CMU Fall’01 18-760 VLSI CAD

Project 1: Implementation of a Basic BDD Package. (Part I)

[120 pts] Out: Thu Sep 20, Due: Thu Oct 5 in class.

---

1. Overview

The goal is to write a Java program to implement the ITE operation on Binary Decision Diagrams. You are going to end up with a **YBDD** (*Your BDD*) package with the following features:

- **Sharing:** Implemented as a Multi-rooted DAG.
- **NO negation arcs:** Though negation arcs do reduce memory requirements, they make the BDD structure unreadable. So we do not require that you implement negation arcs.

Implementing **YBDD** with Java has a couple of advantages:

- **Simple Code:** Java already has Hash Tables and other useful data structures to simplify this assignment.
- **Nice Environment:** it’s interpreted, catches errors, making a possibly simpler programming task.

Your task is to concentrate on the implementation of the ITE operator. We provide a command line interface to drive the BDD package. Here are step by step instructions of what you should be doing:

- All the files for this project exist in: **/afs/ece/class/ee760/proj1**
- Make a directory to work on this project from. Use **fs la <directory_name>** to see the access permissions to that directory, then *type:*
  fs sa <directory_name> <other_user_name> none
  to protect it, where **<other_user_name>** are users other than yourself and the system administrators. **You are required to work alone on this project!** Change into that directory.
- Run the command: **source /afs/ece/class/ee760/bin/setup.script** from that directory. This will work fine if you are on a Solaris or AIX machine, otherwise take a look at the script to see what it is doing (just linking and copying some files).
- There are now two files which you will be concerned with (the rest of the linked directories and files are for the command line interface.) These files are: *bdd_node.java* and *bdd_functs.java*
- The file *bdd_node.java* contains our object to represent BDD nodes with. You don’t have to change this file (although you can if you want). This file is discussed in Section 2.
- The file *bdd_functs.java* contains the functions necessary to implement **YBDD**. There are a number of empty class methods here which you need to fill in. This file will be discussed in more detail in Section 3.

- After you are done making changes, simply type:
  javac bdd.java bdd_node.java bdd_functs.java
  This will rebuild all your classes.
- Then to run your BDD package type: **java bdd**. This will start the command line interface to **YBDD**, which is described in more detail in Section 4.
- A set of test files will beplaced in **/afs/ece/class/ee760/proj1/tests**. These are source files that can be sourced through the command line interface. Look at the README file in this area.
- What you need to submit: the source code of file *bdd_functs.java*, results of all sourced files in **/afs/ece/class/ee760/proj1/tests** and a **word-processed report**. The grading is explained in Section 6.

If you have any questions or concerns about any part of the project, please feel free to contact Amit Singhee (asinghee@ece.cmu.edu) or Rob Rutenbar (rutenbar@ece.cmu.edu) and ask questions.

2. The BDD Data Structure

The BDD structure has been defined for you in *bdd_node.java*. Here is a brief description of each field and method in the BDD object:

Instance Variables:

- **marked:** Used for traversing the BDD. Marks that you have visited a note, used for reursive algorithms that do things like count the number of nodes. When you create a new BDD node, set this field to FALSE.
- **index:** For non-terminal nodes, index defines the variable associated with a node. Variables are indexed 0 to n-1. The index therefore defines the global ordering. For terminal nodes, index should be set to ZERO_INDEX or ONE_INDEX for the constant 0 and 1 nodes, respectively.
- **label:** A String representing the name of the node (redundant information if index is present but useful for the methods described below)
- **low, high:** These are low and high pointers to BDD nodes.

Instance Methods:

- **bdd_node(....):** The constructor method, does instance variable initialization.
- **public int getIndex():** Returns the index of that node.
- **public bdd_node getLow():** Returns the ‘low’ BDD child node.
- **public bdd_node getHigh():** Returns the ‘high’ BDD child node.
- **public String getLabel():** Returns the label of the node.
- **public int Size():** Recursively calculates the size of the BDD structure including the current node and its children.

