F01 Project 2: Transistor-Level Equiv. Checking

- So far, the "logic" we have seen is all made of *gates*
  - AND, OR, EXOR, EXNOR, NOT etc etc
- But, ICs are made up from *transistors*
  - CMOS P- and N-type FETs to be precise
- How can we look at a transistor-level netlist and "extract" the Boolean logic function it implements, and then "check" this?
  - This is project 2
  - Need a good transistor-level *representation* as Boolean values
  - Need to *simplify* away some transistor-level behavior
  - Need to know new techniques for solving *systems* of Boolean eqns
  - All doable with *BDDs* (of course!)

Copyright Notice

Where Are We?

- A *very* realistic application…

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

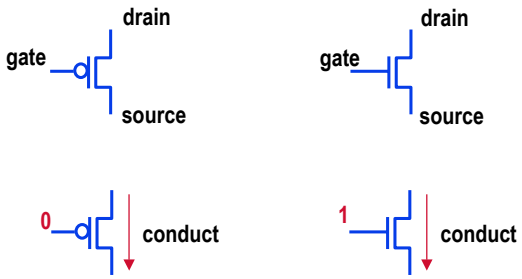
- OUT: 16 Oct 2001
- DUE: 8 Nov 2001 by 5pm
- Logistics:
  - You can work in groups of 2
  - Implementation is in C or C++ using the CUDD BDD package from U Colorado. You link to it.
- For a grade
  - Writeup is a **WEBPAGE**. You put it up, email us the URL by 5pm
  - **DEMO** required, sign-up for times at end of project, will be required to show performance on "live" (new) circuits, explain what happens as they run in real-time.

Readings/Deadlines/Projects

- De Micheli
  - Nothing about this stuff
- Deadlines
  - OK, fine, I give up: HW3 due date BACK to Oct 23, NEXT Tue, in class
    - AFTER mid-semester break
    - *(The things I do for you people…)*
- Project #2
  - Today—the overview
  - Due date: 8 November 2001

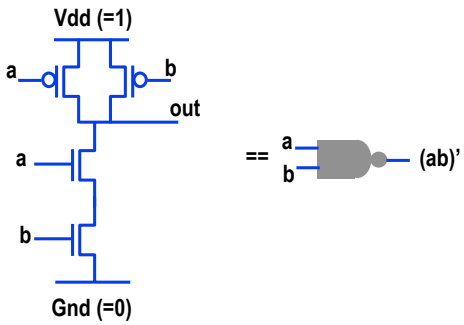
Background: CMOS Logic

- 2 kinds of transistors: P and N
  - N devices conduct when their input == 1
  - P devices conduct when their input == 0
  - Devices have 3 terminals, are bidirectional for us



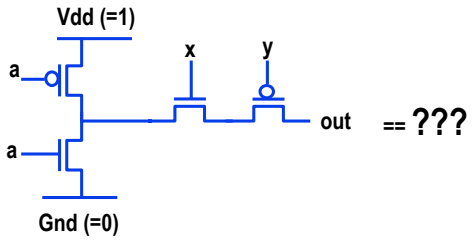
Our Problem

- How do we analyze a transistor netlist and "extract" Boolean functions for its outputs?
  - For example, how can we compute that this netlist is a simple NAND?
  - This is a pretty simple case: this is a static CMOS (series/parallel) gate



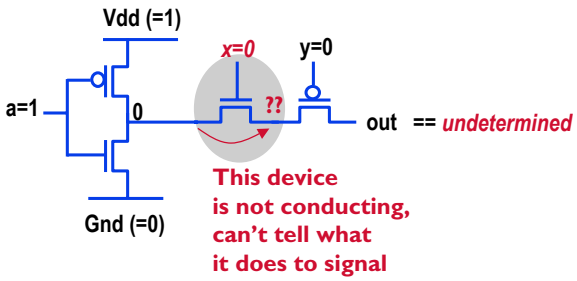
Our Problem

- And what if we allow pass transistor style circuits?
  - The 2 devices at the left are clearly a simple inverter
  - But, the inverted output passes thru 2 *pass transistors* which can "gate" the result to the output only if x==1, y==0



Our Problem

- And what happens when signals take "unreasonable" values?
  - Can allow inputs to be "unknown" in logic networks, see how these unknowns propagate
  - But, in a transistor netlist, all signals can be known, and the output may still be *undetermined*; consider example below



Solution

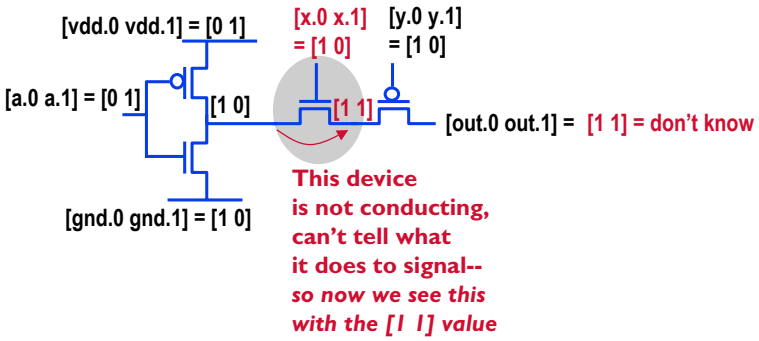
- Need a more sophisticated model of the "steady state" value on a wire in a transistor netlist
  - "Steady state" means combinational circuits only, when they stabilize
  - Need to model not just "=1" and "=0" but also "=X" don't know state
- Need to acknowledge some simplifications
  - All devices have same "strength", as do all storage nodes in circuit
  - No ratio tricks, no overriding of logic values at contended node, etc.
  - Some crazy dynamic circuits, circuits with complex state, won't work in our analysis technique
- Need more powerful solution strategy
  - With a good model for individual nodes in the MOS circuit, we can derive systems of Boolean equations. Trick is in how to solve them…

Better Model of Logic Nodes

- Use 2-bit signal encoding -- just like PCN notation
  - Each node (wire) in a netlist gets a pair of boolean variables that together represent its state
  - So, node "p" represented as [p.0 p.1] pair of values
  - Same encoding as PCN
    - [p.0 p.1] = 0 1 => it's a Boolean 1
    - [p.0 p.1] = 1 0 => it's a Boolean 0
    - [p.0 p.1] = 1 1 => it's *indeterminate* -- don't know what it is
    - [p.0 p.1] = 0 0 => not allowed to happen
- Goal
  - We want the "ordinary" Boolean values (0, 1) to work like ordinary logic gates work
  - But we want to get [1 1] when a node in the circuit simply cannot be determined from the current inputs. In other words we want to *compute* and to *propagate* these indeterminate values correctly

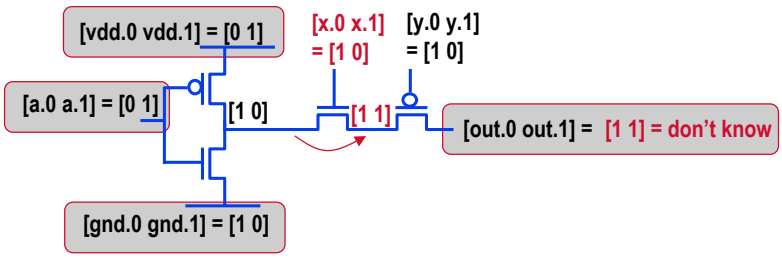
Simple Example

- This is what we expect we should be able to compute



Inputs and Outputs

- Look again at this example
  - Notice that the inputs and the outputs and even the power rails are also represented in this same notation



