# (*Lec 9*) Multi-Level Min III: Role of Don't Cares

## What you know

- **2-level minimization a la ESPRESSO**
- **Multi-level minimization:**
  - **Boolean network model,**
  - **Algebraic model for factoring**
  - **Rectangle covering for extraction**

## What you don't know

- **Don't cares in a multi-level network are very different**
- **They arise naturally as part of the structure of the network model**
- **They can help a great deal in simplifying the network**
- **They can be very hard to get, algorithmically**



# Copyright Notice

# Where Are We?

- In logic synthesis--how *don't cares* are now *very* different beasts

| | 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 |

Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
***Multi-level logic synthesis***
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps



# Readings/Deadlines/Projects

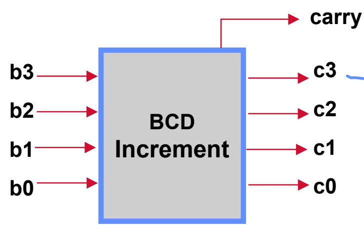
- De Micheli
  - **Section 8.4 is about don't cares in multilevel model**

- Deadlines
  - **Today, Thu Oct 11: Paper 1 Review, Rudell's Dynamic Ordering due**
  - **Thursday Oct 18: HW3 (2-level, multi-level synthesis) due**
    - **As always, check webpage for bugfixes, updates…**
    - **There are some bugs in eqns for Prob #1, fixed shortly… The state diagrams are correct as is.**
- Project #2
  - **We'll do the overview next Tuesday**

# Don't Cares: 2-level

## In basic digital design...

- We told you these were just input patterns that could never happen
- This allowed you to do more simplifications, since you could add a 1 or 0 to the Kmap for that input depending on what was easier to simplify
- Standard example: BCD incrementer circuit




| b1b0 \ b3b2 | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 1 | d |
| 01 | | | | d |
| 11 | | 1 | d | d |
| 10 | | | d | d |

Patterns b3 b2 b1 b0 = 1010, 1011, 1100, 1101, 1110, 1111 cannot happen



# Don't Cares: Multi-level

## To say this differently

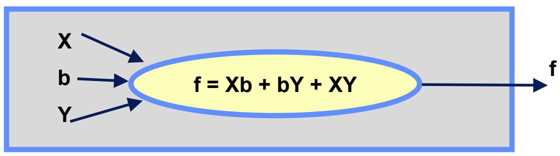
- In basic 2-level designs somebody told you what inputs wouldn't happen...
- *...and you just believed them!*

## What's different in multi-level?

- There can still be these sorts of don't cares at the primary inputs of the Boolean logic network....
- ...but there can also be don't cares arising from structure of the network
- These latter kind are very useful for simplifying the individual vertices in the Boolean logic network (ie you call ESPRESSO which can handle 2-level don't cares)
- But, *you have to go find these don't cares explicitly*

## Informal Tour of DCs in Multilevel Networks

- Suppose we have a Boolean network…
  - And we are looking at node "f" in that network
- Can we say anything about don't cares for node f?
  - NO
  - We don't know any "context" for surrounding parts of network
  - As far as we can tell, all patterns of inputs (X,b,Y) are possible

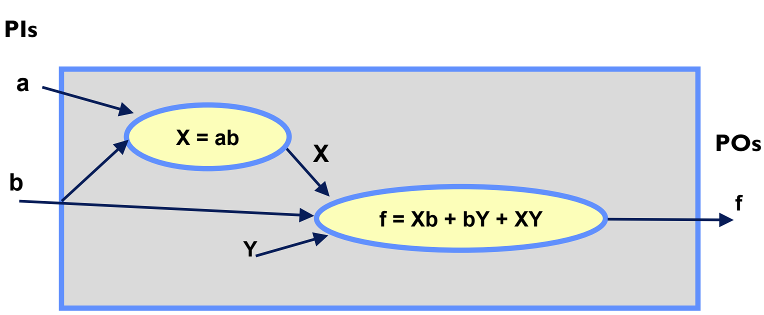



| Y \ Xb | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |



## Informal Multilevel DC Tour

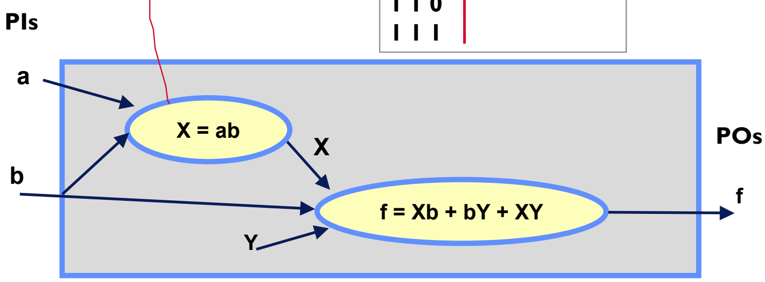
- OK, suppose we know this about input X to f
  - Node X is actually a•b
  - Now can we say something about DCs for node f…?
  - YES



# Informal Multilevel DC Tour

- Go list all the input/output patterns for node X

| a b X | can it occur? |
|---|---|
| 0 0 0 | |
| 0 0 1 | |
| 0 1 0 | |
| 0 1 1 | |
| 1 0 0 | |
| 1 0 1 | |
| 1 1 0 | |
| 1 1 1 | |



