|  |  |  |  |
| --- | --- | --- | --- |
| Date: | **03-08-2020** | Name: | **Varun G Shetty** |
| Course: | **Coursera** | USN: | **4AL17EC093** |
| Topic: | VLSI CAD part 1 | Semester & Section: | **6th & ‘B’** |
| GitHub Repository: | **Varunshetty4** |  |  |

**Report-**

KBDD:

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

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 or (Sum.n)

Sums Destinations for sum outputs: Sum.n-1 ... 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 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

Help print a quick reference of kbdd commands

# anything This line is a comment for readability

quit Exit KBDD

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 iscuss 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 highchild 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])

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])