subsequent line to the current line.

D.2 Logic Gates

A *logic-gate* is:

```
.names <in-1> <in-2> ... <in-n> <output>
```

```
<single-output-cover>
```

single-output-cover is an n-input, 1-output PLA description of the logic function corresponding to the logic gate. {0, 1, –} is used in the "input plane" and {0, 1, –, ~} is used in the "output plane". The logic gate can have only its ON-set specified with 1's in the "output plane", or can also have a DC-set and an OFF-set specified with -'s or 0's in the "output plane". ~ means no connection. For a more complete description of the PLA input format, see *espresso(5)*.

D.3 Module (subcircuit) references

A *module-reference* is

```
    .subckt <module-name> <input-list> <output-list>
```

module-name gives the name of a module being included. It must be previously defined in the file. *input-list* are the actual parameters and are matched one-to-one with the corresponding *decl-input-list* inside the module declaration. Likewise, *output-list* are the actual parameters and are matched one-to-one with the corresponding *decl-output-list* inside the module declaration.

input-list and *output-list* are white-space separated sequence of tokens ending with the end of the line. A "/" may be used to separate *input-list* from *output-list*. The end of *input-list* and the start of *output-list* is, however, always inferred from the number of inputs and outputs in the module declaration (i.e., from *decl-input-list* and *decl-output-list* of the module which is being called).

If both *input-list* and *output-list* are not given, then the formal parameters within the module declaration become the actual parameters. It is an error if the *input-list* and *output-list* are given and the number of inputs or outputs does not match the "call" and the definition.

D.4 PLA References

A *pla-reference* is

```
.pla <pla-name> <input-list> <output-list>
```

pla-name gives the name of a file containing a PLA description in the format understood by *espresso(5)*. *input-list* and *output-list* are matched 1 for 1 with the inputs and outputs of the PLA (which, for arcane reasons are given via .ilb and .ob keywords within the PLA). Similar to *.subckt*, a "/" may separate the input-list and the output-list and if both are missing, the formal parameters become the actual parameters.

Appendix E

Bdsim Transistor Merging

As explained in section 3.6.2 of the main text, bdsim merges two networks together by adding, in series or in parallel, a small network to a larger one. This makes six different types of merges possible: adding a transistor in parallel; adding two parallel transistors in parallel; adding two series transistors in parallel; adding a transistor in series; adding two parallel transistors in series; and adding two series transistors in series. Figures E.2 and E.3 contain pictures representing all of the possible merges that bdsim can perform.

