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

Outline

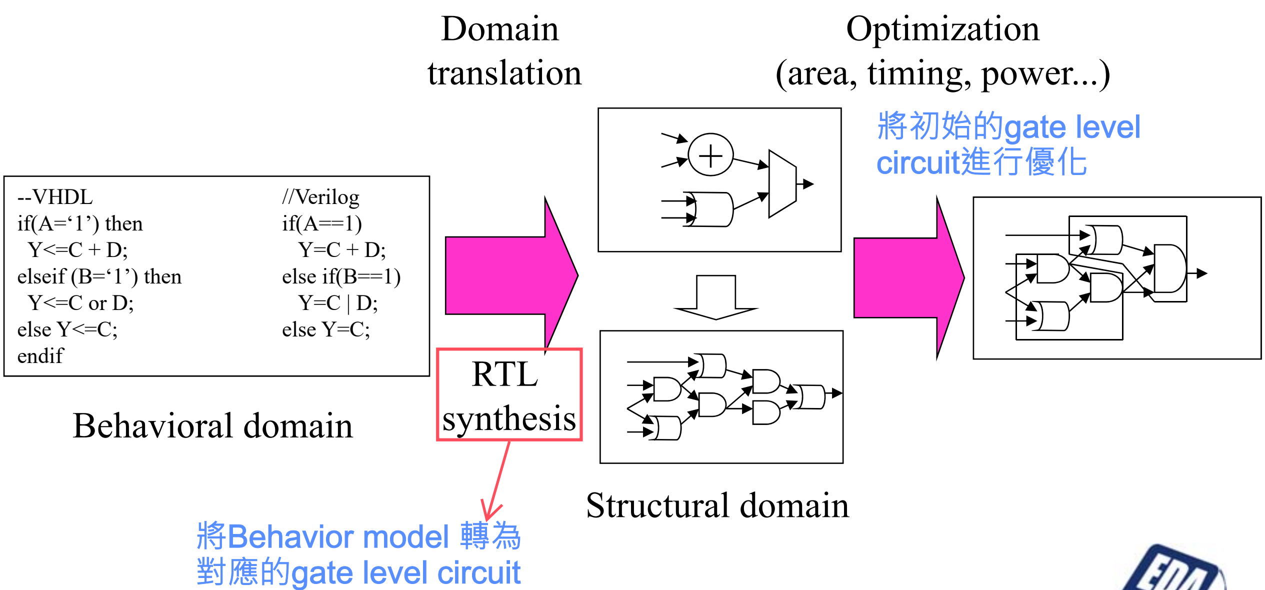
- Synthesis overview
- RTL synthesis
- Two-level logic optimization
- Multi-level logic optimization
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power

HDL Synthesis

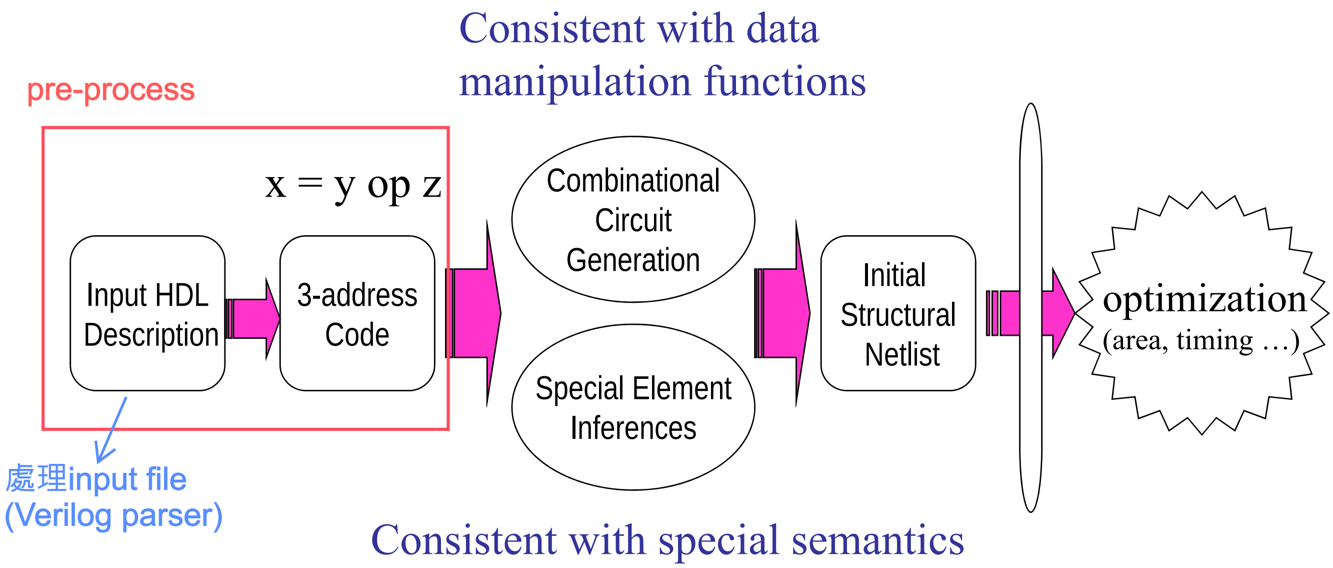
- **Logic synthesis** programs transform Boolean expressions or register-transfer level (RTL) description (in Verilog/VHDL/C) into logic gate networks (netlist) in a particular library.
- Advantages
  - Reduce time to generate netlists
  - Easier to retarget designs from one technology to another
  -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

Synthesis Procedure

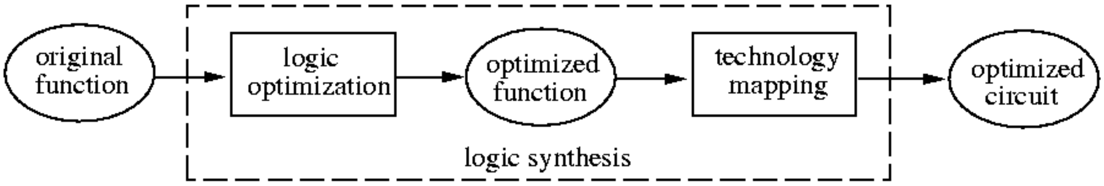
Synthesis = Domain Translation + Optimization



Domain Translation



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

Technology-Independent Logic Optimization

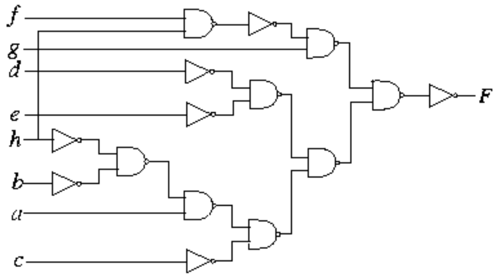
- **Two-level:** minimize the # of product terms.
  - $F = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2 x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 \Rightarrow F = \bar{x}_2 + x_1\bar{x}_3$.
- **Multi-level:** minimize the #'s of literals, variables.
  - E.g., equations are optimized using a smaller number of literals.

尋找是否有元件可以共用

$t1 = a + b\ c;$
$t2 = d + e;$
$t3 = a\ b + d;$
$t4 = t1\ t2 + f\,g;$
$t5 = t4\ h + t2\ t3;$
$F = t5';$

$\xrightarrow{\text{logic optimization}}$

$t1 = d + e;$
$t2 = b + h;$
$t3 = a\ t2 + c;$
$t4 = t1\ t3 + f\,g\,h;$



subject graph for the optimized equations

- Methods/CAD tools: Quine-McCluskey method (exponential-time exact algorithm), Espresso (heuristics for two-level logic), MIS (heuristics for multi-level logic), Synopsys, etc.

