VLSI System Design
Part II : Logic Synthesis (1)

Lecturer : Tsuyoshi Isshiki

Dept. Communications and Computer Engineering,

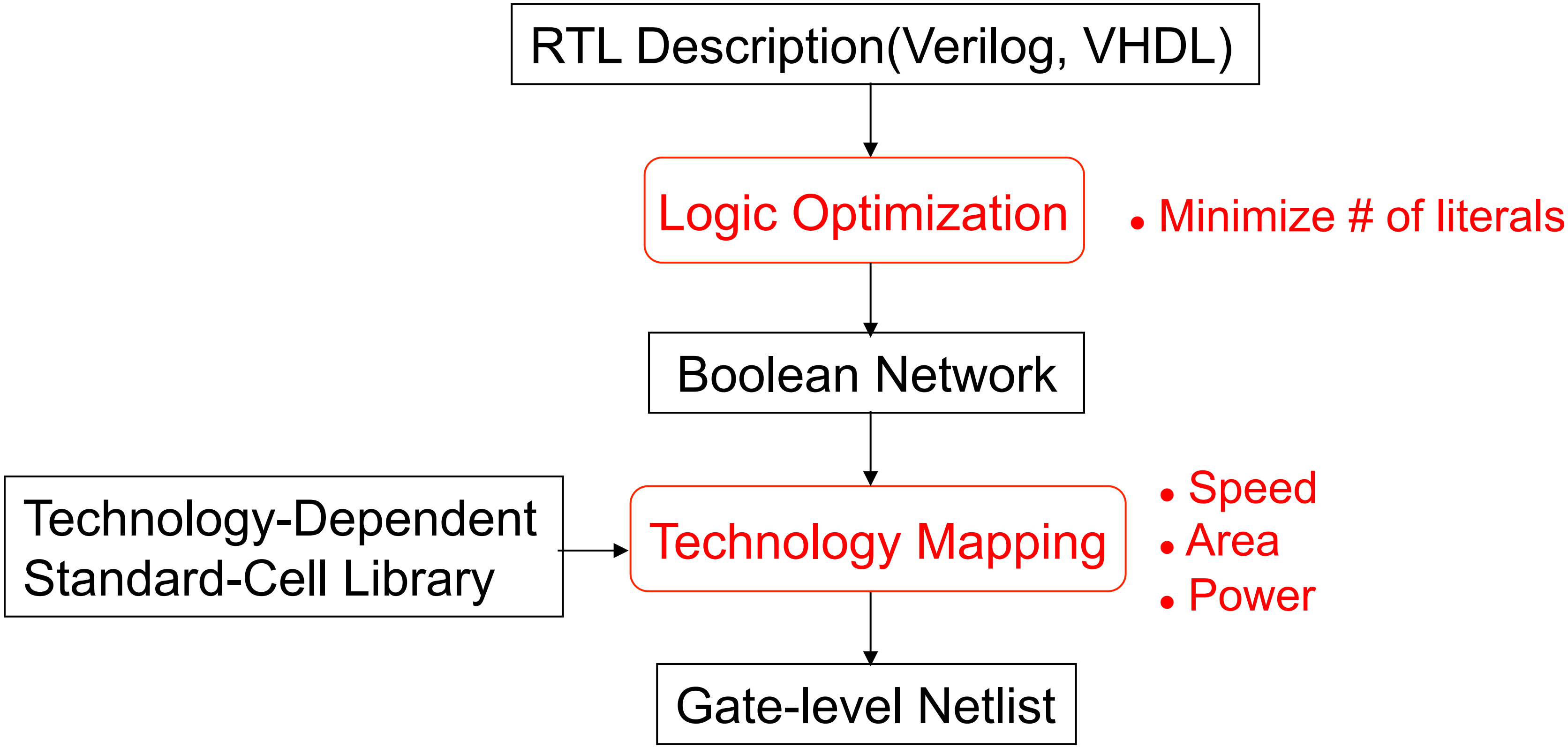
Tokyo Institute of Technology

isshiki@vlsi.ce.titech.ac.jp

Logic Synthesis

1. Logic synthesis types
   a. Combinational logic synthesis
      - Two-level logic
      - Multi-level logic
   b. Sequential logic (finite state machine) synthesis
      - State minimization
      - State encoding
2. Currently available logic synthesis CAD tool
   - Mainly two-level/multi-level logic synthesis
   - State code optimization for sequential logic

Logic Synthesis Flow



RTL-to-Logic Translation (1)

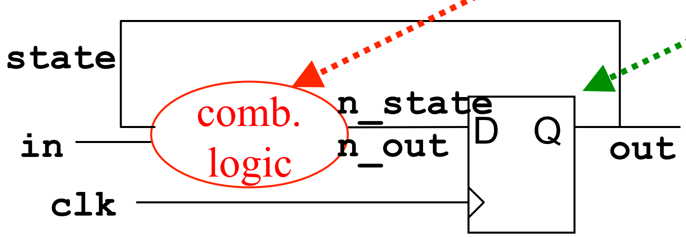
A) *Combinational logic extraction :*

RTL description is partitioned into combinational logic part and storage elements (DFF, latches)

```
module str1101(clk, in, out);
input clk, in;
output out;
reg [1:0] state;
reg out;
always@(posedge clk) begin
  state <= 2'b00;
  out <= 0;
  case(state)
    2'b00: if(in == 1) state <= 2'b01;
    2'b01: if(in == 1) state <= 2'b10;
    2'b10: if(in == 0) state <= 2'b11;
           else state <= 2'b10;
    2'b11: if(in == 1) begin
             out <= 1;
             state <= 2'b01;
             end
    endcase
  end
endmodule
```

```
reg [1:0] n_state;
reg n_out;
always@(in or state) begin
  n_state = 2'b00;
  n_out = 0;
  case(state)
    2'b00: if(in == 1) n_state = 2'b01;
    2'b01: if(in == 1) n_state = 2'b10;
    2'b10: if(in == 0) n_state = 2'b11;
           else n_state = 2'b10;
    2'b11: if(in == 1) begin
             n_out = 1;
             n_state = 2'b01;
             end
    endcase
  end
```

```
always@(posedge clk) begin
    state <= n_state; out <= n_out;
end
```



RTL-to-Logic Translation (2)

B) Logic equation transformation :

For each output variable, compute the conditions in which the value evaluates as 1, 0, and don't-care (DC).

```
begin
  n_state = 2'b00;
  n_out = 0;
  case(state)
    2'b00: if(in == 1) n_state = 2'b01;
    2'b01: if(in == 1) n_state = 2'b10;
    2'b10: if(in == 0) n_state = 2'b11;
           else n_state = 2'b10;
    2'b11: if(in == 1) begin
             n_out = 1;
             n_state = 2'b01;
             end
    endcase
end
```

```
if(state == 2'b00 && in == 1 ||
   state == 2'b10 && in == 0 ||
   state == 2'b11 && in == 1)
   n_state[0] = 1;
else n_state[0] = 0;
```

```
if(state == 2'b01 && in == 1 ||
   state == 2'b10)
   n_state[1] = 1;
else n_state[1] = 0;
```

```
if(state == 2'b11 && in == 1)
   n_out = 1;
else n_out = 0;
```

```
n_state[0] =
  (~state[0] & ~state[1] & in ||
   ~state[0] & state[1] & ~in ||
   state[0] & state[1] & in);

n_state[1] =
  (state[0] & ~state[1] & in ||
   ~state[0] & state[1]);

n_out = state[0] & state[1] & in);
```

RTL-to-Logic Translation (3)

A) Combinational logic extraction

```
module str11011(clk, rst, in, out);
input clk, in;
output out;
reg [2:0] state;
reg out;
always@(posedge clk) begin
  state <= 3'b000;
  out <= 0;
  if(rst == 0)
    case(state)
      3'b000: if(in == 1) state <= 3'b001;
      3'b001: if(in == 1) state <= 3'b010;
      3'b010: if(in == 0) state <= 3'b011;
              else state <= 3'b010;
      3'b011: if(in == 1) state <= 3'b100;
      3'b100: if(in == 1) begin
                out <= 1;
                state <= 3'b010;
                end
      default: begin // don't-care state
               state <= 3'bx;
               out <= x;
               end
    endcase
endmodule
```

```
......
reg [2:0] n_state;
reg n_out;
always@(in or rst or state) begin
  n_state = 3'b000;
  n_out = 0;
  if(rst == 0)
    case(state)
      3'b000: if(in == 1) n_state = 3'b001;
      3'b001: if(in == 1) n_state = 3'b010;
      3'b010: if(in == 0) n_state = 3'b011;
              else n_state = 3'b010;
      3'b011: if(in == 1) n_state = 3'b100;
      3'b100: if(in == 1) begin
                n_out = 1;
                n_state = 3'b010;
                end
      default: begin // don't-care state
               n_state = 3'x;
               out = x;
               end
    endcase
  end
always@(posedge clk) begin
    state <= n_state; out <= n_out;
end
......
```

RTL-to-Logic Translation (4)

