



Logic Synthesis – Part 1

Technology-Independent Optimization

Prof. Chien-Nan Liu
Institute of Electronics
National Chiao-Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nctu.edu.tw
<http://www.ee.ncu.edu.tw/~jimmy>

Courtesy: Prof. Jing-Yang Jou

NCTU M S E D A Lab.



1

Outline

- Synthesis overview
- RTL synthesis
- Two-level logic optimization
- Multi-level logic optimization
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power

NCTU M S E D A Lab.



2

HDL Synthesis

- **Logic synthesis** programs transform Boolean expressions or **register-transfer level (RTL)** description (in Verilog/VHDL/C) into logic gate networks (netlist) in a particular library.
- Advantages
 - Reduce time to generate netlists
 - Easier to retarget designs from one technology to another
 - Reduce debugging effort
- Requirement
 - **Robust** HDL synthesizers

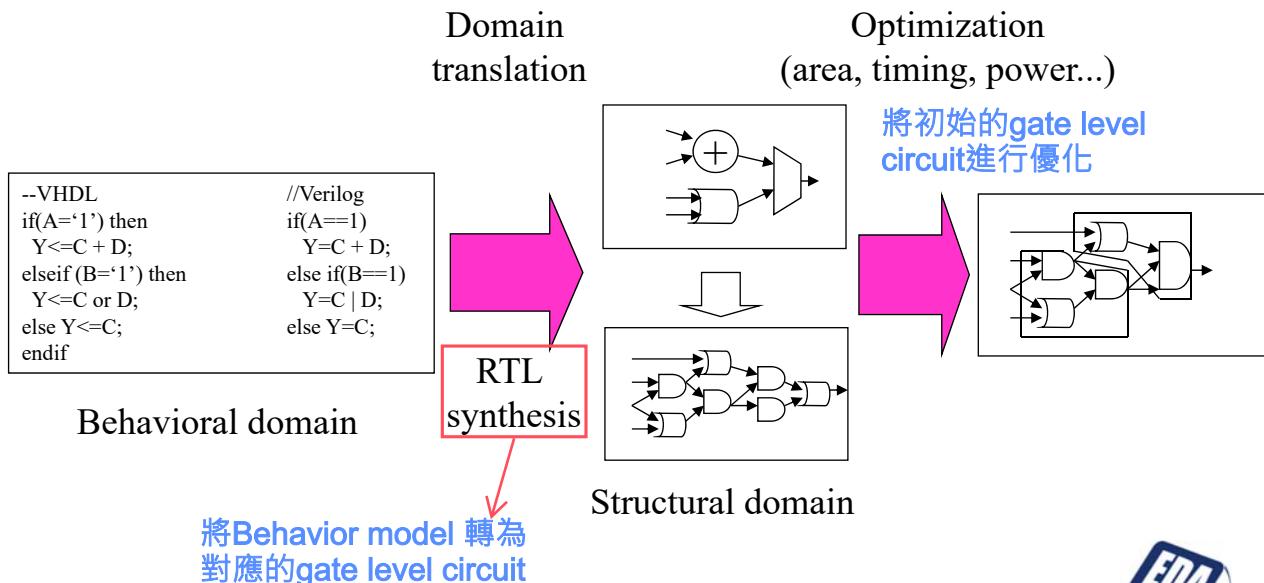
NCTU M SEDA Lab.



3

Synthesis Procedure

Synthesis = Domain Translation + Optimization

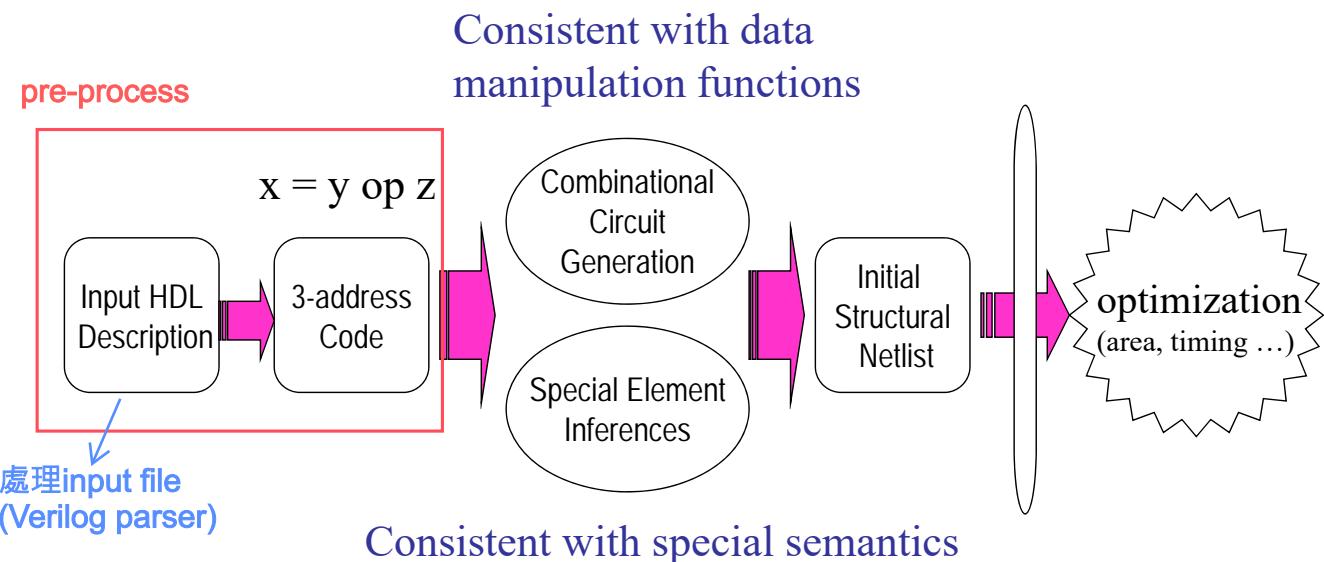


NCTU M SEDA Lab.



4

Domain Translation

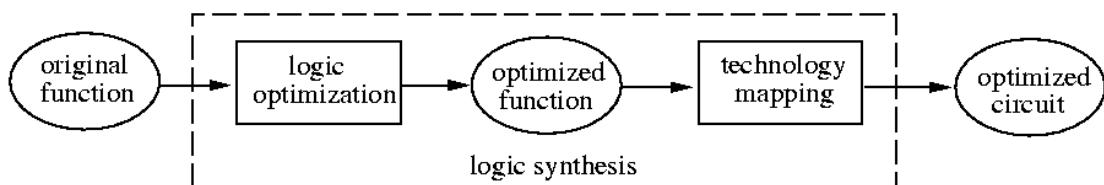


NCTU M S E D A Lab.



5

Optimization



- **Technology-independent** optimization: **logic optimization**

- Work on Boolean expression equivalent
- Estimate size based on # of literals
- Use simple delay models 較不準確

- **Technology-dependent** optimization: **technology mapping/library binding**

- Map Boolean expressions into a particular cell library 將Boolean expression對應到指定的製程檔
- May perform some optimizations in addition to simple mapping 有用的資訊更多，可以進一步做最佳化
- Use more accurate delay models based on cell structures

NCTU M S E D A Lab.



6

Technology-Independent Logic Optimization

- **Two-level:** minimize the # of product terms.

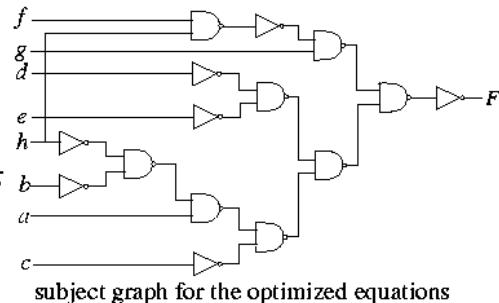
- $F = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 \Rightarrow F = \bar{x}_2 + x_1 \bar{x}_3.$

- **Multi-level:** minimize the #'s of literals, variables.

- E.g., equations are optimized using a smaller number of literals.

尋找是否有元件可以共用

$$\begin{array}{l} t1 = a + b + c; \\ t2 = d + e; \\ t3 = a \cdot b + d; \\ t4 = t1 \cdot t2 + f \cdot g; \\ t5 = t4 \cdot h + t2 \cdot t3; \\ F = t5'; \end{array} \xrightarrow{\text{logic optimization}} \begin{array}{l} t1 = d + e; \\ t2 = b + h; \\ t3 = a \cdot t2 + c; \\ t4 = t1 \cdot t3 + f \cdot g \cdot h; \\ F = t5'; \end{array}$$



subject graph for the optimized equations

- Methods/CAD tools: Quine-McCluskey method

Espresso的團隊就是
新思的創辦團隊

(exponential-time exact algorithm), Espresso (heuristics for two-level logic), MIS (heuristics for multi-level logic), Synopsys, etc.

NCTU M SEDA Lab.



7

Technology Mapping

- Goal: translation of a technology independent representation (e.g. Boolean networks) of a circuit into a circuit in a given technology (e.g. standard cells) with optimal cost
- Optimization criteria:
 - Minimum area
 - Minimum delay
 - Meeting specified timing constraints
 - Meeting specified timing constraints with minimum area
- Usage:
 - Technology mapping after technology independent logic optimization
 - Technology translation

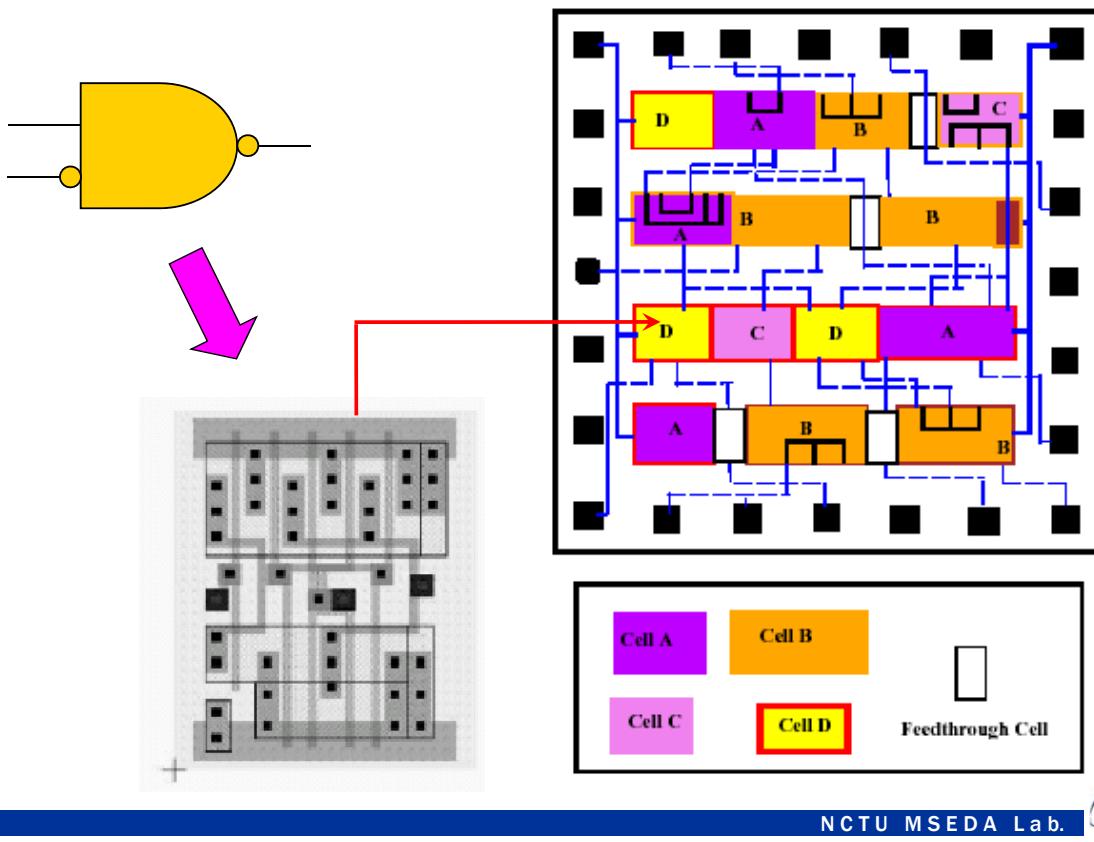
Optimization時需要考慮PPA，但是通常area與 performance是trade off的關係，因此在此階段就會去嘗試各種不同的製程、cell library

NCTU M SEDA Lab.