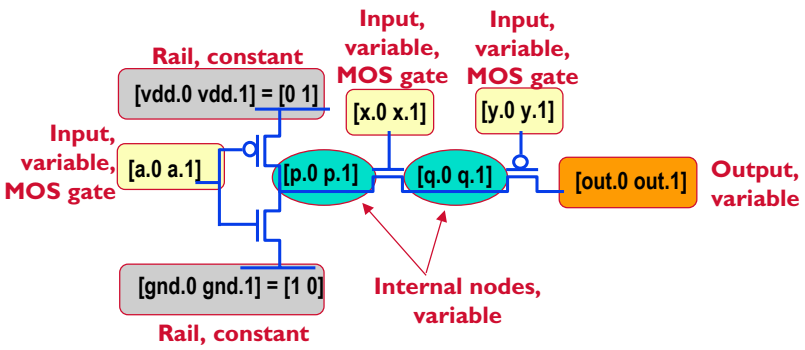
  - Need to be careful to be precise about what is a variable and what is a constant here…

Nodes in Netlist: Inputs, Outputs, Internal

- 4 kinds of nodes (wires) in these circuits, with diff constraints
  - Inputs:
    - represented as variables, can be connected to MOS gates or to MOS drain/source. You need to know which--it will *matter* later
  - Power rails:
    - they are inputs, but special inputs with constant value, they are not variables, they appear to us as constant "1" or "0"
  - Outputs:
    - represented as variables, can connect only to MOS drain/src
  - Internal:
    - everything else in the circuit. Represented as variables. These represent MOS device drains and sources

Nodes in Netlist: Inputs, Outputs, Internal

- Example revisited



What we want is to create a BDD for every internal and output node [v.0 v.1] that captures the correct behavior, ie, [v.0=(some BDD), v.1=(some other BDD)]

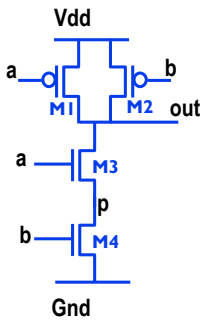
Solving for Node Equations

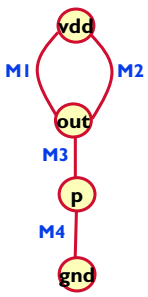
- 5 big steps

- 1. Make the diffusion channel graph for the circuit
  - Graph represents paths thru the circuit we need to model
- 2. Solve for v.1 var for each [v.0 v.1] internal & output node
  - Graph lets us create a set of simultaneous Boolean eqns; solve 'em
- 3. Solve for v.0 var for each [v.0 v.1] internal & output node
  - Similar graph, *different* variables, *same* solution process
- 4. Solve for D=indeterminate conditions for each int/out node
  - Similar graph, different vars, same solution process
- 5. Assemble final solution
  - From v.1, v.0, D at each variable node, we can get Bool eqn we need

Channel Graph

- Represents conducting paths in the CMOS circuit
  - Node = MOS transistor drains and sources. NOT the MOS gate inputs.
  - Edge = one MOS device drain-source path, ie, conducting channel

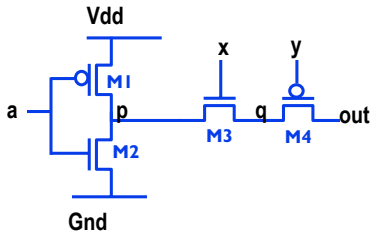


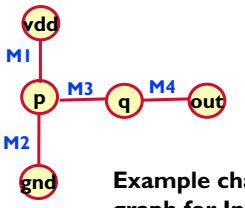


Example channel graph for simple 2 input NAND

Channel Graph

- Our other example





Example channel graph for Inverter with "odd" pass devices at its output

Using the Channel Graph

- Channel graph defines a *system* of Boolean equations
  - We need to know how to set these up, this is how we will solve for v.l, v.0 and D for each node in the circuit
  - We specify a set of "initial" values for the graph, where a "value" is a Boolean equation.
  - Each node and each edge gets an initial value
  - Each node then gets a variable, call it x[n] for node n
  - Goal is to solve for x[n] at each node so that the overall set of Boolean equations defined by the graph is "consistent", ie, makes sense, works out right under some sensible rules
- 3 big questions
  - What does such a system of Boolean equations look like?
  - What does a "consistent" Boolean solution look like?
  - Mechanically, how do we solve to find this solution?

Analogy: Systems of Linear Equations

- Analogy: matrices from linear algebra
  - We have variables, say: t y z w
  - We have a system of linear equations, for example

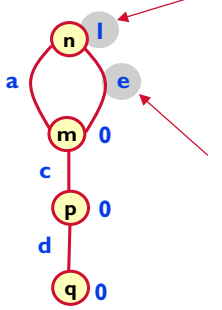
    2t + 3y + 4z + 5w = 32
    t + 7z - 3w = 10
    7t + 3y + w = 17
    2y + 3z + 8w = 45
  - A consistent solution -- in this case, t=1 y=2 z=3 w=4 -- is such that if you take any row of this matrix and substitute these values in, the equation checks out right, eg

    2t + 3y + 4z + 5w = 32
    t + 7z - 3w = 10
    7t + 3y + w = 17 => 7(1) + 3(2) + (4) = 17 …yes, OK
    2y + 3z + 8w = 45
  - A linear solver (eg, Gaussian elimination) can take this system, and having only the 4x4 matrix (call it A) and the 4x1 vector (call it b), can solve the eqn Ax = b for the solution x vector=[1 2 3 4]

Now: Systems of Boolean Equations

- Amazingly enough, the *same* problem
  - Only now, we have "AND" for "•" and "OR" for "+"

Initial setup

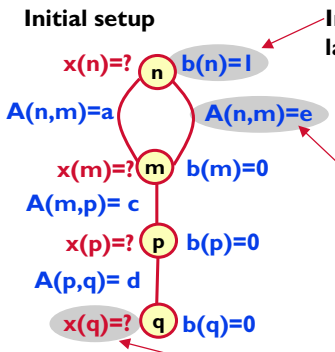


Initial node values are like the "b" constants in a linear algebra Ax=b matrix problem. But, for us, they are boolean eqns

Initial edge values are like the elements of the "A" matrix in a linear algebra Ax=b problem. But, for us, they are again boolean eqns

Now: Systems of Boolean Equations

- Relabel the graph to make this association clear



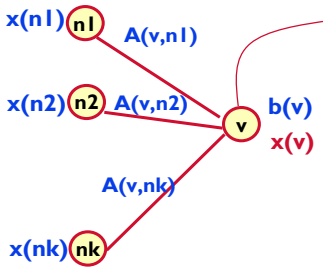
Initial node values for each node v labeled as b(v)

Initial edge values labeled as A(v,u) for each Edge between nodes u, v.
Note one weird thing: can have *multiple* edges between any pair of nodes.

And finally, each node also has a variable x(v) which is the "x" in Ax=b. We need to solve for these unknown x(v) values

Solving Systems of Boolean Equations

- How do we recognize a solution? It's "consistent", it "works"
  - Here is the rule for when a set of eqns for each x(v) "works":



Solution is *consistent* if, for each node v,

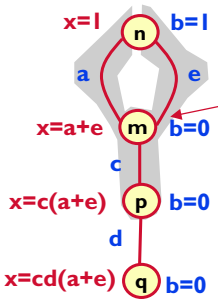
$$x(v) = b(v) + \sum_{\text{Neighbor nodes ni}} A(v,ni) \bullet x(ni)$$

In this example, this means:

$$x(v) = b(v) + A(v,n1) \bullet x(n1) + \ldots + A(v,nk) \bullet x(nk)$$

Example

- A consistent solution



$$x(m) \stackrel{?}{=} b(m) + \sum_{\substack{\text{Neighbor} \\ \text{nodes ni}}} A(m,ni)x(ni)$$

$$x(m) \stackrel{?}{=} b(m) + a\,x(n) + e\,x(n) + c\,x(p)$$

$$\begin{aligned}(a+e) &= 0 + a\,(1) + e\,(1) + c\,[\,c(a+e)\,] \\ &= a + e + ac + ec \\ &= a + e \quad (!!)\end{aligned}$$

- New questions
  - Properties of this solution?
  - And, how do we actually *find* such a solution?