B) Logic equation transformation :

```
begin
  n_state = 3’b000;
  n_out = 0;
  if(rst == 0)
    case(state)
      3’b000: if(in == 1) n_state = 3’b001;
      3’b001: if(in == 1) n_state = 3’b010;
      3’b010: if(in == 0) n_state = 3’b011;
              else n_state = 3’b010;
      3’b011: if(in == 1) n_state = 3’b100;
      3’b100: if(in == 1) begin
                n_out = 1;
                n_state = 3’b010;
                end
      default: begin // don’t-care state
               n_state = 3’x;
               out = x;
               end
    endcase
  end
```

```
if(state == 3’b000 && in == 1 ||
   state == 3’b010 && in == 0)
   n_state[0] = 1;
else if(state == 3’b101 ||
   state == 3’b110 ||
   state == 3’b111)
   n_state[0] = x;
else n_state[0] = 0;
```

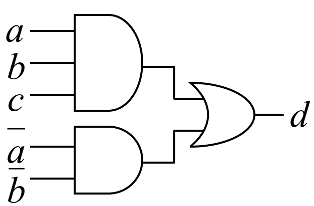
```
if(state == 3’b001 && in == 1 ||
   state == 3’b010 ||
   state == 3’b100 && in == 1)
   n_state[1] = 1;
else if(state == 3’b101 ||
   state == 3’b110 ||
   state == 3’b111)
   n_state[1] = x;
else n_state[1] = 0;
```

```
if(state == 3’b011 && in == 1)
   n_state[2] = 1;
else if(state == 3’b101 ||
   state == 3’b110 ||
   state == 3’b111)
   n_state[2] = x;
else n_state[2] = 0;
```

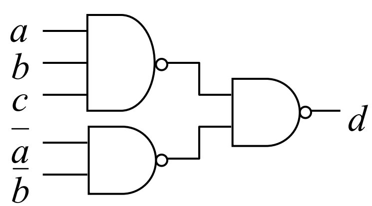
```
if(state == 3’b100 && in == 1)
   n_out = 1;
else if(state == 3’b101 ||
   state == 3’b110 ||
   state == 3’b111)
   n_out = x;
else n_out = 0;
```

Boolean Function Implementation Using Two-Level Logic

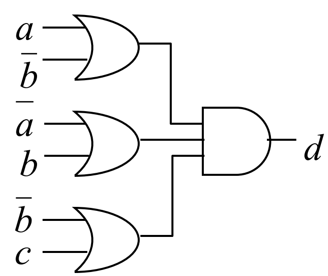
- The study of logic synthesis started from two-level logic
- Optimized two-level logic is often the starting point for multi-level logic synthesis.
- Several types of two-level logic
  - Sum-of-product (1^{st} level : AND, 2^{nd} level : OR)
  - NAND-NAND (has the same structure as sum-of-product)
  - Product-of-sum (1^{st} level : OR, 2^{nd} level : AND)
  - NOR-NOR (has the same structure as product-of-sum)



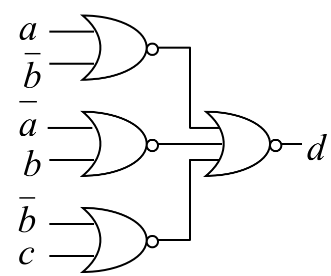
sum-of-product



NAND-NAND



product-of-sum

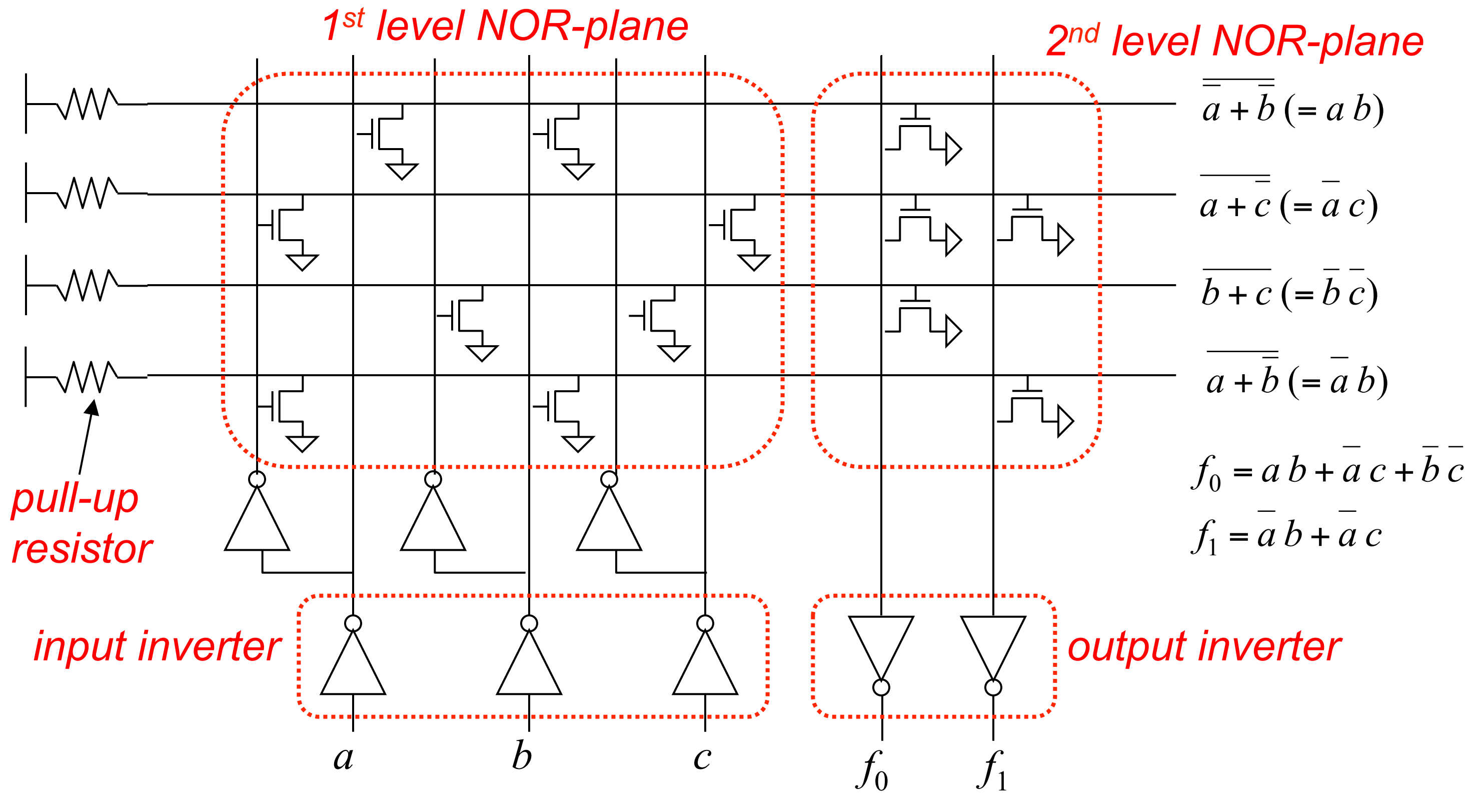


NOR-NOR

All four circuits implement the same function

Programmable Logic Array

- A programmable logic array is a device which can implement arbitrary Boolean function in sum-of-product form with *N* inputs, *M* outputs, and *R* products (cubes).
- Minimizing the number of products *R* results in smaller area (*N* and *M* are fixed for a given function)



Boolean Function Terminologies (1)

1. Boolean function f with N inputs and M outputs is a mapping $f: \{0, 1\}^N \rightarrow \{0, 1, X\}^M$. ($X$: don't-care)
2. If mapping to don't-care values does not exist, the function is said to be *completely specified*. Otherwise it is said to be *incompletely specified*.
3. If $M = 1$, it is called a *single-output function*. Otherwise it is called a *multiple-output function*.
4. For each output f_m of function f :
   - *ON-set* is defined as the set of input values x such that $f_m(x) = 1$
   - *OFF-set* is defined as the set of input values x such that $f_m(x) = 0$
   - *DC-set* is defined as the set of input values x such that $f_m(x) = X$
5. A *literal* is a Boolean variable or its complement.
6. A *cube* is a conjunction of literals (a product term).
7. A *cover* is a set of cubes (interpreted as sum-of-product term).

Boolean Function Terminologies (2)

8. A *bit vector notation* of a cube describes the polarity of each literal (0 : complemented literal, 1 : uncomplemented literal) for each variable in the Boolean function. If a variable does not appear in the cube, it is denoted as ‘-’ (also don’t-care)

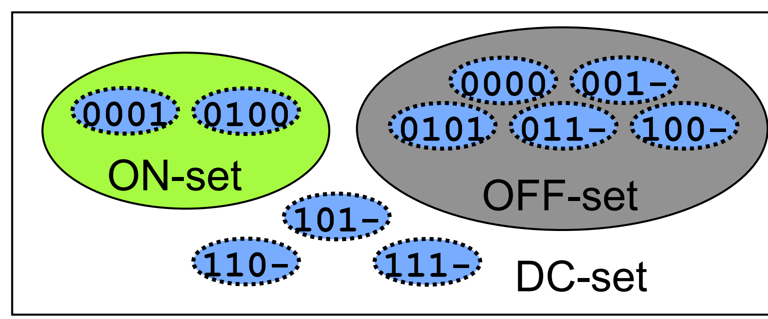
   Ex. $x_3 \overline{x_2} x_1 \overline{x_0} \rightarrow$ **1010** $\quad x_3 x_2 \overline{x_0} \rightarrow$ **11-0**

9. A cube is called a *k*-cube if there are *k* elements of ‘-’ (don’t-care) in the bit vector notation.

10. A *minterm* is a cube that contains all variables in the Boolean function. Each minterm belongs to either the ON-set, OFF-set or the DC-set of a particular output of the function. A minterm is a 0-cube.

