



VLSI CAD: LOGIC TO LAYOUT: Programming Assignment 2: **Serious BDDs**

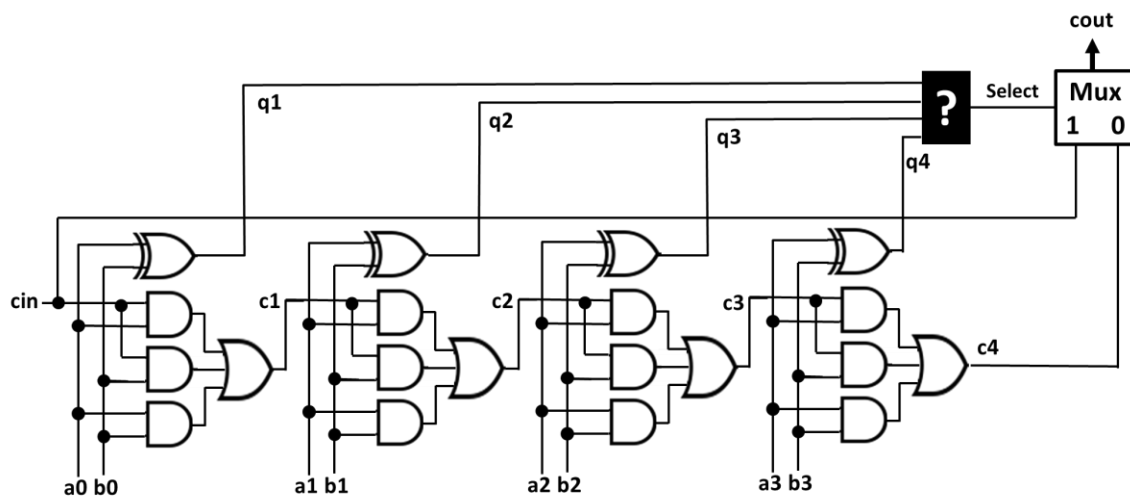
Rob A. Rutenbar
University of Illinois at Champaign-Urbana

You got a chance to try **kbdd** as part of Problem Set #2, where you ran a relatively small verification of two combinational logic networks. In this “programming” assignment, you get to use **kbdd** again, but now on some problems that are too big to do by hand. You also get to think through a novel BDD application.

The assignment is in two parts, both focused on network repair applications, since these exercise a lot of the power of a good BDD package. You will write and exercise **kbdd** scripts to complete this programming assignment.

Part 1: Network Gate Repair

This part of the problem revisits the method we taught in lectures in Week2, and which we illustrated with a small example in the Tutorial for **kbdd**. Now, we want you to do this again – but on a network too complex to do any other way. Consider this new logic network:



We are showing just the carry chain (from carry-in “**cin**” to carry-out “**cout**”) for a 4 bit adder, which adds **a3,a2,a1,a0** to **b3,b2,b1,b0**. This is a more sophisticated architecture called a *carry-bypass* adder. The idea is to use internal information generated by each stage of the addition process to try to *predict* the output carry as

early as possible. In the best case, the predictor is correct and the MUX selects the input carry **cin** as the output carry (it is merely propagated). At worst, the multiplexor (**Mux**) has to wait for the “real” carry, **c4**, to ripple out to generate **cout**.

In this design, we believe the highlighted gate labeled “?” has been implemented incorrectly. Your job is *repair* this network and tell us the proper gate. The method from the lecture should work here – but this is a bigger network, you cannot just do this by hand.

