KUNGL TEKNISKA HÖGSKOLAN

Inte

Master Pr

in System-on-Chip Design

L3: Representations of functions

Representations of Boolean functions

- Boolean expression
  - Two-level sum-of-product form, factorized form
- Truth tables
- Karnaugh maps
- Cubes
  - (MIN,MAX) notation
  - positional cube notation
- Binary Decision Diagrams
- Logic circuits
- Galois field GF(2) polynomials

Two approaches

- Two fundamental approaches:
  - keep representation canonical with respect to the function
    - tautology or SAT check is easy
    - but representation may blow-up in space
    - Examples: Truth tables, Karnaugh maps, BDDs
  - keep representation non-canonical
    - representation can remain compact
    - tautology or SAT check is exponential (co-NP complete)
    - Example: Boolean expressions, logic circuits

Boolean formula

- Any Boolean function can be represented by a formula defined as catenations of
  - parentheses (,)
  - variables x, y, z
  - binary operations "+" (OR) and "·" (AND)
  - unary operation negation, "′"
- Examples:

  $f(x_1,x_2,x_3) = x'_1 \cdot x_2 + x_1 \cdot x_3$

  $f(x_1,x_2,x_3) = (x_1 + x_2)' \cdot x_3$

Canonical SOP form

- Every Boolean function $f: \{0,1\}^n \rightarrow \{0,1\}$ has a canonical sum-of-products (SOP) form of type:

$$f(x_1,x_2,\ldots, x_n) = \sum_{i=0}^{2^n-1} c_i \cdot x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n}$$

where

- $c_i \in \{0,1\}$ is a constant
- $(i_1,i_2,\ldots, i_n)$ is the binary expansion of i
- $x_k^{i_k} = x'$ if $i_k = 0$ and $x_k^{i_k} = x$ if $i_k = 1$

Deriving the canonical form

- The above form can be obtained from Shannon decomposition theorem:

$$f(x_1,x_2, \ldots,x_n) = x'_1 \cdot f|_{x_1=0} + x_1 \cdot f|_{x_1=1}$$

where

$$f|_{x_1=0} := f(0,x_2, \ldots,x_n),\ f|_{x_1=1} := f(1,x_2, \ldots,x_n)$$

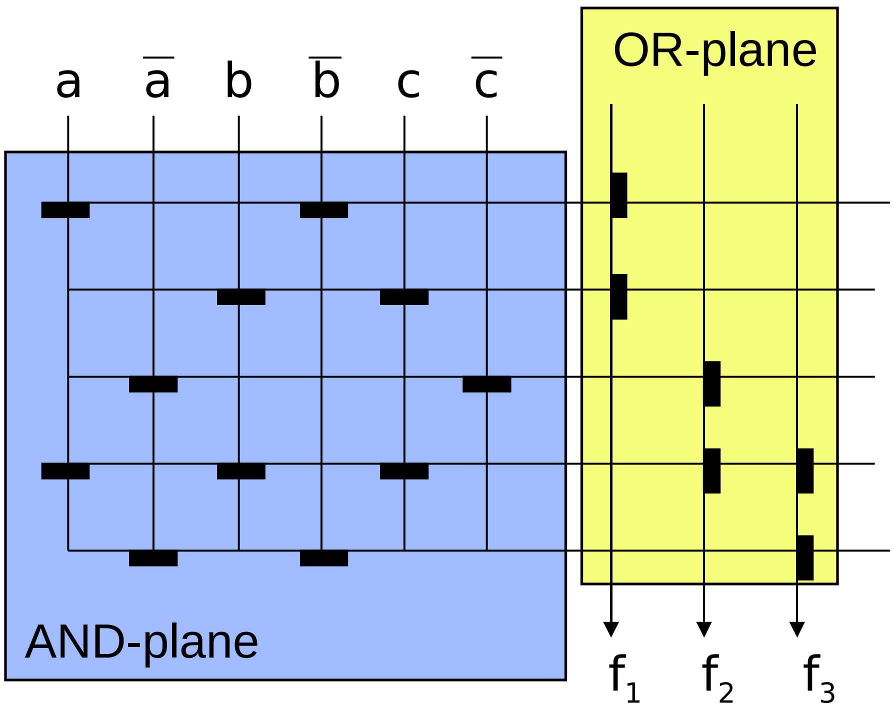
are subfunctions (cofactors) of f

Minimization of SOP

- The number of products in the SOP canonical form is up to 2^n
- It can be simplified using the axioms and properties of Boolean algebra
  - e.g. applying $a + a' = \mathbf{1}$ reduces the number of products by one
  - more general rule is $a \cdot P + a' \cdot P = P$, where P is a product-term

Why minimizing SOP form?

- A SOP expression can be directly implemented by a Programmable Logic Array (PLA):



Cube table

abc	$f_1f_2f_3$
10-	1 ~ ~
-11	1 ~ ~
0-0	~ 1 ~
111	~ 1 1
00-	~ ~ 1

Summary of SOP forms

- Advantages:
  - easy to manipulate and minimize
  - many algorithms available (e.g.AND,OR,TAUTOLOGY)
  - directly map into PLAs
- Disadvantages:
  - poor representative for logic complexity for multi-level implementation. For example

$$f=ad+ae+bd+be+cd+ce \quad f'=a'b'c'+d'e'$$

    these differ in their implementation by an inverter.
  - difficult to estimate progress during optimization

Factored forms

- Factored forms are more compact representations of logic functions than SOP forms
- Example: if the factored form is

  (a+b)(c+d(e+f(g+h+i+j)

  when represented as a SOP form it is

  ac+ade+adfg+adfh+adfi+adfj+bc+bde+bdfg+ bdfh+bdfi+bdfj
- When measured in terms of number of inputs, there are functions whose size is exponential in sum of products representation, but polynomial in factored form.
- Example: Achilles' heel function has n literals in the factored form and $(n/2)\times 2^{n/2}$ literals in the SOP form

$$\prod_{i=1}^{i=n/2} (x_{2i-1} + x_{2i})$$

Factored forms

Advantages

- good representative for logic complexity for multi-level implementation.

