NUMERO: 2421A ANNO: 2019

A P P U N T I

STUDENTE: Ricci Francesco

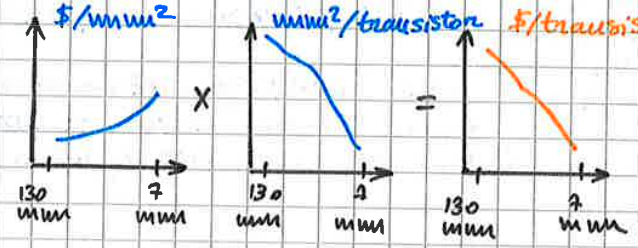
MATERIA: Synthesis and optimization of digital systems - Prof. Calimera Andrea

Introduction

Systems our chips are everywhere, since electronics brings good value to a lot of domains and applications for a small or moderate cost.
Most of consumer electronics like PC and smartphones are not anymore growing in sales ⇨ market saturated... The industry is moved to a **new market driver: IoT**!
The two domains are very different: ICs for desktop are large, power hungry and expensive, while IoT devices must embedd more functions (sensors + RF) in a smaller footprint for a much lower price (and usually must be energy efficient as well: think about wearables or bionic devices).

How has been possible to produce more complex and smaller devices for such a lower price? There are two main **enabling factors**:

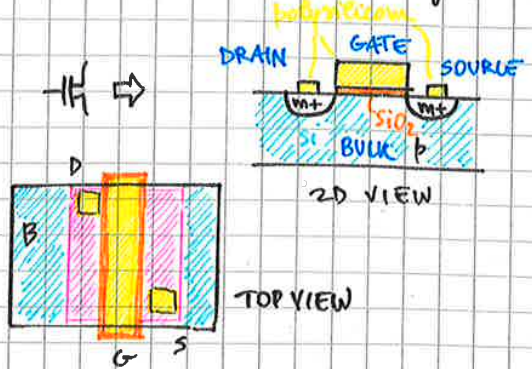
1) **Technology scaling**: silicon vendors are able to provide more complex silicone devices with a larger number of transistors in a smaller area ⇨ Moore's Law: at each new technology node (130 nm, 90 nm, 65 nm, 45 nm, 32 nm, 14 nm, 7 nm) the overall cost/mm^2 is increasing, but since the geometries of transistors are shrinking, you can pack in the same amount of area more transistors (hence mm^2/transistor decreases); as a resultant effect the cost/transistor is reducing esponentially over time

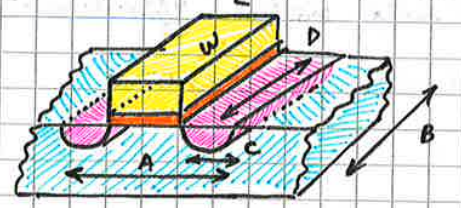


So now more complex circuits can be implemented on a smaller footprint. But more complex means **more design complex**!
Bigger complexity require more time for designing, increasing the time to market. Today, missing a sweet spot in the market due to delays leads to huge economical losses!
How to maintain time to market small, even if complexity grows?

2) **Design automation**: it is not possible anymore to design by hand an electronic device, due to its complexity. But it is not only about design: verification and testing can easily take 60% of design time ⇨ you take more time to design the test-bench wrt time to design the circuit.

A transistor is a 3D object with some parameters characterizing it:





3D VIEW
(missing S and D contacts)

at least 6 parameters x transistor

Total cost = fixed cost (fabrication setup, masks, design cost) + variable costs (testing, raw materials, energy/electricity, maintenance)

Remember that design automation improves QoR (quality of results) and manages complexity:

HIGH VOLUME PRODUCTION : variable cost is the main contribution to final cost : design automation is able to manage complex designs ⇨ they'll be characterized by higher quality; metrics can be optimized to use the least amount of resources x IC ⇨ variable cost decrease

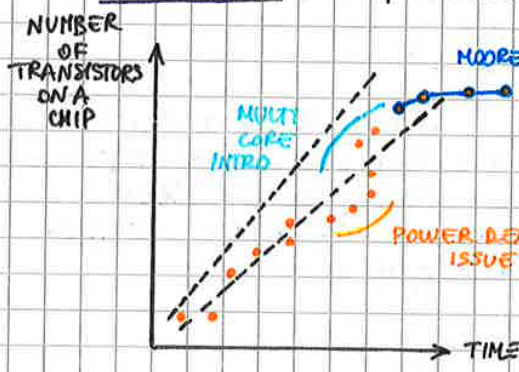
LOW VOLUME PRODUCTION : fixed cost is dominant in final cost : design automation suite reduces the design time, making fixed costs lower

design automation can make the device easier to test, reducing again variable costs

empirical law...

9/03/18

Moore's law : every 18 month, number of transistor doubles



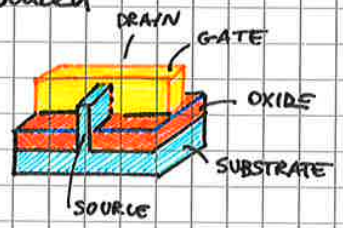
recap: moving from desktop applications to IoT (# decrease, complexity increase)

Enabling factors:
1) Tech scaling ($/tr decrease)
2) Design automation ($/design ↓)
+ manage complex design with less human interaction
+ improve quality of design (QoD)

c) Silicon on insulator (SOI): bulk is substituted by an insulator, making transistors somehow isolated. This approach mitigates the two issues of noise and leakage.

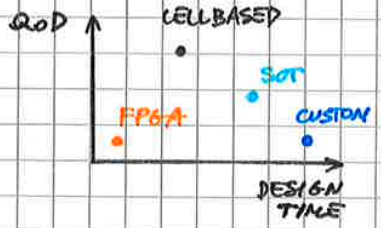
At steady state, a transistor should not have current between source and drain... however a small leakage is present

d) thin FAT: gate is a tri-dimensional object, able to better control the transistor: the switch is better, leakage is reduced

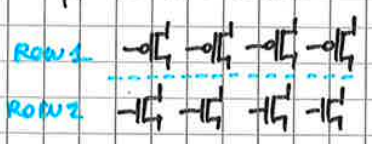


3) DESIGN STYLE:
- CUSTOM
- SEMI CUSTOM
  - CELL BASED
  - ARRAY BASED
    - SEA OF TRANSISTORS (SOT)
    - FPGA

a) custom: define all geometries at layout level: time consuming, complexity, no CAD tools



b) sea of transistors: a silicon vendor provides a chip with transistors (of the same size) already placed: usually organized in rows of pmos and nmos



at design time you must define all the interconnections
design time reduced

c) FPGA: based on a sea of logic blocks approach (CLBs)

CLB array already designed (LUTs) and placed; programmable interconnections between logic blocks

d) cell-based: the designer can define logic and interconnections; but instead of defining each transistor, you have to use a library of standard cells from a silicon vendor.

ALL GATE HAVE THE SAME HEIGHT

For each logic gate multiple layouts are available (the designer is not able to access and see them) each one with a characterization in terms of propagation delay, power consumption, area, etc. ⇨ PLACEMENT PROBLEM

good performance, good flexibility

Recap: to design a SoC, we must translate high level specifications into a layout description (that contains all the geometry info to fabricate the device). As discussed before, design is composed of 3 main phases:

1. Modeling: describe the system on a chip
2. Synthesis and optimization: provide an implementation
3. Validation: check if the implementation correspond to the model

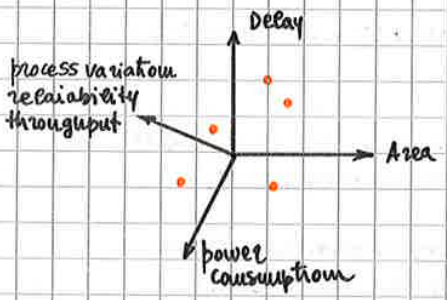
REPEATED AS MANY TIMES AS THE NUMBER OF ABSTRACTION LEVELS

SYNTHESIS WITH NO OPTIMIZATION HAS NO VALUE: optimization is the means to outperform manual design. Optimization has multiple objectives:

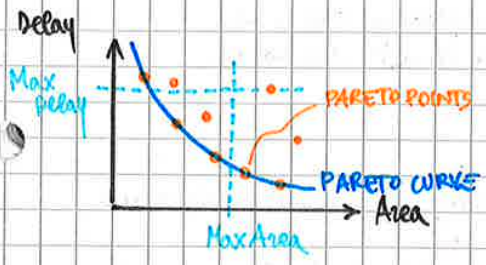
1. Performance (frequency, latency, throughput)
2. Power consumption
3. Area
4. Testability, dependability

for this reason, it is always a tradeoff: if you want to increase performance, you may want to replicate some functional blocks, increasing area and power consumption, etc.

Design space:



each point represent a possible implementation. How to filter those implementation? A best-all-around solution exist?
Unfortunately, as said before, a solution able to minimize all metrics does not exist. However, given a set of solutions, it is possible to identify some pareto points and to trace a pareto curve that connects all pareto points.



Since a global minimum does not exist, and since an algorithm is not capable to handle all the possible metrics, usually to find/explore pareto solutions, some fixed values are assigned to each metric.

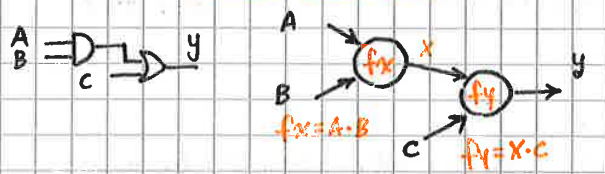
Abstract models

1) Circuit level → Netlist (module oriented / circuit oriented)
2) Logic level → Logic networks → BDD / AIG
   → FSM, state diagrams
3) Architectural, RTL level → Dataflow networks
   → sequencing networks

Logic networks:

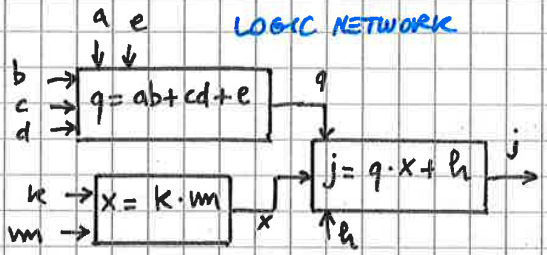
Mixed of behavioral and structural views, are an interconnection of **nodes** (implementing a Boolean function) and **edges** (representing data dependencies).
As always, the model must be acyclic and memoryless.
A special case is a **Mapped network**, where nodes correspond to library elements.

MAPPED NETWORK



nodes are mapped to gates in the library ➡ one-to-one correspondence between a local function and a gate (STRUCTURAL VIEW)

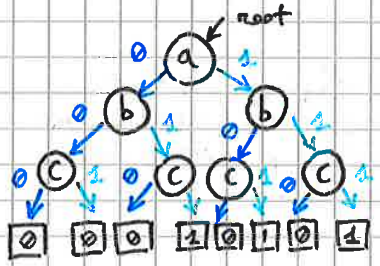
LOGIC NETWORK



Edges represent signal dependency: no info about topology (BEHAVIORAL VIEW)

BDD: Binary Decision Diagrams:

This model is a directed acyclic graph (DAG):
- GRAPH: set of nodes connected by edges
- DIRECTED: edges have direction
- ACYCLIC: no path lead to cycles.



1) Each node = decision on a single variable (a, b, or c)
2) Value of f(a,b,c) found at leaves (□□□□□···)
3) Each path from root to leaf is a row in truth table
4) Variable must appear in the same order along each path from root to leaf: by modifying order, the tree may be reduced/optimised (variable reorder).

REDUCTION RULES: merge equivalent subtrees
remove nodes with identical children

ADV

1. Canonical form: for each function only one tree exist
2. Efficient manipulation of Boolean f.

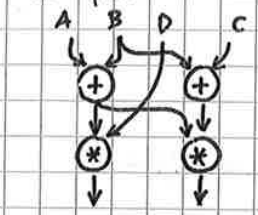
DIS

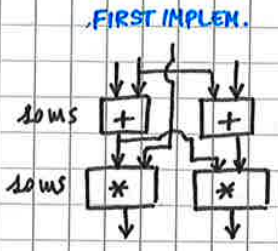
1. Size: a tree requires 2^m nodes, with m number of inputs EXPONENTIAL GROWTH!

Architectural level Synthesis CH 4

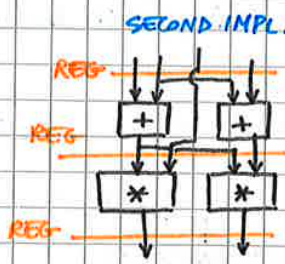
As already said, the higher the abstraction level the faster the design. However, selecting an high abs. level allows to reduce specification of details in the requirements, as well as simplifing the documentation production for a given design.
High level description allow also easy modifications and extensions.
Those advantages comes with a tradeoff: the design space is much larger and synthesis tool are more difficult and expensive to develop.

Example:

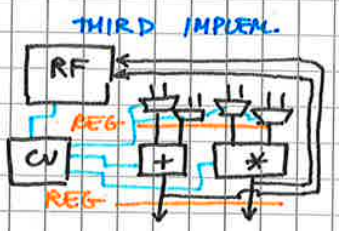




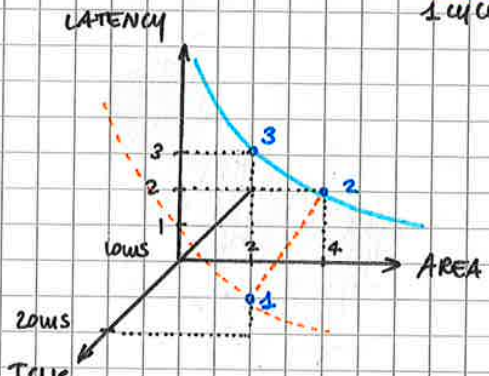
Tclk = 20ns
1 cycle

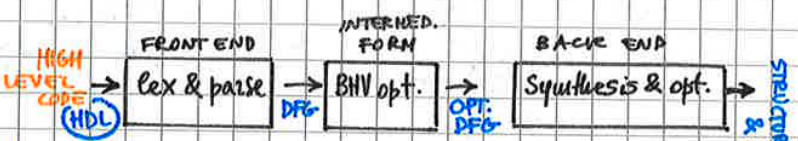


Tclk = 10ns
2 cycles



Tclk = 10ns
3 cycles
RESOURCE SHARING.
BUT NEED CU AND RF





Architectural Level Synthesis Flow

1. Translate HDL models into sequencing graphs (Compilation)
2. Behavioral level optimisations (transformation)
3. Constraint-driven synthesis & Optimization

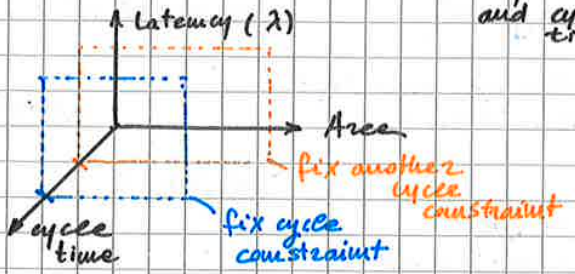
1. **Compilation:** basically it is composed of lexing and parsing to obtain a first tree implementation

2. **Behavioral level optimizations:** based on semantic-preserving transformations aiming at simplifying the model.
Those transformation can be organized into two categories:

A) Dataflow based:
- tree-height reduction
- constant and variable propagation
- common sub-expression elimination
- dead-code elimination
- operator-strength reduction
- code motion

B) Control flow based:
- Model expansion
- conditional expansion
- Loop expansion
- Block level expansion

GOAL: explore design space ⇒ MULTI-DIMENTIONAL: we consider only Area, latency and cycle time (clock period)



then an algorithm must take in account area and latency

1°APPROACH: fix area constraint and explore latency

2°APPROACH: fix latency constraint and explore Area

→ FIND PARETO POINTS !!