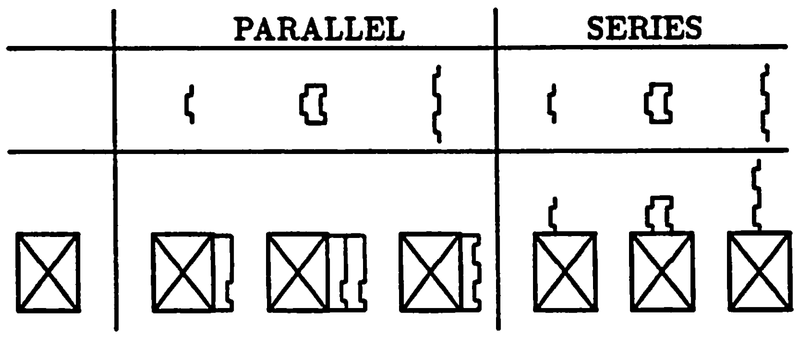


Figure E.1: Six Different Types of Merges

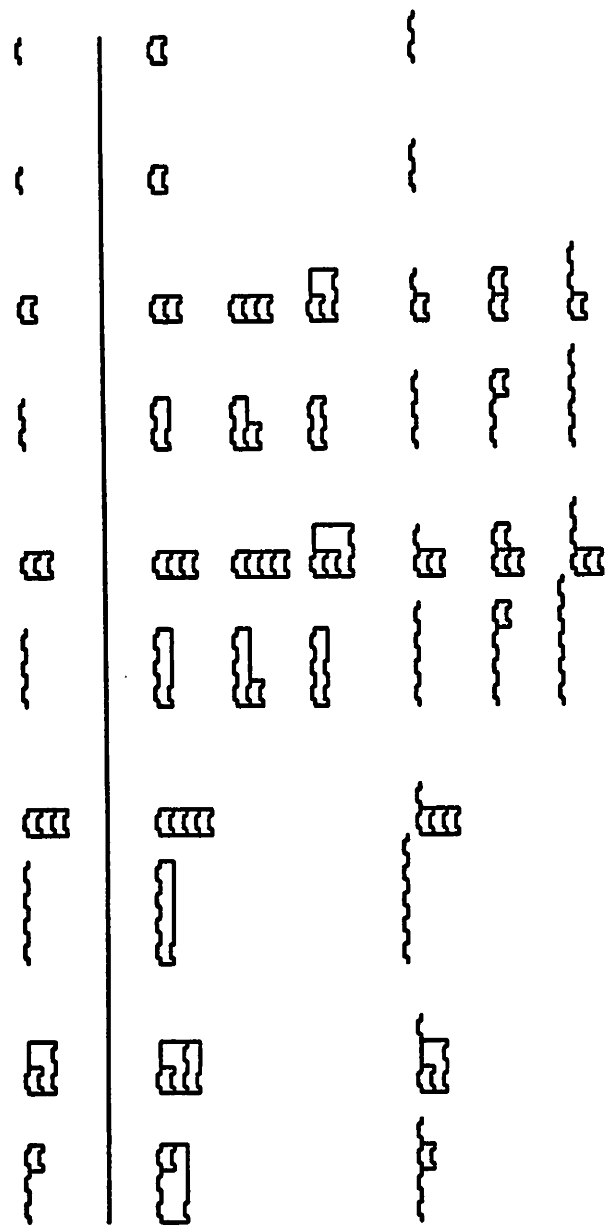


Figure E.2: Merging of Transistor Networks in Bdsim

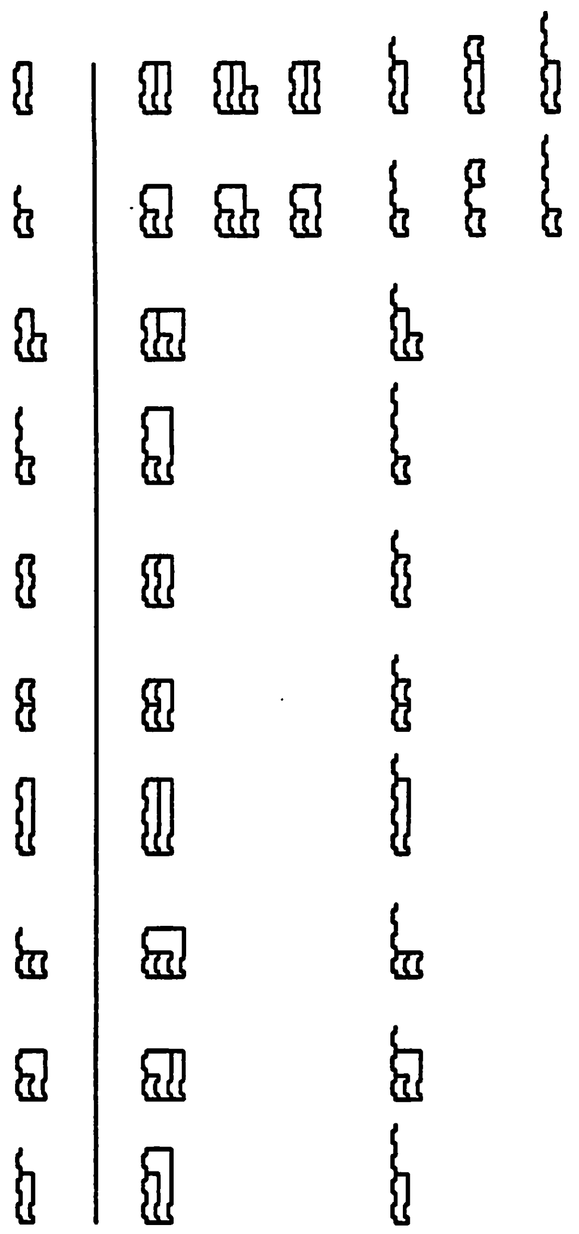


Figure E.3: Merging of Transistor Networks in Bdsim (continued)

Appendix F

Manual Pages

Following are the UNIX manual pages for the programs on which I have worked at Berkeley. They are as follows:

bdnet A program to parse a user supplied net list of instances and create an appropriate OCT file. Bdnet also has provisions for producing net lists from existing OCT cells.

bdsim Zero-one simulator as described in the text of this report.

bdsyn Textual description to logic network translator as described in the text of this report.

octflat Program to produce a flat representation of an OCT hierarchy.

NAME

bdnet – Net-list to Oct Translator

SYNTAX

bdnet [filename]
bdnet -n[c] cellname[:viewname]
bdnet -i[s] cellname[:viewname]

DESCRIPTION

Bdnet is a translator which reads a net-list format and generates an Oct cell. The Oct cell is created to the Oct Symbolic Policy Specification. An extra degree of freedom allows general properties and Oct objects to be created and placed in either the contents or interface facet of the cell. Constructs allow for the automatic attachment of actual terminals on an instance, and for the creation of arrays of instances.

The command line options are:

-n The complete net-list is written to standard output, representing in detail the Oct cell which has been created. (This output is suitable as input to a subsequent run of bdnet.) The resulting net-list shows all connections explicitly.

-c If this option is selected along with **-n** instances in the "CONNECTORS" bag will be displayed. The default is to only show instances that are in the "INSTANCES" bag.

-i Prints an inverted net-list which lists all connections to each net. An inverted net-list is written to standard output. The format for the inverted net-list is:

```
        net_name: inst_name/term_name(s) formal_term_name(s) ...
```

with a single line per net.

-s If this option is selected along with **-i** then only nets with single connections or no connections are listed. This is useful in finding incomplete connections.

SYNTAX

The following is a description of the input format in a pseudo-BNF style. Uppercase is used for keywords, lowercase is used for productions, vertical bars separate a choice of items, braces indicate an optional item, and an asterisk indicates that an item may be repeated zero or more times.

The net-list is described as follows:

```
MODEL model_name ;
{policy_props}*
{terminal_type {formal_terminal_spec}* ;}*
{instance | array_of_instance}*
ENDMODEL;
```

model_name has the form *cell:view* to specify the cell name and view name of the cell to be created. The policy_prop statements set properties on the facet. The statements are:

```
TECHNOLOGY tech_name;
VIEWTYPE view_name;
EDITSTYLE edit_name;
```

If VIEWTYPE or EDITSTYLE are not given, they default to "SYMBOLIC". TECHNOLOGY has no default and will not be set if it is not stated.