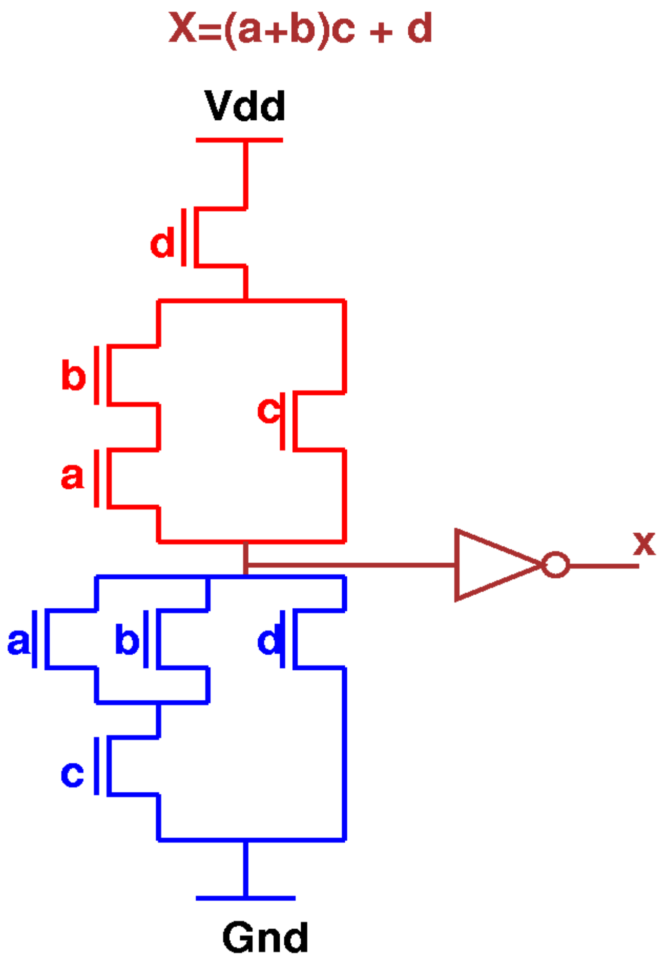
$f=ad+ae+bd+be+cd+ce \qquad f'=a'b'c'+d'e' \Rightarrow f=(a+b+c)(d+e)$

- in many designs (e.g. complex gate CMOS) the implementation of a function corresponds directly to its factored form
- do not blow up easily

Disadvantages

- not as many algorithms available for manipulation
- hence often just convert into SOP before manipulation

Factored forms



Note:

literal count ≈ transistor count ≈ area

- however, area also depends on
  – wiring
  – gate size etc.
- therefore very crude measure

Factored forms

Definition: a factored form can be defined recursively by the following rules. A factored form is either a product or sum where:

- a product is either a single literal (variable or its complement) or a product of factored forms
- a sum is either a single literal or a sum of factored forms

• Any logic function can be represented by a factored form, and any factored form is a representation of some logic function.

Examples

Examples of factored forms:

x

y'

abc'

$a+b'c$

$((a'+b)cd+e)(a+b')+e'$

$(a+b)'c$ is not a factored form since complementation is not allowed, except on literals.

Three equivalent factored forms (factored forms are not unique): $ab+c(a+b)$ $bc+a(b+c)$ $ac+b(a+c)$

Size of factored forms

Definition:

The size of a factored form F (denoted $\rho(F)$) is the number of literals in the factored form.

Example: $\rho((a+b)ca') = 4 \qquad \rho((a+b+cd)(a'+b')) = 6$

Definition:

A factored form is optimal if no other factored form (for that function) has less literals.

Truth tables and maps

- The simplest way to represent an n-variable Boolean function is by giving a truth table containing 2^n rows, each specifying the value of the function for the corresponding values of the variables $x_1, \ldots, x_n$
- Table can be re-aranged in a rectangular n-dimensional Karnaugh map

Example

$x_1x_2x_3$	f_1f_2
000	00
001	1-
010	10
011	11
100	00
101	-1
110	11
111	11

Truth table for a function

$f: \{0,1\}^3 \rightarrow \{0,1,-\}^2$

f_1:

x_3 \ x_1x_2	00	01	11	10
0	0	1	0	0
1	1	1	1	-

f_2:

x_3 \ x_1x_2	00	01	11	10
0	0	0	1	0
1	-	1	1	1

Karnaugh maps

Can we do better?

