ELEC-H-310
Digital Electronics

Lecture01
Numeral systems and Boolean algebra

Dragomir MILOJEVIC
2021

Course information

Course objective

Walkthrough complete **digital system stack**:
from logic circuit design, processor micro-architecture,
to computer programming using C language

This is quite ambitious … but doable!

(obviously it is going to be more in “width” then in “depth”)

Content, lecture organisation & schedules

- Content divided in three parts:

  1. **Logic Circuits** – we will begin with this part
  2. **Programming language** – this is what you will practice during labs
     (I will give an introduction to C, all those who know this don't have to come)
  3. **Micro-processors architecture (micro-controllers)** – basic architecture aspects and link with program performance

- Organisation

  - **Lectures** — 2 ECTS = 12 sessions, 2h/session
  - **Exercises** (pen&paper) — 1 ECTS = 6 sessions, 2h/session
  - **LABs** (with computers) – 2 ECTS = 6 sessions, 4h/session (UA5BEAMS)

- For exact schedules of all these look at Gehol (https://gehol.ulb.ac.be/)

Logic circuit ToC

1. Numeral systems and Boolean Algebra
2. Logic functions & optimisation using K-Maps
3. Circuit synthesis, Quine Mc.Cluskey logic optimisation
4. Synthesis & optimisation of synchronous logic circuits
5. Asynchronous sequential circuits
6. Sequential logic circuits

Important information

- Course material (and many other useful info) could be found on web: **ULB virtual university** (http://uv.ulb.ac.be/)
  - Lecture notes – these slides
  - Exercises – problems that you should try to solve (individually!)
  - Solutions – **Attention!** these are not complete (on purpose …)
- At the end of lectures/exercises/labs I can organise Q&A session (anytime before the exam); typically runs AM or PM, you come ask me all the questions you can think off
  - **But this is not a 2nd lecture!** (so don't ask me things that I have already explained in depth during lectures)
- **Exam** is **written 4h (2h for logic circuits + 2h programming)**
  - You need training since **timing & precision** are crucial for a success
  - The exam is not that difficult … only if you work regularly (by yourself) & **you really understand what are you doing**

Advice(s)

- Work **regularly** & **in order**
  - Doing Lecture/Exercise X without doing X-1 doesn’t make sense
- First part of the lectures (Logic Circuits) is quite **algorithmic**
- You will learn many recipes that you could successfully apply to various problems without necessarily knowing what are you doing (you could behave like a computer!): **this is the wrong attitude!**
- By knowing what you do:
  - you reduce risk of errors during the exam
  - you will be able to solve problems that don’t necessarily look like the ones you have already seen (you will have such problems for your exam)
  - you will be able to understand more complex things related to embedded system design & computing systems in general
- And eventually you don’t want to be compared to a computer 😁

Today

1. Number representation
2. Base conversion techniques
3. Arbitrary base conversions
4. Useful bases
5. Arithmetic operations
6. Negative numbers
7. Boolean algebra

1. Number representation

Real numbers

Decimal numbers, **fixed point**, base 10 :

$$(1372.6450)_{10}$$

1 3 7 2	.	6 4 5 0
Integer part	Decimal point	**decimal part**
10^3 10^2 10^1 10^0		10^{-1} 10^{-2} 10^{-3} 10^{-4}

$1 \times 10^3 + 3 \times 10^2 + 7 \times 10^1 + 2 \times 10^0$	+	$6 \times 10^{-1} + 4 \times 10^{-2} + 5 \times 10^{-3}$

General form (for any base) 1/2

- Number N in base r, noted Nr
- n+1 digits, de a_0, ..., a_n → **integer part** (index i)
- m digits, de b_1, ..., b_m → **decimal part** (index j)