We need models for

- AREA ESTIMATION: $\sum_{i=0}^{\#RESOURCES} A_i$ this do not take into account the wiring overhead as well as RF and cu !! Good approximation if RESOURCE DOMINATED IMPLEMENTATION

- LATENCY ESTIMATION $\sum_{LONGEST\ PATH} d_i$ (delay of each HW resource) this do not take into account delay introduced by the wiring as well as register access and cu delay: good approximation only for RESOURCE DOMINATED IMPL.

NB: $d_0 = 0$, $d_n = 0$ since root and sink are special nodes useful for formalizing the problem.

ABSOLUTE

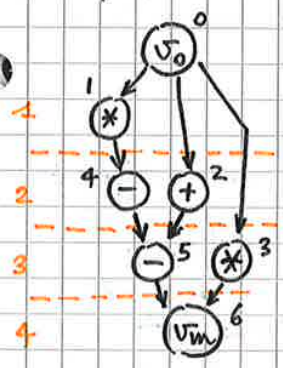
ALAP scheduling algorithm : LATENCY CONSTRAINED

This algorithm requires a user threshold on latency: we can provide the minimum latency discovered with ASAP algorithm : $\bar{\lambda} = 3$

USER DEFINED

```
ALAP ( G(V,E), λ̄ ) {
    schedule vm by setting t_m^L = λ̄ + 1
    repeat {
        select a node vi whose successors are all scheduled
        schedule vi by setting t_i^L = min{t_j} - d_i
                                       j : e_ij ∈ E
    } until ( v0 is scheduled).
    return (t^L);
}
```

Example:



$v_m \Rightarrow t_m = \bar{\lambda} + 1 = 4$

$v_5 \Rightarrow t_5 = \min\{t_m\} - d_5 = 4 - 1 = 3$

$v_3 \Rightarrow t_3 = \min\{t_m\} - d_3 = 4 - 1 = 3$

$v_4 \Rightarrow t_4 = \min\{t_5\} - d_4 = 3 - 1 = 2$

$v_0 \Rightarrow t_0 = \min\{t_1, t_2, t_3\} - d_0 = \min\{1,2,3\} - 0 = 1$

REMARKS: ALAP solves a latency constrained problem. Latency bound can be set to latency computed by ASAP algorithm: we can define **mobility** μ

$$\mu = t_i^{ALAP} - t_i^{ASAP}$$

and it represent the degree of freedom to shift down an operation before violating λ constraint

NB: $\mu_3 = t_3^{ALAP} - t_3^{ASAP} = 3 - 1 = 2$ you can move this unit by max 2 positions

$\mu_1 = \mu_4 = \mu_5 = 0$ OPERATION ON LONGEST PATH HAVE ALWAYS MOBILITY = 0

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 17 di 98

AREA CONSTRAINED:

An area constraint, expressed as the maximum amount of available resources, is present ⇨ LATENCY CHANGES DEPENDING ON WHICH KIND OF RESOURCES ARE AVAILABLE, AND IN WHICH QUANTITY

The dataflow graph we want to schedule has the following characteristics:

$$V = \{ v_i, \; i = 0, \dots, n+1 \}$$

$$E = \{ e_{ij}, \; i \neq j \; ; \; i,j = 0, \dots n \}$$

$$D = \{ d_i, \; i = 0, \dots, n \}$$

$$A = \{ a_k, \; k = 1, \dots \; \#RESOURCES \}$$ ← for each kind of resource, a maximum number of units is set by the user

Find a labeling function $\varphi : V \to \mathbb{Z}^+$ such that:

1) $\varphi(v_i) = t_i$

SEQUENCING RELATIONS 2) $t_i \geq t_j + d_j$

RESOURCE BOUNDS 3) $\left| \{ v_i \mid T(v_i) = k \text{ and } t_i \leq \ell \leq t_i + d_i \} \right| \leq a_k$

(k: RESOURCE TYPE, $k = 1, \dots, \#RES$; ℓ: STEP; a_k: MAX NUMB OF RESOURSE OF THAT KIND; $|\cdot|$: CARDINALITY)

The goal is to minimize the latency, hence minimize t_n (start time of sink)

To solve the area constained problem, exact and approximated algos are available: let's analyze them one by one.

AREA CONSTRAINED: INTEGER LINEAR PROGRAMMING (ILP)

ILP is a mathematical optimization or feasibility problem: the designer must formalize the scheduling problem as a system of equations. Some equations represent the constraints, some other identify an objective (or cost) function. Those equations have as unknowns binary decision variables $X = \{ x_{ie}, \; i = 0, \dots, n, \; e = 1, \dots, \bar{\lambda} \}$ (e: STEP)

The advantage of this approach is modularity, since new constraints can be easily added by introducing new equations inside the system, and existing constraints can be modified by rewriting corresponding equations

The main disadvantage is complexity of design: the designer must write those equations ⇨ different problem formulations can lead to different results (hence you may find a local minimum instead of the global one). Moreover, ILP works well up to few thousand variables.

Let's formalize the area constrained scheduling problem using ILP:

1) First, the start time of each operation must be unique

$$\sum_e x_{i,e} = 1 \qquad \forall i \in 0 \dots n$$

AREA CONSTRAINED : HU'S ALGO

It is a Greedy strategy-based algorithm, which returns exact solutions under some restrictive assumptions, that are:

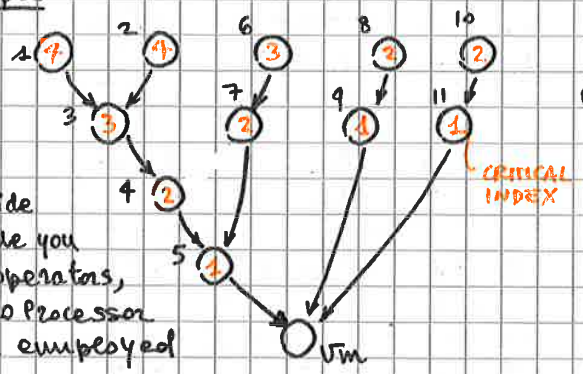
1) Availability of one single type of resource ⇨ Co-Processor { MULT, ADD/SUB, DIV, LOGIC }

2) The HW resource has 1 unit delay: no matter the operations it has to perform, the coprocessor will complete them in a single schedule time cycle.

```
HU (G(V,E), a) {        // a = # CO PROCESSORS
    label the nodes;
    l = 1
    repeat {
        U_l = { v_i | v_i has no predecessors or all predecessors have been already scheduled }
        S_l ⊆ U_l such that |S_l| ≤ a   // # RESOURCES = # CO PROCESSORS
        schedule all v_i ∈ S_l
        l++;
    } until (v_n is scheduled);
    return t_n;
}
```

YOU MUST PICK THE MOST CRITICAL OPERETIONS ACCORDING TO A CRITICALITY INDEX CORRESPONDING TO THE DISTANCE FROM SINK

Example:



NB: inside each node you have no operators, since a CO Processor must be employed

suppose # Coprocessors = a = 2

step	U_l	S_l
l = 1	1, 2, 6, 8, 10	1, 2
2	6, 8, 10, 3	3, 6
3	8, 10, 4, 7	8, 10
4	4, 7, 9, 11	4, 7
5	9, 11, 5	9, 11
6	5	5
7	v_n	v_n

$\lambda = 7 - 1 = 6$

suppose a = 3

step	U_l	S_l
1	1, 2, 6, 8, 10	1, 2, 6
2	3, 8, 10, 7	3, 8, 10
3	4, 7, 9, 11	4, 7, 9
4	5, 11	5, 11
5	v_n	v_n

$\lambda = 5 - 1 = 4$

suppose a = 1 : then $\lambda = 11$ since there are 11 operations of delay = 1 serially scheduled

LATENCY CONSTRAINED : LIST SCHEDULING

minimizes under $\bar{\lambda}$ constraint

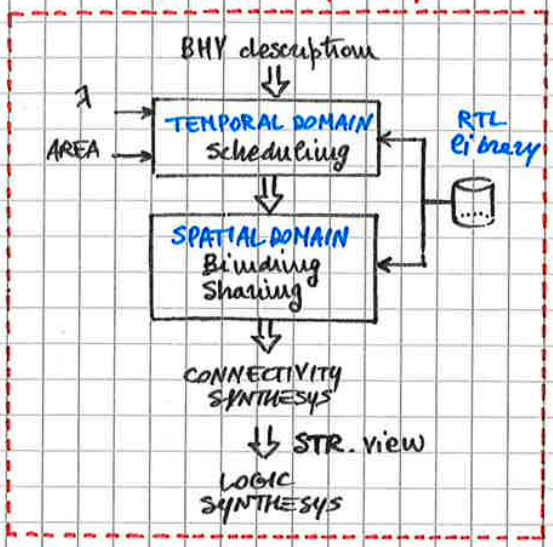
The previous list scheduling algorithm can be modified to tackle the complementary problem: given a certain latency, find an implementation that requires the minimum number of resources.
Remember that list scheduling is approximate method, hence the result may not represent the global optimum solution.

$\bar{\lambda}$ ← LATENCY CONSTRAINT DEFINED BY USER

```
list_R ( G(V,E) , λ̄ ) {
    A = {1, ..., 1} ;   initially all resources are set to minimum value = 1.
    Run ALAP( G(V,E), λ̄ ) to compute the latest start time t_i^L for
        each operation v_i
    If (t_0 < 1) {return FAIL} :   since ALAP was not able to meet
                                   the latency constraint λ̄, there is
                                   no chance list scheduling will
                                   perform better.
    l = 1 ;
    repeat {
        foreach (resource type k (1, ..., #RESOURCES)) {
            U_{l,k} = { v_i | v_i has no predecessors or all predecessors have
                              been executed completely }
            ~~T_{l,k} = { v_i | scheduled before current step and still running }~~
            S_{l,k} = { v_i | cannot be delayed anymore }
                      ↳ SLACK : t_i^L - l     difference between ALAP
                                              start time and current step
                                              If SLACK = 0 v_i MUST
                                              BE SCHEDULED, OTHERWISE
                                              λ̄ NOT MET
            schedule S_{l,k}
                If |S_{l,k}| > a_k , update a_k
            Schedule other ready operations not needing additional resources
        } l++;
    } until ( v_n is scheduled ) ;
    return (t_n, A)
}
```

CH6 Sharing and Binding

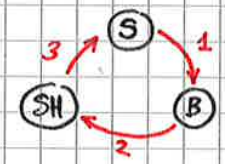
In chapter 4 the architectural level synthesis: as already said, the synthesys process can be classified into TEMPORAL DOMAIN (scheduling) and SPATIAL DOMAIN (binding and sharing).
Scheduling was explained in CH5, let's now focus on sharing and binding

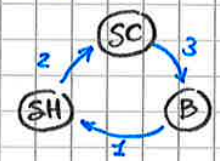


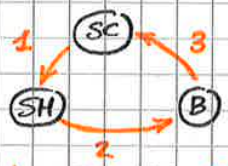
- **BINDING**: assigns a specific HW resources (contained inside the RTL library) to a specific operation in the DFG.
Usually RTL libraries contains multiple implementations/configurations: for example a large & fast or a small and slow.
The binding process can then produce different results based on the RTL library content: the best solution can be filtered by looking at some metrics (like AREA, power consumption, etc) BUT NOT LATENCY (ALREADY DEFINED DURING SCHEDULING).

- **SHARING**: define two or more operations able to share the same HW resource

Remember the circular dependency between scheduling, binding, sharing







WE'LL USE THIS ORDER !

BINDING/SHARING SCENARIOS:

1) **DEDICATED RESOURCES**: minimum λ (op executed ASAP), maximum area (one HW resource per operation)
   - ⇨ NO SHARING
   - ⇨ BINDING INTRINSICALLY DEFINED

2) **ONE MULTI-TASK RESOURCE**: like an arithmetic coprocessor, able to perform various kind of operations (+, −, <, *, ÷)
The HU'S ALGO (with area = 1) can be exploited for scheduling
   - ⇨ NO BINDING: there is not a direct correspondance between op and HW resource, since there is only one resource that performs everything
   - ⇨ MAXIMUM SHARING

3) **MANY MULTI-TASK RESOURCES**: same coprocessor as before, but now more than one is employed; scheduling done with HU'S ALGO (area > 1).
   - ⇨ PARTIAL BINDING
   - ⇨ SHARING IS AN OPTIMIZATION PROBLEM (NP complex)

4) **ONE RESOURCE × TYPE**: no coprocessors, but some adders, multipliers, divisors, etc.
For each component only one configuration is available.
   - ⇨ SCHEDULING: LIST SCHEDULING
   - ⇨ BINDING: PARTIALLY DEFINED
   - ⇨ SHARING: OPTIMIZATION PROBLEM

1. DEFINE PROBLEM: COMPATIBILITY
2. DEFINE ABSTRACT MODEL
3. DEFINE ALGO USING ABSTRACT MODEL

Sharing problem

Starting from a scheduled sequencing graph, where concurrency between operations is well defined, you want to identify two or more operations that can be bound on the same HW resource. The concept of **compatibility** must be introduced.

Two or more operations are compatible (and them can share the same HW) if:
1) they can be implemented by the same resource type
2) they are not concurrent (not used in the same step = not overlapping)

Clearly, since an operation can be mapped on a certain resource type only, the sharing problem benefits of **problem decomposition**: you can analyze sharing for each resource type. ⇨ REDUCE TIME AND COMPLEXITY OF SHARING PROBLEM

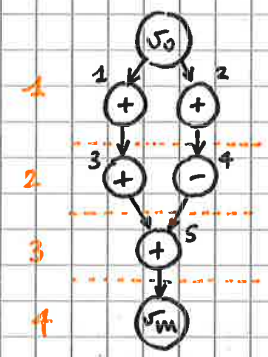
Abstract model: how to model compatibility?