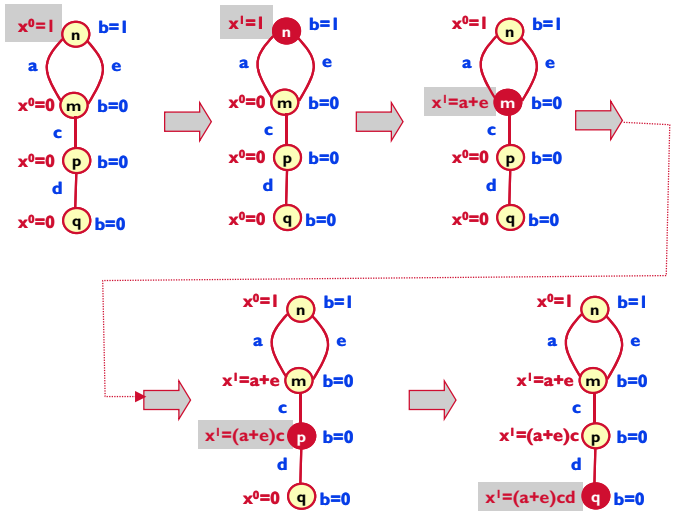
Solution Properties

- The big, useful result (Bryant 1989)
  - Any Boolean system [A b] where A is a set of edge values (boolean eqns) and b is a set of node values (boolean eqns) has a UNIQUE solution x (set of node eqns)
  - This solution is given by the limit of the sequence x^i, where
    - $x^0(v) = b(v)$ for all nodes v
    - $x^i(v) = x^{i-1}(v) + \sum_{\substack{\text{Neighbor} \\ \text{nodes n}}} A(v,n)x^{i-1}(n)$

- In English…
  - There's one unique solution to the set of equations
  - You can find it iteratively--
    - Set all x(v) to b(v) to start
    - Pick a node, update it with the above formula based on its neighbors
    - Continue until each x(v) equation stops changing

Solving Iteratively

- Showing the iterations:
  - go thru each node, do this formula on it, update x^0 value to x^1 value



This one just happens to converge really easily to this consistent solution

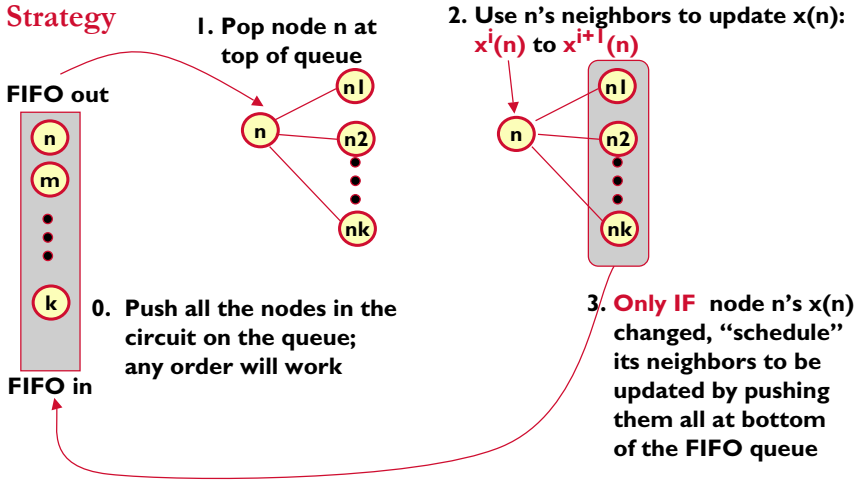
Solving Smarter

- Not smart to update the nodes in totally "random" order
- Randy Bryant in CS (who invented this) says:

> "…if you keep doing updates of the form
> v[i] <-- v[i] OR a[i,j] AND v[j]
> you'll eventually get a convergent solution. Since you're using BDDs, the convergence test becomes feasible. Basically, it then works a lot like the fixed point iterations of model checking.
>
> I think you'll find the iterative method works just fine. The main thing is to set up an "event list" that propagates an update only if the value on the source changes. This should be processed in FIFO order, so that you get the equivalent of breadth first expansion."

Solving Smarter

- Strategy



Repeat: steps 1,2,3 until the queue is empty, ie, all nodes have consistent val's

Aside: Solving VERY Smart

- Iterative is OK, not the best you can do
  - Just like with linear systems
  - Smartest you can do with (nice) linear systems: Gaussian elimination
  - Smartest you can with these Boolean systems: Gaussian elimination
- Yes--you can do Gaussian elimination on these systems
  - Check class web site, I'll post the papers from Bryant
  - More complicated, but optimally fast for larger designs
  - You don't have to do it for this project (but you can if you want to…)

OK: Where Are We?

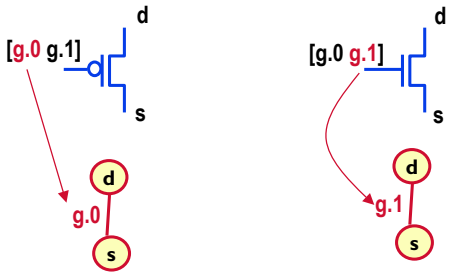
- We have a workable model for the circuit
  - Each node (internal, rail, input, output) is a 2-bit [v.0 v.1] pair
  - Model supports the indeterminate "X" I-don't-know-value state, and propagates it correctly
  - Channel graph models the circuit correctly for us
- We know how to solve systems of Boolean equations
  - Iteratively till convergence
  - Just like Ax=b, but A=edges, b=nodes, x(v) = unknowns on each node, and each of these is a Boolean equation
- What's left?
  - What system(s) of equations do we need to set up whose solution is the right answer for the behavior of these circuits?

Setting Up the Equations

- 3 sets of equations to solve
  - First 2 will seem "natural" when you see them
  - Last one is not so intuitive (at first)
- Strategy
  - Set up and solve the equations to determine v.1 at each node
  - Set up and solve the equations to determine v.0 at each node
  - Look closely and these and see why they are not sufficient

Solving for v.1 & v.0

- Rules
  - Build channel graph for the circuit
  - *Defn:* the *definite* value for a MOS gate input [g.0 g.1] is:
    - N FET: this is g.1 (ie, if g.1=on, then this N FET conducts)
    - P FET: this is g.0 (ditto--if g.0 in on, P FET conducts)
    - (Ignore the [1 1] "X" state for now; we'll come back to this)

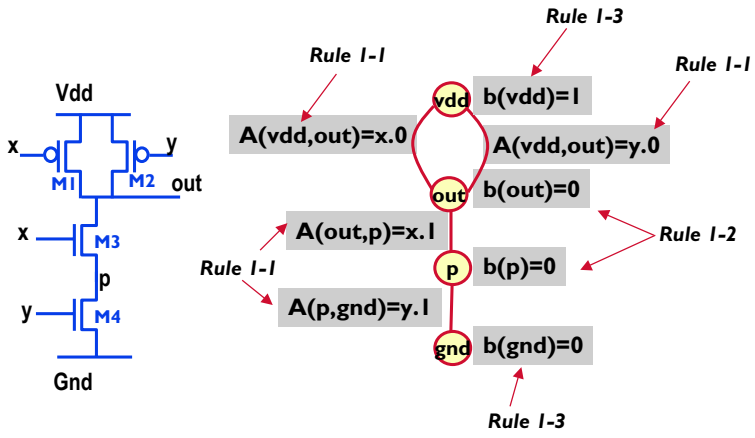


Solving for v.1 & v.0

- To setup to solve v.1:
  - Rule 1-1: edge value A(u,v) is the definite value for the FET associated with this edge in the diffusion graph
  - Rule 1-2: b(v) value for an internal node is 0. A node is *internal* if not connected to an input var, or a power rail
  - Rule 1-3: b(v) value for a rail is 1 if the rail is Vdd, 0 if rail is Gnd
  - Rule 1-4: x(v) value for a drain/source -connected input is i.1 where "i" is the name of the input variable. Do NOT resolve for this x(v), it's fixed.
- To setup to solve v.0 (*almost* identical):
  - Rule 0-1: edge value A(u,v) is the definite value for the FET associated with this edge in the diffusion graph
  - Rule 0-2: b(v) value for an internal node is 0. A node is *internal* if not connected to an input var, or a power rail
  - Rule 0-3: b(v) value for a rail is 0 if the rail is Vdd, 1 if rail is Gnd
  - Rule 0-4: x(v) value for a drain/source -connected input is i.0 where "i" is the name of the input variable. Do NOT resolve for this x(v), it's fixed.

