Logic Optimization

Multi-level Logic Synthesis

Virendra Singh

Professor

Department of Electrical Engineering &

Dept. of Computer Science & Engineering

Indian Institute of Technology Bombay

http://www.ee.iitb.ac.in/~viren/

E-mail: viren@{ee, cse}.iitb.ac.in

EE-677: Foundations of VLSI CAD

Lecture 30 on 21 Oct 2021

CADSL

Logic Minimization

Heuristic

2 level minimization

AND-OR

NAND-NAND

Exact - Heuristic

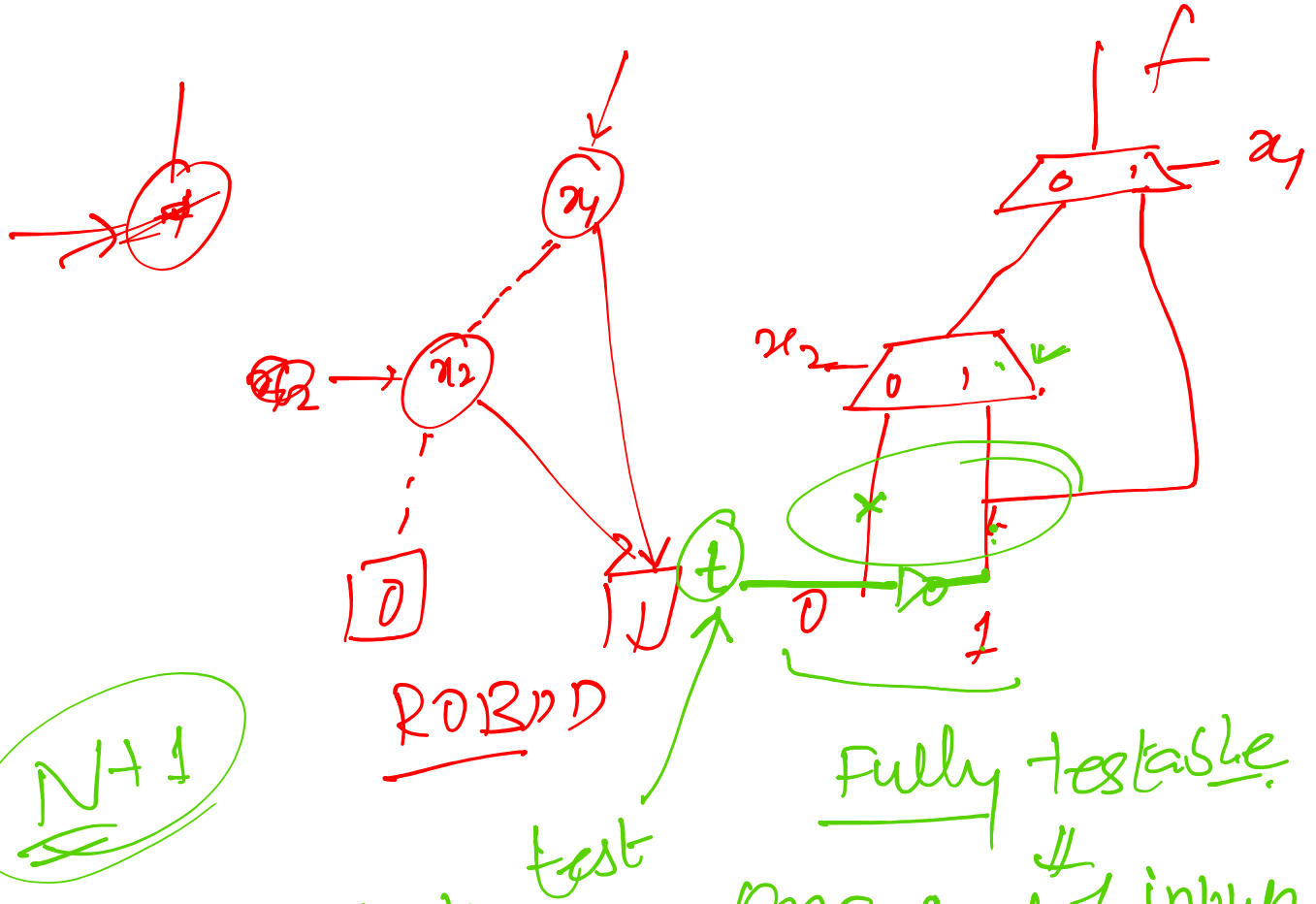
EXPRESSO

Expand - Irredundant - Reduce.

ROBDD based → 2×1 MUX

1 — N

Expand
Reduce
Reshape
Irredundant

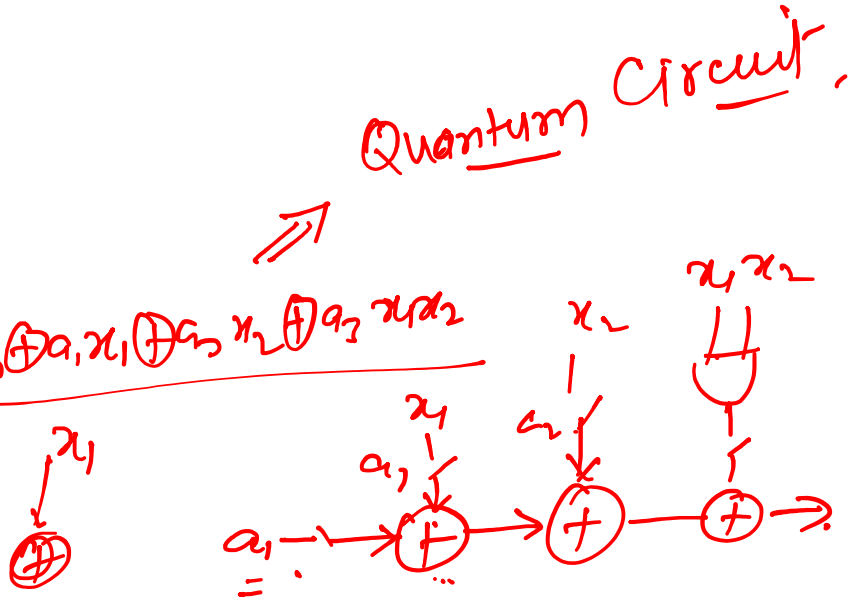

f
x_1
x_2
0
1
t
0
1
ROBDD
N+1
test
Fully testable
⇓
One control input
During test
t → PI
During functional mode t=0.