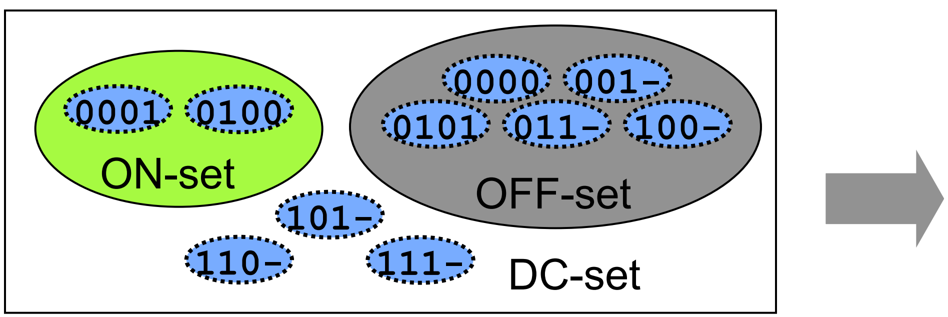
```
if(state == 3’b000 && in == 1 ||
   state == 3’b010 && in == 0)
   n_state[0] = 1;
else if(state == 3’b101 ||
   state == 3’b110 ||
   state == 3’b111)
   n_state[0] = x;
else n_state[0] = 0;
```

function $f_{n\_state[0]}(state[2], state[1], state[0], in)$

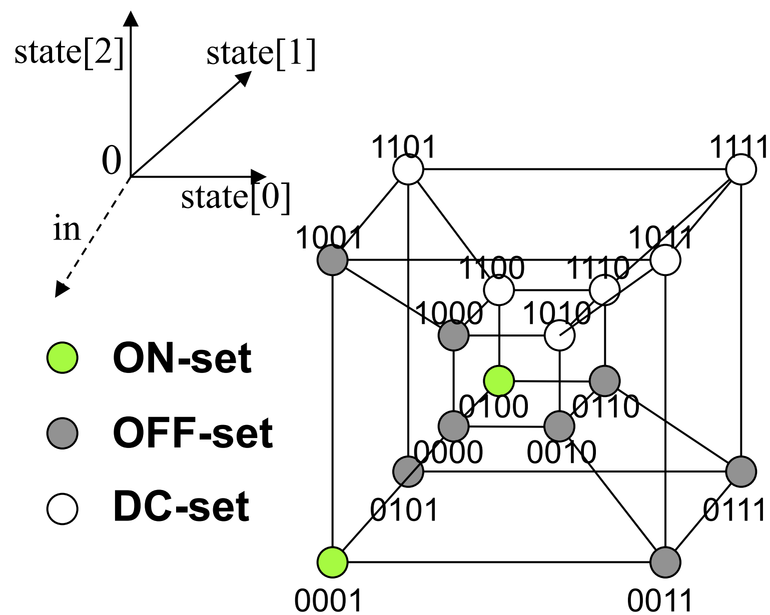


Boolean Function Terminologies (3)

11. The input variable space $\{0, 1\}^N$ can be modeled as a binary *N*-dimensional hypercube
    - Each vertex in the hypercube represents a minterm.
    - *k*-cube is represented by a binary *k*-dimensional hypercube
    - *k*-dimensional hypercube is sometimes referred to as "binary *k*-cube"

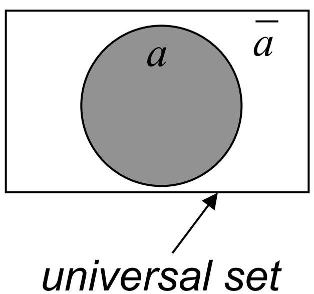


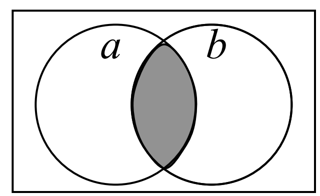
function $f_{n\_state[0]}(state[2], state[1], state[0], in)$



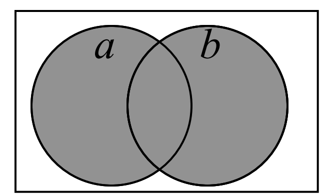
Boolean Function Terminologies (4)

12. Analogy of Boolean algebra to Class calculus (*Set Theory*)
    - logic variable → set
    - logic negation → complement set
    - logical 1 → universal set
    - logical 0 → null set (ϕ)
    - logical AND → set intersection ($a \cdot b \rightarrow a \cap b$)
    - logical OR → set union ($a + b \rightarrow a \cup b$)





$a \cdot b$ ($a \cap b$)



$a + b$ ($a \cup b$)

Boolean Function Terminology (4)

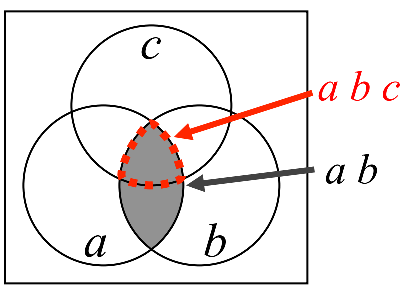
13. Partial order and containment
    - Partial order of logic variables : $f \le g \Leftrightarrow (\text{if } f = 1, \text{ then } g = 1) \Leftrightarrow f \cdot g = f$
        - Interpretation in set theory → containment of sets : $f \subseteq g$
    - Partial order of logic expression (cubes and covers) :

      $a\,b\,c \le a\,b \rightarrow a\,b\,c \subseteq a\,b$

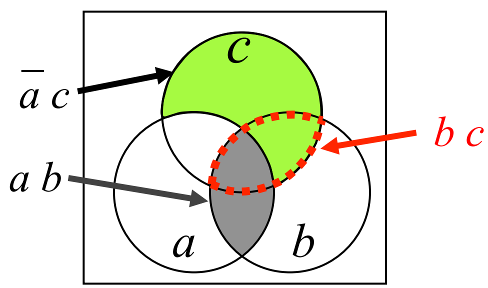
      $b\,c \le a\,b + \bar{a}\,c \rightarrow b\,c \subseteq a\,b + \bar{a}\,c$

      $a\,\bar{c} + b\,c \le a\,b + \bar{a}\,c + a\,\bar{b}\,\bar{c} \rightarrow a\,\bar{c} + b\,c \subseteq a\,b + \bar{a}\,c + a\,\bar{b}\,\bar{c}$

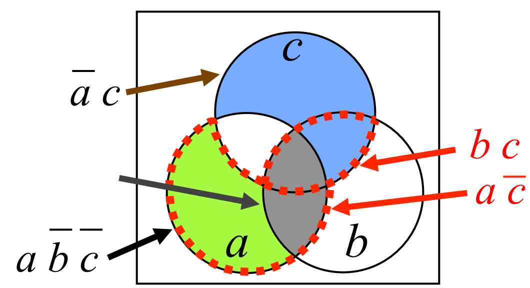
    ✓ *Terminologies for set theory (intersection, union, containment) is often applied to logic expressions.*



$a\,b\,c \subseteq a\,b$



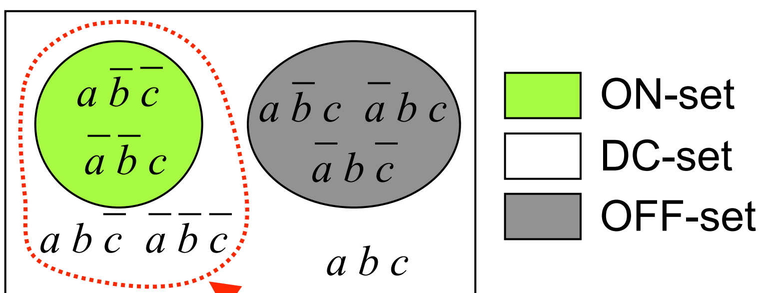
$b\,c \subseteq a\,b + \bar{a}\,c$

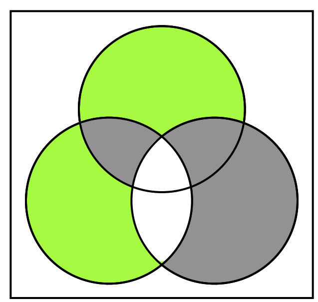


$a\,\bar{c} + b\,c \subseteq a\,b + \bar{a}\,c + a\,\bar{b}\,\bar{c}$

Boolean Function Terminology (5)

14. An *implicant* for a particular output of a function is a cube which contains minterms only in the ON-set and DC-set. (*In other words, a cube which does not intersect with the OFF-set*)
15. A *prime implicant* (or simply, *prime*) is an implicant that is not contained by any other implicant, and intersects with the ON-set.
16. An *essential prime implicant* (or *essential prime*) is a prime that contains one or more minterms which are not contained by other primes.
17. A *legal cover* for a function is a set of implicants which contains the ON-set and does not intersect with OFF-set (may intersect with DC-set).