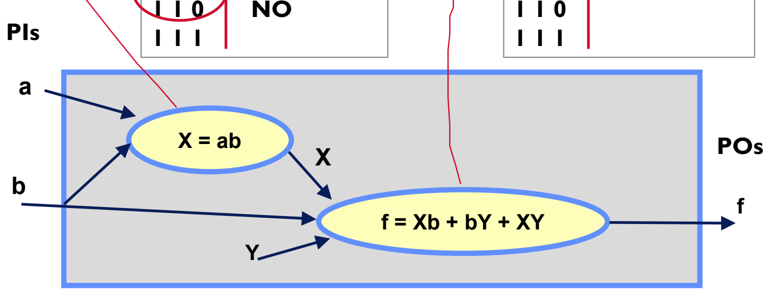



# Informal Multilevel DC Tour

- Impossible a b X patterns => impossible X b Y patterns?

| a b X | can it occur? |
|---|---|
| 0 0 0 | |
| 0 0 1 | NO |
| 0 1 0 | |
| 0 1 1 | NO |
| 1 0 0 | |
| 1 0 1 | NO |
| 1 1 0 | NO |
| 1 1 1 | |

| X bY | can it occur? |
|---|---|
| 0 0 0 | |
| 0 0 1 | |
| 0 1 0 | |
| 0 1 1 | |
| 1 0 0 | |
| 1 0 1 | |
| 1 1 0 | |
| 1 1 1 | |



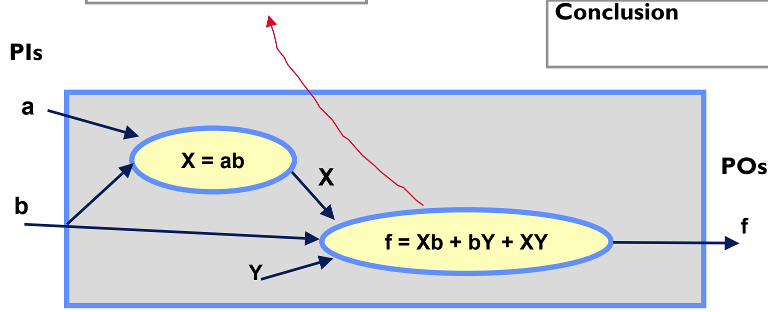
## Informal Multilevel DC Tour

- Impossible X b Y patterns give us DCs for node f
  - Change how we would want to simplify node f (it's Kmap)

Impossible patterns =

| Y \ Xb | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | 1 | |
| 1 | | 1 | 1 | 1 |

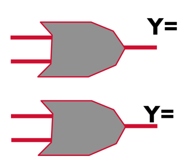
Conclusion



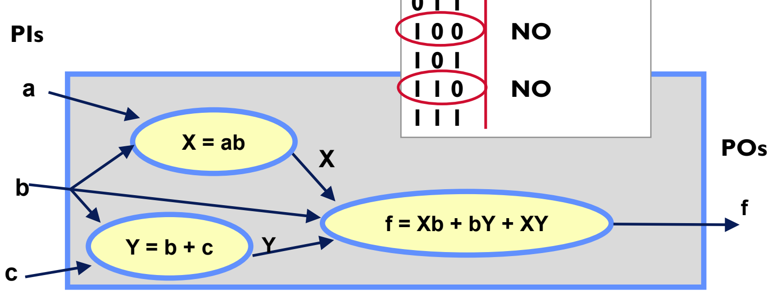



## Informal Multilevel DC Tour

- OK, what if we now know Y = b+c as well
  - Can do this again at Y ...are there impossible patterns of b c Y ?




| b c Y | can it occur? |
|---|---|
| 0 0 0 | |
| 0 0 1 | NO |
| 0 1 0 | NO |
| 0 1 1 | |
| 1 0 0 | NO |
| 1 0 1 | |
| 1 1 0 | NO |
| 1 1 1 | |



## Informal Multilevel DC Tour

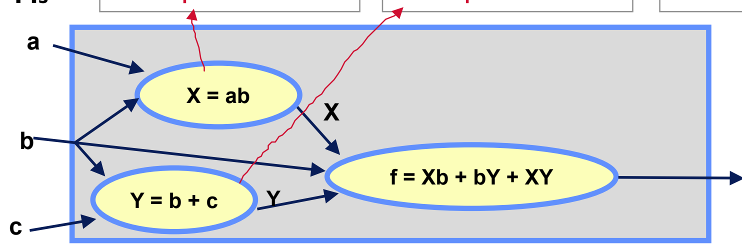
- OK, can we (again) get impossible patterns on X b Y?

| a b X | can it occur? |
|---|---|
| 0 0 0 | |
| 0 0 1 | NO |
| 0 1 0 | |
| 0 1 1 | NO |
| 1 0 0 | |
| 1 0 1 | NO |
| 1 1 0 | NO |
| 1 1 1 | |

| b c Y | can it occur? |
|---|---|
| 0 0 0 | |
| 0 0 1 | NO |
| 0 1 0 | NO |
| 0 1 1 | |
| 1 0 0 | NO |
| 1 0 1 | |
| 1 1 0 | NO |
| 1 1 1 | |

| X bY | can it occur? |
|---|---|
| 0 0 0 | |
| 0 0 1 | |
| 0 1 0 | |
| 0 1 1 | |
| 1 0 0 | |
| 1 0 1 | |
| 1 1 0 | |
| 1 1 1 | |

PIs






## Informal Multilevel DC Tour

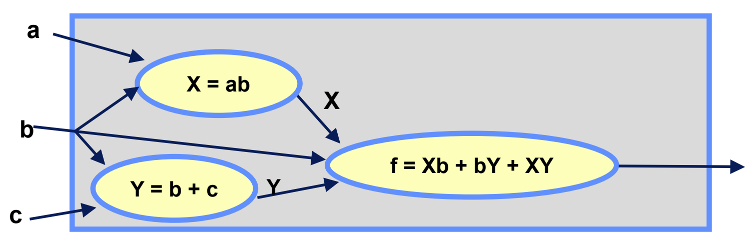
- OK, do these change how we'd simplify inside f?