Circuit Representation

Combinational Circuit Representation

- Sum of Product form (SOP, POS)
- Binary Decision Diagram

– Truth Table
– SOM
– POM
– Reed-Muller ⇒ $a_0 \oplus a_1 x_1 \oplus a_2 x_2 \oplus a_3 x_1 x_2$



XOR, AND

TESTABILITY ⇒ N+4 universal test set

→ # PIs.

Multilevel Logic vs. Two-Level Logic

Example: Let F=a'b+ab', define G and H as follows:

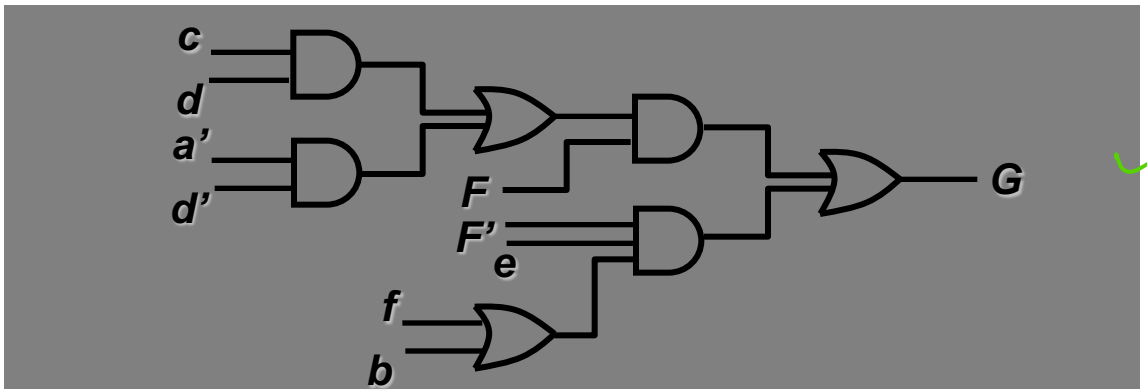
if F is true, then G=cd+a'd', H=cd+a'd'+e(f+b),

else G=e(f+b), H=(cd+a'd')e(f+b).

Multi-Level Implementation:

G=F(cd+a'd')+F'e(f+b) 4-level

H=F(cd+a'd'+e(f+b))+F'(cd+a'd')e(f+b) 5-level



Multilevel Logic vs. Two-Level Logic

Let F=a'b+ab', define G and H as follows:

if F is true, then G=cd+a'd', H=cd+a'd'+e(f+b),

else G=e(f+b), H=(cd+a'd')e(f+b).

G = F(cd+a'd')+F'e(f+b)

= (a'b + ab')(cd + a'd') + (a'b + ab')'(e)(f + b)

= a'bcd + a'bd' + acdb' + e(f+b)(a+b')(a' +b)

= a'bcd + a'bd' + acdb' + (efa + efb' + eba)(a'+b)

= a'bcd + a'bd' + acdb' + efab + efb'a' + eba

THIS IS MORE COMPLICATED TO IMPLEMENT

Multilevel Logic vs. Two-Level Logic

Two-level:

- At most two gates between a primary input and a primary output.
- Real life circuits: PLA.
- Exact optimization methods: well-developed, feasible.
- Heuristics.

Multi-level:

- Any number of gates between a primary input and a primary output.
- Most circuits in real life are multilevel (e.g. standard cells, FPGA).
- Smaller, less power, and (in many cases) faster.
- Exact optimization methods: few, high complexity, impractical.
- Heuristics.

