多级逻辑优化

Multi-level Logic Synthesis

2025-5

主讲人：孙静翎

先回顾一下上节课的内容

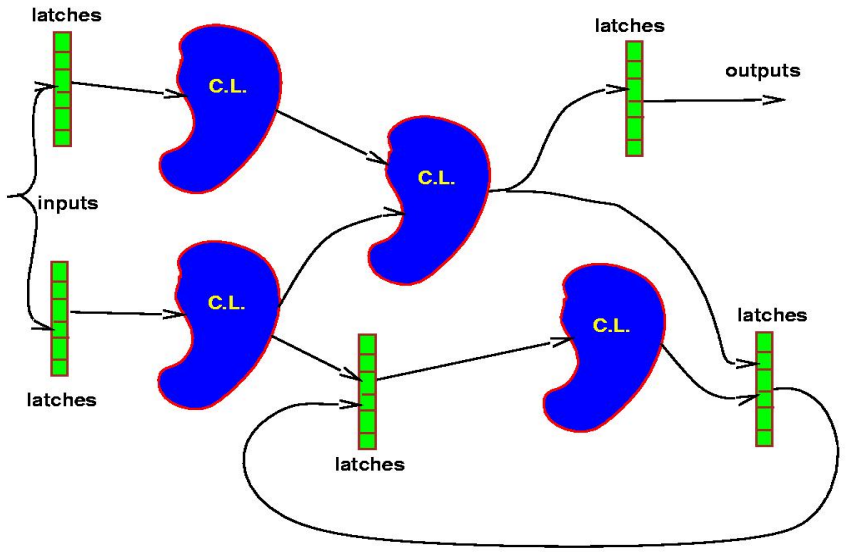
多级逻辑

Multi-level logic

- 逻辑网络
  - 由多个模块互连而成的结构
  - 每个模块由一个布尔函数建模
- 常见的限制条件：
  - 无环
  - 每个函数仅有一个输出、
- 结构由模块之间的连接关系决定

多级逻辑优化

Multi-level logic minimization



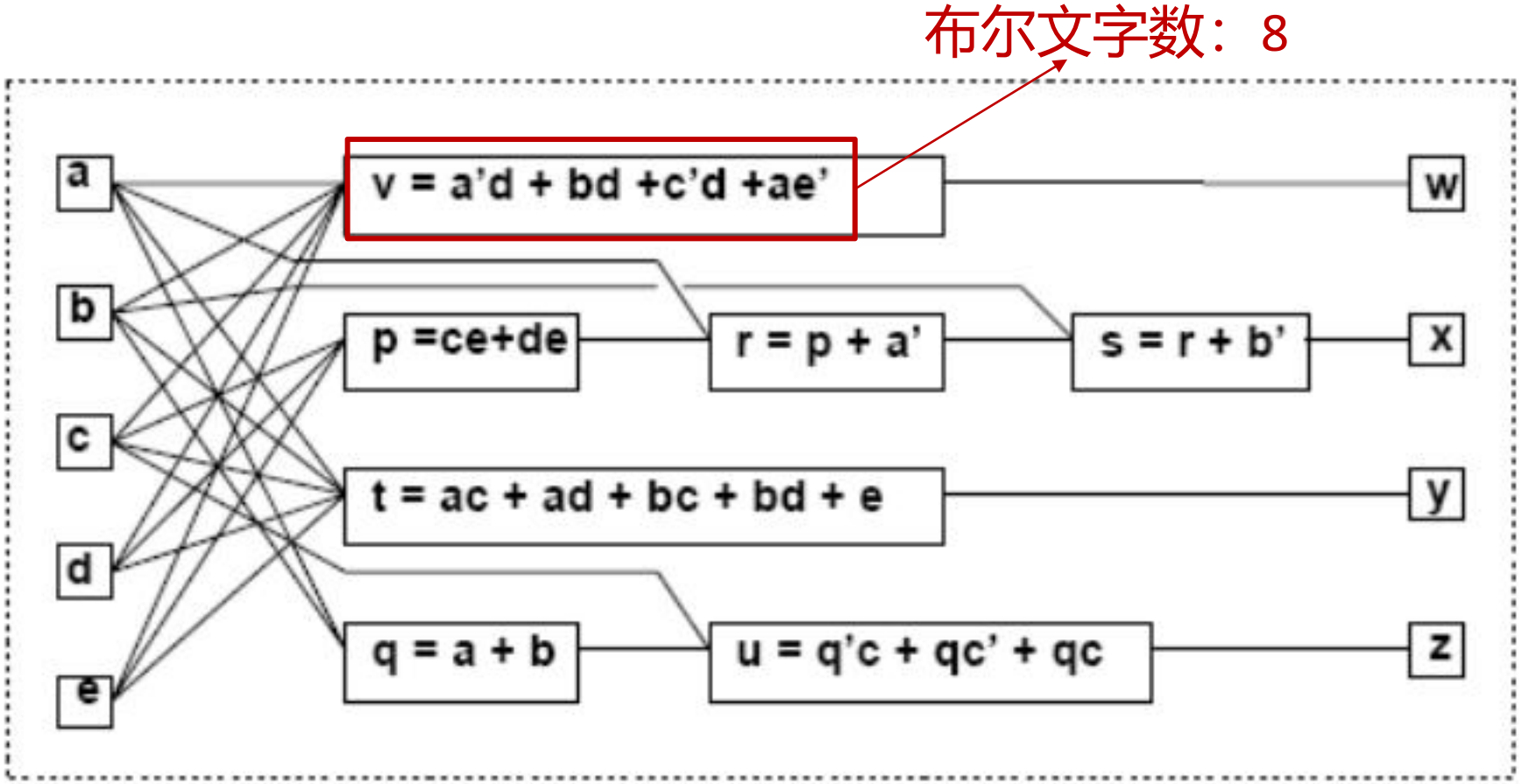
- 多级逻辑网络
  - 提供更多灵活性
  - 可在特定路径上基于不同的约束优化
  - 提升性能表现
- 逻辑综合的重要性随着面向半定制市场的代工厂的发展而同步提升

估算指标

Estimation

- 面积（Area）：
  - 使用布尔文字（literals）的数量来估算
    - 计算简单
    - 被广泛接受
    - 是一种良好的面积估算指标
- 延迟（Delay）：
  - 标准元件可以根据库文件直接得到
  - 非标准元件可以：
    - 根据级数（stages）粗略估计
    - 或根据逻辑函数的复杂度和扇出估算
- 功耗（Power）：
  - 功耗模型通常需要对电路所处的工作环境做出一定的假设
    - 每个节点的切换活动（switching activity）
    - 电容负载（capacitive loads）

一个多级逻辑网络的例子



问题分析

- 即使是最简单的问题，其计算本质也是困难的。
  - 例如，多输入单输出的网络问题
- 目前只提出了少数精确算法：
  - 计算复杂度很高
  - 仅适用于小规模电路
- 近似优化方法：
  - 基于规则的方法（Rule-based methods）
  - 启发式算法（Heuristic algorithms）

优化策略

逐步改进逻辑网络

- 通过电路变换来优化
- 保持网络的输入输出行为不变
- 如有需要，可利用环境中的“无关项” （don’t care 条件）

不同方法的差异体现在：

- 所采用的变换类型
- 变换的选择方式和应用顺序

变换方式总结

1.消除

- 执行变量替换操作

2.拆分

- 将一个函数拆分为更小的函数

3.提取

- 在两个（或多个）表达式中查找公共子表达式

4.替代

- 通过引入一个原本不属于原输入集的已有布尔函数来简化局部函数

5.化简

- 两级逻辑优化

估算指标

Optimization approaches

- 基于规则的方法
  (1) 建立规则数据库
  (2) 维护模式- 替换对
  (3) 按规则执行模式替换
- 算法式方法
  (1) 为每种转换类型定义算法
  (2) 算法充当网络上的操作符
  (3) 通过脚本依次调用这些算法
- 说明：现代综合工具主要采用算法式方法，但仍用规则处理特定问题。

布尔方法与代数方法

Boolean and algebraic methods

- 布尔代数
  - a) 取补运算
  - b) 对称分配律
  - c) 不关心集
- 代数建模思路
  - (1) 将布尔表达式视作多项式
  - (2) 使用乘积之和（Sum- of- Product, SOP）的形式
  - (3) 应用多项式代数

代数模型

Algebraic model

- 给定两个代数表达式

- 当满足下列条件时，表达式 $f_{除式}$ 可“整除” $f_{被除式}$，且 $f_{商} = f_{被除式} / f_{除式}$：
  (1) $f_{被除式} = f_{除式} * f_{商} + f_{余式}$
  (2) $f_{除式} * f_{商} \neq 0$
  (3) $f_{除式}$ 与 $f_{商}$ 所包含的变量集合互不相交

- 注意：$f_{商}$ 与 $f_{除式}$ 在定义上是可互换的

ALGEBRAIC_DIVISION(A, B)

{

对于 i 从 1 到 n //即依次遍历除式B中的每一项

{

D = { C^A_j | $C^A_j \supseteq C^B_i$ };

如果 D 为 ∅ 则返回 (∅, A);

Di = 用 D 中每一项删去 sup(C^B_i) 中的变量所得的结果;

如果 i == 1

则 Q = Di;

否则

Q = Q ∩ Di;

}

R = A – Q × B;

返回 (Q, R);

}

快筛规则

- 给定代数表达式 fi 和 fj，如果出现以下任一情况，则 fi / fj 的结果为空:

  1. fj 包含 fi 中未出现的变量
  2. fj 含有某个乘积，其变量集不被 fi 中任何乘积的变量集包含
  3. fj 的项数多于 fi
  4. fj 中某个变量的数量高于 fi

作业

- 假设被除式为f=abc+acd+abd+e，除式为g=ab+ad
- 请用代数除法计算f/g的商和余式应该是多少

ALGEBRAIC_DIVISION(A, B){

对于 i 从 1 到 n //即依次遍历除式B中的每一项 {

D = { C^A_j | $C^A_j \supseteq C^B_i$ };

如果 D 为 Ø 则返回 (Ø, A);

Di = 用 D 中每一项删去 sup(C^B_i) 中的变量所得的结果;

如果 i == 1 则 Q = Di;

否则 Q = Q ∩ Di; }

R = A – Q × B;

返回 (Q, R); }

作业

- A={abc,acd,abd,e},B={ab,ad}
- 依次遍历B中每一项：
  - 遍历到ab：
    - D为A中包含ab的项：{abc, abd}
    - Di为D中每一项减去ab剩余的内容：{c,d}
  - 遍历到ad:
    - D为A中包含ab的项：{acd,abd}
    - Di为D中每一项减去ab剩余的内容：{c,b}
  - 最终的商为第一次遍历时的Di与第一次遍历时的Di相交：
    - Q = {c,d} ∩ {c,b} = c
  - 余式R = A – Q × B：
    - abc+acd+abd+e-(ab+ad)*c = abd+e