Implicants : $a\,\bar b\,\bar c,\ \bar a\,\bar b\,c,\ a\,b\,c,\ a\,b\,\bar c,\ \bar a\,\bar b\,\bar c,\ a\,b,\ \bar a\,\bar b,\ a\,\bar c,\ \bar b\,\bar c$

Primes : $\bar a\,\bar b,\ a\,\bar c,\ \bar b\,\bar c$

Essential primes : $\bar a\,\bar b\,,\ a\,\bar c$

Legal cover : $\bar a\,\bar b + a\,\bar c$

Two-Level Logic Optimization

- Input : Boolean function representation using
  - Truth table *or*
  - Set of cubes in the ON-, OFF- and DC-sets.
    - ✓ *Since the union of the ON-, OFF- and DC-sets is the universal set, specifying two sets (ex. ON-set and DC-set) is sufficient for describing a Boolean function.*
    - ✓ *For a completely specified function, only the ON-set is needed.*
- Output : optimized Boolean function in terms of number of cubes (or sometimes number of literals)
- Algorithm :

A) Enumerate all prime implicants of the target function

B) Select a minimum set of prime implicants which are required to contain the ON-set of the target function.

Preparation : Cube Reduction

- For a pair of cubes *A* and *B*, if there exists an cube *C* such that *A* + *B* = *C*, then *A* and *B* are said to be *adjacent* and are *reducible* to *C*.
- On the bit-vector representations, adjacency of a pair of implicants can be determined by comparing elements in each position : if only one position is different, and if all '-' positions are same, then the implicant pair is adjacent.

	a	b	c	d	
$a\,b\,c\,\bar{d}$	1	1	1	0	
$a\,b\,c\,d$	1	1	1	1	**reducible**
$a\,b\,c$	1	1	1	-	
$a\,b\,\bar{c}$	1	1	0	-	*don't-cares at the same position*
$a\,b\,c$	1	1	1	-	**reducible**
$a\,b$	1	1	-	-	

Single-Output 2-Level Logic Minimization Using Quine-McCluskey Method (1)

1. Prime implicant extraction

A) From the truth table, delete minterms in OFF-set. (0-cube table : contains only minterm implicants)

B) k=0.

C) Let N be the # of rows in k-cube table.

If N=0, then terminate.

D) for(i = 0; i < N; i ++)

for(j = i + 1; j < N; j ++)

If rows i and j are adjacent,

- mark these 2 rows with '*'
- add a reduced cube to (k+1)-cube table
- Output part of the reduced cube is 1 if it intersects with the ON-set. Otherwise (if it is fully contained in the DC-set), it is *x*.

E) k=k+1. Go to C) .

F) Rows whose output is 1 and without '*' marked are the prime implicants.

truth table

x[3:0]	f[1]
0000	0
0001	1
0010	0
0011	0
0100	0
0101	1
0110	0
0111	x
1000	1
1001	0
1010	1
1011	0
1100	1
1101	1
1110	1
1111	0

0-cube table

x[3:0]	f[1]	
0001	1	*
0101	1	*
0111	x	*
1000	1	*
1010	1	*
1100	1	*
1101	1	*
1110	1	*

1-cube table

x[3:0]	f[1]	
0-01	1	
01-1	1	
-101	1	
10-0	1	*
1-00	1	*
1-10	1	*
110-	1	
11-0	1	*

2-cube table

x[3:0]	f[1]
1--0	1

Single-Output 2-Level Logic Minimization Using Quine-McCluskey Method (2)

0-cube table

x[3:0]	f[1]	
0001	1	*
0101	1	*
0111	x	*
1000	1	*
1010	1	*
1100	1	*
1101	1	*
1110	1	*

1-cube table

x[3:0]	f[1]	
0-01	1	
01-1	1	
-101	1	
10-0	1	*
1-00	1	*
1-10	1	*
110-	1	
11-0	1	*

2-cube table

x[3:0]	f[1]
1--0	1
1--0	1

identical cubes are generated here. (delete the 2nd cube from the list)

Single-Output 2-Level Logic Minimization Using Quine-McCluskey Method (3)

2. Prime implicant table generation

A) Assign ON-set minterms to each row

B) Assign prime implicants to each column

C) For each minterm row, mark an '**x**' at the column whose prime implicant contains this minterm

3. Prime implicant cover extraction containing all ON-set minterm

(*minimum unate covering problem : NP-complete*)

A) Delete dominated prime (column) and dominating minterm (row)

B) Extract essential primes and delete all minterms (rows) which are contained in these essential primes.

C) Arbitrary select a prime and delete all minterms which are contained in this prime.

prime implicant

x[3:0]	f[1]	0-01	01-1	-101	110-	1--0
0001	1	x				
0101	1	x	x	x		
1000	1					x
1010	1					x
1100	1				x	x
1101	1			x	x	
1110	1					x

ON-set minterm

Techniques to reduce the problem complexity (can be applied in any order)

Single-Output 2-Level Logic Minimization Using Quine-McCluskey Method (4)

3.A Elimination of dominating minterms

- **Prime set for a minterm**
  A set of primes which contain the minterm
  Ex: prime set for 0101 is {0-01, 01-1, -101}
- **Dominating minterm :**
  On a pair of minterms, if the prime set of one of the minterm contains that of the other, the former minterm is said to be the *dominating minterm* of the latter.

➢ Prime set is the set of candidate for covering the particular minterm. *Dominating* minterms can be eliminated from the problem since the prime which covers some *dominated* minterm always covers the corresponding *dominating* minterm.

x[3:0]	f[1]	0-01	01-1	-101	110-	1--0
0001	1	x				
~~0101~~	~~1~~	~~x~~	~~x~~	~~x~~		
1000	1					x
1010	1					x
~~1100~~	~~1~~				~~x~~	~~x~~
1101	1			x	x	
1110	1					x

- Row `0101` is the dominating minterm of row `0001`.
- Row `1100` is the dominating minterm of rows `1000`, `1010` and `1110`.

Single-Output 2-Level Logic Minimization Using Quine-McCluskey Method (5)

3.A Elimination of dominated primes

- **Minterm set for a prime**
  A set of minterms which are contained by the prime
  Ex: minterm set for 0-01 is {0001, 0101}
- **Dominated prime :**
  On a pair of primes, if the minterm set of one of the prime contains that of the other, the latter prime is said to be the *dominated prime* of the former.

➢ *Dominated* primes can be eliminated from the problem since the entire minterm set of a *dominated* prime is always covered by the *dominating* prime.

$x_{[3:0]}$	$f_{[1]}$	0-01	01-1	-101	110-	1--0
0001	1	x				
0101	1	x	x	x		
1000	1					x
1010	1					x
1100	1				x	x
1101	1			x	x	
1110	1					x

• Column **01-1** is the dominated prime of column **0-01** and **-101**.

Single-Output 2-Level Logic Minimization Using Quine-McCluskey Method (6)

essential primes

		0	0	-	1	1
		-	1	1	1	-
		0	-	0	0	-
x[3:0]	**f**[1]	1	1	1	-	0
0001	1	x				
~~0101~~	~~1~~	~~x~~	~~x~~	~~x~~		
1000	1					x
1010	1					x
~~1100~~	~~1~~				~~x~~	~~x~~
1101	1			x	x	
1110	1					x

⇨

		0	-	1	1
		-	1	1	-
		0	0	0	-
x[3:0]	**f**[1]	1	1	-	0
0001	1	x			
1000	1				x
1010	1				x
1101	1		x	x	
1110	1				x

⇨

		-	1
		1	1
		0	0
x[3:0]	**f**[1]	1	-
1101	1	x	x

3.B Extraction of essential primes

- An *essential prime implicant* (or *essential prime*) is a prime that has at least one ON-set minterm which are not contained in any other primes. Such minterms are called *essential minterms*.

Single-Output 2-Level Logic Minimization Using Quine-McCluskey Method (7)

select an arbitrary prime

x[3:0]	f[1]	-101	110-
1101	1	x	x

No minterm left to cover
TERMINATE

x[3:0]	f[1]	110-

obtained prime cover

x[3:0]	f[1]	0-01	01-1	-101	110-	1--0
0001	1	x				
0101	1	x	x	x		
1000	1					x
1010	1					x
1100	1				x	x
1101	1			x	x	
1110	1					x

3.C Arbitrary selection of remaining primes

- If 3.A (*elimination of dominating minterms and dominated primes*) and 3.B (*essential prime extraction*) cannot further be applied, select an arbitrary remaining prime and delete the rows (minterms) which is contained in this prime. Try 3.A and 3.B again.
- If all minterms have been covered, then *TERMINATE*.
- In order to obtain an optimal cover, do all combinations of the arbitrary prime selection.