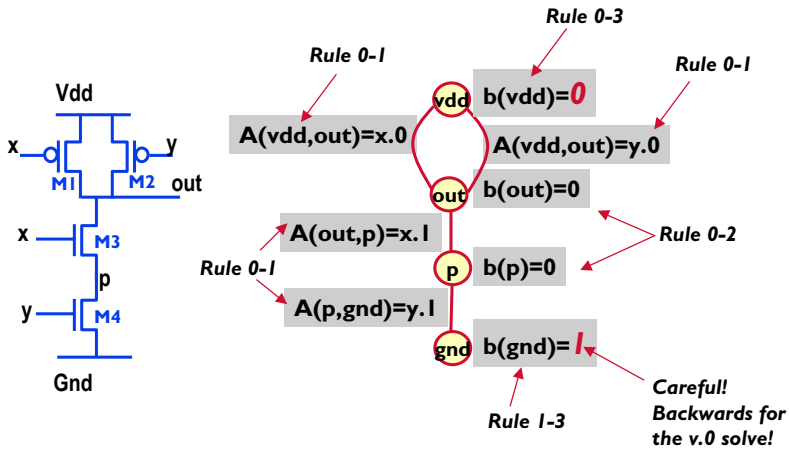
Solving for v.1

- Setup example for 2-input NAND



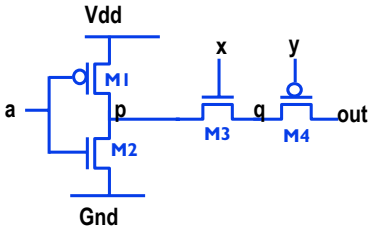
Solving for v.0

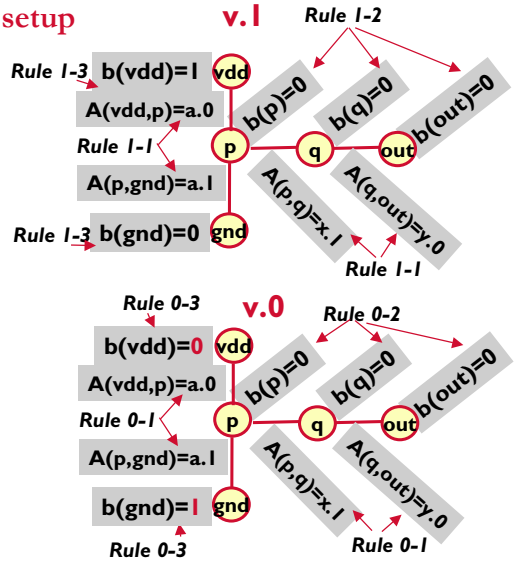
- Setup example for 2-input NAND



Solving for v.1 & v.0

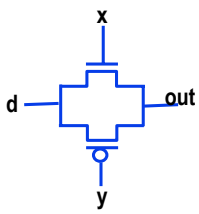
- Inverter + pass transistors setup

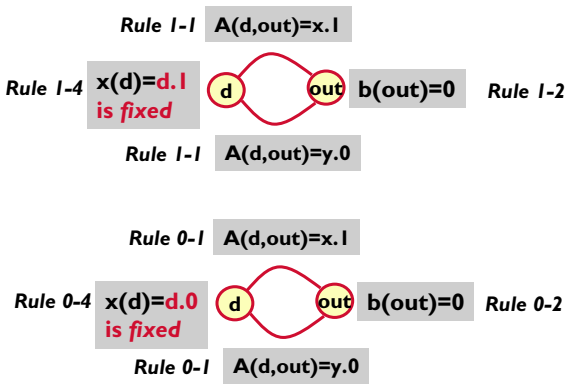




Solving for v.1 & v.0

- New example: pass transistor ckt
  - Note the different treatment of inputs and outputs
  - Inputs: x() = *fixed* value of input--we "force" node to be this input eqn
  - Outputs: b() = 0, ie, we will "solve" for this value as an x()





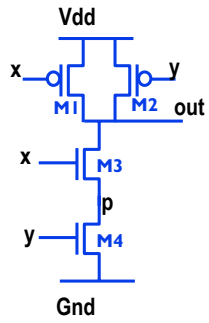
Doing the Solve (*In Detail…*)

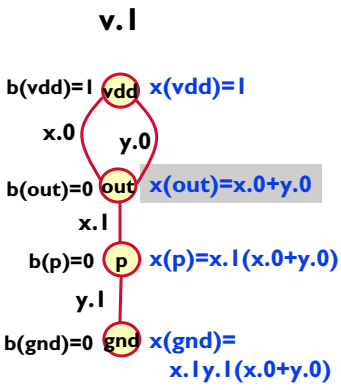
Mechanically

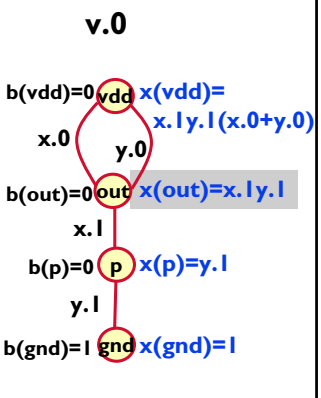
```
//Each non-fixed node v gets an unknown Boolean equation x(v)
for(each node v in graph){
   if (node v is a diffusion input )
     set x(v) to fixed BDD eqn for input node  // we won't try to solve for this one
   else {
     set x(v) == b(v) BDD;
     push node v onto FIFO queue;
   }
}
while (FIFO not empty) {
   v = pop top node on FIFO
   create x^new(v) = x(v)
   for(each node n that is a neighbor of node v)
      x^new(v) = x^new(v) + A(v,n)*x(n)
   // Compare x^new(v) and x(v) -- they're BDDs, it's easy to compare!
   if ( x^new(v) == x(v) ) {
       //Great -- don't reschedule all its neighbor nodes n for update
      let  x(v) =  x^new(v)
   } else {
       // x^new(v) != x(v), so, schedule all neighbor nodes n for update
      for( each neighbor node n of v)
         if( node v is not a fixed diffusion input)
            push n onto the FIFO queue;
      Replace x^new(v) with x(v)
   }
}
```


Solutions for Examples

NAND



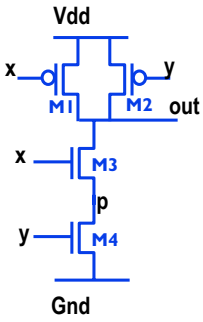




The boolean equations for the *output* nodes are what we care about

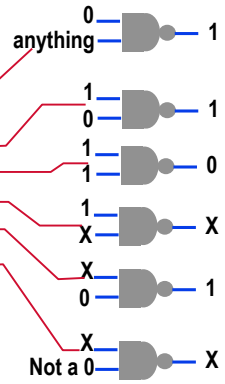
Solutions for Examples

◥ NAND



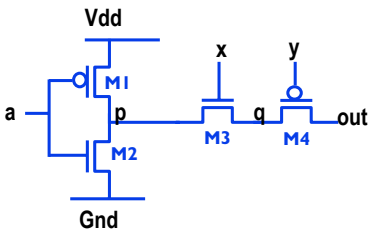
out.1 = v.1 solution for x(out)
out.0 = v.0 solution for x(out), so
out.1=x.0+y.0 out.0 = x.1y.1
Does this make sense? Yes!