Espresso的團隊就是新思的創辦團隊

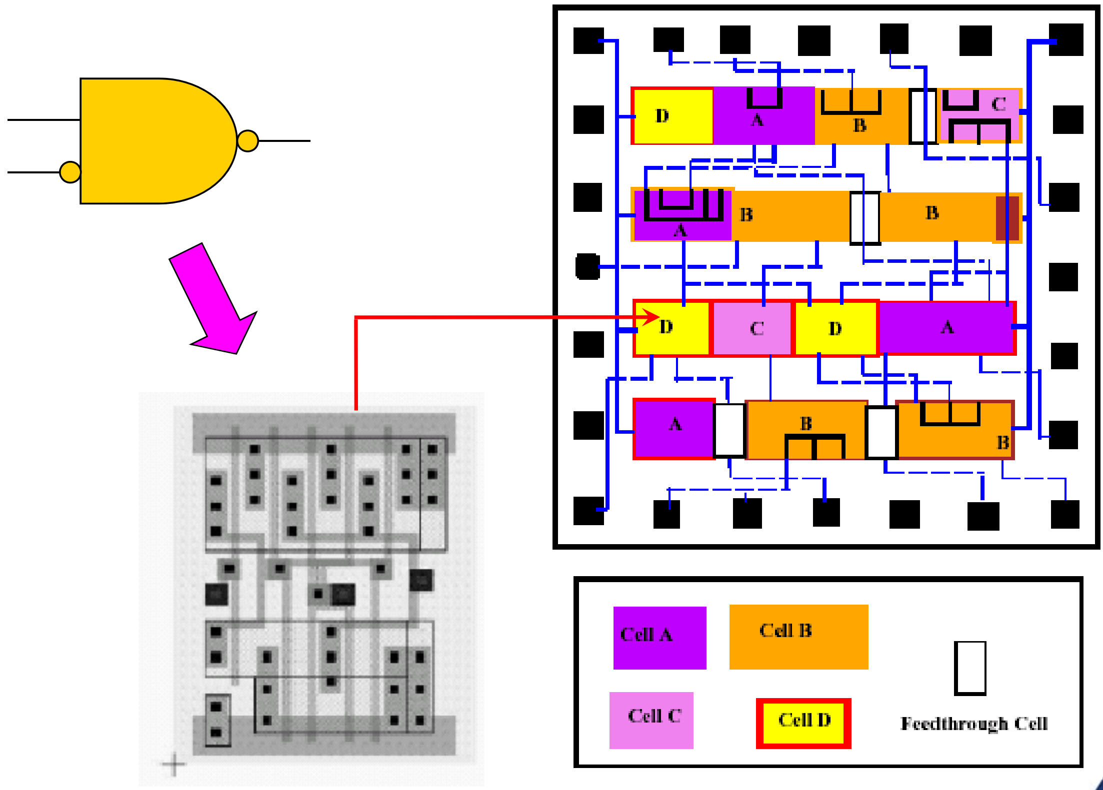
Technology Mapping

- 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

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

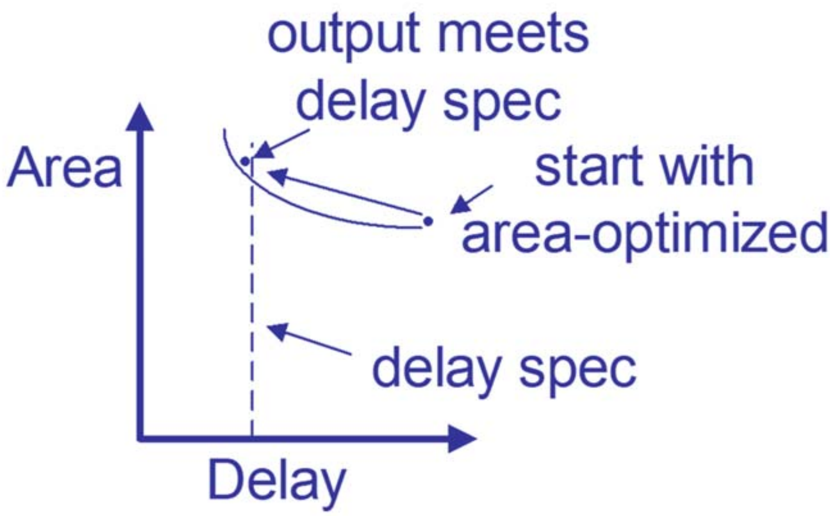
- Usage:
  - Technology mapping after technology independent logic optimization
  - Technology translation

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

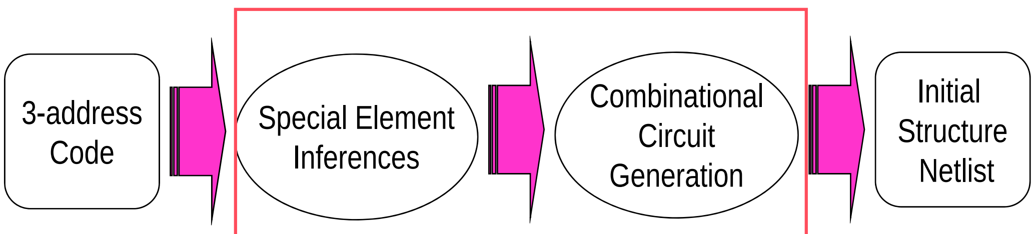


Outline

- 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

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)



x = y op z

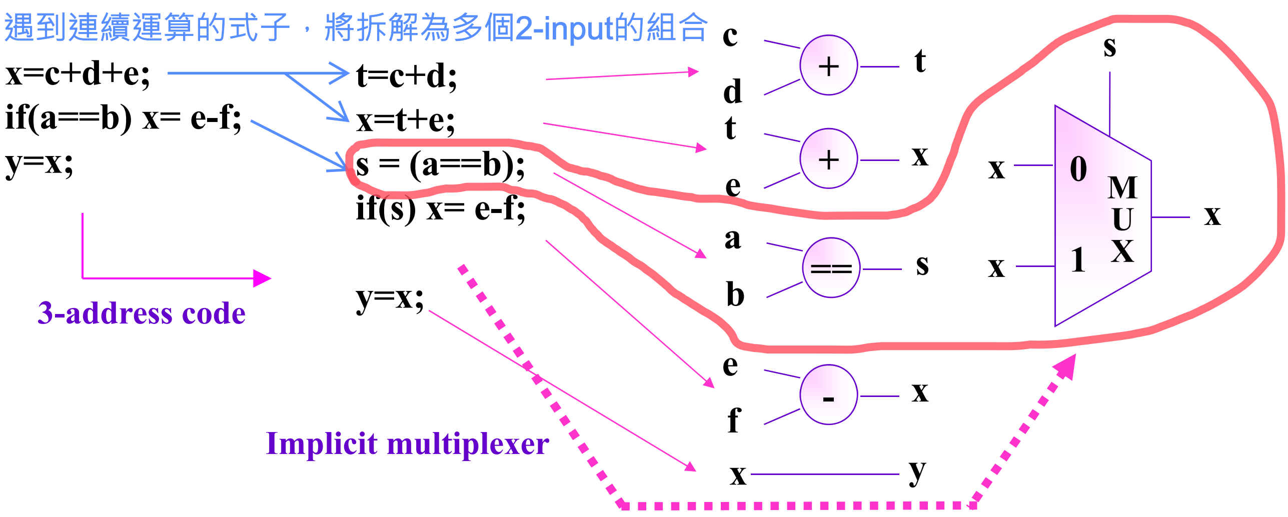
可以互換，不一定是照這順序

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 op z in general form
  - Function unit op with inputs y and z and output x



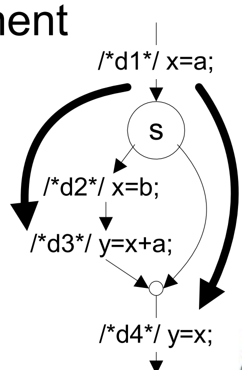
Interconnection Binding

- 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
  - d1 can reach d4 but cannot reach d3
    - d1 is killed by d2 before reaching d3 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;
```



標記*的變數代表當前cycle會用到

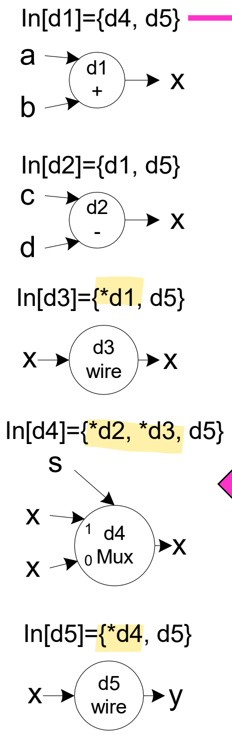
Combinational Circuit Generation: An Example