COMPATIBILITY GRAPH G^+ ⟺ COMPLEMENT ⟺ CONFLICT GRAPH G^-

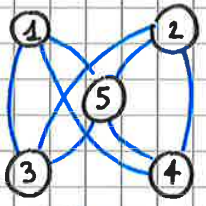
nodes: operations in DFG
edges: e_{ij} ⇒ i and j are compatible

nodes: operations in DFG
edges: e_{ij} ⇒ i and j are not compatible

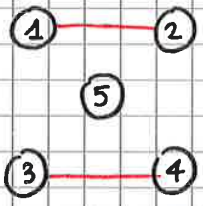
Example:



COMPATIBILITY G^+



CONFLICT G^-



As can be seen, G^+ and G^- are complementary

In order to find the optimum resource sharing you must analyze the G^+ or G^- graphs: in a G^+ graph, it is possible to identify **cliques** (subset of nodes that are fully interconnected between each other); maximum sharing correspond to minimum number of cliques. ⇨ PARTITIONING

In a G^- graph, a clique is a subset of nodes that has no edges between each other; maximum sharing correspond to minimum number of cliques. Usually this problem is tackled by assigning a color to each node such that connected nodes have different colors. ⇨ COLORING

Both partitioning and coloring are NP complex problems. To reduce problem complexity we have to refine the G^+ and G^- graphs into an equivalent representation: **a perfect graph**. Those graphs have polynomial time solutions to coloring and partitioning:
1) Golumbic's algo (comparability)
2) left-edge algo (interval) ← WE'LL SEE THIS ALGO IN THIS COURSE

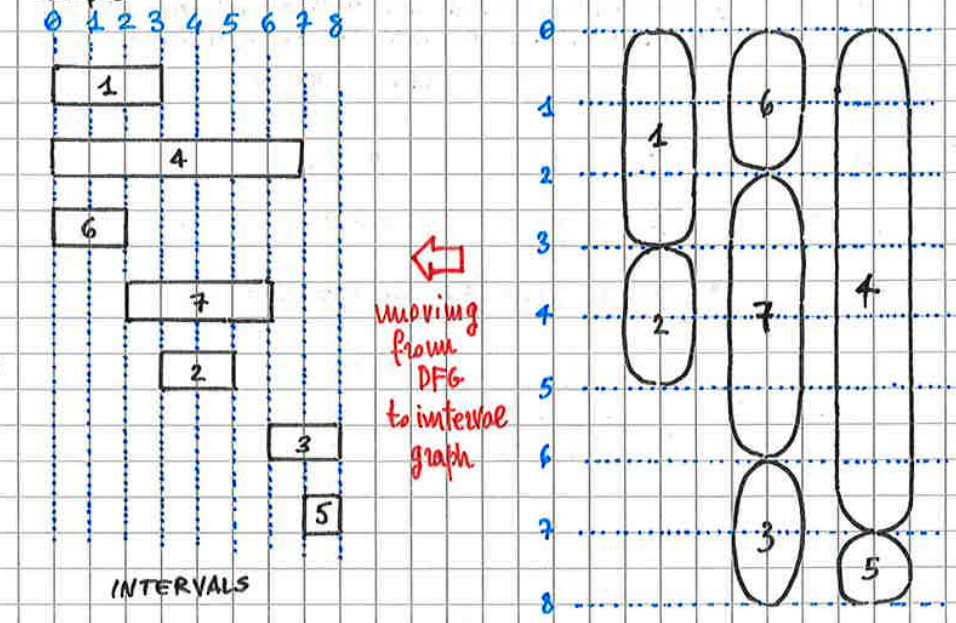
In particular, perfect graphs must specify the "source" of conflict/compatibility by means of **oriented edges** (→): a "partial order" is them defined, imposing that the operation at arrow tip must be executed after the one at the other edge end. ⇨ THIS ORDER INFO IS DERIVED FROM THE SCHEDULE

Left Edge Algorithm

This is a coloring algorithm applied on an interval graph. Hence, the goal is to find the minimum number of colours such that all disjoined nodes share the same colour.

Supported variables:
- $L = \{ I_i$: intervals ordered w.r.t. left edge $\}$
- $C = \{ c_i$: set of colours → unitarily is one colour only $\}$

Example:



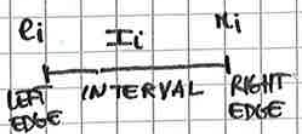
$L = \{ I_1, I_4, I_6, I_7, I_2, I_3, I_5 \}$ NB: is ordered!

```
left_edge (I) {
    I : intervals (not sorted)
    L : intervals (sorted)
    S : set of intervals assigned to a specific colour
    r : right edge of the last coloured interval
    C : number of colours currently used (initially is 1)

    Sort element of I in a list L in ascending order of l_i;
    C = 1;

    while ( there are some remaing uncoloured intervals ) {
        S = { };   // empty set
        r = 0;
        while ( there is at least one interval I_i such that l_i >= r) {
            S = S ∪ I_i   // add I_i to list S
            L = L - I_i   // remove I_i from L
            r = r_i       // update the variable r with the value of right edge
                          //    of the last coloured interval I_i
        }
        Assign colour to S
        Increment number of colours (C++)
    }
```

Supported variables

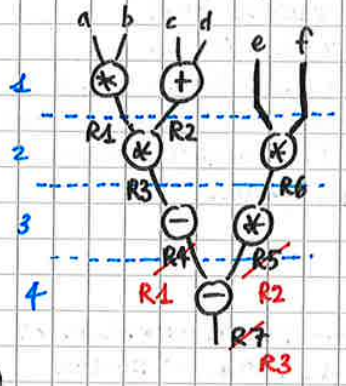


SHARE REGS { REDUCE CARDINALITY; REDUCE POWER CONSUMPTION; REDUCE SIZE AND COST }

Register Sharing problem

If no sharing is present, it is possible to hardwire the output of a module to an input of the following one (if they are scheduled in adjacent time slots)

However, if a certain degree of sharing is applied, registers becomes crucial! Moreover, registers are expensive and in a complex DFG you may need thousands of them; they switch every clock cycle also, consuming large power! For this reason, a register sharing problem must be considered

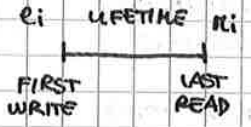


Registers must be used to hold values of variables

It is not possible to share R1 and R3, because the corresponding register has to provide stable inputs for the module (*) but at the same time has to sample its output.... Hence to share a register, it must be present a gap of at least 1 time unit (for example R5 and R4).

Let's formalize the problem: register sharing is based on lifetime of variables

Lifetime: interval between the time at which the variable is generated and the latest time at which it is referenced (as input of another operation). ↳ consumed

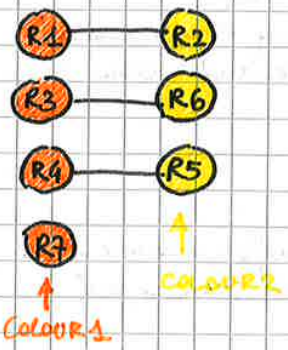


$$\text{lifetime} = t_{LASTREAD} - t_{PRODUCTION}$$

This problem can be solved resorting again on perfect graphs: in particular, for an interval graph we must define:
- Nodes → variables lifetime intervals
- Edges → lifetime overlaps

Using left edge Algo, find minimum number of registers to store all variables

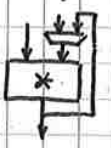
For the circuit above, the interval graph is



NB: after register sharing, you must introduce steering logic (like muxes) to correctly drive the shared regs...

Register sharing for iterative bodies

Some architectures have "variable loop": for example in a multiply and accumulate, the result goes as feedback to 1 input of the multiplexer...



To correctly share registers those loop connections must be considered: hence a circular-arc conflict graph is employed.

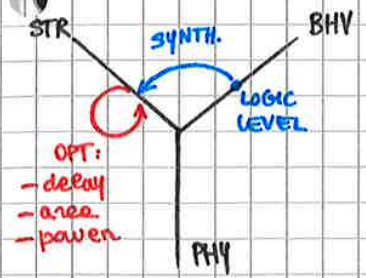
Bus sharing and binding

Find the minimum number of busses to accommodate all data transfer ⇨ conflict variables need dedicated connections

For the previous example, 1 BUS allows one colour transfer, 2 BUSSES allow all variables to be transferred.

Logic synthesis and optimization CH7

Quick recall:



input: behavioural description in HDL
primitives are booleen

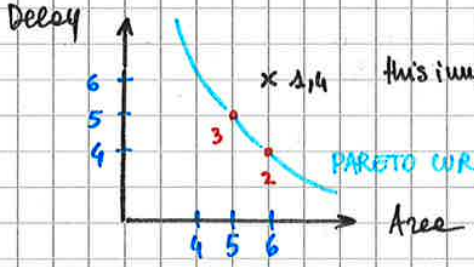
output: netlist (structural description) in HDL
↳ connection of logic gates (from tech. library)

GOAL: define and explore the design space

Example: $f = abcd$ tech library includes AND2 (area 2, delay 2), AND3(3,3)

possible implementations:

① area = 6, delay = 4
② area = 6, delay = 6
③ area = 5, delay = 5
④ area = 6, delay = 6



this implementations are not pareto points.

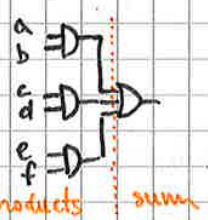
Going forward, in this lecture we'll see some background concepts, some definitions to formulate the problem and finally some exact and approximate algos to solve the problem.

Background:

SYNTHESIS OF TWO LEVEL CIRCUITS	SYNTHESIS OF MULTI LEVEL CIRCUITS

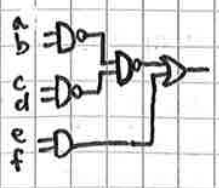
2 LEVEL: SOP

$f = ab + cd + ef$



MULTI LEVEL:

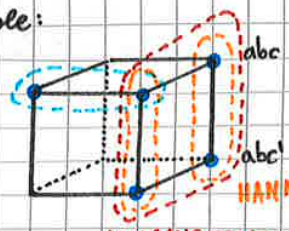
$f = \overline{\overline{ab} \cdot \overline{cd}} + ef$



no restrictions on { kind of gates / number of levels }

Nowadays no one is using two-level circuits anymore, but two level optimization is used within multi level implementations

Example:



$abc \cup abc' = abc + abc' = ab$

HANNING DISTANCE = 1

MERGING TWO IMPLICANTS

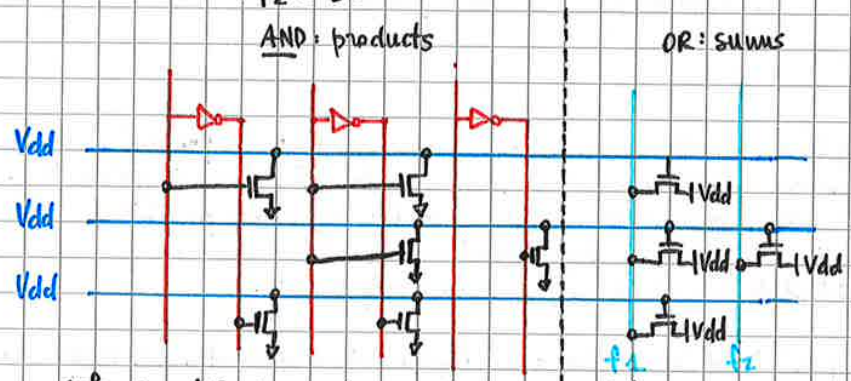
MERGING TWO MINTERMS

PLAs (programmable logic array)

$f: B^3 \to B^2 \quad \begin{cases} f_1 = a'b' + b'c + ab \\ f_2 = b'c \end{cases}$

AND: products

OR: sums



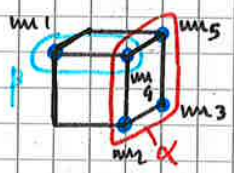
horizontal wires = # products
vertical wires = # literals

horizontal wires = # products
vertical wires = # outputs

If you want to reduce area:

1) find a representation with the minimum number of implicants ⇒ reduce # products
2) within a product, you want to reduce the number of transistors ⇒ reduce # literals

Minimum cover: find minimum number of implicants that cover all minterms



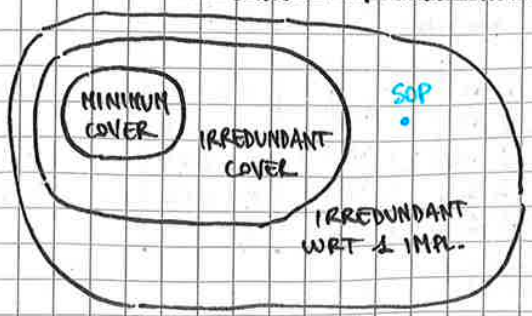
from $f = m_1 + m_2 + m_3 + m_4 + m_5$ to $f = \alpha + \beta$

5 ROWS, 15 TRANS. 2 ROWS, 3 TRANS

Irredundant cover (minimal cover): none implicant can be dropped

Obviously, if any of the implicant is dropped, at least one minterm is not cover anymore. This solution may be not the optimal one. not

Irredundant cover with respect of one implicant: none implicant can be contained into another implicant



The second step of reduction consist on finding an appropriate vector X such that
- $A \cdot X \geq \mathbb{1}$
- $e_1(x)$ is minimum

In practice you must select enough columns to cover all minterms, but at the same time you want to minimize the cardinality of X (hence select the minimum number of cols that cover all minterms).

COVERING PROBLEM ⇒ NP HARD PROBLEM

in the previous example, $X = [1111]^T$, we obtain $Y = [12221]^T$.
But, by selecting $X = [1101]^T$, we obtain $Y = [12111]^T$.
In both cases, the first requirement ($Y \geq \mathbb{1}$) is met, but the second solution has $e_1(x)$ smaller:

$$e_1([1111]^T) = \sum x_i = 1+2+2+2+1 = 8$$
$$e_1([1101]^T) = \sum x_i = 1+2+1+1+1 = 6$$

2-LEVEL OPT.
- EXACT METHODS: they find the minimum cover (slower, high complexity)
  - Quine McCluskey
  - Petrick's
  - Espresso
- APPROXIMATE: they find the minimum irredundant cover (weaker but faster, lower complexity)

The exact methods use need the prime table as a starting point; they all reduce the prime table using different approaches:
1) Quine McCluskey: tabular methods
2) Petrick's: POS clauses
3) Espresso: tabular methods + branch & bound

NB: the "reduce" = optimization problem becomes a "covering problem"!

	A	B	C	D	E	F
m1	1	1	0	0	0	0
m4	0	0	0	0	1	0
m5	1	0	0	0	1	0
m6	0	0	0	0	1	1
m7	0	0	0	0	1	1
m9	0	1	1	0	0	0
m11	0	0	1	1	0	0
m14	0	0	0	0	1	1
m15	0	0	0	1	0	1

1°step

2°step: all miniterms contained in the removed primes are removed from the table

	A	B	C	D
m1	1	1	0	0
m9	0	1	1	0
m11	0	0	1	1

After the essential primes are removed, we must continue to reduce the table.
A rule of thumb suggests to remove the columns with largest number of ones first, because they cover a larger number of miniterms.
In our example B and C are valuable choices.
Let's remove C:

	A	B	C	D
m1	1	1	0	0
m9	0	1	1	0
m11	0	0	1	1

	A	B	D
m1	1	1	0

we remove A for example

$C_{min} = \{A, C, E, F\}$

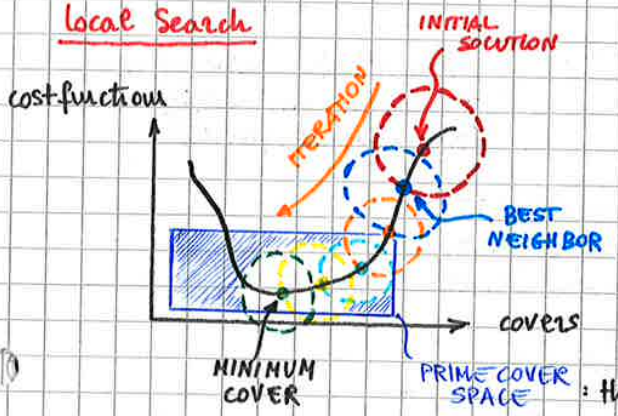
NB: we found a minimum cover, but it is not the only possible one. For example:

$C_{min}^{2} = \{B, C, E, F\}$, $C_{min}^{3} = \{B, D, E, F\}$, ...