| X bY | can it occur? |
|---|---|
| 0 0 0 | |
| 0 0 1 | |
| 0 1 0 | NO |
| 0 1 1 | |
| 1 0 0 | NO |
| 1 0 1 | NO |
| 1 1 0 | NO |
| 1 1 1 | |

| Y \ Xb | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | 1 | |
| 1 | | 1 | 1 | 1 |

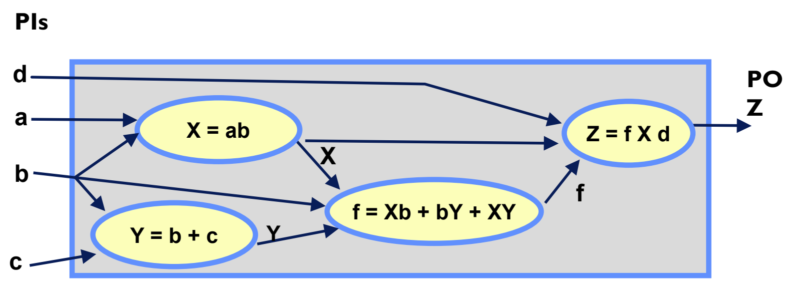
Conclusion

PIs



## Informal Multilevel DC Tour

- OK, now suppose f is not a primary output, Z is…
  - Question: when does a change in the output of node f actually propagate through to change the primary output Z, ie, the output of the overall Boolean logic network
  - Or, reverse question: when does it *not matter* what f is…?
  - Let's go look at patterns of f **X** d at node Z…





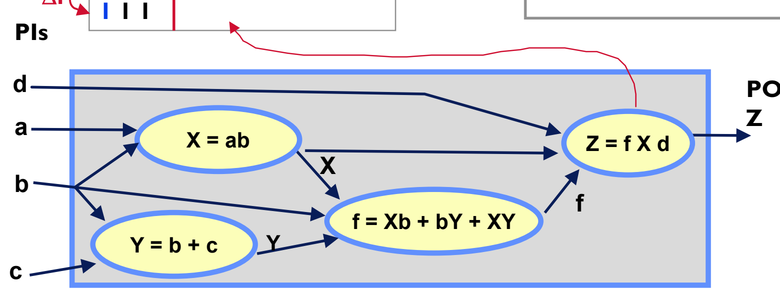

## Informal Multilevel DC Tour

| | f X d | Z does it change? |
|---|---|---|
| Δf | 0 0 0 | |
| | 1 0 0 | |
| Δf | 0 0 1 | |
| | 1 0 1 | |
| Δf | 0 1 0 | |
| | 1 1 0 | |
| Δf | 0 1 1 | |
| | 1 1 1 | |

Patterns at input to f node itself that are DCs just because those patterns make Z output *insensitive* to changes in f

**X b Y**

=



## Informal Multilevel DC Tour

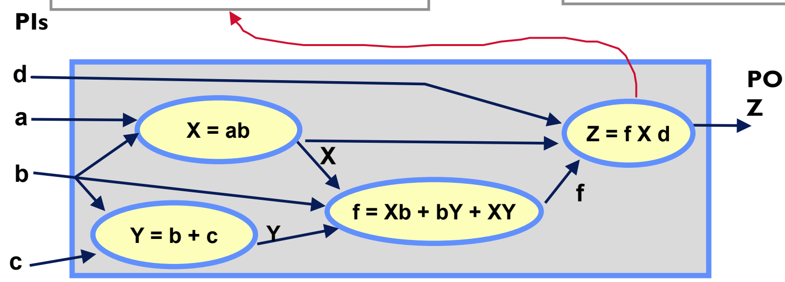
- OK, can we use this X=0 DC pattern to simplify f more?

Patterns at input to f node itself that are DCs just because those patterns make Z output *insensitive* to changes in f

=>

| Y \ Xb | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | d | d | d |
| 1 | | 1 | 1 | d |

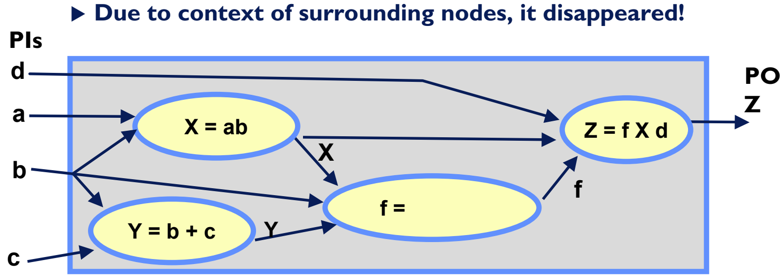
Conclusion

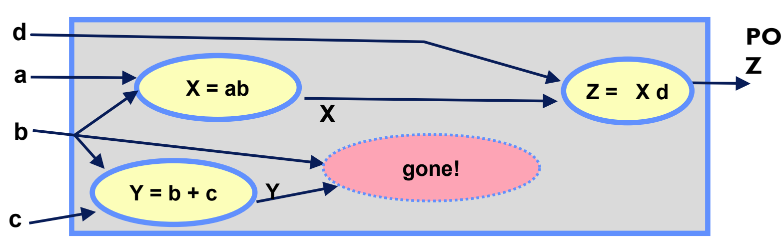





## Informal Multilevel DC Tour

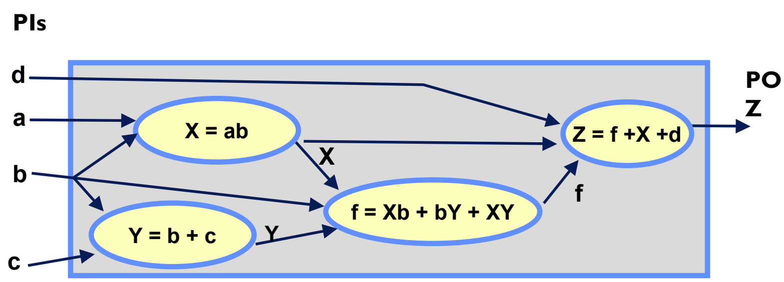
- Hey, look what happened to f node…
  - Due to context of surrounding nodes, it disappeared!