变换方式总结

1.消除

- 执行变量替换操作

2.拆分

- 将一个函数拆分为更小的函数

3.提取

- 在两个（或多个）表达式中查找公共子表达式

4.替代

- 通过引入一个原本不属于原输入集的已有布尔函数来简化局部函数

5.化简

- 两级逻辑优化

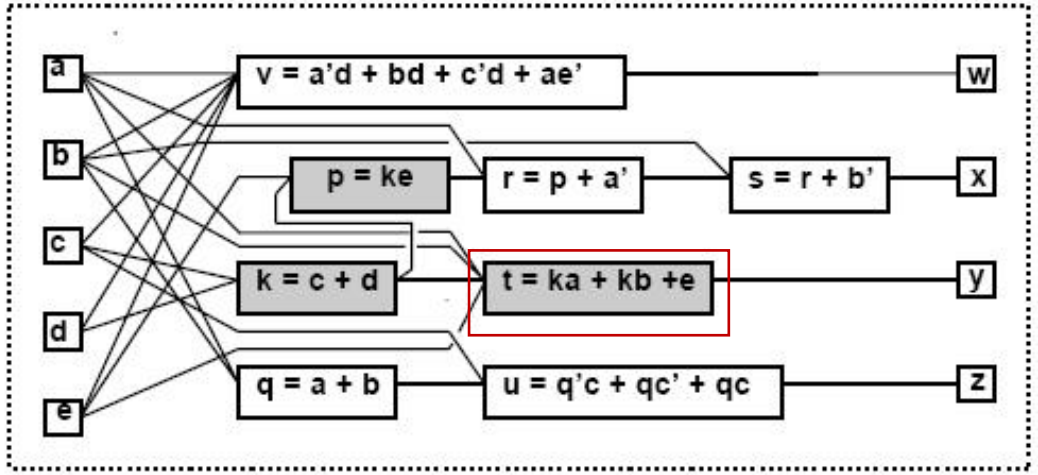
替代

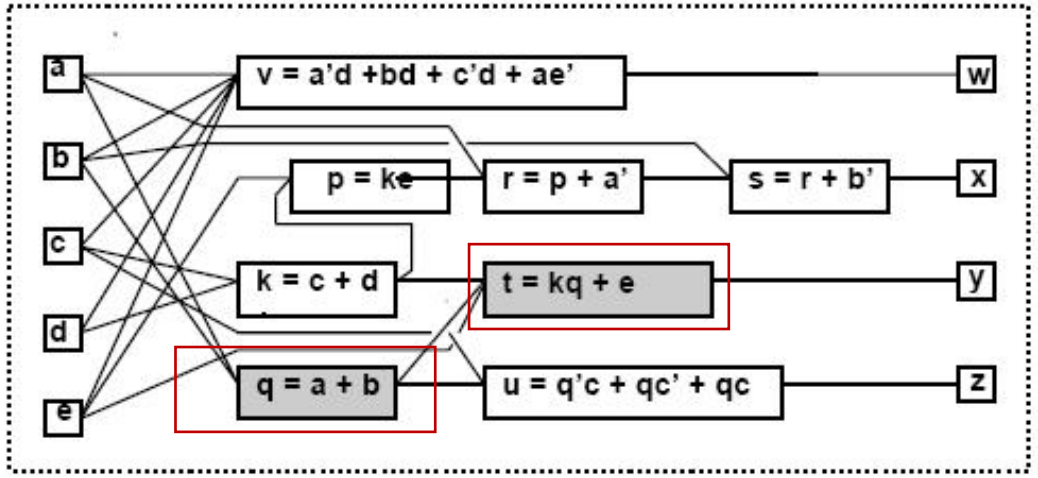
Substitution

- 通过引入一个原本不属于原输入集的附加输入来简化局部函数

- Example:
  - t = ka + kb + e;
  - t = kq + e;
  - Because q = a + b is already part of the network

例子


a
b
c
d
e
v = a'd + bd + c'd + ae'
p = ke
r = p + a'
s = r + b'
k = c + d
t = ka + kb +e
q = a + b
u = q'c + qc' + qc
w
x
y
z


a
b
c
d
e
v = a'd +bd + c'd + ae'
p = ke
r = p + a'
s = r + b'
k = c + d
t = kq + e
q = a + b
u = q'c + qc' + qc
w
x
y
z

代数替代

Algebraic substitution

- 选取一对表达式
- 以任意顺序对它们执行除法
- 若所得商（quotient）非空：
  - 评估面积收益和时间收益
  - 将 $f_{被除式}$ 替换为 $j * f_{商} + f_{余式}$，其中 j 是与 $f_{除式}$ 对应的变量
- 依据前述定理使用快筛规则，减少不必要的计算

```
SUBSTITUTE(Gₙ (V, E))
  {
      对于 i = 1, 2, ···, |V|
      {
          对于 j = 1, 2, ···, |V|（且 j ≠ i）
          {
              A = fᵢ 的乘积集合；
              B = fⱼ  的乘积集合；
              如果 (A, B) 通过快筛规则仍然保留
              {
                  (Q, R) = ALGEBRAIC_DIVISION(A, B);
                  如果 Q ≠ Ø
                  {
                      f商 = Q 中所有乘积之和;
                      f余式 = R 中所有乘积之和;
                      如果替换是有利的
                          fi = j * f商 + f余式;
                  }
              }
          }
      }
  }
```

变换方式总结

1.消除

- 执行变量替换操作

2.拆分

- 将一个函数拆分为更小的函数

3.提取

- 在两个（或多个）表达式中查找公共子表达式

4.替代

- 通过引入一个原本不属于原输入集的已有布尔函数来简化局部函数

5.化简

- 两级逻辑优化

定义

Definitions

- Cube-free（非乘积）表达式
  - 指无法再被任何项因式分解的表达式（即无法写成两个式子相乘的表达式）

- 例子:
  - a + bc 是Cube-free表达式
  - abc 以及 ab + ac 都不是Cube-free表达式

定义

Definitions

- 表达式的kernel（核）
  - 将表达式除以某个项（该项会被称为co-kernel，共核）后得到的Cube-free商，称为核
  - “共核”和“核”是一一对应的
- 表达式 f 的核集合记作 K(f)

核集的计算

Kernel set computation

- 朴素方法
  - 用变量集中所有子集（其幂集）的元素去除原函数。
  - 剔除非cube- free的商。

例子

f = ace + bce + de + g

核搜索：

- 用 a 除 f，得 ce，非Cube-free表达式
- 用 b 除 f，得 ce，非Cube-free表达式
- 用 c 除 f，得 ae + be，非Cube-free表达式
- 用 d 除 f，得 e，非Cube-free表达式
- 用 e 除 f，得 ac + bc + d，为Cube-free表达式 → 核
- 用 g 除 f，得 1，非Cube-free表达式
- 用 ab无法除f, 根据快筛规则
- …
- 用 ce 除 f，得 a + b，为Cube-free表达式 → 核
- …
- 用 1 除 f，得 ace + bce + de + g，为Cube-free表达式 → 核

- K(f) = { a + b，ac + bc + d，ace + bce + de + g }
- CoK(f) = { ce，e，1 }

核集的计算

Kernel set computation

- 朴素方法
  - 用变量集中所有子集（其幂集）的元素去除原函数。
  - 剔除非cube- free的商。

- 改进方法
  - 用递归算法减少计算
  - 利用乘法的交换律来减少重复计算。

正式开始本周的内容

递归算法

Recursive algorithm

- 递归算法是最早用于计算核集的方法，至今仍优于其他方案。
- 它基于的思想是：核集的核集仍为核集
- 例子：f = ace + bce + de + g
- {a + b，ac + bc + d，ace + bce + de + g }是ace + bce + de + g的核集
- {a + b，ac + bc + d}是ac + bc + d的核集

例子

f = ace + bce + de + g

核搜索：

- 用 a 除 f，得 ce，非Cube-free表达式
- 用 b 除 f，得 ce，非Cube-free表达式
- 用 c 除 f，得 ae + be，非Cube-free表达式
- 用 d 除 f，得 e，非Cube-free表达式
- 用 e 除 f，得 ac + bc + d，为Cube-free表达式 → 核
- 用 g 除 f，得 1，非Cube-free表达式
- 用 ab无法除f, 根据快筛规则
- …
- 用 ce 除 f，得 a + b，为Cube-free表达式 → 核
- …
- 用 1 除 f，得 ace + bce + de + g，为Cube-free表达式 → 核

- K(f) = { a + b，ac + bc + d，ace + bce + de + g }
- CoK(f) = { ce，e，1 }

递归算法

Recursive algorithm

- 递归算法是最早用于计算核集的方法，至今仍优于其他方案。
- 它基于的思想是：核集的核集仍为核集
- 例子：f = ace + bce + de + g
- {a + b，ac + bc + d，ace + bce + de + g }是ace + bce + de + g的核集
- {a + b，ac + bc + d}是ac + bc + d的核集

递归算法

Recursive algorithm

- 递归算法是最早用于计算核集的方法，至今仍优于其他方案。
- 它基于的思想是：核集的核集仍为核集
- 我们分两步说明它：
  - R_KERNELS（用于理解基本概念）
  - KERNELS（完整算法）

递归算法

Recursive algorithm

- 算法依赖一个子程序:

  CUBES(f, C) —— 返回 f 中所有文字包含乘积 C 的乘积集合。

- 示例:

  若 f = ace + bce + de + g，则

  CUBES(f, ce) = {ace , bce}

简单的递归算法

Simple recursive algorithm

```
R_KERNELS(f){
   K = Ø;
    foreach variable x ε sup(f){
          if (|CUBES(f,x)| ≥ 2) {
                C = maximal cube containing x,  s.t. CUBES(f,C) = CUBES(f,x);
                K = K U R_KERNELS(f / C);
          }
   }
    K = K U f;
   return(K);
}
```

简单的递归算法

Simple recursive algorithm

```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C =使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);
        }
    }
    K = K ∪ f;
    返回 K;
}
```

f = ace + bce + de + g

```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	
x	
CUBES(f,x)	
C	

f = ace + bce + de + g

```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b,c,d,e,g}
x	a
CUBES(f,x)	
C	

f = ace + bce + de + g

```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b,c,d,e,g}
x	a
CUBES(f,x)	{ace}
C	

f = ace + bce + de + g

```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b,c,d,e,g}
x	b
CUBES(f,x)	{bce}
C	

f = ace + bce + de + g

```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

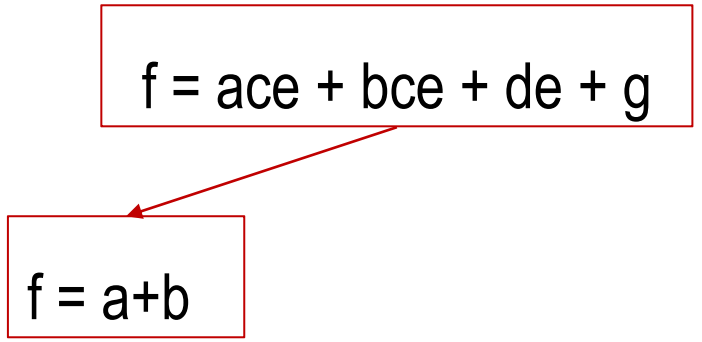
变量	值
K	{}
sup(f)	{a,b,c,d,e,g}
x	c
CUBES(f,x)	{ace, bce}
C	

f = ace + bce + de + g