x		y		out	
x.1	x.0	y.1	y.0	out.1	out.0
0	1	0	1	1	0
		1	0	1	0
		1	1	1	0
1	0	0	1	1	0
		1	0	0	1
		1	1	1	1
1	1	0	1	1	0
		1	0	1	1
		1	1	1	1

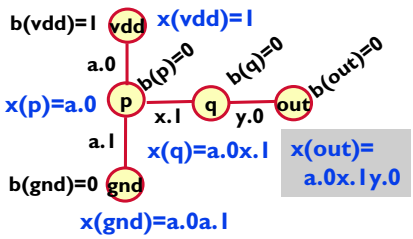


Solutions

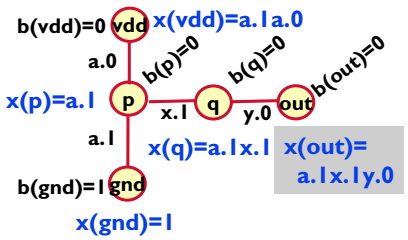
◥ Inverter + pass transistors



v.1



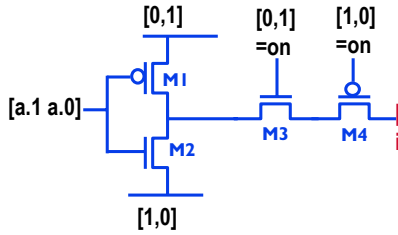
v.0



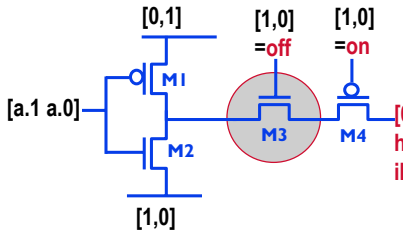
Solutions

- Again--does it make sense? No!

out.1 = v.1 solution for x(out)
out.0 = v.0 solution for x(out), so
out.1=a.0 x.1 y.0 out.0 = a.1 x.1 y.0



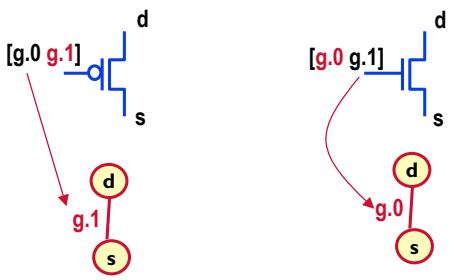
Works OK in this case, behaves as expected, it "inverts"



Surprising, *illegal* 2-bit answer. This is not supposed to happen. What did we do wrong…?

Solving for the Indefinite Case

- We need to *explicitly* solve for the case where the FETs do not make definite conducting paths--the *indefinite* case
  - *Defn:* the *indefinite* value for a MOS gate input [g.0 g1] is:
    - N FET: this is g.0 (ie, if you only know g.0=on, then can't really tell if this N FET conducts)
    - P FET: this is g.1 (ditto--if only know g.1 is on, can't tell if this PFET conducts)



Solving for the Indefinite Case

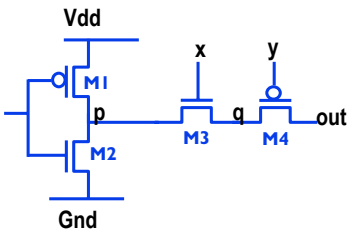
- Same solution strategy, but different system setup rules
  - Rule d-1: edge value A(u,v) is the *complement of the indefinite value* for the FET associated with this edge in the diffusion graph
  - Rule d-2: b(v) value for an internal node is 0. A node is *internal* if not connected to an input var, or a power rail
  - Rule d-3: b(v) value for *any* rail is 1
  - Rule d-4: x(v) value for *any* drain/source -connected input is fixed at 1

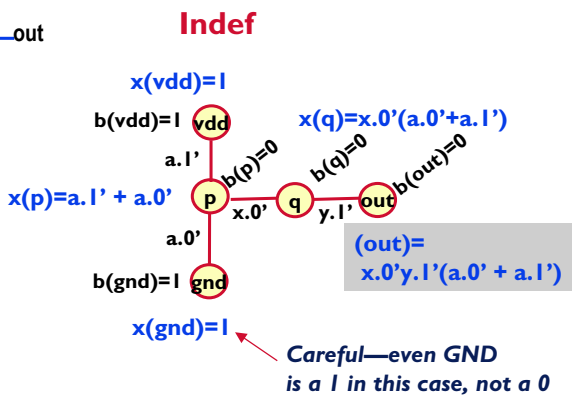
- Rules are similar, interpretation is more subtle
  - Set up and solve this system for x(v)
  - Take the resulting x(v) and *complement* each one, !x(v)
  - Update v.1 solution: out.1 = out.1 + !x(out)
  - Update v.0 solution: out.0 = out.0 + !x(out)

- Look at examples again…

Examples: Indefinite Solution

- Inverter + pass transistors





Examples: Indefinite Case

- Mechanics of final solution

  out.1 = v.1 solution for x(out) = a.0 x.1 y.0
  out.0 = v.0 solution for x(out) = a.1 x.1 y.0
  Indef = "raw" indefinite solution = x.0' y.1' (a.0' + a.1') ; we must invert
  !Indef = useful indefinite solution = x.0 + y.1 + a.1a.0

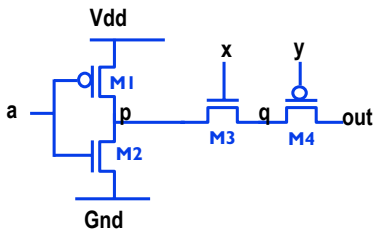
  Complete solution is thus:
  out.1 = a.0 x.1 y.0 + x.0 + y.1 + a.1a.0

  out.0 = a.1 x.1 y.0 + x.0 + y.1 + a.1a.0

- How this works
  - (!Indef) captures the cases where the paths are not defined in the MOS network, because FETs are not "for sure" conducting, OR their own gate inputs are in the "X" state
  - When (!Indef)==1, it means "can't tell if there's a path to a 1,0 here"
  - By ORing (!Indef) into *both* out.1 and out.0, we force the out value to be [1 1] in these cases, which is the "X" encoding, which is answer we want

Examples: Indefinite Case

- OK, now does it make sense? Yes!



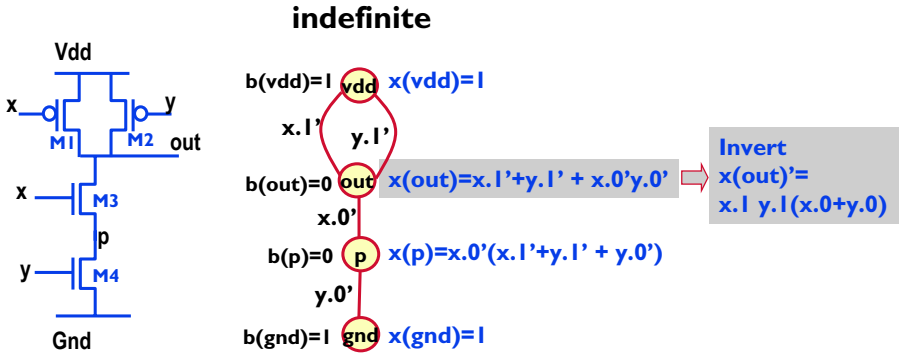
Complete solution is :
out.1 = a.0 x.1 y.0 + x.0 + y.1 + a.1a.0
out.0 = a.1 x.1 y.0 + x.0 + y.1 + a.1a.0

x		y		a		out	
x.1	x.0	y.1	y.0	a.1	a.0	out.1	out.0
-	1	-	-	-	-	1	1
-	-	1	-	-	-	1	1
-	-	-	-	1	1	1	1
1	0	0	1	0	1	1	0
				1	0	0	1

Indefinite cases for pass trans, or impossible input case [0 0]
If input = "X", then output ="X"
Expected out = !a behavior

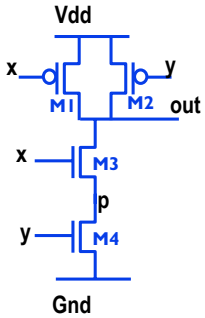
Examples: Indefinite Case

- How about the NAND?