## Informal Multilevel DC Tour

- OK, suppose instead that PO Z = f + X + d (OR not AND)
  - What changes?
  - Answer: no patterns at f inputs that make Z insensitive to changes in f
  - There are still impossible patterns of (f X d) but you cannot specify any of them exactly only knowing the (X b Y) inputs to f
  - f doesn't dissappear, it still simplifies to f = b + X
- Network context matters a lot here!



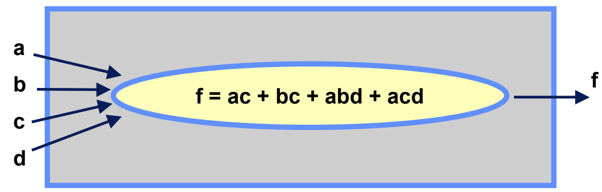



## Formal View of These DCs

- Overall, there are 3 types of formal DCs…
  - Satisfiability don't cares
    - Patterns that can't occur at the inputs to a vertex…
    - … because of internal structure of multi-level logic
  - Controllability don't cares
    - Global, external: patterns that can't happen at primary inputs to our overall Boolean logic network
    - Local, internal: patterns that can't happen at inputs to a vertex
  - Observability don't cares
    - Patterns at input of a vertex that prevent that outputs of the network from being sensitive to changes in output of that vertex
    - Pattern that "mask" outputs
- Let's see if we can clarify where these each come from…

## Don't Care Types: Satisfiability

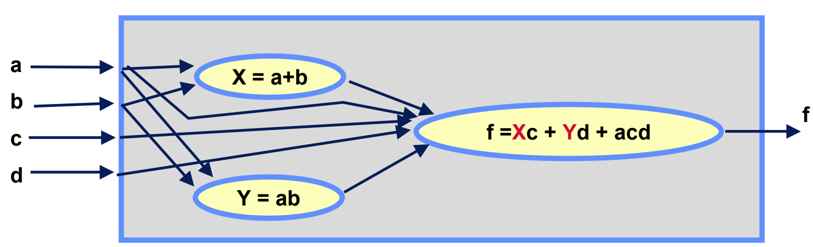
- Satisfiability Don't Cares
  - Happen because of structure of Boolean Logic Network
  - We don't treat the network as one big logic diagram, but rather, as a set of separate, connected logic blocks (vertices)
  - SDCs specify the constraints on these internal connections
- Example
  - Start with just one vertex in network





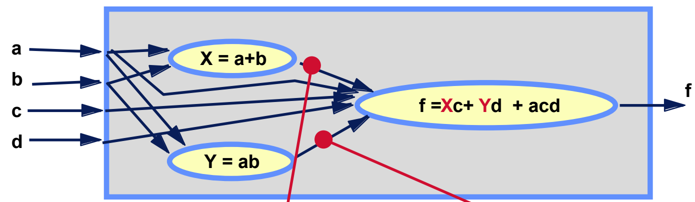

## SDCs

- Now, assume we extract some subexpressions
  - Extract **X=a+b, Y = a•b**
  - This changes structure of network
  - There are now new nodes, feeding node that creates f



# SDCs

- **Notice**
  - In the restructured network, f has different inputs, and so possibly now a different "best" simplification
  - What about don't cares?




| a | b | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

never see abX = 001, 010, 100 110

| a | b | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

never see abY = 001, 011, 101 110



# SDCs

- **These patterns are the *satisfiability* DCs**
  - Easiest to think of them as a separate set of impossible patterns "belonging to" each internal wire in a Boolean Logic Network
  - They are purely structural in origin: outputs can't take values that are not equal to (ie, don't *satisfy*) what the attached vertex creates
    - Cannot have X != a+b

X = a+b

| a | b | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

never see abX = 001, 010, 100 110

Cannot have Y != a•b

Y = a•b

| a | b | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

never see abY = 001, 011, 101 110

## Aside: How Will We Actually Represent DCs?

- Some confusing notation and terminology
  - You're probably used to seeing don't cares just listed in the truth table
  - But, the way we will usually represent these is either:
    - As a set of patterns of 0s 1s on a node's inputs that cannot happen
    - As a function of these inputs that makes a 1 just for those patterns that cannot happen; $DC_G$ == 1 just for impossible patterns for G

| p q r | G |
|---|---|
| 0 0 0 | 1 |
| 0 0 1 | d |
| 0 1 0 | 0 |
| 0 1 1 | 1 |
| 1 0 0 | d |
| 1 0 1 | d |
| 1 1 0 | 0 |
| 1 1 1 | d |

DC patterns = {001, 100, 101, 111}

| p q r | $DC_G$ |
|---|---|
| 0 0 0 | 0 |
| 0 0 1 | 1 |
| 0 1 0 | 0 |
| 0 1 1 | 0 |
| 1 0 0 | 1 |
| 1 0 1 | 1 |
| 1 1 0 | 0 |
| 1 1 1 | 1 |