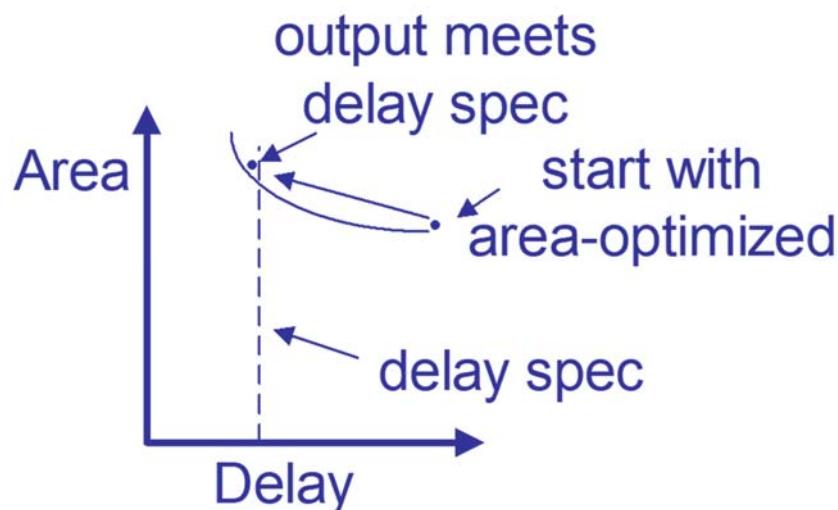
8

Standard Cells for Design Implementation



Timing Optimization

- There is always a trade-off between area and delay
- Optimize timing to meet delay spec. with minimum area



Outline

- Synthesis overview
- **RTL synthesis**
 - Combinational circuit generation
 - Special element inferences
- Two-level logic optimization
- Multi-level logic optimization
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power

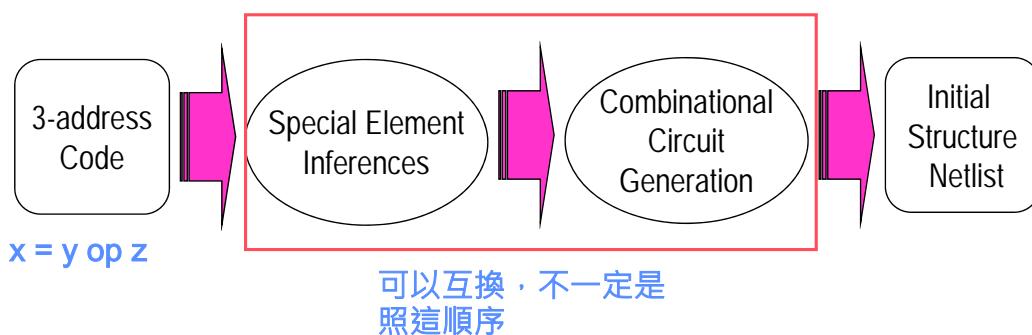
NCTU M SEDA Lab.



11

Typical Domain Translation Flow

- Translate original HDL code into 3-address format
- Conduct special element inferences before combinational circuit generation
- Conduct special element inferences process by process (local view)



NCTU M SEDA Lab.



12

Combinational Circuit Generation

- Functional unit allocation
 - Straightforward mapping with 3-address code
- Interconnection binding
 - Using control/data flow analysis

Functional Unit Allocation

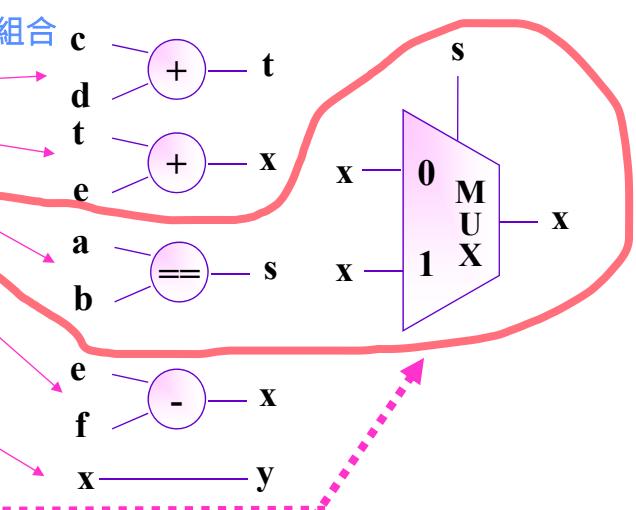
- 3-address code
 - $x = y \text{ op } z$ in general form
 - Function unit op with inputs y and z and output x

遇到連續運算的式子，將拆解為多個2-input的組合

$x=c+d+e;$
 $\text{if}(a==b) x= e-f;$
 $y=x;$

3-address code

Implicit multiplexer



Interconnection Binding

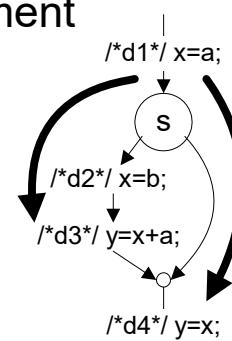
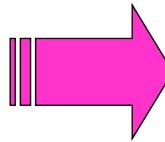
- Need the dependency information among functional units
 - Using **control/data flow analysis**
 - A traditional technique used in compiler design for a variety of code optimizations
 - Statically analyze and compute the set of assignments reaching a particular point in a program



Control/Data Flow Analysis

- Terminology
 - A **definition** of a variable x definition中都會有一個賦值符號"="
 - An **assignment** assigns a value to the variable x
 - d_1 can reach d_4 but cannot reach d_3
 - d_1 is killed by d_2 before reaching d_3 新的宣告會殺死(覆蓋)舊的宣告
- A definition can only be affected by those definitions being able to reach it
- Use a set of data flow equations to compute which assignments can reach a target assignment

```
/*d1*/ x = a;  
    if(s) begin  
/*d2*/     x = b;  
/*d3*/     y = x + a;  
    end  
/*d4*/     y = x;
```



Combinational Circuit Generation: An Example

```
always @ (x or a or b or c or d or s)
begin
/*d1*/ x = a + b;
/*d2*/ if ( s ) x = c - d;
/*d3*/ else x = x;
/*d4*/ y = x;
end
```

Input HDL

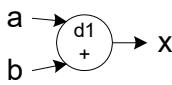
將if內的各種情況做好，
最後再用mux選擇

```
always @ (x or a or b or c or d or s)
begin
/*d1*/ x = a + b;
/*d2*/ if ( s ) x = c - d;
/*d3*/ else x = x;
/*d4*/ x = s mux x;
/*d5*/ y = x;
end
```

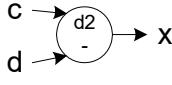
Modified 3-address code

將3-addr code mapping
為各種元件

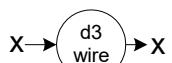
In[d1]={d4, d5} → computed by control/
data flow analysis



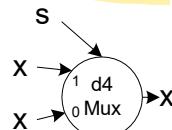
In[d2]={d1, d5}



In[d3]={*d1, d5}



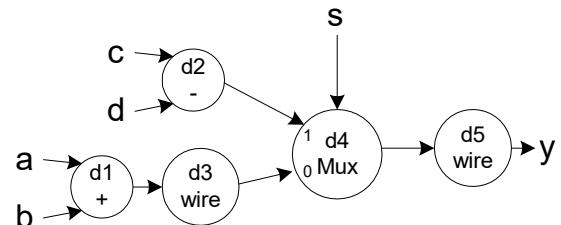
In[d4]={*d2, *d3, d5}



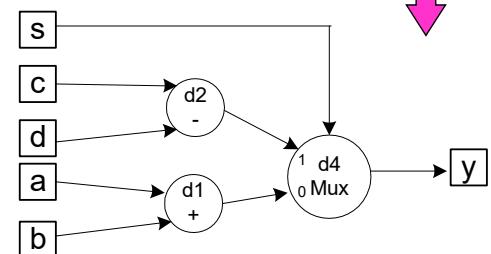
In[d5]={*d4, d5}



Functional unit allocation



Interconnection binding



Final result



Outline

- Synthesis overview
- RTL synthesis
 - Combinational circuit generation
 - Special element inferences
- Two-level logic optimization
- Multi-level logic optimization
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power



Special Element Inferences

- Given a HDL code at RTL, three special elements need to be inferred to keep the special semantics
 - Latch (D-type) inference
 - Flip-Flop (D-type) inference
 - Tri-state buffer inference
- Some simple rules are used in typical approaches

Special element
分為三種

```
reg Q;  
always@(D or en)  
if(en) Q = D;
```

Latch inferred!!

```
reg Q;  
always@(posedge clk)  
Q = D;
```

Flip-flop inferred!!

```
reg Q;  
always@(D or en)  
if(en) Q = D;  
else Q = 1'bz;
```

**Tri-state buffer
inferred!!**



Preliminaries

- Sequential section
 - Edge triggered always statement
- Combinational section
 - All signals whose values are used in the always statement are included in the sensitivity list

```
reg Q;  
always@(posedge clk)  
Q = D;
```

Sequential section
Conduct flip-flop inference

```
reg Q;  
always@(in or en)  
if(en) Q=in;
```

Combinational section
Conduct latch inference



Typical Latch Inference

- Conditional assignments are not completely specified
 - Check if the *else-clause* exists
 - Check if all case items exist
- Outputs conditionally assigned in an if-statement are not assigned before entering or after leaving the if-statement

always@(D or S)

if(S) Q = D;

↳ Infer latch
for Q

combinational circuit中沒寫else就會預設將
output拉回input，因此形成latch
(因為compiler會盡量避免電路中出現floating
point，因此如果沒有指定值就會有memory的行
為產生-->記憶前一個時刻的值)

always@(S or A or B)

begin

Q = A; → Do not infer

if(S) Q = B; latch for Q

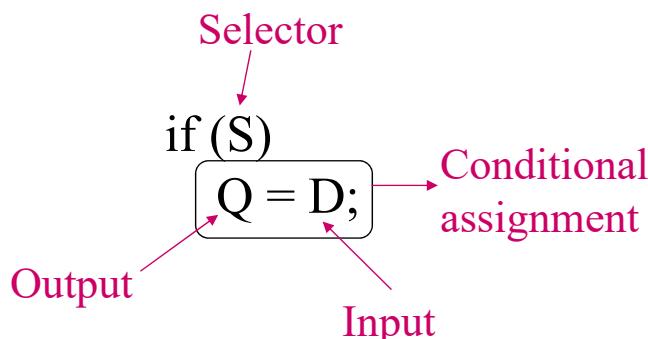
end

也可以直接給一個default值，
這樣就不用寫else



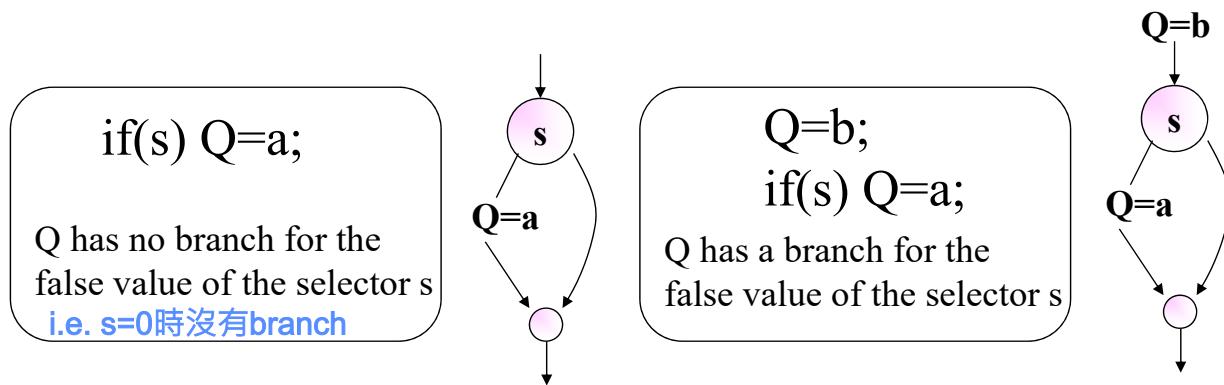
Terminology (1/2)