- **public void Print(....):** Recursively prints the BDD node structure starting at the current node, and working down to the children.
- **public void cleanMarks():** Cleans up after Size or Print have been called.
- **public static void total():** Counts total number of BDD nodes in your multi-rooted DAG, and prints this number.

For more detailed information on any of these variables or methods, refer to comments in *bdd_node.java*.

3. BDD Basics: CoreManipulation Routines

We have provided you with a set of stubs in file *bdd_functs.java*, i.e., *empty* routines that need to be filled in. Do not change these routine names since they are used in the command line interface. You need to fill in the following stubs:

- **public static bdd_node Zero(), public static bdd_node One():** Routines to create the constant 0 and 1 BDD nodes. Remember that these have to be inserted in the unique_table.
- **public static bdd_node NewVar(String label):** Create a new BDD variable.
- **public static bdd_node ITE(I, T, E):** This is the main ITE routine.
- **bdd_node NOT(f), bdd_node AND(f, g), bdd_node OR(f, g), bdd_node XOR(f, g):** These should be defined in terms of the ITE operator.

These are the only methods which you must code in order to create a working BDD package. There are a number of other methods which you may wish to change in order to implement ‘something cool’. For more about ‘something cool’ see section 5.

4. Command Line Interface

A command line interface is linked in with the program bdd. Executing **java bdd** will bring up the command line interface. The package will respond with a prompt:

```
YBDD>
```

At this point the user can issue one of the following commands:

boolean *list-of-var-names*	**eval** *dest expr*	**source** *filename*
bdd *funct*	**size** *funct*	**total**
quit		

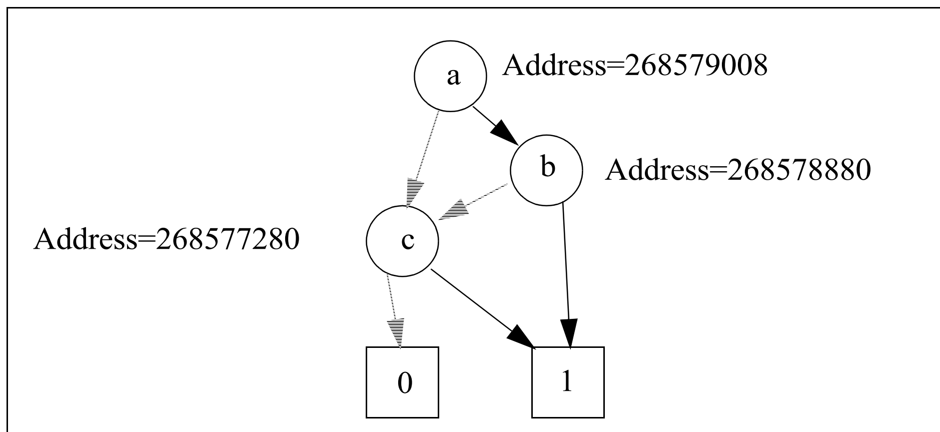
The following describes what each command does.

boolean: The command boolean defines a list of variables. The order of the variables is the order in which they are defined. Example: boolean a b c d command defines variables a, b, c, d with indices 0, 1, 2, 3 respectively.

eval: The command eval evaluates the expression and assigns it to the destination. Expressions can have the following operators: ! (complement), & (AND), + (OR) and ^(XOR). Only one operator per eval command can be used. Example: eval f a & b evaluates the Boolean expression (ab) and creates the corresponding BDD with the function name f, i.e., it does f = ab.

bdd: The command bdd prints the BDD representation for the function. The BDD representation for **a&b + c** and its corresponding pictorial representation are shown below:

```
a:268579008                node
               c:268577280low child
                           [0]
                           [1]
               b:268578880high child
                           c:268577280
                           [1]
```



size: The command size returns the number of BDD nodes in the function. For example size will return 5 for the above function.

total: The command total returns the total number of BDD nodes created.

