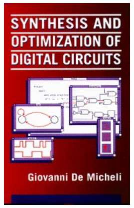
Multi-level Logic Synthesis

Giovanni De Micheli

Integrated Systems Centre

EPF Lausanne



Module 1

- **Objectives**
  - **What is multi-level logic synthesis**
  - **What are the specific goals**
  - **Stepwise transformations**

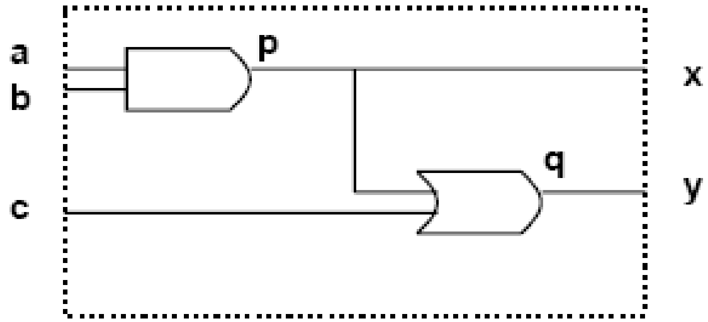
Motivation

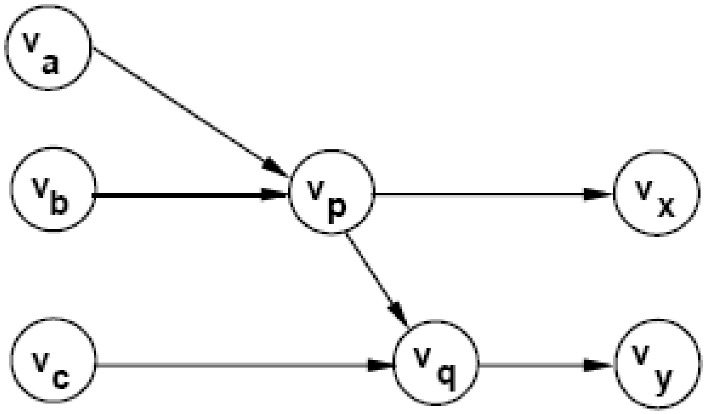
- Multiple-level logic networks
  - Semi-custom libraries
  - Logic gates versus macro-cells
    - More flexibility
    - Privilege specific paths on others
    - Better performance
- Applicable to a large variety of designs
- The importance of logic synthesis grew in parallel with the growth of foundries for the semi custom market

