VLSI CAD: LOGIC TO LAYOUT:
kbdd Software Package

Rob A. Rutenbar
University of Illinois at Champaign-Urbana

kbdd is a BDD calculator done by Prof. Randy Bryant's[1] research group at Carnegie Mellon University (http://www.cs.cmu.edu/~bryant/). **kbdd** has all the operators you would want to use to manipulate Boolean functions, and a simple command line interface to type in functions, etc. We have installed kbdd in the Coursera cloud, and have created an interface to allow you to upload text files to solve useful Boolean computational problems. You upload your text file of commands; we run kbdd in the cloud; a textfile of kbdd outputs is presented back to you on your Coursera class page. You will use kbdd to do some computations that are too big to do by hand.

As a starting point, if you type the *help* command into *kbdd*, this is what will be printed, as a "quick reference" to what commands *kbdd* offers:

```
? [<command>]                  -- Print information about command
adder <n> <sum> <a> <b> <cin>  -- generate formulas for n-bit adder
alu181 <cout> <sum> <m> <s> <a> <b> <cin>
                               -- generate functions for 181 alu
bdd <f>                        -- print out representation for formula
boolean <v1>..<vn>             -- declare boolean variables
echo                           -- rest of line
evaluate <f> <exp>             -- create formula from expression
free <f1> ... <fn>             -- free formula(s)
garbage                        -- force bdd package to do garbage collection
implies <f1> <f2>              -- f1 imply f2 ?
ite <fd> <fi> <ft> <fe>        -- perform if fi then ft else fe
limit <n>                      -- set memory limit for bdds to be n bytes.
mux <n> <out> <sel> <in>       -- generate formulas for n to 2^n bit mux
quantify [<eu>] <fd> <fs> <v1>..<vn>
                               -- quantify formula over variables
quit                           -- exit program
replace <fd> <fs> <v1> <f1>..<vn> <fn>
                               -- replace variable vi with function fi
satisfy <f>                    -- print var assignments that satisfy formula
show [<command>]               -- List hidden commands/Show in menu.
size <f1>..<fn>                -- print number of bdd nodes under formulas
sop <f>                        -- print sop representation of formula
source <file>                  -- Read commands from file
switch [<switch1>:<val1>..<switchn>:<valn>]
                               -- Set/check run time switches
totalsize                      -- print total number of nodes in bdd
verify <f1> <f2>               -- verify that two formulas are equal
```

[1] We gratefully acknowledge Prof. Randy Bryant of CMU for his permission to use his kbdd software package for our University of Illinois MOOC on VLSI CAD.

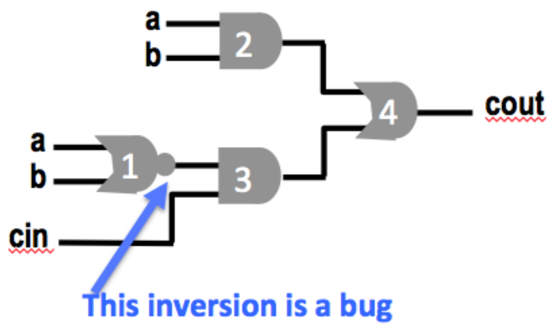
Here is this same information, but with a bit more explanation about what is happening with each command available in **kbdd**:

kbdd Quick Reference Information

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*
	Can also type **eval** for short here
Operations	(decreasing precedence)
(*expr***)**	Parentheses work as usual in any expression
!	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*	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
mux *n Out Sels Ins*	Compute functions for 2^n-bit multiplexor
n	Word size
Out	Destination for output function
Sels	Control inputs: *Sel.n-1 ... Sel.1 Sel.0*
Ins	Data inputs: *In.*2^{n-1} *... In.1 In.0*
help	print a quick reference of kbdd commands
# *anything*	This line is a comment for readability
quit	Exit KBDD

It is helpful to show a concrete example of a BDD computation that kbdd can do. Let us consider another version of the network repair problem we have already described in lecture.

Consider the logic network below. In this example, a simple 1-bit adder circuit for the carry-out *cout* has a NOR gate incorrectly where there should be an OR gate, like this:



We can employ the repair steps, via quantification, etc., as in the lecture video and notes on Computational Boolean Algebra. The basic recipe is:

1. Build a correct BDD for the function we want, *Cout*.
2. Build a BDD for the incorrect logic, but replace the suspect gate – the input NOR – with a 4:1 multiplexor (MUX), with new inputs *d0 d1 d2 d3* as the MUX data inputs.
3. Exclusive NOR (EXNOR) the correct and to-be-repaired functions. This new function *Z* can be satisfied only if the *d* inputs are set correctly to let the MUX mimic the correct gate.
4. Universally quantify away the real logic input (*a,b,cin*) here, so that the *Z* function depends only on the MUX d inputs.
5. Check is there is a satisfying assignment to the d inputs; if so, we have found a viable gate repair.