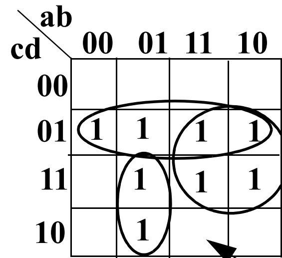
source: All the commands can be placed in a file and then the file can be sourced via the command source. An example source file (for the function: **f = a&b + b&c + a&c**) is:

```
boolean a b c
eval t1 a & b
eval t2 b & c
eval t3 c & a
eval t4 t1 + t2
eval f t4 + t3
size f
bdd f
```

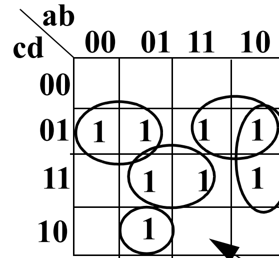
5. BDD Extension: Primes

We are also going to ask you to add one rather unusual feature: the ability to build a BDD that represents the *prime implicants* in a logic function, using metaproduct notation (like on HW1). It turns out that you can this in a way very similar to an ITE implementation.

First, a little review--what exactly are “prime implicants”? Informally, they are what you try to circle in any Karnaugh map. An *implicant* is just any cube or product term in a function. But a *prime-implicant* is as big as it can be, considered geometrically, as a cube. You cannot draw the Kmap “circle” any bigger for a prime. For example



These are **all** PRIMEs, cannot make any of these cubes any bigger



None of these are PRIMEs, can make all of these bigger

It turns out that a function, even a “small” function with only a few variables, can have many many primes. So, trying to find them with Kmaps, and enumerate them all, is hopeless. But, nicely enough, you can find them and represent them all with a BDD, if you use the metaproduct notation from the HW1, and you use an appropriate recursive strategy.

The recursive strategy is a bit tricky, but it uses BDD operations itself, and looks like another ITE routine that builds up the new BDD as the recursion “returns” up to the root. Part II will describe the mechanics for doing this, in detail. Roughly speaking you’re going to do this:

- We want to implement a new routine called PRIME(). PRIME takes a pointer to a function F represented in an MRDAG, and return a pointer to a node in the MRDAG, just like ITE.
- Assume your variables are *x1, x2, x3, ..., xn*, ordered like this. In the final metaproduct result, your new variables will be *r1,s1, r2, s2, r3, s3, ... rn, sn* -- in this exact order. This means if you have *N* variables in your function, to build the prime representation, you need to allocate *another* 2*N* variables for the **occurence vars** *rk*, and the **sign vars** *sk*.
- Computing PRIME(F) is again based on a recusive attack. If you can compute PRIME(F) directly, you do it and return a result. If no, you cofactor using the variable at the top of the BDD DAG you are invoked with, and recursively call PRIME( ) on some appropriate pieces of the function, then “glue” the answers back together.

6. Something *Cool*

The BDD package that you will create if you simply fill in the stubs described in Section 3 is a working, if rather *limited*, BDD package. It is not optimal in a number of ways. Since this is a 700-level class, and presumably you are interested in doing something more than making the worlds biggest-and-slowest BDD package, we want you to think about doing something *more*...

To be precise: *we would like you to do something to this package to impress us*.

Since this is a bit vague, included below are a list of a few ideas for making the **YBDD** package a little cooler and more interesting. Please feel free to come up with your own ideas on how to change it. Feel free to come by and discuss your ideas anytime.