- Both, truth table and map, give a complete list of 2^n points in B^n and therefore can be used only for very small functions
- We can reduce the number of rows:
  - if we use don't care symbol for inputs:

    000 + 001 = 00-
  - if we don't show input combinations $(a_1, \ldots, a_n) \in B^n$, for which $f(a_1,\ldots,a_n) = 0$

Example

$x_1x_2x_3$	f
~~000~~	~~0~~
001	1
010	1
011	1
~~100~~	~~0~~
101	1
110	0
111	1

f_1

x_3 \ x_1x_2	00	01	11	10
0	0	1	0	0
1	1	1	1	1

$x_1x_2x_3$	f
01-	1
--1	1

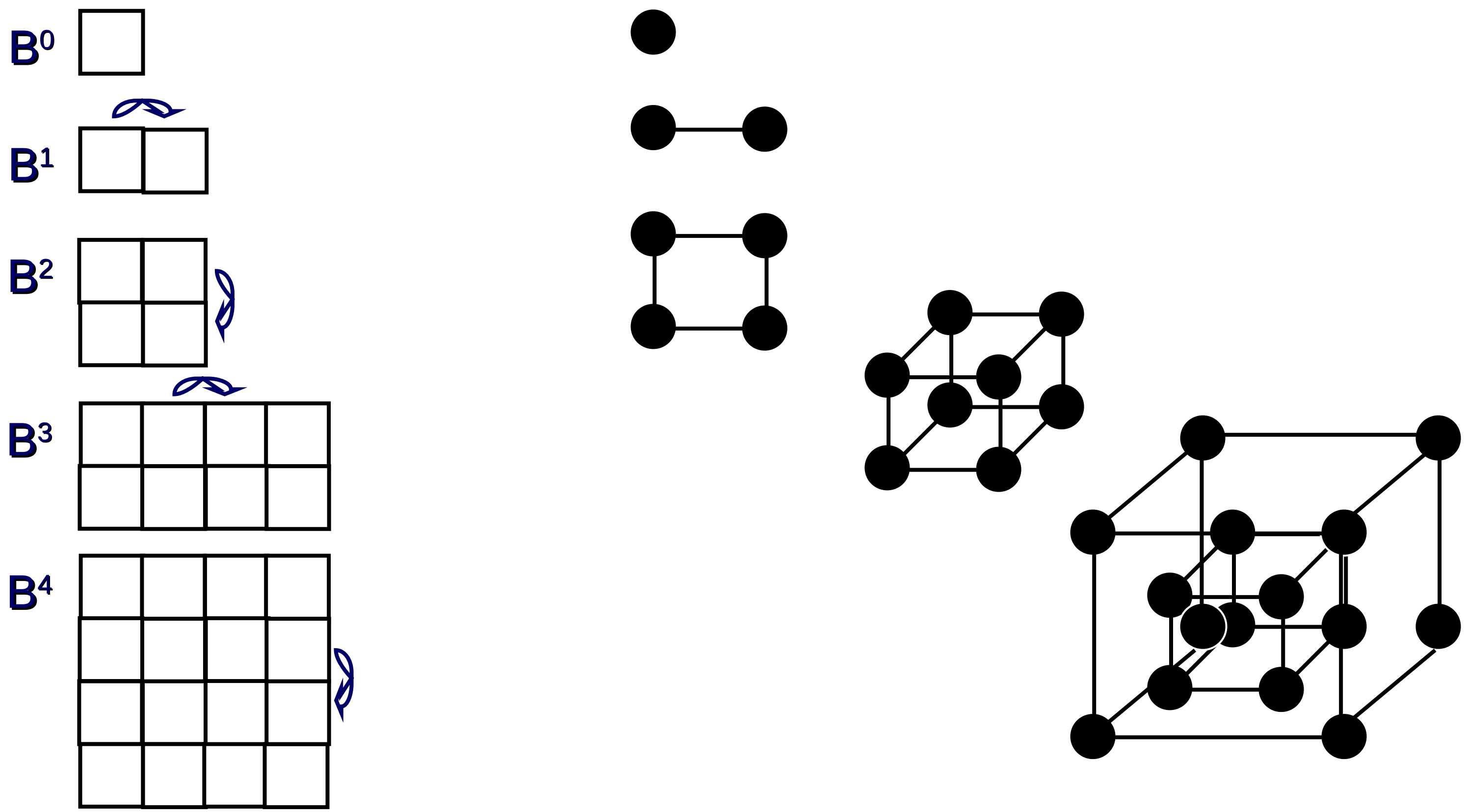
Cube terminology for f: $B^n \to B \cup \{-\}$

- An n-variable Boolean function is interpreted as a set of points in an n-dimentional Boolean space
- Cube is any k-dimentional subspace, $0 \le k \le n$
- operations on Boolean functions are performed as operations on sets