Circuit model

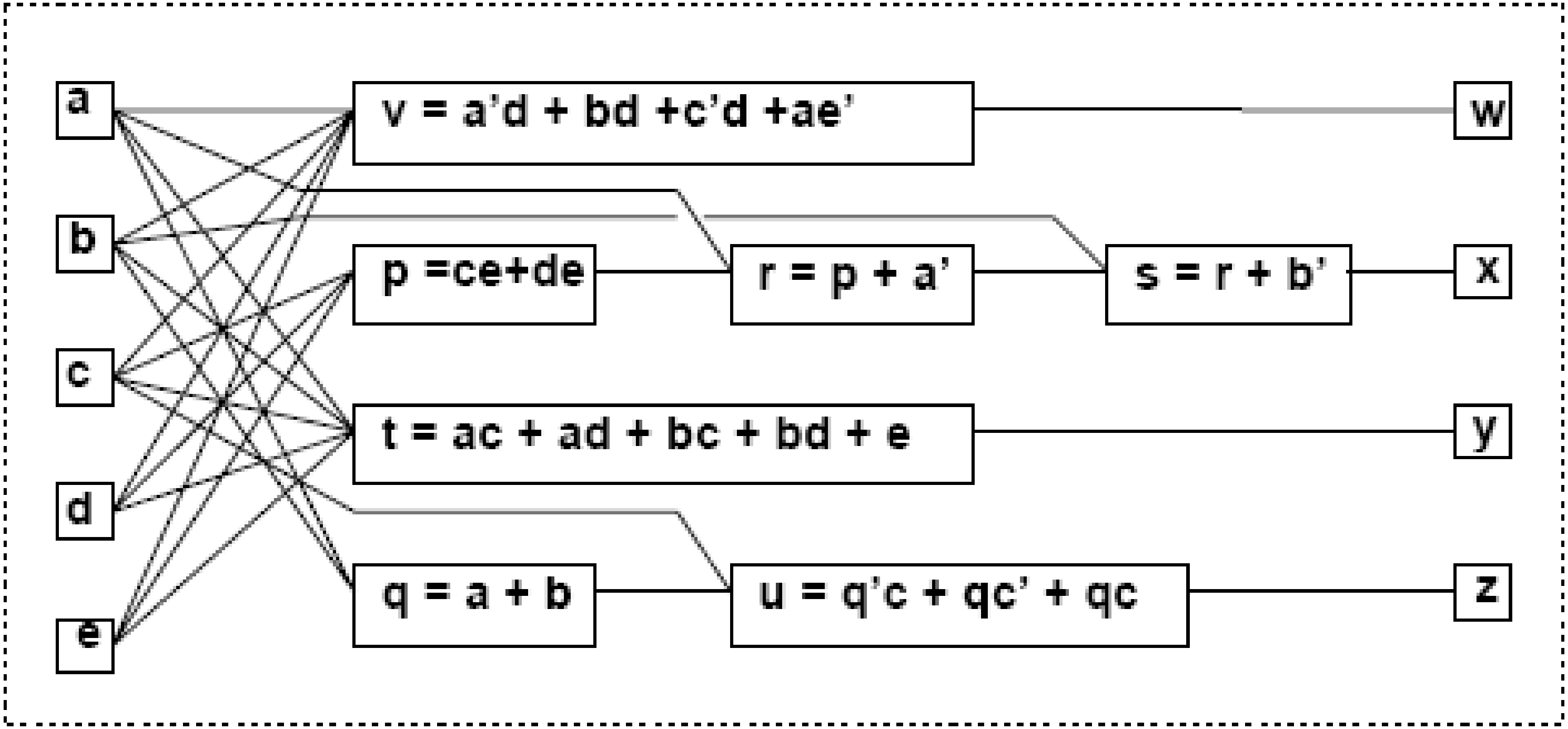
- Logic network
  - An interconnection of blocks
    - Each block modeled by a Boolean function
  - Usual restrictions:
    - Acyclic and memoryless
    - Single-output functions
- The model has a structural/behavioral semantics
  - The structure is induced by the interconnection
- Mapped network
  - Special case when the blocks correspond to library elements