Combinational Circuit

Consider a logic network, with primary input variable {a,b,c,d,e}

And primary output variable {w,x,y,z}

$p = ce + de$

$q = a + b$

$r = p + a'$

$s = r + b'$

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

$u = q'c + qc' + qc$

$v = a'd + bd + c'd + ae'$

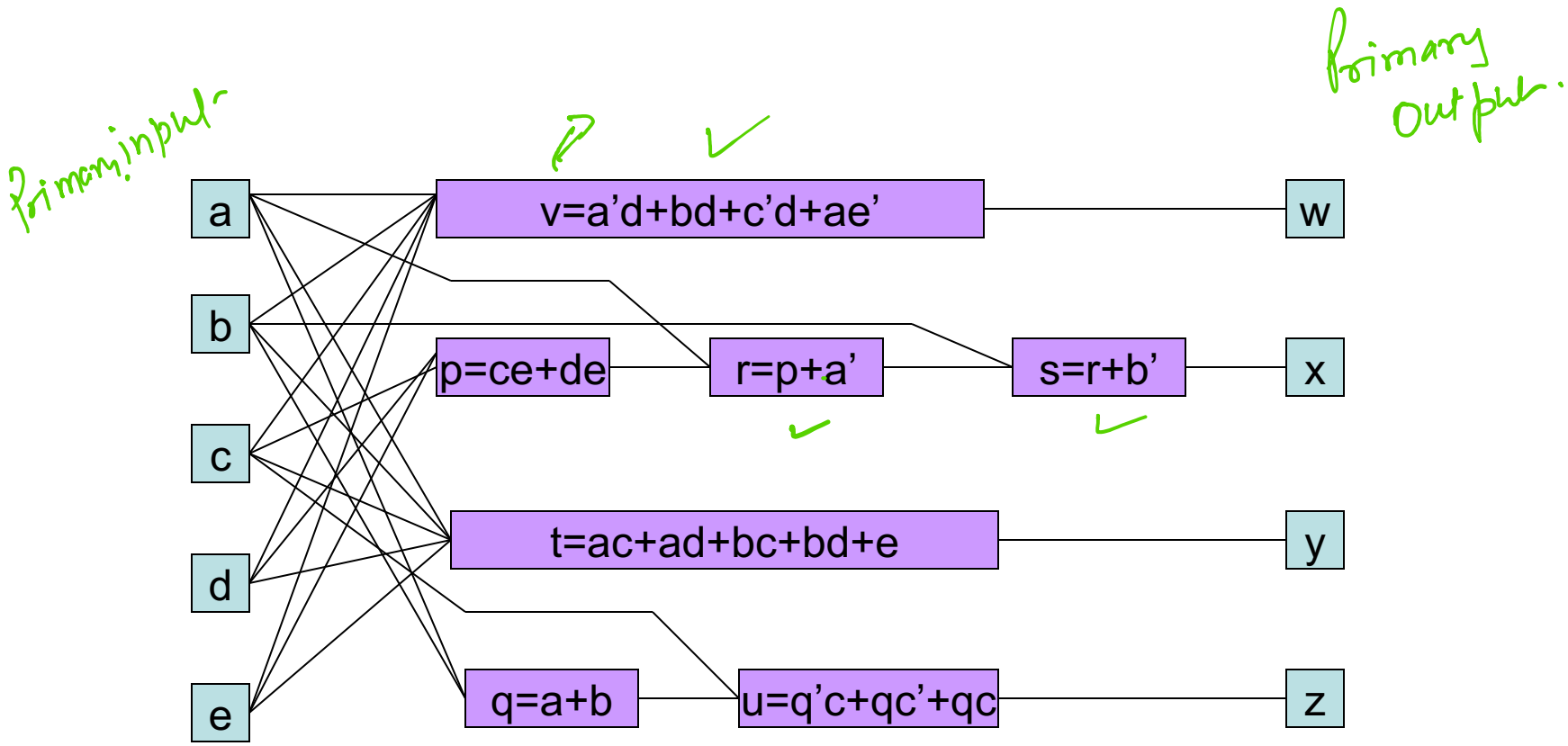
$w = v$

$x = s$

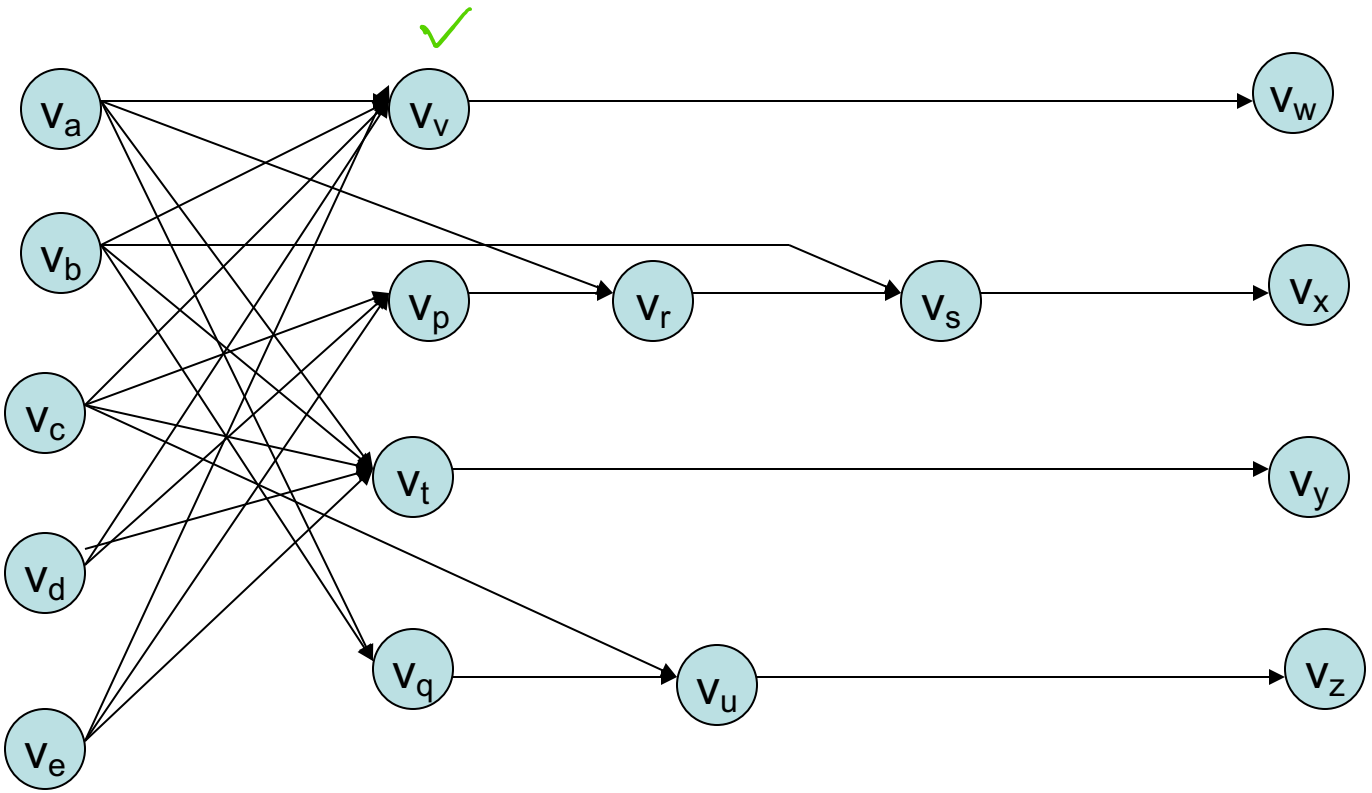
$y = t$

$z = u$

Combinational Circuit



Combinational Circuit



Logic network Graph

Transformation

Heuristic Methods

- Step-wise improvement in the network by means of transformation
- Most of transforms are defined
  - Network equivalence is guaranteed
- Transformation
  - Local
  - Global

Network Optimization

- Minimize maximum delay
  - (Subject to area, power, or testability constraints)
- Minimize area
  - Subject to delay constraints
- Minimize power consumption
  - Subject to timing constraints

Estimation

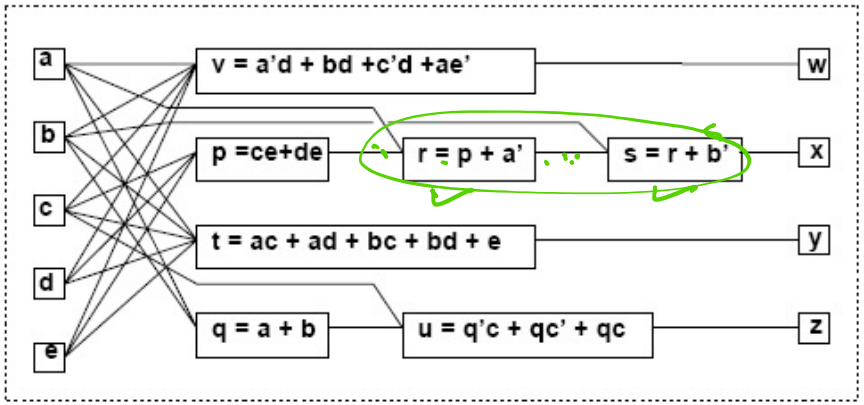
- Area:
  - Number of literals
    - Easy, widely accepted, good estimator
- Delay:
  - Number of stages
  - Gate delay models with wireloads
  - Sensitizable paths
- Power
  - Switching activity at each node
  - Capacitive loads