```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b,c,d,e,g}
x	c
CUBES(f,x)	{ace, bce}
C	ce

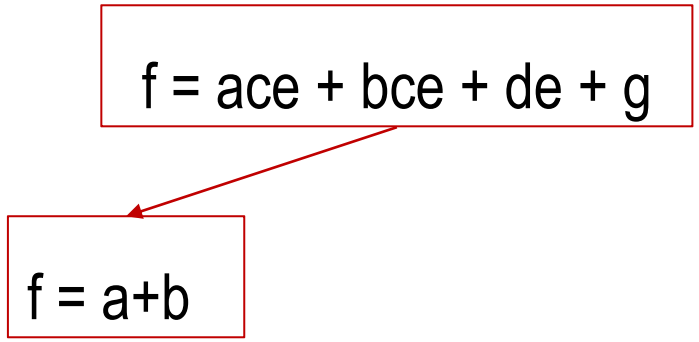
f = a+b




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b,c,d,e,g}
x	c
CUBES(f,x)	{ace, bce}
C	ce
f/C	a+b

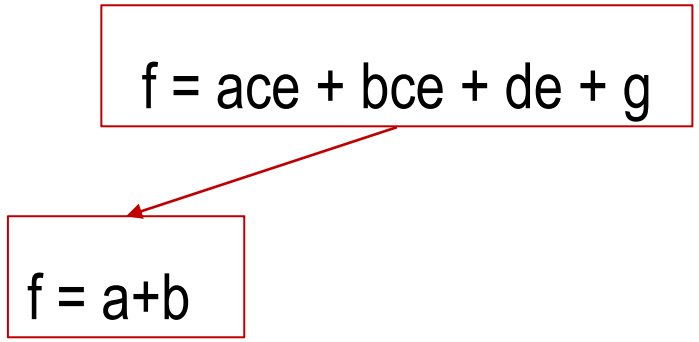
f = a+b




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b}
x	a
CUBES(f,x)	{a}
C	
f/C	

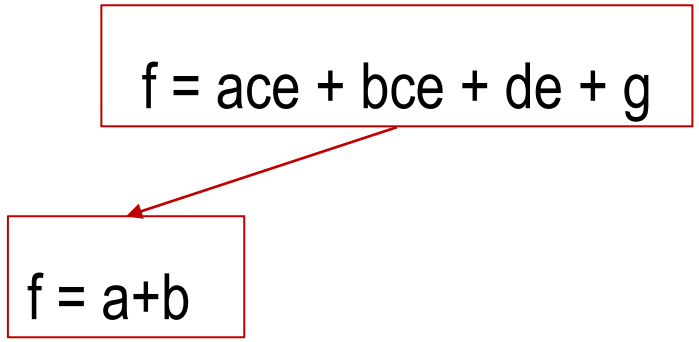
f = a+b




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b}
x	b
CUBES(f,x)	{b}
C	
f/C	

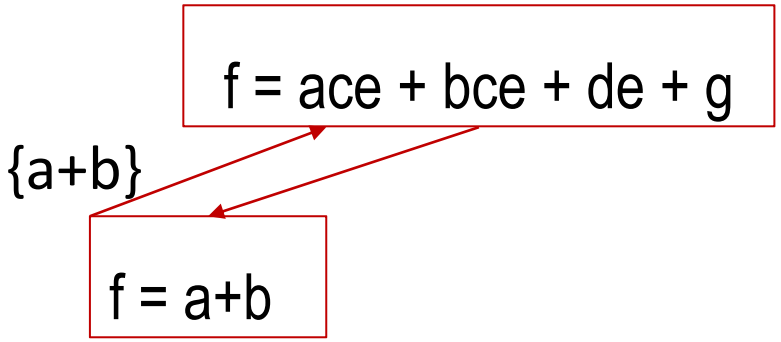
f = a+b




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b}
sup(f)	{a,b}
x	b
CUBES(f,x)	{b}
C	
f/C	

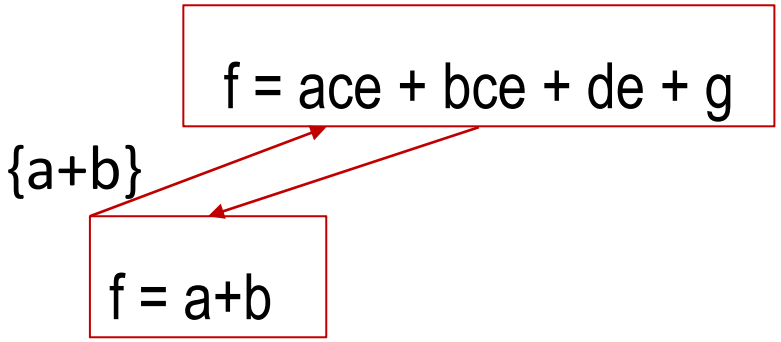
f = ace + bce + de + g




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b}
sup(f)	{a,b,c,d,e,g}
x	c
CUBES(f,x)	{ace, bce}
C	ce
f/C	a+b

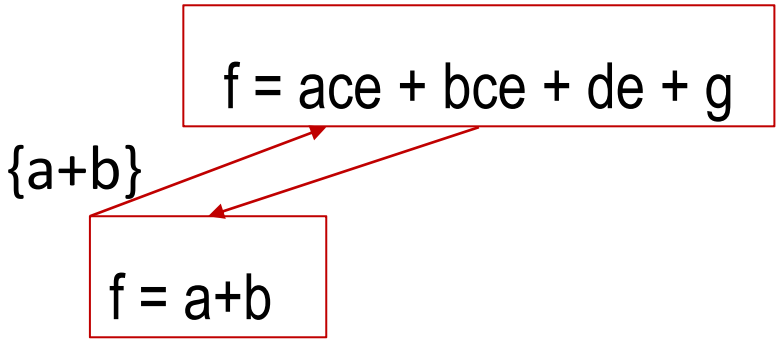
f = ace + bce + de + g




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b}
sup(f)	{a,b,c,d,e,g}
x	d
CUBES(f,x)	{de}
C	ce
f/C	a+b

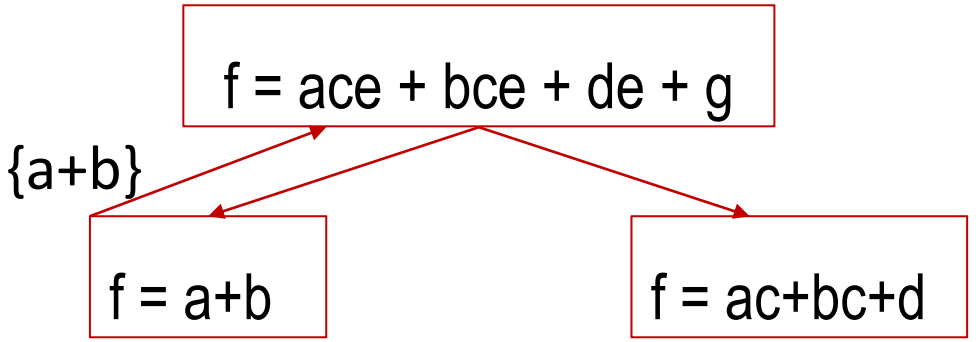
f = ace + bce + de + g




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b}
sup(f)	{a,b,c,d,e,g}
x	e
CUBES(f,x)	{ace, bce, de}
C	ce
f/C	a+b

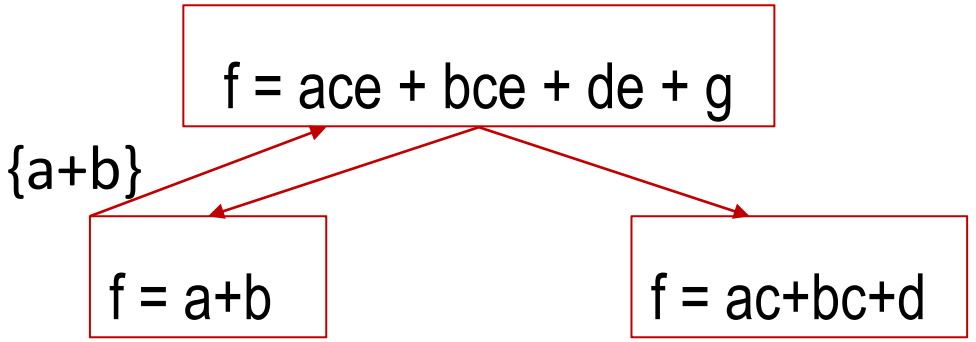
f = ace + bce + de + g




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b}
sup(f)	{a,b,c,d,e,g}
x	e
CUBES(f,x)	{ace, bce, de}
C	e
f/C	ac+bc+d

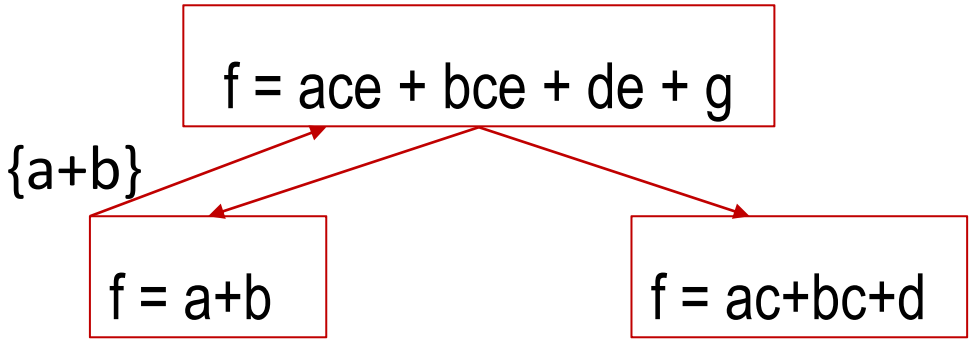
f = ac+bc+d




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b,c,d}
x	a
CUBES(f,x)	{ac}
C	
f/C	