Example of mapped network

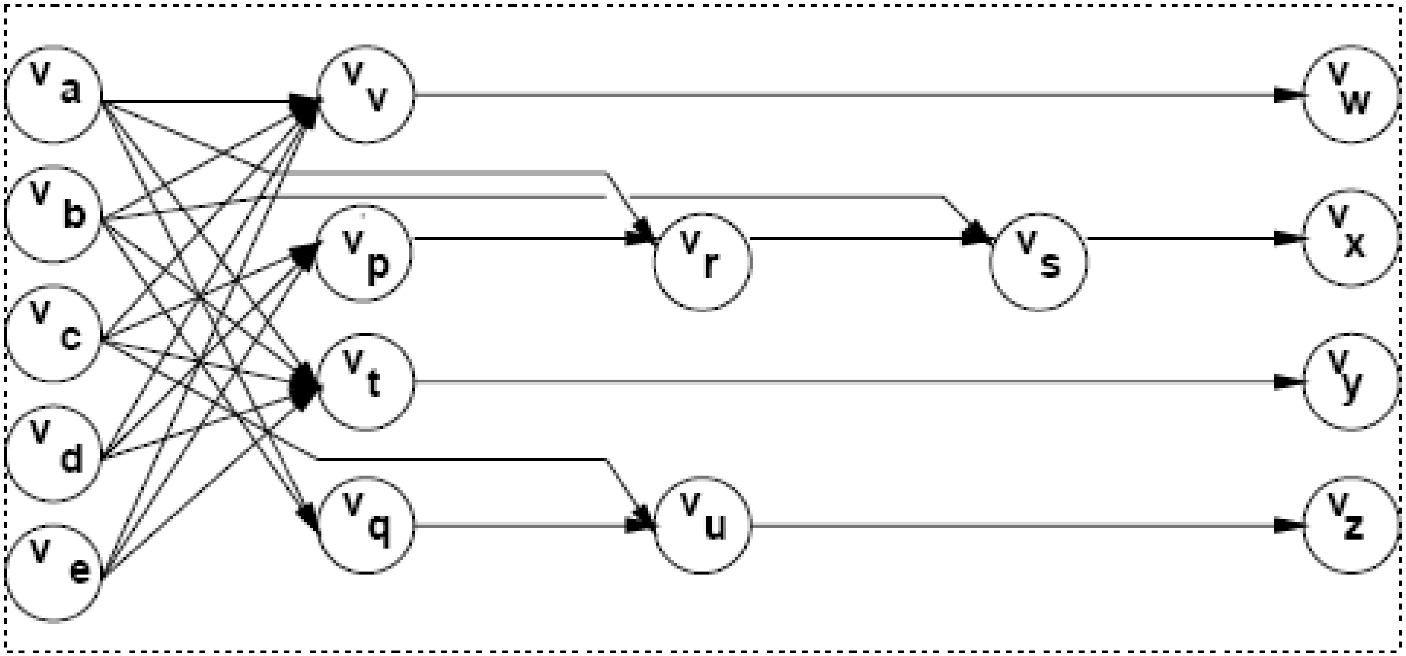




Example of general network



Example of general network graph



Network represented by assignments

$$
\begin{aligned}
p &= ce + de \\
q &= a + b \\
r &= p + a' \\
s &= r + b' \\
t &= ac + ad + bc + bd + e \\
u &= q'c + qc' + qc \\
v &= a'd + bd + c'd + ae' \\
w &= v \\
x &= s \\
y &= t \\
z &= u
\end{aligned}
$$

Example of terminal behavior

- **I/O functional behavior**
  - **Vector with as many entries as primary outputs**
  - **Each entry is a logic function**

$$
f = \begin{bmatrix} a'd + bd + c'd + ae' \\ a' + b' + c + d \\ ac + ad + bc + bd + e \\ a + b + c \end{bmatrix}
$$

Network optimization

- Minimize maximum delay
  - (Subject to area or power 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

Problem analysis

- Even the simplest problems are computationally hard
  - E.g., multi-input single-output network
- Few exact methods proposed
  - High complexity
  - Impractical
- Approximate optimization methods
  - Heuristic algorithms
  - Rule-based methods