This is a series of digits!
(dot here is not a multiplication)

$$
\begin{aligned}
N &= (a_n \cdot a_{n-1} \cdot \ldots \cdot a_0 \cdot b_0 \cdot \ldots \cdot b_m)_r \\
N &= a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \ldots + a_0 \cdot r^0 + b_1 \cdot r^{-1} + b_1 \cdot r^{-2} + b_m \cdot r^{-m} \\
N &= \sum_{i=0}^{i=n} a_i \cdot r^i + \sum_{j=1}^{i=m} b_j \cdot r^{-j}
\end{aligned}
$$

Integer **Decimal**

General form (for any base) 2/2

$$N = \sum_{i=0}^{i=n} a_i r^i + \sum_{j=1}^{i=m} b_j r^{-j}$$

Indexes i, j are called **weights**

- For the integer part we speak about:
  - $i{=}n$ — bit with **highest** weight: **Most Significant Bit – MSB**
  - $i{=}0$ — bit with **lowest** weight: **Least Significant Bit – LSB**

Useful bases

- In these lectures 4 useful bases: base 10 (because of humans) and 3 others because of computers (all power of 2):
  - `r=10 – decimal       {0,1,2,3,4,5,6,7,8,9}`
  - `r= 2 – binary        {0,1}`
  - `r= 8 – octal         {0,1,2,3,4,5,6,7}`
  - `r=16 – hexadecimal   {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}`
- Important to learn: **how to switch from one base to another**

  → **Conversions & arithmetic computations** (basic operations)
- Conversions
  - Arbitrary base: not that hard (if you know the algorithm)
  - For bases 2, 8 et 16: it is even simpler

Basic numbers in different (useful) bases

Decimal	Binary (2)	Octal (8)	Hexadecimal (16)
0	00000	0	0
1	00001	1	1
2	00010	2	2
3	00011	3	3
4	00100	4	4
5	00101	5	5
6	00110	6	6
7	00111	7	7
8	01000	10	8
9	01001	11	9
10	01010	12	A
11	01011	13	B
12	01100	14	C
13	01101	15	D
14	01110	16	E
15	01111	17	F
16	10000	20	10

2. Base conversion techniques

Conversion dec2bin – integer part

Number A, base r=2, coded with 4 digits is written in the following form:

$$A = \sum_{i=0}^{i=3} a_i \cdot r^i$$

$$A = a_0 \cdot 1 + a_1 \cdot 2 + a_2 \cdot 2 \cdot 2 + a_3 \cdot 2 \cdot 2 \cdot 2$$

$$A = 2 \cdot \left(2 \cdot \left(2 \cdot (a_3)1 + a_2\right) + a_1\right) + a_0$$

Remainder (after division)

`(A-a₀):2` → remainder of the division is LSB

`(((A-a₀):2)-a₁):2)` → next bit

… → etc.

Successive divisions for the integer part

(Successive multiplications for the decimal part)

Example: conversion dec2bin – integer part

$(245)_{10}=(?)_2$

Applying method of successive divisions

Number to convert | **Base** | **Remainder**

245	:2	1
122	:2	0
61	:2	1
30	:2	0
15	:2	1
7	:2	1
3	:2	1
1	:2	1
0		

Computation direction

Converted number reading direction (MSB → LSB)

245:2=122 remains 1

122:2= 61 remains 0

...

$(245)_{10}=(11110101)_2$

MSB LSB

Example: conversion dec2bin – decimal part

$(0.345)_{10} = (?)_2$

Applying method of successive multiplications

.345	x2	0.690	0
.690	x2	1.380	1
.380	x2	0.760	0
.760	x2	1.520	1
.520	x2	1.040	1
.040	x2	0.080	0
.080	x2	0.160	0
.160	x2	0.320	0
.320	x2	0.640	0
.640	x2	1.280	1
.280	x2	0.560	0

Computation direction

Converted number reading direction (MSB → LSB)

$(0.345)_{10} = (.0101100001\ldots)_2$

When do you stop?