```
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

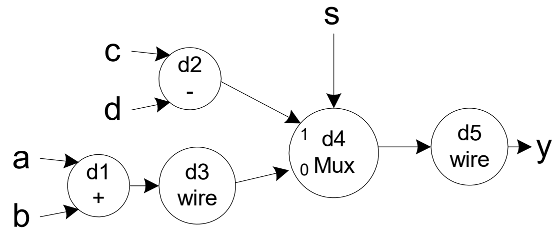
```
always @ (x or a or b or c or d or s)
begin
/*d1*/  x = a + b;
/*d2*/  if ( s ) x = c – d;
/*d3*/  else x = x;
/*d4*/  x = s mux x;
/*d5*/  y = x;
end
```

Modified 3-address code

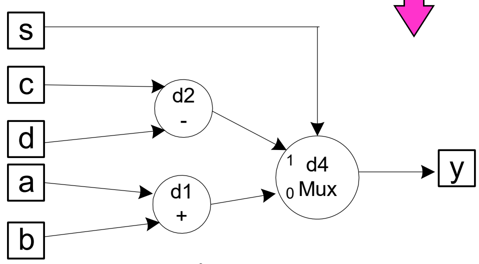


Functional unit allocation

computed by control/ data flow analysis



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

```
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;
```

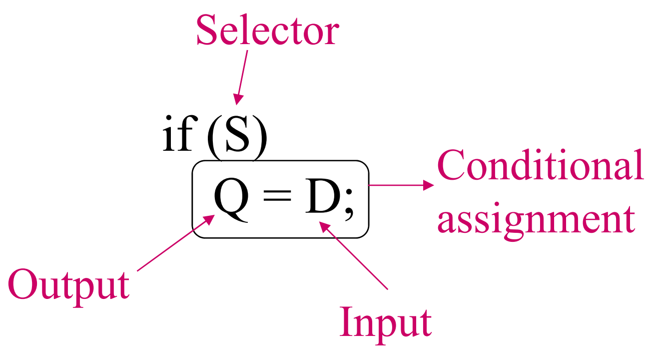
→ Infer latch for Q

```
always@(S or A or B)
begin
Q = A;
if(S) Q = B;
end
```

→ Do not infer latch for Q

Terminology (1/2)

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

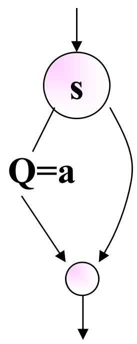


Terminology (2/2)

- 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

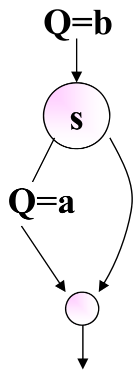
if(s) Q=a;

Q has no branch for the false value of the selector s



Q=b;
if(s) Q=a;

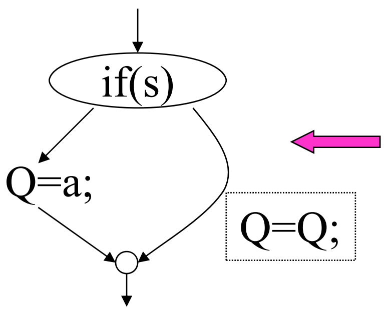
Q has a branch for the false value of the selector s



Rules of Latch Inference (1/2)

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

always@(s or a)
if(s) Q=a;

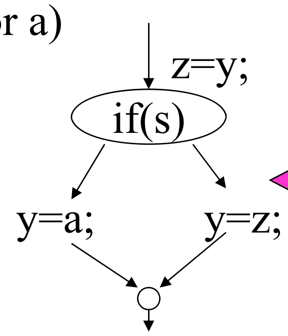


Q depends on its previous value at this branch

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;

Terminology

- Clocked statement: edge-triggered always statement
  - Simple clocked statement

    e.g., always @ (**posedge** clock)
  - Complex clocked statement

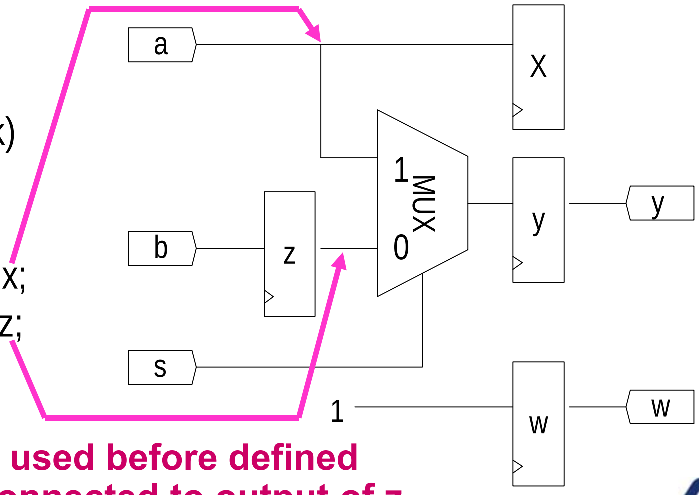
    e.g., always @ (**posedge** clock or **posedge** reset)
- Flip-flop inference must be conducted only when synthesizing the **clocked statements**

Infer FF for Simple Clocked Statements (1/2)

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

```
input a, b, s, clk;
output y, w;
reg x, w, y, z;
always @ (posedge clk)
begin
/* d1 */      x = a;
/* d2 */      if ( s )   y = x;
/* d3 */      else      y = z;
/* d4 */      z = b;
/* d5 */      w = 1'b1;
end
```

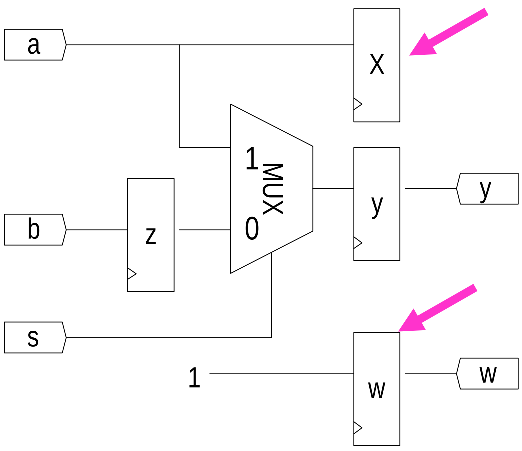
used after defined
connected to input of x

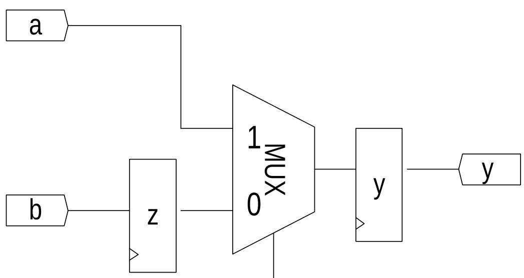


used before defined
connected to output of z

Infer FF for Simple Clocked Statements (2/2)

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





Infer FF for Complex Clocked Statements

- The edge-triggered signal not used in the following operations is chosen as the clock signal
- 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.)