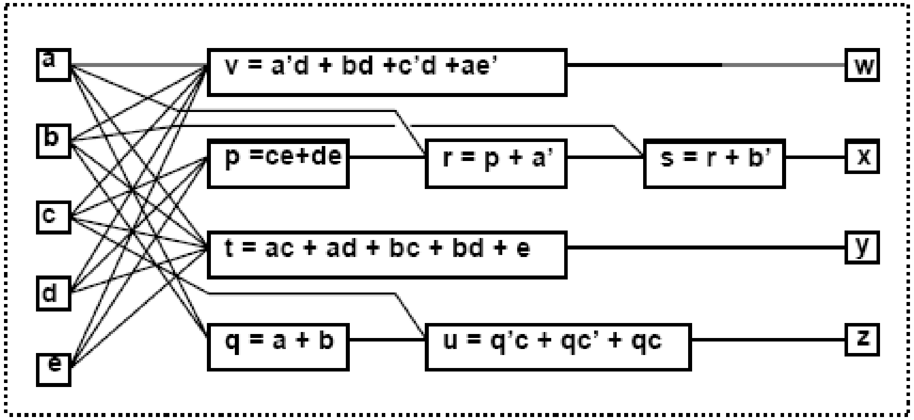
Strategies for optimization

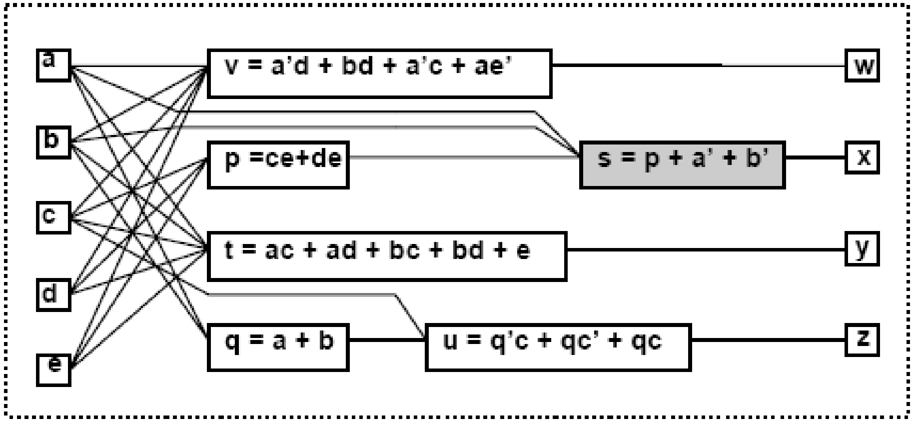
- Improve network step by step
  - Circuit transformations
- Preserve network I/O behavior
  - Exploit environment don’t cares if desired
- Methods differ in:
  - Types of transformations applied
  - Selection and order of the transformations

Elimination

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

Example

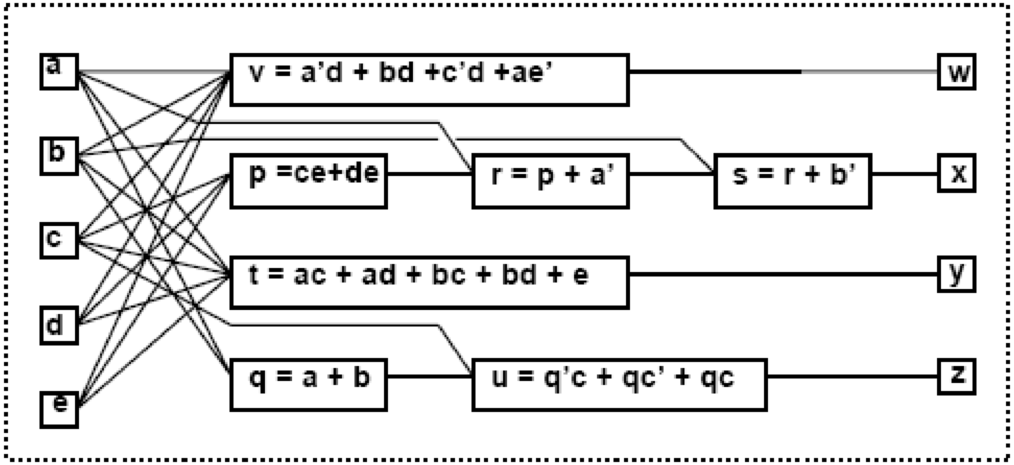

a
b
c
d
e
v = a'd + bd +c'd +ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q'c + qc' + qc
w
x
y
z