Some ways to impress:

- **The smallest code ever:** Use as many Java functions and efficient programming ideas as possible to implement this with as few lines of code as possible. Yeah, we know this code is going to be really, really slow. That's OK.
- **Make it faster:** Using a lot of preprogrammed Java features is not always the fastest way to go. Writing your own Hash routines, instead of using Java's may speed up your program. Find your favorite algorithm to speed up the program. And, we know this will probably add *more* code. That's OK.
- **Better command line options:** Right now it's just 2-operand Booleans, and a few other queries and miscellaneous utilities (like source). Handling n-operand Booleans would be very convenient. Handling real arithmetic, like: ((a&b)+c)&!(a+b) directly on the command would be extremely cool, and also lots more convenient than the way we do it now.
- **Negation edges:** put them back, save 2X in BDD node count. Need to worry about the canonical reductions, and some pointer plumbing.
- **Make it use less memory:** The bdd_node object we've given you is rather big and clunky, especially compared to any real BDD package. Make it as small as possible and you can make bigger BDDs, and it might even run faster. A way to test this is to make a simple function like **a0&b0 + a1&b1 +a2&b2 + a3&b3... + an&bn** and use what you know to be the *worst* variable order: **a0 < a1 < a2 ... < an < b0 < b1 < b2 < ... < bn**. You know this makes an exponential size BDD. How big can you make **n** here if you focus on trying to make the BDD nodes small?
- **Garbage collect:** note that while Java is itself garbage collected, it can only collect unused nodes when it understands that nothing is pointing to them, i.e., they are not in use. In our current simple data structure, we don't ever free up any nodes in a way that Java can understand. So, your BDD node pool just grows and grows. To overcome this, you at least have to add a **reference_count** field to each BDD node, and increment it whenever you add a new pointer to that node, and decrement it whenever somebody stops pointing to that node. You can decide then if you want to get clever and maintain your own BDD node

pool, i.e., recycling freed nodes to your own pool and reusing them (overloading the constructor to use a recycled node when this pool is not empty, rather than just creating a new one). Lots of possibilities here.

- **Graphical Output:** Instead of the simple text representation of BDDs that we have given you, figure out how to draw a BDD on the screen. It doesn't have to look really slick, *anything* would be really impressive. For example, you know that each variable represents a level in the BDD tree, so each node gets drawn on a horizontal line. You could just evenly space the nodes on the line, and represent each node with a text string that is the variable name. Then draw red and green lines that connect the nodes. What is really hard is getting an aesthetically pleasing ordering where none of the edges cross each other. *Don't* worry about this kind of stuff. *Anything* at all graphical would impress us.
- **Added BDD functionality:** Add a cool boolean operation we don't have: **Satisfy()** for example, to actually print out all, or at least one, satisfying assignments. Or actually implement a real **Restrict**() operator (look in the DeMicheli book).
- **Variable Ordering:** Implement one of the variable ordering techniques discussed in class (The especially impressive thing to implement is **dynamic variable ordering**; we'll talk about that later).
- **Other:** Any other cool idea you can think of.

Remember: there are 3 main things necessary to impress us.

1. **Explanation:** *explain* what you wanted to try to accomplish.
2. **Implementation:** show *how* you implemented these features.
3. **Documentation:** show some examples of *what* you can do with this cool stuff.

The "cool" points on a 760 project are always awarded competitively -- we look at what the range of "cool" ideas was, and then use our judgement about "how cool" your addition was to award points. That being said, the threshold for "...a little but cool" is pretty low. Anything interesting will get you some fraction of the points. But to get all the points is *rare*, and goes to those with the most interesting extensions.

7. Grading--What you have to hand in to get credit

On the following pages we describe all the stuff you have to do for this assignment. **Read it carefully, pay attention to the details.** We are asking for you to several tests on the BDD code and for some analysis of what happens when you run these tests. You must not only write the code and tell us about it, but do these experiments and analyses as well.

Grading: 18-760 Fall'01 Project1: YBDD Software Package

NAME: **[120 points total]**

Description of Approach [8 pts]

Briefly describe the approach. What were your goals and how did you figure out if you met them?

Code and Code Quality [20 pts]

Turn in the source code that you modified or created. **Use a marker to highlight only the lines that you added.** (Or, just print with the added stuff in color, if you have access to a color printer.) Does the code look nice? Would you show it to your Mom if she were a programmer? Does it have comments, white space, indentation?