Heuristic optimisation

Approximate methods are used as internal engines within multi-level optimization and synthesis tools. For this reason, they must be faster (w.r.t exact methods); hence bottlenecks like prime generation and storage and covering problems are avoided. Even if the heuristic is able to provide "reasonably small" irredundant covers in a "reasonable time", the overall quality of the final cover may be not optimal (local minimum instead of global one).

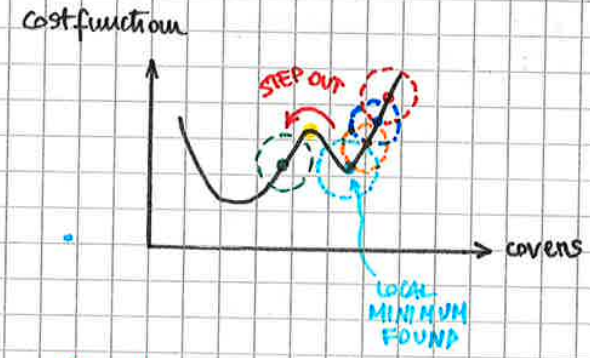
Local Search



Basic strategy:

1) Starting point is a random SOP (cover)
2) Search for feasible neighbors whose cost is lower
3) If one is found, iterate (go back to 1.)

: the exact methods start here, inside the prime cover space.



Drawback:

1) Local search may stuck at local minimum (no convex problem)
2) Need a mechanism to step out (accept worse solution)

Operators:

ITERATE
1) Expand: make implicants prime → make prime cover.
2) Irredundant: make cover irredundant
3) Reduce: reduce size of each implicant while preserving unit-terms covering, returning to a non-prime cover

- Expand and irredundant, make the cover prime and irredundant, therefore the best neighbor is found
- Reduce: perturb the cover finding some worse covers to step out of a local minimum

The possible issues are

1) Validity check of expansion
2) Find the best "direction" to move to
3) Order of expansion (how many literals drop)

Heuristic optimisation tasks

1) Tautology: check if a function is always true

2) Containment: check if a cube is contained in a cover
this task is essential to understand if a given covers are the minterms. It is also used by the "irredundant" operator to find cubes or implicants contained in other implicants of the cover

3) Complementation: compute the complement of a function
OFFSET = complement (ONSET+DCSET)
OFFSET is crucial during "expand" operator, to understand if a given implicant is valid

Background:

(1) given a function $f: B^m \to B^m$, the shannon decomposition w.r.t a variable x_i is:

$$f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$$

f_{x_i}: cofactor w.r.t x_i — $f_{\overline{x_i}}$: cofactor w.r.t $\overline{x_i}$

where
$f = f(x_1, x_2, \dots, x_i, \dots, x_m)$
$f_{x_i} = f(x_1, x_2, \dots, 1, \dots, x_m)$
$f_{\overline{x_i}} = f(x_1, x_2, \dots, 0, \dots, x_m)$

Example:
$f = ab + bc + ac = a(b+c) + \overline{a}(bc)$
$f_a = b + bc + c = b + c$
$f_{\overline{a}} = bc$

(2) a given function f is unate w.r.t. x_i if $f_{x_i} \geq f_{\overline{x_i}}$

if x_i is contained into f always with same polarity, then if
- x_i always positive in f → positive unate
- x_i always negative in f → negative unate

Example:

$f = ab + bc + ac$ → { positive unate wrt a; positive unate wrt b; positive unate wrt c } → f is positive unate

$f = a'b + bc + a'c$ → { negative unate wrt a; pos unate w.r.t b; positive unate wrt c } → f is binate

$f = a'b + bc + ac$ → { binate wrt a; pos unate wrt b; pos unate wrt c } → f is binate

Matrix representation of logic covers

Approximate methods do not uses prime tables (big size, difficult build); instead a different table approach is used:

the table is composed by one row per implicant and one column per variable.

encoding x_{ij} = { 00 if intersection between var j and implicant i is void; 01 if var j is positive in implicant j; 10 if var j is negative in implicant j; 11 if var j is a don't care in implicant j }

This encoding allows a very eigth and easy to build table: one cher contain 4 elements!

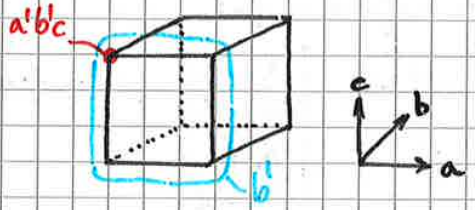
Moreover, binary operations are very fast and can be parallelized:
- BITWISE AND: intersection
- BITWISE OR: supercube
- BITWISE COMPLEMENT: rise a don't care

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

	a	b	c	d
a'd'	10	11	11	10
a'b	10	01	11	11
ab	01	01	11	11
ac'd	01	11	10	01

Example: intersection between $f = a'b'c$ and $g = b'$

	a	b	c	
a'b'c	10	10	01	BITWISE AND
b'	11	10	11	
	10	10	01	= a'b'c



intersection between $f = a'b'c$ and $g = b$

	a	b	c	
a'b'c	10	10	01	BITWISE AND
b	11	01	11	
	10	00	01	

VOID: means the intersection is empty

Example: don't care rise: from ac'd to * c'd

	a	b	c	d	
ac'd	01	11	10	01	BITWISE OR
BITWISE COMPL. OF a	10	00	00	00	
	11	11	10	01	= c'd

remember: a = 01 11 11 11

the expected value

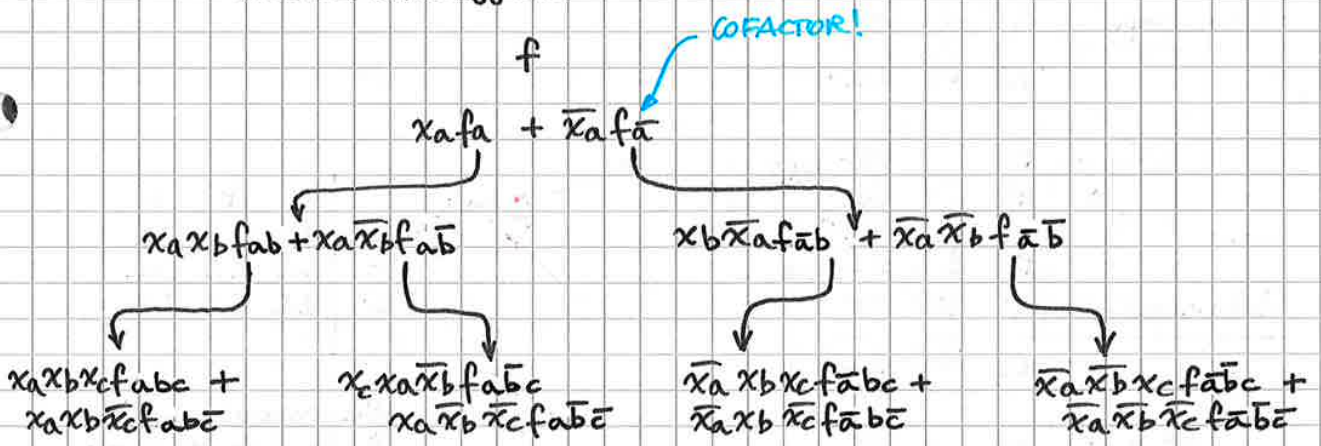
Tautology check

To check if a function is always true, a recursive algorithm must be used. For each variable of the function, the cofactors must be computed: if they are are tautologies, the entire function is a tautology

Terminal cases

1) The cover matrix has a row of ones ⇨ TAUTOLOGY
2) The cover matrix has a column of zeros ⇨ NO TAUTOTOLOGY
3) The cover metrix depends on one variable and there is no column of zeros ⇨ TAUTOLOGY

Example: suppose f depends on 3 variables a, b, c.
to check if f is a tautology, you must explore all the tautology tree:



As soon as one cofactor is not a tautology, the entire function f is not a tautology and therefore no need for computing the remaining cofactors.

If a cofactor is a tautology, you can skip the computations of child cofactors, because they'll be tautologies as well. However, all cofactors must be a tautology to prove that f is a tautology

Computation of $fa\bar{b}$

1° row b'	11 01 11 11 10 11	AND
	11 00 11	drop
	VOID	
2° row b'	11 11 01 11 10 11	AND
	11 10 01	OR
bitw compl b'	00 01 00	
	11 11 01	
3° row b'	11 10 10 11 10 11	AND
	11 10 10	OR
bitw compl b'	00 01 00	
	11 11 10	

$fa\bar{b}$:

	a	b	c
c	11	11	01
c'	11	11	10

fab is a tautology, because it depends on a single variable and there is no column of zeros

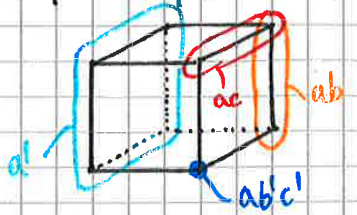
⇩

Since both fab and $fa\bar{b}$ are tautologies, fa is a tautology as well. What is missing is a check on cofactor $f\bar{a}$:

Computation of $f\bar{a}$

1° row a'	01 01 11 10 11 11	AND
	00 01 11	drop
	VOID	
2° row a'	01 11 01 10 11 11	AND
	00 11 01	drop
	VOID	
3° row a'	01 10 10 10 11 11	AND
	00 10 10	drop
	VOID	
4° row a'	10 11 11 10 11 11	AND
	10 11 11	OR
bitw compl a'	01 00 00	
	11 11 11	tautology! (row of ones)

Since fa and $f\bar{a}$ are tautologies, f is a tautology! In fact:

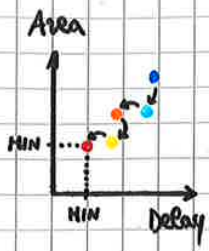


The cube is fully covered, therefore it is a tautology

Multilevel optimization and logic synthesis CH8

Summary: it is possible to implement a boolean function using a two-level representation (sum of products) or a multilevel representation. The main differences between the two representations are the number of possible levels (2 vs ≥2) and the boolean gates used (AND, OR vs any restriction, as long is contained inside the tech library).

Motivations: why choose multilevel logic over two level one? Because the implementation of circuits as multiple-level logic networks allow BETTER QUALITY of final result.

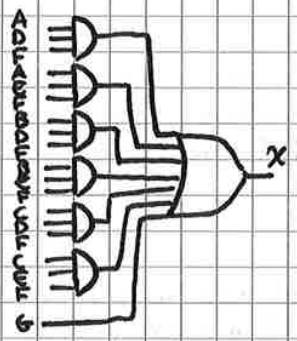


In a two-level optimization problem, area and delay are compatible objectives: you reduce area by minimizing the number of products, that allows for a smaller and moreover faster OR gate (lower # inputs ⇒ lower delay). This is the ONLY POSSIBLE SOLUTION!

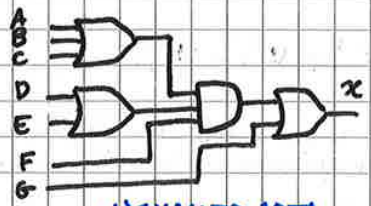
In a multilevel representation, you can explore the whole design space instead: hence you can implement a slow and small circuit up to a big but fast one. Moreover, usually the number of gates (as well as their # inputs) is reduced wrt a SOP implementation.

Example: $X = ADF + AEF + BDF + BEF + CDF + CEF + G$

$X = (A+B+C)(D+E)F + G$

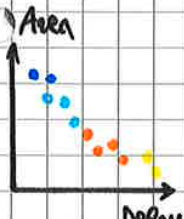


SOP representation



1) SMALLER AREA
2) REDUCE # INTERCONNECTIONS (∝ AREA)
3) MAYBE SLOWER

multilevel implementation



The ability of exploring the whole design space is the key for Multiobjective optimization: multiple constraints can be specified ⇒ explore design space to find the best solution

The importance of multilevel synthesis grew in parallel with the growth of foundaries for the semiconductor market: nowadays silicon vendors provides a predesigned set of cells modeling basic logic (INV, NAND, NOR, ...) that is fully characterized (AREA, DELAY, POWER CONSUMPTION, ...) and optimized for the specific manufacturing process of that silicon vendor. The customer can not access the cell layout, and after synthesis and layout has a circuit ready to be manufactured.
Because transistor technology has became so complex (and expensive) the customer is not able to design its own cell; and because circuits are more and more complex he does not want to! This design approach is called standard cell design

Estimators How to define metrics like area, delay, power consumption for a multilevel optimisation problem?

- **Area**: there is a direct proportionality between complexity and # literals. It is an easy and well accepted estimator
- **Delay**: it depends on the set of vertices V_G (proportional to # literals) and set of edges E (that is estimated using MAX+SUM propagation explained later).
- **Power**: it is possible to define a probabilistic model based on switching activity at each node + capacitive loads (fanout)

Methodology

METHODS
- EXACT (HIGH COMPLEXITY, IMPRACTICAL)
- APPROXIMATE
  - RULE-BASED
  - HEURISTICS BASED ON ALGORITHMIC TRANSFORMATIONS

The goal of multilevel synthesis is to obtain an equivalent representation of a given logic function that is optimal wrt a cost function.

With the topology-degree of freedom, even simplest problems becomes computationally hard. For this reason there are few exact methods (none of which is practical in real life) and some approximate optimization methods, classified in:
- Rule-based methods
- Heuristic algorithms

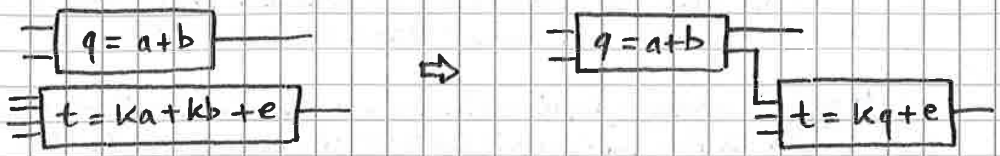
① **Rule based methods** rely on a rule database, composed by input patterns and an equivalent optimal implementation wrt a cost function

pattern	optimum equivalent pattern

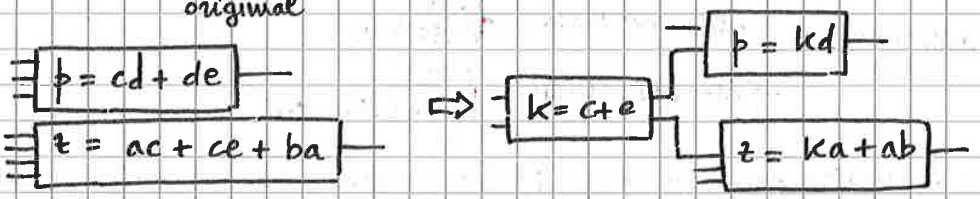
The method performs an algorithmic transformation, identifying patterns in the logic network and optimizing the design step by step

The two main drawbacks are:
1) BUILD AND MAINTAIN THE DATABASE, by selecting the most useful patterns for a particular design. Very time consuming and there is no evidence that it will affect positively the optimization
2) TECHNOLOGY CHANGE: usually a new tech has different layouts to take into account the quantum effects: hence the previous may not apply the new tech ⇒ REBUILD DATABASE

Substitution: simplify a local function by using an additional input that was not previously in its support set. This transformation simplify local functions, but increase famout



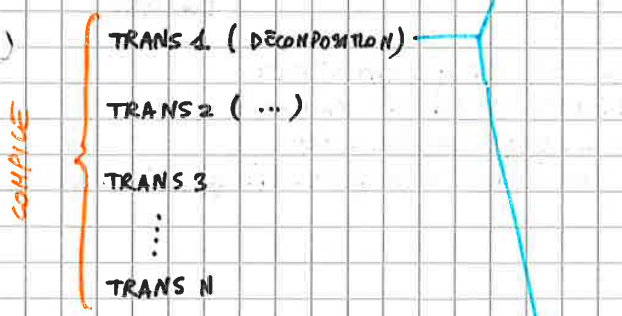
Extraction: this is one of the most powerful transformations; it finds a sub-expression of two (or more) expressions. It extract the sub expression, making a new node in the network and simplify the local functions.