Examples: Indefinite Case

- How about the NAND, cont?



out.1 = v.1 solution for x(out) = x.0 + y.0
out.0 = v.0 solution for x(out) = x.0 y.0
!Indef = useful indefinite solution = x.1 y.1 (x.0 + y.0)

Complete solution is thus:

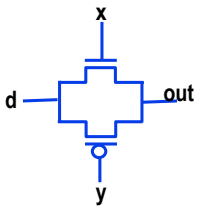
out.1 = x.0 + y.0 + x.1 y.1 (x.0 + y.0) = x.0 + y.0 (nice)

out.0 = x.1 y.1 + x.1 y.1 (x.0 + y.0) = x.1 y.1 (*very* nice)

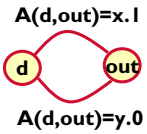
Nice result: no change to solution. There's no indefinite paths in this one, it's a nice (safe) static CMOS gate, nothing "bad" can happen here.

Examples

- How about this one?
  - …watch out for diffusion inputs

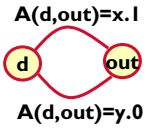


v.l x(d)=d.l is *fixed*



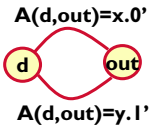
b(out)=0
x(out)=d.l(x.l+y.0)

v.0 x(d)=d.0 is *fixed*



b(out)=0
x(out)=d.0(x.l+y.0)

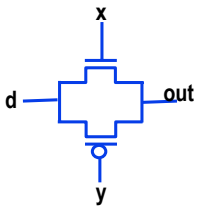
indef x(d)=l is *fixed*



b(out)=0
x(out)=x.0' + y.0'
Invert: x.0 y.l

Examples

- How about this one?
  - Works OK, even makes sense--but do watch out for diffusion inputs



out.l = v.l solution for x(out) = d.l(x.l + y.0)
out.0 = v.0 solution for x(out) = d.0(x.l + y.0)
!Indef = useful indefinite solution = x.0 y.l

Complete solution is thus:
out.l = d.l(x.l + y.0) + x.0 y.l
out.0 = d.0(x.l + y.0) + x.0 y.l

"d" input goes to output if EITHER x or y pass tran conducts

But if BOTH x, y pass transistors are OFF, then we get the [l l]="X" state at the output

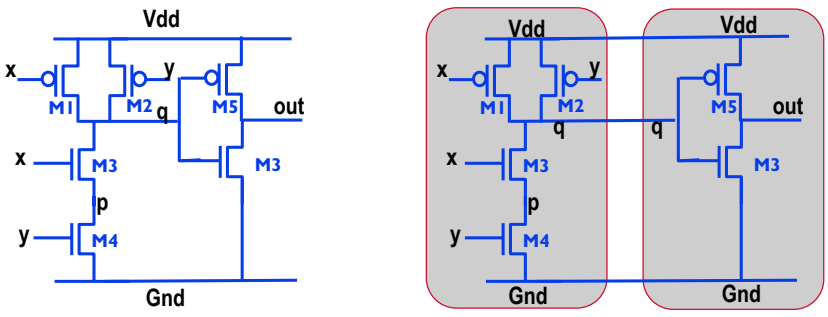
So, Where Are We?

- To analyze a transistor level netlist
  - Read in netlist
  - Build channel graph for it
  - Set and solve, in order: v.1 system, v.0 system, indef system
  - Construct solution BDD for each output node
    - Out.1 = v.1 solution + !(indef solution)
    - Out.0 = v.0 solution + !(indef solution)
  - Result is a pair of BDDs [Out.0 Out.1] that correctly describe behavior of the output node, including "X" behavior and "indefinite path" behav

- What's missing here…?
  - One little thing…

Multiple, Disconnected Channel Graphs

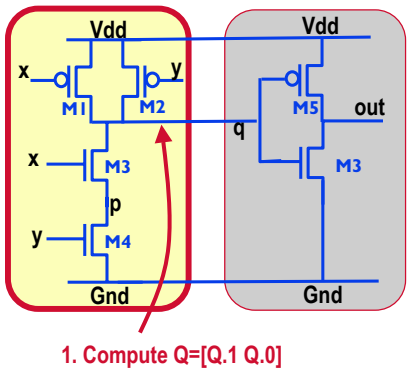
- Real netlists have many distinct channel-connected regions
  - You have to analyze *each one separately* using these solver techniques
  - Then, you need to "glue" the final solution together, in the right order



2 channel graphs in here

Multiple, Disconnected Channel Graphs

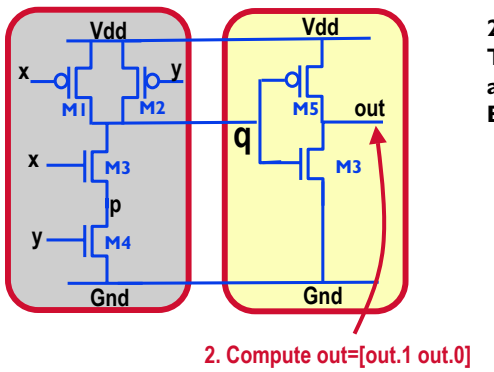
- Simple strategy
  - You have to analyze each one separately using these solver techniques



Analyze the NAND first.
Build [Q.0 Q.1] solution for its output node.

Multiple, Disconnected Channel Graphs

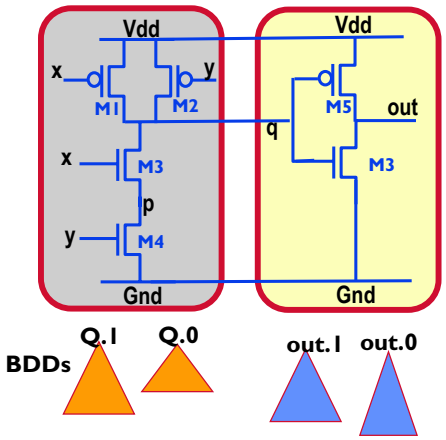
- Simple strategy
  - You have to analyze each one separately using these solver techniques



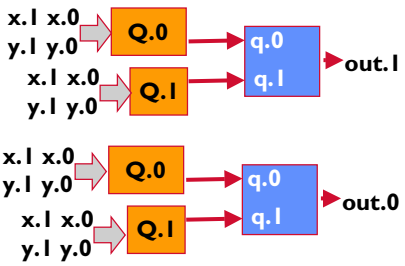
2. Analyze INVERTER second.
Treat input q as just another atomic variable.
Build BDDs for [out.1 out.0]

Multiple, Disconnected Channel Graphs

- Simple strategy
  - Then, you need to "glue" the final solution together, in the right order



Replace input var "q" in the INVERTER result with BDDs for Q using composition



Mechanics

- You need to read in the netlist
  - There's a simple format for transistor netlists
- You DO NOT need to identify each individual channel graph
  - *We will provide you with labeling to ID the channel connected parts*
  - *But, you need to build a new graph which lets you determine the right order in which to glue together the results for individual channel graphs*
- You need to analyze the function of each channel graph
  - Determine the inputs/outputs. Inputs may be "temp" variables.
  - Set up and solve the v.1 v.0 & indef systems for each graph
- Glue the final answers together
  - Requires you to visit the channel graphs in the right order, and to substitute variables in the right order