Implementation of Simple Functions [4 pts]

Using the **YBDD** package, build the BDDs for the following functions, and compare the answers with manually drawn ones:

- ❑ XOR of 3 variables;
- ❑ X**N**OR of 3 variables;
- ❑ The function of 4 variables that gives a 0 if exactly 3 of the variables are 1, and gives a 1 otherwise.
- ❑ The function of 4 variables that gives a 1 if exactly 3 of the variables are 1, and gives a 0 otherwise.

Implementation o*f Something Cool* [20 pts]

Tell us what you did that was interesting (*if* you did) and why it should impress us. Show how you compared the package *with* your cool thing, to the package *without*.

Results of Test Cases [60 pts]

Test **YBDD** on all the tests in /afs/ece/class/ee760/proj1/tests and turn in the results. Specifically we are looking for the following:

- ❑ **and.src, abc.src [1pt]** - These are sanity tests. Just report the BDD structures printed by **YBDD**.
- ❑ **tautology.src [1pt]** - Is the function a tautology? If so it should be the constant 1 node.
- ❑ **identity.src [1pt]** - Are the two functions the same in each of the cases? How do you tell?

- ❑ **addr4.src, addr8.src, addr16.src, addr64.src, addw4.src, addw8.src, addw16.src, addw64.src [16 pts]** - Set of adders. 4,8,16 and 64 bit adders with two different variable orderings. Report sizes of BDDs. Also see if you can draw conclusions as to which is a better variable ordering and why? Again, can you specify how the BDD size grows with increasing adder size? Can you derive the *exact* formula for the BDD size? Do some analysis. NOTE: addw16.src and addw64.src might not be able to finish in any reasonable amount of time on your machines. That's fine. Just report that they did not finish.
- ❑ **mplr2.src, mplr3.src, mplr4.src, mplr5.src, mplr6.src, mplw2.src, mplw3.src, mplw4.src, mplw5.src, mplw6.src [16 pts]** - Set of multipliers (2x2bit, 3x3bit, 4x4bit, 5x5bit, 6x6bit) with two different variable orderings. Report the total size of BDDs. Also tell if any one of the variable orderings is drastically better than the other and why? Again can you specify (even *approximately*) how the BDD size grows with increasing multipier size? NOTE: mplr6.src and mplw6.src might not be able to finish on your machine. That's still fine. Just report it.
- ❑ **Primes calculation** - - more detail on thisone later. **[20 pts]**
- ❑ **An interesting test case of your own choosing [5 pts]**.

Comparisons with KBDD [4 pts]

Run the source files compare.src and muxw2.src on KBDD. Compare the BDDs reported by YBDD and KBDD. Remember KBDD has negated pointers. Comment on the differences.

Time Allocation [4 pts]

How much time did you spend on each of the major tasks? What did you think was most difficult about the project? What debugging techniques did you use?

...and, that's it. Like we said: we're asking you to run a **lot** of test cases, and do a bunch of basic analysis. Don't put this off to the very last minute, and **do** pay attention to doing the things we ask.

Appendix: More Notes on the KBDD Package

You will use this BDD package both for homeworks and as a point of comparison in this project. This BDD package has multi-rooted DAGs with automatic garbage collection and negation arcs. Karl Brace got a PhD here in ECE for implementing this package (KBDD = Karl's BDD). The command line interface for KBDD is much more sophisticated than the one in YBDD. Now an eval command allows multiple operations. We can also quantify, find the satisfying set, define adders and muxes in a short hand notation and much more.

Executable: **/afs/ece/class/ee760/bin/kbdd**

Detailed man page: **/afs/ece/class/ee760/man/man1/kbdd.1**

Some other KBDD documentation (including a lab/workshop): **/afs/ece/class/ee760/doc**

To view the man page:

cd /afs/ece/class/ee760/man

man kbdd

To run KBDD in interactive mode:

- Type /afs/ece/class/ee760/bin/kbdd
- KBDD will return with a KBDD prompt. Enter one KBDD command per line.