Strategies for optimization

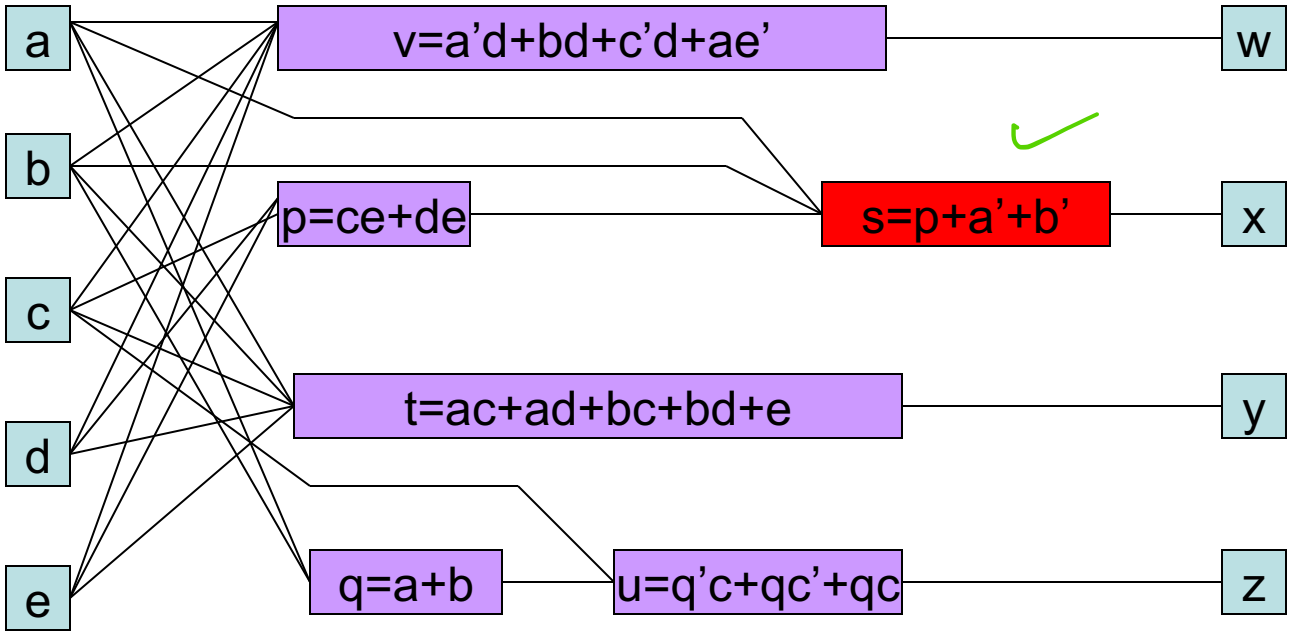
- Improve network step by step
  - Circuit transformations
- Preserve network I/O behavior
  - Exploit environment don't cares if desired
- Methods differ in:
  - Types of transformations applied
  - Selection and order of the transformations

Elimination

- Eliminate one function from the network
  - Similar to Gaussian elimination
- Perform variable substitution
- Example:
  - s = r + b’; r = p + a’;
  - s = p + a’ + b’;



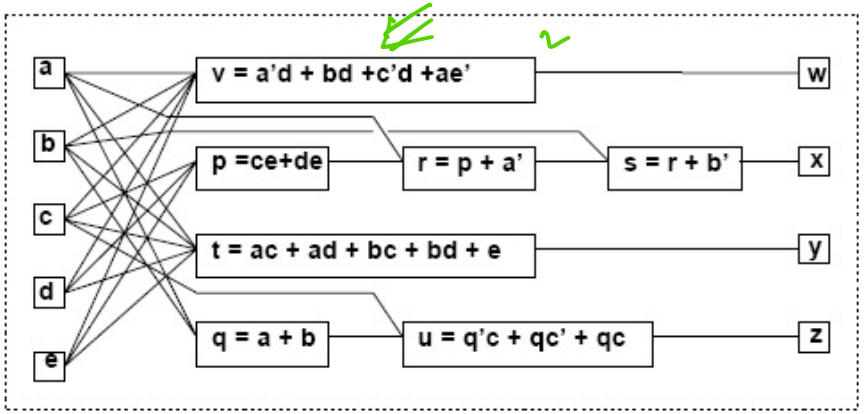
Elimination



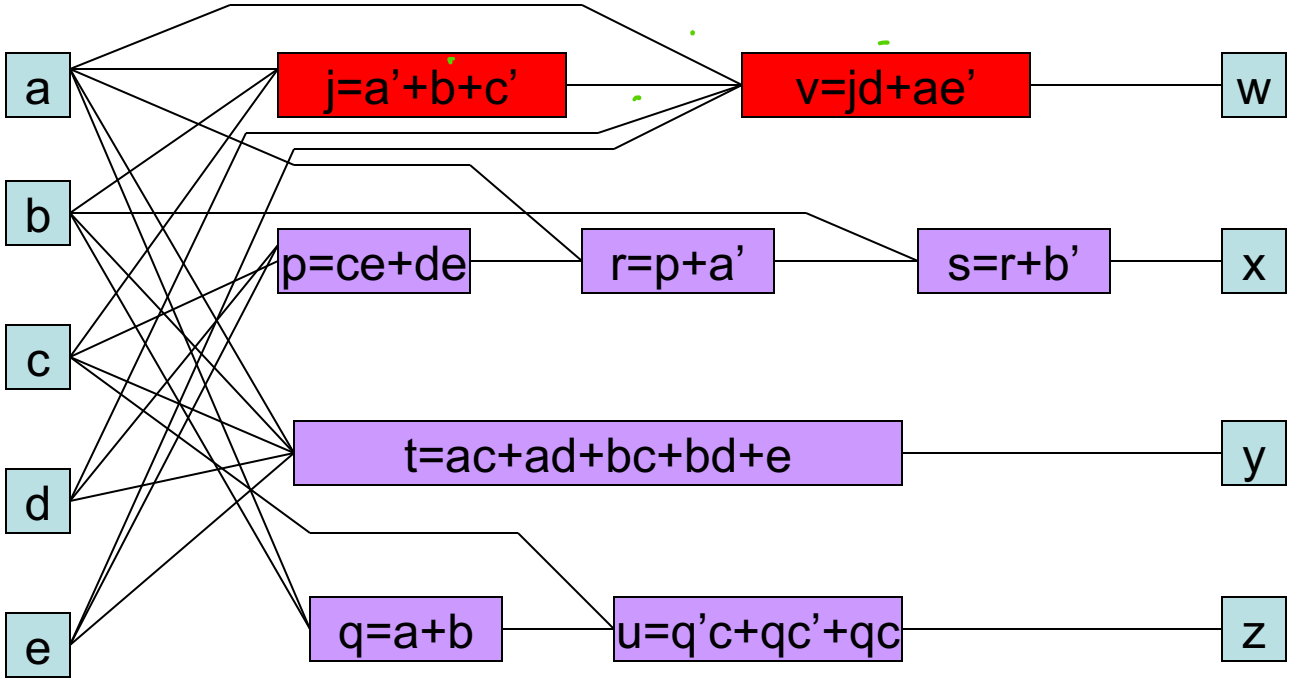
Removal of internal node from the network