Format: Basic Transistor Netlist

```
NUMMODS <number_of_transistors>
NUMNETS <number_of_nets>
NUMINPUTPADS <number_of_inputs>
NUMOUTPUTPADS <number_of_outputs>

VDD <VDD_net_num>
GND <GND_net_num>

INPUT <input_1_net_num>
INPUT <input_2_net_num>
...... for all circuit inputs

OUTPUT <output_1_net_num>
... for all circuit outputs

P1  <channel graph ID> <source> <gate> <drain>
P2  <channel graph ID> <source> <gate> <drain>
N1  <channel graph ID> <source> <gate> <drain>
.... for all transistors

END

---------------

Some comments on the net numbers
-> If we convert from a gate-level netlist,
(1) ..(VDD-1) are gate-level node numbers
(GND+1) .. (NUMNETS) are the new nodes introduced
```


Example: Basic Transistor Netlist

```
--- Example File c17.TRAN -----

NUMMODS 24
NUMNETS 19
NUMINPUTPADS 5
NUMOUTPUTPADS 2

VDD 12
GND 13
INPUT 1
INPUT 2
INPUT 3
INPUT 4
INPUT 5
OUTPUT 6
OUTPUT 7
P1 1 12 1 8
N1 1 8 1 14
P2 1 12 3 8
N2 1 14 3 13
P3 2 12 3 9
N3 2 9 3 15
P4 2 12 4 9
N4 2 15 4 13
P5 3 12 2 10
N5 3 10 2 16
P6 3 12 9 10
N6 3 16 9 13
```

```
P7 4 12 9 11
N7 4 11 9 17
P8 4 12 5 11
N8 4 17 5 13
P9 5 12 8 6
N9 5 6 8 18
P10 5 12 10 6
N10 5 18 10 13
P11 6 12 10 7
N11 6 7 10 19
P12 6 12 11 7
N12 6 19 11 13
END
```

Format: Basic Gate-Level Netlist

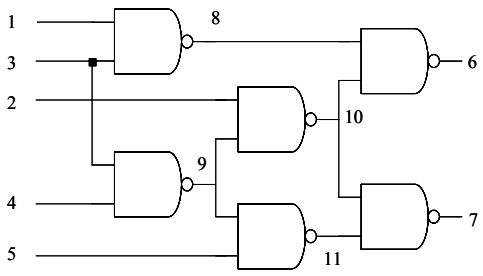
- Why do we need this?
  - So you can compare not just transistor netlists, but a transistor netlist against the gate-level logic netlist its supposed to be implementing…

```
----------- File Format for .GATE file ------

NUMMODS <number_of_gates>
NUMNETS <number_of_nets>
NUMINPUTPADS <number_of_inputs>
NUMOUTPUTPADS <number_of_outputs>

INPUT <input_1_net_num>
INPUT <input_2_net_num>
..... for all circuit inputs

OUTPUT <output_1_net_num>
.. for all circuit outputs

GATE_TYPE <num_inputs> <input1> <input2> ... <output_node>
... for all gates

END
```


Example: Basic Gate-Level Netlist

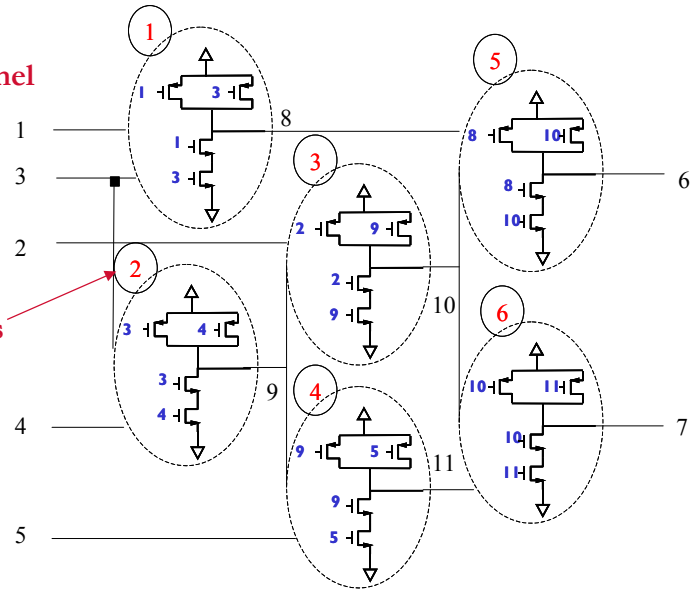
- This is the same circuit (c17) but at gate level
  - We guarantee that the input and output var numbers are same
  - Also, any "nodes" in the gate level netlist that still exist in the transistor level netlist, will also be given same numbers

```
--------- Example c17.GATE ---------

NUMMODS 6
NUMNETS 11
NUMINPUTPADS 5
NUMOUTPUTPADS 2

INPUT 1
INPUT 2
INPUT 3
INPUT 4
INPUT 5
OUTPUT 6
OUTPUT 7

NAND 2 1 3 8
NAND 2 3 4 9
NAND 2 2 9 10
NAND 2 9 5 11
NAND 2 8 10 6
NAND 2 10 11 7

END
```



Example: Gate vs CMOS Circuit for c17

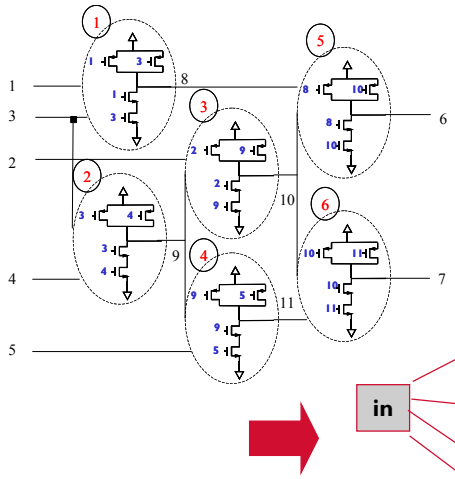
- Note it has 6 different channel graphs to deal with
- Each graph is numbered here; this is the first num on each FET's input line



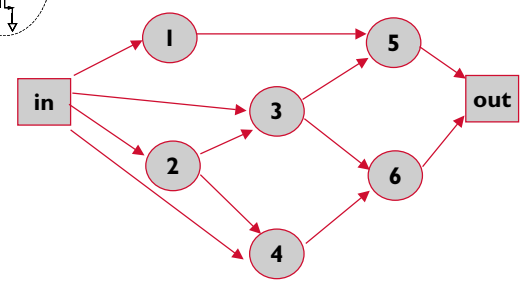
How to Deal with Different Channel Graphs

- We will ID which graph each transistor belongs to, in input file
  - You need to make a 2nd graph, to tell you in which *order* to glue together the results of the boolean analysis of each channel graph
  - Call this graph the "signal propagation" graph
- Building the sig-prop graph
  - One distinguished vertix called "input"
  - One distinguished vertex called "output"
  - One vertex for each channel graph (labled in input deck)
  - Directed edge *from* "input" node *to* any channel graph node that connects to an external input
  - Directed edge *from* any channel graph node *to* the output node for every channel graph that connects to an external output
  - Direct edge from one channel graph node N to another channel graph node M if a MOS diffusion output from N connects to a MOS gate input in M

Building the Sig-Prop Graph



Sig-prop graph



Using the Sig-Prop Graph

- What do we do with it?
  - We use it to determine the right order in which to connect the Boolean equations we have for node outputs to node inputs between diffusion graphs
- But--wait, isn't the right order for this example 1-2-3-4-5?
  - Yes, but this one just got numbered in the right order
  - In general, you CANNOT count on the channel graph ID being the same as the proper order
- How do we order the nodes in the graph properly?
  - *Topological sorting*
  - A nice, simple depth-first search algorithm on the sig-prop graph