## Aside: Representing Don't Cares

- Representation
  - Will even frequently see the DC function actually written in terms of an SOP cover, a Boolean expression

| a b c | G |
|---|---|
| 0 0 0 | 1 |
| 0 0 1 | d |
| 0 1 0 | 0 |
| 0 1 1 | 1 |
| 1 0 0 | d |
| 1 0 1 | d |
| 1 1 0 | 0 |
| 1 1 1 | d |

DC set = {001, 100, 101, 111}

| a b c | $DC_G$ |
|---|---|
| 0 0 0 | 0 |
| 0 0 1 | 1 |
| 0 1 0 | 0 |
| 0 1 1 | 0 |
| 1 0 0 | 1 |
| 1 0 1 | 1 |
| 1 1 0 | 0 |
| 1 1 1 | 1 |

$$DC_G = a' b' c + a b' c' + a b' c + a b c$$

## Back to SDCs

- SDC "function" for wire is just a cover of illegal patterns

X = a+b

possible

| a | b | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

impossible

| a b X | $SDC_X$ |
|---|---|
| 0 0 0 | 0 |
| 0 0 1 | 1 |
| 0 1 0 | 1 |
| 0 1 1 | 0 |
| 1 0 0 | 1 |
| 1 0 1 | 0 |
| 1 1 0 | 1 |
| 1 1 1 | 0 |

$SDC_X$ =

Y= a•b

possible

| a | b | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

impossible

| a b Y | $SDC_Y$ |
|---|---|
| 0 0 0 | 0 |
| 0 0 1 | 1 |
| 0 1 0 | 0 |
| 0 1 1 | 1 |
| 1 0 0 | 0 |
| 1 0 1 | 1 |
| 1 1 0 | 1 |
| 1 1 1 | 0 |

$SDC_Y$ = a'b'Y + a'bY + ab'Y + abY'



## SDCs: How Do We Actually Use Them?

- What impact on simplification of f...?
  - Look at SDCs for each input wire x and y

$SDC_X$ = a'b'X + a'bX' + ab'X' + abX'

=> impossible patterns of a b X

$SDC_Y$ = a'b'Y + a'bY + ab'Y + abY'

=> impossible patterns of a b Y

??

f = Xc + Yd + acd

=> but I want to know impossible patterns a c d X Y !

- Oops! Problem
  - SDCs in terms of a b X and a b Y
  - But, f is now only a function of a c d and X Y
  - How to resolve?

## SDCs

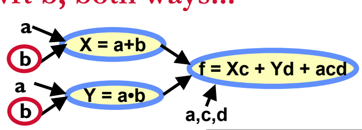
- **Need to quantify out the "b" in SDCs, but how?**
  - Just try each way and see what happens, for insight
  - Recall: given f(x,y,z,w)
    - $(\exists\, x\, f)(y,z,w) = f_x + f_{x'}$ (existential quantification)
    - $(\forall\, x\, f)(y,z,w) = f_x \bullet f_{x'}$ (universal quantification)
- **In English**
  - *Existential quantification*: removes var x, resulting function is true for (y,z,w) whenever there is some pattern,
    either (x=1,y,z,w) OR (x=0,y,z,w) that made original f == 1
  - *Universal quantification*: removes var x, resulting function is true for (y,z,w) whenever both patterns
    (x=1,y,z,w) AND (x=0,y,z,w) made original function f==1



## SDCs

- **Try quantifying wrt b, both ways...**

X = a+b

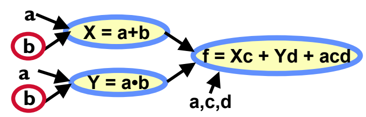


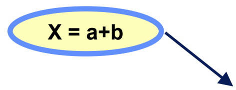

Y = a•b

| SDC for X | SDC for Y |
|---|---|
| $SDC_X = a'b'X + a'bX' + ab'X' + abX'$ | $SDC_Y = a'b'Y + a'bY + ab'Y + abY'$ |
| $(SDC_X)_b =$ | $(SDC_Y)_b =$ |
| $(SDC_X)_{b'} =$ | $(SDC_Y)_{b'} =$ |
| $(\exists\, b\, SDC_X)(a,X) = (SDC_X)_b + (SDC_X)_{b'} =$ | $(\exists\, b\, SDC_Y)(a,Y) = (SDC_Y)_b + (SDC_Y)_{b'} =$ |
| $(\forall\, b\, SDC_X)(a,X) = (SDC_X)_b \bullet (SDC_X)_{b'} =$ | $(\forall\, b\, SDC_Y)(a,Y) = (SDC_Y)_b \bullet (SDC_Y)_{b'} =$ |

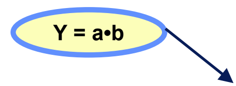
## SDCs

- Which is right?




never see abX = 001, 010, 100 110




never see abY = 001, 011, 101 110

| | |
|---|---|
| $(\exists$ b $SDC_X)(a,X)$ = X' + a' | $(\exists$ b $SDC_Y)(a,Y)$ = Y + a |
| $(\forall$ b $SDC_X)(a,X)$ = aX' | $(\forall$ b $SDC_Y)(a,Y)$ = a'Y |



## SDCs

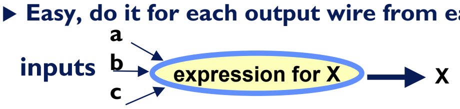
- Aside: why *universal* quantification does the trick
  - Because you want to know which patterns, independent of the value of the var(s) you get rid of, are still impossible
  - The "independent of value of var" part is the key, it's what universal quantification does

$(\exists$ b $SDC_X)(a,X)$ =X' + a' doesn't work...

