CMU Fall'01 18-760 VLSI CAD

[120 pts] Homework 2. Out Thu Sep 13, Due Thu Sep 27 '01.

1. BDD ordering [10 pts]

We saw that variable order is highly significant for something as simple as a multiplexor. How about something like a *comparator*? A simple comparator takes two 2-bit unsigned binary numbers a1a0 and b1b0 and compares their magnitude, and sets the output z=1 just if a1a0 is **less than or equal** to b1b0. Do this:

- Is there a particularly *good* variable ordering for this function? **Show** it--show the BDD. Is there a particularly *bad* variable ordering for this function? **Show** it--show the BDD.
- **Draw** a gate-level netlist using any AND, OR, NOT, EXOR gates you want, to implement the simple comparator from the previous problem. Apply Minato's ordering heuristic (and where you need to break ties or make any arbitrary ordering decicion--just **tell** us what you did and **show** the work). **Show** what variable ordering it produces.

2. ITE for Gates [10 pts]

What is the **fewest** number of calls you need to make to ITE to implement $a \oplus b$? In other words, you don't want to use ITE several times to build AND, OR, and NOT, to do *a'b + ab'* —that's too easy. You can do it **much** more simply if you think about it Draw the multiplexor-hardware picture of the ITE and label clearly what's going in and out of each ITE.

3. ITE Decomposition [10 pts]

Using what you know of Boolean algebra, the definition of ITE, and the properties of cofactors, show that the ITE decomposition below (from class) actually works:

$$\mathrm{ITE}(I,T,E) = x \bullet \mathrm{ITE}(I_x,T_x,E_x) + \bar{x} \bullet \mathrm{ITE}(I_{\bar{x}},T_{\bar{x}},E_{\bar{x}}) \quad \text{(EQ 1)}$$

4. ITE Recursion [10 pts]

Let $f(x, y) = x \bullet y$ and $g(x, y) = x \overline{\oplus} y$ (this is *exclusive-nor*). Assume we have a multi-rooted DAG for the BDDs representing these two functions (you need to draw them.) We want to EXOR these functions, and compute a new function $Q(x, y) = (f \oplus g)(x,y)$. Using the ITE operator as discussed in the notes, show how you would implement this EXOR operation. As in the slides, show how the recursive computation proceeds as ITE calls itself. At each node of the recursive call tree, tell what ITE is computing (label the nodes in your BDD in some sensible way) and show clearly when each recursive call terminates. Draw the final recursive call tree.

Draw the final result, i.e., show the final form of the multi-rooted DAG that now represents functions *f*, *g*, and *Q*.

5. Derived Operators [20 pts]

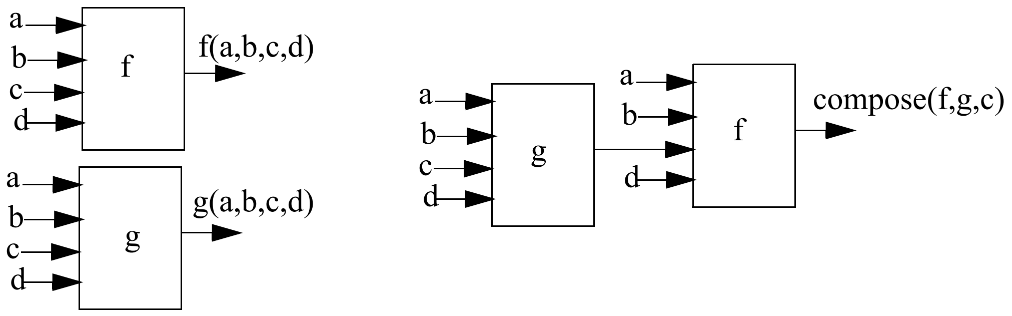
Suppose we have a software package that has data structures representing variables and Boolean functions as BDDs, and that the following operations are available as subroutines in this software. (We will write these in a simplified sort of C language notation):