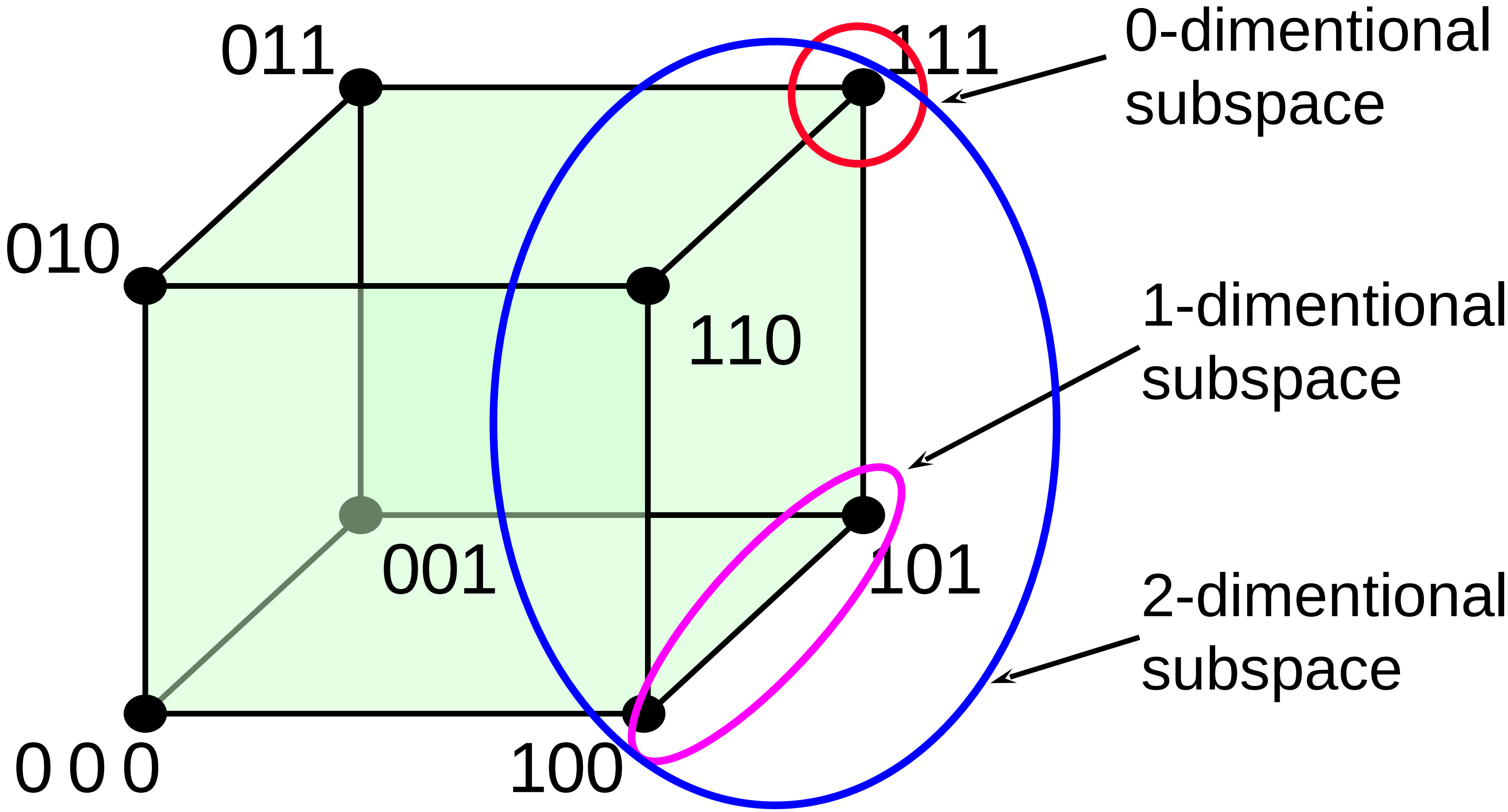
  AND $\equiv$ intersection "$\cap$", OR $\equiv$ union "$\cup$"

Boolean Space B^n

Karnaugh Maps: Boolean space:



3-dimentional Boolean space



Minterm

- A point in an n-dimentional Boolean space is called a minterm ($\equiv$ assignment $(a_1, \dots, a_n) \in B^n$ of values for the variables $x_1, \dots, x_n$ of f)
- Note: it is not necessary that $f(a_1,\dots,a_n) = 1$ for a minterm $(a_1, \dots, a_n)$, any of 2^n assignments $(a_1,\dots,a_n) \in B^n$ is a minterm
- Recall that cube is any k-dimentional subspace, $0 \leq k \leq n$

Minterms and cubes

- Informally speaking, minterm is an n-tuple containing “0” and “1” only, while cube is an n-tuple which can contain “-” as well
  - Examples of minterms for n=3:

    000, 010
  - Examples of cubes for n=3:

    000, -11, ---

Relation between SOPs and cubes

- There is one-to-one correspondence between the cube notation and sum-of-products notation:
  - cube = product-term
    - 0-1 is a cube; $x_1' x_3$ is a product-term
  - minterm = product-term with all variables present
    - 011 is a minterm; $x_1' x_2 x_3$ is a product-term
  - set of cubes = sum-of-product expression
    - {0-1, -1-} is a set of cubes representing the on-set of f; $x_1' x_3 + x_2$ is the corresponding sum-of-product expression for f