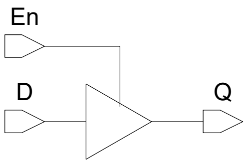
always @ (**posedge** clock or **posedge** reset or **negedge** set)

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

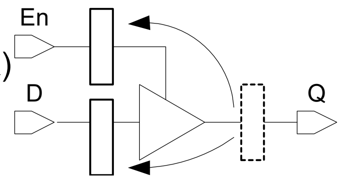
Typical Tri-State Buffer Inference (1/2)

- 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

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




```
reg Q;
always @ (posedge clk)
if(En) Q = D;
else Q = 1'bz;
```

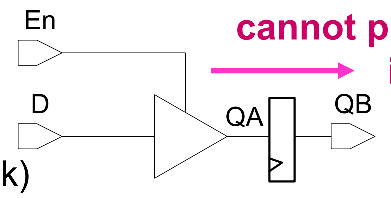


Typical Tri-State Buffer Inference (2/2)

- 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;
```



cannot propagate Hi-Z in real hardware

assignment can pass Hi-Z to QB in simulation

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: $F = XYZ + X\overline{Y}\overline{Z} + X\overline{Y}Z + \overline{X}YZ + XY\overline{Y}Z$

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

Optimization Approach

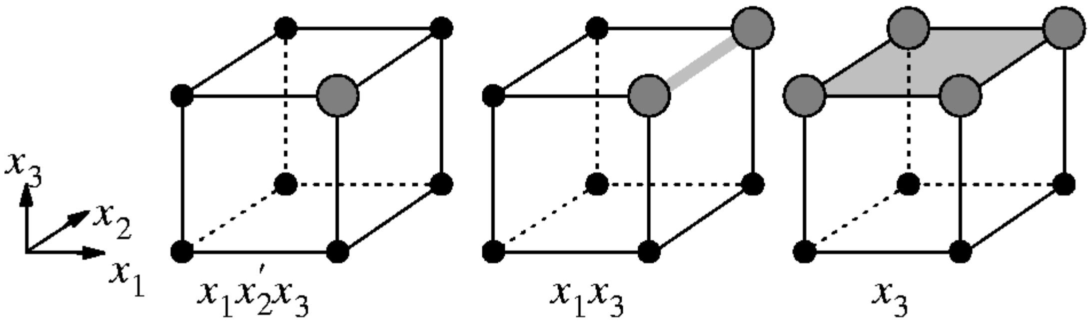
- Exact Methods:
  - Compute minimum cover
  - Often impossible for large functions
  - Ex: Karnaugh maps, Quine-McCluskey
- Heuristic Methods:
  - Compute minimal covers (possibly minimum) in reasonable time
  - Large variety of methods and programs
  - Ex: MINI, PRESTO, ESPRESSO

Boolean Functions

- $B = \{0,1\}$, $Y = \{0,1,D\}$
- A Boolean function $f: B^m \rightarrow Y^n$
  - $f = \overline{x}_1 \overline{x}_2 + \overline{x}_1 \overline{x}_3 + \overline{x}_2 x_3 + x_1 x_2 + x_2 \overline{x}_3 + x_1 x_3$
- Input variables: $x_1, x_2, \ldots$
- The value of the output partitions B^m into three sets
  - the on-set
  - the off-set
  - the dc-set (don't-care set)

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.

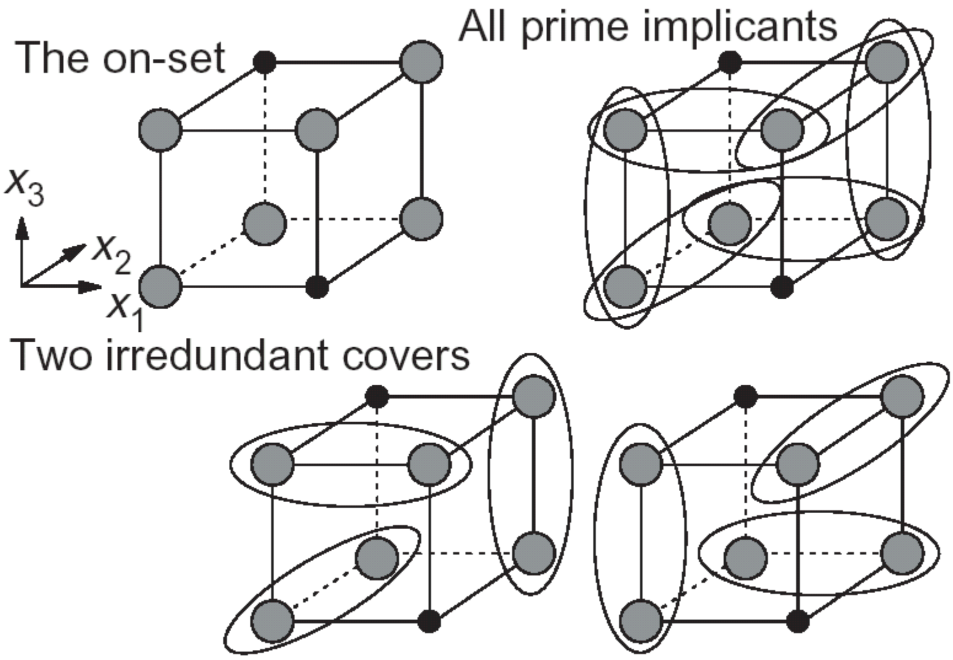


Implicant and Cover

- An **implicant** is a cube whose points are either in the on-set or the dc-set.
- A **prime implicant** is an implicant that is not included in any other implicant.
- 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.
- 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.

Cover Examples

- $f = \overline{x_1}\,\overline{x_3} + \overline{x_2}\,x_3 + x_1\,x_2$
- $f = \overline{x_1}\,\overline{x_2} + x_2\,\overline{x_3} + x_1\,x_3$



The Positional-Cube Notation

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

ϕ	00
0	10
1	01
*	11

One 32-bit integer → 16 binary digits

digit 15	digit 14	………	digit 1	digit 0
2b	2b		2b	2b

- Example: f = a'd' + a'b + ab' + ac'd

10	11	11	10	(a' — — d')
10	01	11	11	(a' b — —)
01	10	11	11	(a b' — —)
01	11	10	01	(a — c' d)

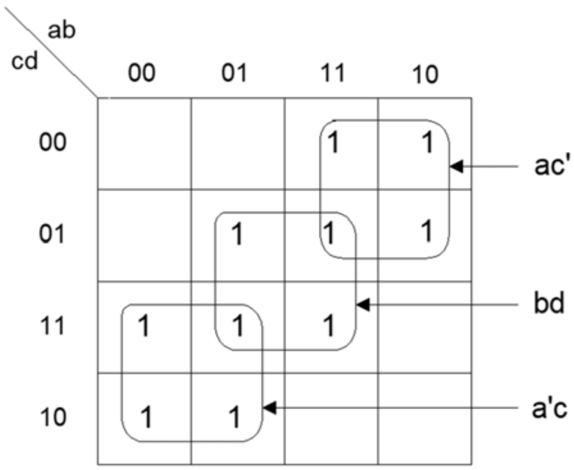
- Example: f_1 = a'b' + ab; f_2 = ab; f_3 = ab' + a'b

10	10	1 0 0	(a'b')
10	01	0 0 1	(a'b)
01	10	0 0 1	(ab')
01	01	1 1 0	(ab)

f1 f2 f3

AND Operation

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



	10	11	01	11	a'c
∩	11	01	11	01	b d
	10	01	01	01	a'b c d

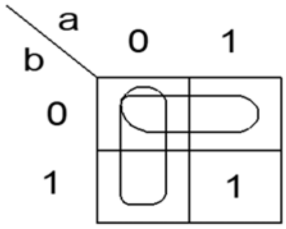
	10	11	01	11	a'c
∩	01	11	10	11	a c'
	00	11	00	11	ϕ

Sharp Operation