bdd **var2func**(*var* x)	Generate the BDD corresponding to a single variable x. Input is a single variable (of type *var*), and the returned output is a BDD.
bdd **ITE**(*bdd* I, *bdd* T, *bdd* E)	Compute the if-then-else operation. Inputs are 3 BDDs, called I (*if* part), T (*then* part) E (*else* part), and the returned output is another BDD.
int **iszero**(*bdd* func)	Returns integer 1 just if func is the always-zero function. Input is a BDD, output is integer 1 or 0 (it's *not* a BDD)
bdd **cofactor**(*bdd* func, *var* x , *int* val)	Computes cofactor of func with respect to *var* x, setting x = val. func is a BDD, var is a variable, val is integer 0 or 1
bdd **AND**(*bdd* f, *bdd* g) *bdd* **OR**(*bdd* f, *bdd* g) *bdd* **EXOR**(*bdd* f, *bdd* g) *bdd* **NOT**(*bdd* f)	Compute the basic logical (gate type) operations on BDDs. AND, OR and EXOR create new BDDs representing the logical AND, OR and exclusive-or of their inputs. NOT creates the BDD for the complement of its input.
bdd CONST1, CONST0	You can assume these two BDDs are already defined. These are just the constant 1 function and the constant 0 function. Note that **iszero**(CONST0) == (int) 1.

No other operations are implemented, and there is no way for you to examine the BDD data structure directly.

Describe (in C-like pseudo-code notation–we *don't* need real code here!) how you would implement the following operations:

- *int* **depends**(*bdd* f, *var* x) Determine whether function f depends on the specified variable x. This means if you change x, at least sometime the output of f will also change. f is a BDD, x is a variable, **depends** returns integer 0 or 1.
- *bdd* **univquant**(*bdd* f, *var* x)
  Compute universal quantification of function f with respect to variable x. f is a BDD, x is a variable, **univquant** returns a BDD.
- *int* **opposite**(*bdd* f, *bdd* g) Determine whether two functions are complementary, i.e., if one of them is the complement of the other. f and g are BDDs, **opposite** returns integer 0 or 1.
- *bdd* **exchange**(*bdd* f, *var* a, *var* b)
  Exchange roles of specified variables in function f. For example, **exchange**(a•b + d, a, d) —> d•b + a. f is a BDD, a and b are variables, **exchange** returns a BDD.
- *bdd* **compose**(*bdd* f, *bdd* g, *var* x)
  Creates a new function (*composition* function) with variable x in f set to the output of g. The picture below clarifies what we are computing. f and g are BDDs, x is a variable, and **compose** returns a BDD.



HINTS: lots of things that look difficult are easier when you **cofactor** them and look at the cofactors. Play around with ITE of various combinations of the cofactors. The first 3 operators are pretty straightforward, the last 2 are rather tricky. *None* of these things requires some sophisticated recursive algorithm, just a few lines of calls to the right operators with the right inputs. To emphasize this, here is the answer for the first part, for **depends**(*bdd* f, *var* x):

```
int depends(bdd f, var x) {
    return(1 - iszero(  EXOR( cofactor(f, x, 0), cofactor(f, x, 1) )   );

}
```

Notice how this works. If function f( ) depends on variable x, then if I change x from x=0 to x=1, there ought to be at least some pattern for the remaining inputs that makes the output of the function *change*. But this is exactly what the Boolean Difference tries to compute. So, we compute the BDD for a new function f(... x=0 ...) ⊕ f(... x=1 ...) using calls to **cofactor** and to **EXOR**. What does this new function tell us? If the new function is zero always, for all inputs, then you cannot affect the output of function f by changing variable x. In other words, f does not depend on variable x. So if the **iszero**( ) function returns integer 1, it means the original f function does *not* depend on x. To get the true/false return for **depends**() correct, we have to invert this, which is what the (1 - ...) does.