Topological Sorting

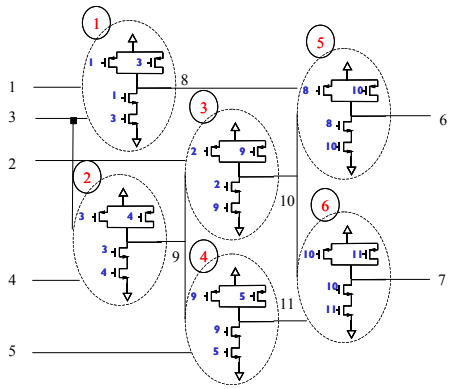
- Basic algorithm setup
  - Every node in sig-prop graph has a flag, initialized = "*clear*" (untouched)
  - We also need a global stack to store the nodes, call it S

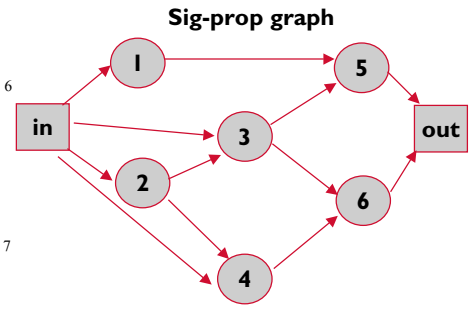
- Recursive algorithm is a variant of depth first search

```
topsort( sig-prop graph node N) {
    mark node N as "first time we have seen it"
    for( each node v that is adjacent to node N) {
        if ( flag(v)==clear )
            topsort( v )
    mark node N as "touched" (ie, we are done with it)
    push node N on global stack S
}
```

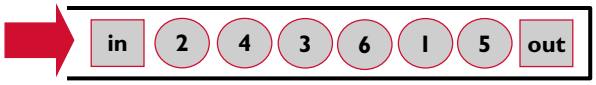
- To sort the sig-prop graph, just run topsort(in-node)
  - Result is nodes in right order, when you POP them off stack S

Topological Sorting Example





Run topsort(in), stack POPS nodes in this left-right order, assuming topsort visits neighbors top-to-bottom in our picture



Stack S

Putting It All Together

- Overall algorithm

```
Read input transistor netlist
build sig-prop graph
topsort( in node )
while ( global stack S not empty ) {
    G = POP stack S
    if ( G is  unique input or output node in sig-prop)
       continue
    allocate (possible temporary) vars for BDDs for G's inputs
    compute x.0, x.1, x.indef for each vertex in channel graph G
    compute final x.0 x.1 solution for each output in channel graph G
    for (each MOS gate input "a" in graph G connected to the
          output of some other MOS device in the netlist) {
       substitute the BDD (equation) already  computed for "a" from
        a previous channel graph into the BDDs for each output of G
    }
}
```

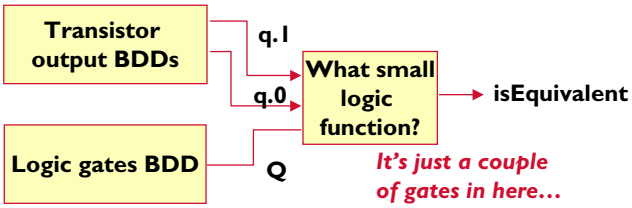
  - At this point, you have a BDD for out.1 out.0 for every "real" output of this transistor level netlist

So, What Do You Actually Do?

- Build a program that…
  - Reads in either:
    - (1) 2 transistor netlists or
    - (2) a transistor netlist and a gate netlist
  - Build the BDD behavior representation of each netlist you read in. Easy for the gate netlist. A lot more work for the transistors.
  - Output whether the netlists are the same or not, logically
  - If not, output something "illuminating", like some counter example input values
- One final technical trick…
  - How do we compare transistor and gate-level netlists?

Comparing Logic to CMOS

- Our transistor analysis supports "X", but we don't expect you to do this for the gate-level netlists
  - So, each CMOS node will have [q.0 q.1] BDDs built for it
  - But, the SAME node in the gate-level description just gets BDD Q
  - How to compare
- Simple answer: ignore the "X" stuff
  - Only insist that the "real" 1s and 0s outputs always agree
  - *Ignore* the other possibilities, ie, [q.0 q.1] = [0 0] or [1 1]



For Credit

- Logistics
  - You can work in groups of 2 or alone. STRONGLY suggest 2 people
- Code
  - C++ on SUN Solaris or on IBM AIX
  - You get to use a "real" BDD package, CUDD from U Colorado Boulder.
  - See class web page for more info/examples on how to use CUDD
- Checking
  - We will provide a CHECKER program that can tell you if your code is doing the *right* thing.
  - You will have to dump program output in a specified form for the CHECKER to check.
  - We will make an executable of CHECKER publicly available

For Credit

- Writeup
  - Not paper. *Web page.* You submit it to us via email of the URL.
  - PLEASE make it portable: we copy the whole directory structure to our machines to grade it. If you put absolute pathnames, links, it messes up
  - Suggestion
    - Make a directory: <yourname>760Web, eg, *bubba760Web*
    - Inside it, put all your html web pages: foo*.html
    - Inside it, also make 2 directories: 760Stuff and 760Code
    - Inside 760Stuff, put ALL your graphics and pics and sounds and explanatory video clips, etc. Inside 760Code, put all your code.
    - Use only relative link names for internals: ./760Stuff/foo.gif etc
    - If its on the machine in your dorm room, and it will disappear at random times--TELL US WHEN.
    - If we don't see a web page, you don't get a grade…
  - As in all things in 760 (and in life): style counts

For Credit

- About Writeup--basic pieces
  - Introduction: summarize the problem
  - Formulation: you had to make some assumptions, since there are some degrees of freedom in this project. Explain them. Justify them.
  - Optimization goals: tell us what you tried to do *well*.
  - Implementation: describe any interesting data structures, algorithms, optimizations, tricks, etc
  - Results: what did you run, how well did you do?
    - Explain your results: why did they happen like this
  - Post mortem: given you could do it over, what would you do different?
  - Code: put it someplace in the web page (preferably in 760Code dir)

For Credit

- You have to *demo*, too
  - Last week of project on a couple days--signup sheets
  - We will release some new benchmarks during the demo, and ask you to run 3 of them. They will be small; available in a couple of flavors.
  - You should print something *enlightening*
  - You run the CHECKER, we look over your shoulder and see what it says
  - Goal: *it works, it gives an OK answer.*

Points = [120] (But Weighted Big Overall)

- Breakdown
  - [30 pts] Web Writeup: Approach & Implementation
  - [30 pts] Web Writeup: Results & Analysis
  - [10 pts] Code: Reasonableness
  - [30 pts] Demo: Works, Quality, Style, Discussion
  - [20 pts] Coolness
    - Results (you created some bigger benchmarks, you ran them better, your code was faster, your webpage was slicker, etc)
    - You actually implemented Bryant's Gaussian Elimination algorithm, rather than the simpler iterative technique from class
    - Interesting algorithms (more sophisticated attacks)
    - Interesting implementation (eg, did it in PERL, but its *not* slow…)
    - You have graphical output of the solver process
    - Etc etc

Benchmarks

- Will be in /afs/ece/class/ee760/proj2/benchmarks
- 5 kinds of test cases
  - Level 0: sanity checks with only one diffusion graph in them. Simple things like one inverter, 2-input NAND, etc., labeled clearly, for your debugging
  - Level 1: sanity checks with multiple diffusion graphs in them. Simple things like N inverters in a chain, small trees of NAND gates, etc., labeled clearly, for your debugging
  - Level 2: small transistor-level netlists. *You tell us:* what are they? as logical functions.
  - Level 3: pairs of transistor-level netlists. *You tell us:* equivalent or not? If not, give us one counter-example of input values
  - Level 4: pairs of netlists, one transistor, one gate-level. You tell us: equivalent or not? If not, give counter-example input values
- Size
  - Num of transistors or gates: from 2 up to a few thousand.