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

- Example

$$11 \quad 11 \;\#\; 01 \quad 01 \;=\; \begin{array}{cc} 10 & 11 \\ 11 & 10 \end{array}$$

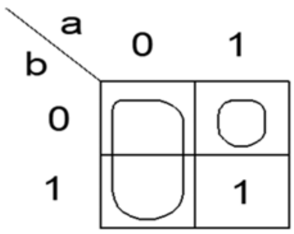


Disjoint Sharp Operation

$$\alpha \circledast \beta = \begin{array}{ccccc} a_1 b_1' & a_2 & \ldots & a_{n-1} & a_n \\ a_1 b_1 & a_2 b_2' & \ldots & a_{n-1} & a_n \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_1 b_1 & a_2 b_2 & \ldots & a_{n-1} b_{n-1}' & a_n \\ a_1 b_1 & a_2 b_2 & \ldots & a_{n-1} b_{n-1} & a_n b_n' \end{array}$$

- Example

$$11 \quad 11 \circledast 01 \quad 01 \;=\; \begin{array}{cc} 10 & 11 \\ 01 & 10 \end{array}$$



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
  - 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
- The (disjoint) sharp operation can be used to compute the **complementation**

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

Cofactor (Restriction)

- Cofactor of f with respect to $x_i = 0$
  - $f_{x_i'} = f_{x_i=0} = f(x_1,x_2,\ldots,x_i=0\ldots,x_n)$
- Cofactor of f with respect to $x_i = 1$
  - $f_{x_i} = f_{x_i=1} = f(x_1,x_2,\ldots,x_i=1\ldots,x_n)$
  - Example:

    $f(x,y,z) = xy + yz' + x'z'$

    $\rightarrow f_{x=0} = yz' + z' \qquad f_{x=1} = y + yz'$
- Cofactor with respect to any cube
  - Example:

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

    $f_{x'y'} = f_{x=0,y=0} = zw' + w'$

    $f_{xy'} = f_{x=1,y=0} = zw'$

Cofactor of Implicants

- The cofactor of an implicant α w.r.t an implicant β is:
- $\alpha_\beta = \phi$ when α does not intersect β

  Otherwise, $\alpha_\beta = a_1+b_1' \quad a_2+b_2' \quad \ldots \quad a_n+b_n'$

- Example: Given f = a'b' + ab, calculate f_a

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

  The cofactor of the first implicant is void

  — a'b' intersect with a is empty

  The cofactor of the second implicant is 11 01

  — (01 01) + (10 00) = (11 01)

  → $f_a = b$

Shannon Expansion

- $f = x' \bullet f_{x=0} + x \bullet f_{x=1}$

  $= x_i' \bullet y_j' \bullet f_{x_i' y_j'} + x_i \bullet y_j' \bullet f_{x_i y_j'} + x_i' \bullet y_j \bullet f_{x_i' y_j} + x_i \bullet y_j \bullet f_{x_i y_j}$

- 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

  $f = x' f_{x'} + x f_x$

- 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: $\forall x(f) = f_x \bullet f_{x'}$
- $\forall x(f)$ evaluate f to be true for x = 1 and x = 0
- Represent the component that is independent of that variable
- Example:

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

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

Smoothing Operations

- Definition: $\exists x(f) = f_x + f_{x'}$
- $\exists x(f)$ evaluate f to be true when x = 1 or x = 0
- Example:

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

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

Boolean Difference

- $\frac{\partial f}{\partial x}$ is called Boolean difference of f with respect to x