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

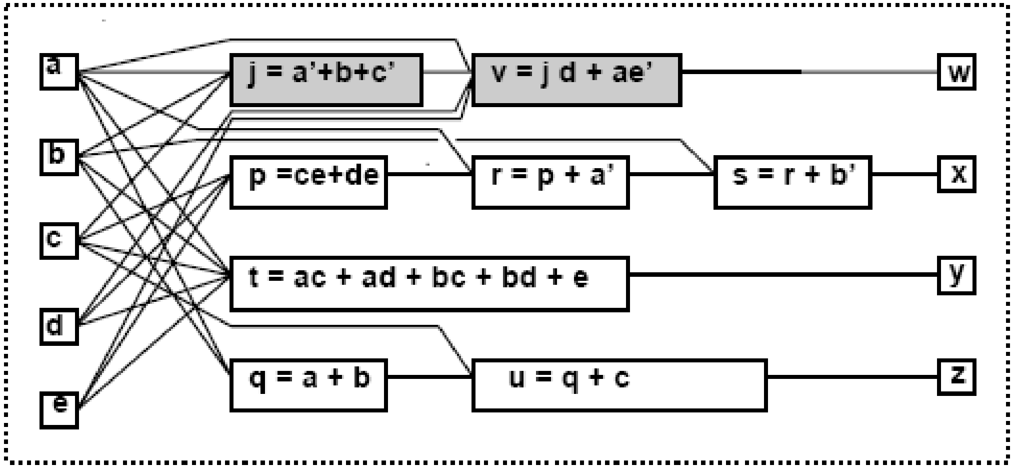
Decomposition

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

Example



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

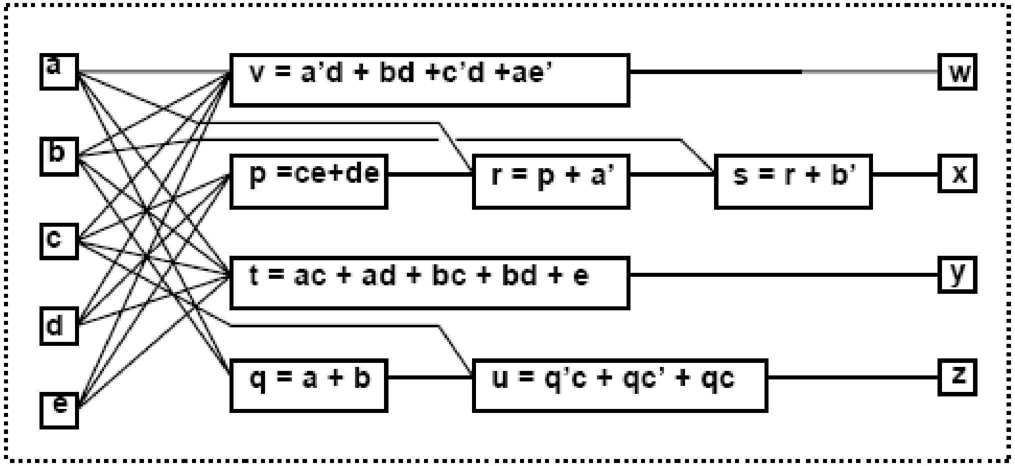


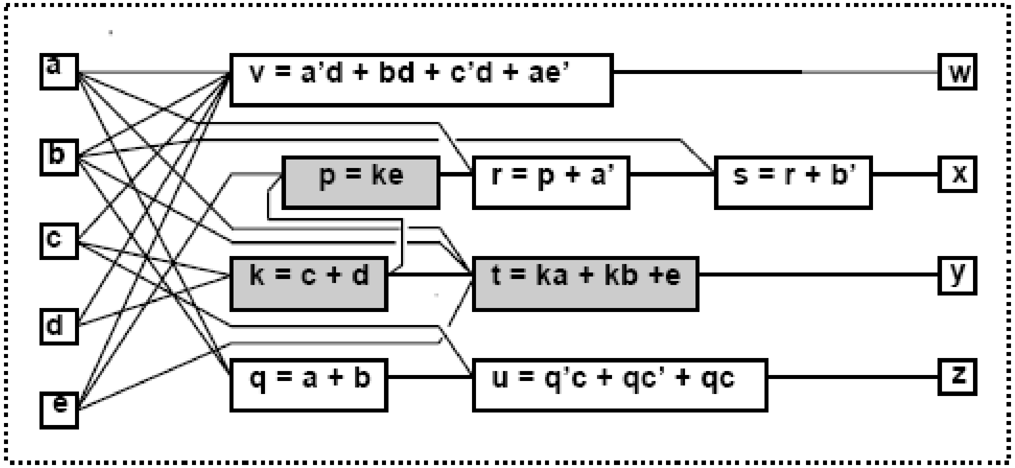
a
b
c
d
e
j = a'+b+c'
v = j d + ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q + c
w
x
y
z