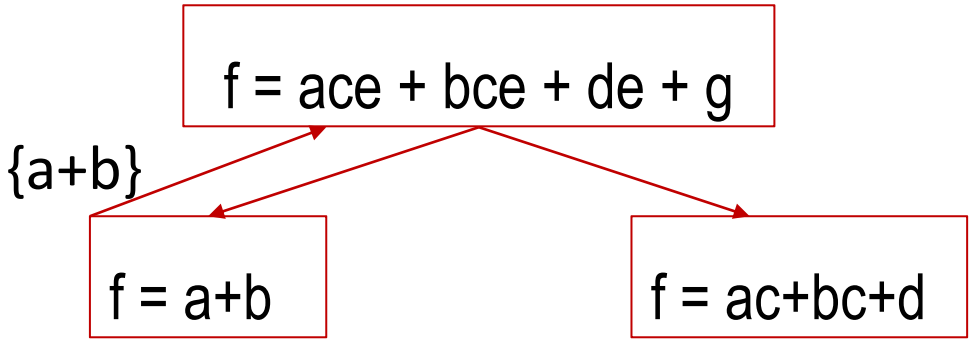
f = ac+bc+d




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b,c,d}
x	b
CUBES(f,x)	{bc}
C	
f/C	

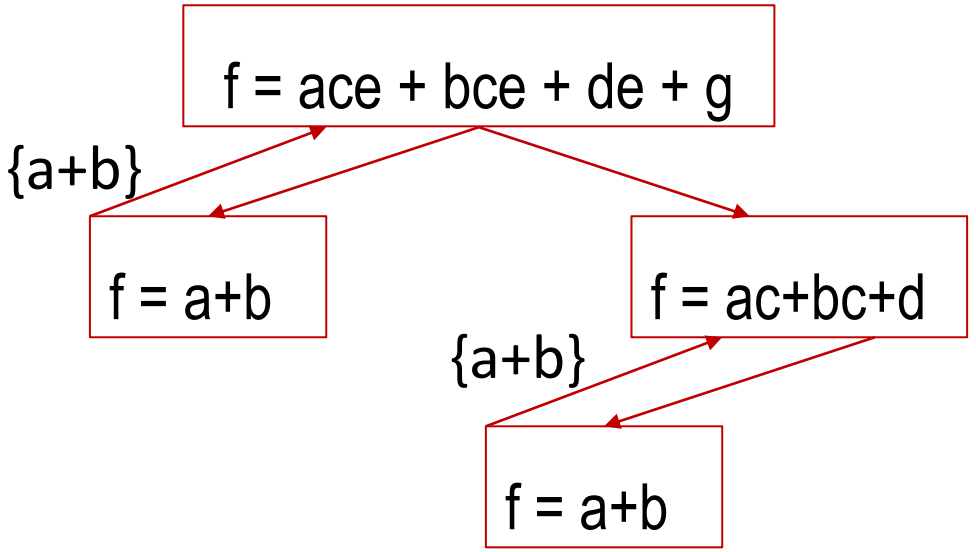
f = ac+bc+d




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{}
sup(f)	{a,b,c,d}
x	c
CUBES(f,x)	{ac, bc}
C	
f/C	

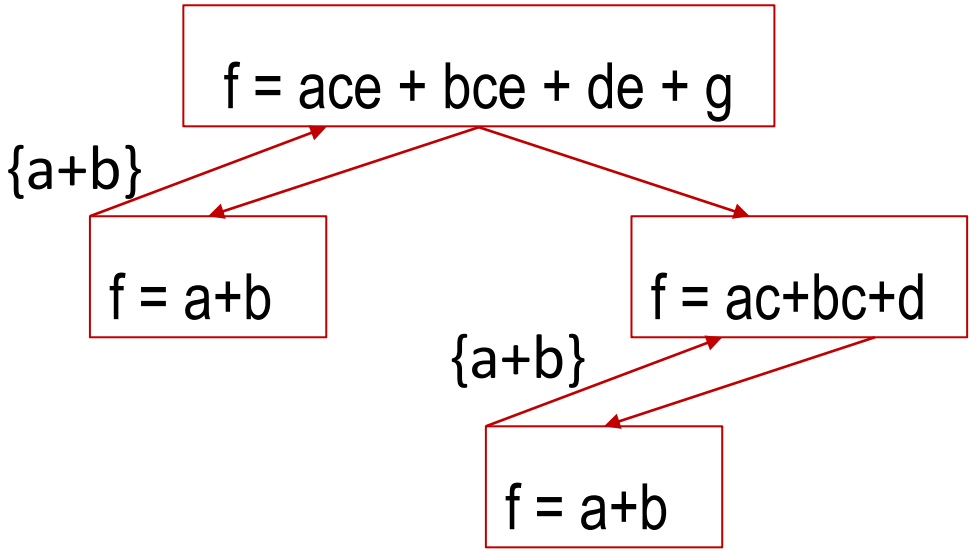
f = ac+bc+d




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b}
sup(f)	{a,b,c,d}
x	c
CUBES(f,x)	{ac, bc}
C	c
f/C	a+b

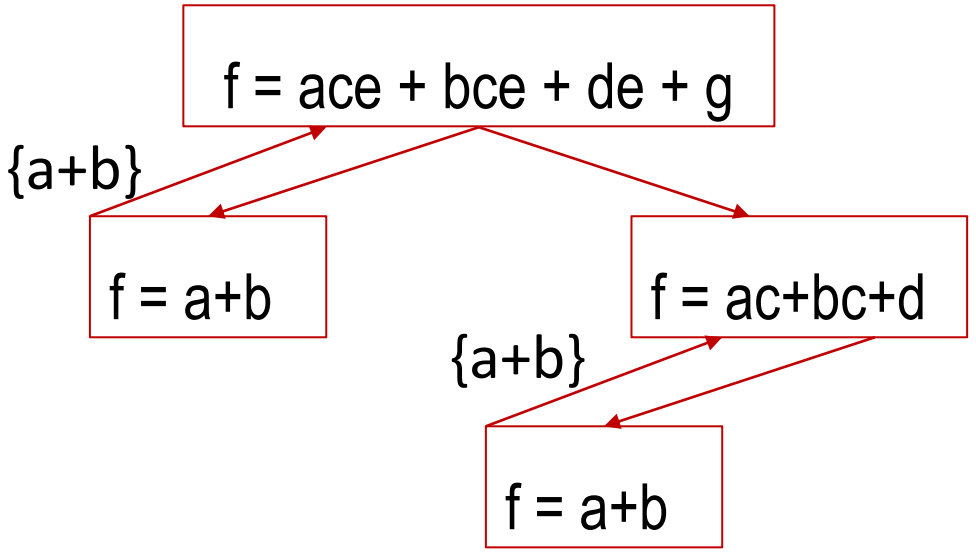
f = ac+bc+d




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b}
sup(f)	{a,b,c,d}
x	d
CUBES(f,x)	{d}
C	c
f/C	a+b

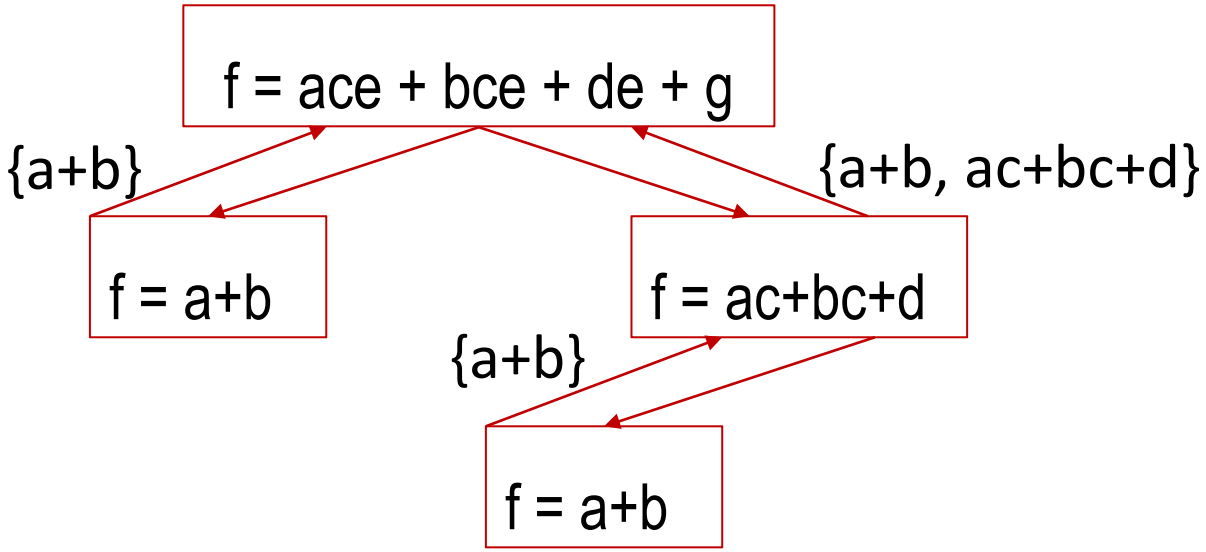
f = ac+bc+d




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b, ac+bc+d}
sup(f)	{a,b,c,d}
x	d
CUBES(f,x)	{d}
C	c
f/C	a+b

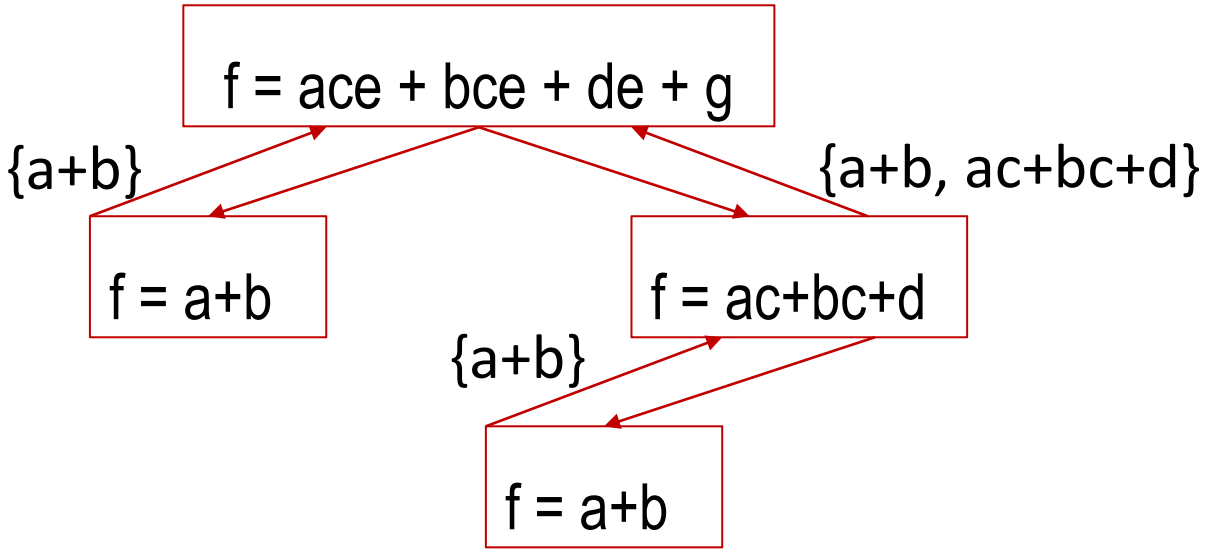
f = ace + bce + de + g




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b, ac+bc+d}
sup(f)	{a,b,c,d,e,g}
x	e
CUBES(f,x)	{ace, bce, de}
C	e
f/C	ac+bc+d