6. Combinational Verification in KBDD [30 pts]

kbdd is a BDD calculator done by Prof. Randy Bryant's research group that has all the operators you'd want to use to manipulate Boolean functions, and a simple command line interface to type in functions, etc. You will use this to try to verify the correctness of a logic gate network whose BDDs are much too big to do by hand.

kbdd lives in /afs/ece/class/ee760/bin/kbdd, and it works on IBM AIX boxes and on SUN Solaris boxes.

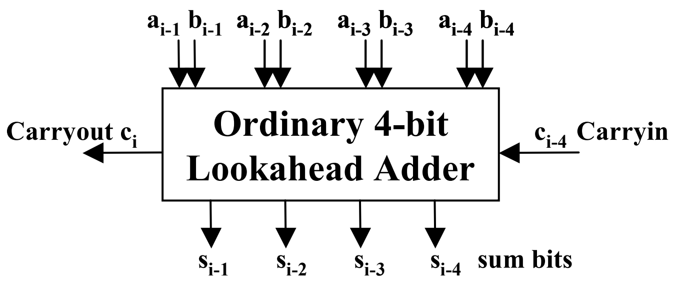
As a starting point, the following page shows a complete trace of a session with kbdd, using it to do the network repair problem on the last homework assignment. Inputs are in normal font, outputs *italics*, kbdd's prompts for input shown in bold as **KBDD**:

```
% /afs/ece/class/ee760/bin/kbdd
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
%
```

KBDD Quick Reference

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

So, what shall we use *kbdd* to verify? It turns out that *adders* are a very good choice here, because they come in so many different styles, and we know that the BDD for a basic adder is very simple and small. Let us look at a not-so-straightforward design for a 4-bit binary adder block. To start, here is a reminder of how an ordinary carry lookahead adder works


a_{i-1} b_{i-1} a_{i-2} b_{i-2} a_{i-3} b_{i-3} a_{i-4} b_{i-4}
Carryout c_i
Ordinary 4-bit Lookahead Adder
c_{i-4} Carryin
s_{i-1} s_{i-2} s_{i-3} s_{i-4} sum bits

In a conventional carry lookahead adder, we would do this:

$g_i = a_i b_i$ **per bit carry generate signal**
$p_i = a_i \oplus p_i$ **per bit carry propagate signal**

$s_i = p_i \oplus g_i$ **output sum bit**

Lookahead-style output carry:
$c_i = g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + g_{i-4}p_{i-3}p_{i-2}p_{i-1} + c_{i-4}p_{i-4}p_{i-3}p_{i-2}p_{i-1}$

You could make an 8-bit lookahead adder here by connected 2 of these 4-bit blocks, and connecting the lower-block's carry out to the carry in of the higher block.

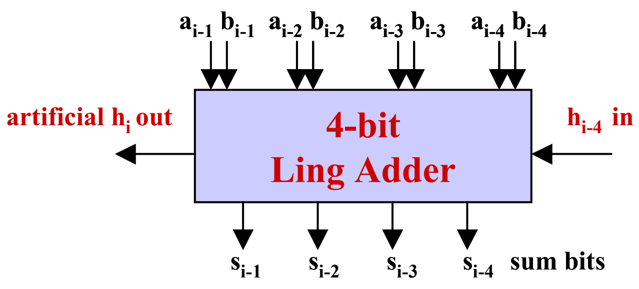
It turns out that there are many other ways of doing lookahead ideas. The adder on the next page is one of the more famous ones: the **Ling Adder**.

Question: Does this thing really work? (One does *hope* so, it's a pretty famous adder design, from Ling of IBM in 1981. But, we did copy these equations from the lit, who knows, maybe we got them wrong...?) To test this, create the equations for an **8-bit Ling adder**. This just means take 2 of the 4-bit blocks on the previous page, and connect them with the $\mathbf{h_4}$ signal rippling between them. You can assume that $\mathbf{h_0}$ = Cin for "carry in" variable.