Multiple-Output 2-Level Logic Minimization Using Quine-McCluskey Method (1)

1. Prime Implicant Extraction

A) Extract the prime implicants for each output seperately.

truth table

x[3:0]	f[1:0]
0000	00
0001	10
0010	00
0011	0x
0100	0x
0101	10
0110	01
0111	x0
1000	11
1001	00
1010	11
1011	01
1100	1x
1101	11
1110	10
1111	00

f[1]

0-cube table

x[3:0]	f[1]	
0001	1	*
0101	1	*
0111	x	*
1000	1	*
1010	1	*
1100	1	*
1101	1	*
1110	1	*

f[1]

1-cube table

x[3:0]	f[1]	
0-01	1	
01-1	1	
-101	1	
10-0	1	*
1-00	1	*
1-10	1	*
110-	1	
11-0	1	*

f[1]

2-cube table

x[3:0]	f[1]
1--0	1

f[0]

0-cube table

x[3:0]	f[0]	
0011	x	*
0100	x	*
0110	1	
1000	1	*
1010	1	*
1011	1	*
1100	x	*
1101	1	*

f[0]

1-cube table

x[3:0]	f[0]
-011	1
-100	x
10-0	1
1-00	1
101-	1
110-	1

not a prime implicant because the output is 'x'

Multiple-Output 2-Level Logic Minimization Using Quine-McCluskey Method (2)

2. Prime Implicant List Merging

- *m*th prime implicant list corresponds to the prime implicant list for the *m*th output
- *m*th outputs in *m*th prime implicant list are all 1s *by definition*

A) For each prime *p* in the *all prime implicant lists*

- *m*th output is 1 if there exists a prime in the *m*th prime implicant list which contains *p*.
- *m*th output is 0 otherwise.

➢ *This allows implicants other than the primes to be included in the candidate for minterm covering.*

prime implicant list

```
x[3:0] f[1:0]
----------
0-01   10
01-1   10
-101   10
110-   11
1--0   10

0110   01
-011   01
10-0   11
1-00   11
101-   01
110-   11
```

these are identical primes delete one of them from the list

Multiple-Output 2-Level Logic Minimization Using Quine-McCluskey Method (3)

3. Prime implicant table generation

A) Assign ON-set minterms to each row *for each output*

B) Assign prime implicants to each column

C) For each minterm row,

- mark an '|' at the column whose output part of the corresponding prime implicant is 0 for the corresponding output of this minterm
- *Otherwise*, mark an 'X' at the column whose prime implicant contains this minterm

x[3:0]	f[1:0]	0-01	01-1	-101	110-	1--0	0110	-011	10-0	1-00	101-
0001	1-	x					|	|			|
0101	1-	x	x	x			|	|			|
1000	1-					x	|	|	x	x	|
1010	1-					x	|	|	x		|
1100	1-				x	x	|	|		x	|
1101	1-			x	x		|	|			|
1110	1-					x	|	|			|
0110	-1	|	|	|		|	x				
1000	-1	|	|	|		|			x	x	
1010	-1	|	|	|		|			x		x
1011	-1	|	|	|		|		x			x
1101	-1	|	|	|	x	|					

These primes cannot be used for covering because they intersect with the OFF-set of the corresponding output

Multiple-Output 2-Level Logic Minimization Using Quine-McCluskey Method (4)

4. Prime implicant cover extraction containing all ON-set minterms

(same as the single-output case)

x[3:0]	f[1:0]	0-01	01-1	-101	110-	1--0	0110	-011	10-0	1-00	101-
0001	1-	x					|	|			|
0101	1-	x	x	x			|	|			|
1000	1-					x	|	|	x	x	|
1010	1-					x	|	|	x		|
1100	1-				x	x	|	|		x	|
1101	1-			x	x		|	|			|
1110	1-					x	|	|			|
0110	-1	|	|	|		|	x				
1000	-1	|	|	|		|			x	x	
1010	-1	|	|	|		|			x		x
1011	-1	|	|	|		|		x			x
1101	-1	|	|	|	x	|					

x[3:0]	f[1:0]	0-01	-101	110-	1--0	0110	10-0	1-00	101-
0001	1-	x				|			|
1101	1-		x	x		|			|
1110	1-				x	|			|
0110	-1	|	|		|	x			
1000	-1	|	|		|		x	x	
1010	-1	|	|		|		x		x
1011	-1	|	|		|				x
1101	-1	|	|	x	|				

x[3:0]	f[1:0]	0-01	110-	1--0	0110	10-0	101-
0001	1-	x			|		|
1110	1-			x	|		|
0110	-1	|		|	x		
1000	-1	|		|		x	
1011	-1	|		|			x
1101	-1	|	x	|			

Improving Quine-McCluskey Method
(Espresso-EXACT, UC Berkeley)

- **Problems**
  - Need to specify all minterms
  - Need a large number of cube reducibility tests.
    - Only a small portion will pass the test to generate reduced cubes.
    - Identical primes may be generated multiple times.
  - Size of the prime implicant table is large since each row corresponds to minterms
- **Improvements**
  - Extract all the prime implicants directly without enumerating minterms.
  - Generate a *reduced prime implicant table* and solve the minimum covering problem on this smaller table.

Direct Extraction of Prime Implicants (Preperation 1)

- ***Corollary 1*** : Let P be a cover for a *completely specified* function f. For any implicant c of f, there exists $c' \in P$ such that $c \subseteq c'$ if and only if P includes all primes of f.
- ***Theorem 1*** : Let P_f and P_g be the covers for *completely specified functions* f, and g, respectively. And let P_{fg} be $P_f \cdot P_g$ that is expanded in sum-of-product form. If P_f and P_g include all primes for f and g, respectively, then P_{fg} includes all primes of function $f \cdot g$.
- **Proof** :
  - By definition, P_{fg} is a cover whose cube elements are the non-zero conjunctions of a cube in P_f and cube in P_g;

$$P_{fg} = \{c_f \cdot c_g \mid c_f \in P_f, c_g \in P_g, c_f \cdot c_g \neq 0\}.$$

  - Any implicant of the function $f \cdot g$ is also an implicant for both f and g (if $f \cdot g$ is true, then both f and g must be true as well). Thus for any implicant c of $f \cdot g$, there exists $c_f \in P_f$ and $c_g \in P_g$ such that $c \subseteq c_f$ and $c \subseteq c_g$. Therefore $c \subseteq c_f \cdot c_g \in P_{fg}$.

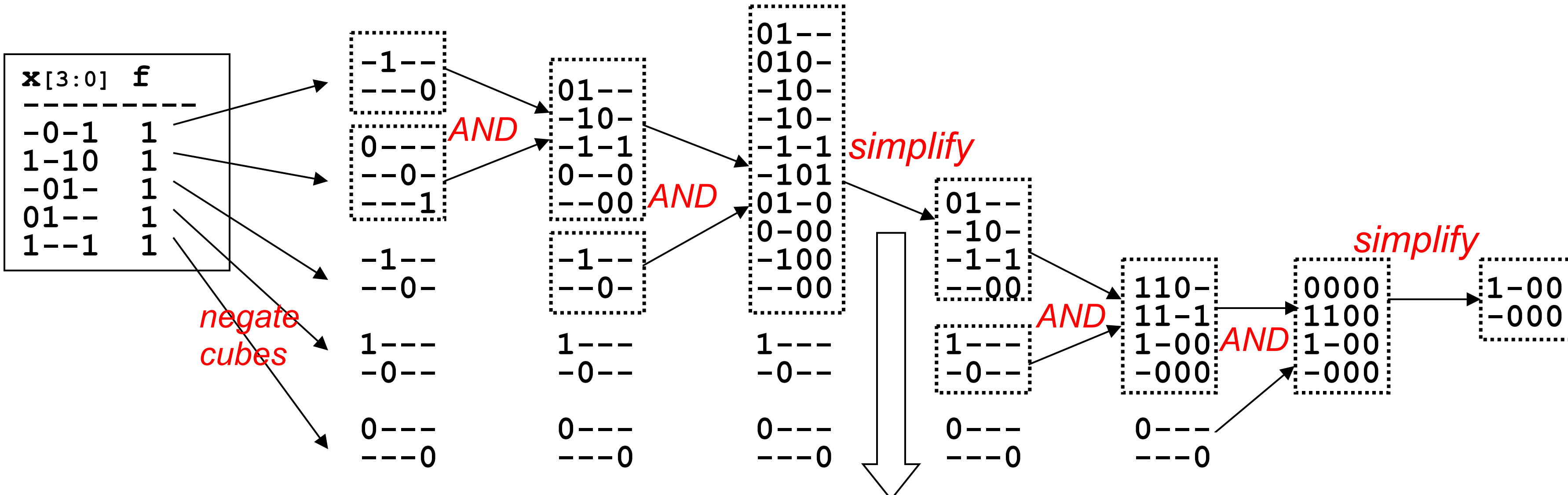
Direct Extraction of Prime Implicants (Preperation 2)

- ***Theorem 2***: Let P be a cover for a *completely specified* function f, and P' be the cover for the complement of f (denoted as $\overline{f}$) which is obtained by applying De-Morgan's Law to P and then expanding it to sum-of-product form. P' includes all primes of $\overline{f}$.

  (Let us call this the ***Negate-And-Expand Method***)
- ***Proof*** :
  - Let $P = c_0 + c_1 + ... + c_n$ (c_i is a cube)
  - By De-Morgan's Law : $\overline{P} = \overline{c_0 + c_1 + ... + c_n} = \overline{c}_0 \cdot \overline{c}_1 \cdot ... \cdot \overline{c}_n$ ---- (1)
  - A complement of a cube becomes a cover composed of single-literal cubes. Each single-literal cube is the prime of this cover.