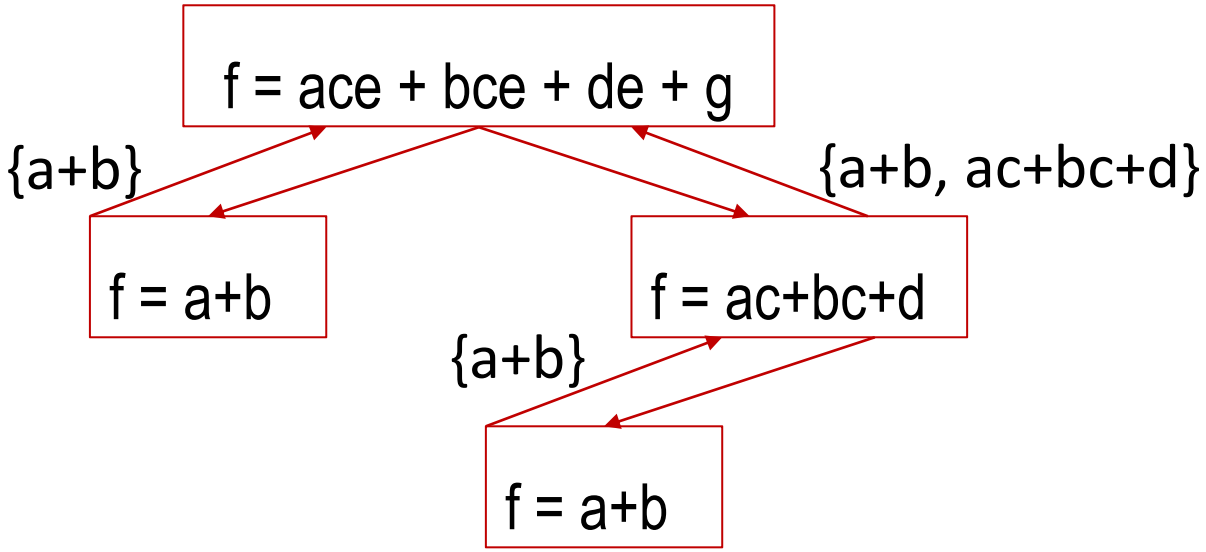
f = ace + bce + de + g




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b, ac+bc+d}
sup(f)	{a,b,c,d,e,g}
x	g
CUBES(f,x)	{g}
C	e
f/C	ac+bc+d

f = ace + bce + de + g




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b, ac+bc+d, ace + bce + de + g}
sup(f)	{a,b,c,d,e,g}
x	g
CUBES(f,x)	{g}
C	e
f/C	ac+bc+d

分析

- 递归算法在递归过程中会执行一些冗余计算

  示例:

  - 对 ce做除法
  - 先对 e做除法，再对 c 做除法

    结果会得到重复的 kernel
- 改进:
  - 利用乘法的交换律
  - 为已使用的文字保留一个指针

完整的递归核计算

Recursive kernel computation

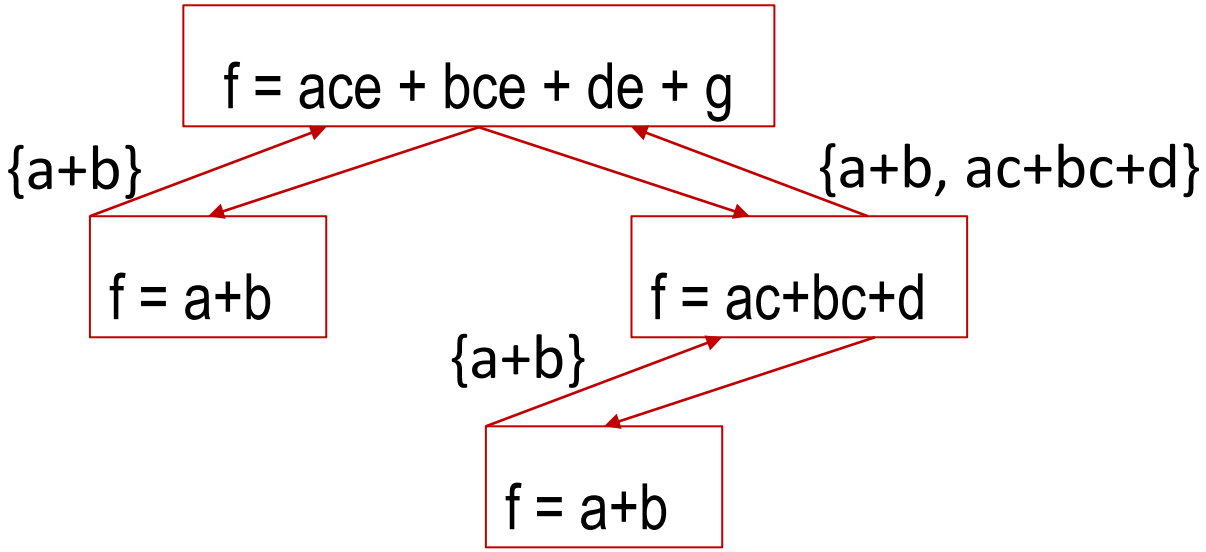
```
KERNELS(f, j){
        K = Ø;
        for i = j to n  {
                if (|CUBES(f,x_i)| ≥ 2) {
                        C = maximal cube containing x_i,
                        s.t. CUBES(f,C) = CUBES(f,x_i);
                        if (C has no variable x_k , k < i )
                                K = K U KERNELS( f / C ,i+1);
                }
        }
        K = K U f;
        return(K);
}
```

完整的递归核计算

Recursive kernel computation

```
KERNELS(f, j) {
    K = Ø;
    对于 i = j 到 n {
        如果 |CUBES(f, xi)| ≥ 2 {
            C = 最大乘积，满足 xi ∈ C 且 CUBES(f, C) = CUBES(f, xi);
            如果 C 不含 xk, k < i
                K = K ∪ KERNELS(f / C, i + 1);
        }
    }
    K = K ∪ f;
    返回(K);
}
```

f = ace + bce + de + g




```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

变量	值
K	{a+b, ac+bc+d, ace + bce + de + g}
sup(f)	{a,b,c,d,e,g}
x	g
CUBES(f,x)	{g}
C	e
f/C	ac+bc+d

核的矩阵表示

Matrix representation of kernels

- 关联矩阵（Incidence matrix）
  - 立方项 vs. 变量
- 矩形（Rectangle）
  - 所有元素均为 1 的若干行与列的子集
- 素矩形（Prime rectangle）
  - 不被包含在其他矩形中的矩形
- 共核（Co-kernel）
  - 至少包含两行的素矩形
- 示例：
  - 素矩形：({1,2}, {3,5}), ({1,2,3}, {5})
  - 共核：ce

	var	a	b	c	d	e	g
cube	$R\backslash C$	1	2	3	4	5	6
ace	1	1	0	1	0	1	0
bce	2	0	1	1	0	1	0
de	3	0	0	0	1	1	0
g	4	0	0	0	0	0	1

作业

- 请用穷举法、递归法或者矩阵法求出以下函数的所有核和共核:

$$F=abe+ab'ce+bce+cd$$

```
R_KERNELS(f){
    K = Ø;
    对 sup(f) 中的每个变量 x{
        如果 |CUBES(f,x)| ≥ 2 {
            C = 使得 CUBES(f,C) = CUBES(f,x)的含 x 的最大项;
            K = K ∪ R_KERNELS(f / C);}
    }
    K = K ∪ f;
    返回 K;}
```

作业

- 请用穷举法、递归法或者矩阵法求出以下函数的所有核和共核:

$$F=abe+ab'ce+bce+cd$$