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

Example

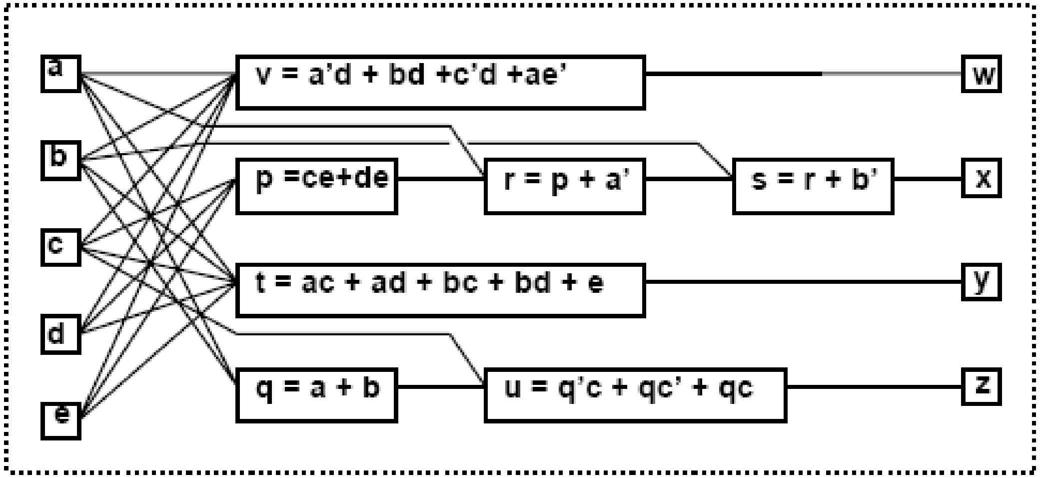

a
b
c
d
e
v = a'd + bd +c'd +ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q'c + qc' + qc
w
x
y
z

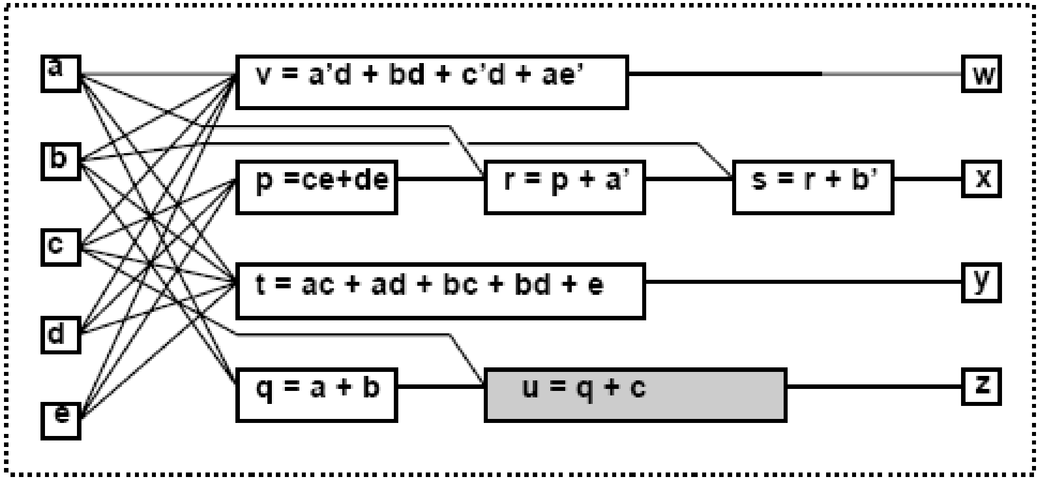

a
b
c
d
e
v = a'd + bd + c'd + ae'
p = ke
r = p + a'
s = r + b'
k = c + d
t = ka + kb +e
q = a + b
u = q'c + qc' + qc
w
x
y
z

Simplification

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

Example

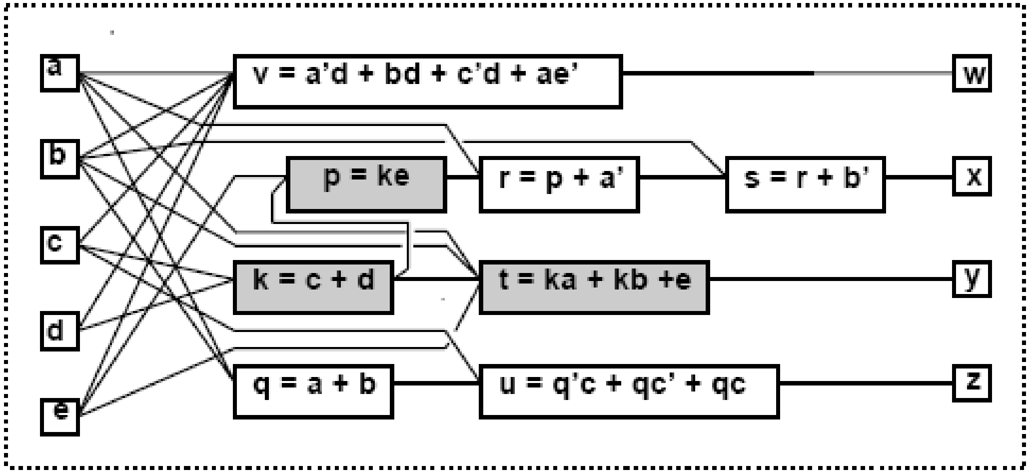

a
b
c
d
e
v = a'd + bd +c'd +ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q'c + qc' + qc
w
x
y
z