Pleasantly enough, this is all quite easy in **kbdd.**

The following shows an example of a session with **kbdd**. Inputs are in normal font (these are what *you* would type into a plain textfile, and upload to our Coursera cloud-based version of **kbdd**). **kbdd** outputs are **blue**, **kbdd's** prompts for input shown in bold as **KBDD**:

Kbdd Example Session for Adder Carry-out Repair

```
KBDD: # input variables
KBDD: boolean a b cin d0 d1 d2 d3
KBDD: #
KBDD: # define the correct equation for the adder’s carry out
KBDD: eval cout a&b + (a+b)&cin
cout: a&b + (a+b)&cin
KBDD: #
KBDD: # define the incorrect version of this equation (just for fun)
KBDD: eval wrong a&b + (!(a&b))&cin
wrong: a&b + (!(a&b))&cin
KBDD: #
KBDD: # define the to-be-repaired version with the MUX
KBDD: eval repair a&b + (d0&!a&!b + d1&!a&b + d2&a&!b + d3&a&b)&cin
repair: a&b + (d0&!a&!b + d1&!a&b + d2&a&!b + d3&a&b)&cin
KBDD: #
KBDD: # make the Z function that compares the right version of
KBDD: # the network and the version with the MUX replacing the
KBDD: # suspect gate  (this is EXNOR of cout and repair functions)
KBDD: eval Z repair&cout + !repair&!cout
Z: repair&cout + !repair&!cout
KBDD: # universally quantify away the non-mux vars: a b cin
KBDD: quantify u ForallZ  Z a b cin
KBDD: #
KBDD: # let’s ask kbdd to show an equation for this quantified function
KBDD: sop ForallZ
  !d0 & d1 & d2
KBDD: #
KBDD: # what values of the d’s make this function == 1?
KBDD: satisfy ForallZ
Variables: d0 d1 d2
011
KBDD: #
KBDD: # that’s it!
KBDD: quit
%
```

As always, it is important to use your brain to analyze what the software tool is telling you. Observer that kbdd says that a satisfying assignment of the MUX inputs is this:

```
Variables: d0 d1 d2
011
```

This means d3's value does not matter. So, in fact, there are two solutions: d0 d1 d2 d3 = 0111, and 0110. These specify and OR and an EXOR gate, respectively, as feasible repairs of the network.

Usage Notes for kbdd: Adders

Using the built-in functions like adders and the extended range notation

kbdd has basic n-bit adders built in, so this is very convenient. But, there is a bug in the online "help" output for this version of kbdd, for the syntax for the adder command. The example shown here clears up exactly how to use this:

```
KBDD: #declare inputs to a 4 bit adder
KBDD: boolean a[3..0] b[3..0] c0
KBDD: # now, build all the outputs of the 4b adder
KBDD: adder 4 c4 s[3..0] a[3..0] b[3..0] c0
KBDD: # now DRAW the BDD itself in text form
KBDD: # here is the low order sum bit s0
KBDD: bdd s0
(a0:1753429896
   (b0:1753429864
      (c0:1753429784)
      ![c0:1753429784])
   ![b0:1753429864])
KBDD: # now ask how BIG this s0 BDD is
KBDD: size s0
size [ s0 ]
3
```