Do This:

- Build a “correct” version of the circuit. This means you just need an ordinary 4 bit adder with a carry-in and a carry-out from the high-order bit. You can do this however you like – but do recall that **kbdd** has an adder primitive built in.
(Hint: it looks like this: **adder 4 cout_gold sum[3..0] a[3..0] b[3..0] cin** Note you still have to list the sum output bits even though you don’t need to use them for this problem).
- Replace the “?” gate with a multiplexor, to emulate the gate you need to find. Note that you are working to repair a 4-input gate, with inputs **q1,q2,q3,q4**. So, you need a 16:1 mux, since any truth table for this gate is a function of 4 variables, so it has 2^4 rows you need to solve for. You can write the equation for this in detail yourself, or you can recall that **kbdd** also has a MUX primitive built in.
(Hint: it looks like this: **mux 4 Select q1 q2 q3 q4 d[15..0])**
- Set up the BDD-based gate repair as we specified in the lecture: replace the gate with the mux; EXNOR the correct circuit with this incorrect circuit with the mux inserted; quantify away the right variables; satisfy this result; look **very carefully** at this result. Are all the variables still here? What value(s) – if any – satisfy this result?

Grading Details [40 points]:

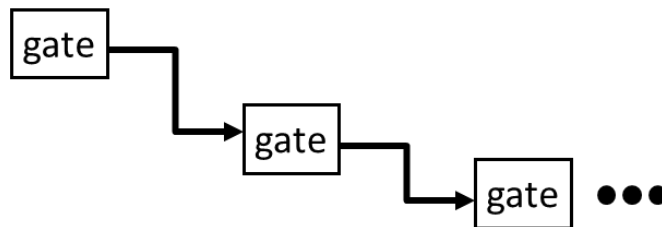
You can run **kbdd** however you want to solve this problem. We will create an “auxiliary quiz” (which looks just like a Problem Set) to let you answer questions about this result. You can submit as many times as you like – but for this quiz, you have to wait until the quiz is past the deadline to see the final result. (There’s just not that many useful answers here...). See details at the end of this writeup for more about the Auxiliary Quiz.

Note: Both Part 1 and Part 2 questions will appear in the Auxiliary Quiz. So, we recommend you complete **both parts** before you try the Quiz.

Part 2: Unknown Inversion Find-and-Repair

A common error in a large logic network is an *incorrect inversion*: on some internal wire, there is error that can be corrected by adding exactly one inverter to this wire in the network. But, we do *not know* where the error is. It turns out we can still use a BDD to solve this new, seemingly more difficult problem. We can find the wrong wire and then invert it to repair things.

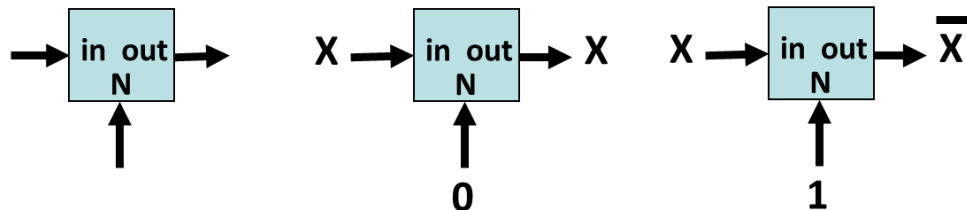
It is easiest to see the idea – which you need to work out – on a small design. Suppose we have this small section of a larger logic network, and we want to ask the simpler question: which one of these two wires is incorrectly inverted?



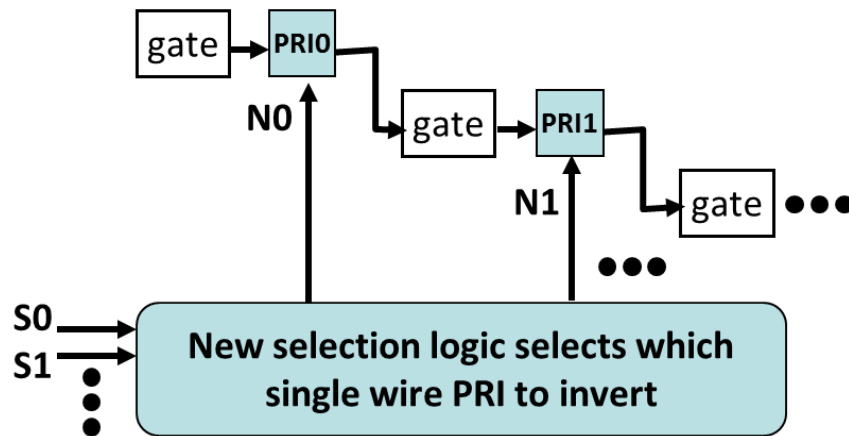
Then we modify the network as follows:

- On each suspect wire, we insert a *programmable inverter (PRI)*. A programmable inverter is a small piece of logic (2 inputs, 1 output) that has this function: if the control input $N=0$, then $\text{out}=\text{in}$; if the control input $N=1$, then $\text{out}=\text{in}'$. In other words, the N signal *negates* the input, when $N=1$.

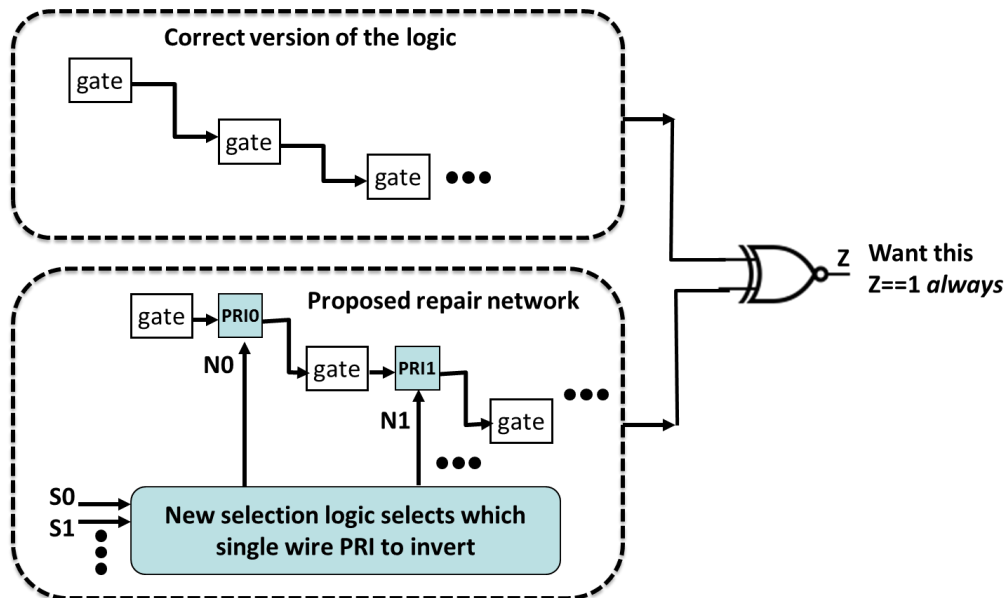
Programmable Inverter



- We then build another piece of new logic that *selects* one – and *only* one! – of the programmable inverters, and sets its control signal $N_i = 1$. All the other programmable inverters are set to $N_j = 0$. This selection logic takes a new set of inputs the select which one of the PRI control inputs to set to 1. Suppose you had 16 potential wires to check for an inversion error. Then this selection logic has $\log_2(16) = 4$ inputs, S_3, S_2, S_1, S_0 , which we regard as a binary number, to select one PRI to enable. For example, if $S_3, S_2, S_1, S_0 = 1001$, then we will set $N_9 = 1$, and set all other $N_j = 0$, for $j \neq 9$. Let's call this version of the design, with the PRI's and the selection logic, the "proposed repair" network. This is shown below.

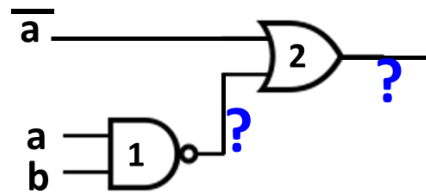


- Just as with our previous network repair methods, we need to have available some correct version of the logic. We connect the correct logic output, and our “proposed repair” version of the network, to an EXNOR gate. We want to solve for values of the selection inputs $S_{n-1}...S_2S_1S_0$ so that the output Z of the EXNOR is always 1. We use *quantification* again, and get rid of all inputs other than the selection inputs.