On-, off- and don't care-sets

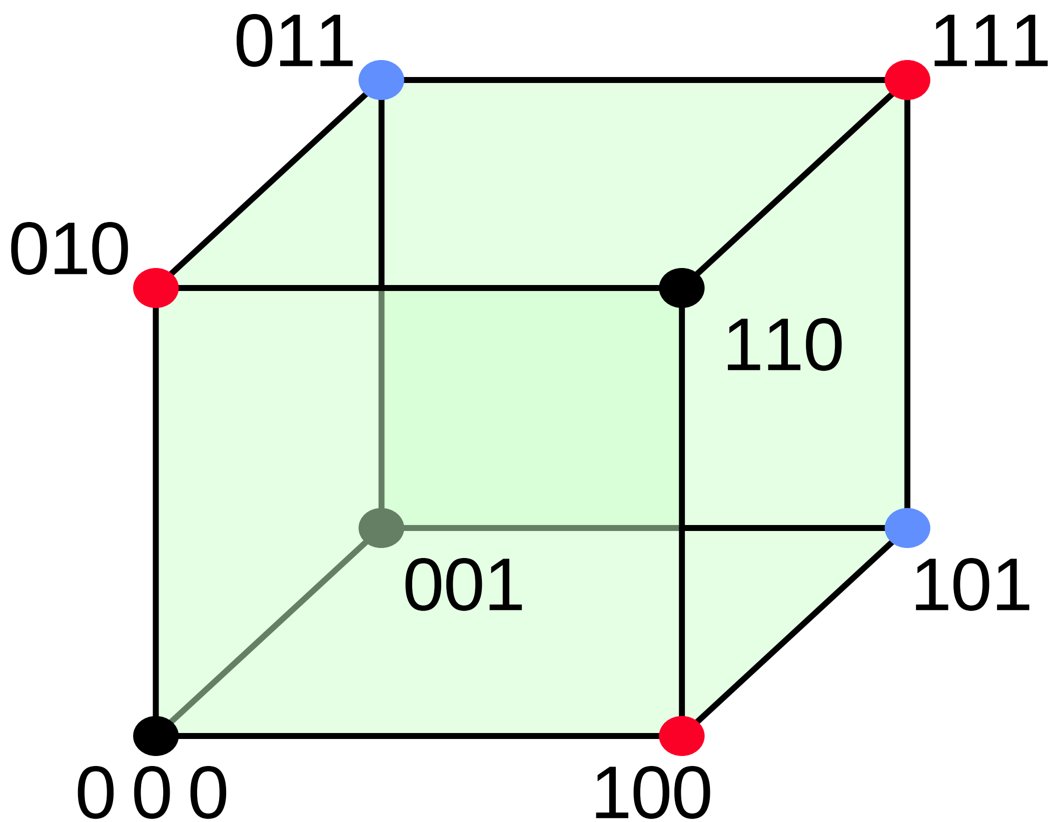
- On-set F_f is subset of B^n containing all minterms mapped to "1"
- Off-set R_f is subset of B^n containing all minterms mapped to "0"
- Don't care-set D_f is subset of B^n containing all minterms mapped to "-"

$$F_f \cup R_f \cup D_f = B^n$$

Some definitions

- If $f = B^n$, then f is tautology
- If $f = \varnothing$, then f is not satisfyable
  - satisfying truth assignment is in assignment $(a_1, \ldots, a_n) \in B^n$ for which $f(a_1, \ldots, a_n) = 1$
- f and g are equivalent if $f(a_1, \ldots, a_n) = g(a_1, \ldots, a_n)$ for all $(a_1, \ldots, a_n) \in B^n$
  - are two sub-circuits functionally identical?
  - is a particular change in the circuit valid?

Example of a 3-varable function



$F_f = \{010,100,111\}$

$D_f = \{011,101\}$

$R_f = \{000,001,110\}$

Cube representation

- We can represent any function of type $f: B^n \rightarrow B \cup \{-\}$ by listing two out of three sets F_f, R_f, D_f
  - since $F_f \cup R_f \cup D_f = B^n$, the third one can be always computed
  - for example, if F_f and D_f are given (standard case), we can compute R_f as

$$R_f = B^n - (F_f \cup D_f)$$

Espresso (or .pla) format

- .i specifies the number of inputs
- .o specifies the number of outputs
- cubes for F_f and D_f are listed as, for example for n=4

