DIGITAL SYSTEMS

Hans du Buf and Paulo Oliveira
Department of Electronics and Computer Science
Faculty of Sciences and Technology
University of the Algarve

Contents

	Chapter	Page
1	Foreword	2
2	Number systems and codes	4
3	Boole algebra and logical functions	16
4	Physical elements	21
5	Series developments	25
6	Simplification of logical functions	36
7	Memory elements	46
8	Sequential circuits	59
9	Registers and functional elements	74
10	ALU design	82
11	Microprogramming	89
12	Mass memory	94

1. Foreword

Sistemas Digitais, or Digital Systems, is a one-semester course in the two majors Electronics and Computer Science. The objective is a general introduction into digital logic and circuits and, in fact, a preparation for understanding the bits and bytes of computers that will be treated in the succeeding course, *Arquitectura de Computadores,* or Computer Architecture. More explicitly, we will see

1. Number systems, conversions, arithmetic, and codes
2. Boole algebra
3. Combinatorial circuits and simplifications
4. Flip-flops and counters
5. Sequential circuits, synchronous and asynchronous
6. Registers and functional elements
7. A simple ALU design
8. Bus structure of a simple microprocessor
9. Mass memory, RAM and ROM, static and dynamic, address decoding
10. Logic families, in particular the *TEXAS INSTRUMENTS* SN74xx TTL series

At the end of this course, students must be able to understand and work with logical functions like NAND and NOR, read schemes of digital circuits, design circuits and have a basic notion of the anatomy of a simple computer.

In a one-semester course it is simply impossible to treat all subjects in depth. Hence, this course is only an *introduction* into the most common and fundamental techniques. This syllabus, which is based on notes from similar courses, notably from the Technical University of Eindhoven, The Netherlands, is meant to contain **all** examination material. Examination can be done without any problem after having studied this syllabus. For a more detailed analysis of e.g. sequential circuits, I advise the reading of some basic text books: "Introduction to switching theory and logical design," F.J. Hill and G.R. Peterson, Wiley and Sons, 1974, or "Modern digital design," R.S. Sandige, McGraw-Hill, 1990.

Because students in computer science must be able to read original computer documentation, this syllabus is completely written in English.

Hans du Buf and Paulo Oliveira, 2004

PS: This syllabus may still contain errors. For preparing a better and error-free version, please report your suggestions for improvements (and errors you detect) using the next page. Doing this, you may gain extra points in your final result!

Digital Systems

Name:

Number: **Course:**

Errors:

Page ..... line .....
Reads:
Should be:

Page ..... line .....
Reads:
Should be:

Page ..... line .....
Reads:
Should be:

Page ..... line .....
Reads:
Should be:

Page ..... line .....
Reads:
Should be:

Other comments and suggestions:

Please send or hand to Prof. Hans du Buf, FCT, Campus de Gambelas.

2. Number systems and codes

We are surrounded by codes. We are even built according to a code, the genetic one: DNA. Languages can be seen as codes and different languages can be grouped into similar code groups. For example, Portuguese is one of the Roman languages (leite, latte, leche, du lait), whereas others belong to the Germanic group (Milch, milk, melk).

Computers and control systems must be able to deal with numbers and to communicate in a way that does not leave any room for ambiguities. Furthermore, because it is easier to construct electronic circuits that can work with two signal levels (zero and one), computer codes and arithmetic operations have been developed for the so-called binary system.

Numerical systems can be symbolic, like the Roman one. In the Roman system we have the following coding:

symbol	I	V	X	L	C	D	M
value	1	5	10	50	100	500	1000

These symbols allow the coding of numbers by sequences of symbols, like I, II, III, IV, V, VI, VII, VIII, IX, X, XI, etc. A repetition of symbols is allowed for most symbols: XX=20, XXX=30, CC=200, CCC=300, but not for V, L, and D. A lower symbol in front of a higher one means a subtraction: CM=900, XC=90.

Example: 1996 = MCMXCVI.