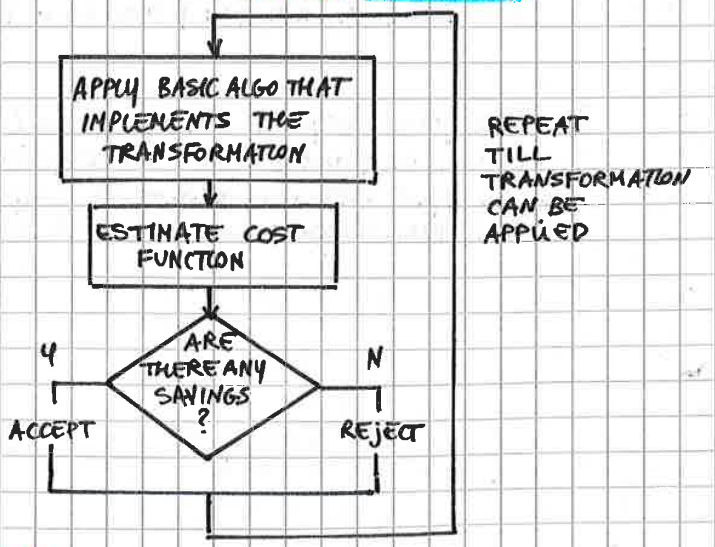
Simplification: simplify a local function by means of an heuristic minimizer like Espresso. This may modify the famin of the target node.

Algorithmic approach: implementation: we have seen what each transformation performs. But how a transformation is implemented?

Script composed of a sequence of transformations:



HOW A TRANSFORMATION IS IMPLEMENTED?
AS AN OPERATOR.



NB: some transformations require extra inputs (alongside the logic boolean network). Those inputs are the variables tuning the cost function.

Algebraic division

A function $f_{DIVISOR}$ is an **algebraic divisor** of another function $f_{DIVIDEND}$ if exist a function $f_{QUOTIENT}$ such that $f_{DIVIDEND} = f_{DIVISOR} \cdot f_{QUOTIENT} + f_{REMAINDER}$ if and only if:

1) $f_{DIVISOR} \cdot f_{QUOTIENT} \neq \{0\}$: this is trivial, because otherwise the result of the division is equal to $f_{REMAINDER} = f_{DIVIDEND}$. This condition must be verified also for the Boolean division.

2) $sup(f_{DIVISOR}) \cap sup(f_{QUOTIENT}) = \{\emptyset\}$ the intersection between the support sets must be void, otherwise some identity rules are needed (and are not allowed in Algebraic domain) - This property is not valid for Boolean division

Examples:

1) $f_{DIVIDEND} = ac + ad + bc + bd + e = (a+b)(c+d) + e$
$f_{DIVISOR} = a+b$
$f_{QUOTIENT} = c+d$
$f_{REMAINDER} = e$
$sup(f_{QUOTIENT}) \cap sup(f_{DIVISOR}) = \{\emptyset\}$ OK!

2) $f_{DIVIDEND} = a + bc$
$f_{DIVISOR} = a+b$
$f_{QUOTIENT} = a+c$
$f_{REMAINDER} = \{\emptyset\}$
$sup(f_{QUOTIENT}) \cap sup(f_{DIVISOR}) = \{a\}$ NOT OK

Let's check: $f_{DIVIDEND} = f_{DIVISOR} \cdot f_{QUOTIENT} + f_{REMAINDER}$
$= (a+b)(a+c) = a + ac + ab + bc =$
$= a(1 + b + c) + bc = a + bc$

$1 + b + c$: IT HAS NO MEANING IN ALGEBRAIC DOMAIN... (IDENTITY RULE)

$= a + bc$: IN BOOLEAN DOMAIN

The Algebraic division is the basic operator used by 3 transformations:

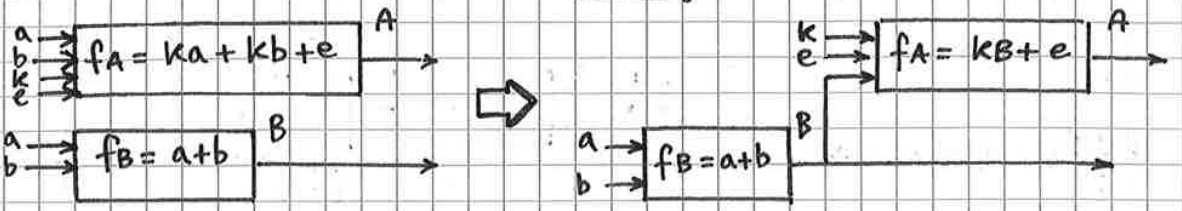
1) **Decomposition** : a node (local function) is split into two smaller ones. The family node is a divisor for the other node

2) **Substitution** : a local function is simplified by using as additional input another node of the network. The additional node is a divisor for the original one

3) **Extraction** : a common sub-expression between two or more nodes becomes a stand alone node. This new node is a common divisor of the two original local functions

Algebraic substitution

The substitution transformation aims at simplify the local function of a given node by using an additional input previously not present in the support set. The additional input comes from an existing node of the network



The substitution, as can be easy understood, is a straightforward application of algebraic division. The algorithm can be summerized in the following steps:

1) Consider all possible couples of nodes } complexity $\propto (\text{nodes})^2$
2) Apply division (in any order) between them } quadratic
3) Use filters based on previous theorem to reduce computation
4) If a quotient is not void, evaluate area and delay gain
5) If they are higher than the user threshold, modify the network by increasing the fanout of one node (DIVISOR) and substituing of $f_{DIVIDEND}$ with
$f_{QUOTIENT} \cdot f_{DIVISOR} + f_{REMAINDER}$
node label

A basic algorithm:

```
Substitution (G(V,E)) {
    foreach node i of network G {
        foreach node j of network G such that i≠j {

            A = set of cubes of fi
            B = set of cubes of fj

            if (A and B pass the filter test) {

                (Q,R) = Algebraic_division (A,B);

                if (Q ≠ ∅) {

                    fQUOTIENT = sum of cubes of Q
                    fREMAINDER = sum of cubes of R

                    if (substitution is advantageous) {

                        fi = j fQUOTIENT + fREMAINDER

                    }
                }
            }
        }
    }
}
```

2) Recursive algorithm: it is based on the property that a "kernel of a kernel is still a kernel" (a kernel is of level "n" if has "n" recursive kernels)

two algorithms based on recursion are proposed:
- R_KERNEL (with no pointer) — it does some redundant calculations
- KERNEL (complete algo) — more optimized, exploit commutativity of vars

Both algorithms uses a function CUBES(f, ci) which returns the cubes of f whose literals include those of cube ci

NB: kernel extraction is valid only if f is cube free ⇨ if f is not a kernel, we cannot extract kernels...

```
R_KERNEL(f){
    K = {∅}     // kernel set initialized to void
    foreach variable vi in sup(f){
        if( |CUBES(f, vi)| ≥ 2 ){        ← we are interested in cube free expressions, that implies multi-cube expressions ⇨ hence at least two cubes before performing division
            co-kernel → C = maximal subcube among the cubes returned by CUBES(f, vi)
            K = K ∪ R_Kernel(f/C)        ← RECURSION; DIVISION!
        }
    }
    K = K ∪ f    // the function itself is a kernel as well
}
```

Example:

$f = ace + bce + de + g$ $\quad sup(f) = \{a, b, c, d, e, g\}$

function : var	CUBES(f, var)	co-K	K	
f : a	$\lvert\{ace\}\rvert \geq 2$ NO	/	/	
f : b	$\lvert\{bce\}\rvert \geq 2$ NO	/	/	
f : c	$\lvert\{ace, bce\}\rvert \geq 2$ YES	ce	R.Kernel(f/ce)	recursion ev1
a+b : a	$\lvert\{a\}\rvert \geq 2$ NO	/	/	
a+b : b	$\lvert\{b\}\rvert \geq 2$ NO	/	/	
				K = {a+b}
f : d	$\lvert\{de\}\rvert \geq 2$ NO	/	/	
f : e	$\lvert\{ace, bce, de\}\rvert \geq 2$ YES	e	R.Kernel(f/e)	recursion ev1
ac+bc+d : a	$\lvert\{ac\}\rvert \geq 2$ NO	/	/	
ac+bc+d : b	$\lvert\{bc\}\rvert \geq 2$ NO	/	/	
ac+bc+d : c	$\lvert\{ac, bc\}\rvert \geq 2$ YES	c	R.Kernel(f/c)	recursion ev2
a+b : a	$\lvert\{a\}\rvert \geq 2$ NO	/	/	
a+b : b	$\lvert\{b\}\rvert \geq 2$ NO	/	/	
				K = K ∪ {a+b} = {a+b}
ac+bc+d : d	$\lvert\{d\}\rvert \geq 2$ NO	/	/	
				K = K ∪ {ab+bc+d} = {a+b, ab+bc+d}
f : g	$\lvert\{g\}\rvert \geq 2$ NO	/	/	

Theorem (Braytow and Mc Mullem)

Two expressions fa and fb have a common multiple-cube divisor fd if and only if there exist at least 2 kernels, one from K(fa) and one from K(fb) for which $|K(fa) \cap K(fb)| \geq 2$. The common divisor to be extracted must be the one with the <u>largest cardinality</u>.

This theorem has two consequences:

1) It reinforce the idea that kernel set computation is necessary for the kernel intersection and hence the transformation extraction
2) Introduces some filter conditions:
   a) if an expression (local function) has no kernels, the corresponding node can be dropped from consideration
   b) if kernel intersection is void, then the search for common sub-expressions can be dropped

Finally, it is possible to define a basic algorithm for <u>algebraic extraction</u>, by combining kernel extraction, kernel set intersection and substitution:

```
Multiple_Extraction (G(V,E), m, k) {

    do {
        foreach node compute the kernel set of level ≤ k
        Compute all the kernel set intersections (Brayton theorem)
          and save them into a list L
        Rank the list L based on cardinality (descending order)

        for (i = 1 to m) {

            create a new node j with local function f_j = L(i)
              and add it to the network

ALGEBRAIC   Replace all nodes whose local function has f_j as
SUBSTITUTION  a divisor with (j) f_QUOTIENT + f_REMAINDER
                                  NODE LABEL

            Update the kernel set of the modified nodes by
              removing the kernel(s) in which the
              common divisor is contained
        }

    } while (I is not empty)
}
```

Final comments:

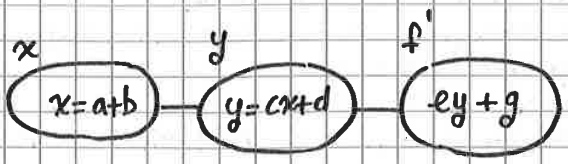
1) k represent the level for kernel extraction: basically, for each nested recursion, the level k increase. Hence this parameter set a bound on the complexity of kernel extraction
2) m represent the amount of algebraic extractions performed before recomputing all kernel sets and kernel sets intersections. Hence it is a tradeoff between quality of result (small m) and execution speed (big m).

Example: $f = ace + bce + de + g$ $\quad K(f) = \{ a+b,\ ac+bc+d,\ \underbrace{ace+bce+de+g}_{\text{decomposition is useless cause function itself.}} \}$

1° approach: largest kernel

decomposition(f): $y = ac+bc+d$

$f' = ey + g$



recursion on y: $x = a+b$

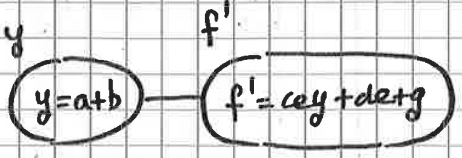
$y' = cx+d$

$f'' = f' = ey+g$

2° approach: random kernel

decomposition(f): $y = a+b$

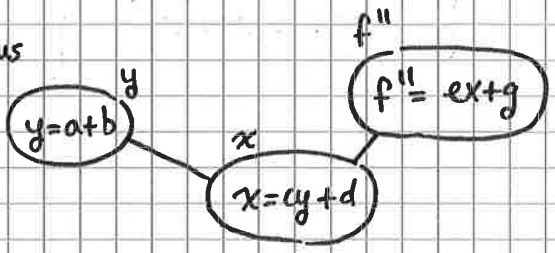
$f' = ce\,y + de + g$



recursion on y: two possible operations

decomposition(f'): $x = cy + d$

$f'' = ex + g$

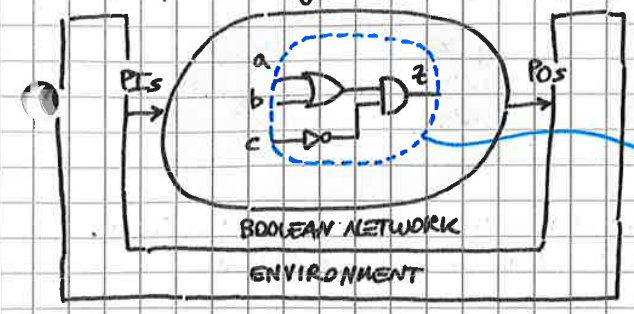


recursion on x: two possible operations

To achieve the same result, we need to run decomposition a larger number of times: more computationally expensive

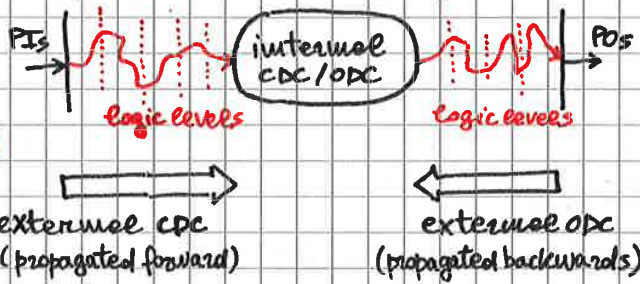
Summary on Algebraic methods: they abstract local functions as polynomials and perform division-based transformations. All transformations are reversible, hence it is easy to trade between area, delay and other metrics. They are fast and widely used methods, but they are weaker with respect to boolean methods.