The *terminal_type* keywords are used to set the TERMTYPE and DIRECTION properties for declared formal terminals as shown:

KEYWORD	TERMTYPE	DIRECTION
INPUT	SIGNAL	INPUT
OUTPUT	SIGNAL	OUTPUT
INOUT	SIGNAL	INOUT
TRISTATE	TRISTATE	OUTPUT

CLOCK	CLOCK	INPUT
OUTCLOCK	CLOCK	OUTPUT
SUPPLY	SUPPLY	—
GROUND	GROUND	—

formal_terminal_spec may be in either of two forms:

```
        terminal_name
        terminal_name : net_list
```

net_list has one of two formats, with '&' implying a concatenation operation:

```
        net_name
        net_name & net_name & ... & net_name
```

When no net is specified in the *formal_terminal_spec*, a net is created for each formal terminal with the same name as the formal terminal. The the formal terminals are connected to the appropriate net.

Each *formal_terminal_name*, *actual_terminal_name*, and *net_name* is of three forms: *name<bit>*, *name<hi:lo>*, or *name*. *bit*, *hi*, and *lo* may be constant-valued expressions as described below. The first two formats are shorthand for the names *name<bit>*, and *name<hi>*, ... , *name<lo>*. If no bit subscript is given on an actual terminal name, a check is first made for a terminal with this name. If not found, a further check is made for the terminal *name<0>*. (This allows compatibility with bdsyn(1).)

Each *instance* is described as follows:

```
        INSTANCE master_name {modifiers}*
        {terminal_attachment}*
```

master_name has the format *cell:view* to specify the cell name and view name of the instance. The master and interface facet of the instance must exist prior to running bdnet.

The allowed *modifiers* are:

```
        [ x_position, y_position ]
        CONNECTOR
        NAME = instance_name
        PROMOTE
```

Two expressions in square brackets can be used to specify a specific oct coordinate for an instance. If this modifier is missing, bdnet will use the coordinates [0, 0].

The CONNECTOR modifier will cause the instance to be attached to the "CONNECTORS" bag rather than the "INSTANCES" bag.

The NAME modifier assigns an instance name to the instance. (By default, the instance name is the same as the cell name; however, all instance names will be made unique by appending an underscore and an integer as necessary.)

The PROMOTE modifier automatically assumes connections from each actual terminal of the instance to a net of the same name for each actual terminal which is otherwise not specifically connected or unconnected.

terminal_attachment takes three forms:

```
        actual_terminal_name : net_list;
        actual_terminal_name # formal_terminal_list;
        actual_terminal_name : UNCONNECTED;
```

The first form attaches the actual terminals to the given nets. The second form attaches the actual terminals to the given formal terminals. If the current instance is not a CONNECTOR instance, the actual terminal terminal is also connected to the net of the given formal terminal. (If this seems confusing, refer to the Oct Symbolic Policy.) The third form is for specifying explicitly that an actual terminal is not connected.

An *array_of_instance* has the form

```
        ARRAY %variable FROM start TO end {BY increment} OF
        {instance | array_of_instance}
```

start, *end*, and *increment* may be constant-valued expressions as described below. Within the scope of the ARRAY statement, *variable* takes on the values *start, start + increment, ... , end. variable* may be used within the bit subscripts to connect the nets to each instance.

Expressions

Expressions composed of constants and ARRAY indices are are allowed in bit-subscripts. Constants are by default base 10, but arbitrary bases may be specified using the bdsyn notation *value#base*. The operators are "+", "-", "*" and "()" with their usual meaning and precedence.

Macro Definitions

Macro definitions are allowed and are identical to those of bdsyn. They may appear anywhere in the file, and are defined for the remainder of the file.

Include files are supported using the bdsyn syntax of:

```
        REQUIRE 'filename.ext';
```

Bdnet Keywords

The following is a list of keywords used by bdnet. These words must be quoted if they are to be used as user names.

ALIAS	ARRAY	ATERM	BAG	BOX
BY	CLOCK	CONNECTOR	CREATE	EDITSTYLE
ENDMACRO	ENDMODEL	FACET	FROM	FTERM
GROUND	INOUT	INPUT	INSTANCE	IPROP
LAYER	MACRO	NET	NAME	OF
OUTCLOCK	OUTPUT	PATH	PROMOTE	REQUIRE
RPROP	SUPPLY	SPROP	TECHNOLOGY	TRISTATE
TO	UNCONNECTED	VIEWTYPE		

Annotation of Oct Properties, Bags, Etc.

Bdnet has a mechanism to create trees of Oct Objects to be placed in an Oct facet. The annotation must occur outside of a model block and has the form:

```
        (FACET cell:view:facet {list}* )
```

The *cell, view* and *facet* specify an Oct cell to annotate. The facet name may be left out and defaults to "contents".