For us, the decimal system is more readable, simply because we are used to think in powers of 10 (we have 10 fingers; note, however, that the French also use 20, which is the number of fingers and toes: 90 = quatre-vingt-dix). In general, a number is represented by a sequence of digits separated by a comma (or point in the English system):

$$N_r = d_j d_{j-1} \ldots d_3 d_2 d_1 d_0 . d_{-1} d_{-2} d_{-3} \ldots d_{-k+1} d_{-k}$$

where:
N = the number representation
r = the number base (or radix)
$d_j \ldots d_0$ = the integer part, and
$d_{-1} \ldots d_{-k}$ = the fraction.

The parameter r can have *any* integer value ≥ 2, but only the values 2, 8, 10, and 16 are normally used. In the expression above we have the following digit weighting:

$$N_r = d_j \cdot r^j + \ldots + d_2 \cdot r^2 + d_1 \cdot r^1 + d_0 \cdot r^0 + d_{-1} \cdot r^{-1} + d_{-2} \cdot r^{-2} + \ldots + d_{-k} \cdot r^{-k}$$

In the case of the **decimal** system, $r = 10$ and we use the 10 symbols $0, 1, 2, 3, \ldots, 9$. *Example:*

$$1996 = 1 \cdot 10^3 + 9 \cdot 10^2 + 9 \cdot 10^1 + 6 \cdot 10^0 = 1 \cdot 1000 + 9 \cdot 100 + 9 \cdot 10 + 6 \cdot 1.$$

The **binary** system, with $r = 2$ and only the two symbols 0 and 1, is, since the appearance of computers, normal because it is very easy to make electronic circuits with only two stable states (voltages). The digits are called **bits** (from BInary digiTS). We note here

that circuits with 3 or more stable states (voltages) can be made, but these require much more components like resistors and transistors. Binary circuits allow a much higher integration density, e.g. VLSI (Very Large Scale Integration): Intel's Pentium-4 has 42 million transistors. According to Moore's law, ICs will have a density of 10^{12} bits/cm^2 in the year 2015. The human brain has about 10^{12} synapses/cm^3. If the technological development can sustain Moore's law, the number of atoms necessary for having one bit, which was in 2001 equal to 10^{10}, will be 1 in the year 2035! [1]

Sometimes we use the **octal** system $(0, 1, 2, 3, \ldots, 7;\ r = 8)$ or the **hexadecimal** system $(0, 1, 2, \ldots, 8, 9, A, B, C, D, E, F;\ r = 16)$, mainly for representing long series of bits. For example, we can define a colour in a webpage (html) by `color="#rrggbb"`, in which each colour R, G and B can go from 00_{16} to FF_{16} (values between 0 and 255), i.e. 1 byte per colour component. The most important representations are given in Table 1.

decimal	binary	octal	hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Table 1: Common number codes.

The next decimal number is 16, which is written as 10000 in binary, as 20 in octal and as 10 in hexadecimal. Hence, the code 10 (in decimal) can mean 10, 2, 8 or 16. Normally, we use a fixed number of bits, for example 4. Hence, it is usual to count in binary like 0000, 0001, 0010, 0011, 0100, etc.

2.1 Number conversions

Converting binary to/from octal and hexadecimal is fairly easy if we know Table 1. It is a simple matter of grouping the bits into groups of 3 or 4 starting at the comma, followed by a conversion of each group.

[1] IEEE Computer, Jan 2001, p. 34.

Examples:
$10011100101.01010_2 = 0100\ 1110\ 0101\ .\ 0101\ 0000_2 = 4E5.50_{16}$
$10011100101.01010_2 = 010\ 011\ 100\ 101\ .\ 010\ 100_2 = 2345.24_8$

The reason is that these systems are all based on powers of 2. This is different for other bases, like 10. The conversion is then based on successive divisions. If

$$N = d_3 \cdot r^3 + d_2 \cdot r^2 + d_1 \cdot r^1 + d_0 \cdot r^0$$

we have

$$N/r = d_3 \cdot r^2 + d_2 \cdot r^1 + d_1 + d_0/r.$$

The last term is a fraction, the **rest** d_0. Dividing successively all quotients, we obtain the other digits (rests) d_1, d_2, and d_3.