- Conditional assignment
- Selector: S
- Input: D
- Output: Q



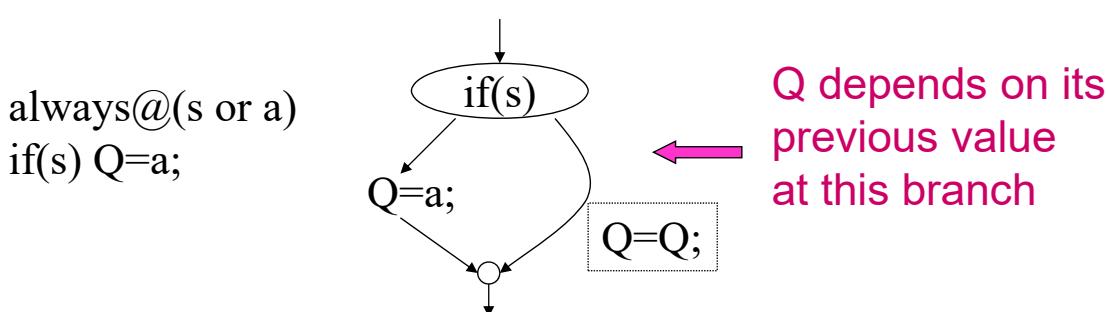
Terminology (2/2)

- A variable Q has a *branch* for a value of selector s
 - The variable Q is assigned a value in a path going through the branch



Rules of Latch Inference (1/2)

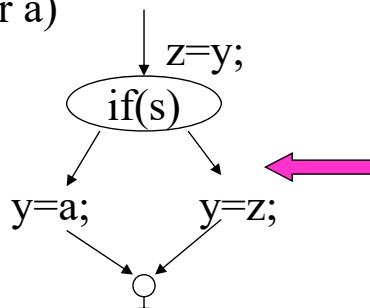
- Condition 1: There is no branch associated with the output of a conditional assignment for a value of the selector
 - Output depends on its previous value implicitly



Rules of Latch Inference (2/2)

- Condition 2: The output value of a conditional assignment depends on its previous value explicitly

```
always@(s or z or y or a)
begin
    z = y;
    if(s) y=a;
    else y=z;
end
```



y depends on its previous value at this branch via the assignment $z=y$;

在else的地方，y的value是由z給予的，而z是y於變化前的value，因此其實else等價於"y維持不變"
--> 產生latch



Terminology

- Clocked statement: edge-triggered always statement
 - Simple clocked statement
 - e.g., **always @ (posedge clock)**
 - Complex clocked statement
 - e.g., **always @ (posedge clock or posedge reset)**
- Flip-flop inference must be conducted only when synthesizing the **clocked statements**

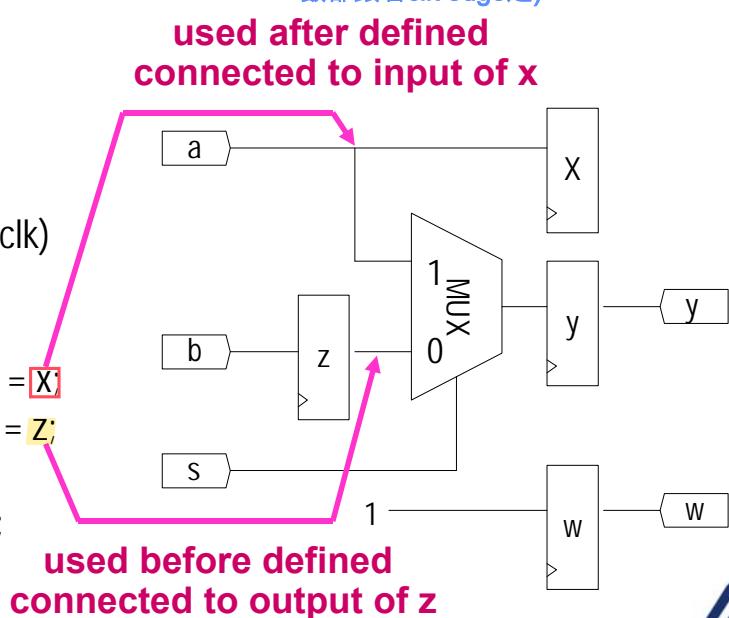
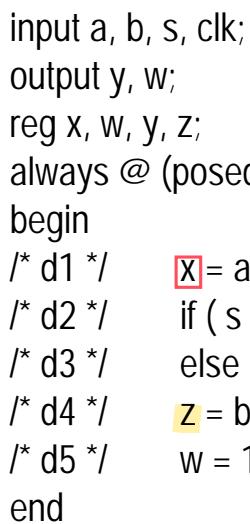
如果是edge trigger，compiler就必须合成出flip-flop
才能符合behavior model 的行為



Infer FF for Simple Clocked Statements (1/2)

- Infer a flip-flop for **each variable** being assigned in the simple clocked statement

可以發現mux吃的value有 x 與 z ，其中 z 是used before define，因此將flip-flop z 的output接在mux前面，確保mux是吃到更新前的值。 x 則是use after define，因此將flip-flop x 的input與mux接在一起



NCTU MSEA Lab.

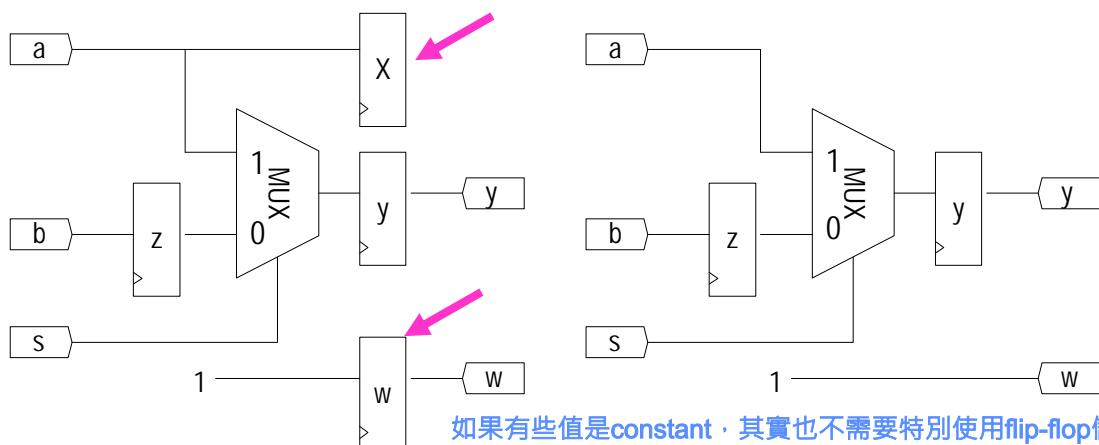


27

Infer FF for Simple Clocked Statements (2/2)

- Two post-processes
 - Propagating constants
 - Removing the flip-flops without fanouts

如果發現某些flip-flop沒有fanout，代表其所儲存的值並不會被用到，所以可以將該flip-flop移除



如果有些值是constant，其實也不需要特別使用flip-flop儲存他的值



Infer FF for Complex Clocked Statements

- The edge-triggered signal not used in the following operations is chosen as the clock signal
眾多trigger訊號中，判斷何者為clock的方法就是看哪個訊號在always block中沒有被用到
- The usage of asynchronous control pins requires the following syntactic template
 - An if-statement immediately follows the always statement
 - Each variable in the event list except the clock signal must be a selective signal of the if-statements
 - Assignments in the blocks B1 and B2 must be constant assignments (e.g., $x=1$, etc.)

always @ (posedge clock or posedge reset or negedge set)

```
if(reset) begin B1 end
else if ( !set) begin B2 end
else begin B3 end
```

NCTU M S E D A Lab.



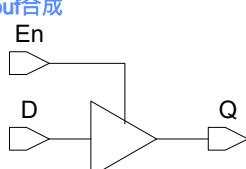
29

Typical Tri-State Buffer Inference (1/2)

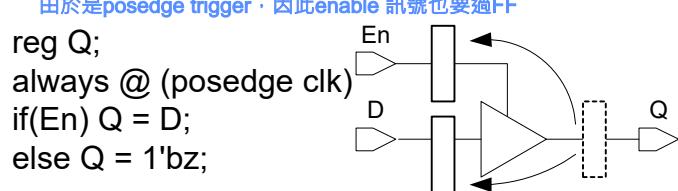
- If a data object Q is assigned a high impedance value 'Z' in a multi-way branch statement (if, case, ?:)
 - Associated Q with a tri-state buffer
- If Q associated with a tri-state buffer has also a memory attribute (latch, flip-flop)
 - Have the **Hi-Z propagation problem**
 - Real hardware cannot propagate Hi-Z value
 - Require two memory elements for the control and the data inputs of tri-state buffer

只要電路中的FF出現Hi-Z，後續的FF就會全部都變成Hi-Z
因此tri-state buf只能放在電路的最後一級，避免Hi-Z propagation

Combinational block中的tri-state buf合成
reg Q;
always @ (En or D)
if(En) Q = D;
else Q = 1'bz;



Sequential block中的tri-state buf合成
這邊就將原本位在tri-state buf後方的FF一道tri-state buf之前
由於是posedge trigger，因此enable 訊號也要過FF
reg Q;
always @ (posedge clk)
if(En) Q = D;
else Q = 1'bz;



NCTU M S E D A Lab.

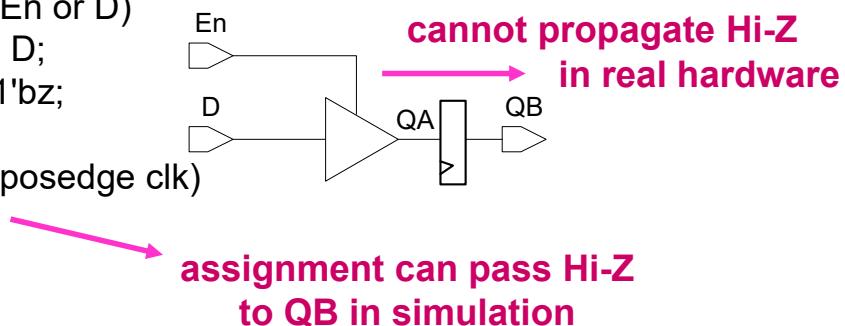


30

Typical Tri-State Buffer Inference (2/2)

- It may suffer from mismatches between synthesis and simulation
 - Process by process
 - May incur the Hi-Z propagation problem

```
reg QA, QB;  
always @ (En or D)  
if(En) QA = D;  
else QA = 1'bz;  
  
always @ (posedge clk)  
QB = QA;
```



Outline

- Synthesis overview
- RTL synthesis
- Two-level logic optimization
 - Basic logic operations
 - Exact minimization
 - Heuristic methods
- Multi-level logic optimization
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power



Two-Level Logic Optimization

- Two-level logic optimization
 - Key technique in logic optimization
 - Many efficient algorithms to find a near minimal representation in a practical amount of time
 - In commercial use for several years
 - Minimization criteria: **number of product terms**
- Example: $F = XYZ + X\bar{Y}\bar{Z} + X\bar{Y}Z + \bar{X}YZ + XY\bar{Y}Z$



$$F = X\bar{Y} + YZ$$



Optimization Approach

- Exact Methods:
 - Compute minimum cover
 - Often impossible for large functions
 - Ex: Karnaugh maps, Quine-McCluskey
K-map較難寫為程式執行
- Heuristic Methods:
 - Compute minimal covers (possibly minimum) in reasonable time
 - Large variety of methods and programs
 - Ex: MINI, PRESTO, ESPRESSO



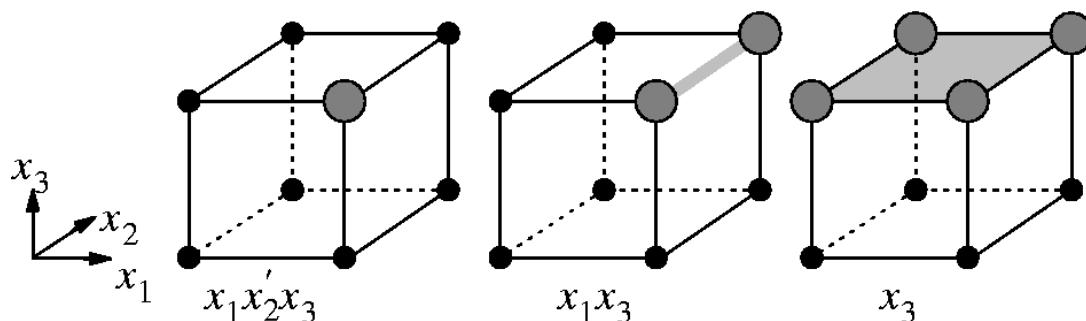
Boolean Functions

- $B = \{0, 1\}$, $Y = \{0, 1, D\}$
- A Boolean function $f: B^m \rightarrow Y^n$
 - $f = \bar{x}_1 \bar{x}_2 + \bar{x}_1 \bar{x}_3 + \bar{x}_2 x_3 + x_1 x_2 + x_2 \bar{x}_3 + x_1 x_3$
- Input variables: x_1, x_2, \dots
- The value of the output partitions B^m into three sets
 - the **on-set** 所有使output為1的邏輯組合
 - the **off-set** 所有使output為0的邏輯組合
 - the **dc-set (don't-care set)** 如果有dont care，就會導致每個人化簡出的邏輯不太一樣



Minterms and Cubes

- A **minterm** is a product of all input variables or their negations.
 - A minterm corresponds to a single point in B^n .
- A **cube** is a product of the input variables or their negations.
 - The fewer the number of variables in the product, the bigger the space covered by the cube.