A *list* is one of the following:

```
        ( INSTANCE instance_name {list}* ) |
        ( NET net_name {list}* ) |
        ( FTERM formal_terminal_name {list}* ) |
        ( ATERM actual_terminal_name {list}* ) |
        ( {CREATE} IPROP property_name integer_value {list}* {alias} ) |
        ( {CREATE} RPROP property_name "real_value" {list}* {alias} ) |
        ( {CREATE} SPROP property_name string_value {list}* {alias} ) |
        ( {CREATE} BAG bag_name {list}* {alias} ) |
        ( {CREATE} LAYER layer_name {list}* {alias} ) |
        ( BOX ll_x ll_y ur_x ur_y {list}* {alias} ) |
        ( ALIAS alias {list}* ) |
```

The meaning is straightforward. A tree of attachments is built which followed the tree of *lists*. (Arbitrary graphs are possible using ALIAS.) INSTANCE, NET, FTERM, and ATERM require that the object already exist in the given facet. They can not be created using the annotation mechanism. ATERM refers to a actual terminal of the instance last touched. This implies that INSTANCE must precede ATERM.

IPROP, RPROP, and SPROP look for an existing Oct property of the same name. If it exists it is modified to the given property value. If not it is created.

BAG and LAYER look for an existing Oct object of the same name and type. If it does not exist, it is created.

BOX creates the specified box.

IPROP, SPROP, RPROP, BAG, and LAYER all use an existing object if it is there. If the keyword CREATE is present, *bdnet* will create a new object whether one already exists or not.

IPROP, SPROP, RPROP, BAG, LAYER, and BOX definitions may contain a string that represents an "alias" for that object. Defining an alias for an object allows referencing this object via the ALIAS definition. In this way, arbitrary graphs of Oct structures may be built (as shown below).

COMMENTS

Bdnet differs from bdsyn in that all names (except keywords) are case sensitive. Names in bdnet must be quoted if they contain special characters such as '!', ';', ',', '#', '*', '-', '+', ')', '(', ']', or '['. Also, names that start with digits, RPROP values, and bdnet keywords that are being used as names should be quoted.

EXAMPLE

This example connects a bdsyn logic block (traffic_light:logic) to a set of latches.

```
MODEL traffic_light:symbolic;
   INPUT c<0> ts<0> tl<0>;
   OUTPUT hl<1:0> fl<1:0> st<0>;
   SUPPLY vdd;
   GROUND gnd;
   CLOCK phi1;

   INSTANCE traffic_light:logic PROMOTE;
            pState<1:0> : presentState<1:0>;
            nState<1:0> : nextState<1:0>;

   ARRAY %i FROM 0 TO 1 OF
            INSTANCE latch
               d: nextState<%i>;
               q: presentState<%i>;
               clk: phi1;
               vdd: vdd;
               gnd: gnd;
ENDMODEL;

(FACET traffic_light:symbolic
   (NET vdd (SPROP NETTYPE SUPPLY))
   (NET gnd (SPROP NETTYPE GROUND))
   (LAYER MET1
            (BOX 0 0 2000 2000 box_alias))        ! For floor-planning
   (BAG floor_plan (ALIAS box_alias)))
```

SEE ALSO

bdsyn(1) vulcan(1)
Oct Symbolic Policy Specification
~cad/doc/bdsyn.doc (BDSYN Users' Manual)

AUTHOR

Russell Segal

BUGS

All instances are assumed to reside in the current directory. Eventually we will rely on the PATH mechanism of Oct to remove this restriction; however, this will require the addition of a mechanism to insert a path into a cell.

It is possible to specify a connection of nets which does not conform to symbolic policy. In particular, two distinct nets may be attached to JOINEDTERMS.

NAME

bdsim – Multi-level simulator

SYNTAX

bdsim [-cmuw] oct_cell[:view]
bdsim -s[-cmuw] sim_file(s)

DESCRIPTION

Bdsim is a multi-level simulator. Its primary mode of operation is switch level simulation of MOS transistors. Provisions have been made within *bdsim* to allow simulation of higher level constructs such as logic gates, latches, and memory cells. The interface to these functions is not yet complete. An interface will also be provided to allow stand-alone programs to drive simulations. This will hopefully be achieved by linking user modules to bdsim or through the use of UNIX pipes.

Bdsim performs preprocessing steps on transistor networks to increase the speed of simulation. This is includes merging of series-parallel transistor networks, and recognization of pull-down and pull-up transistors.

The command line options are:

-s	The list of files that follow the options are standard sim(5) format.
-c	By default instances in a hierarchy are displayed using their "instance path" using the character '/' as a separator (i.e. /top/middle/bottom). The -c option is followed by a single character to be used instead of '/' as a separator.
-m	Turn off the merging optimization (described below).
-p	Turn off the pulldown/pullup optimization (described below).
-w	The floating point number that follows -w specifies the threshold for weak transistors. Transistors with W/L ratios that are strictly less than the threshold are considered weak. The default threshold is 1.0.

GENERAL STRUCTURE

The *bdsim* data structure is set up as a collection of instances and nodes. (See nodeinfo and instinfo commands described below.) Each instance represents a functional unit. An instance may be a transistor, a network of transistors, a logic gate, or some more complex, user-defined unit. The instances have connections to one or more nodes. (e.g. A transistor connects to three nodes -- one at its gate, one at its source, and one at its drain.)