  0011 1 *a space separates the input part from the output part*

  - 010 1

  0110 ~ *means "unspecified" or "specified elsewhere"*
- .e ends the description

Example of espresso format

$x_1x_2x_3$	f
01-	1
--1	1

Cube table for F_f

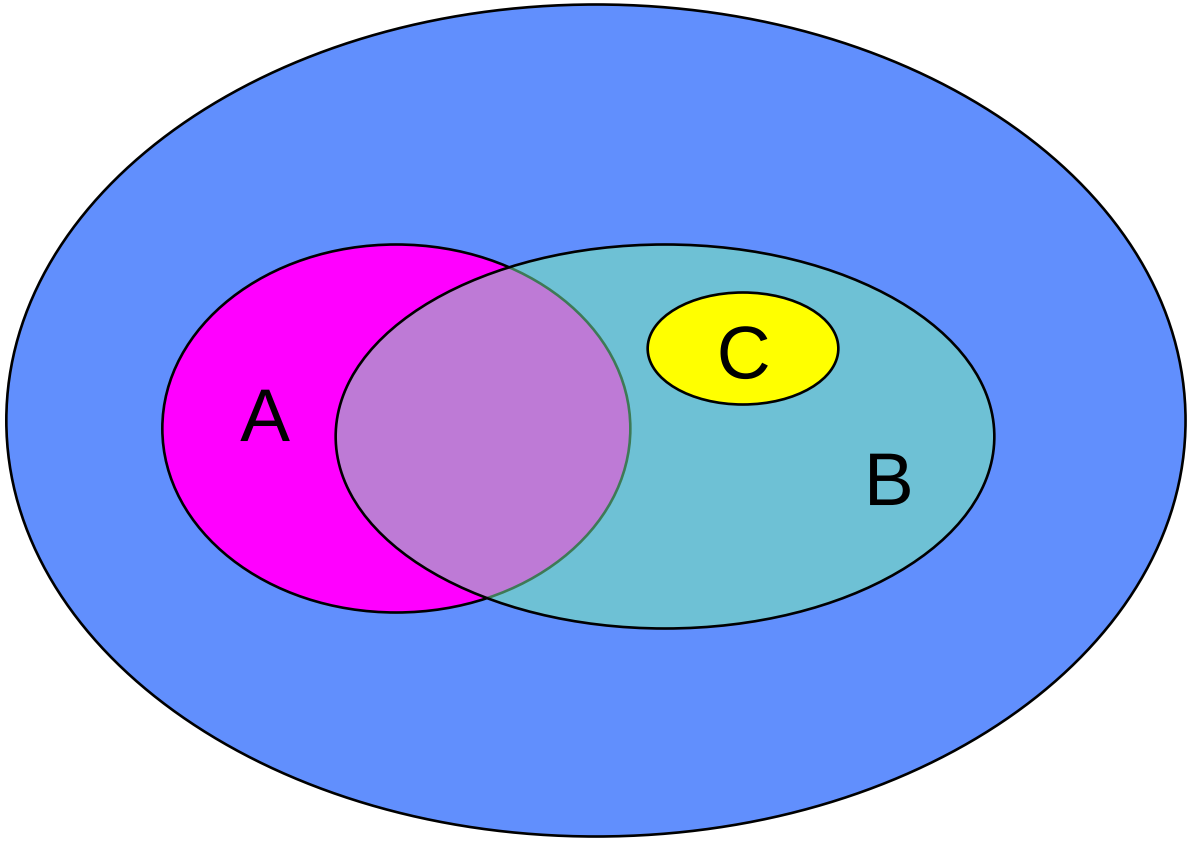
```
.i 3
.o 1
01-  1
--1  1
.e
```

espresso format

Basic operations on cubes

- Next, we define some basic operations of cubes
  - intersection of two cubes
  - complement of a cube
  - containment
  - supercube of two cubes
- Let n-tuples $A = (a_1\, a_2 \ldots a_n)$, $B = (b_1\, b_2 \ldots b_n)$, $C = (c_1\, c_2 \ldots c_n)$, $a_i, b_i, c_i \in \{0,1,-\}$ be some n-dimensional cubes

Set operations



Intersection of two cubes

- Intersection of cubes A and B is a cube C such that $c_i = a_i \cap b_i$. If any of $a_i \cap b_i = \emptyset$, then $C = \emptyset$ (empty intersection)

$$(a_1 \ldots a_n) \cap (b_1 \ldots b_n) = (a_1 \cap b_1 \ \ldots a_n \cap b_n)$$

- Element-wise intersection is defined by:

$\cap$	**0**	**-**	**1**
0	0	0	$\emptyset$
-	0	-	1
1	$\emptyset$	1	1

Example of intersection

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	1
1	0	1	1	0

$x_1x_2x_3$	f
-11	1
11-	1
1-0	1

$-11 \cap 1\text{-}0 = \emptyset$

$-11 \cap 11\text{-} = 111$

$1\text{-}0 \cap 11\text{-} = 110$

Complement of a cube

- Complement of a cube $A = (a_1\, a_2 \ldots a_n)$, is a set of cubes $B^n - A$
- An easy way to compute a complement:

$$B^n - A = \{C_1, C_2, \ldots, C_n\}$$

- where cube C_i has completented a_i in the i_{th} position and has "-" elsewhere

$C_1 = (\bar{a}_1 - \ldots -)$, $C_2 = (- \bar{a}_2 - \ldots -)$, $C_n = (-- \ldots - \bar{a}_n)$

Example of complement

- Complement of a cube (11-) = $\{C_1, C_2\}$:
  - cube C_i has completented a_i in the i_{th} position and has "-" elsewhere

$C_1 = (\bar{a}_1 - -)$, $C_2 = (- \bar{a}_2 -)$

  - don't cares are skipped (3rd position is don't care)

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	0
1	0	0	1	0

(11-)' = {(0--), (-0-)}

Containment

- Cube A is contained in cube B if and only if $a_i \subseteq b_i$ for all $i \in \{0,1,\ldots,n\}$
- The containment relation is defined by:

$$R_{\subseteq} := \{(0,0), (0,-), (1,1), (1,-)\}$$

Example of containment

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	1
1	0	1	1	0

$x_1x_2x_3$	f
-11	1
110	1
1-0	1

110 is contained in 1-0

Supercube of two cubes

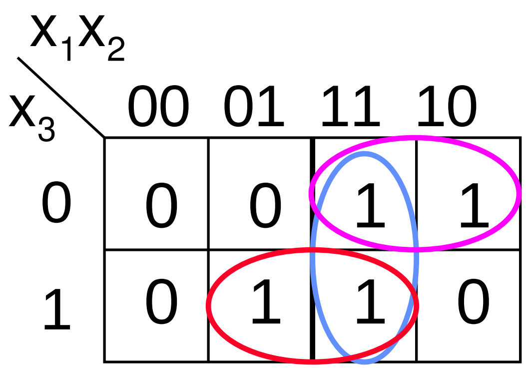
- A supercube of two cubes A and B is the smallest cube containing both A and B
- sup(A,B) can be computed as:

$$\text{sup}(A,B) = (a_1 \cup b_1 \ \ldots \ a_n \cup b_n)$$

- Element-wise union $\cup$ is defined by:

$\cup$	**0**	**-**	**1**
0	0	-	-
-	-	-	-
1	-	-	1

Example of supercube



$x_1x_2x_3$	f
-11	1
11-	1
1-0	1

sup(-11,11-) = -1-

sup(1-0,11-) = 1--

sup(-11,1-0) = ---

Problem caused by cubes

- Use of cubes reduces the number of rows in a truth table
- However, since we introduce the 3rd symbol "-", we will need more bits (2 instead of 1) to code each element of the row in computer's memory
- There are 2 conventions for coding: parallel and sequential

Parallel coding

- Use a pair of integers (min, max) to represent a cube
  - min is the integer = binary encoding of the cube, when all “-” are replaced by “0”
    - for example, min for the cube 1-0 is $(100)_2 = 4$
  - max is the integer = binary encoding of the cube, when all “-” are replaced by “1”
    - for example, max for the cube 1-0 is $(110)_2 = 6$

Parallel coding

- Function is stored dynamically as 3 lists of cubes (for the sets F_f, R_f, D_f)
- Lists are represented by a stucture declared as:

```
typedef struct ListofCubes {
    long int min;               /* min value of the cube */
    long int max;               /* max value of the cube */
    struct ListofCubes *next; /* pointer to next cube */
} one_cube;                     /* name of the new type */
```

Parallel coding

- Parallel coding allows a fast bit-wise implementation of many basic operations
  - e.g. max for supercube = bit-wise OR "|" of max parts of cubes
- Functions of up to 32 inputs can be represented in this way (32 = long int)
  - can handle larger functions by storing each of min and max in 2 or more words

Sequential coding = positional cube notation

- Cube is represented by substituting each of the symbols {0,1,-} by a 2-bit field:

$\emptyset$	00
0	10
1	01
-	11

- empty set stands for a non-allowed symbol

Sequential coding

- Sequential coding allows a fast bit-wise implementation of many basic operations
  - e.g. intersection of two cubes = bit-wise AND “&” of two cubes
- No restriction on the number on inputs/outputs

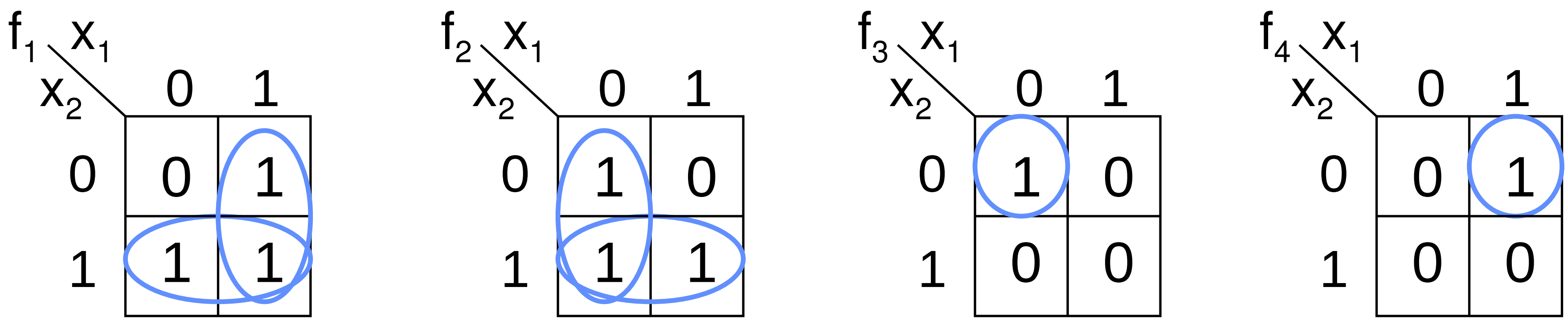
A possible project topic

- Develop an algorithm which reads in a list of cubes (in espresso format) representing an AND-OR expression of a Boolean function f and compute a minimal AND-XOR expression for f. Give the in the form of the list of cubes whose XOR gives us f
- Example:
  - Read in cubes {01,10} corresponding to a'b + b'a.
  - Read out cubes {-1,1-} corresponding to $a \oplus b$

Multiple-output functions

- All the theory we considered so far applies to single-output functions only
- Most of the real-life functions are multiple-output
- A multiple-output function can be treated by performing the operations on each output separately
  - but then the optimality is often lost, e.g. cubes common for several functions will not be found

Example



4-output function:

6 cubes if the functions are treated separately

Extension of cube representation to multiple-output functions

- We will extend cubes to multiple-output Boolean functions $f: B^n \rightarrow (B \cup \{\sim\})^k$ by introducing "output" part of the cube
  - what we called "cube" before, now we call "input part of cube" (a sub-space of B^n)
  - output part of a cube is "0", "1" or "~"
    - "1" in the i_{th} position of the output part means "the cube belongs to the function f_i "
    - "0" in the i_{th} position of the output part means "the cube doesn't belong to the function f_i "
    - "~" in the i_{th} position of the output part means "the cube is specified elsewhere for f_i" or "non-specified for f_i"

Example I

$x_1x_2x_3$	f_1f_2
000	00
001	10
010	00
011	11
100	00
101	01
110	11
111	11

Truth table

f_1

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	0
1	1	1	1	0

f_2

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Karnaugh maps

$x_1x_2x_3$	f_1f_2
001	10
-11	11
101	01
11-	11

Cube table for F_f

Example II

- We can always specify a multiple-output function by listing all single-output functions with ”1” in the corresponding output part and wiriting ”~” for all other output parts