Example: Convert 430_{10} to binary; successive division by 2.

	430	215	107	53	26	13	6	3	1
quotient	215	107	53	26	13	6	3	1	0
rest	0	1	1	1	0	1	0	1	1

Reading the rests from right to left, the result is 110101110_2. The division process stops when the quotient is 0.

If we have a fraction, we use successive multiplications instead of divisions. If

$$N = d_{-1} \cdot r^{-1} + d_{-2} \cdot r^{-2} + d_{-3} \cdot r^{-3} + d_{-4} \cdot r^{-4}$$

we have

$$rN = d_{-1} + d_{-2} \cdot r^{-1} + d_{-3} \cdot r^{-2} + d_{-4} \cdot r^{-3}.$$

The first term is the integer digit d_{-1} and successive multiplication of the fraction gives the other digits d_{-2}, d_{-3}, and d_{-4}.

Example: Convert $.523_{10}$ to binary; successive multiplication by 2.

	.523	.046	.092	.184	.368	.736	.472	.944 etc
product	1.046	0.092	0.184	0.368	0.736	1.472	0.944	1.888
digit	1	0	0	0	0	1	0	1

The 8-bit approximation result is $.10000101_2$ and the multiplication can continue until the product becomes exactly 1.0. The more digits, the more accurate the conversion will be[2], although numbers which are a simple combination of a few powers of two can be represented exactly by a few bits, e.g. $0.625_{10} = 0.5_{10} + 0.125_{10} = \frac{1}{2} + \frac{1}{8} = 0.101_2$.

Example: Convert $.523_{10}$ to hexadecimal; successive multiplication by 16.

[2]The millionth decimal of the number π is 0, but π has more decimals. In binary, π needs many more bits, but in practice we use approximations in floating point.

```
          .523     .368     .888  etc
        x  16    x  16    x  16
        -----    -----    -----
         3138     2208     5328
         523      368      888
        +-----   +-----   +-----
product  8.368    5.888   14.208
  digit  8        5        E
```

This gives .85E$_{16}$, which equals .100001011110$_2$ (series 1000 0101 1110; see Table 1).

Check: .85E$_{16}$ = ? in decimal.

Sometimes we apply a trick in the conversion, but we need to know the powers of two or have a table: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, ...
Suppose we need to convert 1000_{10} to binary, octal or hexadecimal. We know that $1000 = 1024 - 24$, but it is easier to use $1023 - 23$ because we can eliminate corresponding bits instead of doing a binary subtraction:

```
1024 = 1 0 0 0 0 0 0 0 0 0 0
1023 = 0 1 1 1 1 1 1 1 1 1 1
  23 = 0 0 0 0 0 0 1 0 1 1 1
1000 = 0 1 1 1 1 1 0 1 0 0 0
```

Another handy trick can be applied if we have long "runs" of ones. This is based on the detection of the combinations 01 and 10 and is called number re-coding (applied, for example, in digital multipliers). The combination 01 becomes 10 (the one is positive) and the combination 10 becomes $\overline{1}0$ (the $\overline{1}$ is negative). Example:

```
0 0 1 1 1 1 0 = 16 + 8 + 4 + 2 = 30

          _
0 1 0 0 0 1 0 = 32 - 2 = 30
```

Mathematically: $N = \sum_{i=a}^{b} 2^i$, if bit a is the least significant 1 and b is the most significant 1. Then $2^{b+1} = N + \sum_{i=0}^{a-1} 2^i = N + 2^a$. Hence $N = 2^{b+1} - 2^a$.

2.2 Negative numbers

The numbers that we have seen above are all assumed to be positive. This means that in front of all numbers we can think a + sign, irrespective of the number basis (radix). If we want negative numbers, we put a – sign, of course. This is quite natural for us, but not for computers! For binary numbers we can use three different forms.