Depends on the precision you want to achieve! (this is a priori given)

Example of bin2dec conversion (the other way)

- Integer part:

$(11110101)_2 = (?)_{10}$

$= 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 1 \times 2^6 + 1 \times 2^7$

$= 1+0+4+0+16+32+64+128$

$= 245$

- Decimal part:

$(.0101100001..)_2 = (?)_{10}$

$= 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5} + 0 \times 2^{-6} + 0 \times 2^{-7} + 0 \times 2^{-8} + ...$

$= 0 \times 1/2 + 1 \times 1/4 + 0 \times 1/8 + 1 \times 1/16 + 1 \times 1/32 + 0 \times 1/64 + ..$

$= (8+2+1)/32 = 11/32 = 0.34375$

Example of dec2oct conversion

Attention this is octal !!!

Converted number reading direction
(MSB → LSB)

245	:8	5
30	:8	6
3	:8	3
0		

Computation direction

```
245:8=30 reminder 5
 30:8= 3 reminder 6
  3:8= 0 reminder 3
```

$$(245)_{10}=(365)_8$$

Verification:

$$
\begin{aligned}
(365)_8 &= 5\mathrm{x}8^0+6\mathrm{x}8^1+3\mathrm{x}8^2\\
&= 5+48+192\\
&= (245)_{10}
\end{aligned}
$$

Example of dec2hex conversion

Attention this is HEX !!!

245	:16	5
15	:16	F
0		

245:16 = 15 remainder 5

15:16 = 0 remainder F

$$(245)_{10} = (F5)_{16}$$

$$
\begin{aligned}
(A1F.1C)_{16} &= (?)_{10} \\
&= A*16^{2}+1*16^{1}+F*16^{0}+1*16^{-1}+C*16^{-2} \\
&= 10*16^{2}+1*16^{1}+15*16^{0}+1*16^{-1}+12*16^{-2} \\
&= 2560+16+15+28/256 \\
&= 2591.1093...
\end{aligned}
$$

3. Arbitrary base conversions

Problem definition

- Convert a number N, from base p into a number X in base q

$$(N)_p \quad \rightarrow \quad (X)_q$$

- How do you do this?
  - If you want to do this directly, it could be hard … and you will do it during the exercises **but not during the exam** ☺
  - Easy way (for the exam) you do the **intermediate conversion into base 10**:

$$(N)_p \quad \rightarrow \quad (x)_{10} \quad \rightarrow \quad (X)_q$$

- For **useful basis** start with base 2 first, the others will follow

$$(N)_{10} \quad \rightarrow \quad (?)_2 \quad \rightarrow \quad (?)_8 \quad \rightarrow \quad (?)_{10}$$

Example of arbitrary base conversion

$(25.34)_8 = (?)_5$

$(25.34)_8 = (?)_{10} = 2x8+5x1+3x8^{-1}+4x8^{-2}=...= (21.4375)_{10}$

$(21.4375)_{10} = (?)_5$

21	:5	1
4	:5	4
0		

.4375	x5	2.1875	2
.1875	x5	0.9375	0
.9375	x5	4.6875	4
.6875	x5	3.4375	3
.4375	x5	2.1875	2
.1875	x5	0.9375	0

$(21.4375)_{10} = (41.20432.....)_5$

4. Useful bases

Conversion technique for base 8 & 16

- Simple: you need to **group** binary digits into **packs of 3 bits** for octal **or 4 bits** for hexadecimal conversion (starting from LSB)

Base	Number		
10	245		
2	11110101		
Par 3	11	110	101
8	3	6	5
Par 4		1111	0101
16		F	5

Base	Number		
16	1F2		
2	0001 | 1111 | 0010		
Par 3	111	110	010
8	7	6	2

Conversion example from binary

$(378)_{10} = (0101111010)_2 = (?)_8, (?)_{16}$

$(x)_{10} = (x)_2$	101111010		
Grouping in 3 bits packs	101	111	010
Base 8 (octal)	5	7	2
Grouping in 4 bits packs	0001	0111	1010
Base 16 (hexa)	1	7	A

5. Arithmetic operations

Binary addition of two words (1/2)