- Definition: $\frac{\partial f}{\partial x} = f_x \oplus f_{\bar{x}}$
- 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') ⊕ (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)

cd \ ab	00	01	11	10
00	X	1	0	1
01	0	1	1	1
11	0	X	X	0
10	0	1	0	1

a: ab = 11, 10; b: ab = 01, 11; c: cd = 11, 10; d: cd = 01, 11

Primary Implicant Generation (2/5)

zero "1" → 0000

one "1" → 0100, 1000

two "1" → 0101, 0110, 1001, 1010

three "1" → 0111, 1101

four "1" → 1111

Implication Table	
Column I	
0000	
0100	
1000	
0101	
0110	
1001	
1010	
0111	
1101	
1111	

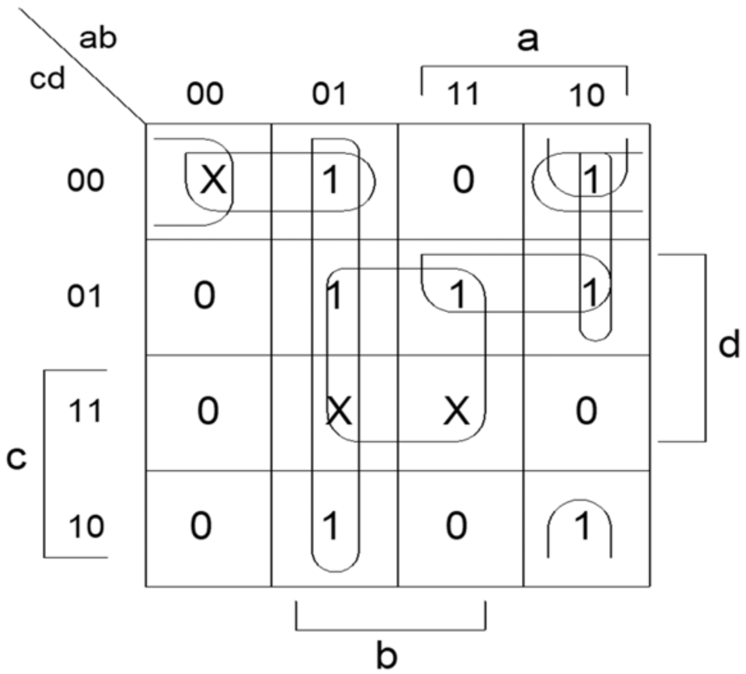
Primary Implicant Generation (3/5)

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-	
1101 |	1-01	
1111 |	-111	
	11-1	

Primary Implicant Generation (4/5)

<table>
<tr><th colspan="3">Implication Table</th></tr>
<tr><th>Column I</th><th>Column II</th><th>Column III</th></tr>
<tr><td>0000 |</td><td>0-00 *</td><td>01-- *</td></tr>
<tr><td></td><td>-000 *</td><td></td></tr>
<tr><td>0100 |</td><td></td><td>-1-1 *</td></tr>
<tr><td>1000 |</td><td>010- |</td><td></td></tr>
<tr><td></td><td>01-0 |</td><td></td></tr>
<tr><td>0101 |</td><td>100- *</td><td></td></tr>
<tr><td>0110 |</td><td>10-0 *</td><td></td></tr>
<tr><td>1001 |</td><td></td><td></td></tr>
<tr><td>1010 |</td><td>01-1 |</td><td></td></tr>
<tr><td></td><td>-101 |</td><td></td></tr>
<tr><td>0111 |</td><td>011- |</td><td></td></tr>
<tr><td>1101 |</td><td>1-01 *</td><td></td></tr>
<tr><td>1111 |</td><td>-111 |</td><td></td></tr>
<tr><td></td><td>11-1 |</td><td></td></tr>
</table>

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)	×						
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)		×					×

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)	×						
0,8 (-000)				×			
8,9 (100-)				×	×		
$\boxed{\text{8,10 (10-0)}}$				×		×	
9,13 (1-01)					×		×
$\boxed{\text{4,5,6,7 (01- -)}}$	×	×	×				
5,7,13,15 (-1-1)		×					×

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 = (P1+P6)(P6+P7)P6(P2+P3+P4)(P3+P5)P4(P5+P7)=1

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

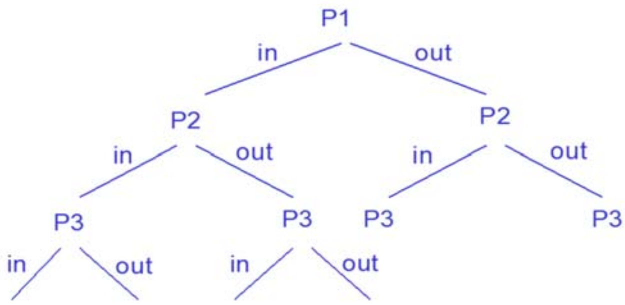
- Each term represents a corresponding column
- Each column must be chosen at least once
- All columns must be covered

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 $x_i = \phi(v)$.

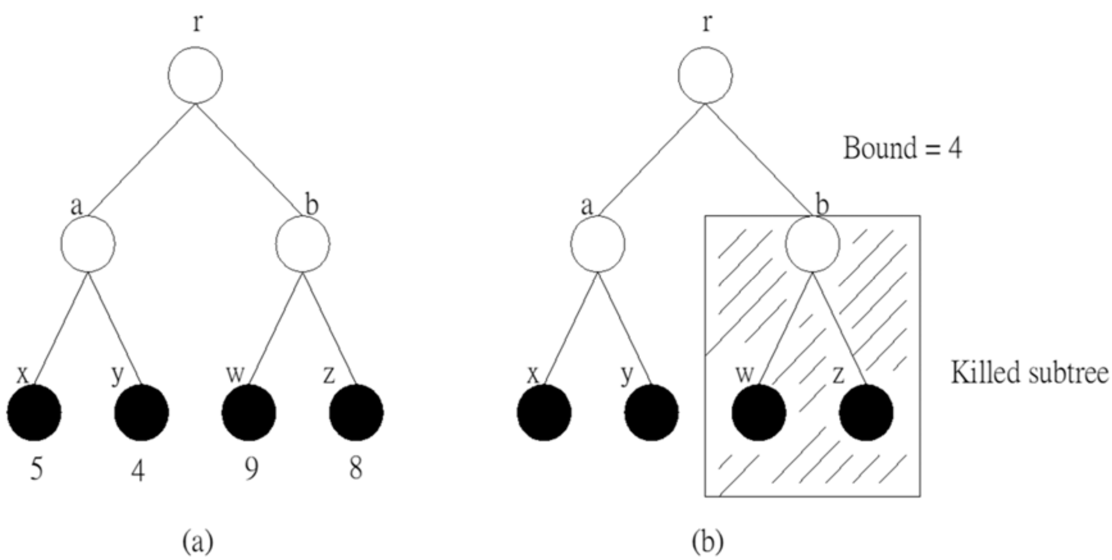
Brute Force Technique

- Brute force technique: Consider all possible elements



- Complete branching tree has $2^{|P|}$ 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

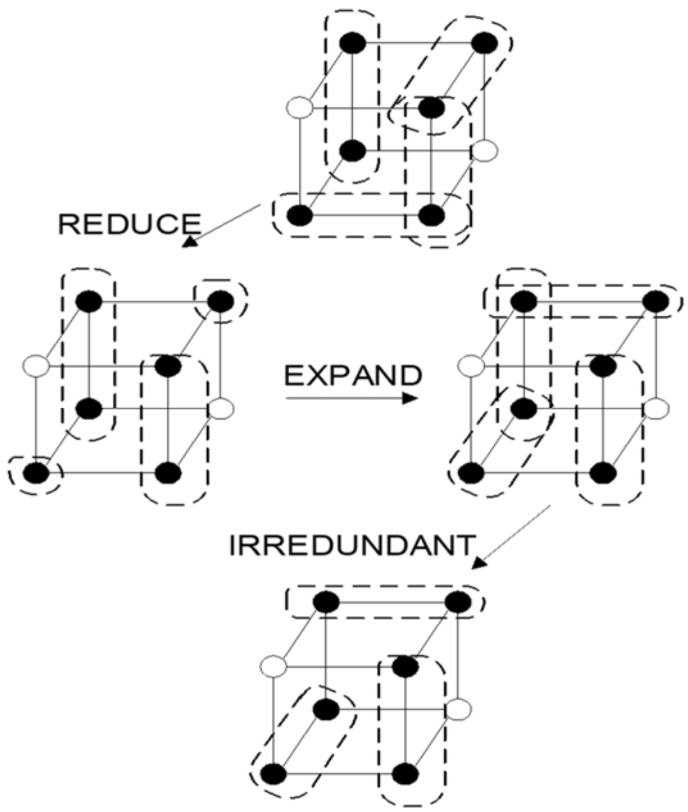
Why Heuristic Optimization ?

- 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

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}:

	10	10	10
choose lowest weight →	$\boxed{01\ \ 10\ \ 10}$		
	10	01	10
	+) 10	10	01
column count →	31	31	31

F^{DC}:

01 01 10

F^{OFF}:

01 11 01

11 01 01

Column count vector = $[313131]^T$