It is possible to generalize the concept of external don't cares:



it is possible to focus on a part of the boolean network, transforming the rest of the boolean network into the environment! It is then possible to define internal don't cares (ODC and CDC)

This approach allows to work with a much finer granularity, at the point that for each node of the network, an ODC and CDC set can be defined. Internal and external don't cares are different: who is the responsable? The topology is the responsable; it may inject/remove don't care conditions by moving forward through the levels of the Boolean network:



For any generic node of the boolean network, the controllability of that node is defined by the external CDC propagated forward till that point - The observability of any node is given by the external ODC propagated backward till that point. Each logic gate on which ODC/CDC propagates may inject/remove new don't care conditions

Satisfiability don't care conditions:

The topology is responsible for the introduction/removal of internal don't care conditions. To find controllability don't cares generated by a local function (≡ node), satisfiability don't care conditions must be used.

satisfiability don't care conditions: $SDC = \sum_{\text{all internal nodes}} x \oplus f_x$ (x: output and node label; f_x: local function of the node)

This simple condition represent all possible input/output combinations that are never produced ⇨ Hence don't care conditions. In fact, the XOR can be seen as the "opposite of a comparator": each condition for which the two inputs (x and f_x) are different, makes the XOR true.

Example:

$$SDC_x = x \oplus (a'+b) = x'(a'+b) + x\overline{(a'+b)}$$

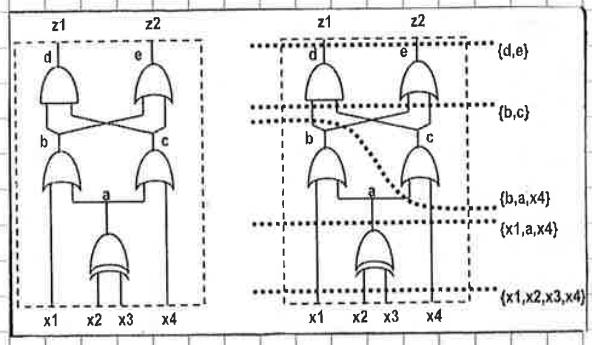
(De Morgan: $\overline{a'+b} = ab'$)

$$= x'(a'+b) + x(ab') = x'a' + x'b + xab'$$

Look closely: the first two conditions say that if one input is true, the output is false (never produced) the last condition says that the output is true if both inputs are false (never produced)

Example of internal CDC computation

Compute CDC_{out} knowing $CDC_{in} = \bar{x}_1\bar{x}_4$



$cut = \{PIs\} = \{x_1, x_2, x_3, x_4\}$

$CDC_{out} = CDC_{in} = \bar{x}_1\bar{x}_4$

---------- ITERATION 1

$cut = \{x_1, x_2, x_3, x_4, a\}$

$SDC_a = a \oplus f_a = a \oplus (x_2 \oplus x_3)$

$CDC_{out} = CDC_{out} \cup SDC_a = \bar{x}_1\bar{x}_4 + a \oplus (x_2 \oplus x_3)$

$D = \{x_2, x_3\}$

$$\mathcal{C}(CDC_{out})|_{x_2} = \bar{x}_1\bar{x}_4 + a \oplus (x_2 \oplus x_3)\Big|_{x_2} \cdot \bar{x}_1\bar{x}_4 + a \oplus (x_2 \oplus x_3)\Big|_{\bar{x}_2} =$$

$$= (\bar{x}_1\bar{x}_4 + a \oplus \bar{x}_3)(\bar{x}_1\bar{x}_4 + a \oplus x_3)$$

$$\mathcal{C}(CDC_{out})|_{x_3} = \left[(\bar{x}_1\bar{x}_4 + a \oplus \bar{x}_3)(\bar{x}_1\bar{x}_4 + a \oplus x_3)\right]\Big|_{x_3} \cdot \left[(\bar{x}_1\bar{x}_4 + a \oplus \bar{x}_3)(\bar{x}_1\bar{x}_4 + a \oplus x_3)\right]\Big|_{\bar{x}_3}$$

$$= (\bar{x}_1\bar{x}_4 + a)(\bar{x}_1\bar{x}_4 + \bar{a})(\bar{x}_1\bar{x}_4 + \bar{a})(\bar{x}_1\bar{x}_4 + a)$$

cube dropped because redundant

$$= (\bar{x}_1\bar{x}_4 + a)(\bar{x}_1\bar{x}_4 + \bar{a}) = \bar{x}_1\bar{x}_4 + a\bar{x}_1\bar{x}_4 + \bar{a}\,\bar{x}_1\bar{x}_4 = \bar{x}_1\bar{x}_4$$

$cut = cut - D = \{x_1, x_4, a\}$

---------- ITERATION 2

$cut = \{x_1, x_4, a, b\}$

$SDC_b = b \oplus f_b = b \oplus (x_1 + a)$

$CDC_{out} = CDC_{out} \cup SDC_b = \bar{x}_1\bar{x}_4 + b \oplus (x_1 + a)$

$D = \{x_1\}$

$$\mathcal{C}(CDC_{out})|_{x_1} = \left[\bar{x}_1\bar{x}_4 + b \oplus (x_1 + a)\right]\Big|_{x_1} \cdot \left[\bar{x}_1\bar{x}_4 + b \oplus (x_1 + a)\right]\Big|_{\bar{x}_1}$$

$$= [0 + b \oplus 1] \cdot [\bar{x}_4 + b \oplus a] = \bar{b}(\bar{x}_4 + b \oplus a) =$$

$$= \bar{b}(\bar{x}_4 + \bar{b}a + \bar{a}b) = \bar{b}\bar{x}_4 + a\bar{b}$$

$cut = cut - D = \{x_4, a, b\}$

---------- ITERATION 3

$cut = \{x_4, a, b, c\}$

$SDC_c = c \oplus f_c = c \oplus (a + x_4)$

$CDC_{out} = CDC_{out} \cup SDC_c = \bar{b}\bar{x}_4 + a\bar{b} + c \oplus (a + x_4)$

$D = \{x_4, a\}$

$$\mathcal{C}(CDC_{out})|_{x_4} = \bar{b}\bar{x}_4 + a\bar{b} + c \oplus (a + x_4)\Big|_{x_4} \cdot \bar{b}\bar{x}_4 + a\bar{b} + c \oplus (a + x_4)\Big|_{\bar{x}_4}$$

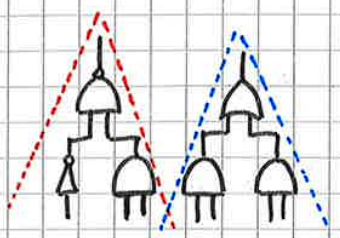
$$= [0 + a\bar{b} + c \oplus 1] \cdot [\bar{b} + a\bar{b} + c \oplus a] =$$

Internal ODC computation

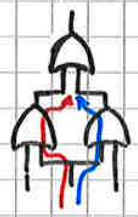
The external ODC must be propagated backward among the logic levels of the Boolean network to compute the internal ODC at the desired node. Each time the ODC goes through a logic level, don't care conditions can be added/removed. Hence two things are necessary:

1) A way to move backwards in the Boolean network: **tree-network traverse algorithm**. This time, there are two issues:

a) **multiple outputs**: if the Boolean network has multiple outputs, the problem became more complex, because observability depends on output: specific ODC conditions must be specified at each output, and one at a time they have to be propagated backwards.
This is the reason why during technology mapping (CH11) there is a phase called "partitioning" where the boolean network is divided into "logic cones" (many-in, single-out network).

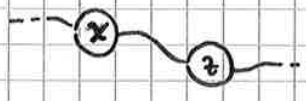


b) **Non-tree network traversal**: for networks with generic topology, a node may have multiple fanout and multiple paths to any output. This situation, called **fanout reconvergence** is hard to handle both in logic optimization and testing problems. The problem lies in combining the observability of a variable along different paths. A naive, but wrong, assumption is that the ODC set in the intersection of the ODC sets related to the different fanout stems. Indeed, we must take into account the interplay of the ability of the paths to propagate to the output a change of polarity of the corresponding variable.
More about reconvergence in testing and fault tolerance course.



reconvergent paths

2) A way to compute don't care conditions introduced by a node of the Boolean network: the observability don't care set of a node is given by:
   - the previous ODC conditions
   - the complement of the boolean difference $\frac{\partial f_z}{\partial x}$, hence $f_z|_x \oplus f_z|_{\bar{x}}$.



In fact, the boolean difference represent if an output follows an input. Hence, the complement of the boolean difference represent all input combination that do not modify the output
⇒ observability don't cares.

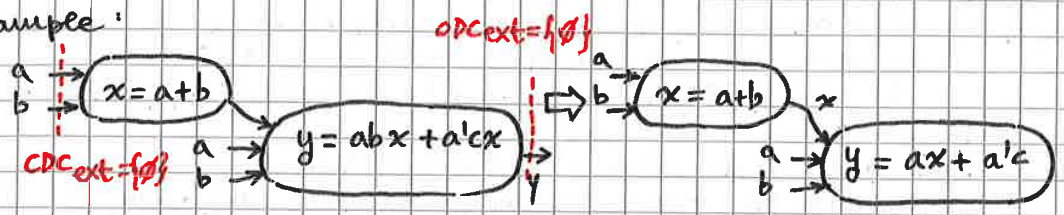
A possible algo could be:

```
OBSERVABILITY ( G(V,E), ODCout) {
        foreach node x in reverse topological order {
                ODC = ODC + ∂f current node / ∂x
        }
}
```

Boolean simplification

At the beginning of this chapter, it was said that all 5 algebraic transformations can be collapsed into one boolean transformation: the simplification.
The goal is to manipulate a local function to obtain one with less literals by using the internal don't care conditions.
The difference with respect to heuristic minimizers (like Espresso) is that the boolean simplification uses don't care conditions coming from the nodes around (topology), while Espresso can only rely on don't care conditions of the single node.

Example:



Suppose external CDC and external ODC are empty.
The CDC and ODC conditions can be propagated through the network:

$$CDC_x = x \oplus f_x = x \oplus (a+b) = ab'x + a'x' + bx'$$

$$ODC_x = \overline{\frac{\partial f_y}{\partial x}} = (abx + a'cx)\Big|_x \oplus (abx + a'cx)\Big|_{\bar{x}} = 1 \oplus 0 = 0$$

$$DC_x = CDC_x + ODC_x = ab'x + a'x' + bx'$$

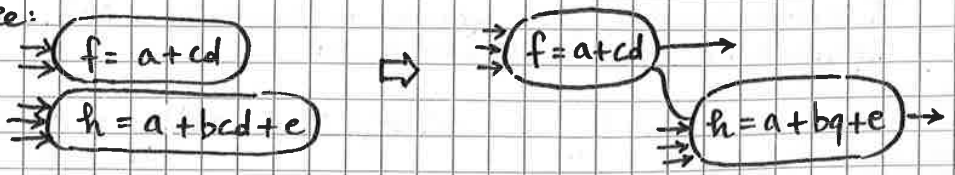
By exploiting don't care conditions, the node y can be simplified:

$$y = \underbrace{ax}_{\text{because } axb' \text{ never produced}} + \underbrace{a'c}_{\text{because } a'x' \text{ is never produced}}$$

Boolean substitution:

The substitution is empowered by the boolean simplification: given two nodes, the support set of one must be modified to include the other node; then, by means of don't care condition, the node can be simplified, otherwise, if no optimization is possible, the formant connection is removed.

Example:



$$SDC_f = f \oplus f_f = f \oplus (a+cd) = \bar{f}(a+cd) + f\overline{(a+cd)} = a\bar{f} + \bar{f}cd + f\bar{a}\bar{c}\bar{d}$$

is $cd = f$? this relation is true when

$$\begin{cases} cdf' \in DC_{set} \\ \bar{c}\bar{d}f \end{cases}$$

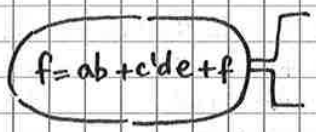
in the DC set there is $q\bar{c}\bar{d}\bar{a}$: when a is true, f is true, so it doesn't matter

Timing issues in multilevel logic optimisation — CH10

Delay is a crucial parameter of an electronic circuit: during optimisation it is possible to either select it as a constraint and search for minimum area, or viceversa set an Area constraint and search for minimum delay. Delay estimation is important also during verification, to verify the I/O delay constraints as well as cycle time constraints. Hence it is critical to define a delay estimator / timing model.

Delay modeling: it is possible to distinguish:

1) **Logic / stage (gate) delay modeling**: given a generic Boolean network, each node represents a local function. Depending on the local function complexity, the delay may change. A good estimator takes into account the # literals as well as the node fanout:

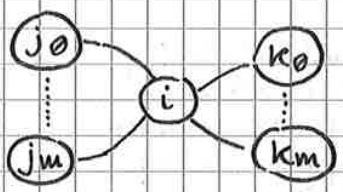


#literals = 6
fanout = 2
Delay = $\alpha 6 + \beta 2$

$$Delay = \alpha \#literals + \beta \cdot fanout$$

where α and β are weighting coefficients. A cheaper estimator can take into account only the number of literals. Each node of the network is annoted with this delay estimation. In case of mapped Boolean networks, delay can be found in technology.

2) **Network delay modeling**: for each node it is possible to define:



a) **Propagation delay**: a positive number, possibly accounting for load an function size (computed with gate delay modeling). Notation: d_i

b) **Arrival time**: it is defined at the input of the considered network, and can be propagated forward through the logic levels:

NODE DELAY

$$t_i = \max(t_j \mid e_{j \to i} \in E) + d_i$$

the arrival time is the time after which the output of the node is ready

c) **Required time**: it is specified at primary outputs, and can be propagated backwards through the logic levels:

$$\overline{t_i} = \min(\overline{t_k} - d_k \mid e_{i \to k} \in E)$$

d) **Slack**: difference between required time and arrival time:

$$s_i = \overline{t_i} - t_i$$

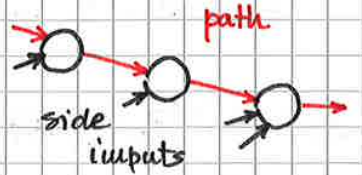
If positive, there is some margin
If negative, there is a violation

Sensitizable path

How to check if a topological critical path is a true critical path?
If a path is ~~sentiz~~ sensitizable, an event can propagate from its tail to its head. Two definitions of sensitization exist:

1) **Dynamic sensitization**: for each node of the path, it must be verified if the output follows the input transition. This condition can be easily modeled with **boolean derivative**. The boolean derivatives must be true **at the time at which the event propagates**.
The boolean derivatives will be function of side inputs

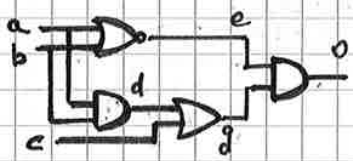
$$\frac{\partial f_{v_i}}{\partial v_{i-1}} = 1 \; @ t_{v_{i-1}} \quad \forall v_i \in T.C.P.$$



2) **Static sensitization**: like for dynamic sensitization, for each node of the path it must be verified that the output follows the input transition. This condition is modeled with boolean derivative. The boolean derivatives will be function of side inputs. This time, the timing information is neglected: this choice makes the model simpler but weaker.
In fact it may lead to **underestimated delays** due to glitches or fanout reconvergence.

$$\frac{\partial f_{v_i}}{\partial v_{i-1}} = 1 \quad \forall v_i \in T.C.P.$$

Example:



Suppose all gates have unit delay. Then the topological critical paths are:
$\{\{a, d, g, o\}, \{b, d, g, o\}\}$
Due to circuit simmetry, we can analyze only one.