    Ex. $\overline{x_0 \cdot \overline{x}_1 \cdot x_2} = \overline{x}_0 + x_1 + \overline{x}_2$
  - Since each term in eq(1) becomes a cover composed of primes for that cover, expanding these terms into sum-of-product form results in a cover composed of all primes of $\overline{f}$. (according to Theorem 1)

Negate-And-Expand Method



Single Cube Containment Minimality

For each cube in the list, if some other cube contains it, then delete this cube from the list.

Ex:

`01--` contains `010-` (delete `010-`)

`-1-1` contains `-101` (delete `-101`)

```
01--
010-  *   delete
-10-
-10-  *
-1-1
-101  *
01-0  *
0-00  *
-100  *
--00
```

Direct Extraction of Prime Implicants for Completely Specified Functions

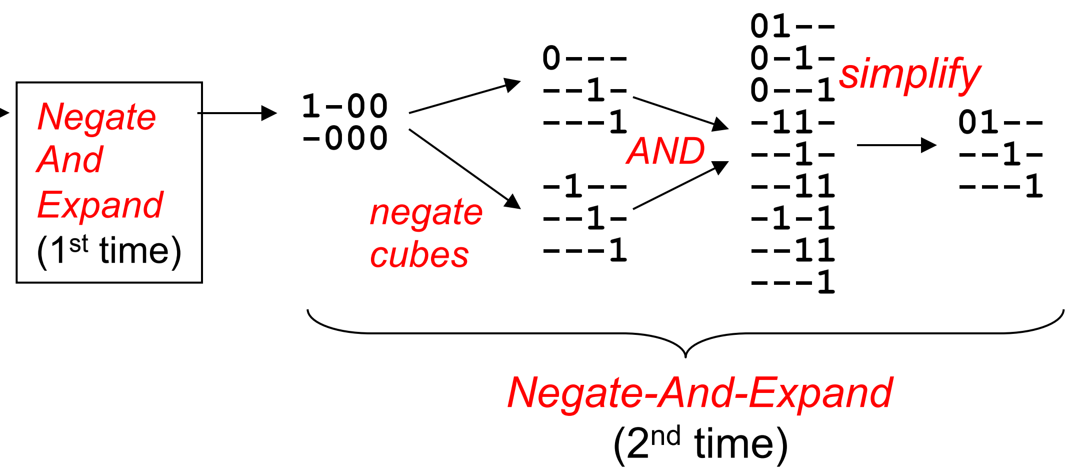
- By applying *Negate-And-Expand* twice on a cover for a *completely specified* function f, the obtained cover becomes the entire set of primes for f .

  Ex. $f = x_0 \overline{x}_2 + \overline{x}_0 \, x_1 \, x_3 + x_1 \, \overline{x}_2 + x_2 \, \overline{x}_3 + x_0 \, x_3$

truth table

x[3:0]	f
0000	0
0001	1
0010	1
0011	1
0100	1
0101	1
0110	1
0111	1
1000	0
1001	1
1010	1
1011	1
1100	0
1101	1
1110	1
1111	1

x[3:0]	f
-0-1	1
1-10	1
-01-	1
01--	1
1--1	1



Direct Extraction of Prime Implicants for *Incompletely* Specified Functions

- For an incompletely specified function f, apply the Negate-And-Expand operations twice on the cover containing both the ON-set and DC-set. The obtained cover includes all primes of f and possibly other implicants *which do not intersect with the ON-set*. (*DC-implicants*)

truth table

x[3:0]	f
0000	0
0001	1
0010	x
0011	1
0100	x
0101	x
0110	x
0111	x
1000	0
1001	1
1010	1
1011	1
1100	0
1101	x
1110	1
1111	x

x[3:0]	f	
-0-1	1	ON-set
1-10	1	ON-set
-01-	x	DC-set
01--	x	DC-set
1--1	x	DC-set

Negate-And-Expand (1st time)

```
1-00
-000
```

$R = \overline{F \cup D}$ (OFF-set cover)

Negate-And-Expand (2nd time)

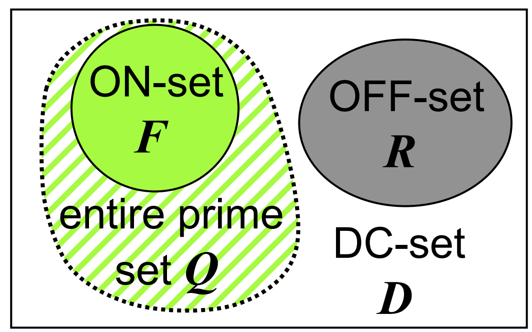
```
01--
--1-
---1
```

$Q' = F \cup D$ (Prime set + DC-implicants)

Eliminate DC-implicants

```
--1-
---1
```

Q (Prime set)



Function Negation Methods (1)

- Computation time of *Negate-And-Expand* operation can become very long when there are a large degree of redundancy in the cover representation of the function (i.e. a large number of small cubes).
- While the 2nd negation requires *Negate-And-Expand* operation in order to obtain the entire prime set, the obtained cover after the 1st negation (OFF-set cover) does not have to be the entire prime set for the negated function.
- *Shannon Expansion* method can be used for the 1st negation to obtain the OFF-set cover.
- The cover obtained by Shannon Expansion does not include all primes for the negated function, but its redundancy is relatively low. Also, the computational complexity is significantly lower than Negate-And-Expand Method.

Shannon Expansion

- f_{x_i} : cofactor of f with respect to factor x_i

  $$f_{x_i} = f(x_0, \ldots, x_{i-1}, 1, x_{i-1}, \ldots, x_{n-1}),\ f_{\bar{x}_i} = f(x_0, \ldots, x_{i-1}, 0, x_{i-1}, \ldots, x_{n-1})$$

  Ex : $f = a\,\bar{b}\,\bar{c} + a\,c\,\bar{d} + \bar{b}\,c\,d$

  $$f_a = \bar{b}\,\bar{c} + c\,\bar{d} + \bar{b}\,c\,d,\ f_{\bar{a}} = \bar{b}\,c\,d,\ f_{ac} = \bar{d} + \bar{b}\,d$$

- Shannon expansion : $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$
- Shannon expansion negation : $\overline{f} = x_i \overline{f_{x_i}} + \bar{x}_i \overline{f_{\bar{x}_i}}$
- Recursive Shannon expansion negation :

Ex : $f = a\,\bar{b}\,\bar{c} + a\,b\,c + \bar{b}\,c$

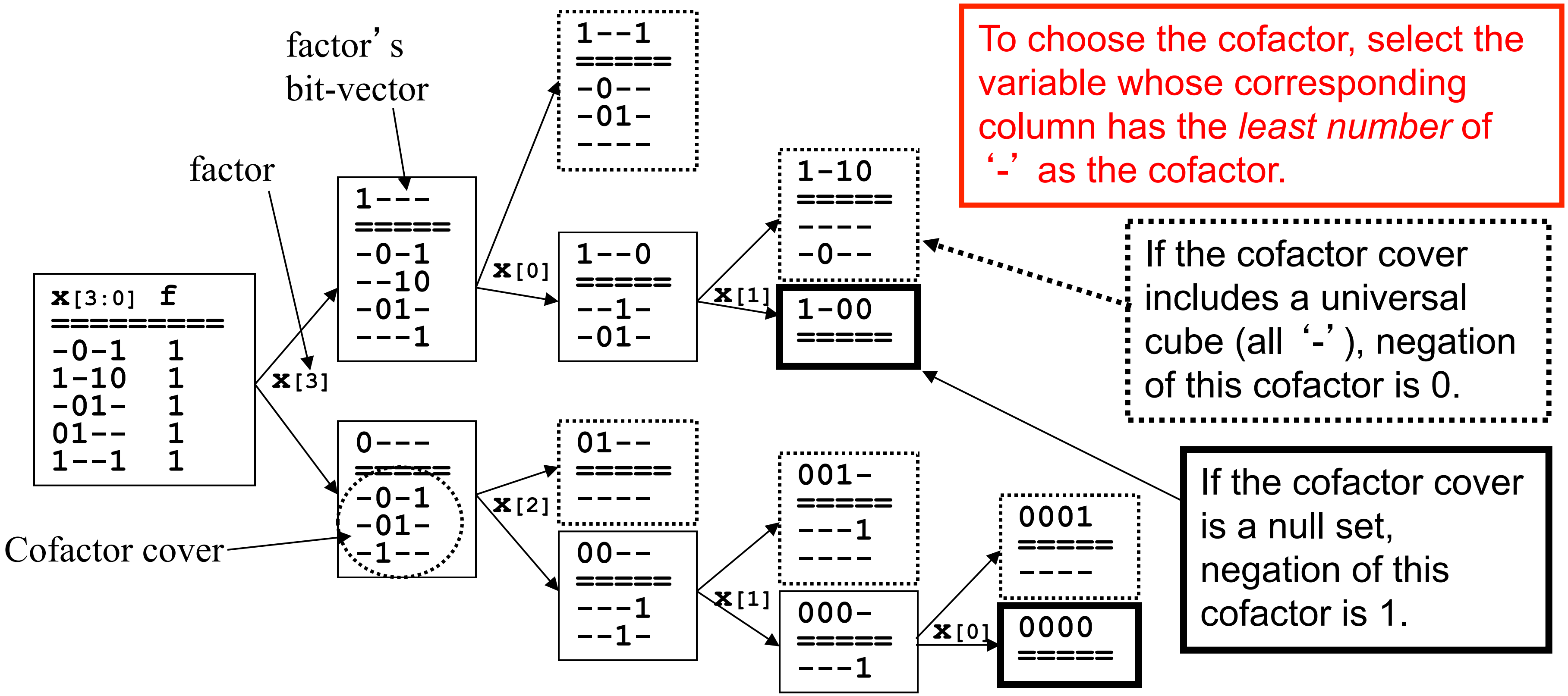
$$\overline{f} = a\,\overline{f_a} + \bar{a}\,\overline{f_{\bar{a}}} = a\,(\overline{\bar{b}\,\bar{c} + b\,c + \bar{b}\,c}) + \bar{a}\,(\overline{\bar{b}\,c})$$

$$\overline{f_a} = b\,\overline{f_{ab}} + \bar{b}\,\overline{f_{a\bar{b}}} = b\,(\bar{c}) + \bar{b}\,(\overline{\bar{c} + c}) = b\,\bar{c}$$