A node may be in any one of 8 states. Each state has a representative letter as shown:

'o'	Charged low	'i'	Charged high
'0'	Driven low	'1'	Driven high
'O'	Set low by the user	'I'	Set high by the user
'x'	Uninitialized	'X'	Driven both high and low

Unlike many other simulators, *bdsim* treats every node as a storage node. Once a node is charged high or low, it will remain that way until the node is driven by an instance or set by the user. It is important to note that *bdsim* does not model charge sharing. If a 'o' node and a 'i' node are electrically connected by a transistor, the nodes will not change in value. This would not be the case in a physical system.

Each node in *bdsim* has a counter associated with it. The counter is used to detect oscillations in the simulation. (This could be caused by a ring of inverters, for example.) If a node is determined to have made more than 256 transitions in one simulation step, a warning will be printed, and the simulation will be interrupted. *Bdsim* will find and report all of the nodes which contribute to oscillation in a circuit.

Bdsim has two different transistor models, regular and weak. The current carried by a weak transistor is such that it may be overdriven. The classic use of a weak transistor is in an NMOS inverter. The pull-up device of an NMOS inverter is weak. When the pull-down transistor is off, the output is pulled high by the weak transistor. When the pull-down device is on, the pull-up is overdriven and the output is pulled low. If the pull-up device were to be regular, the result of both transistors being on would be an 'X' at the output. Another use of a weak transistor is a static memory cell, where values are written by overdriving the cell's internal transistors.

When sim(5) format files are being used as input, weak transistors are recognized in two ways. A weak transistor is generated if its width/length ratio is less than 1 (or the value specified by -w). Alternatively, a weak transistor is generated if the sim transistor has a "weak" gate attribute:

```
        n g_node s_node d_node 2 4 0 0 g=weak
```

Magic(1) gives an easy mechanism for entering the gate attribute. Simply place the label "weak^" in the active region of the transistor. In Oct, weak transistors can be selected by attaching a property "BDSIM_WEAK" to either the transistor instance or the instance's interface facet.

Switch level simulators have a fundamental flaw in that they cannot correctly handle certain transistor configurations (e.g. Pass gate exclusive OR circuits). This limitation may be overcome by restricting certain transistors to be unidirectional. When sim(5) format files are being used as input, this is done by giving the "in" attribute for either the source or drain of the transistor:

```
        n g_node s_node d_node 2 4 0 0 g=weak d=in
```

Magic(1) gives an easy mechanism for entering the source or drain attribute. Simply place the label "in$" on the edge of the active region of the transistor adjacent to the source or drain. In Oct unidirectional transistor are selected by attaching a property "BDSIM_IN" to either the SOURCE or DRAIN terminal of the transistor instance.

In the following sections, the term "element" is used. An element is the user's name for a node. Many element names may correspond to the same node.

HIGHER LEVEL GATES

Bdsim will also simulate higher level gates which it reads from the Oct database. (There is no way to specify high level gates in sim(5) format.) These gates may be logic gates, as specified in the Oct symbolic policy, or they may be latches. Latches are specified via a set of Oct properties. Firstly a latch must be CELLTYPE MEMORY and have the property "SYNCH-MODEL" in its interface facet. The "SYNCH-MODEL" property may take on the values "TRANSPARENT-LATCH" or "MASTER-SLAVE-LATCH". (The Master-Slave latch is two cascaded transparent latches with the slave clock fed by the inverted master clock). A further property, called "SYNCH-ACTIVE", attached to the interface facet specifies whether the transparent latch (or master latch) conducts when the control signal is low or high. If "SYNCH-ACTIVE" has the value "LOW" the latch is active low. Latches default to active high.

The terminals of lathes are identified a property called "SYNCH-TERM" attached to the interface formal terminals. The legal values for this property are INPUT, OUTPUT, OUTPUTBAR, and CONTROL. A latch must have exactly one control, exactly one input, and at least one type of output terminal.

An industrious programmer may create his own bdsim gates. Several routines must be written and linked into the bdsim program. Details may be found in the file "USER_GATES" in the bdsim source directory.

SYNOPSIS OF COMMANDS

Commands in bdsim are invoked by typing a command name followed by a variable number of arguments. Each command has a two letter abbreviated form which can be used in place of the command name.

help (?)
This prints a short synopsis of each command.

lopen log_file (lo)
Open a file to record all transactions within bdsim.

lclose (lc)
Close the current log file.

macro macro_name (ma)
Define a macro command. After typing the macro command, bdsim will give a '>' prompt. At this point, the user should type a sequence of commands that make up the macro. Macros may be nested, as long as the nested macro is defined prior to the current macro. The macro definition is concluded by typing "$end".

Macro commands may have arguments that are used in the definition. In the definition, these arguments are written as '$1' for the first argument, '$2' for the second, etc. As an example, the following macro sets the first argument to one, sets the second argument to zero, evaluates, and prints some information.

```
        macro seteval
           set $1 1
           set $2 0
           evaluate
           show "cycle %d0 #cycle
           set #cycle = #cycle + 1
        $end
```

The "seteval" macro would be invoked by typing:

```
        seteval phi1 phi1_bar
```

savestate save_file (ss)
Save the current state of the simulation. It is required that there be no pending events when a savestate is attempted.

loadstate save_file (ls)
Load a previously saved state back into bdsim. The circuit itself may not be changed between saving state and loading state back.

evaluate (ev)
Cause all pending changes to be propagated through the network.

source [options] source_file (sr)
Run commands from a file. (See the section below for a detailed description of the options.)