Usage Notes for kbdd: Graph Structure

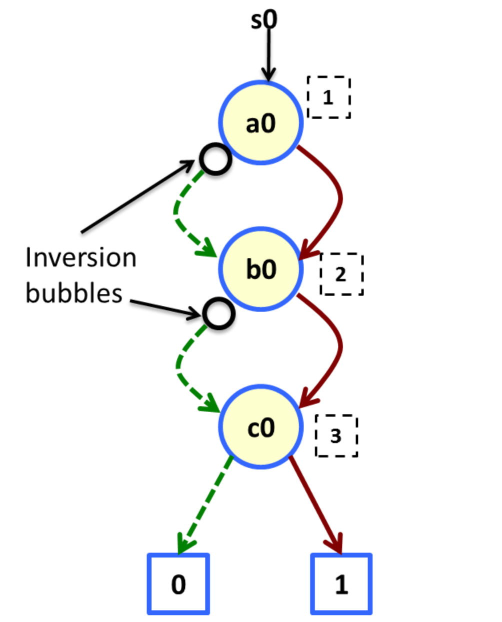
It is important to note that **kbdd** is using an additional "trick" that we did not discuss in lecture. This trick is something called **negation arcs**. In digital design, suppose we have a function **F**, and we want to build logic for the complement **F'**. We might optimize **F'** directly as gates. Or, we might just build **F** itself, and then send its output through a simple *inverter* gate. We would like to choose the option that gives us the fewest logic gates. One can apply a similar idea to BDDs. Sometimes, it is easiest to build the BDD for **F'** directly. But sometimes it is easier to just build **F**, and then indicate in the data structure that we have "inverted it". This is the idea of the negation arc: it is exactly like a simple inverter gate. We put an inversion bubble on the edge leaving a BDD node, and the bubble means "interpret the BDD to which this edge points as being inverted". It turns out that a simple set of Shannon factor tricks, and some basic DeMorgan complement laws, can be used to build the rules for how this can work. Nicely enough, one again creates *canonical* structures: a function **F** makes one and only one BDD, and always the *same* BDD. The complement bubbles just arrange themselves in the right places. The big advantage is that one can save, on average, about half the nodes in the BDD. The big disadvantage (and this is rather minor) is that BDDs become rather hard to "read", visually, as graphs.

For our BDD example, the printout with parentheses and big numbers, has this interpretation:

- **Letters**: these are the variable name
- **Numbers**: these are the actual BDD node addresses in memory
- **"!":** this is an inversion bubble on a negation arc
- **Indentation**: each indent means "we go down one level in the BDD graph"; children of a particular node are listed on lines with the same indent under their parents.
- **Ordering**: We first list the high-child (variable=1) on the *first* indented line under a BDD node. We list the low-child (variable=0) on the *last* indented line with the *same indent*, under a BDD node. If you see an indent anywhere, it means "this is the child of the thing above, one level less indented).
- **Constants**: kbdd will print **"[0]"** or **"[1]"** when an internal node has a child that one of the two constants. However, for nodes which have the "standard" children at the very bottom of the tree – that is a variable "x" whose high-child is **[1]** and low-child is **[0]** – **kbdd** omits printing these child nodes.

So, if we return to the BDD printout from our adder, this is the actual graph:

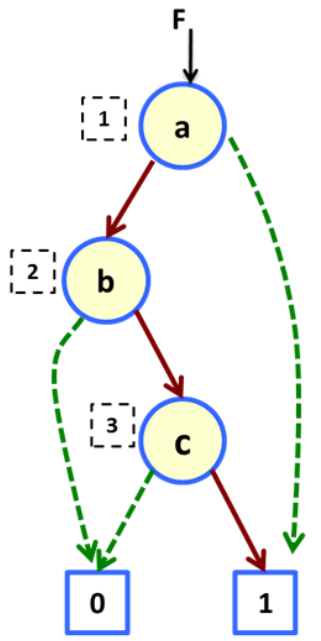
```
(a0:1753429896
   (b0:1753429864
      (c0:1753429784)
      ![c0:1753429784])
   ![b0:1753429864])
```



¦#¦: for describing the diagram in "alternate text" for visually impaired course participants

We can also show another example to illustrate that we don't always need negation arcs. This BDD has a more familiar structure:

```
KBDD: boolean a b c
KBDD: eval F !a + b&c
F: !a + b&c
KBDD: bdd F
(a:1812523160
   (b:1812523176
      (c:1812523096)
      [0])
   [1])
```



¦#¦: for describing the diagram in "alternate text" for visually impaired course participants

Content-type: text/html

KBDD

Section: User Commands (1)
Updated: 01/15/90
Index Return to Main Contents

NAME

kbdd - an interactive or batch Boolean manipulation interface

SYNOPSIS

kbdd [srcfile]

DESCRIPTION

The package allows you to declare Boolean variables and perform Boolean operations to create new Boolean formulas. The formulas have string names. A variable is just a special formula. Formulas may be tested for equality to each other. The formula is represented as a binary decision diagram (BDD). A formula may be printed out in SOP form, or as a BDD. The size of the BDD for a set of formulas can be printed.

Commands are one per line and may be abbreviated to a unique prefix. Since commands are kept sorted in alphabetical order, shorter abbreviations will pick the first match. As an example, "q" will be interpreted as "quantify" and not "quit". Commands are typed at the "KBDD: " prompt or sourced from srcfile, or sourced from the file specified in a "source" command.

A note about variable orders. In a BDD, the order of the input variables is critical to a reasonable representation. This package only reads the order that the user provides, so do put some effort into finding a good order for your application.