三個變數指定一個點 兩個變數指定一條線 一個變數指定一個面



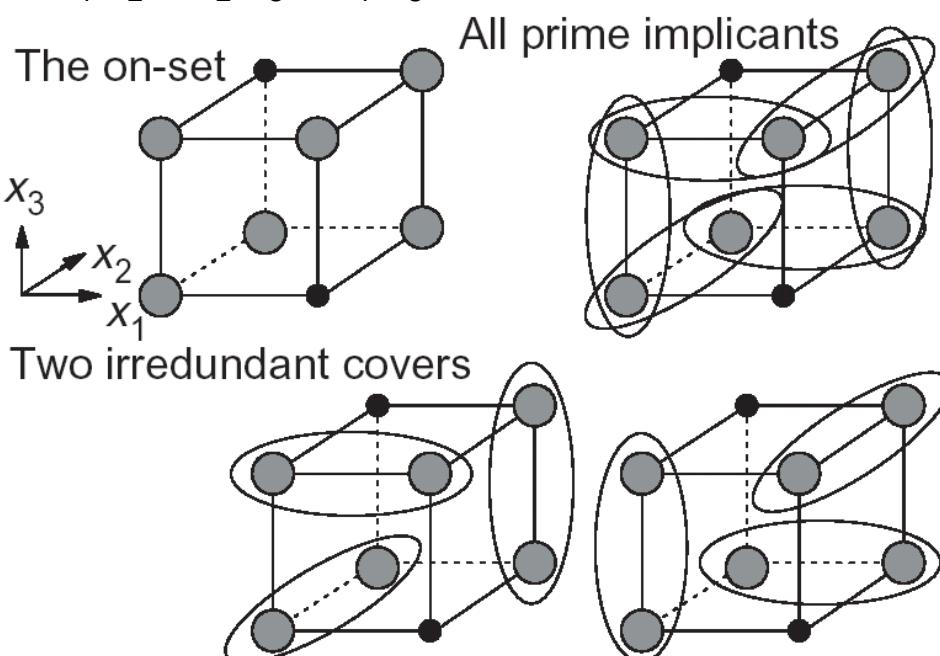
Implicant and Cover

- An **implicant** is a cube whose points are either in the on-set or the dc-set. implicant就是只圈1與don't care
- A **prime implicant** is an implicant that is not included in any other implicant. 就是K-map中local最大的圈圈(不能再長大)
- A set of prime implicants that together cover all points in the on-set (and some or all points of the dc-set) is called a prime cover. prime cover就是將能夠cover所有on-set的prime implicants中重複的一些元素拿掉，變成最簡
- A prime cover is **irredundant** when none of its prime implicants can be removed from the cover.
- An irredundant prime cover is **minimal** when the cover has the minimal number of prime implicants.
但是prime cover中也有不同的cost，因此我們的目標就是找到minimal prime cover



Cover Examples

- $f = \bar{x}_1 \bar{x}_3 + \bar{x}_2 x_3 + x_1 x_2$
- $f = \bar{x}_1 \bar{x}_2 + x_2 \bar{x}_3 + x_1 x_3$



The Positional-Cube Notation 一種節省記憶體的方法

- Encode each symbol by 2-bit fields as follows:

Hi-Z	ϕ	00	電路中只會有四種狀態，因此只需要兩個bit
	0	10	就可以表示所有組合
	1	01	因此一個32-bit的integer就可以表示16個variables的狀態
don't care	*	11	

One 32-bit integer \rightarrow 16 binary digits



- Example: $f = a'd' + a'b + ab' + ac'd$ f中有a,b,c,d四個variables

10 11 11 10 ($a' - d'$) $a'd'$ 中， $a=0, b=X, c=X, d=0$ ，對應上表可以得到左邊的8個數字
 10 01 11 11 ($a' b - -$)
 01 10 11 11 ($a b' - -$)
 01 11 10 01 ($a - c' d$)

- Example: $f_1 = a'b' + ab; f_2 = ab; f_3 = ab' + a'b$

10 10 100 ($a'b'$)
 10 01 001 ($a'b$)
 01 10 001 (ab')
 01 01 110 (ab)
 f1 ↑ f2 ↑ f3

先把所有ab的組合用上面的代號寫出來，再看哪些f會用到，就將其紀錄為1
 ex: f1會用到 $a'b'$ 與 ab ，所以在對應的列就會標為1

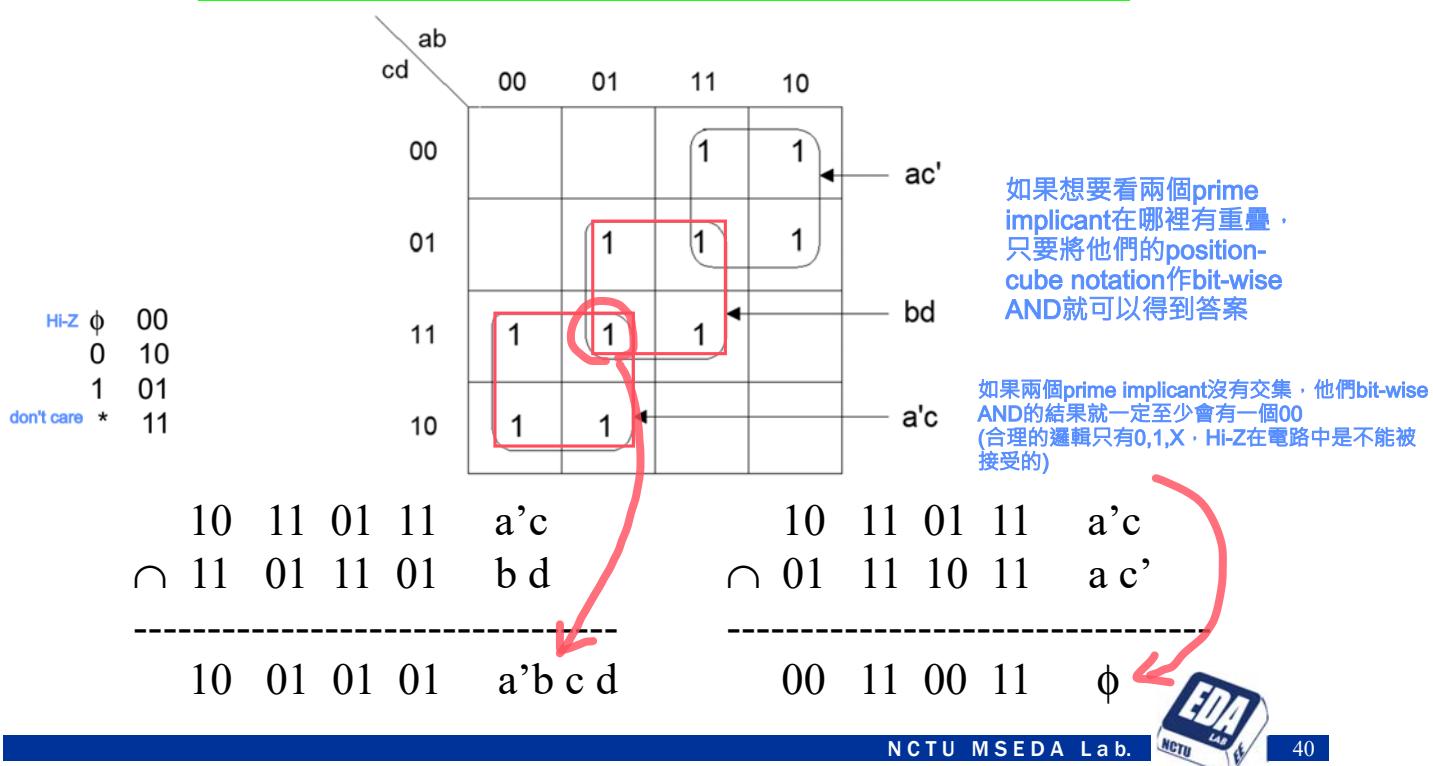
NCTU M SEDA Lab.



39

AND Operation

Can be finished with a bit-wise AND instruction !!



Sharp Operation

$$\alpha \# \beta = \begin{array}{ccccccc} a_1 b_1' & a_2 & \dots & a_{n-1} & a_n & 11*(01)' & 11 \\ a_1 & a_2 b_2' & \dots & a_{n-1} & a_n & 11 & 11*(01)' \\ \dots & \dots & \dots & \dots & \dots & \rightarrow & \\ a_1 & \dots & \dots & a_{n-1} b_{n-1}' & a_n & 11*10 & 11 \\ a_1 & \dots & \dots & a_{n-1} & a_n & 11 & 11*10 \\ & & & a_n b_n' & & 10 & 11 \\ & & & & & 11 & 10 \end{array}$$

sharp operation類似減法，表示一個prime implicant 在去除某個元素後的結果(以一組prime implicant表示)
操作方法就是先看要扣除的prime implicant有幾個元素，下面的例子來說01 01就是兩個元素
因此先將a抄兩列，然後將b1', b2'照上圖擺放後作bit-wise AND便可得到結果

- Example

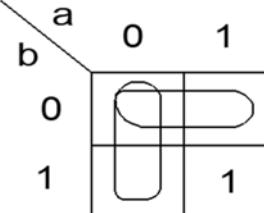
$$11\ 11 \# 01\ 01 = 10\ 11$$

11 11代表XX，就是整張圖

01 01代表右圖中右下角的那個格子

整張圖扣掉右下角的格子，可以由兩組prime implicant表示其結果

$\rightarrow 0X$ 與 $X0$ (對應成cube-notation就是10 11與11 10)



NCTU M SEDA Lab.



41

Disjoint Sharp Operation

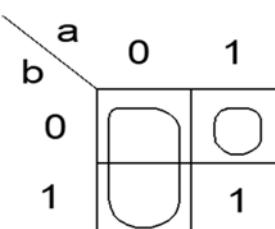
sharp operation的變形，
其產出的prime implicant
不會重疊

$$\alpha \# \beta = \begin{array}{ccccccc} a_1 b_1' & a_2 & \dots & a_{n-1} & a_n & & \\ a_1 b_1' & a_2 b_2' & \dots & a_{n-1} & a_n & & \\ \dots & \dots & \dots & \dots & \dots & & \\ a_1 b_1' & a_2 b_2' & \dots & a_{n-1} b_{n-1}' & a_n & & \\ a_1 b_1' & a_2 b_2' & \dots & a_{n-1} b_{n-1}' & a_n b_n' & & \end{array}$$

相較於普通的sharp operation
只需要補bn'，disjoint的還需要
在下三角額外補上bn'

- Example

$$11\ 11 \# 01\ 01 = 10\ 11$$



NCTU M SEDA Lab.



42

Effects of Basic Logic Operations

- Consider each implicant as a set
- **Intersection** is the largest cube contained in both implicants and is computed by **AND operation**
- The **distance** between two implicants is the **number of empty fields in their intersection** 前面有提到如果bit-wise AND的結果中有00就是disjoint，而00的個數就代表這兩個prime implicant之間的距離
 - If there is any empty field, the two implicants are disjoint
- The **supercube** of two sets (the sum of two functions) can be obtained by union the sets (**bit-wise OR**)
 - The smallest cube containing both implicants 作bit-wise OR就相當於兩個prime implicant的聯集，產出的結果就是一個足夠大的長方形將兩個小圓圈框起來
- The **(disjoint) sharp operation** can be used to compute the **complementation** 前面提到的sharp operation可以用來計算complement
→ 0去減掉其他東西即可

$$R = U \# (F^{ON} \cup F^{DC})$$

NCTU M S E D A Lab.