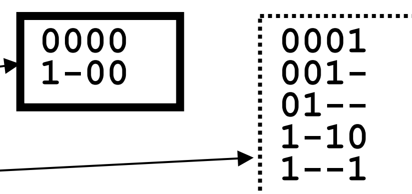
$$\overline{f_{\bar{a}}} = b\,\overline{f_{\bar{a}b}} + \bar{b}\,\overline{f_{\bar{a}\bar{b}}} = b\,(\bar{0}) + \bar{b}\,(\bar{c}) = b + \bar{b}\,\bar{c}$$

$$\overline{f} = a\,\overline{f_a} + \bar{a}\,\overline{f_{\bar{a}}} = a\,(b\,\bar{c}) + \bar{a}\,(b + \bar{b}\,\bar{c}) = a\,b\,\bar{c} + \bar{a}\,b + \bar{a}\,\bar{b}\,\bar{c}$$

Function Negation by Shannon Expansion (1)



- The set of factors whose leaf cofactor is 0 is equivalent to the OFF-set cover.
- The set of factors whose leaf cofactor is 1 is equivalent to the ON-set cover.

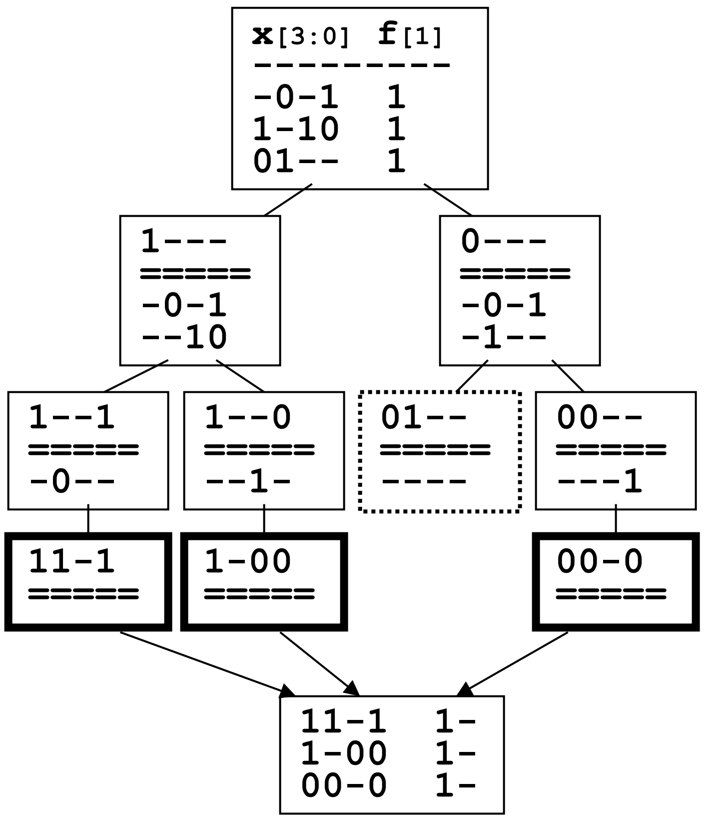


Shannon Expansion on Multiple-Output Function

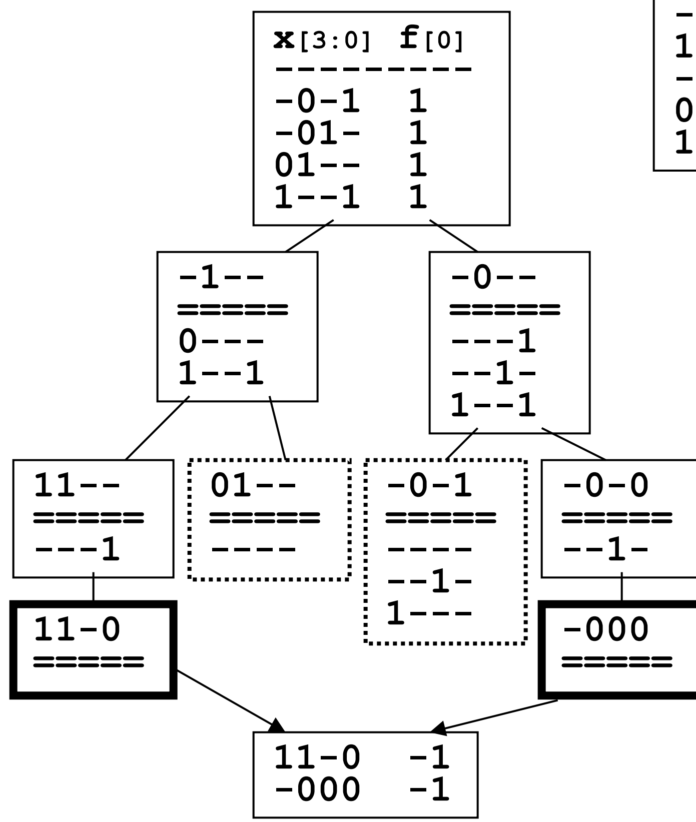
$$f_1 = x_0 x_2 + x_0 x_1 x_3 + x_2 x_3$$