2.2.1 Sign–magnitude

The unsigned number, called magnitude, is complemented with a sign bit s to the left of the most significant number bit. This sign bit is 0 for positive numbers and 1 for negative ones. Assuming 4-bit numbers sxxx with xxx=101 (the absolute value 5), we get 0101=+5 and 1101=–5.

2.2.2 One's and two's complements

In order to simplify the electronic circuits for addition and subtraction, the so-called 1's and 2's complement codes are used. Reason: a 2's complement subtraction needs only two circuits: an adder and an inverter, and an adder is required for additions (see Chapter 10).

The 1's complement is obtained by inverting all bits of the corresponding positive number:

$$-5 = -0101 = \bar{0}\bar{1}\bar{0}\bar{1} = 1010.$$

The 2's complement (the name comes from subtracting a fraction from 2) is obtained by inverting all bits and then adding a 1:

$$-5 = -0101 = \bar{0}\bar{1}\bar{0}\bar{1} + 1 = 1010 + 1 = 1011.$$

This means that the 2's complement equals the 1's complement plus 1. We therefore get Table 2.

decimal	sign-magnitude	1's complement	2's complement
+127	0111 1111	0111 1111	0111 1111
+126	0111 1110	0111 1110	0111 1110
+25	0001 1001	0001 1001	0001 1001
+2	0000 0010	0000 0010	0000 0010
+1	0000 0001	0000 0001	0000 0001
+0	0000 0000	0000 0000	0000 0000
–0	1000 0000	1111 1111	0000 0000
–1	1000 0001	1111 1110	1111 1111
–2	1000 0010	1111 1101	1111 1110
–25	1001 1001	1110 0110	1110 0111
–127	1111 1111	1000 0000	1000 0001
–128	–	–	1000 0000

Table 2: Common 8-bit number codes.

From Table 2 we can see:

- The positive numbers are the same in all representations.
- The most significant bit (sign bit) is always 0 for positive and 1 for negative numbers.
- Only the 2's complement has the same representation for ± 0.
- The 2's complement allows to represent numbers that the other ones don't: –128.

All this means that the 2's complement is to be preferred. Note: another way to compute the 2's complement is to scan the bits from right to left (from least significant to most significant), copying the bits until we have the first 1 and then inverting all bits. *Check this!*

2.3 Arithmetic

2.3.1 Addition

A binary addition is done in the same way as a decimal one: adding the digits going from right to left and adding a carry (a 1) to the digits to the left, if required.

Examples:

```
  5+3 =  0000 0101      5+(-3) =  0000 0101
         0000 0011                 1111 1101  2's complement
        +----------               +----------
         0000 1000 = 8             10000 0010 = 2 (ignore carry bit!)
                                   ^carry
```

2.3.2 Subtraction

A subtraction can be done by using addition and the 2's complement:

$$N_1 - N_2 = N_1 + (-N_2) = N_1 + (\bar{N}_2 + 1).$$

Example:

```
 15-8 =  0000 1111
         1111 1000
        +----------
         10000 0111 = 7
```

NOTE: the carry bit is ignored if we use the 2's complement. By contrast, if we use the 1's complement, the carry bit needs to be added to the least significant bit. This is called an end-around carry. *Check this!*
If we don't use the 2's complement, we need to "borrow" from the left: $8 = 4 + 4 = 4 + 2 + 2 = 4 + 2 + 1 + 1$. For example:

```
 8 - 4 =  1 0 0 0        8 - 2 =  1 0 0 0        8 - 1 =  1 0 0 0
          0 1 0 0                 0 0 1 0                 0 0 0 1
        - -------               - -------               - -------
          0 1 0 0 = 4             0 1 1 0 = 6             0 1 1 1 = 7
```

Note: subtraction using the 1's complement can also be done in the decimal system, but then we use the 9's complement, of course. This was implemented in one of the most sophisticated mechanical calculators, the Curta[3]. For example,

```
   788139        0788139  left digit is zero (positive)
     4890        9995109  left digit is nine (negative)
 - ------      + -------
   783249        0783248
                       1  end-around carry!
               + -------
                 0783249
```

[3]Invented by Curt Herzstark, who developed some ideas in the Nazi concentration camp Buchenwald. See Scientific American, Jan. 2004; also `www.curta.org`.