43

Cofactor (Restriction) 就是將指定的值代入

- Cofactor of f with respect to $x_i = 0$
 - $f_{x_i} = f_{x_i=0} = f(x_1, x_2, \dots, x_i=0, \dots, x_n)$
- Cofactor of f with respect to $x_i = 1$
 - $f_{x_i} = f_{x_i=1} = f(x_1, x_2, \dots, x_i=1, \dots, x_n)$
 - Example:
 $f(x, y, z) = xy + yz' + x'z'$
 $\rightarrow f_{x=0} = yz' + z' \quad f_{x=1} = y + yz'$
- Cofactor with respect to any cube
 - Example:
 $f(x, y, z, w) = xy + zw' + w'x'$
 $f_{x'y'} = f_{x=0, y=0} = zw' + w'$
 $f_{xy'} = f_{x=1, y=0} = zw'$

f_a 的意思就是 f 對 a 的cofactor
也可以一個 f 對應很多cube

NCTU M S E D A Lab.



44

Cofactor of Implicants

此處的f寫作alpha · 對應的cube稱為beta

- The cofactor of an implicant α w.r.t an implicant β is:
- $\alpha_\beta = \phi$ when α does not intersect β 當前prime implicant與beta沒有交集-->跳過

Otherwise, $\alpha_\beta = a_1 + b_1' a_2 + b_2' \dots a_n + b_n'$ 當前prime implicant與beta
有交集就套用這個公式

- Example: Given $f = a'b' + ab$, calculate f_a 現在要計算f對應a的cofactor

$$f = \begin{matrix} 10 & 10 & a'b' \\ 01 & 01 & ab \end{matrix} \quad \begin{matrix} c(a) = 01 & 11 & aX \\ c(a') = 10 & 00 & (aX) \end{matrix} \quad (\text{cube representation})$$

我們人在自己算的時候可以直接把a=1代入f · 但是也可以利用cube representation得到 f_a

The cofactor of the first implicant is void

— $a'b'$ intersect with a is empty

首先將f用cube representation表示 · 然後將f的每一項都與c(a)去作bit-wise AND檢查是否有交集(c(a)就是a=1, b=X)

檢查後發現a'b'與c(a) AND後的結果包含00 · 沒有交集
ab與c(a) AND後的結果沒有00-->有交集

The cofactor of the second implicant is 11 01

— $(01 01) + (10 00) = (11 01)$ 因此a'b'跳過(沒有交集) · ab則與c(a)套用上面紅色框框的公式級可得到 f_a

$$\rightarrow f_a = b$$



Shannon Expansion

- $f = x' \cdot f_{x=0} + x \cdot f_{x=1}$ Shannon expansion就是將f拆為x=0與x=1兩種狀況
可以再繼續分解下去
 $= x_i' \cdot y_j' \cdot f_{x_i'y_j'} + x_i \cdot y_j' \cdot f_{x_i'y_j'} + x_i' \cdot y_j \cdot f_{x_i'y_j} + x_i \cdot y_j \cdot f_{x_i'y_j}$

- Example:

$$f_x = y + zw'$$

$$f_{x'} = zw' + w'$$

$$f = x(y + zw') + x'(zw' + w')$$

- Decompose a function into two components, one for the subspace $x = 0$, the other for the subspace $x = 1$

$$f = x'f_{x'} + xf_x \quad \text{因此就可以套用前面求cofactor的方法來實作Shannon expansion}$$

- Allow a divide and conquer strategy on several problems
 - $f_{x'}$ and f_x do not depend on x and thus have one less variable



Consensus Operator

- Definition: $\forall x(f) = f_x \cdot f_{x'}$ 求出的 $x(f)$ 與 x 無關(不管 $x=0$ 或 $x=1$, $x(f)$ 都會是 1)
- $\forall x(f)$ evaluate f to be true for $x = 1$ and $x = 0$
- Represent the component that is independent of that variable
- Example:

$$f(x,y,z,w) = xy + zw' + w'x' \quad \begin{array}{l} x=0 \rightarrow zw' + w' = (z+1)w' = w' \\ x=1 \rightarrow y + zw' \end{array}$$

因此作 AND 的結果為 $w'(y+zw') = w'y + zw'$

$$f_x \cdot f_{x'} = zw' + w'y$$



Smoothing Operations

- Definition: $\exists x(f) = f_x + f_{x'}$ 跟上一頁很像 · 只是換成 OR
- $\exists x(f)$ evaluate f to be true when $x = 1$ or $x = 0$
- Example:

$$f(x,y,z,w) = xy + zw' + w'x' \quad \exists x(f) = f_x + f_{x'} = (zw' + w') + (zw' + y)$$



- $\frac{\partial f}{\partial x}$ is called Boolean difference of f with respect to x
- Definition: $\frac{\partial f}{\partial x} = f_x \oplus f_{\bar{x}}$ 因為是觀察針對某訊號的變化，其記號就是用偏微分表示
- f is sensitive to the value of x when $\frac{\partial f}{\partial x} = 1$
- Example: $f(x,y,z,w) = xy + zw' + w'x'$
 $f_{x'} = f(x=0,y,z,w) = zw' + w'$
 $f_x = f(x=1,y,z,w) = y + zw'$
 $f_{x'} \oplus f_x = (zw' + w') \oplus (y + zw')$



Outline

- Synthesis overview
- RTL synthesis
- Two-level logic optimization
 - Basic logic operations
 - Exact minimization
 - Heuristic methods
- Multi-level logic optimization
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power



The Quine-McCluskey Algorithm

- Theorem:[Quine,McCluskey] There exists a minimum cover for F that is prime
 - Need to look just at primes (reduces the search space)
- Classical methods: two-step process
 1. Generation of all prime implicants (of the union of the on-set and dc-set)
 2. Extraction of a minimum cover (covering problem)
- Exponential-time exact algorithm, huge amounts of memory!
- Other methods do not first enumerate all prime implicants; they use an implicit representation by means of ROBDDs.



Primary Implicant Generation (1/5)

		ab		a	
		cd			
c	d	00	01	11	10
		X	1	0	1
00	01	0	1	1	1
11		0	X	X	0
10		0	1	0	1

The table shows a Karnaugh map for four variables (a, b, c, d). The columns are labeled ab (00, 01, 11, 10) and the rows are labeled cd (00, 01, 11, 10). The values in the cells represent minterms: X, 1, 0, 1, 0, 1, X, X, 0, 1, 0, 1. The variable 'a' is indicated above the last two columns (11, 10), and 'b' is indicated below the last two rows (11, 10). The variables 'cd' and 'c' are indicated to the left of the first two rows (00, 01), and 'd' is indicated to the right of the last two columns (11, 10).



Primary Implicant Generation (2/5)

Implication Table	
Column I	
zero "1"	0000
one "1"	0100 1000
two "1"	0101 0110 1001 1010
three "1"	0111 1101
four "1"	1111



Primary Implicant Generation (3/5)

Implication Table		
Column I	Column II	
0000	0-00 -000	
0100		
1000	010- 01-0	
0101	100-	
0110	10-0	
1001		
1010	01-1 -101	
0111	011- 1-01	
1101		
1111	-111 11-1	

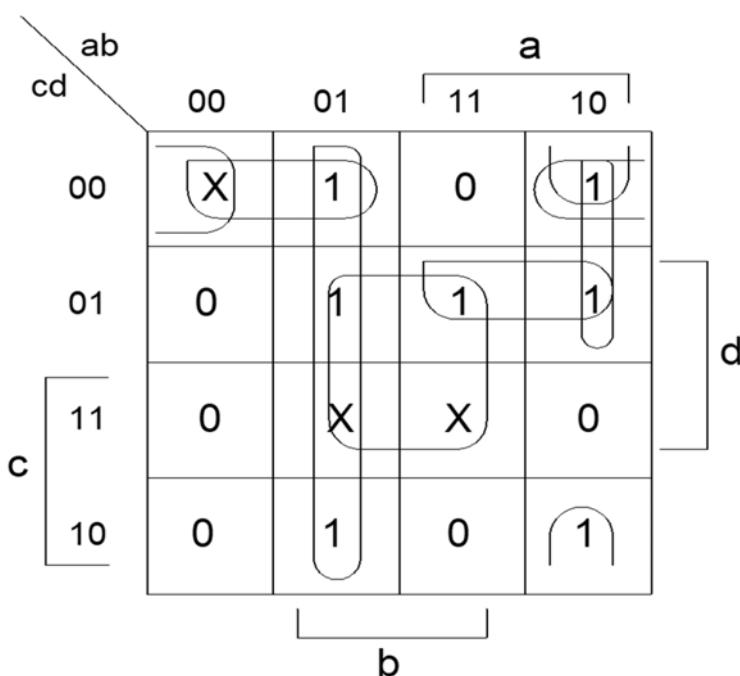


Primary Implicant Generation (4/5)

Implication Table		
Column I	Column II	Column III
0000	0-00 * -000 *	01-- *
0100		-1-1 *
1000	010- 01-0	
0101	100- *	
0110	10-0 *	
1001		
1010	01-1 -101	
0111	011-	
1101	1-01 *	
1111	-111 11-1	



Primary Implicant Generation (5/5)



Prime Implicants:

- 0-00 = $a'c'd'$
- 100- = $ab'c'$
- 1-01 = $ac'd$
- 1-1 = bd
- 000 = $b'c'd'$
- 10-0 = $ab'd'$
- 01-- = $a'b$



Column Covering (1/4)

	4	5	6	8	9	10	13
0,4 (0-00)	X						
0,8 (-000)				X			
8,9 (100-)			X	X			
8,10 (10-0)			X			X	
9,13 (1-01)				X			X
4,5,6,7 (01- -)	X	X	X				
5,7,13,15 (-1-1)		X				X	

rows = prime implicants

columns = ON-set elements

place an "X" if ON-set element
is covered by the prime implicant



Column Covering (2/4)

	4	5	6	8	9	10	13
0,4 (0-00)	X						
0,8 (-000)				X			
8,9 (100-)			X	X			
8,10 (10-0)				X		X	
9,13 (1-01)					X		X
4,5,6,7 (01- -)	X	X	X	X			
5,7,13,15 (-1-1)		X				X	

If column has a single X, then the
implicant associated with the row
is essential. It must appear in
minimum cover



Column Covering (3/4)

	4	5	6	8	9	10	13
0,4 (0-00)	✗						
0,8 (-000)				✗			
8,9 (100-)				✗	✗		
8,10 (10-0)				✗		✗	
9,13 (1-01)					✗		✗
4,5,6,7 (01- -)	✗	✗	✗				
5,7,13,15 (-1-1)		✗					✗

Eliminate all columns covered by essential primes



Column Covering (4/4)

	4	5	6	8	9	10	13
0,4 (0-00)	✗						
0,8 (-000)				✗			
8,9 (100-)				✗	✗		
8,10 (10-0)				✗		✗	
9,13 (1-01)					✗		✗
4,5,6,7 (01- -)	✗	✗	✗				
5,7,13,15 (-1-1)		✗					✗

Find minimum set of rows that cover the remaining columns

$$f = ab'd' + ac'd + a'b$$



Petrick's Method

- Solve the **satisfiability** problem of the following function

$$P = (P_1 + P_6)(P_6 + P_7)P_6(P_2 + P_3 + P_4)(P_3 + P_5)P_4(P_5 + P_7) = 1$$

	4	5	6	8	9	10	13
P1	0,4 (0-00)		X				
P2	0,8 (-000)			X			
P3	8,9 (100-)			X	X		
P4	8,10 (10-0)			X		X	
P5	9,13 (1-01)				X		X
P6	4,5,6,7 (01--)	X	X	X			
P7	5,7,13,15 (-1-1)		X			X	

- Each term represents a corresponding column
- Each column must be chosen at least once
- All columns must be covered



ROBDDs and Satisfiability

- A Boolean function is **satisfiable** if an assignment to its variables exists for which the function becomes '1'
- Any Boolean function whose ROBDD is unequal to '0' is satisfiable.
- Suppose that choosing a Boolean variable x_i to be '1' costs c_i . Then, the **minimum-cost satisfiability** problem asks to minimize: $\sum_{i=1}^n c_i \mu(x_i)$

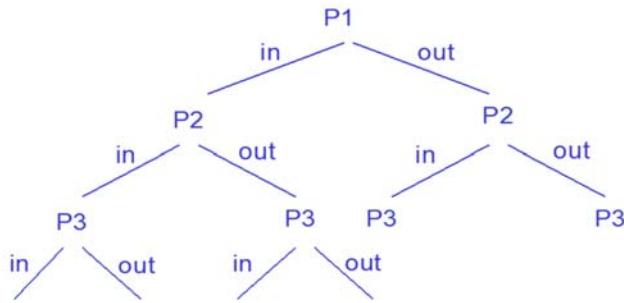
where $\mu(x_i) = 1$ when $x_i = '1'$ and $\mu(x_i) = 0$ when $x_i = '0'$.