This means, we want you to build the BDDs for all 8 of the sum bits $\mathbf{s_7}$, $\mathbf{s_6}$, ... $\mathbf{s_0}$ for the Ling adder, and compare them to an "ordinary" adder. Are they identically the same Boolean functions? Ignore the carry out of the high bit -- the Ling adder doesn't generate this explicitly so we will ignore it.

Use *kbdd* to formally verify this 8-bit design. Include a listing of your session with *kbdd* showing how you did this. Note especially that *kbdd* has basic n-bit adders built in, so to create the "baseline" sum-bit equations is a simple operation. For example, to do a basic 4-bit adder, this will suffice:

```
kbdd: boolean a[3..0] b[3..0] s[3..0] cout cin
kbdd: adder 4 cout s[3..0] a[3..0] b[3..0] cin
```


a_{i-1} b_{i-1} a_{i-2} b_{i-2} a_{i-3} b_{i-3} a_{i-4} b_{i-4}
artificial h_i out
4-bit Ling Adder
h_{i-4} in
s_{i-1} s_{i-2} s_{i-3} s_{i-4} sum bits

In a Ling Adder, instead of propagating the carry c_i from stage to stage, we propagate an "artificial" signal $h_i = c_i + c_{i-1}$
The motivation is that it's faster to compute h_i
We do it like this:

$g_i = a_i b_i$	**per bit carry generate signal**
$t_i = a_i + p_i$	**Ling-style propagate signal**

Ling-style lookahead output :

$$h_i = g_{i-1} + g_{i-2} + g_{i-3}t_{i-2} + g_{i-4}t_{i-3}t_{i-2} + h_{i-4}t_{i-4}t_{i-3}t_{i-2}$$

Intermediate "h" signals use the same pattern, but just have fewer terms. For example, since i=4 above, then:

$$h_3 = \underbrace{g_2 + g_1}_{\text{generated carry from 2 prev bits}} + \underbrace{g_0t_1t_2t_3}_{\text{propagate } g_0 \text{ in bit 0 thru bits 1 2 and 3}} + \underbrace{h_0t_1t_2t_3}_{\text{propagate } h_0 \text{ "carry in" thru bits 0 1 2 and 3}}$$

This "sort-of-a-carry" propagation is nicer – less levels of logic. However, the sum calculation becomes more complicated:

$$s_i = (t_i \oplus h_{i+1}) + h_i g_i t_{i-i}$$

All the work is in *creating* the Ling adder in *kbdd*, and then comparing it appropriately.

Note: It's probably easiest to edit a script file that you then run through kbdd using the *source kbdd* command. In UNIX, if you type the following italics stuff:

% *script*

% */afs/ece/class/ee760/bin/kbdd*

kbdd: *source myfilename*

kbdd: *quit*

<you hit control-D>

then it will (1) start saving everything in a file called *typescript*, (2) run *kbdd*, (3) tell *kbdd* to run your commands in your file *myfilename.* You type control-D to stop the script saving. You can then print this *typescript* file and hand it in, and include some comments so when we read it, we understand what you did.

7. Multi-Terminal BDDs [10 pts]

BDDs come in a number of specialized variants, one of which is the Multi-Terminal BDD, or MTBDD.

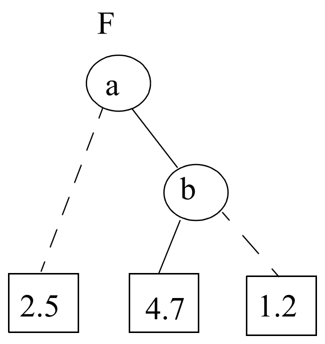
MTBDDs are a generalization of BDDs that allow an arbitrary number of real-valued terminals instead of just the basic binary terminals, 0 and 1. While a BDD represents function that returns a Boolean value for any assignment to its variables, an MTBDD returns a real number. More formally:

BDDs	F : B^n -> B
MTBDDs	F : B^n -> R

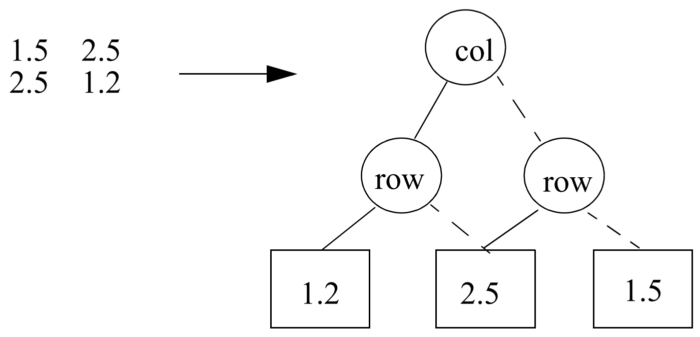
Suppose you wanted to represent the following function:

F(a,b) = 1.2 for a=1, b=0
F(a,b) = 2.5 for a=0
F(a,b) = 4.7 for a=1, b=1

These numbers could have any number of meanings. One example would be the power in mW consumed by a particular circuit when it's input changes from "a" to "b". Or it could be the delay in ns, or the rise time of the output. The MTBDD for this function would be the following:

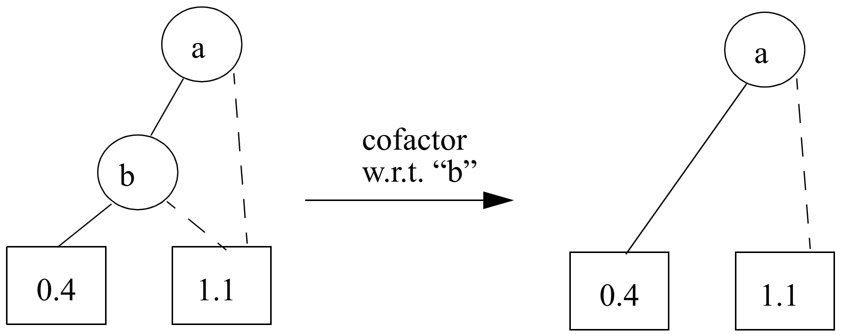


Furthermore, MTBDDs can be used to represent real-valued matrices fairly efficiently. To do so, we introduce log(# rows) + log(# columns) variables to encode the row position, and column position. For a simple 2 by 2 matrix, we get the following:



For very large matrices with lots of repeated numbers, the MTBDD representation can be considerably smaller than a simple listing of the values, and operations like matrix-addition or matrix-multiplication are correspondingly faster.

It turns out that working with MTBDDs is just as easy as with BDDs. To apply an arbitrary binary operator (function with two operands, like "a+b"), we can use the same expansion and cofactoring rules as with BDDs. The cofactor of a function F with respect to variable x, or F_x, is obtained by redrawing the MTBDD without all of the "x" nodes, and redirecting their incoming edges to the "hi" son. For example:



Using cofactors, we can decompose operations on MTBDDs in exactly the same manner as for BDDs:

$$F = x\, F_x + x' F_{x'}$$

$$F+G = x\,(F_x + G_x) + x'\,(F_{x'} + G_{x'})$$

$$F*G = x\,(F_x * G_x) + x'\,(F_{x'} * G_{x'})$$

This results in a nice recursive algorithm for performing arbitrary operations on two input MTBDDs that looks remarkably similar to algorithms for BDDs.

Do this:

1.) Draw the MTBDDs for the two following functions:

F(a,b) =	3	for a=0
	0	for a=1,b=0
	4	for a=1,b=1
G(a,b) =	0	for b=1
	4	for b=0

2.) Compute and draw the resulting MTBDDs for H=F+G and M=F*G

(**Hint**: This will be easier if you think about what the results should be for each of the assignments to a and b first, and then try drawing the MTBDD.)