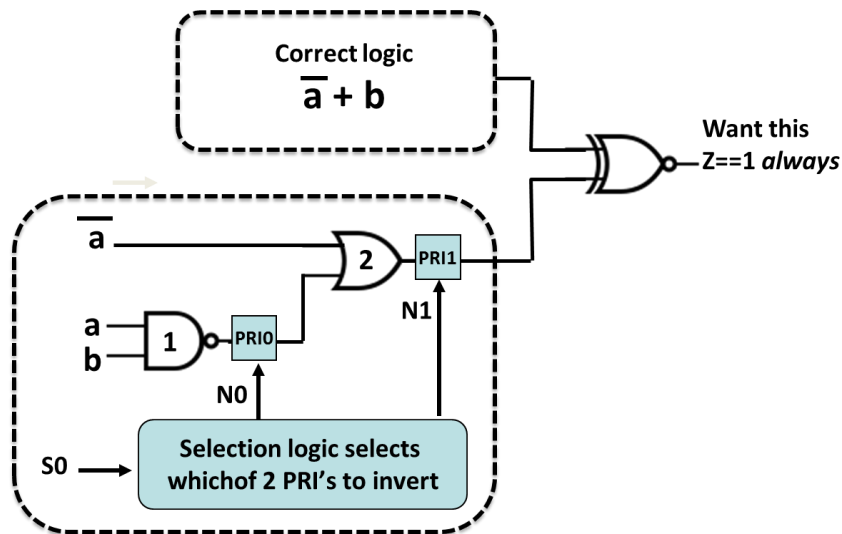


This is the general “recipe” for finding and correcting an inversion error on a wire. It is helpful to consider a very small, concrete example, to work through the details you need to determine to complete this problem. Here is a little example.

Suppose we have this small logic network. The wires with “?” labels are perhaps wrong. We want to determine which of these needs to have an inverter added.



We perform the steps described above: we add a PRI to each suspect wire; we create selection logic to choose 1 of exactly 2 PRIs to invert (so, there is just one select input **S0**); we create a new network which is the EXNOR of our “proposed repair” network and a correct version of the logic (which in this case is easy to see); we quantify away the inputs. This construction looks like this:



Our advice to you is: Figure out exactly how to solve this little problem *first*. When you understand all the pieces, then you can use **kbdd** to execute this repair recipe on a bigger design. This is what you will do for credit for this assignment.

Getting A Logic Network To Repair

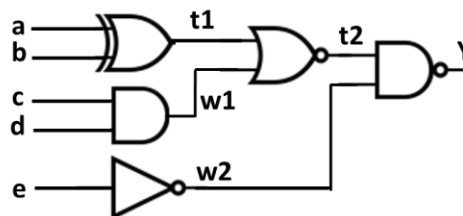
The first step to do Part 2 is to go to the course website, go look under Project 2 and go get your personal **random logic design**. We are using your user ID to generate a design that is specific to **you**. This means your friends will each get a **different** design. Just follow the directions on this web page, and you will receive two pieces of information you need to do this project:

1. **NETLIST**: A short, readable description of the logic network (this is called a “netlist”, by the way) on your private output webpage. This is the design with an error in it. **To get your netlist, please submit the file “submitThisFileToGetYourNetlist.txt” (in the starter files) to the grader, and your netlist will be shown on feedback page.** You should copy-and-paste this into a file, and then **draw** the logic for yourself, so you can see the design.
2. **KBBD PARTIAL SCRIPT**: We will give you a partial solution to this problem, that tells **kbdd** what to do. Your job is to fill in all the missing details, then upload this script to our checker program to grade it.