To run KBDD in batch mode:

- Create a source file of KBDD commands (assume foo.src).
- Type /afs/ece/class/ee760/bin/kbdd foo.src

1. Several KBDD Examples

The source files for these examples can be found in **/afs/ece/class/ee760/proj1/ex.kbdd**

EXAMPLE 1:

Note that in KBDD, an eval expression allows multiple operations. Comments begin with the '#' sign. Following is the script file for a homework problem:

```
# This is the "fix the adder with quantification" problem from homework 1.
# Define Boolean variables.
#   a,b,Cin - function inputs.
#   d0,d1,d2,d3 - Mux data inputs.
boolean a b Cin d0 d1 d2 d3
# Desired function.
eval fd a&b + (a+b)&Cin
# Function with MUX.
eval t  a&b&d3 + a&!b&d2 + !a&b&d1 + !a&!b&d0
eval fm a&b + t&Cin
```

```
# Generate the Z function which is the XNOR of both.
eval Z !(fd^fm)
# Generate a new function Z1 which is
# the consensus of Z wrt a, b, Cin
quantify u Z1 Z a b Cin
# Find the satisfying set for Z1.
satisfy Z1
quit
```

To run this file type:

/afs/ece/class/ee760/bin/kbdd /afs/ece/class/ee760/proj1/ex.kbdd/hw1.prob7.src

KBDD will print the following:

```
KBDD: # This is the "fix the adder with quantification" problem from homework 1.
KBDD: # Define Boolean variables.
KBDD: #   a,b,Cin - function inputs.
KBDD: #   d0,d1,d2,d3 - Mux data inputs.
KBDD: boolean a b Cin d0 d1 d2 d3
KBDD: # Desired function.
KBDD: eval fd a&b + (a+b)&Cin
fd: a&b + (a+b)&Cin
KBDD: # Function with MUX.
KBDD: eval t  a&b&d3 + a&!b&d2 + !a&b&d1 + !a&!b&d0
t: a&b&d3 + a&!b&d2 + !a&b&d1 + !a&!b&d0
KBDD: eval fm a&b + t&Cin
fm: a&b + t&Cin
KBDD: # Generate the Z function which is the XNOR of both.
KBDD: eval Z !(fd^fm)
Z: !(fd^fm)
KBDD: # Generate a new function Z1 which is
KBDD: # the consensus of Z wrt a, b, Cin
KBDD: quantify u Z1 Z a b Cin
KBDD: # Find the satisfying set for Z1.
KBDD: satisfy Z1
Variables: d0 d1 d2
011
KBDD: quit
```

Note that the satisfying set is 011-, thus defining both an OR and an XOR gate.

EXAMPLE 2:

Here is an example of the extended naming notation. The notation has been used to define two different variable orderings for an adder. Subsequently both 4 and 8 bit adders have been defined with each ordering.

```
# Using extended naming notation to describe
# various variable orderings for adders.
###############################################
# Short form for a_7 ... a_0 b_7 ... b_0 cin
boolean a_[7..0] b_[7..0] cin
# Define a 4 bit adder
adder 4 s4_[4..0] a_[3..0] b_[3..0] cin
size s4_[4..0]
# Define an 8 bit adder
adder 8 s8_[8..0] a_[7..0] b_[7..0] cin
size s8_[8..0]
###############################################
# Short form for aa_7 bb_7 ... aa_0 bb_0 ccin
boolean {aa,bb}_[7..0] ccin
# Define a 4 bit adder
adder 4 ss4_[4..0] aa_[3..0] bb_[3..0] ccin
size ss4_[4..0]
# Define an 8 bit adder
adder 8 ss8_[8..0] aa_[7..0] bb_[7..0] ccin
size ss8_[8..0]
quit
```