```
KERNELS(f, j) {
    K = Ø;
    对于 i = j 到 n {
        如果 |CUBES(f, xi)| ≥ 2 {
            C = 最大乘积，满足 xi ∈ C 且 CUBES(f, C) = CUBES(f, xi);
            如果 C 不含 xk, k < i
                K = K ∪ KERNELS(f / C, i + 1);
        }
    }
    K = K ∪ f;
    返回(K);
}
```

变换方式总结

1.消除

- 执行变量替换操作

2.拆分

- 将一个函数拆分为更小的函数

3.提取

- 在两个（或多个）表达式中查找公共子表达式

4.替代

- 通过引入一个原本不属于原输入集的已有布尔函数来简化局部函数

5.化简

- 两级逻辑优化

利用代数法完成拆分操作

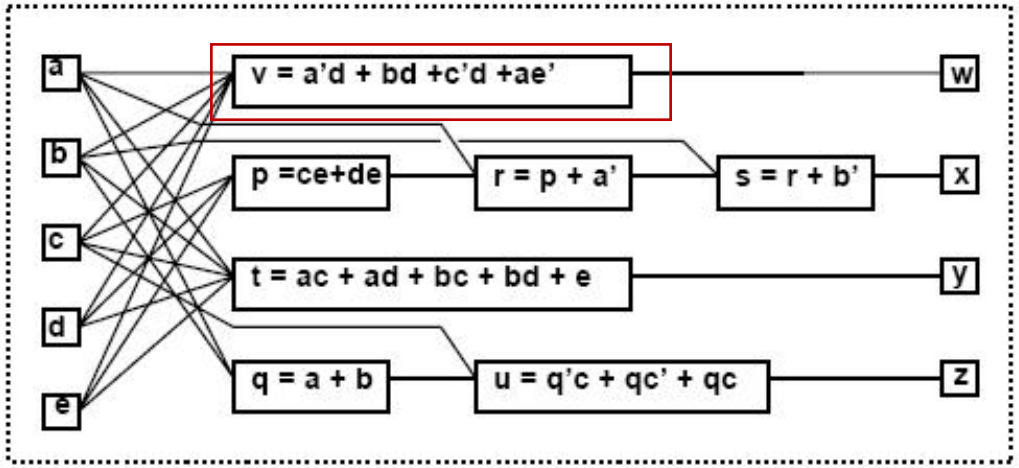
拆分

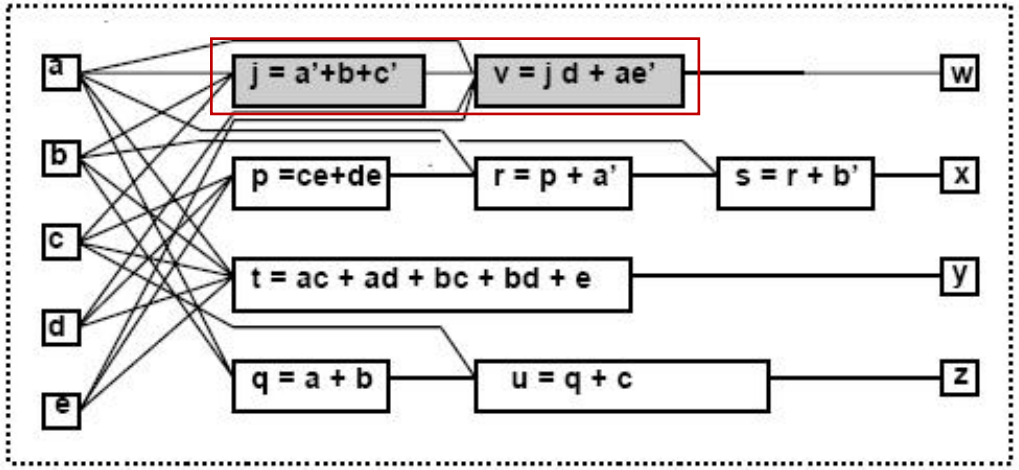
Decomposition

- 将一个函数拆分为更小的函数
  - 与“消除”操作相反
- 向网络中引入新的变量或模块

- Example:
  - v = a’d + bd +c’d +ae’
  - j = a’ + b + c’; v = jd + ae’;

例子


a
b
c
d
e
v = a'd + bd +c'd +ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q'c + qc' + qc
w
x
y
z


a
b
c
d
e
j = a'+b+c'
v = j d + ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q + c
w
x
y
z

基于核的拆分

Kernel-based decomposition

- 对多级逻辑中的每个结点
  - 找到除自己以外最大的核
  - 把核提取出来作为新的结点，改写原函数
  - 需要时可以对核进行进一步改写

例子

- 对 f = ace + bce + de + g，计算出：

K(f) = { a + b，ac + bc + d，ace + bce + de + g }

- 选取 kernel： ac + bc + d
  - 分解为： f = t e + g，　　t = ac + bc + d

- 对t = ac + bc + d，计算出：

K(f) = { a + b，ac + bc + d}

  - 选取 kernel： a + b
  - 将 t 分解为： t = s c + d，　　s = a + b

利用代数法完成提取操作

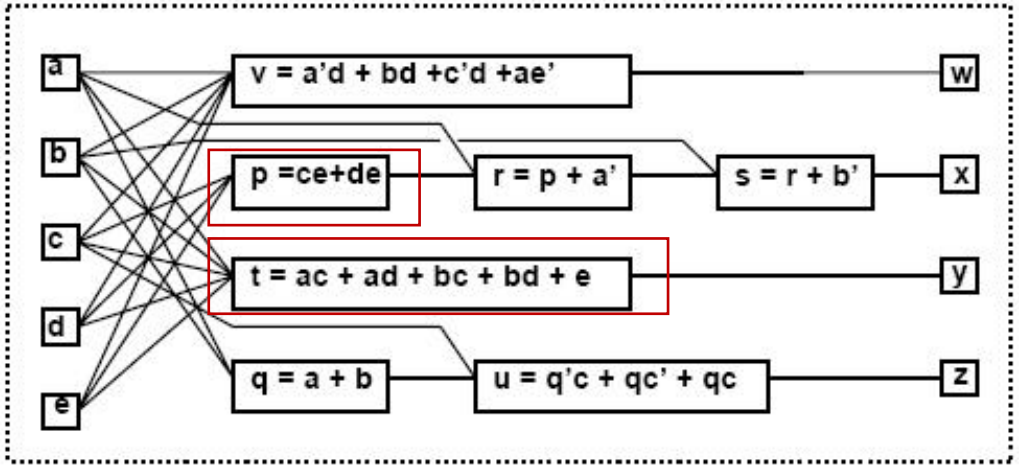
提取

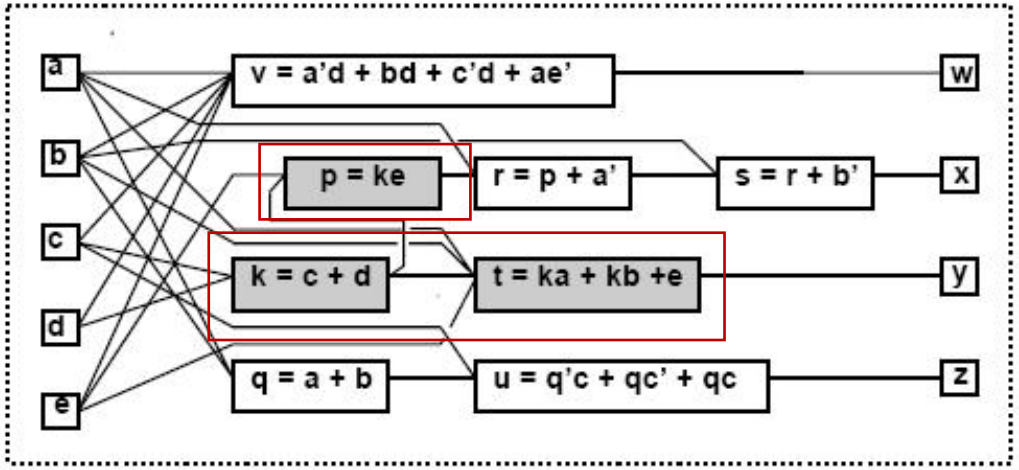
Extraction

- 在两个（或多个）表达式中查找公共子表达式
  - 提取该子表达式作为新的函数
  - 向电路中引入一个新的模块

- 例子：
  - p = ce + de; t = ac + ad + bc + bd + e;
  - p = (c + d) e; t = (c + d) (a + b) + e;
  - k = c + d; p = ke; t = ka + kb + e;

例子


a
b
c
d
e
v = a'd + bd +c'd +ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q'c + qc' + qc
w
x
y
z


a
b
c
d
e
v = a'd + bd + c'd + ae'
p = ke
r = p + a'
s = r + b'
k = c + d
t = ka + kb +e
q = a + b
u = q'c + qc' + qc
w
x
y
z

提取

extraction

- 单乘积子表达式：

  f=abc+ace；g=abd+abg

  k=ab；f=kc+ace；g=kd+kg

- 多乘积子表达式：

  p = ce + de; t = ac + ad + bc + bd + e;

  k = c + d; p = ke; t = ka + kb + e;

单乘积提取

Single-cube extraction

- 构造一个辅助表达式，它是所有局部表达式的并（求和）
- 找到最大的共核

- 反例：
- f=abcg+aceg；g=abd+abg
- h=abcg+aceg+abd+abg
- 最大共核是acg，对应的核是b+e

单乘积提取

Single-cube extraction

- 构造一个辅助表达式，它是所有局部表达式的并（求和）
- 找到最大的共核

- 对应的 kernel 必须同时属于两个（或更多）不同的表达式
- 使用额外变量对各表达式进行标记
- 提取选定的共核

例子

- $f_x = ace + bce + de + g$
- $f_y = ce + b$

•辅助函数:

- $f_{aux} = ace + bce + de + g + ce + b$

•标记后:

- $f_{aux} = xace + xbce + xde + xg + yce + yb$

•共核: ce； 核: xa+xb+y

•提取子表达式后:

- $f_z = ce$
- $f_x = za + bz + de + g$
- $f_y = z + b$

多乘积提取

Multiple-cube extraction

Brayton 和 McMullen定理

如果存在 ka ∈ K(fa) 与 kb ∈ K(fb)，且 ka ∩ kb 的结果为两个（或大于两个）乘积之和，则ka ∩ kb 为 fa 与 fb 的多乘积公共子表达式fd 。

- 用新的变量重新标记各乘积
- 在这些新变量中，核就成为新的乘积
- 等价地，我们只需要寻找辅助表达式（由重新标记后的所有表达式求和得到）的一个共核

例子

- f = ace + bce
  - K(f) = {(a+b)}
- g = ae + be + d
  - K(g) = {(a+b); (ae +be+d)}
- 重命名: x_a=a; x_b=b; x_{ae}=ae; x_{be}=be; x_d=d
  - 则 K(f) ={{x_a,x_b}} 且 K(g) = {{x_a,x_b},{x_{ae},x_{be},x_d}}
  - f_{aux} = f x_a x_b + g x_a x_b + g x_{ae} x_{be} x_d