$x_1x_2x_3$	f_1f_2
001	1~
-11	1~
11-	1~
101	~1
11-	~1
-11	~1

cube table for the function from the previous slide

Example of intersection and supercube

f_1:

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	0
1	0	1	1	0

f_2:

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	1
1	0	0	1	0

$x_1x_2x_3$	f_1f_2
-11	10
1-0	01
11-	11

$\cap$ for outputs is bit-wise AND:

-11 10 $\cap$ 1-0 01 = $\emptyset$

-11 10 $\cap$ 11- 11 = 111 10

sup for outputs is bit-wise OR:

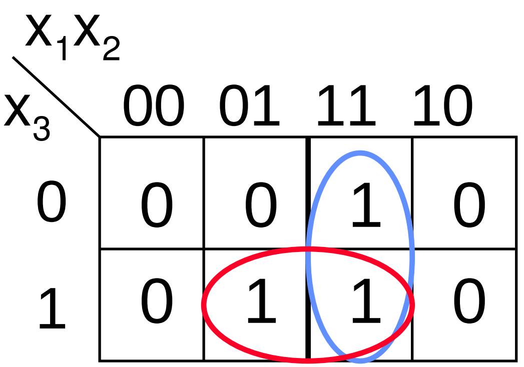
-11 10 $\cup$ 1-0 01 = - - - 11

-11 10 $\cup$ 11- 11 = -1- 11

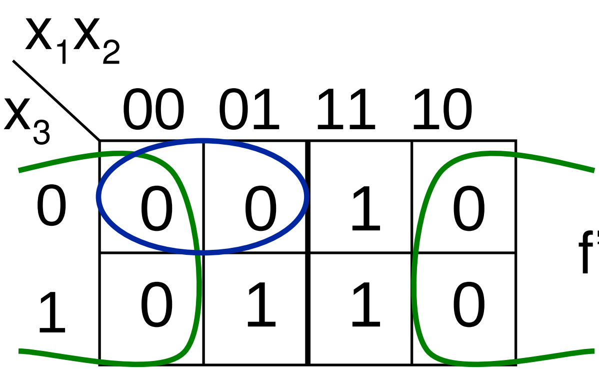
Complement

- Complement of a multiple-output function can be computed by taking complements for each output separately
- Computing the complement of a set of cubes:
  - compute complement for each cube
    - for each cube, you get a set of cubes (up to the length of input part)
  - find the intersection of these sets
  - remove cubes which are contained in other cubes

Example of complement of a set of cubes



f



f'

(11-)' = {(0--), (-0-)}

(-11)' = {(-0-), (--0)}

0-- ∩ -0- = 00-

0-- ∩ --0 = 0-0

-0- ∩ -0- = -0-

-0- ∩ --0 = -00

-0- contains 00- and -00, so

the result is {(0-0), (-0-)}

Example of complement for multiple-output functions

- Compute complements for each output:

$x_1x_2x_3$	f_1f_2
-11	10
1-0	01
11-	11

A = (11- 11)' = {(0-- 11), (-0- 11)}

B = (-11 10)' = {(-0- 10), (--0 10)}

C = (1-0 01)' = {(0-- 01), (--1 01)}

for f_1 we get A $\cap$ B = {(0-0 10), (-0- 10)}

for f_2 we get A $\cap$ C = {(0-0 01), (0-1 01), (-01 01)}

Example (cont.)

- Resulting complement for the functions is representing by the following cube table
  - recall, that "1" in the i_{th} position of the output part means "the cube belongs to the function f_i"
  - note that the solution is non-optimal

$x_1x_2x_3$	f_1f_2
0-0	10
-0-	10
0-0	01
0-1	01
-01	01

f_1:

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	0
1	0	1	1	0

f_2:

$x_3 \backslash x_1x_2$	00	01	11	10
0	0	0	1	1
1	0	0	1	0

Parallel coding for multiple-output functions (incompletely specified f)

- Use two pairs of integers (min, max): one for the input part and one for the output part of the cube
- ListofCubes stucture is modified as:

```
typedef struct ListofCubes {
    long int min;       /* min value of the input part  */
    long int max;       /* min value of the input part  */
    long int omin;      /* min value of the output part */
    long int omax;      /* max value of the output part */
    ...
```

Simplified coding for multiple-output functions (completely specified f)

- Use a single integer to represent the output part of the cube
- ListofCubes stucture is modified as:

```
typedef struct ListofCubes {
    long int min;       /* min value of the input part  */
    long int max;       /* min value of the input part  */
    long int output;  /* value of the output part */
    ...
```

An alternative way

- An alternative way to treat a multiple-output Boolean function $f: B^n \rightarrow B^k$ is to consider it as an (n+1)-variable 1-output function of type

$$f: B^n \times \{0,1,\ldots,k-1\} \rightarrow B$$

- All the theory we learned for single-output Boolean functions applies to the functions of the above type

Example

f_1: x_2 \ x_1	0	1
0	0	1
1	1	1

f_2: x_2 \ x_1	0	1
0	1	0
1	1	1

f_3: x_2 \ x_1	0	1
0	1	0
1	0	0

f_4: x_2 \ x_1	0	1
0	0	1
1	0	0

5 cubes

x_3 \ x_1x_2	00	01	11	10	
0	0	1	1	1	f_1
1	1	1	1	0	f_2
2	1	0	0	0	f_3
3	0	0	0	1	f_4

3 cubes