Here is a simple example of a logic gate netlist, so you can understand how they are put together:

```
input a b c d e
output Y
wires t1 t2 w1 w2
t1 = EXOR(a, b)
w1 = AND(c, d)
w2 = INV(e)
t2 = NOR(t1, w1)
Y = NAND(t2, w2)
```

If you draw this logic, it should look like this:



The logic netlist format is as follows:

- All inputs and outputs and intermediate values have variable names, that look like a name you could use in **kbdd**, e.g, **a** or **w7** or **Y_123**.
- The first line lists the inputs to the gate network.
- The next line lists the single output of the gate network.
- The next line lists all the internal wires in the gate network (i.e., gate outputs)
- The following lines each specify one logic gate, in the format
output = GATE(input, input) or output = GATE(input)
- Gate types can be: AND, OR, NAND, NOR, EXOR, EXNOR, INV (inverter)

Do this:

First, go the Project 2 webpage, and get your personal random netlist. Although random, each netlist has these properties:

- The inputs are: **a b c d e f g h**
- The output is: **Y_repair**
- There are several internal wires, and 16 total logic gates.
- There are exactly 8 wires where we believe an inversion error might exist. These wires are always named: **w0, w1, w2, w3, w4, w5, w6, w7**

Your task is to use **kbdd** to execute this repair recipe on this logic network. You should be connecting a PRI to each of these 8 wires, and you should connect the **Ni** selection logic output to the PRI you put on wire **wi**.

kbdd Script Submission Details

Pay close attention to these details here, since for this part, we will be grading your “live” **kbdd** script running out in the Coursera cloud.

- In addition to the random netlist, we will give you a *partial script* to start with, that matches this netlist. Do not change the commands in this script! This will define all the right input variables and the variable order. This will also define for you a *correct* version of the logic, which is called **Y**.
- Use **kbdd** to describe the equation of one Programmable Inverter. The inputs of this PRI are **pri_N** (control) and **pri_in** (data input). Output of this PRI is called **PRI**. We want you to fill in the equation (“**eval PRI something...**”) for one PRI.
- Use **kbdd** to describe the selection logic. The inputs to your selection logic are **S2,S1,S0** (3 inputs to select one of 8 outputs). The outputs of your selection logic are: **N7,N6,N5,N4,N3,N2,N1,N0**. You will write 8 lines of equations (“**eval Ni something...**”) to define each of these **Ni**.
- Use **kbdd** to build your “proposed repair” version of the Y function above – call it **Y_repair** – with the 8 PRIs inserted in the proper places. This is your personal random netlist logic, with the PRIs inserted on 8 wires.
- Use **kbdd** to build the **Z** function (we will name it capital “**Z**”!) as specified in this repair method. We will give a correct version of the function. This is just the EXNOR of the **Y** and **Y_repair** logic, to make **Z**.
- Use **kbdd** to quantify away the “right stuff”. Call this quantified function **Z_quantify**.
- Use **kbdd** to satisfy **Z_quantify**, and use these results to determine how to repair this network.
- Use **kbdd** to tell us the right wire to invert. We will define 8 new variables called wire0, wire1, ... wire7, and we will set all of them to constant 0. If you find that wire wK is the one to be inverted, then set wireK to be 1. So, for example, if you determine **w1** is the wire that needs the inverter, you add this line: **eval wire1 1** at the very end of the script.