step [[options] source_file] (st)
The "step" command, when followed by a source_file name will begin executing the commands in the file one at a time. (See the section below for a detailed description of the options.) When "step" is given with no arguments, the next command in the current source file will be run in a single step fashion.

continue (co)
Continue an running a "evaluate" or "source" that has been interrupted. Using continue will preserve the event count, and preserve the node transition counts that are used to detect cycles in the network.

watch watch_set [vectors | elements] (wa)
Listed vectors and elements are added to the given watch set. All elements of a particular watch set are listed together, when a watch set is used in the "show" command.

listsinsts [instance_name] (li)
Print the name of all instances that are contained in the given instance. If no instance name is given, the current working instance is used. This is used when examining instance hierarchies and is not useful for sim(5) format input.

listelems [instance_name] (le)
Print the name of all elements that are contained in the given instance. If no instance name is given, the current working instance is used. This is used when examining instance hierarchies and is not useful for sim(5) format input.

printinst (pi)
Print the name of the working instance that is currently being used as the default for referencing elements (like a working directory in UNIX). This is used when examining instance hierarchies and is not useful for sim(5) format input.

changeinst instance_name (ci)
Change the working instance that is currently being used as the default for referencing elements (like a working directory in UNIX). This is used when examining instance hierarchies and is not useful for sim(5) format input.

equivalent element (eq)
List all equivalent names (aliases) for an element.

makevector vector_name [elements] (mv)
Define the set of given elements (node names) to be a vector named "vector_name". The elements are listed from most significant bit to least significant bit. Once a vector is defined, it may be set (set command) and displayed (show command) as one unit.

Makevector assigns special meaning to the colon (':') character. When a colon is encountered, *bdsim* will generate a series of element names that are numbered consecutively from the number preceding the colon to the number following the colon. For example:

```
        makevector busA busA<4:0>
is equivalent to
        makevector busA busA<4> busA<3> busA<2> busA<1> busA<0>
Similarly:
        makevector cellvalues cell1:3value
is equivalent to
        makevector cellvalues cell1value cell2value cell3value
```

show [(sh)
Print out the current value of all given vectors, elements, and all members of given watch sets. If a constant (denoted by a leading '#') is given, the current value of the constant is printed in decimal. By default, vectors are printed in hexadecimal and elements are printed in binary. The optional format_string is a quoted string which resembles 'C' printf format string. It may contain regular characters which are echoed in the output, %b to insert the binary representation of numbers, %h to insert the hexadecimal representation of numbers, for carriage returns, and for tabs.

set [vector | element] value (se)
Assign the value to the given vector or element. The value is assumed to be a binary number (1's and 0's). When an element is set, it is held at the given value, regardless of the state of the simulation. An 'x' may be used in place of the binary digit will allow the corresponding node to change as the simulation dictates. Hexadecimal values may be specified by preceding the number with an 'H'. In the case of a hexadecimal number, an 'x' represents four 'x' bits.

set #constant expression (se)
Set may also be used to set the value of constants. Constants are denoted by a leading '#'. The expression from which the constant is set is evaluated from left to right and may contain other constants, integer numbers, '=', '+', '-', '*', '/', and '%'. Be careful to assure that all operators, numbers, and constants are separated by spaces.

verify [vector | element] value (ve)
Check to see if the vector or element has the given value. The value may contain 1's, 0's, X's, and .'s. An 'X' will match the 'X' or 'x' node states in the vector or element. A '.' is a don't care and will match any value in the vector or element. Hexadecimal values may be specified by preceding the

number with an 'H'. In the case of a hexadecimal number, an 'X' represents four 'X' bits, and a '.' represents four '.' bits. If the verify fails, an error message is printed.

nodeinfo [elements] (ni)
Print interesting information about a node. The value of the node is printed along with the node's fanin and fanout. This command is very useful in inspecting the state of the network.

instinfo [instances] (ii)
Print interesting information about an instance (transistor or more complex instances). For transistor networks, the logic function is listed along with its current state (off or on). Also, the nodes that are connected to the instance's terminals are listed. This command is very useful in inspecting the state of the network.

backtrace element [levels] (bt)
Prints information about the instances which are responsible for driving the selected node. If the levels parameter is given, backtrace will be called recursively on nodes which fanin to the given instance. The levels parameter tells *bdsim* how many levels to backtrace.

Backtrace does not simply list the instances that are connected to a node, it lists instances which *drive* a node. This makes it particularly useful for answering the question, "What caused this node to be this value?"

setbreak [vector | element] value (sb)
Not implemented yet

MORE UTILITY AND SOURCE CONTROL

Bdsim includes a built in "more" utility. This utility is used to control the output of data to the screen. After *bdsim* has printed an entire screenful of data, it will pause with the message "-- more --". There are four legal responses to the more prompt. Typing carriage return will cause the listing to continue for another page and prompt "-- more --" once again. Typing "C" will cause the listing to continue through its completion, without any further interruption. Typing "S" will terminate the listing. And typing "L" will direct the listing to only the current log file (if there is one) without dumping the information to the screen.