STATIC

$$\frac{\partial f_{v_o}}{\partial g} = e \cdot g|_g \oplus e \cdot g|_{\bar{g}} = e \oplus 0 = e$$

$$\frac{\partial f_{v_g}}{\partial d} = d + c|_d \oplus d + c|_{\bar{d}} = 1 \oplus c = \bar{c}$$

$$\frac{\partial f_{v_d}}{\partial a} = a \cdot b|_a \oplus a \cdot b|_{\bar{a}} = b \oplus 0 = b$$

The three conditions must be valid (=1) together: therefore AND(conditions) = 1

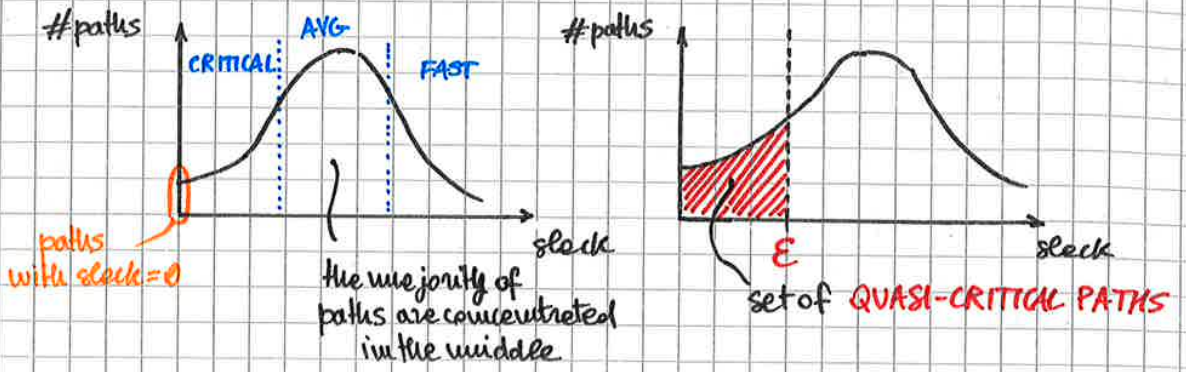
$$e \cdot \bar{c} \cdot b = 1 \rightarrow (\overline{a+b})\,\bar{c} \cdot b = \bar{a}\bar{b}\,\bar{c}\,b = 0$$

this path is not statically sensitizable.

Algorithm for delay optimization/minimization

Given as inputs the arrival time of PIs and required time of POs, the algorithm must alternate between critical path computation and logic transformations on nodes belonging to critical paths.

A possible algorithm must iterate on critical paths: but how to discriminate between "normal" and "critical" paths? The user must specify a slack treshold ε:



The algorithm must operate on multiple paths at the same time to:
1) Improve quality of result
2) Reduce number of transformations applied
3) Avoid "ping-pong" effect
4) Improve convergency of the algorithm

A possible algorithm is the following:

```
Reduce_Delay ( G(V,E), ti of PIs, t̄i of POs, ε) {
    repeat {
        propagate the arrival time forward
        propagate require time backward
        compute the slack of each node
        Identify topological critical paths
        Filter TCPs using sensitization conditions
        Identify all sensitizable paths with slack lower than ε
        Select a subset of the total node on which apply transformations
    } until (no possible savings)
}
```

static timing analysis

Algorithmic based approach

The algorithmic-based approach can be subdivided in 3 substages:

1) PARTITIONING: this stage aims at dividing the unbounded generic boolean network (multi input, multi output) into a set of logic cones (multi input, single output).

2) DECOMPOSITION: each logic cone is composed by generic (and maybe complex) local functions. This stage decomposes the logic cones (using decomposition transformation) using a subset of Boolean operators available in the library (in our course we use NAND-INV). This step is crucial because:
   a) Guarantees the existence of a solution (trivial binding)
   b) Simplifies the algorithmic solution of the covering problem

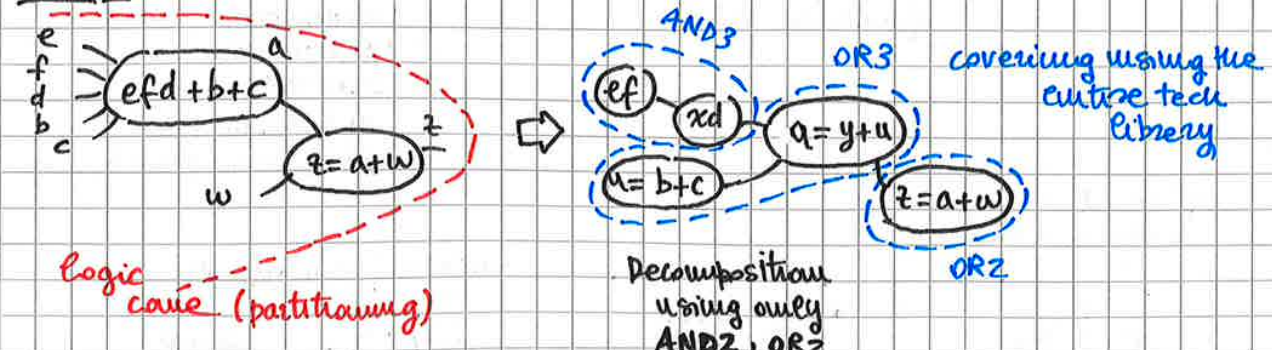
3) COVERING: this stage aims at optimising the trivial binding solution found at the decomposition stage. To achieve this goal, two main operations must be performed:

   a) identify covers: partition the logic cone into a set of smaller Boolean networks; each one of them can be replaced with a specific gate inside the tech library

   b) matching: a smaller Boolean network and a gate (cell) in tech library do match if they show the same terminal behavior.

The covering problem can be solved by means of:
   a) Exact methods based on Boolean matching
   b) Heuristic methods (dynamic programming) based on Boolean tautology or structural approach (tree based).

Example:



tech library: AND2, AND3, OR2, OR3

A possibee algorithm for covering (tree based):

for each node in SUBJECTTREE (from input to output) {

1) Check for possible matching. There are two possible options:

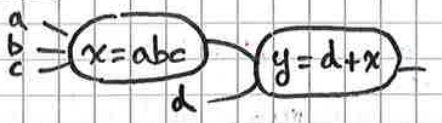
a) the locally rooted sub-tree (till the inputs) does match with a pattern tree (cell)
⇨ COST OF CELL

b) a subtree of the locally rooted subtree does match with a pattern tree (cell)
⇨ COST OF CELL + COST OF LEAVES

2) take the one with minimum cost

}

Example:



The given tech library is composed of

INV	$\overline{a}$	cost = 1
NAND2	$\overline{ab}$	cost = 2
AND2	ab	cost = 4
NOR2	$\overline{a+b}$	cost = 3
OR2	$a+b$	cost = 5
AOI21	$\overline{ab+c}$	cost = 7
AOI22	$\overline{ab+cd}$	cost = 8
NAND3	$\overline{abc}$	cost = 4

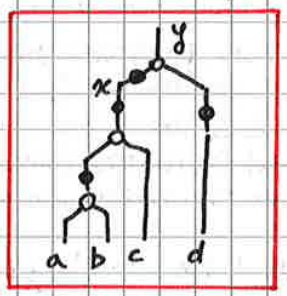
1° stage: partitioning. The boolean network is already a logic cone with only one output

2° stage Decomposition: bind the network to NAND2 + INV cells:

$$x = abc = \overline{(ab)c} = \overline{\overline{ab} \cdot c}$$

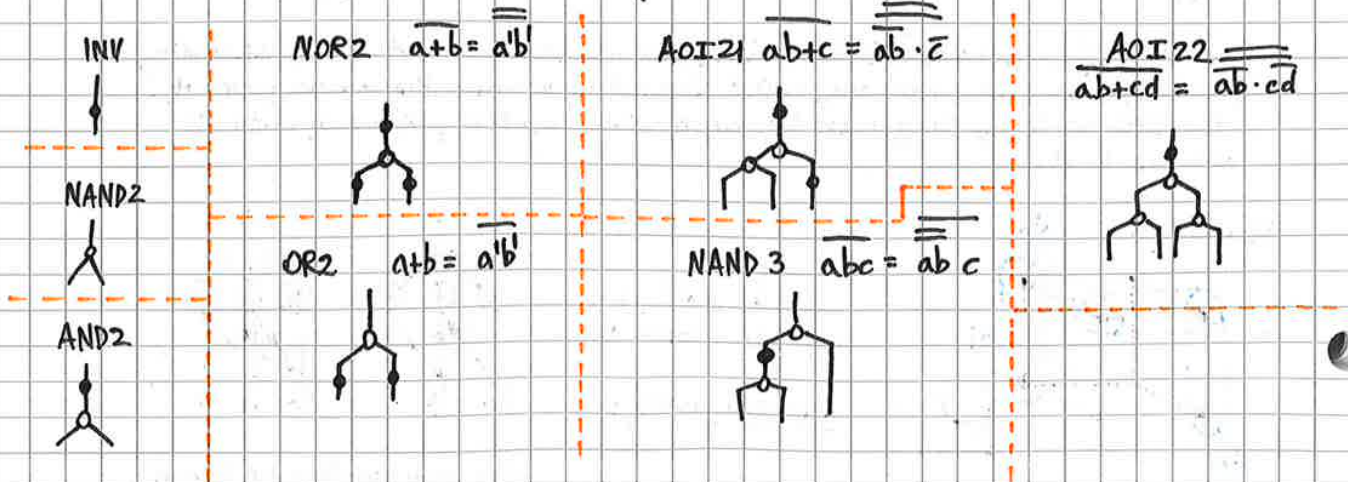
$$y = d+x = \overline{d+x} = \overline{d'x'}$$

Let's identify NAND2 as  and INV as



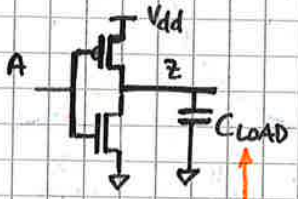
SUBJECT TREE

3° stage: Covering. Bind the network using the entire tech library. Therefore, we must build first the library as tree patterns using only NAND2 and INV. In this way, possible matching can happen.

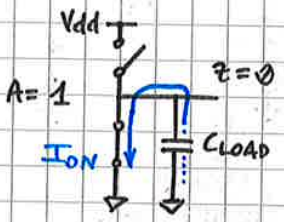


Power consumption modeling and estimation — CH 12

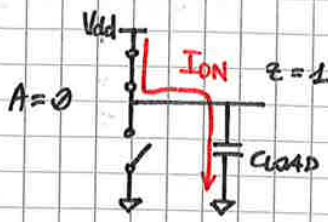
Earlier in the course an area estimator (# literals, or if you have a tech library, sum of physical area occupied by each gate of logic circuit) and a delay estimator (based on static timing analysis engine, also implemented in commercial tools) were introduced.
What is missing is a power estimator(s).
For a CMOS circuit, the power consumptions can be divided into 3 main contributions: dynamic, static and short circuit power.
Let's use an example: a CMOS inverter:



LUMPED CAPACITANCE taking into account:
- cap. of fanout gates
- cap of wire
- etc.



capacitance is discharged



capacitance is charged

$$Energy = C_{LOAD} \cdot V_{dd}^2$$

$$Energy_{STORED} = \frac{1}{2} Energy = \frac{1}{2} C_{LOAD} V_{dd}^2$$

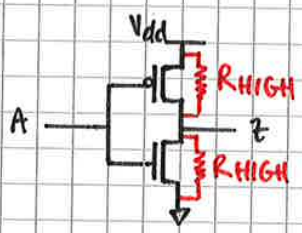
$$Energy_{WASTED} = \frac{1}{2} C_{LOAD} V_{dd}^2$$ energy consumed as HEAT

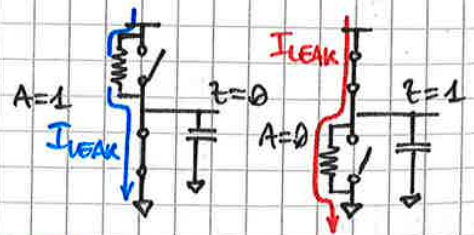
$$P = \frac{\Delta E}{\Delta T} = \frac{1}{2} C_{LOAD} V_{dd}^2 \frac{1}{T_{clk}} = \frac{1}{2} C_{LOAD} V_{dd}^2 f_{clk}$$

$$P_{dynamic} = \frac{1}{2} C_{LOAD} V_{dd}^2 f_{clk} \cdot E_{sw}$$

where E_{sw} is the switching activity: it represent the probability that a given gate output makes a transition

Due to non idealities in the CMOS transistors, they cannot be modeled as ideal switches. Even if they are open, there is still a leakage current flowing, that is modeled by means of an High value resistor R_{HIGH}:



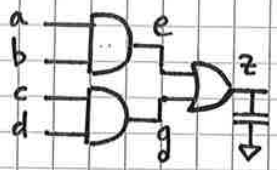


$$P_{static} = I_{leakage} \cdot V_{dd}$$

Example: state event

$$\left.\begin{aligned} P_{z:0\to1} &= P_{\bar{z}} P_z \\ P_{z:1\to0} &= P_z P_{\bar{z}} \end{aligned}\right\} E_{sw} = P_{z:0\to1} + P_{z:1\to0} = 2 P_z P_{\bar{z}} = 2 P_z (1 - P_z)$$

Example: switching activity propagation over a given topology



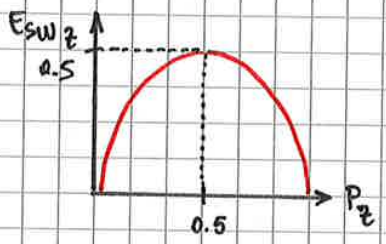
Suppose $P_a = P_b = P_c = P_d = \frac{1}{2}$

$$P_e = P_a \cdot P_b = \frac{1}{2}\cdot\frac{1}{2} = \frac{1}{4} \to P_{\bar{e}} = 1 - P_e = \frac{3}{4} \qquad E_{sw_e} = 2 P_e (1 - P_e) = 2 \cdot \frac{1}{4}\cdot\frac{3}{4} = \frac{3}{8}$$

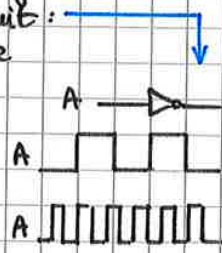
$$P_g = P_c \cdot P_d = \frac{1}{2}\cdot\frac{1}{2} = \frac{1}{4} \to P_{\bar{g}} = 1 - P_g = \frac{3}{4} \qquad E_{sw_g} = 2 P_g (1 - P_g) = 2 \cdot \frac{1}{4}\cdot\frac{3}{4} = \frac{3}{8}$$

$$P_z = 1 - P_e P_g = 1 - \frac{3}{4}\cdot\frac{3}{4} = 7/16 \to P_{\bar{z}} = 1 - P_z = \frac{9}{16} \qquad E_{sw_z} = 2 P_z (1 - P_z) = \frac{7}{16}\cdot\frac{9}{16} \cdot 2 = \frac{63}{128}$$

If the static probability is not propagated through the circuit, the switching activity for each node cannot be computed. Therefore Synopsys tests assume the worst case scenario: $P_z = \frac{1}{2} \to E_{sw} = \frac{1}{2}$