  - CoK(f_{aux}) = x_a x_b
- 最终结果:
  - 从 f 和 g 中提取 (a + b)

变换方式总结

1.消除

- 执行变量替换操作

2.拆分

- 将一个函数拆分为更小的函数

3.提取

- 在两个（或多个）表达式中查找公共子表达式

4.替代

- 通过引入一个原本不属于原输入集的已有布尔函数来简化局部函数

5.化简

- 两级逻辑优化

功能验证

Functional Verification

2025-5

主讲人：孙静翎

功能验证

Decomposition

- 在典型的集成电路 （IC） 设计流程中，功能验证可确保实现符合规范。
- 功能验证至关重要，因为设计中未检测到的错误可能会给公司带来重大的经济损失。

奔腾FDIV Bug

Pentium FDIV Bug

- 起源
  - Intel 为奔腾 CPU 的浮点除法指令 FDIV 加入了一种新型的实现。依赖一个硬件查找表。
- 发现
  - 1994年，美国教授Thomas Nicely为研究孪生质数，在进行长除法运算时，发现处理器给出的答案总是错误的。

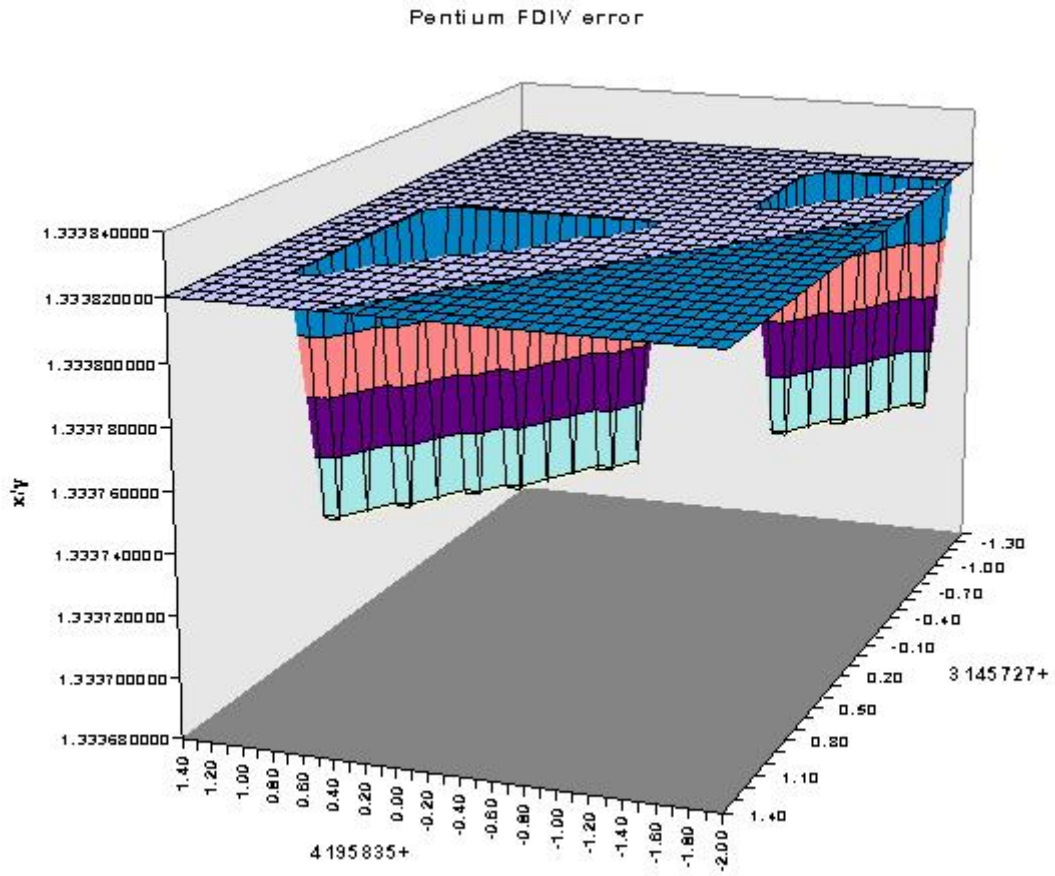


图1 Pentium FDIV Error，资料来源：pvs-studio

奔腾FDIV Bug

Pentium FDIV Bug

- 舆情
  - CNN 在电视节目上报道了奔腾 CPU 的这个 bug，将事态升级为了国民级新闻。
- 影响
  - 这是Intel历史上的首次产品召回。据估计，召回行动给英特尔带来了高达4.75亿美元的经济损失。

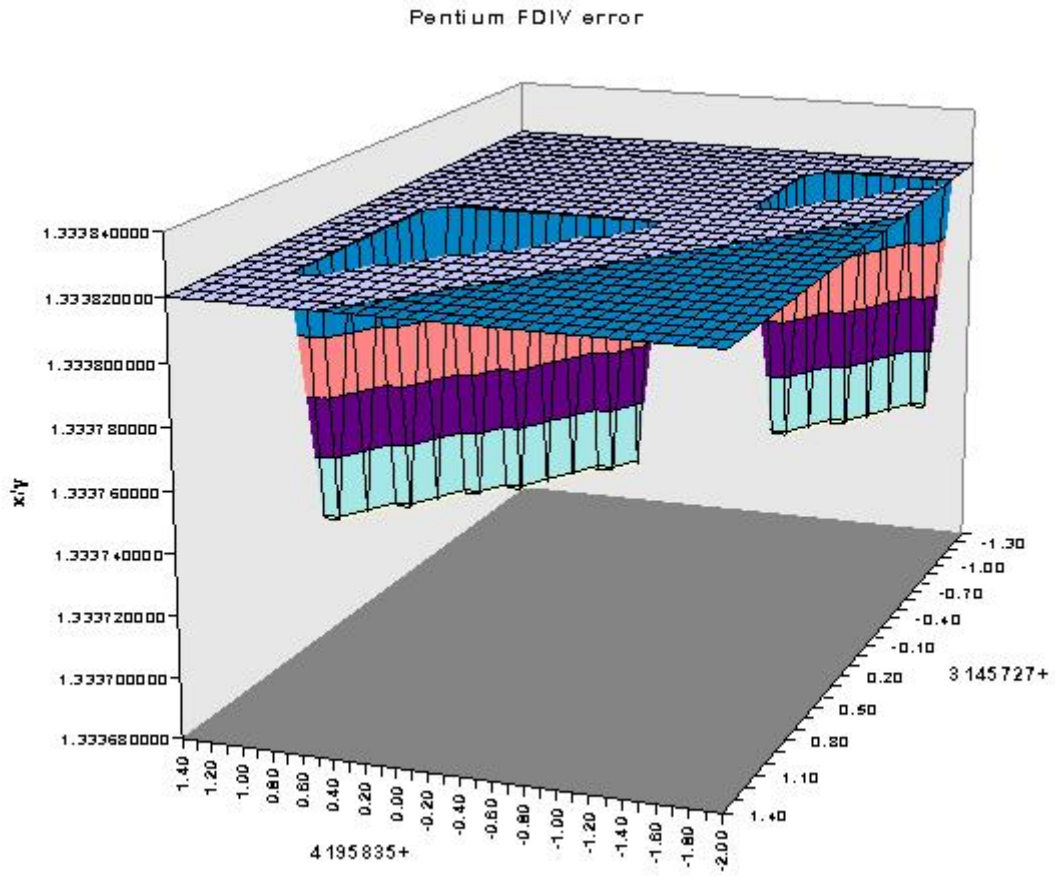


图1 Pentium FDIV Error，资料来源：pvs-studio

功能验证

Decomposition

- 在典型的集成电路 （IC） 设计流程中，功能验证可确保实现符合规范。
- 功能验证至关重要，因为设计中未检测到的错误可能会给公司带来重大的经济损失。
- 据统计，功能验证过程消耗了超过 70% 的设计工作，并且这个数字可能会继续增加。

功能验证

Decomposition

- 功能验证的基本流程
- 功能验证的级别
- 覆盖率指标
- 常用的功能验证技术

定义

Definition

- the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services or documents conform to specified requirements.

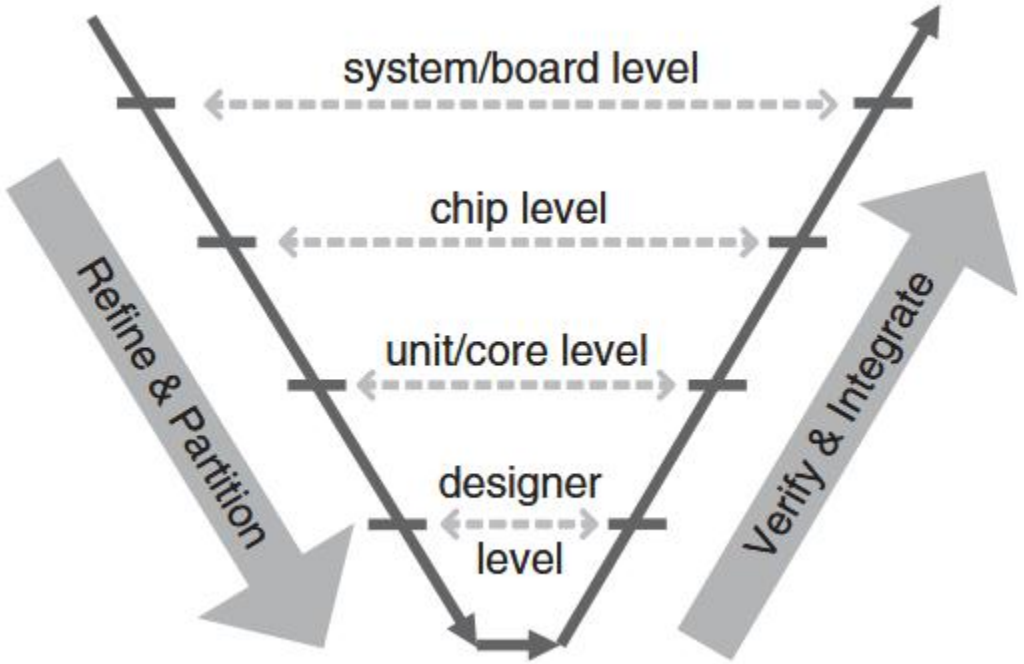
- 审查、检查、测试、检查、审计或以其他方式建立和记录项目、流程、服务或文件是否符合特定要求的行为

验证面临的主要挑战

- 1. 规范不明确
- 2.复杂性爆炸
- 3.充分性衡量

功能验证的级别

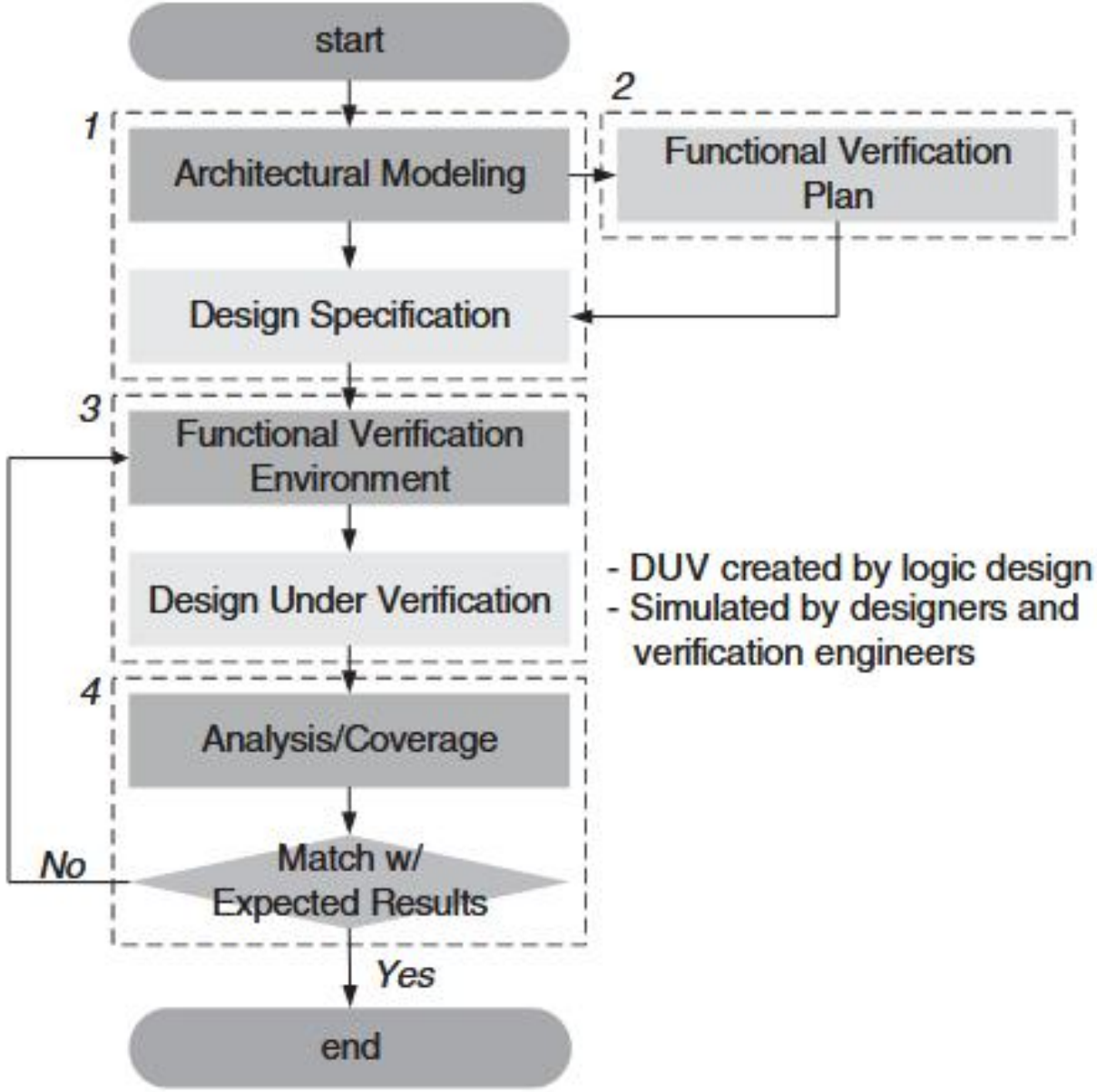
验证的层次



- 现代芯片设计采用模块化和层次化结构
- 验证也需依赖相应层次进行：
  - Designer-level verification

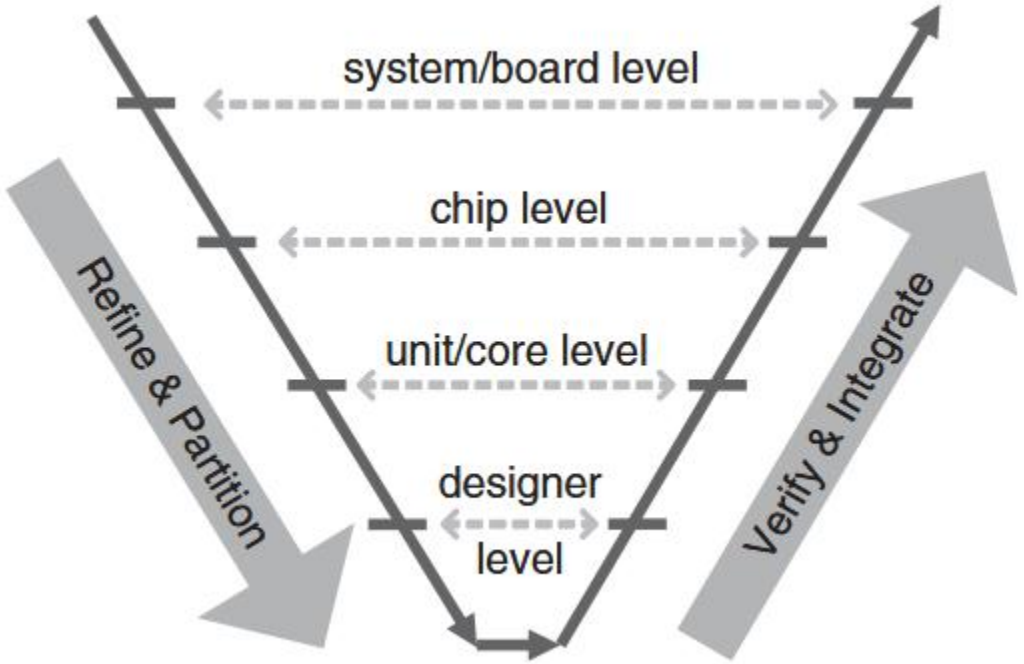
- 每个层次关注不同的行为与交互关系，验证策略应相匹配

每个层次的通用流程


start
1
Architectural Modeling
2
Functional Verification Plan
Design Specification
3
Functional Verification Environment
Design Under Verification
- DUV created by logic design
- Simulated by designers and verification engineers
4
Analysis/Coverage
Match w/ Expected Results
No
Yes
end

验证的层次

- 现代芯片设计采用模块化和层次化结构
- 验证也需依赖相应层次进行:
  - Designer-level verification
  - Unit-level verification
  - Core-level Verification
  - Subsystem-level Verification
  - Chip-level Verification



1. 设计者级验证

Designer-level verification

- 验证最小的 RTL 单元模块（如 FIFO、仲裁器）
- 由设计者个人负责，功能需求清晰，目标具体
- 不涉及模块间交互，适合早期并行开发

2.单元级验证

Unit-level verification

- Unit-level 是多个 RTL 模块组合形成的功能单元（如 ALU + 控制器）
- 验证跨模块交互、功能集成是否正确
- 是设计者级验证向 Core-level 验证过渡的桥梁
- 更注重功能协同而非单元行为

3.核心级验证

Core-level Verification

- 多个模块组合形成处理核心（如 CPU Core）
- 验证模块间接口交互与子功能协同
- 可复用已有单元级测试结果

4. 子系统级验证

Subsystem-level Verification

- 包含多个 core + 外设（如 memory、DMA、IO）
- 验证更复杂数据流、总线仲裁、共享资源
- 仿真时需关注高阶交互关系与死锁

5. 芯片级验证

Chip-level Verification

- 验证完整芯片设计（SoC）、多子系统集成
- 涉及封装接口、功耗、初始化序列等
- 目标是满足规格说明书的整体功能

6. 系统级验证

System-level Verification

- 验证芯片与软件栈（操作系统、中间件）的协同工作
- 通常通过仿真+系统原型板来进行
- 目标是端到端功能验证，如启动、加载、外设通信等

覆盖率指标

随机测试

Random testing

- 自动生成大量随机输入，以快速验证实现是否有错误

- 随机测试生成通常需要两种类型的输入来约束测试生成过程:
- （1） 用作测试用例框架的模板，其中包含一组未知的输入字段，
- （2） 一组参数，在生成过程中可以为其设置值。用户无需直接手动创建测试，而是在其合法范围内为输入字段指定这些参数。

随机测试

Random testing

- 例子：测试一个微处理器的指令执行单元：

- 测试模板：
  MUL <random R1–R4> <random R4–R8> <random R8–R20>

- 测试用例：