8. Metaproducts [20 pts]

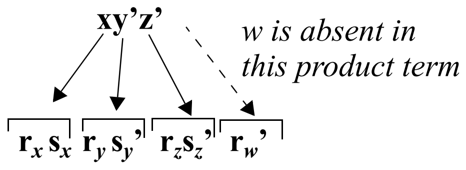
A wonderful property of BDDs is that they only represent the "abstract" Boolean function, *not* the way you chose to implement it with logic gates. This is why BDDs are so useful for verification: you can implement **x + x'** , or **x + y + x'y'**, or just plain "**1**", and in all these cases, you get the identical BDD.

Of course, sometimes we actually *want* to represent the *way* we have implemented something as logic gates. Suppose we really want to represent--for whatever odd reason--the SOP expression = **x + x'**. Is there any way we can represent this directly, without immediately resolving it to the "1" function? In other words, can we preserve the SOP product structure of this expression? The answer is "yes" -- sort of.

We need a new representation of the function that "records" the SOP structure, but which also behaves as much like a BDD as possible. It turns out there is a very elegant trick for recasting the original function in a new set of variables, and then just representing this new function as a BDD, that does much of what we want it to do. This new "SOP preserving" structure is called **metaproduct notation**. (The idea is due to Olivier Coudert, originally of Bull Research Center in France.) Here's the trick:

- For each variable **x** in your SOP form, the metaproduct formula has 2 different variables: $\mathbf{r}_x$ and $\mathbf{s}_x$. $\mathbf{r}_x$ is the *occurence* variable for **x**; $\mathbf{s}_x$ is the *sign* variable for **x**.
- Suppose your function is **f(x,y,z,w)**. For each product term in your SOP form, for example **xy'z'**, you get a corresponding metaproduct term.
  If your literal is in positive form, like **x**, you get ($\mathbf{r}_x\mathbf{s}_x$) in the metaproduct.
  If your literal is in negative form, like **x'**, you get ($\mathbf{r}_x\mathbf{s}_x$') in the metaproduct.
  If a variable is missing from the product, like **w**, you get ($\mathbf{r}_w$') in the metaproduct.
  (It turns out the $\mathbf{s}_w$ variable doesn't matter in this case, since **w** is not present, sign doesn't matter)
- So, **xy'z'** would get transformed into **(** $\mathbf{r}_x\mathbf{s}_x\mathbf{r}_y\mathbf{s}_y$'$\mathbf{r}_z\mathbf{s}_z$'$\mathbf{r}_w$' **)** in metaproduct form.

Read $\mathbf{r}_x$ **=1** as meaning "the variable **x** occurs in the product." Read $\mathbf{r}_x$ **=0** as meaning "the variable **x** does *not* occur in the product." Similarly, read $\mathbf{s}_x$=0 as "the polarity of x is positive" and $\mathbf{s}_x$=1 as "the polarity of x is negative". For example:



So, for example, if we actually tried to represent **f(x)=**(**x** + **x'**) we would get ($\mathbf{r}_x\,\mathbf{s}_x$ + $\mathbf{r}_x\mathbf{s}_x$') for the metaproduct form. To manipulate this, we represent it as a BDD. we get one more rule here:

- Interpret the **paths** from the BDD root to the "1" leaf as specifying the individual product terms in the metaproduct form. If a variable is omitted on a path, you need to include it in **both** polarities in the final metaproduct.

This sounds more complicated than it really is. Let's try it. Do this:

- Draw the BDD for the metaproduct for the single-variable function **f(x)=x+x'**. Show how walking the paths from root to "1" leaf generates the correct metaproduct for this SOP form.
- Using what you know about BDDs, *complement* this BDD for this metaproduct. Again, walk the paths from root to "1" and generate the new metaproduct for f'(x). Interpret the result -- does this makes sense? Why?
- Draw the BDD for the metaproduct of the 4 variable function:

  f(x,y,z,w) = yw' + xzw' + xy'zw'

  Again, show how the paths from root to leaf in this BDD generate the correct metaproduct. (It's OK to use kbdd for this one, as long as you can figure out the BDD structure yourself.)
- Again, complement this BDD, and show that the result makes sense as a metaproduct. (For this one, you might want to use kbdd -- if it's too messy to do by hand.)