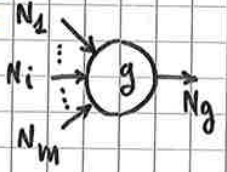


Moreover, the switching activity alone is not enough for considering the dynamic behavior of the circuit: the **toggle number** must be computed:



same E_{sw}, different power consumption

NUMBER OF TOGGLES OF INPUT

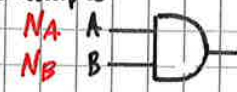


NUMBER OF TOGGLES

$$N_g = \sum_{\text{inputs}} N_i \, P\left(\frac{\partial f_g}{\partial i}\right)$$

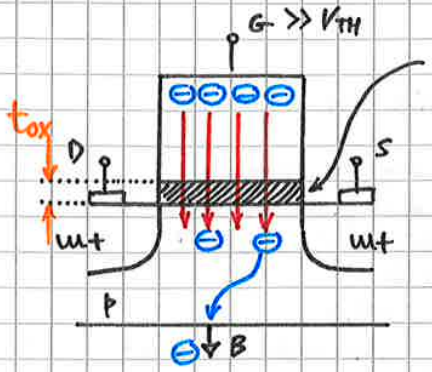
BOOLEAN DIFFERENCE

Example:



$$N_z = N_A \cdot P\left(\frac{\partial f_z}{\partial A}\right) + N_B \cdot P\left(\frac{\partial f_z}{\partial B}\right)$$

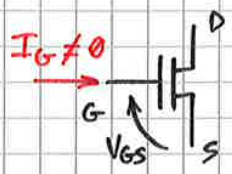
$$\left.\begin{aligned} \frac{\partial f_z}{\partial A} &= ab|_A \oplus ab|_{\bar{A}} = b \oplus 0 = b \\ \frac{\partial f_z}{\partial B} &= ab|_B \oplus ab|_{\bar{B}} = a \oplus 0 = a \end{aligned}\right\} N_z = N_A \cdot P(b) + N_B P(a)$$



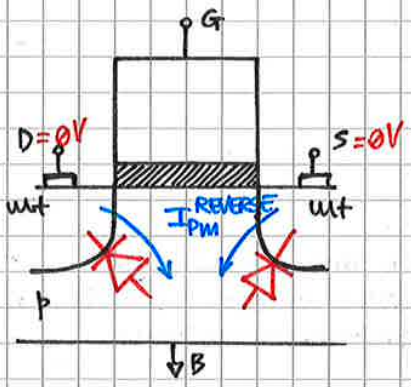
SiO_2 : silicon dioxide (insulator)

The gate metal and the p-doped semiconductor have in between a thin layer of insulator: this is a parallel plate capacitor!

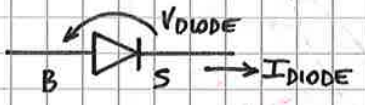
Therefore, when V_G is positive, a strong electric field is generated: due to quantum effects (tunneling), some electrons overcome the insulator voltage barrier, reaching the p-doped semiconductor, and flowing into the bulk and so into ground. To summarize, due to quantum effects, there is a gate current non negligible:

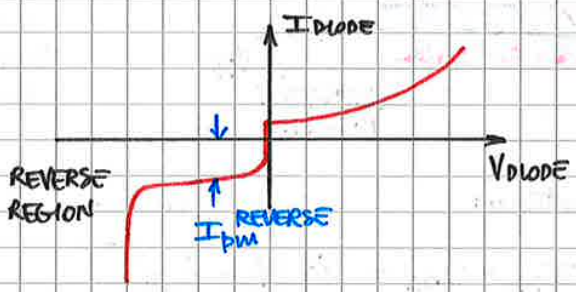


$$I_{GATE} \propto WL \left(\frac{V_{GS}}{t_{ox}}\right)^2$$



The n-doped and p doped regions form a pn junction ≡ DIODE!
If the bulk voltage becomes sufficiently large, a current will flow





To summaries

$$I_{LEAKAGE} = I_{SUBTH} + I_{GATE} + I_{pn}^{REVERSE}$$

$$P_{STATIC} = I_{LEAKAGE} \cdot V_{dd}$$

Techniques for reducing leakage power

1) Multi V_{th} — static: during optimization phase, permanent changes are introduced → Dual V_{th} / Dual V_{th} + Gate sizing
   - dynamic: during optimization some changes are introduced; during device operation, it is possible to tune leak power → Body effect (Bulk voltage)

They work on I_{SUBTH} only; I_{GATE} depends from technology (t_{ox})

2) Input based: a certain circuit state consume less leakage power than another.

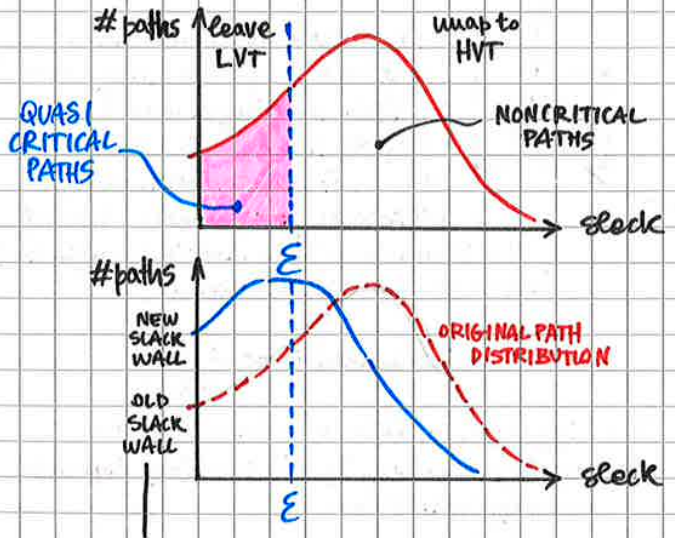
Let's talk about Multi V_{th} (Dual) static optimization: usually a standard cell library is "duplicated" for different threshold voltages.

The library used during laboratories experiences contains:

HVT	(high V_{th})	leakage ↓	Propagation delay ↑
SVT	(standard V_{th})	—	—
LVT	(low V_{th})	leakage ↑	Propagation delay ↓

Therefore, low threshold voltage cells are faster but leaky, while high threshold voltage cells are slower but less leaky wrt to the standard V_{th} cells.

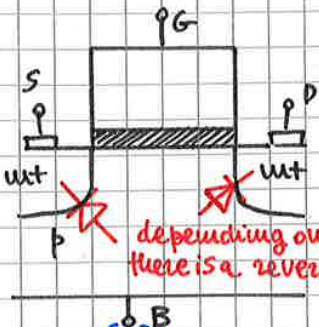
The multi vth approach allows leakage power minimization WITHOUT reducing the operating frequency:



1) Synthesize circuit using only LVT cells to obtain best performances.
2) Identify a threshold ε for slack, to distinguish between quasi critical paths and non critical paths.
3) Swap cells of non critical paths with HVT cells, to keep same overall timing performances but less power consumption (leakage)

Slack wall: the number of paths with slack = 0 increases. Due to process variations, the probability that a slower cell is in a timing critical path increases, and therefore the probability that the manufactured product works are decreased (≡ yield ↓)

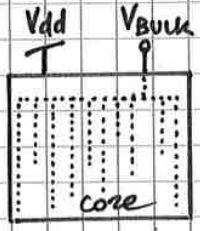
Multi V_{th} dynamic approach: Body effect



depending on V_{BULK}, V_S, V_D there is a reverse current flowing

V_{BULK} CAN BE TUNED DYNAMICALLY AT RUN TIME

$V_S = 0$, $V_{BULK} < 0$	$V_S = 0$, $V_{BULK} > 0$
Reverse Body Biasing	Forward Body Biasing
$V_{TH} \uparrow$	$V_{TH} \downarrow$
$P_{LEAK} \downarrow$ $D_P \uparrow$	$P_{LEAK} \uparrow$ $D_P \downarrow$



During IDLE, the bulk voltage is driven to a very low value
⇨ STAND BY LEAKAGE

The core is slower, but because it is not used, it does not matter!

Another applications for this technique:
Suppose that after manufacturing, process variations made your device too slow: instead of decreasing the clock frequency or binning the device, the V_{BULK} can be increased such that D_P is compliant with specs. This can effectively increase speed, but V_{BULK} can be increased by a smaller amount otherwise too much leakage power

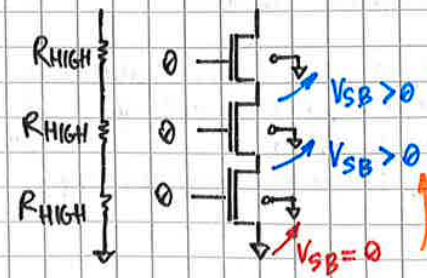
Input based approach for reducing leakage

This technique is very easy to implement, but is not very effective as the previous ones. We already know that depending on the input state, a gate can consume a different amount of leakage power (see lib files).
For example, it is possible to rank input combinations (based on P_{LEAK}) of a CMOS NAND gate:

A	B	Z	leak
0	0	1	1 (MINIMUM LEAKAGE)
0	1	1	3
1	0	1	2
1	1	0	4 (MAX LEAKAGE)

This phenomenon is related to body biasing. In fact a chain of NMOS is more leakage efficient than a parallel of PMOS (in the specific case of the NAND GATE)

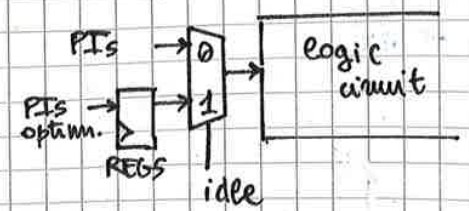
eq. model



Reverse Biased $V_{TH} \uparrow\uparrow$ $P_{LEAK} \downarrow$
Reverse Biased $V_{TH} \uparrow$ $P_{LEAK} \downarrow$

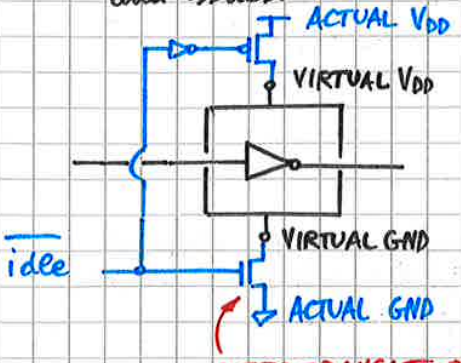
$V_{DS} = R_{HIGH} I_{LEAK} \neq 0$

Therefore, there is an optimum input pattern such that leakage is minimized:



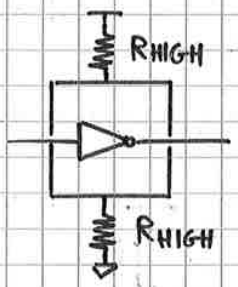
Power gating

This approach is in the middle between static and dynamic. It allows 100% leakage savings, but has some drawbacks and issues.



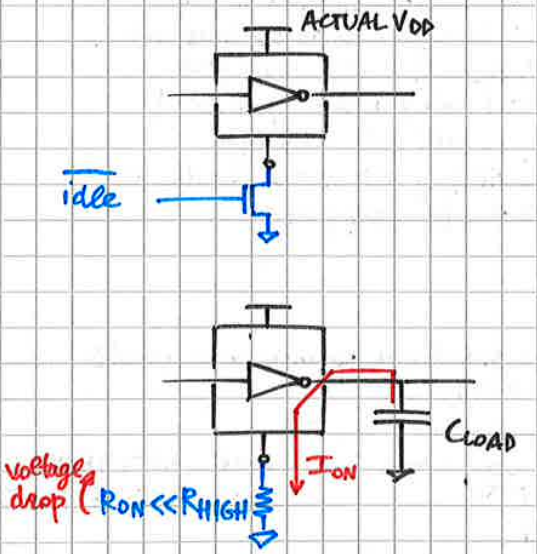
SLEEP TRANSISTOR: HVT cell with almost no leakage (very high V_{TH} ...)

in reality



power gating can be applied only to functional cores. It is possible to tune the sleeping strength.

The same result can be obtained with one transistor (NMOS) only! In this way 50% area is saved, but some issues may arise.... I'll discuss it later on

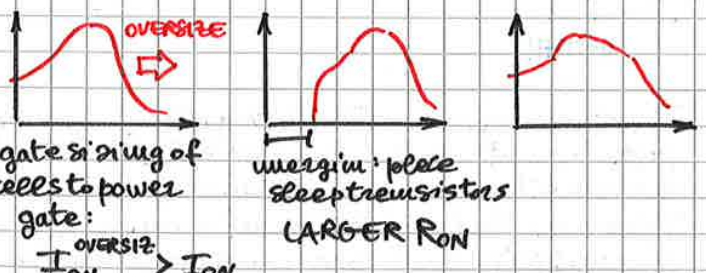


One of the most noticiable drawbacks is that the cell becomes slower! the sleep transistor may become a bottleneck for current I_{DS}: it must be sized properly:

$$\Delta D_p^{max} \leq 3\% D_p^{NOMINAL}$$

$$R_{ON}^{MAX} = \frac{V_x^{MAX}}{I_{ON}}$$

To overcome the propagation delay issue, you must oversize the cells and use the slack margin for the sleep transistors



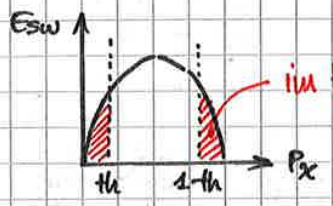
How to find out the maximum I_{DS} current to correctly size the sleep transistor? Very complex to calculate: try all possible permutations of all possible inputs patterns, run a full simulation and annotate biggest current. Too much consuming, use Heuristics!

Accounting for signal probabilities

Given a gate and probability of its inputs, the switching activity can be defined as:

$$Esw = 2 P_x (1 - P_x)$$

where P_x is the probability that output x of considered gate is true.
There is an interesting correlation between P_x and Esw:



in the highlighted regions the switching activity is low. Therefore during optimization, a solution that provides a smaller Esw can be preferred to another one, by modifying the cost function/estimators:

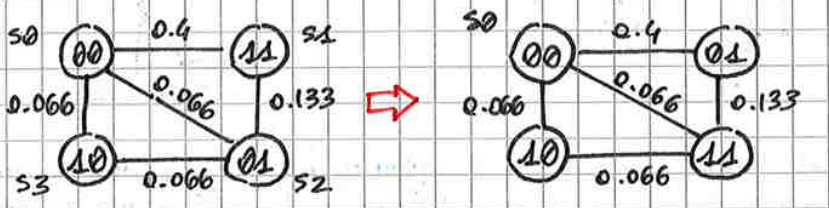
$$area = \#literals \cdot P_x(1-P_x)$$

Or an heuristic can be exploited:

1) set probability threshold th
2) for each node compute don't care set and output signal probability
   a) if probability is a desired one ... (below th or above 1-th) minimize the function using don't cares and traditional cost function
   b) if probability is not a desired value minimize the function using don't care conditions and modified cost function that takes into account for signal probability

State Assignment

In an FSM, to minimize power consumed by state registers, the binary code of each state must be assigned taking into account transition probabilities: the larger the prob, the smaller the hamming distance between the two state codes:



cost function = 1,2

cost function = 0.8
optimized