You will upload this script to the Coursera server, and we will append some additional **kbdd** commands to compare your answer to the right answer, to grade it.

```
#HERE IS AN EXAMPLE OF A PARTIAL SCRIPT FOR PROGRAM 2 PART 2
#
# input variables - do NOT change this
boolean a b c d e f g h pri_in pri_N S2 S1 S0
#
# Here is the the correct logic function Y
eval Y we-put-the-correct-boolean-function-here-for-you
#
# YOU DO THIS: Define one programmable inverter block [8 points]
#           (Signal input = pri_in   Output = PRI
#           Inversion control signal = pri_N)
eval PRI you-fill-this-part-in
#
# YOU DO THIS: Define Selection Logic [8 points]
#           (3 select inputs, 8 PRI control outputs Ni)
eval N0 you-fill-this-part-in
eval N1 you-fill-this-part-in
eval N2 you-fill-this-part-in
eval N3 you-fill-this-part-in
eval N4 you-fill-this-part-in
eval N5 you-fill-this-part-in
eval N6 you-fill-this-part-in
eval N7 you-fill-this-part-in
# YOU DO THIS: Build a correct implementation of the Y network
#           with 8 programmable inverters (PRIs) embedded in it.
#           Call it Y_repair.
#Hint: Build this one one gate at a time from the netlist.
#           16 gates → 16 lines of Evals.
#           Embed each PRI in equation for gate output it inverts.
#
eval t1 you-fill-this-part-in
eval w0 you-fill-this-part-in
eval t2 you-fill-this-part-in
eval w1 you-fill-this-part-in
eval w2 you-fill-this-part-in
eval w3 you-fill-this-part-in
eval t3 you-fill-this-part-in
eval t4 you-fill-this-part-in
eval w4 you-fill-this-part-in
eval t5 you-fill-this-part-in
eval t6 you-fill-this-part-in
eval w5 you-fill-this-part-in
eval w6 you-fill-this-part-in
eval t7 you-fill-this-part-in
eval w7 you-fill-this-part-in
eval Y_repair you-fill-this-part-in
#
# Do EXNOR of the correct Y function and your Y_repair [16 points]
eval Z !(Y^Y_repair)
```



```

#
# YOU DO THIS: what is the right quantification to solve for repair?
# [10 points]
quantify what-kind? Z_quantify Z you-fill-this-part-in-what-variables?
#
# what values of Si's make this function == 1?
satisfy Z_quantify
#
# You look at this result, and tell us how to repair network!
# boolean wire0 wire1 wire2 wire3 wire4 wire5 wire6 wire7
eval wire0 0
eval wire1 0
eval wire2 0
eval wire3 0
eval wire4 0
eval wire5 0
eval wire6 0
eval wire7 0

# Here is where you put YOUR solution, which wire to invert? [8 pts]
eval you-fill-in-which-wire 1
#

```

Grading Details [60 points total]

You will submit your completed **kbdd** script and our auto-grader will evaluate it. 50 points of the total 60 are allocated by the auto-grader, as follows:

- **[8 points]** Correct implementation of one Programmable Inverter **PRI**.
- **[8 points]** Correct implementation of the Selection Logic (3 select inputs, 8 PRI control outputs).
- **[16 points]** Correct implementation of the **Z** function.
- **[10 points]** Correct implementation of **Z_quantify** (i.e., correct quantification)
- **[8 points]** Correct assignment of **just one wireN = 1**, to tell us which wire needs an inverter to repair the network.

The remaining 10 points will be allocated in the Auxiliary Quiz, which will ask you some questions about how you formulated and solved Part 2 of this assignment. We discuss this next.

Project Grading: the Auxiliary Quiz

We will use an **Auxiliary Quiz** – which looks just like a homework problem set -- to ask you a few questions about Part1 and about Part 2. To be clear about how the points are allocated, here is the overall breakdown;

- **Part 1 Network Gate Repair.** 40 points in all. Points given by answering questions on the Auxiliary Quiz.
- **Part 2 Network Gate Repair: kbdd Script.** 50 points in all. Points given by uploading your completed script to the auto-grader, which will evaluate you script and assign points appropriately.
- **Part 2 Network Gate Repair: Analysis.** 10 points in all. Points given by answering question on the Auxiliary Quiz.
-

So, the total assignment is 100 points. 50 points come from doing the Auxiliary Quiz (40 for Part 1, 10 for Part 2). And 50 points come evaluation of your submitted **kbdd** script.

You can submit as many times as you like; but you won't be able to see the solution until after the deadline. We don't want people guessing the right answer(s) by just resubmitting lots of times. Questions on the quiz will ask you about what the proper repair is (if it exists!) for the adder in Part 1, how you formulated the adder repair, how you formulated the Inverter repair problem (e.g., "what kind of quantification did you need and why?) and ask other detailed questions about the two kbdd scripts you created and ran.