Decomposition

- Break a function into smaller ones
  - Opposite to elimination
- Introduce new variables/blocks into the network
- Example:
  - v = a'd + bd +c'd +ae'
  - j = a' + b + c'; v = jd + ae';



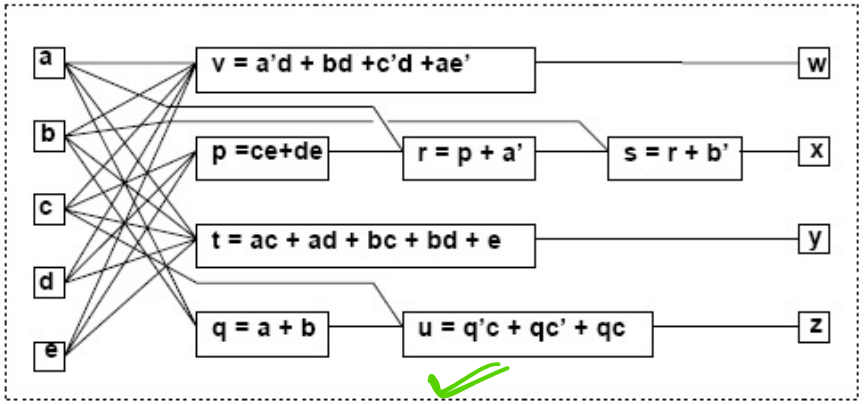
Decomposition



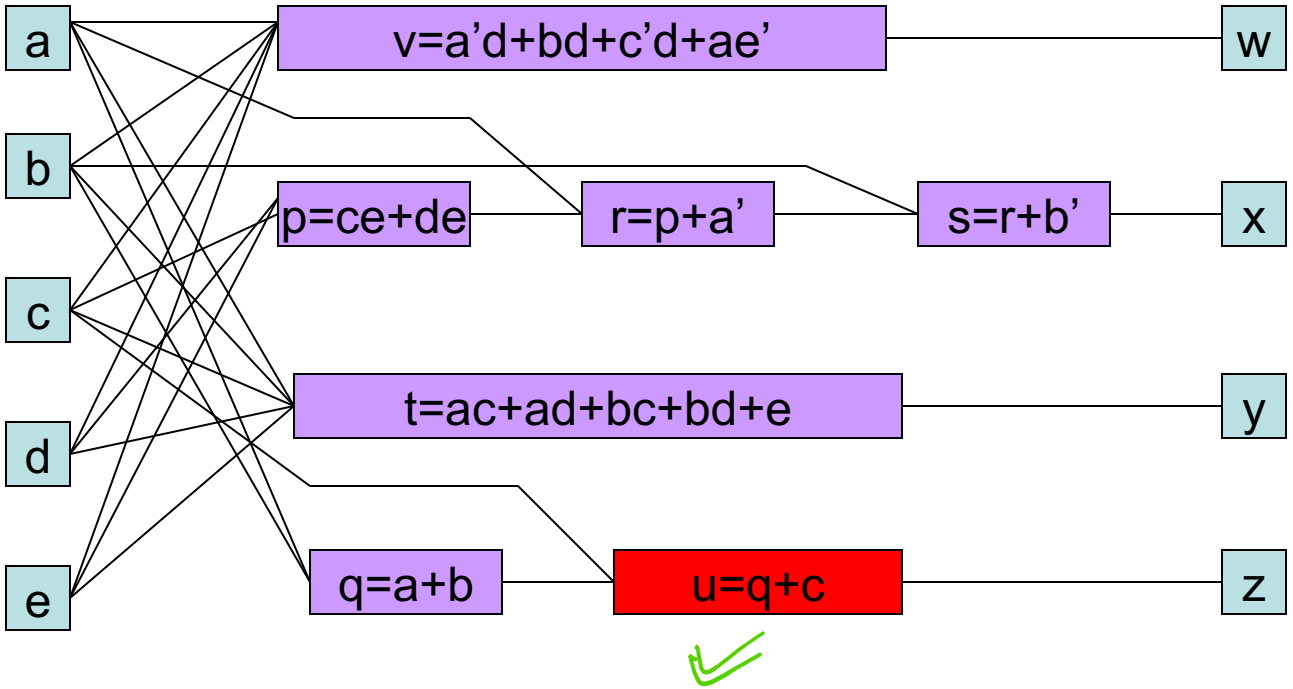
Decomposition of an internal vertex is its replacement by two or more vertices

Simplification

- Simplify local function
  - Use heuristic minimizer like Espresso
  - Modify fanin of target node
- Example:
  - u = q'c + qc' + qc;
  - u = q + c;