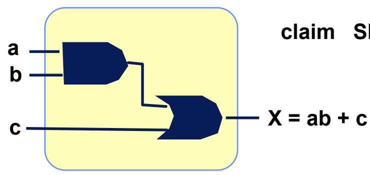

$(\forall$ b $SDC_X)(a,X)$ =aX' is right

## SDCs

- So, how do you actually *compute* SDCs?
  - Easy, do it for each output wire from each Boolean node




  - You want an expression that's ==1 when X != (expression for X)
  - But this is just
    - Remember (expression for X) doesn't have X in it!!
  - Try it on something simple to convince yourself

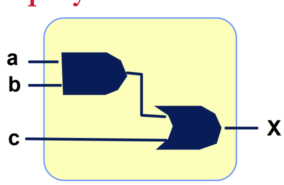



claim $SDC_X$ =



## SDCs

- Simplify…

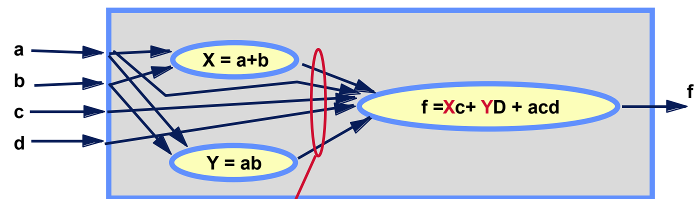



| abcX | SDC |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 1 |
| 0011 | 0 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 0 |
| 1000 | 0 |
| 1001 | 1 |
| 1010 | 1 |
| 1011 | 0 |
| 1100 | 1 |
| 1101 | 0 |
| 1110 | 1 |
| 1111 | 0 |

SDC = $X \oplus (ab + c)$ =

## SDCs

- **How to deal with SDCs on many wires into vertex?**
  - In other words, how do I actually use $SDC_X$ $SDC_Y$ to simplify f?
  - Answer: just OR the SDCs for each input wire to vertex, then quantify away vars that are *not* inputs to f




vars are **X, Y, a, c, d**

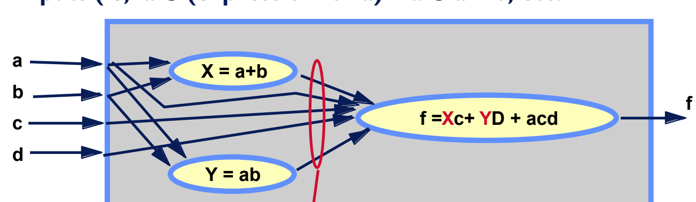
$$SDC = (\forall \text{ vars not input to f}) \left( \sum_{\text{inputs } i} SDCi \right)$$

=patterns that cannot occur on f inputs



## SDCs

- **Try it and see**
  - Note we can ignore SDCs on a, c, d inputs to f since they are primary inputs (ie, a ⊕ (expression for a) = a ⊕ a = 0, etc.

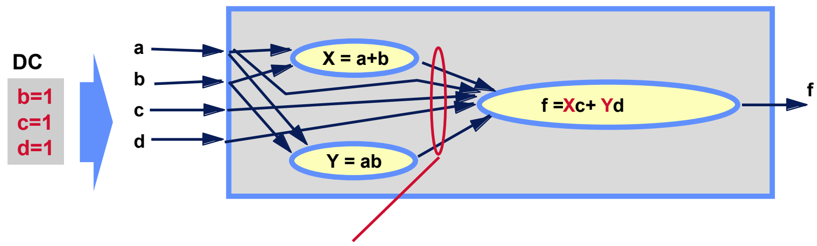



$$SDC = (\forall\, b)\big( [X \oplus (a+b)] + [Y \oplus ab] \big) =$$

$$\big( [X \oplus (a+b)] + [Y \oplus ab] \big)_{b=1} \cdot \big( [X \oplus (a+b)] + [Y \oplus ab] \big)_{b=0}$$

# SDCs

- Twist: but what if there are actually DCs for *network* inputs, impossible *external* patterns for abcd?
  - Example: suppose **b=1 c=1 d=1** can never happen
  - How to handle for computing SDCs for f?
  - Answer: just OR in cover for these DCs in SDC expression



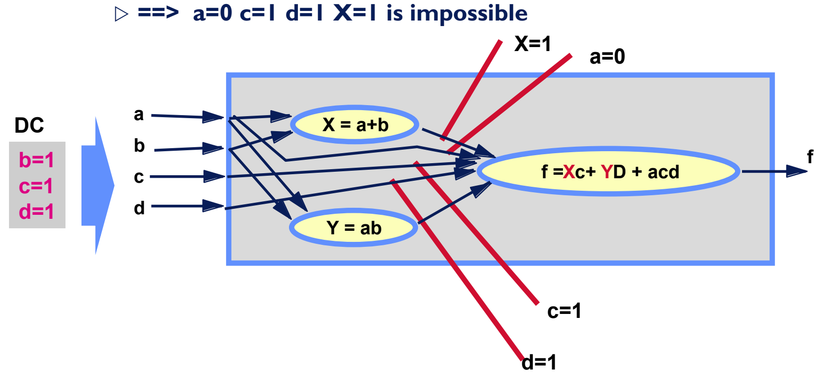

$$SDC = (\forall\, b)\big( [X \oplus (a+b)] + [Y \oplus ab] + \mathbf{bcd} \big) =$$

=[ stuff we got before without these external DCs ] + $\boxed{\phantom{xxxxxxxx}}$