2.3.3 Multiplication

The binary multiplication in the sign-magnitude form is performed by determining the resulting sign from the multiplication of the two signs of the operands (+x+=+, +x–=–, –x–=+; this an exclusive-OR operation, see Chapter 4) and placing this sign in front of the product of the magnitudes.

Example:

```
      5x107 =   110 1011
                x    101
                --------
                110 1011
               0000 000
              1 1010 11
             +-----------
 Sign ->  S 10 0001 0111 = 535
```

We see that, like in the decimal system, it is a repetition of shifts and additions (store intermediate results). If we have 2's complement numbers, one way to do the multiplication is to convert the number(s) to sign-magnitude, do the s-m multiplication, and then convert the result to the 2's complement. However, this is stupid because much more efficient ways are available.

2.3.4 Division

If a multiplication is based on a repetition of additions, it should be clear that a division is based on a repetition of subtractions. The most trivial case is the sign-magnitude representation. For example, when computing $17/4 = 4.25$, we get the following subtractions plus shifts for the fractional part:

```
    1 0 0 0 1
  - 0 0 1 0 0
    ---------
    0 1 1 0 1   1
  - 0 0 1 0 0
    ---------
    0 1 0 0 1   10
  - 0 0 1 0 0
    ---------
    0 0 1 0 1   11
  - 0 0 1 0 0
    ---------
    0 0 0 0 1   100
    0 0 0 1 0   shift, no subtraction, 100.0
    0 0 1 0 0   . subtraction possible!      . shifts
  - 0 0 1 0 0
    ---------
    0 0 0 0 0   100.01
```

Special hardware for multiplications and divisions is normally present in a computer. In many systems there are two circuits: a fixed-point multiplier and a circuit to compute the reciprocal (1/N), apart from the much more important floating-point units.

2.3.5 Over/Underflow

When adding two N-bit numbers, the result may not be representable with the N bits available; in many cases N+1 bits are required. In this case an **overflow** occurs. This will be studied in Chapter 10. In general, this condition is detected by a special electronic circuit, and normally in an adder unit or ALU (Arithmetic and Logical Unit) of the CPU (Central Processing Unit) of a computer. The same holds for a multiplication, e.g. multiplying two numbers of 8 bits can require 16 bits.

2.4 Codes

Not only numbers can be represented by sequences of 1s and 0s, but also letters, characters and other symbols. Combinations of bits are called digital codes. Codes can be classified as follows:

- Numerical codes for representing numbers.
- Alphanumerical codes for representing numbers and characters.
- Special codes for error detection and correction.

Here we will look at two numerical codes, i.e. the **8421** or **BCD** code and the **GRAY** code, as well as at an alphanumerical code, the **ASCII** code.

2.4.1 BCD

In the 8421 or BCD code (Binary Coded Decimal) each decimal digit is represented by a binary code of 4 bits. The weights of the bits are 8, 4, 2, and 1. The advantage of the BCD code is that it allows an easy conversion from binary to decimal formats and this code is often used in computer peripherals for entering or printing data. The BCD code is shown in Table 3.

The other bit combinations (e.g. 1011) are not valid. Representing a decimal number in BCD requires a substitution of each decimal digit by its binary code.

Example: $39.5_{10} = 0011\ 1001\ .\ 0101_{\text{BCD}}$

The reverse conversion requires grouping the bits in sets of 4, starting from the point or comma.

Example: $010100110010._{\text{BCD}} = 0101\ 0011\ 0010._{\text{BCD}} = 532_{10}$

The **addition** in BCD code is done as follows:

1. Add the two numbers as is done in the binary addition.
2. If the sum of 4 bits is equal or smaller than 9, this is a valid BCD number.

decimal	8421
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 3: The BCD or 8421 code.

3. If the sum is greater than 9 or if there is a carry, the result is not valid. In this case, we need to add 6 in order to skip the 6 invalid codes 1010 ... 1111 (see Table 3). A carry must be added to the bit group to the left.