- Solving minimum-cost satisfiability amounts to computing the shortest path in an ROBDD, which can be solved in linear time.
 - Weights: $w(v, \eta(v)) = c_i$, $w(v, \lambda(v)) = 0$, variable $x_i = \phi(v)$.



Brute Force Technique

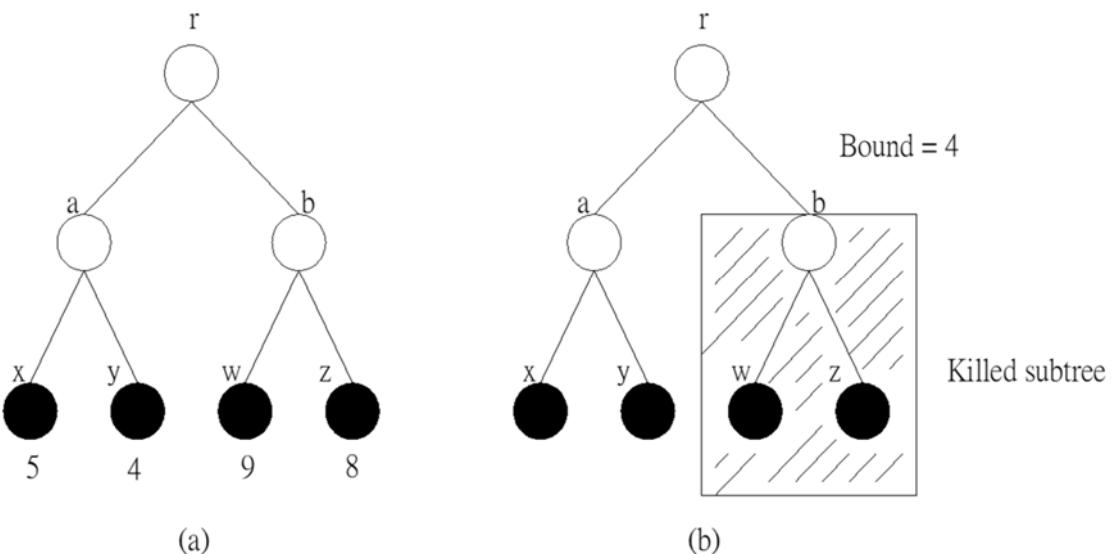
- Brute force technique: Consider all possible elements



- Complete branching tree has $2^{|P_1|}$ leaves!!
 - Need to prune it
- Complexity reduction
 - Essential primes can be included right away
 - If there is a row with a singleton “1” for the column
 - Keep track of best solution seen so far
 - Classic **branch and bound**



Branch and Bound Algorithm



Outline

- Synthesis overview
- RTL synthesis
- Two-level logic optimization
 - Basic logic operations
 - Exact minimization
 - **Heuristic methods**
- Multi-level logic optimization
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power

NCTU M SEDA Lab.



65

Why Heuristic Optimization ?

- Generation of **all** prime implicants is impractical
 - The number of prime implicants for functions with n variables is in the order of $3^n/n$
- Finding an **exact** minimum cover is NP-hard
 - Cannot be finished in polynomial time
- Heuristic method: provide irredundant covers with reasonably small cardinality
 - Fast and applicable to most functions
- Key idea: avoid generation of all prime implicants
 - Given initial cover
 - Make it prime
 - Make it irredundant
- Iterative improvement by modifying the implicants

NCTU M SEDA Lab.



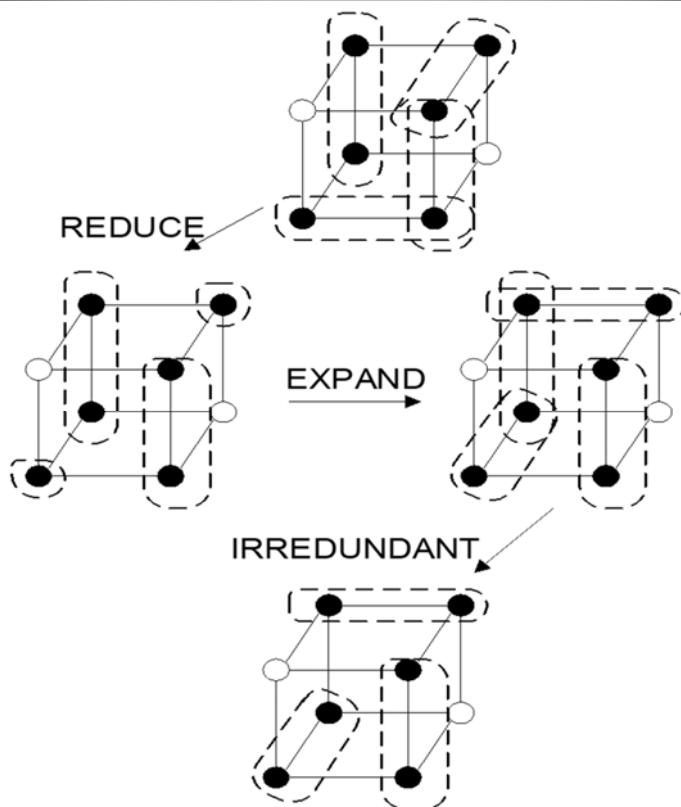
66

Logic Minimizer -- ESPRESSO

- “ESPRESSO” developed by UC Berkeley
 - The kernel of existing synthesis tools
- EXPAND:
 - A minterm of $ON(f)$ is selected, and expanded until it becomes a prime implicant
 - Make implicants prime
- IRREDUNDANT COVER:
 - The prime implicant is put in the final cover, and all minterms covered by this prime implicant are removed
 - Make cover irredundant
- REDUCE:
 - Reduce size of each implicant while preserving cover
- Iteratively find alternative covers
 - Repeat the 3 steps to find the solutions with lower costs



ESPRESSO - Illustrated



Pseudo Code of ESPRESSO

```
espresso (F, D) /* F = ON_SET , D = DC_SET*/
{
    R = complement (F + D); /* R = OFF_SET */
    F = expand (F, R); /* initial expansion */
    F = irredundant_cover (F, D); /* initial irredundant cover */
    E = essential_primes (F, D); /* extract essential primes */
    C = F - E;
    D = D + E;
    repeat {
        C = reduce (C, D);
        C = expand (C, R);
        C = irredundant_cover (C, D);
    } until (C unchanged);
    return C + E ;
}
```



Expand (1/3)

- Increase the size of each implicant
 - Implicants of smaller size can be covered and deleted
 - Maximally expanded implicants are primes
 - Raising one (or more) of its 0s to 1
- Validity checking
 - Checking for an intersection of the expanded implicant with F^{OFF}
- Two factors affect the quality and the efficiency of the algorithm
 - The order in which the implicants are selected
 - The order in which the 0 entries are raised to 1



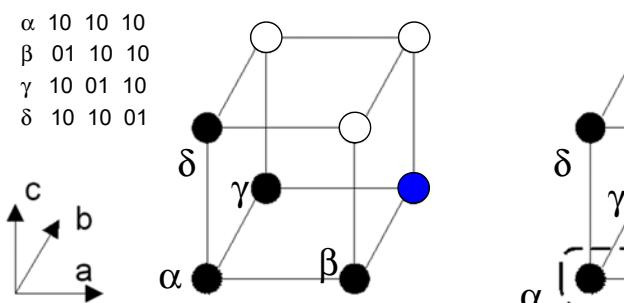
Expand (2/3)

- Heuristic on the order of implicants
 - Compute column count vector (number of '1' in each column)
 - The weight of each cube is the inner product of itself and the column count vector
 - Sort implicants in **ascending** order of weight
 - Low weight correlates to having few 1s in the columns
 - **Expand first those cubes that are unlikely to be covered**
- Ex: $f = a'b'c' + ab'c' + a'bc' + a'b'c$; don't care : abc'

F^{ON} :	F^{DC} :	Column count vector = [313131] ^T
choose lowest weight →	10 10 10 01 10 10	Weight of the implicants = (9,7,7,7)
column count →	10 01 10 +) 10 10 01 31 31 31	Ex: 2 nd implicant of F^{ON}
	01 11 01 11 01 01	01 10 10 x) 31 31 31 01 30 30
		2 nd weight = 0+1+3+0+3+0 = 7

EDA Lab. NCTU M SEDA Lab. 71

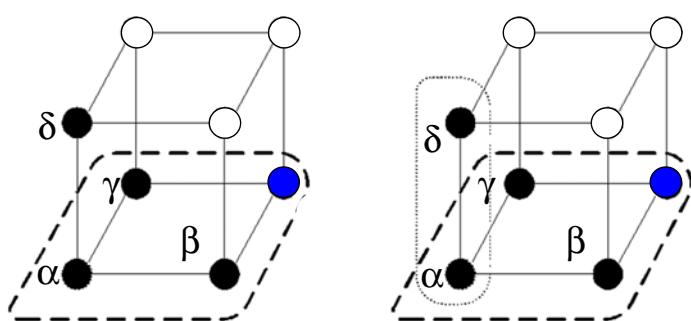
Expand (3/3)



01 10 10 → 11 10 10 **OK**
 11 10 10 → 11 11 10 **OK**
 11 11 10 → 11 11 11 **X**

F updated to :

11	11	10
10	10	01



10 10 01 → 11 10 01 **X**
 10 10 01 → 10 11 01 **X**
 10 10 01 → 10 10 11 **OK**

Expanded cover :

11	11	10
10	10	11

Reduce (1/2)

- Decrease the size of each implicant of a given cover F
→ successive expansion may lead to smaller cover
 - A reduced implicant is valid when, along with the remaining implicants, it still covers the function
 - The reduced cover has the same cardinality as the original one
- Let $\alpha \in F$ be an implicant and $Q = F \cup F^{DC} - \{\alpha\}$
 - The maximally reduced cube is
$$\alpha'' = \alpha \cap \text{supercube}(Q_\alpha') \quad // \text{the part not covered by other implicants}$$
- $\alpha \# Q = \alpha \cap Q'$ can yield a set of cubes
$$\begin{aligned}\alpha'' &= \alpha \cap \text{supercube}(Q') \\ &= \alpha \cap \text{supercube}((\alpha \cap Q_\alpha') \cup (\alpha' \cap Q_{\alpha'}')) \\ &= \alpha \cap \text{supercube}(Q_\alpha')\end{aligned}$$



Reduce (2/2)

- Sorting the implicants
 - Weight the implicants as for the Expand operator
 - Sort implicants in **descending** order of weight
 - First process those that overlap many other implicants
 - Lower as many * as possible to 1 or 0
- Replacing each implicant by the maximally reduced one

$F: \alpha \ 11 \ 11 \ 10$
 $\beta \ 10 \ 10 \ 11$

column count vector = [212121]^T

weight vector = [8,7]

Reduce α first → fail

Reduce β :

$Q = F \cup F^{DC} - \{\beta\} = \{\alpha, \beta\} - \{\beta\}$
 $= 11 \ 11 \ 10 \quad // \text{only } \alpha \text{ is left}$

$Q' = 11 \ 11 \ 01, Q_\beta' = Q'$

$\text{supercube}(Q_\beta') = Q' \quad // \text{not in } Q$

$\beta'' = \beta \cap Q' = 10 \ 10 \ 01$

Reduced cover is

11 11 10

10 10 01

$\beta'' \rightarrow \text{In } \beta \text{ but not in } Q$



Outline

- Synthesis overview
- RTL synthesis
- Two-level logic optimization
- Multi-level logic optimization
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power



Multi-Level Logic Optimization

- Level: maximum number of gates cascaded in series between the inputs and outputs of a network
 - Can be considered as an indication for worst-case delay
 - Assume all variables and their complements are available
- Two-level networks have the least depth, not least area
 - It's possible to further reduce the number of gates by increasing the logic levels and reusing existing logic gates
 - Common factors or kernel extraction
 - Common expression resubstitution
- Example:

$$\begin{aligned} f_1 &= abcd + abce + \bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}\bar{c}\bar{d} + \\ &\quad \bar{a}\bar{c} + cdf + \bar{a}\bar{b}\bar{c}\bar{d}\bar{e} + \bar{a}\bar{b}\bar{c}\bar{d}\bar{f} \\ f_2 &= bdg + \bar{b}dfg + \bar{b}\bar{d}g + b\bar{d}eg \end{aligned}$$