EXTENDED NAMING NOTATION

There is often a need to describe a large number of variables or formulas while using the BDD package. The package has an extended naming notation that can be used to describe a set of names. The notation divides the set of names into the prefix, midfix and postfix sets. The prefix is a single string or a set of space/comma separated strings within curly braces. The midfix is an optional single string. The postfix is a set of space/comma separated positive integer or positive integer-ranges within square brackets. Ranges are specified as integers separated by two periods m..n . The range can be specified to be in either ascending or descending order. This extended notation is expanded inline to give a set of names. Each name is the result of a concatentation of a prefix string, the midfix, and a postfix integer. The ordering of strings in the prefix and ordering of integers in the postfix is important since it defines the order in which the names are generated.

Some examples of extended notations and their inline expansions are:

```
{a,b,c}_[0..2]       =          a_0 b_0 c_0 a_1 b_1 c_1 a_2 b_2 c_2
{a,b,c}_[2..0]       =          a_2 b_2 c_2 a_1 b_1 c_1 a_0 b_0 c_0
{s,r}.[1,3,5,7]      =          s.1 r.1 s.3 r.3 s.5 r.5 s.7 r.7
{a b}[1 2..4 6]      =          a1 b1 a2 b2 a3 b3 a4 b4 a6 b6
input[8..4]          =          input8 input7 input6 input5 input4
```

Note that the prefix strings cannot have the characters `**}**', `**,**'. Strings with `**[**' have to be enclosed within curly braces.

COMMANDS

```
quit
bye
exit
(control D)
```

Terminate the program. If encountered while sourcing a file of commands, the commands quit, bye and exit terminate the program, while EOF returns control to the interactive mode.

```
boolean u v w...
```

Declare new variables named u, v, w... The variable order in the BDD is the order in which the boolean commands occur. Multiple boolean commands are fine, or they can be combined as shown, in which case the order is left to right on that line.

```
insert_after newbool oldbool
```

Insert a new variable newbool immediately after oldbool in the variable order. This command is initially hidden from the help menu. See the show command for more details.

```
evaluate f exp
```

Create a new formula f from the expression exp. The expression can contain !, ^, &, + representing operations NOT, XOR, AND and OR (in descending order of precedence). An example:

```
eval foo a&!b + !(a^c)
```

If the formula foo already existed, then a warning is printed and the old formula foo is freed first. If foo was an input variable, it is an error.

```
ite f a b c
```

Assign the formula f to be "If a then b else c"

```
help [command]
?    [command]
```

List the set of useful commands. If a command is specified then help information is given for that particular command. Otherwise help lists all commands currently in the help menu. Certain commands are hidden from the help menu. Look at the show and hide commands for more details.

```
echo rest of line
# rest of line
```

Print the rest of the line after the command. There must be a space after the "#".

```
garbage
```

Force the BDD package to collect garbage. See bdd(3), the function bdd_garbage().

```
size f g h ...
```

Print the number of BDD nodes under the formulas f, g, h... This accounts for sharing, so is less than or equal to the sum of the individual sizes.

```
totalsize
```

Print the number of nodes total in the BDD.

```
free f g h ...
```

Dispose of formulas f, g, h ... Frees up the space for these formulas.

```
implies f g
```

Does f imply g? Replies with "1 yes" or "0 no".

```
limit n
```

Set the memory limit for the BDD to be n bytes. If n is 0, then simply report the current limit. Reports the resulting limit in either case "limit is n".

```
replace fdest fsource var1 f1 var2 f2 . . .
```

Create formula fdest to be: In formula fsource replace variable var1 with formula f1, var2 with formula var2, etc.

```
source srcfile
```

Begin taking commands from the file srcfile. End with EOF to return to interactive, or with quit, exit or bye to quit the program. A kbdd command line argument is an implicit source command.

```
verify f g
```

Verify that f and g are equal, replies with either "1 verify ok" if f and g are equal, otherwise "0 verify failed".

```
adder n  sum_n ... sum_0  a_n-1 ... a_0  b_n-1....b_0  carry
```

Generate formulas for an n-bit adder. n is an integer, the number of bits in the adder. The command creates the formulas named sum_n ... sum_0 as the output of the adder, where sum_0 is the LSB and sum_n is the carry out. a_n-1 ... a_0 , b_n-1 ... b_0 are previously declared input variables or regular formulas. a_n-1, b_n-1 serve as the MSB and a_0, b_0 as the LSB for the adder. carry is a previously declared formula, variable, or 0 or 1, and is the carry input for the adder.