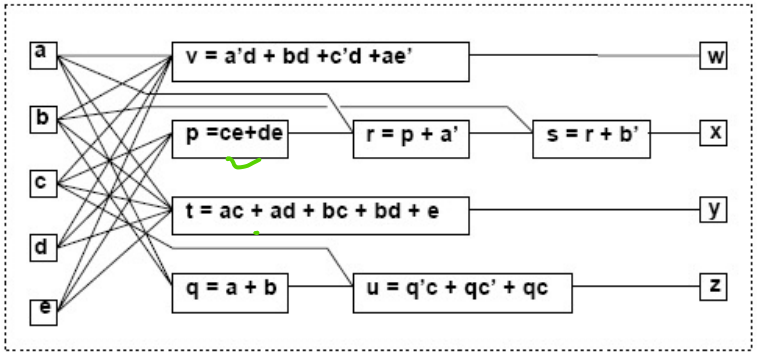


Simplification

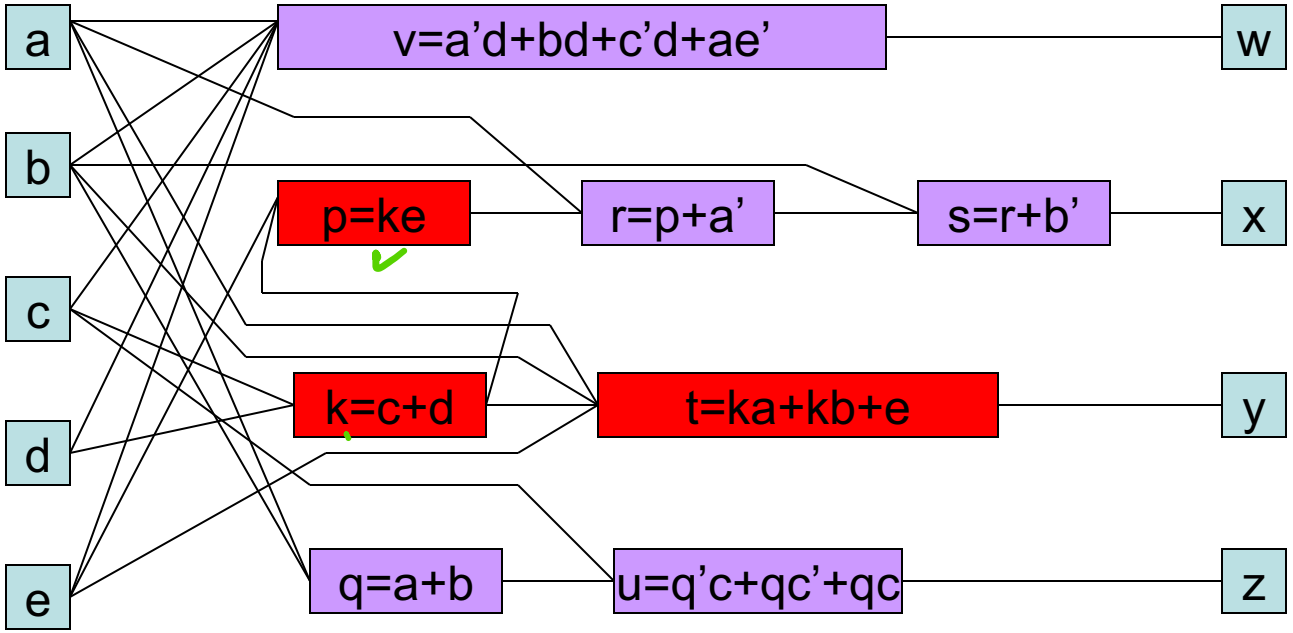


Extraction

- Find a common sub-expression of two (or more) expressions
  - Extract new sub-expression as new function
  - Introduce new block into the circuit
- Example
  - 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;



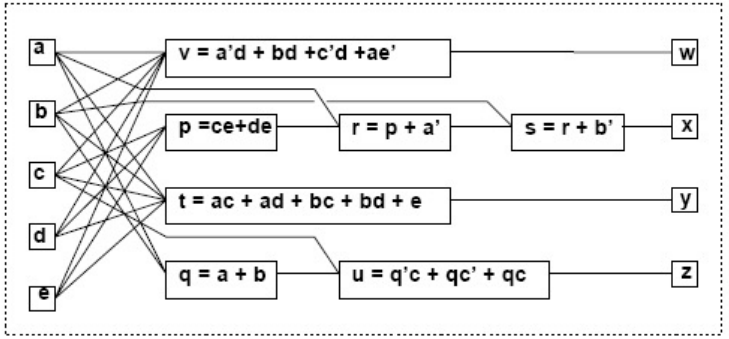
Extraction



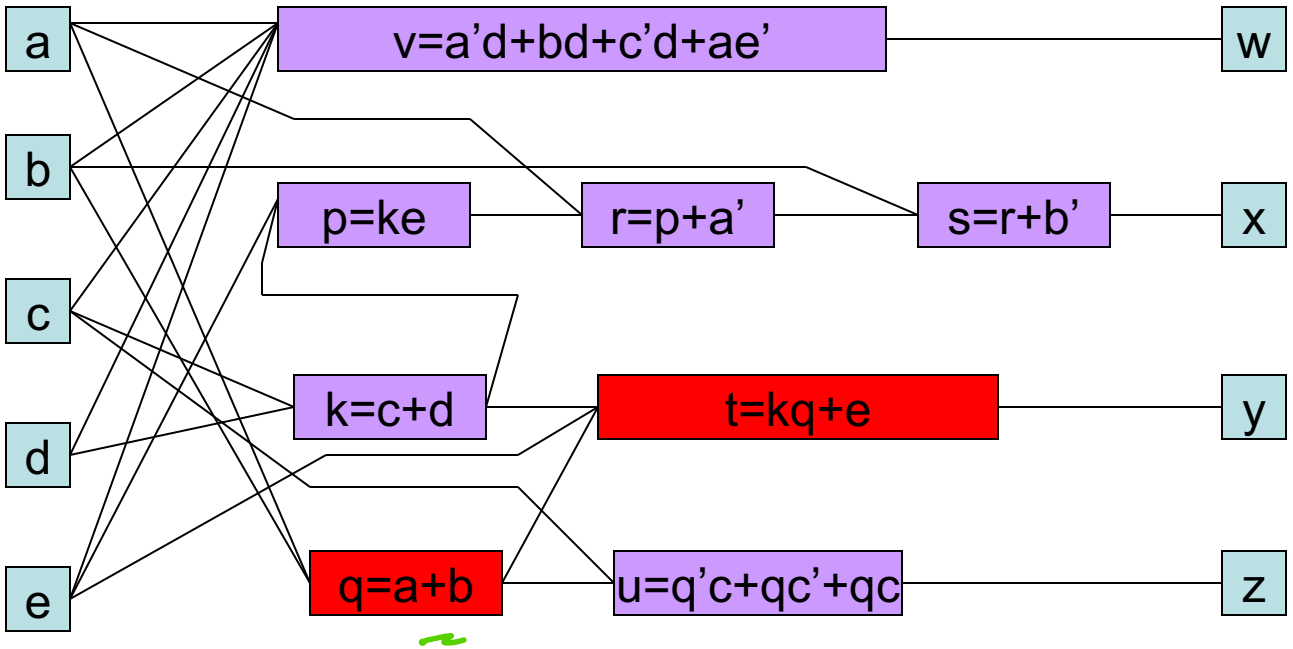
A common sub-expression of two functions associated with two vertices can be extracted by creating a new vertex associated with the sub-expression

Substitution

- Simplify a local function by using and additional input
  that was not previously in its support set
- Example:
  - t = ka + kb + e;
  - t = kq + e;
  - Because q = a + b is already part of the network