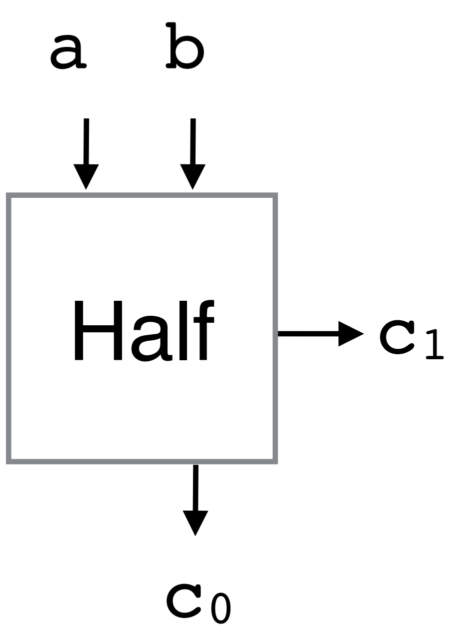
- **Half-adder** – elementary circuit, two 1 bit wide words `a,b`

$$c=a+b$$

Result, the word c, encoded with 2 bits c_1c_0

a	b	c_1	c_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Report
(Carry)



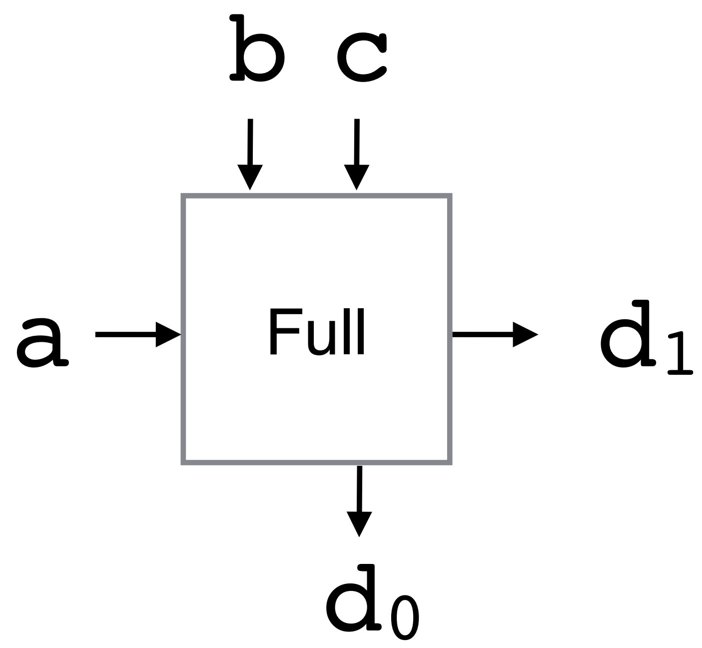
Binary addition of three words (2/2)

- **Full-adder** – elementary circuit, three 1 bit wide words `a,b,c`

$$d=a+b+c$$

Result, word `d`, encoded with bits $d_1 d_0$

a	b	c	d_1	d_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Addition of two n-bits words

`c=a+b`

Result is the word `c` encoded with `n+1` bits !

r_{n-1} … r_0

----

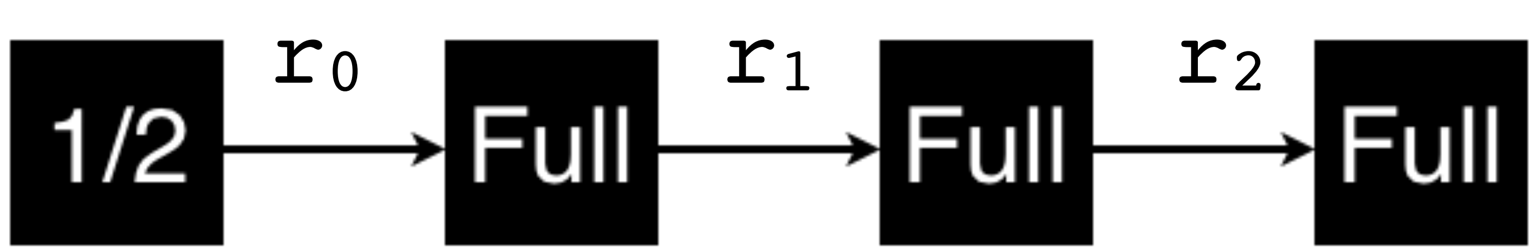
a_{n-1} … a_0

b_{n-1} … b_0

----

c_n … c_0

- We do sum of every bit, starting with LSB
- First sum (a_0 , b_0) can be computed using **one half-adder circuit**
- The others can be computed using **`n-1` full adder circuits**
- These are assembled in a **serial circuit**