Example: 36+84=120

```
  0011 0110  =  36
  1000 0100  =  84
 +----------
  1011 1010  both invalid because > 9
  0110 0110  add 6 to each group
 +----------
0001 0010 0000  = 120
```

2.4.2 GRAY

The Gray code allows error detection, when going from one position to the next/previous, because these only differ in one bit. It is used in systems which need to detect positions, for example in a computer mouse. The Gray code is also called mirror code, because of the way it is constructed (vertical mirroring around the –, changing 0s to 1s to the left):

```
0 0       0 0 0
0 1       0 0 1
  -       0 1 1
1 1       0 1 0
1 0         ---
          1 1 0
          1 1 1
          1 0 1
          1 0 0
```

Table 4 shows the complete 4-bit Gray code.
In order to convert from binary to Gray, we need to:
1. Keep the MSB (most significant bit)

decimal	Gray
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Table 4: The 4-bit Gray or mirror code.

2. Going from left to right, take the sum of two adjacent bits, ignoring carries.

Example:

```
10010111  =  binary
11011100  =  Gray
```

In order to convert from Gray to binary, we need to:
1. Keep the MSB (most significant bit)
2. Sum the next Gray bit with the previous binary bit.

Example:

```
11011100  =  Gray
10010111  =  binary
 ^
 0=1(left binary)+1(above Gray)
```

2.4.3 ASCII

The ASCII code (American Standard Code for Information Interchange) is an alphanumerical code for communication, for example for connecting a keyboard to a computer. An alphanumerical code has at least representations for the 10 decimal numbers and the 26 characters of the alphabet. Together this is already 36 codes and requires 6 bits (6 bits corresponds to a maximum of 64 different bit combinations). It also normally contains special codes for manipulating the information stream. Examples of special codes are LF (line feed) and CR (carriage return), which serve to start a new line in the first column of a computer screen. Normally, the ASCII code takes 7 or 8 bits (8 bits is 1 byte); see Table 5.

LSBs	MSBs							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	↑	n	˜
1111	SI	US	/	?	O	_	o	DEL

Table 5: The 7-bit ASCII code.

The most important abbreviations used in Table 5 are:
MSBs: most significant bits, LSBs: least significant bits

ACK: acknowledge
BEL: bell
BS: back space
CR: carriage return
EOT: end of transmission
ESC: escape
ETB: end of transmission block
FF: form feed
LF: line feed
SP: space

Example: a small text string could be

```
hexadecimal: 4D 75 69 74 6F 20 70 72 61 7A 65 72 0A 0D
             M  u  i  t  o     p  r  a  z  e  r
                            SP                  LF CR
```

3. Boole algebra

A Boole algebra is a set of elements $B = \{a, b, \ldots\}$ and two binary operators, i.e. operators defined between two elements. These operators are:

$\wedge$ meaning intersection, and
$\vee$ meaning union,

often, and also here, denoted by $\cdot$ (or .) and $+$. So $\wedge \equiv .$ and $\vee \equiv +$. These operators for the Boole algebra are defined as follows (the so-called *postulates;* Huntington, 1904):

1. $\forall a, \forall b, \exists c \in B$ such that $a + b = c$
   $\forall a, \forall b, \exists d$ such that $a.b = d$

2. **Commutative law:** $\forall a, \forall b$
   2.1 $a + b = b + a$
   2.2 $a.b = b.a$

3. **Associative law:** $\forall a, \forall b, \forall c$
   3.1 $(a + b) + c = a + (b + c)$
   3.2 $(a.b).c = a.(b.c)$

4. **Distributive law:** $\forall a, \forall b, \forall c$
   4.1 $a + (b.c) = (a + b).(a + c)$
   4.2 $a.(b + c) = (a.b) + (a.c)$

5. $\forall a \in B, \exists! z$ such that $a + z = a$
   We will denote this element by the symbol 0, so $a + 0 = a$.

6. $\forall a \in B, \exists! e$ such that $a.e = a$
   We will denote this element by the symbol 1, so $a.1 = a$.