$$f_0 = x_0 \overline{x_2} + x_1 \overline{x_2} + x_2 \overline{x_3} + x_0 x_3$$

x[3:0]	f[1:0]
-0-1	11
1-10	10
-01-	01
01--	11
1--1	01

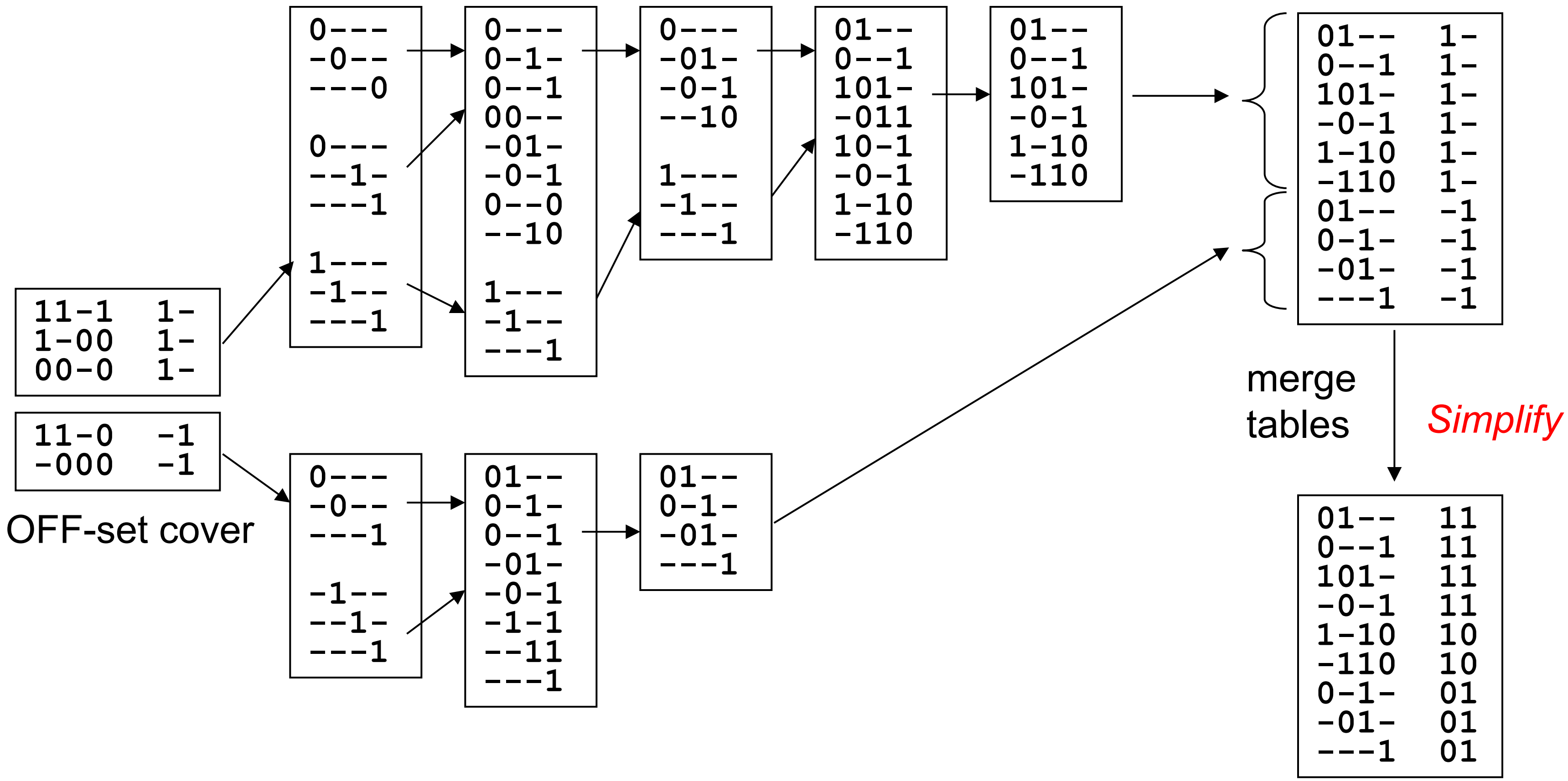


OFF-set cover for f_1



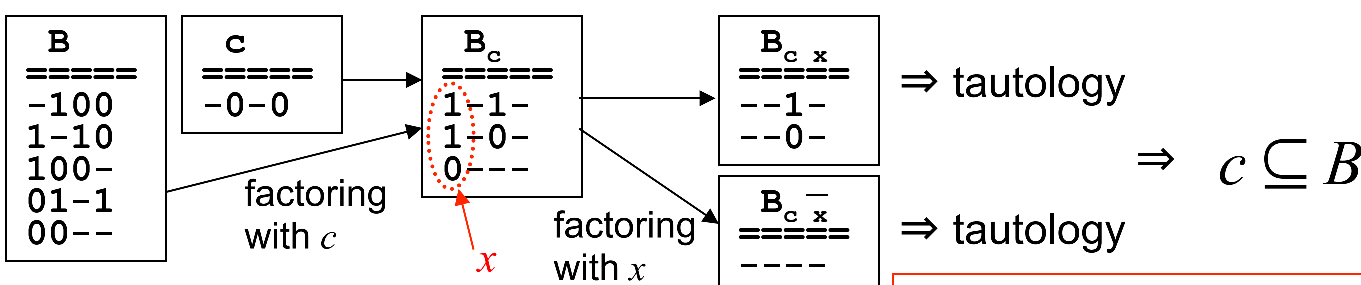
OFF-set cover for f_0

Negate-And-Expand Method for Multiple-Output Functions



Reduced Prime Implicant Table Generation (1)

- Essential prime set $E_r = \{c \mid c \in Q, F \not\subseteq Q - c\}$:
  - c is an essential prime if the prime set excluding c ("$Q - c$" denotes the set Q whose element c is eliminated) does not contain the ON-set cover F. (therefore c is *essential* for covering F)
  - Checking $F \cap c \not\subseteq Q - c$ (instead of $F \not\subseteq Q - c$) is sufficient.
- Containment check
  - $A \subseteq B \Leftrightarrow c \subseteq B \text{ for } \forall c \in A$ (A, B : cover, c : cube)
    - ✓ In order for a cover to be contained in another (partial order), all cube included in the former needs to be contained in the latter.
  - $c \subseteq B \Leftrightarrow B_c \equiv 1$ (B_c : cofactor of B with respect to cube c)
  - $B \equiv 1 \Leftrightarrow B_x \equiv 1 \wedge B_{\overline{x}} \equiv 1$ (tautology check by recursion)



Recall: $c \subseteq B \Rightarrow c \cdot B = c$

Reduced Prime Implicant Table Generation (2)

- Relatively redundant prime set $R_r = Q - E_r$
- Totally redundant prime set $R_t = \{c \mid c \in R_r,\ c \subseteq E_r\}$
- Partially redundant prime set $R_p = R_r - R_t$
- On obtaining a minimal prime set which covers the ON-set F
  - ✓ E_r is always included
  - ✓ R_t is never included
  - ✓ R_p is the portion of the total prime set which is considered in the minimum covering problem.
- ➢ Each element of R_p corresponds to the columns of the reduced prime implicant table.
- ➢ Minterm set M_p which needs to be covered（rows of the reduced prime implicant table）
  - ✓ $M_p = \overline{E}_r \cap R_p$
  - ✓ $m \cap E_r = \phi\ (m \in M_p)$

Reduced Prime Implicant Table Generation (3)

- Computation of minterm set M_p (*actually, each row may represent a collection of minterms*)
  - For each cube $c \in R_p$, consider the set $R' = R_p - c$.
  - Recursively divide c into smaller cubes at its don't-care variables
    - Ex. **0-1-** → **(001-,011-)** → **((0010,0011),(0110,0111))**
  - On each divided cubes c':
    - If $c' \subseteq E_r$, then c' is not included in M_p.
    - If there exists a cube $d \in R'$ such that $c' \subseteq d$, then all minterms included in c' is covered by the prime d. If so, add c' to the row and mark 'X' to all columns which contain c'. (Note that there may be several cubes which contain c')

    If one of the two conditions above is satisfied, then c' does not have to be divided anymore.

Reduced Prime Implicant Table Generation (4)

```
 E_r
=====
01--
10--
```

```
 R_p
=====
0-1-
-01-
-101
1-01
```

c = 0-1- →
$001\text{-} \subseteq \text{-}01\text{-} \in R'$
$011\text{-} \subseteq 01\text{--} \in E_r$

```
 R'
=====
-01-
-101
1-01
```

```
      | 0   -   -   1
      | -   0   1   -
      | 1   1   0   0
      | -   -   1   1
------|------------
001-  | x   x
```

c = -01- →
$001\text{-} \subseteq 0\text{-}1\text{-} \in R'$
$101\text{-} \subseteq 10\text{--} \in E_r$

```
 R'
=====
0-1-
-101
1-01
```

```
      | 0   -   -   1
      | -   0   1   -
      | 1   1   0   0
      | -   -   1   1
------|------------
001-  | x   x
```

c = -101 →
$0101 \subseteq 01\text{--} \in E_r$
$1101 \subseteq 1\text{-}01 \in R'$

```
 R'
=====
0-1-
-01-
1-01
```

```
      | 0   -   -   1
      | -   0   1   -
      | 1   1   0   0
      | -   -   1   1
------|------------
001-  | x   x
1101  |         x   x
```

c = 1-01 →
$1001 \subseteq 10\text{--} \in E_r$
$1101 \subseteq \text{-}101 \in R'$

```
 R'
=====
0-1-
-01-
-101
```

```
      | 0   -   -   1
      | -   0   1   -
      | 1   1   0   0
      | -   -   1   1
------|------------
001-  | x   x
1101  |         x   x
```

Reduced Prime Implicant Table Generation (5)

ON-set cover (initial)	*OFF-set cover*	*Prime set cover*	*Essential prime set*	*Minterms uncovered by E_r*
F	**R**	**Q**	$\mathbf{E_r}$	$\mathbf{M_p}$
-100	11-1	100-	100-	0010
1-10	1-11	1--0	1--0	0011
100-	000-	010-		0100
01-1	0110	-100		0101
001-		01-1		0111
0-11		001-		
		-010		
		0-11		

$\mathbf{R_p}$
=====
010-
-100
01-1
001-
-010
0-11

$\mathbf{R_t} = \phi$

Partially redundant prime set

Totally redundant prime set

Prime implicant table

min	100-	1--0	010-	-100	01-1	001-	-010	0-11
0010						x	x	
0011						x		x
0100			x	x				
0101			x		x			
0111					x			x
1000	x	x						
1001	x							
1010		x					x	
1100		x		x				
1110		x						

Reduced prime implicant table

$\mathbf{M_p}$	010-	-100	01-1	001-	-010	0-11
0010				x	x	
0011				x		x
0100	x	x				
0101	x		x			
0111			x			x

Reduced Prime Implicant Table Generation (6)

```
F
=====
-1001
1010-
11--1
011-0
1-11-
00-01
-11-1
```

```
R
=====
00-1-
00--0
100--
-001-
0-01-
11-00
--0-0
```

```
Q
=====
101--
1-11-
1-1-1
11--1
011--
-111-
-11-1
-1-01
0--01
--101
```

E_r
```
=====
101--
11--1
011--
0--01
```

R_p
```
=====
1-11-
-111-
```

R_t
```
=====
1-1-1
-11-1
-1-01
--101
```

M_p
```
=======
11110
```

Prime implicant table

min	101--	1-11-	1-1-1	11--1	011--	-111-	-11-1	-1-01	0--01	--101
00001									x	
00101									x	x
01001								x	x	
01100					x					
01101					x		x	x	x	x
01110					x	x				
01111					x	x	x			
10100	x									
10101	x		x							x
10110	x	x								
10111	x	x	x							
11001				x				x		
11011				x						
11101			x	x			x	x		x
11110		x				x				
11111		x	x	x		x	x			

Reduced prime implicant table

M_p	1-11-	-111-
11110	x	x

Summary on Two-Level Logic Optimization

- Two-level logic optimization is first proposed by Quine and McCluskey, and since then has been studied widely.
- Based on Quine-McCluskey method, improvements have been made in prime extraction, prime table generation, covering techniques to reduce the computation time.
- Even though the computational complexity is NP-complete (due to prime covering problem), near-optimal solution can be obtained in short time.
- There are heuristic algorithms which solve the prime extraction/prime covering problems iteratively.