The *bdsim* environment works on the concept of levels. When it is first started, *bdsim* is in level 0. For each nested "source" (or macro) command, the level is increased by one. At any particular level, several environment variables may be set via options to the "source" or "step" command. There are four main listing modes options. They are:

M	use the regular More utility
C	output data Continuously, without the more utility
L	output data only to the current Log file
O	output One page to the screen, then only to the log file

In addition, a number may be specified to set the number of lines in a "more" page. This number will be used as the number of lines between "-- more --" prompts, and the page length for the 'O' option. For example, if one wanted only 5 lines of output for each command to be printed on the screen, he could type:

```
source -O5
```

If he wanted to run all the commands in a source file, without any output to the screen, he could type:

```
source -L source_file
```

Note that the 'O5' option above applies to the current level, but the 'L' option in te second example applies only to the level of "source_file."

Commands in a source file are normally executed without stopping, but the running of source files can be stopped by a number of causes. The execution of source files is halted by an interrupt (control C), a break generated by the "setbreak" command, or a hard error (such as a syntax error in the source file).

There are also two soft errors that may interrupt the running of a source file. The first is a verify failure, generated by the "verify" command. The other is a "not found" error, which occurs when an element is accessed but does not exist. *Bdsim* will stop on these errors based on the status of the 'V' and 'F' flags. The flags are set by preceding the option by a '+', and cleared by preceding the option by a '-'. For example, to run a source file stopping on verify errors but not on "not found" errors, one would type:

```
        source +V -F source_file
```

If one wished to stop processing after every command of a source file. The processing should be initiated as follows:

```
        step source_file
```

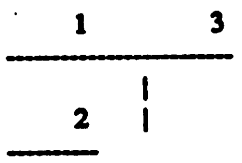
Each following command could then be run by typing "step".

The 'R' option of the "source" command resets the level of the source file. This allows one to skip the remaining commands in a source file, and return to the "level 0" interactive mode of *bdsim*.

OPTIMIZATION

Bdsim performs two optimization steps to the transistor networks as they are read in to memory. The first is to recognize pull-up and pull-down transistors. A pull-up transistor is a transistor with either its source or its drain tied to Vdd. Similarly a pull-down transistor has its source or drain tied to GND. Recognizing this special case of transistor is important since it may be treated as a unidirectional device. Regular transistors are bidirectional and are therefore slower to simulate.

The second optimization is the merging of transistors into series-parallel networks of transistors. For example, three PMOS transistors in parallel may be modeled as one instance. The "instinfo" command gives a representation of the function of the transistor network. A '*' denotes transistors in series, and a '+' denotes transistors in parallel. For example:
(* (+ 1 2) 3) would represent the following network:



COMMENTS

When reading sim(5) format files, *bdsim* will ignore everything except the transistor declarations and node aliases. The read-in time for a circuit may be decreased by removing extraneous resistor and capacitor declarations.

When creating pullups and pulldowns bdsim keys off the elements named "GND" and "Vdd". If these names are not found, no pullups or pulldowns are created.

SEE ALSO

sim(5), magic(1)
src/bdsim/USER_GATES
The General Structure of Oct
Oct Symbolic Policy Specification

AUTHOR

Russell Segal

BUGS

The default names Vdd and GND can not be overridden.

The savestate command will not save the internal state of instances. For this reason, it is impossible to save the state of a circuit containing master-slave latches.

NAME

bdsyn – BDS subset translator for describing logic networks

SYNTAX

bdsyn [–bcdensuz] [filename]

DESCRIPTION

Bdsyn is a translator which supports the generation of multiple-level logic networks. It takes as input programs written in the hardware behavioral language BDS, and generates a BLIF (Berkeley Logic Intermediate Format) description of an equivalent multiple-level logic network. BDS is a behavioral description language which is part of Digital Equipment Corporation's hardware simulation system DECSIM. BLIF is suitable input for the multiple-level logic optimization tool *mis(1)*. Using *mis*, BLIF can be converted into either Oct symbolic view or a PLA in Berkeley PLA format (*pla(5)*).

Bdsyn generates only combinational logic equations. In order to generate entire sequential systems, latches and tri-state devices must be integrated with the combinational logic from bdsyn in an Oct symbolic view. This can be accomplished using a net list translator such as *bdnet(1)*.

The command line options are:

–b Tells bdsyn not to do its cleanup evaluation. This cleanup evaluation process in bdsyn eliminates redundant and unnecessary logic. Unfortunately, it may also have the undesired effect of eliminating user specified intermediate variables that bdsyn considers unimportant. The **–b** makes sure that the variable are preserved, at the price of extra logic.

–c is followed by a number which tells bdsyn how much "collapsing" of logic that it should do. Collapsing is done in order to decrease the amount of output which bdsyn creates. Although quite rare, it is possible to create an input description which causes bdsyn to collapse too much logic and run for an inordinate amount of time. The option **–c1** will limit the amount of collapsing that is done, and will alleviate the problem. Bdsyn uses MIS to do logic collapsing for it. The actual communication with MIS is accomplished through the use of unix pipes. On non-unix systems or on a system where MIS is not available, the option **–c0** should be used to turn off collapsing altogether.

–n *Bdsyn* assumes that any variable that is not assigned to has the value of zero. The **–n** option causes a table to be printed in the output for each variable for which this assumption is made. The user should refer to the "Bdsyn Users' Manual" for details on how to make use of this information.

–s Causes all SELECTALL's to be changed into SELECT's. This is only useful if you are using DECSIM and wish to use expressions in SELECT cases.

–u *Bdsyn* will provide periodic updates as to the progress of its execution.