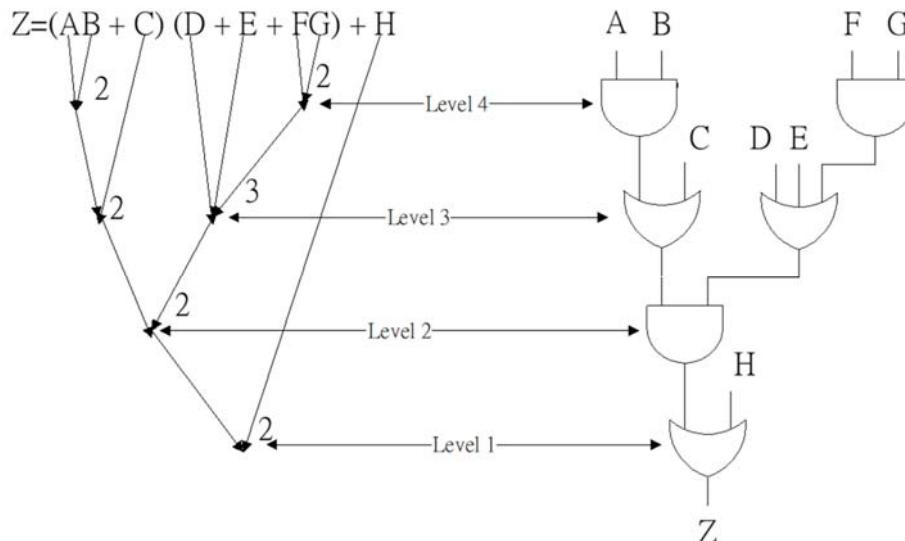


$$\begin{aligned} f_1 &= c(\bar{a} + x) + \bar{a}\bar{c}\bar{x} \\ f_2 &= gx \\ x &= d(b + f) + \bar{d}(\bar{b} + e) \end{aligned}$$



Multi-Level Logic

- Multi-level logic:
 - A set of logic equations with no cyclic dependencies
- Example: $Z = (AB + C)(D + E + FG) + H$
 - 4-level, 6 gates, 13 gate inputs



NCTU M SEDA Lab.



77

Multi-Level v.s. Two-Level

- Two-level:
 - Often used in control logic design
 - Only x_1x_4 shared
 - Sharing restricted to common cube
 - Multi-level:
 - Datapath or control logic design
 - Can share $x_2 + x_3$ between the two expressions
 - Can use complex gates
- $$f_1 = x_1x_2 + x_1x_3 + x_1x_4$$
- $$f_2 = x_1'x_2 + x_1'x_3 + x_1x_4$$
- $$g_1 = x_2 + x_3$$
- $$g_2 = x_1x_4$$
- $$f_1 = x_1y_1 + y_2$$
- $$f_2 = x_1'y_1 + y_2$$
- (y_i is the output of gate g_i)

NCTU M SEDA Lab.



78

Factored Forms (1/2)

- A *factored form* is defined recursively by the following rules:
 - A literal is a factored form
 - A sum of two factored form is a factored form
 - A product of two factored forms is a factor form
 - A factored form describes an implementation of the function as a complex gate
 - Any depth of sum-of-product

a'

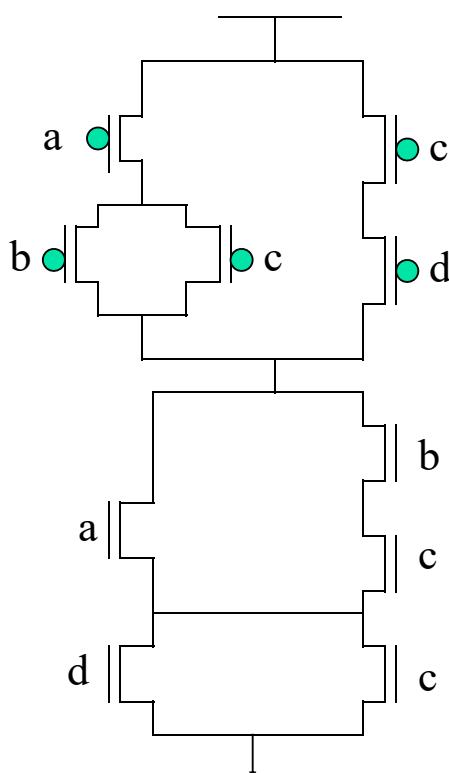
ab'c

$$ab + c'd$$

$$(a + b)(c + a' + de) + f$$



Factored Forms (2/2)



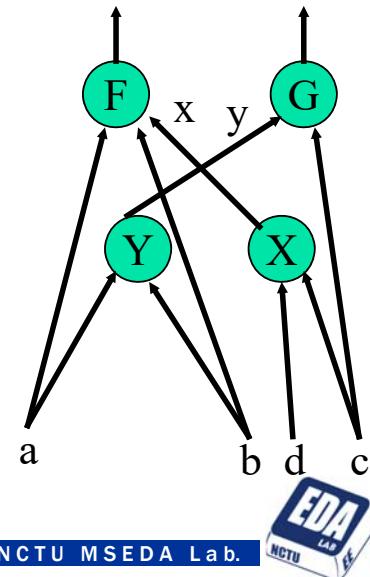
- A CMOS complex gate implementing
 $f = ((a + bc)(c + d))'$
 $2 * \text{literal count} = \# \text{ transistors}$
 - Adv:
 - Nature multi-level representation
 - Good estimate of the complexity of function
 - Represent both the function and its complement
 - Disadv:
 - More difficult to manipulate than two-level form
 - Lack of the notion of optimality



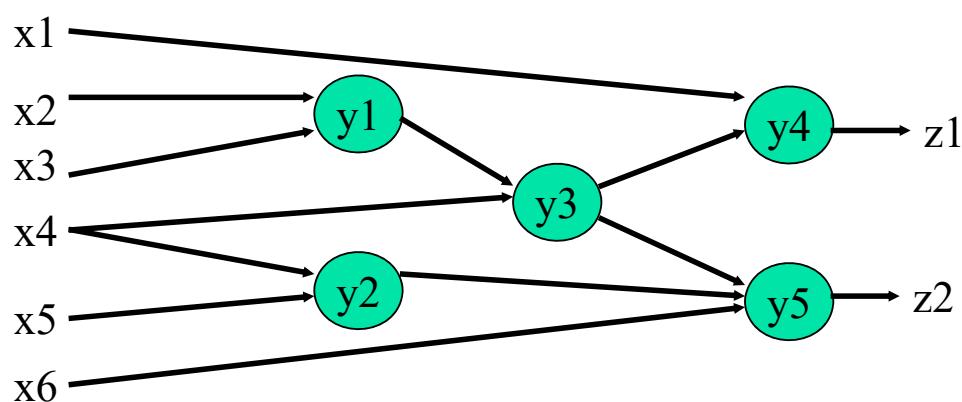
Boolean Network

- Directed acyclic graph (DAG)
- Each source node is a primary input
- Each sink node is a primary output
- Each internal node represents an equation
- Arcs represent variable dependencies

fanin of y : a, b
fanout of x : F



Boolean Network : An Example



$$y_1 = f_1(x_2, x_3) = x_2' + x_3'$$

$$y_2 = f_2(x_4, x_5) = x_4' + x_5'$$

$$y_3 = f_3(x_4, y_1) = x_4'y_1'$$

$$y_4 = f_4(x_1, y_3) = x_1 + y_3'$$

$$y_5 = f_5(x_6, y_2, y_3) = x_6y_2 + x_6'y_3'$$



Multi-Level Logic Optimization

- Technology independent
- Decomposition/Restructuring
 - Algebraic
 - Functional
- Node optimization
 - Two-level logic optimization techniques are used



Decomposition / Restructuring

- Goal : given initial network, find best network
- Two problems:
 - Find good **common subfunctions**
 - How to perform **division**
- Example:

$$\begin{aligned}f_1 &= abcd + abce + ab'cd' + ab'c'd' + a'c + cdf + abc'd'e' + ab'c'df' \\f_2 &= bdg + b'dfg + b'd'g + bd'eg\end{aligned}$$

minimize (in sum-of-products form):

$$\begin{aligned}f_1 &= bcd + bce + b'd' + b'f + a'c + abc'd'e' + ab'c'df' \\f_2 &= bdg + dfg + b'd'g + d'eg\end{aligned}$$

decompose:

$$\begin{aligned}f_1 &= c(a' + x) + ac'x' \quad x = d(b + d) + d'(b' + e) \\f_2 &= gx\end{aligned}$$



Basic Operations (1/2)

1. decomposition

(single function)

$$f = abc + abd + a'c'd' + b'c'd'$$



$$f = xy + x'y'$$

$$x = ab$$

$$y = c + d$$

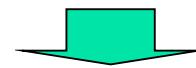
2. extraction

(multiple functions)

$$f = (az + bz')cd + e$$

$$g = (az + bz')e'$$

$$h = cde$$



$$f = xy + e$$

$$g = xe'$$

$$h = ye$$

$$x = az + bz'$$

$$y = cd$$

NCTU M SEDA Lab.



85

Basic Operations (2/2)

3. factoring

(series-parallel decomposition)

$$f = ac + ad + bc + bd + e$$



$$f = (a + b)(c + d) + e$$

4. substitution

(with complement)

$$g = a + b$$

$$f = a + bc + b'c'$$



$$f = g(a + c) + g'c'$$

5. elimination

$$f = ga + g'b$$

$$g = c + d$$



$$f = ac + ad + bc'd'$$

$$g = c + d$$



NCTU M SEDA Lab.



86

Division

- Division: p is a Boolean divisor of f if $q \neq \phi$ and r exist such that $f = pq + r$
 - p is said to be a factor of f if in addition $r = \phi$:
$$f = pq$$
 - q is called the **quotient**
 - r is called the **remainder**
 - q and r are **not unique**
- **Weak division:** the unique algebraic division such that r has as few cubes as possible
 - The quotient q resulting from weak division is denoted by f / p (it is **unique**)



Weak Division Algorithm (1/2)

Weak_div(f, p):

$U =$ Set $\{u_j\}$ of cubes in f with literals not in p deleted

$V =$ Set $\{v_j\}$ of cubes in f with literals in p deleted

/* note that $u_j v_j$ is the j -th cube of f */

$V^i = \{v_j \in V : u_j = p_j\}$

$q = \cap V^i$

$r = f - pq$

return(q, r)



Weak Division Algorithm (2/2)

- Example

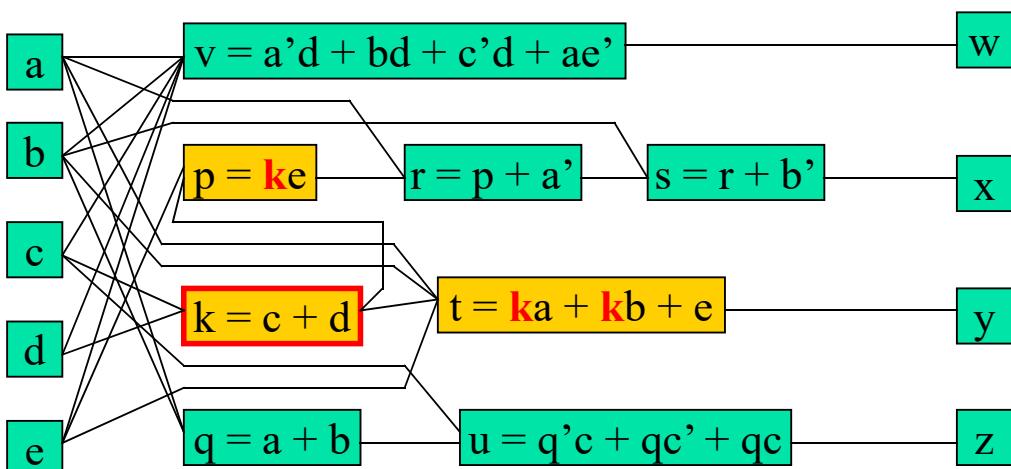
common expressions

$$\begin{aligned}
 f &= acg + adg + ae + bc + bd + be + a'b \\
 p &= ag + b \\
 U &= ag + ag + a + b + b + b + b \\
 V &= c + d + e + c + d + e + a' \\
 V^{ag} &= c + d \\
 V^b &= c + d + e + a' \\
 q &= c + d = f/p
 \end{aligned}$$



Algebraic Substitution (1/3)