Weight of the implicants = (9,7,7,7)

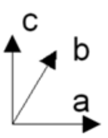
Ex: 2^{nd} implicant of F^{ON}

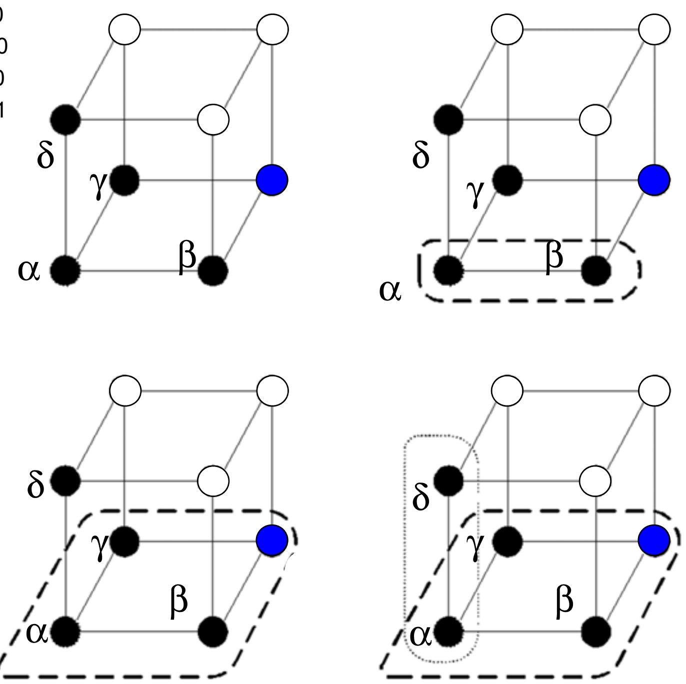
	01	10	10
x)	31	31	31
	01	30	30

2^{nd} weight
= 0+1+3+0+3+0
= 7

Expand (3/3)

α 10 10 10
β 01 10 10
γ 10 01 10
δ 10 10 01





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} - {α }
  - The maximally reduced cube is
    $\alpha'' = \alpha \cap \text{supercube}(Q_\alpha')$ // the part not covered by other implicants
- α # 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

β" → In β but not in Q

F: α 11 11 10
β 10 10 11

column count vector = $[212121]^T$

weight vector = [8,7]

Reduce α first → fail

Reduce β:

Q = F $\cup$ F^{DC} - {β} = {α, β} - {β}
= 11 11 10 // only *α is left*

Q' = 11 11 01 , Q_β' = Q'

supercube(Q_β') = Q' // not in Q

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

Reduced cover is

11 11 10
10 10 **01**

Outline

- Synthesis overview
- RTL synthesis
- Two-level logic optimization
- $\boxed{\text{Multi-level logic optimization}}$
- Technology mapping
- Timing analysis
- Timing optimization
- Synthesis for low power

Multi-Level Logic Optimization

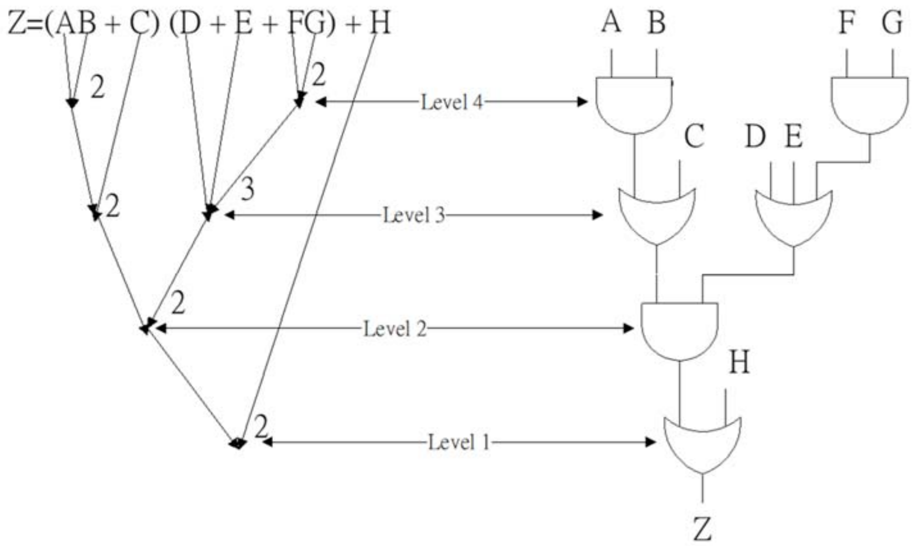
- Level: maximum number of gates cascaded in series between the inputs and outputs of a network
  - Can be considered as an indication for worst-case delay
  - Assume all variables and their complements are available
- Two-level networks have the least depth, not least area
  - It's possible to further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

$$f1 = abcd + abce + a\bar{b}c\bar{d} + a\bar{b}\bar{c}\bar{d} + \bar{a}c + cdf + ab\bar{c}\bar{d}\bar{e} + a\bar{b}\bar{c}d\bar{f}$$
$$f2 = bdg + \bar{b}dfg + \bar{b}\bar{d}g + b\bar{d}eg$$

$$f1 = c\,(\bar{a} + x) + a\bar{c}\bar{x}$$
$$f2 = gx$$
$$x = d\,(b + f) + \bar{d}\,(\bar{b} + e)$$

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



Multi-Level v.s. Two-Level

- Two-level:
  - Often used in control logic design

    $f_1 = x_1x_2 + x_1x_3 + x_1x_4$

    $f_2 = x_1'x_2 + x_1'x_3 + x_1x_4$
  - 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

    $g_1 = x_2 + x_3$

    $g_2 = x_1x_4$

    $f_1 = x_1y_1 + y_2$

    $f_2 = x_1'y_1 + y_2$

    (y_i is the output of gate g_i)

Factored Forms (1/2)

- A *factored from* 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
- Ex: a

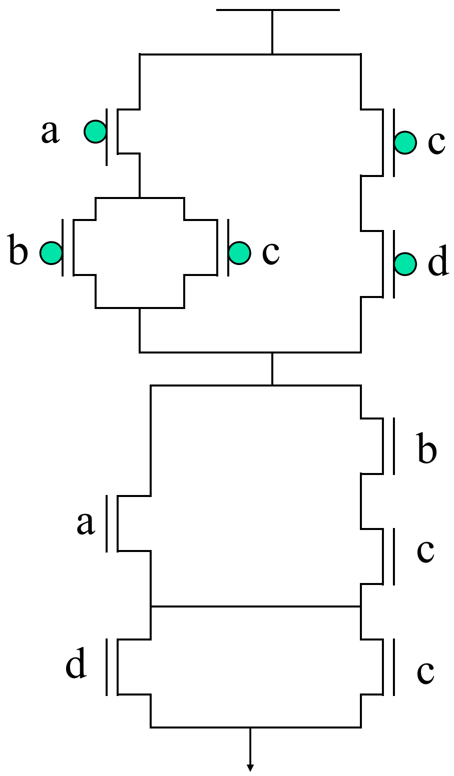
  a’

  ab’c

  ab + c’d

  (a + b)(c + a’ + de) + f

Factored Forms (2/2)



- A CMOS complex gate implementing

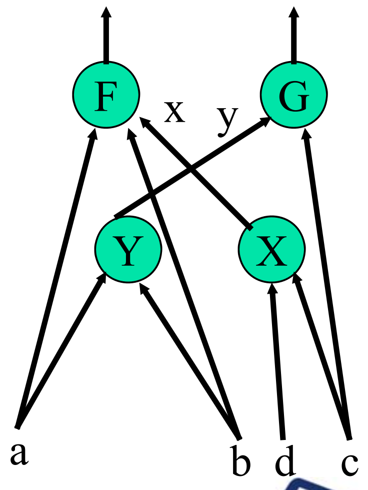
  f = ((a + bc)(c + d))’

  2 *literal count = # 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

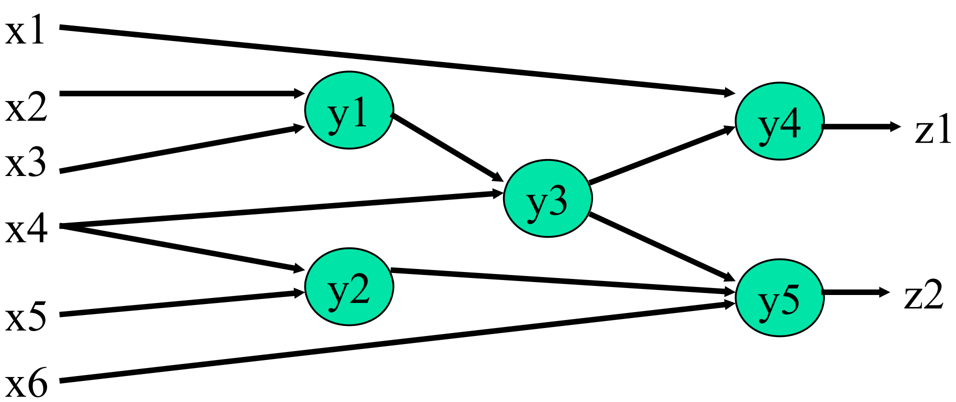
Boolean Network

- 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



$y1 = f_1(x2, x3) = x2' + x3'$

$y2 = f_2(x4, x5) = x4' + x5'$

$y3 = f_3(x4, y1) = x4'y1'$

$y4 = f_4(x1, y3) = x1 + y3'$