–z Causes DONT_CARE's in the input file to be assigned the value zero.

The debugging options (not intended for general users) are:

–d Dumps the lexical tokens as the input file is parsed.

–e execute_level
–e must be followed by a number which specifies to what stage the translation should be run. The default is to run through completion.

Bdsyn Files

Bdsyn uses a standard library of macro definitions to implement complex operations on logic-variable arguments. The library is written in standard BDS syntax and is automatically loaded at run time. *Bdsyn* will first look in the current directory for "bdsyn.lib". If it is not found, it will look in "~cad/lib/bdsyn/bdsyn.lib" for the standard library.

On unix systems, *bdsyn* forks *mis(1)* as a child process. Unless the option **–c0** is used, *bdsyn* expects that the *mis* program can be found in the users search path.

FILES

mis
./bdsyn.lib
~cad/lib/bdsyn/bdsyn.lib

SEE ALSO

mis(1), espresso(1), bdnet(1), pla(5)
~cad/doc/bdsyn.doc (BDSYN Users' Manual)
~cad/doc/blif.doc (BLIF description document)

MISCELLANEOUS

To generate a PLA in the Berkeley standard format, use the command

```
mis -c clp -Tpla file.blif > file.pla
```

AUTHORS

Richard Rudell
Russell Segal

NAME

octflat – OCT symbolic hierarchy flattener

SYNTAX

octflat [-l level] [-o outcell[:outview]] incell[:inview]

DESCRIPTION

Octflat takes a hierarchy of Oct symbolic cells and flattens it to the equivalent single level hierarchy. The Oct cell must be created to the Oct Symbolic Policy Specification. All connectivity information is correctly translated.

The command line options are:

-l level The -l option must be followed by a number which specifies at what level *octflat* will stop flattening.

Level 0 (the default) flattens until no instances remain.

Level 1 stops when cells with VIEWTYPE of PHYSICAL are reached.

Level 2 stops based on a user specified function, if the function has been linked using "make userflat".

Level 3 stops when a cell with CELLCLASS of LEAF is reached.

-o This option is used to specify the output facet. The default is to flatten in place.

Flattening may be stopped at a level other than that given by the -l option. To do this the property FLAT_STOP must be attached to either the instance that should not be flattened, or to the interface facet of the instance.

The user may specify his own stopping criterion for the flattener by linking in his own function. The function must be called userStop and must take an instance and an interface facet as arguments:

```
        int userStop(instance, interface)
        octObject *instance, *interface;
```

If the function returns true (!= 0) the the instance is not flattened. To link in userStop, create a file containing userStop called "userflat.c" and make a new executable using "make userflat".

SEE ALSO

bdnet(1)
Oct Symbolic Policy Specification

BUGS

This is a bit of a hack.

AUTHOR

Russell Segal

Bibliography

[1] *The Ella System Overview.* Praxis Systems plc., Bath, England, 1985.

[2] *VAX DECSIM Reference Manual.* Digital Equipment Corporation, Hudson, Mass., 1986.

[3] *VHDL User's Manual.* Intermetrics, Bethesda, Maryland, 1985.

[4] B. Brayton, E. Detjens, S. Krishna, T. Ma, P. McGeer, L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung, and A. Sangiovanni-Vincentelli. Multiple-level logic optimization system. In *ICCAD*, pages 356–359, November 1986.

[5] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[6] N. Brenner. The yorktown logic language: an apl-like design language for vlsi specification. In *IEEE International Conference on Computer Design*, pages 11–15, 1984.

[7] B. Cmelik. Eqntott manual page. *1986 VLSI Tools: Still More Works by the Original Artists*, 1986. Report No. UCB/CSD 86/272.

[8] D. S. Harrison, P. Moore, R. L. Spickelmier, and A. R. Newton. Data management and graphics editing in the berkeley design environment. In *ICCAD*, pages 20–24, November 1986.

[9] S. Johnson. Yacc: yet another compiler-compiler. *Unix Programmer's Manual*, March 1984. Report No. UCB/CSD 86/272.

[10] G. M. Ordy and C. W. Rose. Functional simulation shortens the development cycle of a new computer. In *ACM IEEE 20th Design Automation Conference*, pages 520–526, Miami, Florida, June 1983.

[11] D. L. RavenScroft and M. R. Lightner. Functional language extractor and boolean cover genrator. In *ICCAD*, pages 120–124, November 1986.

[12] C. W. Rose, L. A. Rogers, and R. V. Straubs. The n.mpc system description facility. In *16th Design Automation Conference*, pages 520–528, San Diego, California, June 1979.

[13] R. Rudell, A. Sangiovanni-Vincentelli, and G. De Michelli. A finite-state machine synthesis system. In *International Symposium on Circuits and Systems*, 1984.

[14] Richard Rudell. *Multiple-Valued Logic Minimization for PLA Synthesis*. Master's thesis, University of California, Berkeley, 1986. Memorandum No. UCB/ERL M86/65.

[15] C. Terman. Esim manual page. *1986 VLSI Tools: Still More Works by the Original Artists*, 1986. Report No. UCB/CSD 86/272.

[16] A. Wang. Mis manual page. *OCT Tools Distribution 1.0*, 1987.

[17] D. Wood. Meg manual page. *1986 VLSI Tools: Still More Works by the Original Artists*, 1986. Report No. UCB/CSD 86/272.