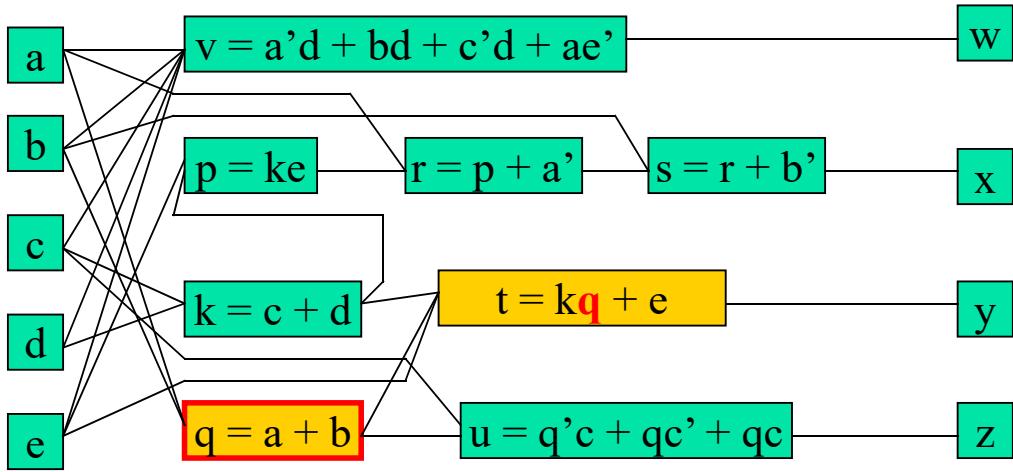
- Idea: An existing node in a network may be a useful divisor in another node.



$$f_t = ka + kb + e \quad f_q = a + b$$



Algebraic Substitution (2/3)



$$f_t = kq + e$$



Algebraic Substitution (3/3)

- Consist of the process of dividing the function f_i at node i in the network by the function f_j (or by f_j') pairwise
- During substitution, f_i is transformed into

$$f_i = (f_i/f_j)y_j + (f_i/f_j')y_j' + r$$
 if f_i/f_j and/or f_i/f_j' are not null
- No need to try all pairs. The cases where f_j is not an algebraic divisor of f_i can be excluded
 - f_j contains a literal not in f_i
 - f_j contains more terms than f_i
 - for any literal, the count in f_j exceeds that in f_i
 - f_i is f_j 's transitive fanin (cycle)



Algebraic Divisor

- Example:

$$X = (a + b + c)de + f$$

$$Y = (b + c + d)g + aef$$

$$Z = aeg + bc$$

- Single-cube divisor: ae
- Multiple-cube divisor: b + c
- Extraction of **common sub-expression** is a global area optimization effort



Kernels and Kernel Intersections

- An expression is **cube-free** if no cube divides the expression evenly
 - e.g., $ab + c$ is cube-free; $ab + ac$ and abc are not cube-free
 - A cube-free expression must have more than one cube
- The **primary divisors** of an expression f are the set of expressions
$$D(f) = \{f/c \mid c \text{ is a cube}\}$$
- The **kernels** of an expression f are the set of expressions
$$K(f) = \{g \mid g \in D(f) \text{ and } g \text{ is cube free}\}$$



Co-Kernels

- A cube c used to obtain the kernel $k = f/c$ is called a **co-kernel** of k
 - $C(f)$ is used to denote the set of co-kernels of f
 - Example

$$\begin{aligned} x &= \text{adf} + \text{aef} + \text{bdf} + \text{bef} + \text{cdf} + \text{cef} + g \\ &= (a + b + c)(d + e)f + g \end{aligned}$$

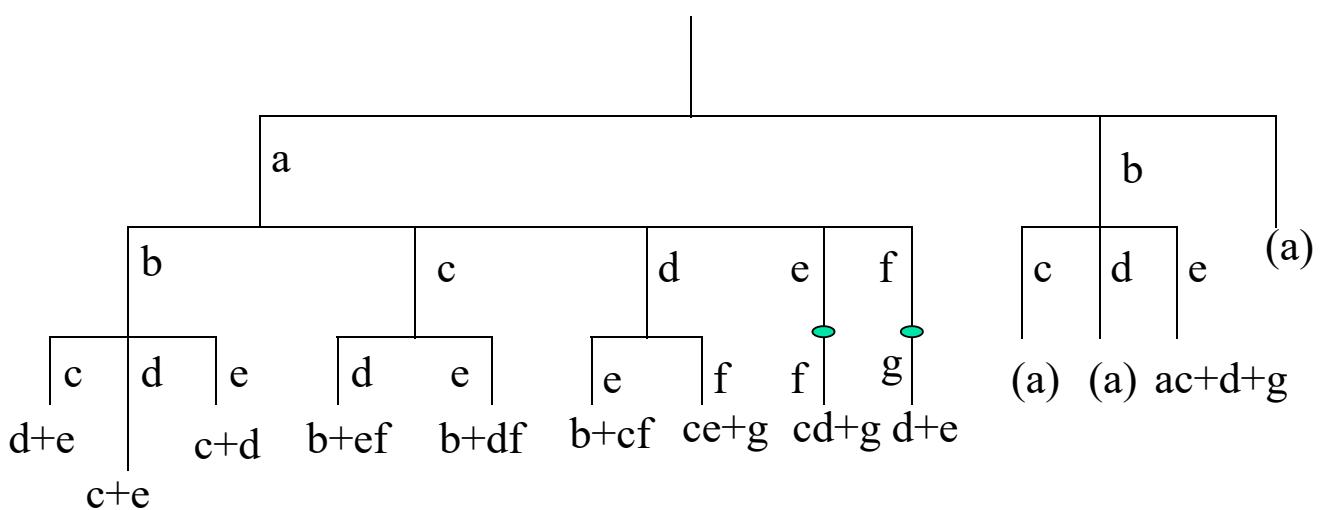
Kernel	Co-kernel
$a + b + c$	df, ef
$d + e$	af, bf, cf
$(a + b + c)(d + e)f + g$	1

- Kernels and co-kernels can help to find common divisors between expressions



Kerneling Illustrated

$$abcd + abce + adfg + ae fg + adbe + acdef + beg$$



Cube-Literal Matrix & Rectangles (1/2)

- Cube-literal matrix
 - Each matrix element indicates if this literal appears in the cube
- Ex: $f = x_1x_2x_3x_4x_7 + x_1x_2x_3x_4x_8 + x_1x_2x_3x_5 + x_1x_2x_3x_6 + x_1x_2x_9$

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9
$x_1x_2x_3x_4x_7$	1	1	1	1	0	0	1	0	0
$x_1x_2x_3x_4x_8$	1	1	1	1	0	0	0	1	0
$x_1x_2x_3x_5$	1	1	1	0	1	0	0	0	0
$x_1x_2x_3x_6$	1	1	1	0	0	1	0	0	0
$x_1x_2x_9$	1	1	0	0	0	0	0	0	1



Cube-Literal Matrix & Rectangles (2/2)

- A **rectangle** (R, C) of a matrix A is a subset of rows R and columns C such that

$$A_{ij} = 1 \forall i \in R, j \in C$$

- Rows and columns need not be continuous

- A **prime rectangle** is a rectangle not contained in any other rectangle
 - A prime rectangle indicates a co-kernel kernel pair

- Example:

$$R = \{\{1, 2, 3, 4\}, \{1, 2, 3\}\}$$

- co-kernel: $x_1x_2x_3$

- kernel: $x_4x_7 + x_4x_8 + x_5 + x_6$

	X_1	X_2	X_3	X_4
$x_1x_2x_3x_4x_7$	1	1	1	1
$x_1x_2x_3x_4x_8$	1	1	1	1
$x_1x_2x_3x_5$	1	1	1	0
$x_1x_2x_3x_6$	1	1	1	0
$x_1x_2x_9$	1	1	0	0



Rectangles and Logic Synthesis

- Kernels \Leftrightarrow prime rectangles of the cube-literal matrix
- Optimum selection of kernels \Leftrightarrow rectangle covering
 - Kernel intersection \Leftrightarrow finding rectangles
- Ex: single cube extraction

$$F = abc + abd + eg$$

$$G = abfg$$

$$H = bd + ef$$

$$(\{1,2,4\}, \{1,2\}) \Leftrightarrow ab$$

$$(\{2,5\}, \{2,4\}) \Leftrightarrow bd$$

$$F = Xc + XY + eg, \quad X = ab$$

$$G = Xfg, \quad Y = bd$$

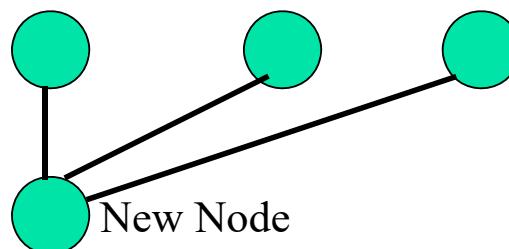
$$H = Y + ef$$

	a 1	b 2	c 3	d 4	e 5	f 6	g 7
abc	1	1	1	0	0	0	0
abd	2	1	1	0	1	0	0
eg	3	0	0	0	0	1	0
abfg	4	1	1	0	0	0	1
bd	5	0	1	0	1	0	0
ef	6	0	0	0	0	1	1



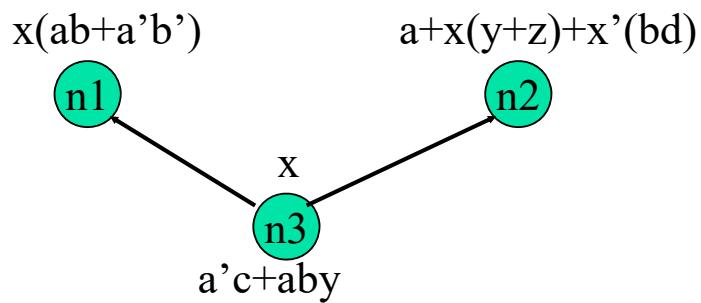
Kernel Extraction (1/2)

- 1.Find all kernels of all functions
- 2.Choose one with best “value”
- 3.Create new node with this as function
- 4.Algebraically substitute new node everywhere
- Repeat 1,2,3,4 until best value \leq threshold



Kernel Extraction (2/2)

- After



- Before

$n1$	$n2$
$(a'c+aby)(ab+a'b')$	$a+(a'c+aby)(y+z)+bd(a+c')(a'+b'+y')$

Literals after = $5+7+5=17$ Literals before = $9+15 = \underline{24}$
 before – after = value = 7



Example – Decomposition (1/2)

Original: $f_1 = ab(c(d + e) + f + g) + h$ (literal = $8+8=16$)
 $f_2 = ai(c(d + e) + f + j) + k$

Kernel extraction: (literal = $2+7+7=16$)

$$K^0(f_1) = K^0(f_2) = \{d + e\} \rightarrow f_1 = ab(cl + f + g) + h$$

$$l = d + e \qquad \qquad f_2 = ai(cl + f + j) + k$$

Kernel extraction: (literal = $2+3+5+5=15$)

$$K^0(f_1) = \{cl + f + g\} \rightarrow m = cl + f$$

$$K^0(f_2) = \{cl + f + j\} \rightarrow f_1 = ab(m + g) + h$$

$$K^0(f_1) \cap K^0(f_2) = cl + f \qquad \qquad f_2 = ai(m + j) + k$$

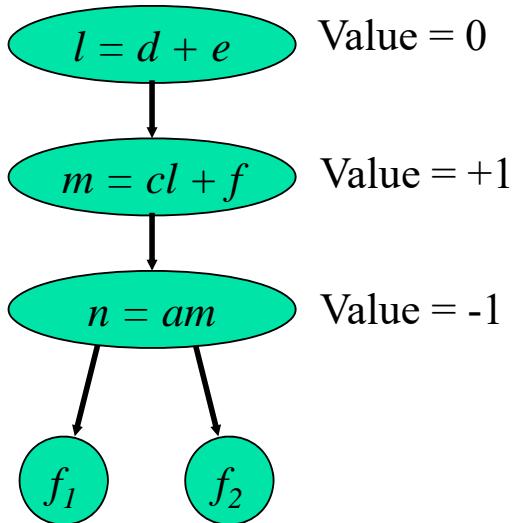
Cube extraction: (literal = $2+3+2+5+5=17$)

$$n = am \rightarrow f_1 = b(n + ag) + h$$

$$f_2 = i(n + aj) + k$$



Example – Decomposition (2/2)



- Eliminate -1

$$n = a(c(d + e) + f)$$

$$f_1 = b(n + ag) + h$$

$$f_2 = i(n + aj) + k$$

merge 3 levels
together