$y5 = f_5(x6, y2, y3) = x6y2 + x6'y3'$

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:

$f_1 = abcd + abce + ab'cd' + ab'c'd' + a'c + cdf + abc'd'e' + ab'c'df'$

$f_2 = bdg + b'dfg + b'd'g + bd'eg$

minimize (in sum-of-products form):

$f_1 = bcd + bce + b'd' + b'f + a'c + abc'd'e' + ab'c'df'$

$f_2 = bdg + dfg + b'd'g + d'eg$

decompose:

$f_1 = c(a' + x) + ac'x' \quad x = d(b + d) + d'(b' + e)$

$f_2 = gx$

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)

$f = (az + bz')cd + e$
$g = (az + bz')e'$
$h = cde$

$f = xy + e$
$g = xe'$
$h = ye$
$x = az + bz'$
$y = cd$

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

$f = ga + g'b$
$g = c + d$

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

"Division" plays a key role !!

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 */

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

$q = \cap V^i$

$r = f - pq$

return(q, r)

Weak Division Algorithm (2/2)

- Example

common expressions

$f = acg + adg + ae + bc + bd + be + a'b$

$p = ag + b$

$U = ag + ag + a + b + b + b + b$

$V = c + d + e + c + d + e + a'$

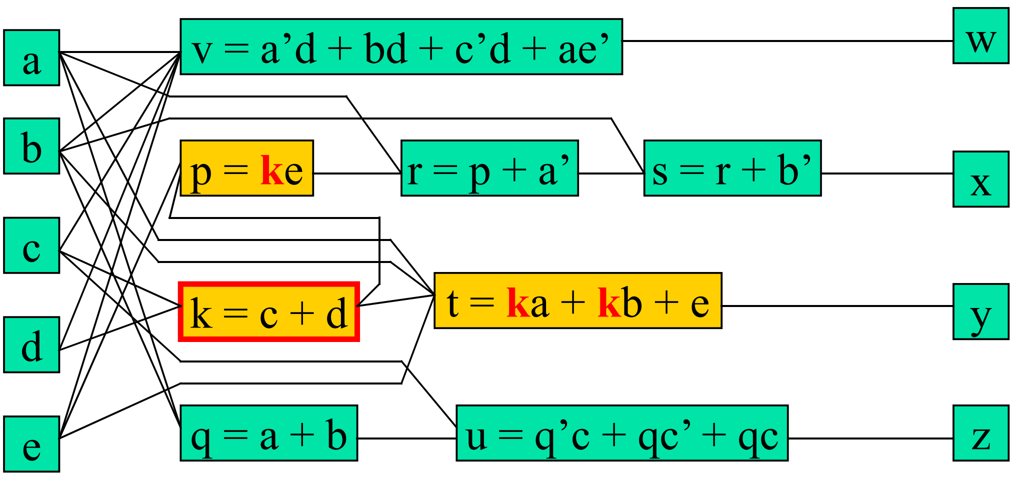
$V^{ag} = c + d$

$V^{b} = c + d + e + a'$

$q = c + d = f/p$

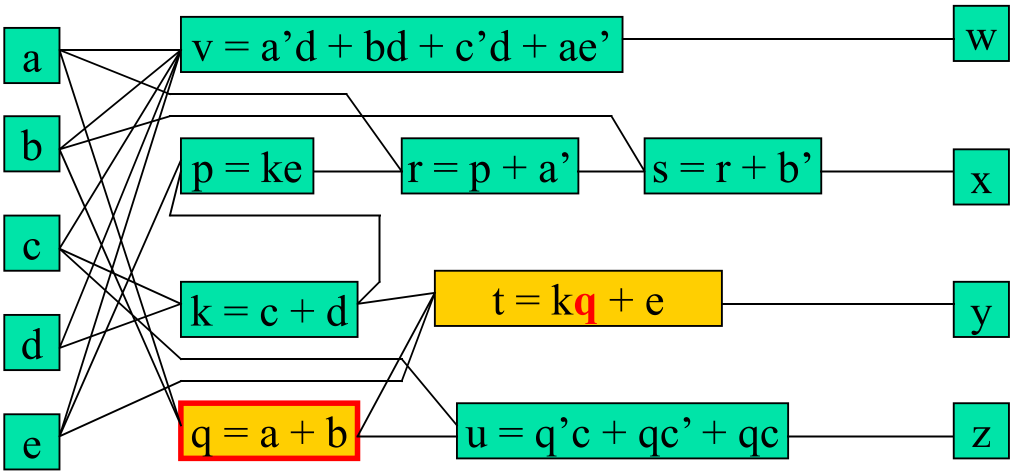
Algebraic Substitution (1/3)

- 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 | c is a cube}
- The ***kernels*** of an expression *f* are the set of expressions

  K(f) = {g | g∈ D(f) and g 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

  x = adf + aef + bdf + bef + cdf + cef + g

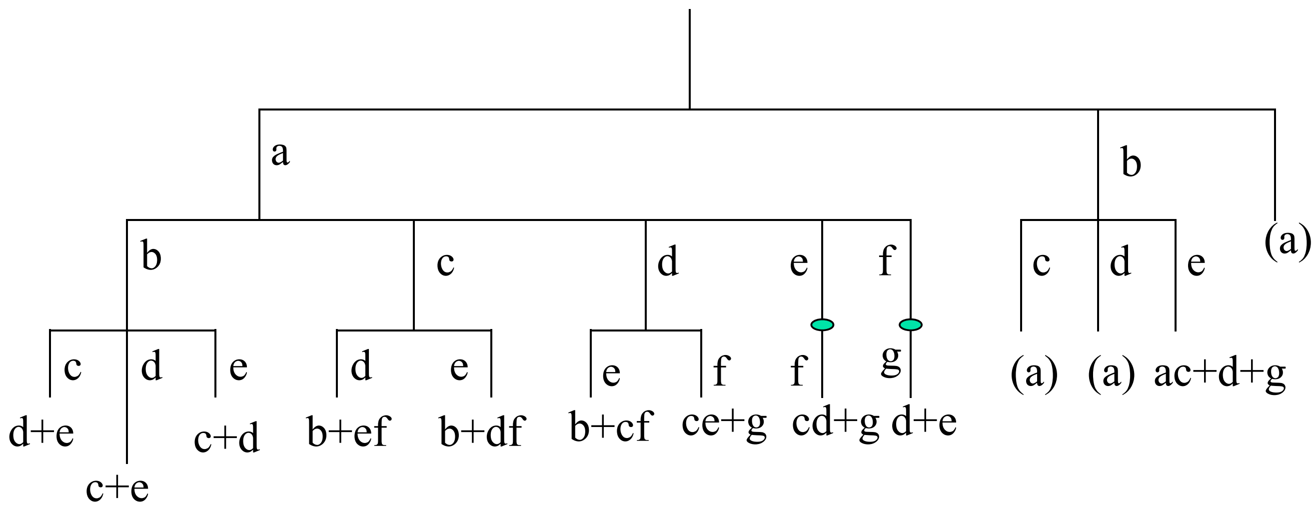
  = (a + b + c)(d + e)f + g

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 + aefg + 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 <=> prime rectangles of the cube-literal martrix
- Optimum selection of kernels <=> rectangle covering
  - Kernel intersection <=> finding rectangles
- Ex: single cube extraction

F = abc + abd + eg

G = abfg

H = bd + ef

({1,2,4},{1,2}) <=> ab

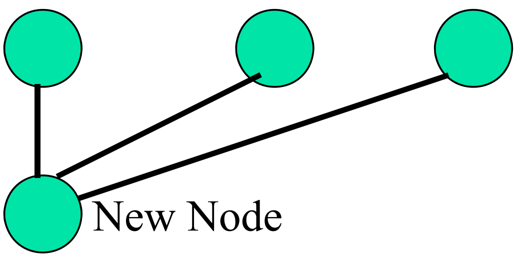
({2,5},{2,4}) <=> bd

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

F = Xc + XY + eg,	X = ab
G = Xfg,	Y = bd
H = Y + ef	

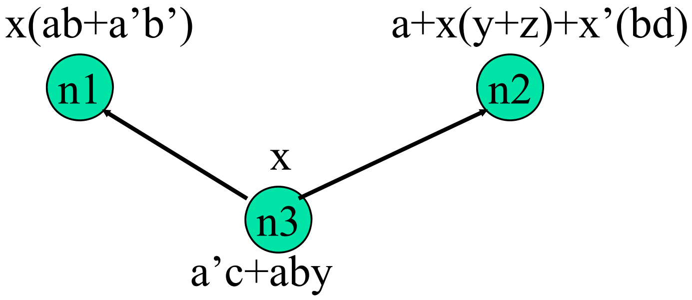
Kernel Extraction (1/2)

- 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 ≤ threshold

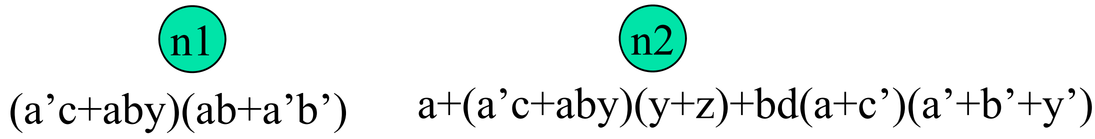


Kernel Extraction (2/2)

- After



- Before



Literals after =5+7+5=17 Literals before =9+15 =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)

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

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

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

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

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

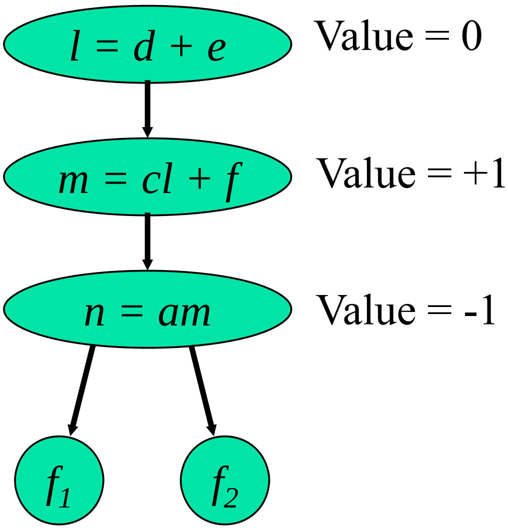
$K^0(f_1) \cap K^0(f_2) = cl + f$ $\quad f_2 = ai(m + j) + k$

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