Addition example: two 8 bit words

`c = a + b = 236 + 170 = ?`

		128	64	32	16	8	4	2	1
Bit		7	6	5	4	3	2	1	0
Report	1	1	1	0	1	0	0	0	
a		1	1	1	0	1	1	0	0
b		1	0	1	0	1	0	1	0
c	1	1	0	0	1	0	1	1	0

Verification:

$236+170=(436)_{10}$

$(436)_{10}$ = 1 1001 0110

Problem of the **overflow** – result will be **truncated**

What does this mean?

(if the above result is encoded into a 8-bit word)

6. Negative numbers

In this section

- Three different **forms** of negative numbers representation:

  a. **Sign and Magnitude (SM)**

  b. **One's complement (C1)**

  c. **Two's complement (C2)**

- **Conversion** – How to convert numbers in any of the above (from base 10 most of the time)

- **Basic arithmetic operations** – Here we focus mostly on C2 (this is what every single computing machine is using today, the others are there for your general knowledge only)

a. Signed Magnitude (SM) representation

- One bit reserved for the **sign**, by convention:
  - `0` – positive
  - `1` – negative
- Other bits are reserved for **magnitude**
- Example for an 8-bit word:
  - We have 1 bit for the sign, and
  - 7 bits for magnitude; so we have 2^7 values, maximum is $2^7–1=127$
  - Therefore we can represent from `-127` to `+127` since:

```
1 1111111 (-127)      0 1111111 (+127)
```

a. SM: problem of 2 zeros

Decimal	Binary	SM
0	0 0000000	(+)0
1	0 0000001	1
...	...	...
127	0 1111111	+127
128	1 0000000	(-)0
129	1 0000001	-1
...	...	...
255	1 1111111	-127

Positive

Negative

Two different zeros

a. Arithmetics using SM

- **How to do arithmetics?** Let's take the example of subtraction …
  - We need to compare both **signs** to decide how to proceed
  - We need to compare the **magnitude** to decide on computation direction
  - Example:
    - ✓ A, B are of the same (+) positive sign
    - ✓ if $A > B$, then $A - B$
    - ✓ it $A < B$, then $B - A$
- **Advantages** – easy to understand, as we normally use it with base 10 (by placing a negative sign in front); it is easy to convert to and from …
- **Disadvantages** – there are 2 different symbols with the same meaning; computations are more complex and slow, since we need adjustments; to do that we need special circuits to deal with (so more expensive HW)

b. One's complement

- Works for any base!
- **Motivation — compute subtraction like an addition**; this means that the same HW circuit could do two different operations (add / sub)
- Assume two words A & B in base r encoded using **`m` digits**

$$
\begin{aligned}
A-B &= A + (-B) \\
&= A + (r^m - B) = A + B'
\end{aligned}
$$

Complement to base 10
(radix complement)

$B+B'=r^m$

Example: $r=10$, $m=3$, $r^m=1000$

```
+87  →           87     →       87
-53  →  1000-53=947     →  +   947
--------------------------------------
                              1034
```

b. Complement to base

- To do A-B we need to compute: $B' = (r^m - B)$
- This is subtraction anyhow …
- We re-organise:
  $B' = (r^m - B) = ((r^m - 1) - B) + 1$
  where $(r^m - 1) - B$ is **complement of each digit** of B

m digits of $r-1$ e.g. $m=3$, $r=10$ -> $1000-1=999$

$(r^m-1)-B = ((r-1)(r-1)\ldots(r-1))-(b_{m-1}b_{m-2}\ldots b_0)$

Note ! this is not multiplication, but **concatenation**

$= ((r-1)-b_{m-1})((r-1)-b_{m-2})\ldots((r-1)-b_0)$

$= b'_{m-1}b'_{m-2}\ldots b'_0$

Complement of each digit → in binary this is very handy:
each digit is simply inverted !!! (very simple to do in HW)

b. Two variants of the complement

One we have already introduced:

1. **One's complement** `(r`^{m}`−1)−B`

Attention we still have 2 zeros :

`00000000` and `11111111`

2. **Two's complement** `(r`^{m}`−1)−B+1`

there is a **skew**, but only one zero...

Decimal	Binary	C1
0	00000000	(+)0
1	00000001	1
...	...	...
127	01111111	+127
128	10000000	-127
129	10000001	-126
...	...	...
255	11111111	(-)0

Decimal	Binary	C2
0	00000000	0
1	00000001	1
...	...	...
127	01111111	+127
128	10000000	-128
129	10000001	-127
...	...	...
255	11111111	-1

Conversions to One’s & Two’s complement

- Conversion of the negative number
  - Absolute value of the number is converted into binary
  - We complete the number that we get with ‘`0`’ to get the desired length of the word (say `m` bits)
  - Every bit of this word is inverted to get **One’s complement** (C1)
  - We add 1’ to C1 and derive **Two’s complement** (C2)
- Properties of the **Two’c complement**
  - For a word of 8 bits we can represent from -128 to 127
  - Single zero: 00000000
  - Arithmetical operations are much more simple…

Arithmetic in One's & Two's complement

- We can do arithmetics using C1, but there are many particular cases that need to be analysed in order to decide on how to proceed
- In two's complement arithmetics is straightforward!
  - When the number is negative, convert this into C2
  - Perform subtraction as an addition
  - Exception: handling the overflow, and it is simple !
    - ✓ example of the overflow: `01000000+01000000=10000000`
- Rule — overflow bit is ignored as long as:
  - the result of the operation $c=a-b$ is in the range: $-r^m < c < r^m-1$
  - for 8 bits this is : $-2^7 < c < 2^7-1$, so: $-128 < c < 127$
  - and the two last bits have the same value

Example: conversion to SM, C1 & C2

- -23 using 8 bits: magnitude $(23)_{10}=(10111)_2$
  - Negative number: convert to binary, invert for C1 and add +1 for C2

SVA	C1	C2
1 001 0111	0001 0111	
	1110 1000	1110 1001

- 25 using 8 bits: magnitude $(25)_{10}=(11001)_2$
  - Positive number: don't do anything after the conversion to binary !!!

SVA	C1	C2
0 001 1001	0001 1001	0001 1001

Example: subtraction in C2

```
57 - 23 = ?

-23 in C2 with 8 bits : (23)10=( 1 0111)2 C2: (1110 1001)

 57 in C2 with 8 bits : (57)10=(11 1001)2 C2: (0011 1001)

  1 1111    1

    1110 1001

    0011 1001
_________________

 (1 0010 0010)2 = (34)10
```

Rule – We examine last two bits of the report:

* if they are the same (00 or 11) result is OK

* **if not there is an overflow !!!**

Here we can ignore the overflow because two bits are the same and the result is in the range:

$$-128<34<127$$

7. Boolean algebra

Definition of Boolean algebra

- Boolean algebra is a **quadruplet** `{B,‘, •, +}`, where:
  - `B` → is a two-element set
  - `‘` → is the complement operator
    (also represented with a vertical line above the symbol)
  - `•` → is the operator AND
  - `+` → is the operator OR
- Depending on how we define the set B and the 3 operators, we can define multiple Boolean algebras
- Here we focus on two valued Boolean algebra introduced & formalised by C. Shannon (1938) as a follow-up of the work from Edward V. Huntington (1904)