# SDCs

- Notice this works…
  - Just look at the new terms we added to SDC: **( a'cdX + acdX' + cdY)**
  - Pick **a'cdX** as example
    - ==> a=0 c=1 d=1 X=1 is impossible




Correct! This can only happen for input abcd = 0111 which is impossible

## Controllability Don't Cares

- Defines those input patterns that cannot happen for specific vertices, or for entire network
  - **But, we've already seen these!**
  - **External global CDCs: come from outside for entire network, like b=1 c=1 d=1 is impossible, in our example**
  - **Internal local CDCs: just patterns that cannot appear at any vertex**

$$= (\forall \text{ vars not inputs})\Big(\sum_{\substack{\text{vertex}\\ \text{inputs}}} (\text{local SDCs}) + \text{ext. global CDC}\Big)$$

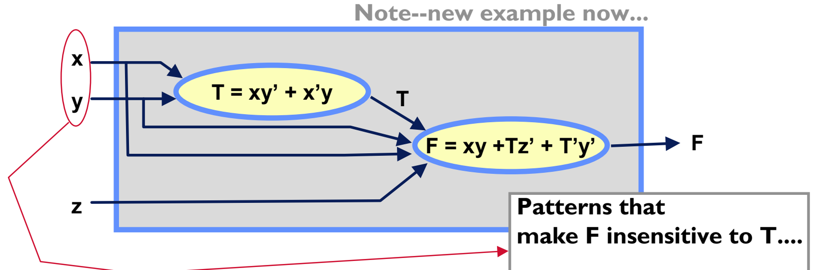
- SDCs versus CDCs ...?
  - **SDCs: think of as belonging to each internal wire in network**
  - **CDCs: think of as belonging to each internal vertex in network**



## Observability Don't Cares

- ODCs belong to *each* output of a vertex in network
  - **Patterns that will make this output not observable at network output**
  - **"Not observable" means a change 0<->1 on this vertex output doesn't not change ANY network output, for this pattern**
  - **New example**

Note--new example now...




  - **Look at $ODC_T$ for output wire of vertex T**

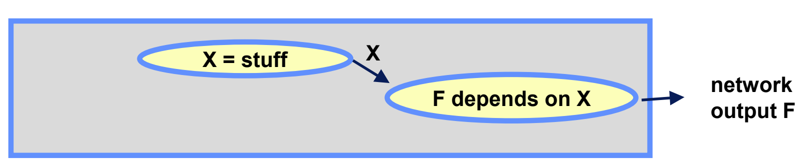
## ODCs

- In English…
  - ODC for T are patterns of inputs to the vertex for T (patterns of x,y) such that we can compute F without caring about what T is
  - Since F= xy + Tz' + T'y', observe
    - If x=1 y=1 then F = 1 + Tz' + T'y' =1 = independent of T
    - Note there are patterns of other vars that do this too:
    - If z=1 y=1 then F = xy +T•0 + T'•0 = xy = independent of T
    - If z=0 y=0 then F = xy + T•1 + T'•1 = xy + T + T' = 1 = indep. of T
  - So, our guess is that $ODC_T$ = xy
    - This is the *only* pattern that depends just on vars input to T
    - For this pattern, network output *insensitive* to changes in T
  - How to compute, mechanically?



## ODCs

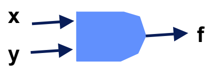
- When is network output F *insensitive* to internal var X?



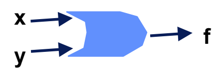

- Be precise
  - *Insensitive* means X changes => but F never changes
  - More precisely: if we specify F as function of X, then $F_X = F_{X'}$
  - So, what *patterns* of the *other* inputs to F cause F(… X=0 ….)=F(…X=1 …)?
  - When *these patterns* are applied, changing X does not ever matter to output at F
  - But we've already seen something close to this…

# ODCs

- **Boolean difference, $\partial F / \partial X$**
  - **Defined as $\partial F / \partial X = F_X \oplus F_{X'}$**
  - **Recall we observed that patterns that make $\partial F / \partial X = 1$ correspond to patterns where a change in X causes some change in F**

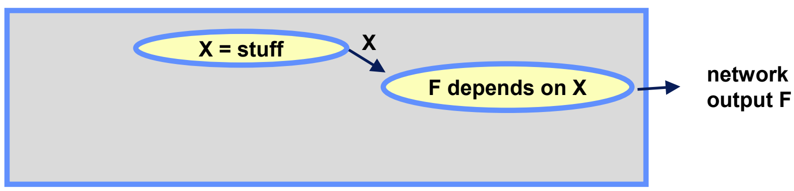



$\partial F / \partial x = y$




$\partial F / \partial x = y'$

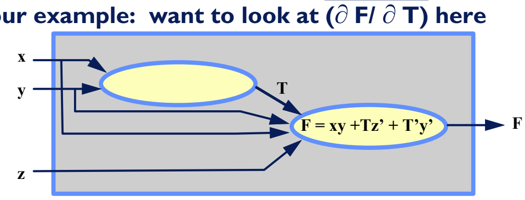
- **Stated differently**
  - **Boolean difference • $\partial F / \partial X$ is a function that is 1 for those patterns that make X *observable***






# ODCs

- **But we want patterns that make vertex output X *unobservable*, since we want don't care patterns**
  - **So, if $\partial F / \partial X$ is patterns that make X observable**
  - **...then $\overline{(\partial F / \partial X)}$ is patterns that make X *unobservable***
  - **Back to our example: want to look at $\overline{(\partial F / \partial T)}$ here**