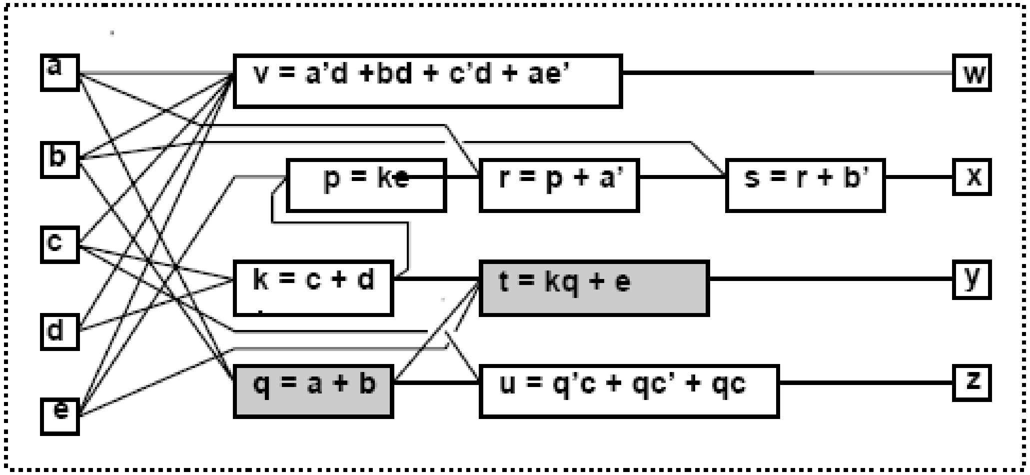

a
b
c
d
e
v = a'd + bd + c'd + ae'
p =ce+de
r = p + a'
s = r + b'
t = ac + ad + bc + bd + e
q = a + b
u = q + c
w
x
y
z