```
MUL R2, R5, R9
MUL R3, R7, R14
MUL R4, R4, R19
MUL R1, R6, R10
MUL R2, R8, R11
MUL R3, R5, R13
MUL R1, R7, R20
MUL R4, R6, R8
MUL R2, R4, R18
MUL R3, R8, R12
```

随机测试

Random testing

- 例子：测试一个微处理器的指令执行单元：

- 测试模板：
  <Pr(ADD) = 90%, Pr(SUB) = 10%> R3 R5 <random R4–R7>

- 测试用例：

```
ADD R3, R5, R4
ADD R3, R5, R6
ADD R3, R5, R7
ADD R3, R5, R5
ADD R3, R5, R4
ADD R3, R5, R7
ADD R3, R5, R6
ADD R3, R5, R4
ADD R3, R5, R5
SUB R3, R5, R7
```

随机测试

Random testing

- 自动生成大量随机输入，用于刺激设计

- 存在的问题：
  ❌难以生成“目标导向”的测试场景
- ❌难以触发罕见 bug（如 corner case）
- ❌覆盖效果波动大，验证效率低下

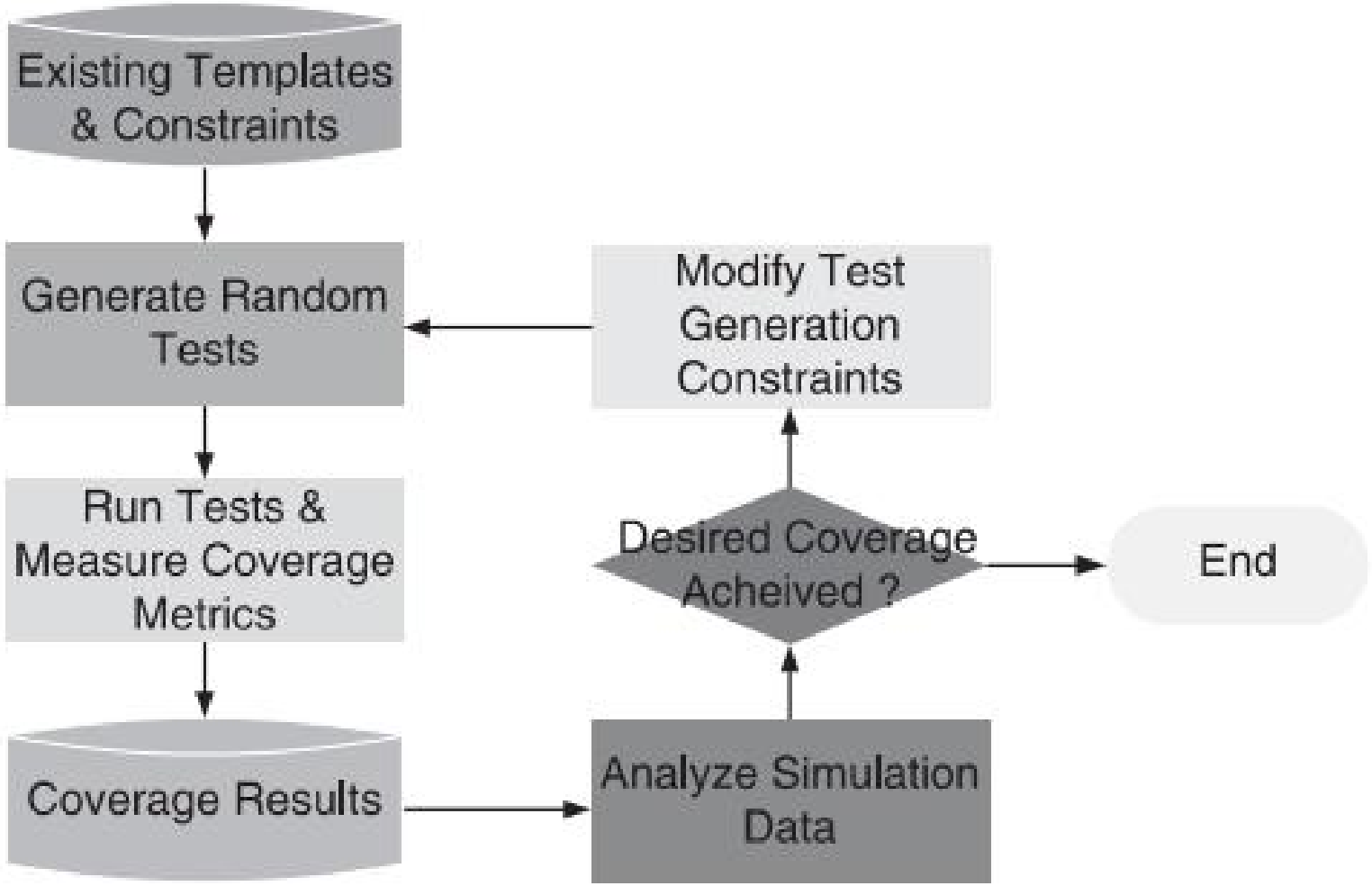
覆盖率驱动验证

Coverage-driven Verification

- CDV 是一种将覆盖率结果反馈给测试生成器的方法
- 基于“当前验证达成情况”，动态调整测试策略
- 属于功能覆盖率导向的验证方式
- 核心机制：
  - 仿真时记录功能覆盖信息（如断言命中、场景触发）
  - 根据“未覆盖功能点”，调整模板或 bias 参数生成新的测试
  - 最终目标是覆盖全部功能需求点（coverage closure）

覆盖率驱动验证

Coverage-driven Verification



结构覆盖率指标

Structural coverage metrics

- Line coverage (a.k.a. statement coverage) 行覆盖率
- Toggle coverage 翻转覆盖
- Branch/path coverage 分支/路径覆盖
- Expression coverage / condition coverage 表达式/条件覆盖率