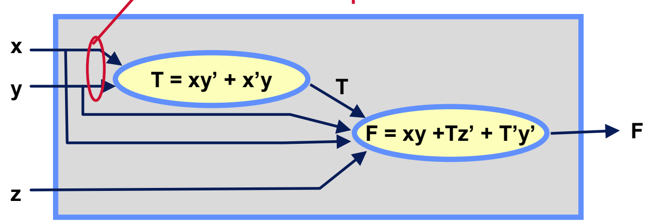



$\overline{\partial F / \partial T} = \overline{F_T \oplus F_{T'}} =$

## ODCs

- So $ODC_T = xy + xz' + y'z' + x'yz$
  - But, same problem: can't use this to simplify vertex for T since T is only a function of x and y
  - What to do?
  - Same as before: *universal* quantification over vars not input to T
  - In this case, want $(\forall z)(xy + xz' + y'z' + x'yz) = xy$ which is correct

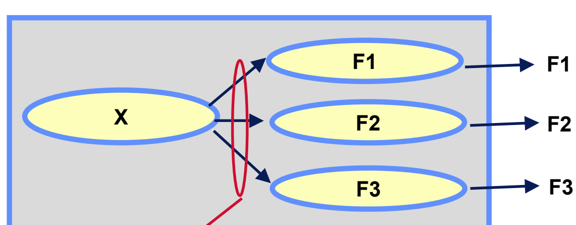
$ODC_T = xy$ => x=1 y=1 is don't care since it makes T unobservable at network output F




© R. Rutenbar 2001, CMU 18-760, Fall 2001 45

## ODCs

- More general: what if many network outputs?
  - Only patterns that are unobservable at ALL outputs can be ODCs




AND all derivatives together, for each output

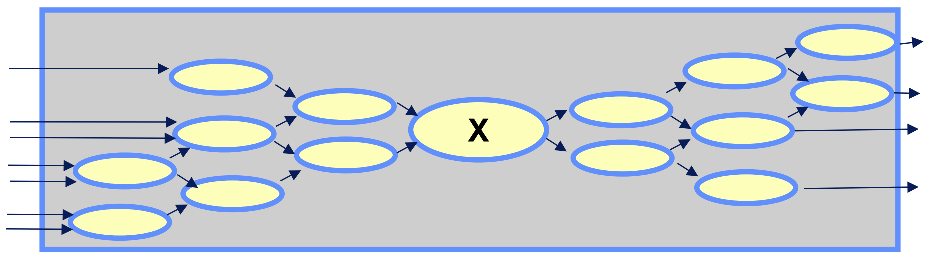
$$ODC = (\forall \text{ vars not input to } x)\left[\prod_{\text{outputs } F_i} \overline{\partial F_i / \partial x}\right]$$

= patterns that make x unobservable at ALL f outputs

© R. Rutenbar 2001, CMU 18-760, Fall 2001 46

# Don't Cares, In General

- Why is getting these things so *very hard*?




- Because real networks are big, and the vertex **X** you want to simplify may be *very far* from the primary inputs, and primary outputs
- Inputs to your vertex are function of a lot of stuff
- Network outputs are functions of your vertex and lots of *other* stuff
- Representing all this stuff can be explosively large, *even* with BDDs

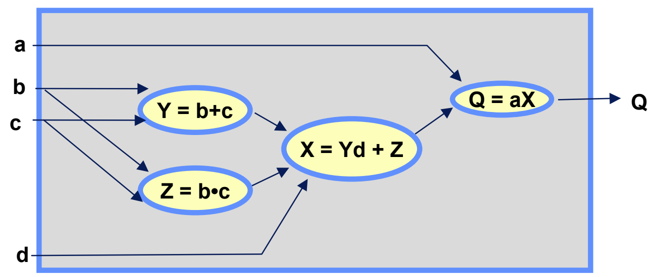


# Getting Network DCs

- How do people do it
  - In general, they don't
    - Usually suffice with getting the local SDCs, which just requires looking at outputs of antecedent vertices and computing the SDC patterns, which is easy (no big search)
    - There are also incremental, vertex-by-vertex algorithms that walk the network to compute full CDC set for **X**, and full ODC set for **X**, but these can be very expensive in space
  - Also, some tricks called FILTERS
    - You want to find patterns you can use as don't cares to simplify vertex **X**
    - Instead of finding all such DC patterns, can restrict search to avoid patterns that cannot possibly be useful to simplify **X**
    - Such algorithms called "filters" -- they get rid of DCs you don't need
  - See De Michelli for details about all this stuff
    - For us, knowing the straightforward brute force formula is OK

## Example

- Now know enough to do this
  - Simplify node **X** inside this network
  - Assume pattern a=0 d=0 never occurs at network input




- How?
  - Compute SDC for **X**, including external global DC=a'd'
  - Compute ODC for **X** by doing ( $\partial$ Q/ $\partial$ **X** )'
  - You get to use anyplace $SDC_X + ODC_X == 1$ as a don't care for **X**



## Summary

- New kinds of don't cares in multi-level networks
  - Byproducts of the network model
  - It's not all one big function, it's a bunch of little functions (vertices) connected by wires
  - **Satisfiability DCs:** structural in origin, can't have output of a vertex not equal to the expression for that vertex
  - **Output DCs:** some patterns make vertex output unobservable at network outputs
  - **SDC + ODC:** for any given vertex, can use this expression as places for don't cares to simplify the vertex function
- In practice
  - Very *hard* to get these, esp ODCs (see the book)
  - Usually just use the local SDC from antecedent vertices
  - Also, there are algorithms ( *filters* ) that can just go find *useful* don't cares for simplification