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

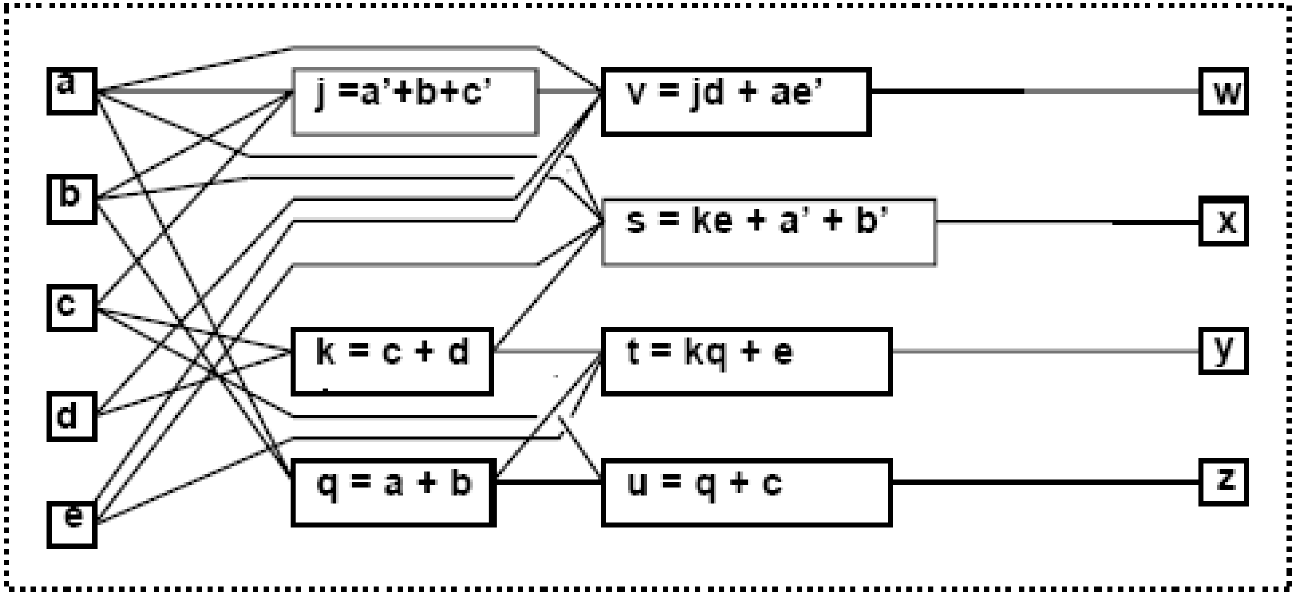
Example


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

Example – Sequence of transformations

- $j = a' + b + c$
- $k = c + d$
- $q = a + b$
- $s = ke + a' + b'$
- $t = kq + e$
- $u = q + c$
- $v = jd + ae'$



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

Module 2

- **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.to 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

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 $Q = \{c,d\} \cap \{c,d\} = \{c,d\}$
- 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)
- 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
- $K(f) = \{ (ace+bce+de+g), (ac+bc+d), (a+b) \}$

Matrix representation of kernels

- $f = ace + bce + de + g$
- Incidence matrix
  - Cubes vs. variables
- Rectangle
  - Subset of rows/columns with all entries equal to 1
- Prime rectangle
  - Rectangle not included in another rectangle
- A co-kernel is a prime rectangle with at least two rows
- Example:
  - Prime rectangle ({1,2},{3,5})
  - Co-kernel 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

Application of kernel methods

- Single cube extraction
  - Extract one cube from two (or more) sub-expressions [Brayton]
- Kernel extraction
  - Extract a multiple-cube expression [Brayton]
- Double-cube extraction
  - Newer fast and efficient routine [Rajski]
- Kernel-based decomposition

Single-cube extraction

- Form an auxiliary expression, which is the union (sum) of all local expression
- Find the largest co-kernel
  - Corresponding kernel must belong to two (or more) different expressions
  - Use additional variables to tag the expressions
- Extract chosen co-kernel
- The problem can be well visualized by a matrix representation and the extraction of a prime rectangle

Example

- **Expressions:**
  - $f_x = ace + bce + de + g$
  - $f_s = cde + b$
- **Auxiliary function:**
  - $f_{aux} = ace + bce + de + g + cde + b$
- **Tagging:**
  - $f_{aux} = xace + xbce + xde + xg + scde + sb$
- **Co-kernel:** ce
- **After cube extraction**
  - $f_z = ce$
  - $f_x = z\ (a+b) + de + g$
  - $f_s = ze + b$

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

Multiple-cube extraction

- We need a cube/kernel matrix
  - Relabel cubes by new variables
  - Kernels are now cubes in these new variables
- Find a prime rectangle
- Equivalently, find a co-kernel of the auxiliary expression that is the sum of the relabeled expressions

Example

- $f = ace + bce$
  - $K(f) = \{(a+b)\}$
- $g = ae + be + d$
  - $K(g) = \{(a+b); (ae + be + d)\}$
- Relabeling: $x_a = a$; $x_b = b$; $x_{ae} = ae$; $x_{be} = be$; $x_d = d$
  - Then $K(f) = \{\{x_a, x_b\}\}$ and $K(g) = \{\{x_a, x_b\}, \{x_{ae}, x_{be}, x_d\}\}$
  - $f_{aux} = f\, x_a\, x_b + g\, x_a\, x_b + g\, x_{ae}\, x_{be}\, x_d$
  - $CoK(f_{aux}) = x_a\, x_b$
- Go back to original variables
  - Extract $(a + b)$ from f and g

Kernel-based decomposition

- There are many different ways of performing decomposition
  - Several classic approaches (e.g., Ashenhurst & Curtis)
- Algebraic decomposition
  - Find good algebraic divisors
  - Use kernels and decompose recursively

Example

- Decompose $f = ace + bce + de + g$
- Select kernel $ac + bc + d$
- Decompose as: $f = te + g$; $t = ac + bc + d$
- Recur on quotient t
- Select kernel $a + b$
- Decompose $t = sc + d$; $s = a + b$; $f = te + g$;

Summary
algebraic methods

- Algebraic methods abstract functions as polynomials
  - Polynomial division
- Methods are fast and widely applicable
- Algebraic methods miss opportunities for optimization
  - As compared to Boolean methods
- Algebraic transformations are reversible
  - Ease transformations back and forward to trade off area and speed