Two-valued Boolean algebra

- Quadruplet `{B,‘, •, +}` is defined as follows:
  - `B = {0,1}` where
    - ✓ `‘0‘` — FALSE and `‘1‘` — TRUE
  - and two binary **operators**:
    - ✓ + : used to denote **inclusive** (difference with exclusive) OR and
    - ✓ • : used to denote AND
- Operators definition is given using **Truth Tables**
  - **Truth Table –** tabular technique for listing all possible combinations of input variables or arguments and their resulting truth value
- **Attention : • & + are not arithmetical operators !** (although we might refer to them as “times” or “plus”)

AND/OR operators

- Following truth tables define AND & OR operators (here both inclusive & exclusive OR):

x	y	x•y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x⊕y
0	0	0
0	1	1
1	0	1
1	1	0

With these we can verify the 6 axioms introduced by E. V. Huntigton

Axioms of Boolean algebra

- For a quadruplet `{B,‘, •, +}` to be a Boolean algebra, the following axioms needs to be satisfied:
  - Axiom 1. **Closure** of `B` for (+) and for (•)
  - Axiom 2. **Neutral element** in B for (+) & (•) it is ‘`0`’ & ‘`1`’
  - Axiom 3. **Commutativity** for + et •
  - Axiom 4. **Distributivity** for • wrt + & + wrt •
  - Axiom 5. **Complement** for x
  - Axiom 6. There are 2 elements `x,y` in `B` so that `x ≠ y`
- **Associativity** is the consequence of the previous:

`(a+b)+c=a+(b+c) & (a•b)•c=a•(b•c)`

Axiom 1: Closure property for `B={0,1}`

- Result of each of the three operations (`+,•,'`) is in `B`
- We can examine the **Truth Tables** for all three operators:

x	x'
0	1
1	0

x	y	x•y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

- Results of all these operations do remain in `B` …

Axiom 2: Identity elements for values in B

- 0 et 1 are neutral elements for + and • respectively
- This can be verified using the definition of the operators + and • (cf. Truth Tables of these operators)
- For + :

$$x + 0 = x \qquad\qquad \begin{array}{l} 0 + 0 = 0 \\ 1 + 0 = 1 \end{array}$$

- For • :

$$x \bullet 1 = x \qquad\qquad \begin{array}{l} 0 \bullet 1 = 0 \\ 1 \bullet 1 = 1 \end{array}$$

Axiom 3: Commutativity of + and •

- For + :

  `x + y = y + x`

  This can be easily verified:

  `0 + 1 = 0 + 1 = 1`
  `1 + 0 = 1 + 0 = 1`
  `1 + 1 = 1 + 1 = 1`

- For • :

  `x • y = y • x`

  This can be (also) easily verified:

  `0 • 1 = 1 • 0 = 0`
  `1 • 0 = 0 • 1 = 0`
  `1 • 1 = 1 • 1 = 1`

Axiom 4: Distributivity of • with respect to +

$$x \bullet (y+z) = x \bullet y + x \bullet z$$

Verification using Truth Tables (one truth table per = side):

x	y	z	y+z	x•(y+z)	x•y	x•z	x•y+x•z
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Axiom 4: Distributivity of + with respect to •

$$x+(y \bullet z)=(x+y) \bullet (x+z)$$

Verification using Truth Tables (one truth table per = side):

x	y	z	y•z	x+(y•z)	x+y	x+z	(x+y)•(x+z)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Axiom 5: Complements

- ‘0’ and ‘1’ are the complements:

$$0 + 0' = 0 + 1 = 1$$

$$0 + 1' = 0 + 0 = 0$$

$$0 \bullet 0' = 0 \bullet 1 = 0$$

$$0 \bullet 1' = 0 \bullet 0 = 0$$

Complement can be seen as a
third operator: NOT (inverter)

x	y=NOT(x)
0	1
1	0

Axiom 6: Elements of B

- There are at least 2 elements x,y in B so that $x \neq y$
- Let's take our $B=\{0,1\}$
- There are 2 elements in $B=\{0,1\}$
- And they are indeed different since:

$0 \neq 1$

and

$1 \neq 0$

Basic theorems of Boolean algebra

- Th1: **Idempotency** for + et •

  $x+x=x$ & $x \bullet x=x$
- Th2: **Null**

  $x+1=1$ & $x \bullet 0=0$
- Th3: **Absorption**

  $x \bullet (x+y)=x$
- Th4: **Involution**

  $(x')'=x$
- Th5: **Associativity**

  $(x+y)+z=x+(y+z)$ & $(x \bullet y) \bullet z=x \bullet (y \bullet z)$
- Th6: **De Morgan's laws** (19th century)

  $(x+y)' = x' \bullet y'$ & $(x \bullet y)' = x' + y'$
- Th7: **Consensus**

  $x \bullet y + x' \bullet z + y \bullet z = x \bullet y + x' \bullet z$

Proofs

- Different ways to **prove** any theorem of Boolean algebra
  - **Proof** – demonstrate the correction of a given Boolean proposition
- Different methods:

  a) Using **axioms** and/or **already proven theorems**

  - We will do this during exercises but just as an illustration (I am not going to ask you to do this for the exam)

  b) **Truth Tables**

  - We can do this since the number of combinations is limited

  c) **Duality principle**

a) How to prove propositions & simple example

- We start with one of the two proposition (either side of = sign)
- We transform the expression using axioms and/or theorems, typically one at the time, stating exactly which axiom/theorem we are using
- Let's give a proof for Th1 (`x+x=x`) using axioms only :

```
x+x      = (x+x)•1          Ax2. Neutral
         = (x+x)(x+x')      Ax5. Complement
         = x+x•x'           Ax4. Distributivity
         = x+0              Ax5. Complement
         = x                Ax2. Neutral
```

b) Proving theorems using Truth Tables

- Similar to what we did for the verification of distributivity axiom
- In two valued Boolean algebra theorems are **combinatorial problems**
- There is a **limited number of possibilities** for the function arguments
- We check both expressions **for all combinations** of the arguments
- Number of variables determines the size of the truth table
- For each side of the proposition (left, right of the =) we will have one truth table (two truth tables in all)
- Two sides of the proposition need to be equal, so the two truth tables need to be the same
- We compare these two; if they are identical, the proposition is ok

b) Example of the proof

De Morgan's laws

$(x+y)' = x' \bullet y'$

x	y	x+y	(x+y)'	x'	y'	x'•y'
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

b) Example of the proof

De Morgan's laws

$(x \bullet y)' = x' + y'$

x	y	x•y	(x•y)'	x'	y'	x'+y'
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

c) Duality principal

- In Boolean algebra every result has a **dual**
- If there is a result S in Boolean algebra, $S*$ is the dual of S
- Dual $S*$ can be obtained by systematically **permuting**:
  - operators $+$ and $\bullet$ **and**
  - symbols 0 & 1 in B
- What does this really means is the following:

Duality principal asserts that Boolean algebra is unchanged when all dual pairs are interchanged

c) Duality principle – example

- Let's take the following result S of Boolean algebra:

  $\forall x,\ x \in B,\ x+x=x,$ this is **idempotency**

  the dual of S noted S^* is:

  $$\forall x,\ x \in B,\ x \bullet x = x$$

- Note that in the above we replaced x' with x
- This can be applied to De Morgan's laws:
  - If the following is true:

    $(x+y)' = x' \bullet y'$

  - then, **the following is true too:**

    $(x \bullet y)' = x' + y'$

c) Duality principle – generalising to n variables

It is possible to generalise the previous result on n variables:

$$\forall x_i,\ x_i \in B\ ,\ E(x_1, \ldots, x_n) \rightarrow E^*(x_1, \ldots, x_n)$$

Example of application: **De Morgan's laws with n variables**

$$(x_1 + \ldots + x_n)' = x_1' \bullet \ldots \bullet x_n'$$

and also:

$$(x_1 \bullet \ldots \bullet x_n)' = x_1' + \ldots + x_n'$$