To run this file type: **/afs/ece/class/ee760/bin/kbdd /afs/ece/class/ee760/proj1/ex.kbdd/adder.src**

KBDD will print the following:

```
KBDD: # Using extended naming notation to describe
KBDD: # various variable orderings for adders.
KBDD: ###############################################
KBDD: # Short form for a_7 ... a_0 b_7 ... b_0 cin
KBDD: boolean a_[7..0] b_[7..0] cin
KBDD: # Define a 4 bit adder
KBDD: adder 4 s4_[4..0] a_[3..0] b_[3..0] cin
KBDD: size s4_[4..0]
size [ s4_4 s4_3 s4_2 s4_1 s4_0 ]
76
KBDD: # Define an 8 bit adder
KBDD: adder 8 s8_[8..0] a_[7..0] b_[7..0] cin
KBDD: size s8_[8..0]
size [ s8_8 s8_7 s8_6 s8_5 s8_4 s8_3 s8_2 s8_1 s8_0 ]
```

1276

KBDD: ##

KBDD: # Short form for aa_7 bb_7 ... aa_0 bb_0 ccin

KBDD: boolean {aa,bb}_[7..0] ccin

KBDD: # Define a 4 bit adder

KBDD: adder 4 ss4_[4..0] aa_[3..0] bb_[3..0] ccin

KBDD: size ss4_[4..0]

size [ss4_4 ss4_3 ss4_2 ss4_1 ss4_0]

21

KBDD: # Define an 8 bit adder

KBDD: adder 8 ss8_[8..0] aa_[7..0] bb_[7..0] ccin

KBDD: size ss8_[8..0]

size [ss8_8 ss8_7 ss8_6 ss8_5 ss8_4 ss8_3 ss8_2 ss8_1 ss8_0]

41

KBDD: quit

Note the exponential and linear growth in the BDD size for the two different orderings.

KBDD Quick Reference

Command	Description
boolean *var* …	Declare variables and variable ordering
Extended naming	
var[*m* .. *n*]	Numeric range (ascending or descending)
{*s1,s2,…*}	Enumeration
evaluate *dest expr*	*dest* := bdd for boolean expression *expr*
Operations	(decreasing precedence)
!	Complement
^	Exclusive-Or
&	And
+	Or
bdd *funct*	Print BDD DAG as lisp-like representation
sop *funct*	Print sum-of-products representation of *funct*
satisfy *funct*	Print all satisfying variable assignments of *funct*
verify *f1 f2*	Verify that two functions *f1 f2* are equivalent
size *funct* …	Compute total BDD nodes for set of functions
replace *dest funct var replace*	Functional composition: *dest* := *funct* with variable *var* replaced by *replace* function output
quantify [**u**|**e**] *dest funct var …*	*dest* := Quantification of function *funct* over variables *var …*
e	Existential quantification is done
u	Universal quantification is done
adder *n Cout Sums As Bs* Cin	Compute functions for n -bit adder
n	Word size
Cout	Carry output
Sums	Destinations for sum outputs: *Sum.n … Sum.0*
As	A inputs: *A.n-1 … A.2 A.1 A.0*
Bs	B inputs: *B.n-1 … B.2 B.1 B.0*
Cin	Carry input
alu181 *Cout Fs M Ss Cin As Bs*	Compute functions for ‘181 TTL ALU
Cout	Destination for carry output
Fs	Destinations for function outputs: *F.3 F.2 F.1 F.0*
M	Mode input
Ss	Operation inputs: *S.3 S.2 S.1 S.0*
Cin	Carry input function
As	A inputs: *A.3 A.2 A.1 A.0*
Bs	B inputs: *B.3 B.2 B.1 B.0*
mux *n Out Sels Ins*	Compute functions for 2n-bit multiplexor
n	Word size
Out	Destination for output function
Sels	Control inputs: *Sel.n-1 … Sel.1 Sel.0*
Ins	Data inputs: *In.2n – 1 … In.1 In.0*
quit	Exit KBDD