Here is an example to add foo and bar to create sum and measure the combined size of the sum and carry out (sum.2).

```
bool initcarry {foo,bar}.[0,1]
adder 2 sum.[2..0] foo.[1..0] bar.[1..0] initcarry
size sum.[2..0]
```

```
alu181  cout  f_3 ... f_0  m  s_3 ... s_0  a_3 ... a_0  b_3 ... b_0  cin
```

Generate formulas for the carry output (cout) and function (f) of a '181 ALU. This is a 4 bit alu. m is a previously declared formula, variable, or 0 or 1, which represents the mode indicators for the ALU. s_3 ... s_0 (control signals), a_3 ... a_0 (data word A), b_3 ... b_0 (data word B) are either input variables or regular formulas. cin is a previously declared formula, variables, or 0 or 1, which represents the carry input for the ALU. This command creates the formulas named cout and f_3 ... f_0.

```
mux  n  out  select_n-1 ... select_0  in_2^n-1 ... in_0
```

Generate formulas for an n to 2^n bit mux. Generate the formula for the output (out) of the mux. select_n-1 ... select_0 (select lines) and in_2^n-1 ... in_0 are either input variables or regular formulas.

Here is an example of a mux. foo is the root name of the 2 select lines and bar is the root name of the data lines, and the size of the resulting formula muxout is calculated.

```
bool foo_[1,0] bar_[3..0]
mux 2 muxout foo_[1..0] bar_[3..0]
size muxout
```

```
quant [eu] dest f b1 b2 b3...
```

Perform the existential (e) or universal (u) quantification of f with respect to the variables bi and put the result in dest.

```
sop f
```

Print out an SOP representation of f. The representation is not minimized very well.

```
satisfy f
```

Print out all variable assignments that satisfy f.

```
bdd f
```

Print out a representation of the BDD for f. It is best described by an example. Suppose the BDD is (If node_w then (if y then 1 else x) else not (if z then x else 0). This would look like:

```
(node_w:1
   (y:2
      [1]
      (x:3))
   !(z:4
      [x:3]
      [0]))
```

Nodes are surrounded by parentheses or brackets. The node names come first, then a colon and an arbitrary id for this node for this printing. The first time a node is listed, it is surrounded by parentheses and its structure is given, subsequent times it is identified only by it's variable:id and is surrounded by brackets. The constants 0 and 1 are treated as if they have previously been listed, so are in brackets. The "!" indicates that the node is complemented, think of it as an inverter on that edge of the graph.

```
switch [switch_1:value_1 switch_2:value_2 ...]
```

Set/Check run time switches. Without any argument the command lists all the switches with their current values. With arguments the command is used to set switches at run time. The current list of switches with their default values is as follows:

```
error:100               -- Max. number errors before forced exit
inline:0                -- Echo inline expansion?
quiet:0                 -- Run in quiet mode without echoing command lines?
```

```
show [command]
```

List/show hidden commands. Some of the commands are hidden from the help menu so as to make the help menu more readable. The show command without any arguments lists all the currently hidden command. With an argument, the show command can be used to bring back a hidden command to help menu. Alternatively any command currently in the help menu can be hidden with the use of the hide command.

```
hide command
```

Hide command from the help menu. If the help menu is cluttered with a large number of commands then the hide command can be used to remove any command from the help menu. Alternatively the show command can be used to bring back commands to the help menu.

```
gencofactor dest f c
```

Create dest as the generalized cofactor of f with respect to care set c. This command is initially hidden from the help menu. See the show command for more details.

```
stats
```

Print obscure statistical information about the BDD. This command is initially hidden from the help menu. See the show command for more details.

```
markout f
```

Declare that f is a primary output. This command is initially hidden from the help menu. See the show command for more details.

```
po
```

Print a list of the names of the primary outputs, separated by newlines, in the order they were declared with the markout command. This command is initially hidden from the help menu. See the show command for more details.

```
pi
```

Print a list of the names of the input variables, separated by newlines, in the order they were declared with the boolean command. This command is initially hidden from the help menu. See the show command for more details.

```
read_blif blif_file [01]
```

Make a BDD for all of the nodes in a .blif file. If the second argument is a "0", intermediate nodes are bdd_freed leaving only primary outputs defined. If the second argument is a "1" then BDD's for all nodes remain. This command is initially hidden from the help menu. See the show command for more details.

BUGS

Probably lots of bugs, mostly in handling syntax errors. It is best to create source files automatically to avoid syntax errors.

AUTHOR

```
Karl S. Brace
```

Index

This document was created by man2html, using the manual pages.
Time: 21:21:20 GMT, March 15, 2013