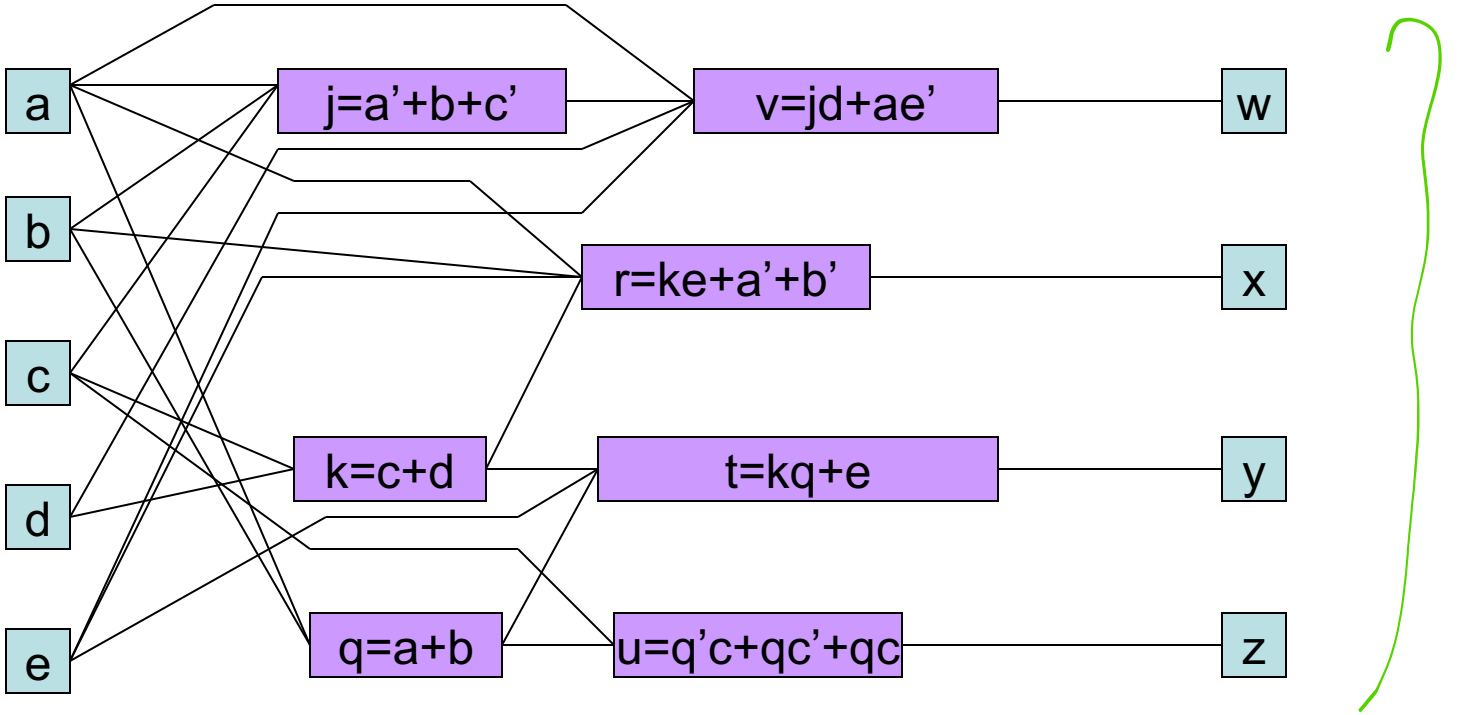


Substitution



A function is reduced in complexity by using an additional input that was not previously in its support set.

Minimized Logic



Transformation - order of transformations.

Optimization Approaches

- Algorithmic approach
  - Define an algorithm for each transformation type
  - Algorithm is an *operator* on the network
  - Algorithms are sequenced by *scripts*
- Rule-based approach
  - Rule data base
    - Set of pattern pairs
  - Pattern replacement is driven by rules
- Most modern tools use the algorithmic approach to synthesis, even though rules are used to address specific issues

Boolean and Algebraic Methods

- Boolean methods for multilevel synthesis
  - Exploit properties of Boolean functions
  - Use *don't care* conditions
  - Computationally intensive
- Algebraic methods
  - Use polynomial abstraction of logic function
  - Simpler, faster, weaker
  - Widely used

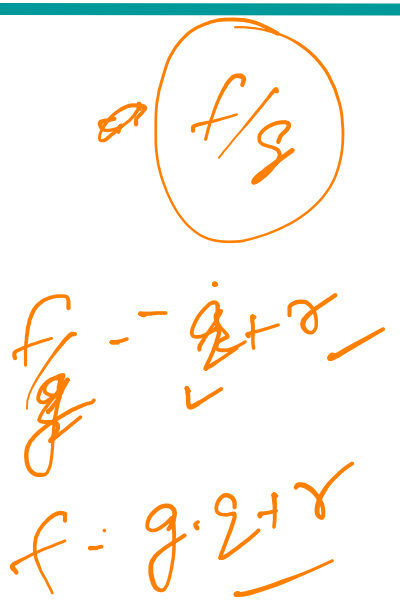
Example

- Boolean substitution:
  - h = a + bcd + c; q = a + cd;
  - h = a + bq + e;
  - Because a + bq +e = a + b(a+cd) + e = a + bcd + e;
- Algebraic substitution:
  - t = ka + kb + e;
  - t = kq + e;
  - Because q = a + b;

easy

Algebraic Method

- Objective
  - Algebraic model
  - Algebraic division
  - Kernel theory and applications



Algebraic Model

- Boolean algebra
  - Complement
  - Symmetric distribution laws
  - *Don't care* sets
- Algebraic models
  - Look at Boolean expressions as polynomials
  - Use sum of product forms
    - Minimal w.r.t. 1-cube containment
  - Use polynomial algebra

Algebraic Division

- Given two algebraic expressions
  - An expression divides algebraically the other
  - $f_{quotient} = f_{dividend} / f_{divisor}$ when:
  - $f_{dividend} = f_{divisor}\ f_{quotient} + f_{remainder}$
  - $f_{divisor}\ f_{quotient} \neq 0$
  - The support of $f_{divisor}$ and $f_{quotient}$ is disjoint
- Note that the $f_{quotient}$ and $f_{divisor}$ are interchangeable

Example

- Algebraic division
  - $f_{dividend} = ac + ad + bc + bd + e$
  - $f_{divisor} = a + b$
  - Then $f_{quotient} = c + d$ and $f_{remainder} = e$
    because $(a+b)(c+d) + e = f_{dividend}$
    and $\{a,b\} \cap \{c,d\} = \emptyset$
- Non-algebraic division:
  - ✓ $f_i = a + bc$ and $f_j = a+b$
  - ✓ Then $(a+b)(a+c) = f_i$
    but $\{a,b\} \cap \{a,c\} \neq \emptyset$

An Algorithm for Division

- Division can be performed in different way
  - Straightforward algorithm by literal sorting
    - Simple, quadratic complexity
  - Advanced algorithm using sorting
    - N-logN complexity
  - Typically algebraic division runs fast – small-sized problems
- Definitions
  - A = set of cubes C^A_j of the dividend. There are l
  - B = set of cubes C^B_i of the divisor. There are n
  - Q = quotient; R = remainder

An Algorithm for Division

ALGEBRAIC_DIVISION(A,B)

```
{
    for (i = 1 to n)
    {
        D = {C^A_j such that C^A_j ⊇ C^B_i };
        if (D == Ø) return(Ø,A);
        D_i = D with variables in sup(C^B_i) dropped;
        if i = 1
                Q = D_i ;
        else
                Q = Q ∩ D_i ;
    }
    R = A – Q x B;
    return(Q,R);
}
```

Example

$f_{dividend} = ac+ad+bc+bd+e;\quad f_{divisor} = a+b$

- A = {ac,ad,bc,bd,e} and B = {a,b}
- i = 1:
  - C^B_1 = a, D = {ac,ad} and D_1 = {c,d}
  - Then Q = {c,d}
- i = 2 = n:
  - C^B_2 = b, D = {bc,bd} and D_2 = {c,d}
  - Then Q = {c,d} ∩ {c,d} = {c,d}
- Result:
  - Q = {c,d} and R = {e}
  - $f_{quotient}$ = c + d and $f_{remainder}$ = e

Theorem

- Given algebraic expression f_i and f_j then f_i / f_j is empty when either:
  - f_j contains a variable not in f_i
  - f_j contains a cube whose support is not contained in that of any cube of f_i
  - f_j contains more terms than f_i
  - The count of any variable in f_j is higher than in f_i

Algebraic Substitution

- Consider expression pairs
- Apply division (in any order)s
- If quotient is not void:
  - Evaluate area and delay gain
  - Substitute $f_{dividend}$ by $j\ f_{quotient} + f_{remainder}$ where j is the variable corresponding to $f_{divisor}$
- Use filters based on previous theorem to reduce computation

Substitution Algorithm

```
SUBSTITUTE(Gn(V,E)){
    for (i = 1,2,…,|V|){
        for (j = 1,2,…,|V|;j ≠ i){
            A = set of cubes of f_i;
            B = set of cubes of f_j;
            if (A,B pass the filter test){
                (Q,R) = ALGEBRAIC_DIVISION(A,B);
                if (Q ≠ Ø){
                    f_quotient = sum of cubes of Q;
                    f_remainder = sum of cubes of R;
                    if (substitution is favorable)
                        f_i = j  f_quotient + f_remainder;
                }
            }
        }
    }
}
```

Extraction

- Search for common sub-expressions
  - Single-cube extraction
  - Multiple-cube extraction (kernel extraction)
- Search for appropriate divisors
- Extraction is still done using the original kernel theory of Brayton and others [IBM]

Definitions

- Cube-free expression
  - Expression that cannot be factored by a cube
  - Example:
    - a + bc is cube free
    - abc and ab + ac are not
- Kernel of an expression
  - Cube-free quotient of the expression divided by a cube, called co-kernel
  - Note that since divisors and quotients are interchangeable, kernels are just a subset of divisors
- Kernel set of an expression f is denoted by K(f)

Example

- f = ace + bce + de + g
- Trivial kernel search:
  - Divide f by a. Get ce. Not cube free
  - Divide f by b. Get ce. Not cube free
  - Divide f by c. Get ae + be. Not cube free
  - Divide f by ce. Get a + b. Cube free. KERNEL!
  - Divide f by d. Get e. Not cube free
  - Divide f by e. Get ac + bc + d. Cube free. KERNEL!
  - Divide f by g. Get 1. Not cube free
  - Divide f by 1. Get f. Cube free. KERNEL!
- K(f) ={ (a+b); (ac+bc+d); (ace+bce+de+g) }
- CoK(f) = { ce, e, 1}

Theorem: Brayton and McMullen

- Two expressions f_a and f_b have a common multiple-cube divisor f_d if and only if
  - There exist kernels k_a in $K(f_a)$ and k_b in $K(f_b)$ such that f_d is the sum of two (or more) cubes in $k_a \cap k_b$

- Consequences
  - ✓ If kernel intersection is void, then the search for common sub-expression can be dropped
  - ✓ If an expression has no kernels, it can be dropped from consideration
  - ✓ The kernel intersection is the basis for constructing the expression to extract

Example

- f_x = ace + bce + de + g
- f_y = ad + bd + cde + ge
- f_z = abc
- K(f_x) = { (a+b); (ac+bc+d); (ace+bce+de+g) }
- K(f_y) = { (a+b+ce); (cd+g); (ad+bd+cde+ge) }
- The kernel set of f_z is empty
- Select intersection (a+b)
  - f_w = a + b
  - f_x= wce + de + g
  - f_y = wd + cde + ge
  - f_z = abc

Kernel set computation

- Naïve method
  - Divide function by the elements of the power set of its support set
  - Weed out non cube-free quotients
- Smart way
  - Use recursion
    - Kernels of kernels are kernels
  - Exploit commutativity of multiplication

Recursive Algorithm

- The recursive algorithm is the first one proposed for kernel computation and still outperforms others
- It will be explained in two steps
  - R_KERNELS (with no pointer) to understand the concept
  - KERNELS (Complete algorithm)
- The algorithms use a subroutine
  - CUBES(f,C) which returns the cubes of f whose literals include those of cube C
  - Example: f = ace +bce + de + g --

    CUBES(f, ce) = ace + bce

Simple Recursive Algorithm

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

Analysis

- The recursive algorithm does some redundant computation in the recursion
  - Example
    - Divide by a and then by b
    - Divide by b and then by a
  - Obtain duplicate kernels
- Improvement
  - Exploit commutativity of multiplication
  - Keep a pointer to the literals used so far

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 ∪ KERNELS( f / C ,i+1);
                }
        }
        K = K ∪ f;
        return(K);
}
```

Example

- f = ace + bce+ de + g
- Literals a and b. No action required
- Literal c. Select cube ce
  - Recursive call with argument f/ce= a+b. Pointer j = 3+1
  - Call considers variables {d,e,g}. No kernel.
  - Adds a + b to the kernel set at the last step.
- Literal d. No action required.
- Literal e. Select cube e
  - Recursive call with argument f/e = ac + bc + d. Pointer j = 5+1
  - Call considers variables {g}. No Kernel
  - Adds ac+bc+d to the kernel set at the last step of recursion
- Literal g. No action required
- Add f = ace + bce + de + g to kernel set
- K(f) = { (ace+bce+de+g),(ac+bc+d),(a+b) }

Thank You