7. $\forall a \in B, \exists! \bar{a}$ such that $a.\bar{a} = 0$ and $a + \bar{a} = 1$
   This element is called the **inverse** of a, and also denoted $a\prime$, $\neg a$, "not a" or "a bar".

Ad 5: z is unique.
Proof: Suppose $z_1 + a = a$
If $a = z_2$ then $z_1 + z_2 = z_2$
Suppose $z_2 + a = a$
If $a = z_1$ then $z_1 + z_2 = z_1$
Hence: $z_1 = z_1 + z_2 = z_2 \rightsquigarrow z_1 = z_2 = z$

Ad 6: e is unique.
Proof: Suppose $e_1.a = a$
If $a = e_2$ then $e_1.e_2 = e_2$
Suppose $e_2.a = a$
If $a = e_1$ then $e_1.e_2 = e_1$
Hence: $e_1 = e_1.e_2 = e_2 \rightsquigarrow e_1 = e_2 = e$

Ad 7: $\bar{a}$ is unique.
Proof: Suppose there are $\bar{a}_1$ and $\bar{a}_2$
$a + \bar{a}_1 = 1$
$a.\bar{a}_1 = 0$
$a + \bar{a}_2 = 1$
$a.\bar{a}_2 = 0$

Continue yourself with **6** *and* **4.2**!

Results of the postulates (corollaries)

R1:
$a.a = a.a + 0 = a.a + a.\bar{a} = a.(a + \bar{a})$ (see 4.2)
$= a.1 = a$. We say that a is *idempotent:* $a.a = a$

R2:
$a + a = (a + a).1 = (a + a).(a + \bar{a}) = a + (a.\bar{a})$ (see 4.1)
$= a + 0 = a$: $a + a = a$

R3:
$a + 1 = a + (a + \bar{a}) = (a + a) + \bar{a} = a + \bar{a} = 1$: $a + 1 = 1$

R4:
$a.0 = a.(a.\bar{a}) = (a.a).\bar{a} = a.\bar{a} = 0$: $a.0 = 0$

R5:
$a + a.b = a.1 + a.b = a.(1 + b) = a.1 = a$: $a + a.b = a$

We can now understand 4.1.
We have $(a + b)(a + c) = aa + ab + ac + bc$. Applying R1 simplifies aa to a. Then, with R5, we have $a + ab = a$ and $a + ac = a$. This leaves $a + bc$.

R6:
$a.(a + b) = a.a + a.b = a + a.b = a$ (use R5): $a.(a + b) = a$
R5 and R6 are the so-called *absorption theorems.*

R7:
$a + \bar{a}.b = (a + \bar{a}).(a + b)$ (see 4.1)
$= 1.(a + b)$: $a + \bar{a}b = a + b$

R8:
$a.(\bar{a} + b) = a.\bar{a} + a.b = 0 + a.b = a.b$: $a.(\bar{a} + b) = a.b$

R9:
$a'' = a$
Proof: Substitute $(a')'$ for b in R7.
This gives $a + (a')' = a + a'.(a')' = a + 0 = a$
Now substitute $(a')'$ for b in R8.
This gives $a.(a')' = a.(a' + (a')') = a.1 = a$. Hence $(a')' = a$

R10:
$(a + b)' = ?$
$(a + b).(\bar{a}.\bar{b}) = [a.(\bar{a}.\bar{b})] + [b.(\bar{a}.\bar{b})] = a.\bar{a}.\bar{b} + b.\bar{b}.\bar{a} = 0$
$(a+b)+\bar{a}.\bar{b} = [(a+b)+\bar{a}].[(a+b)+\bar{b}] = [(a+\bar{a})+b].[a+(b+\bar{b})] = (1+b).(a+1) = 1.1 = 1$

So $\bar{a}.\bar{b}$ is apparently the inverse of $a+b$: $\underline{(a+b)\prime = \bar{a}.\bar{b}}$
In general we have **R10a:**

$$(a+b+c+\cdots)\prime = \bar{a}.\bar{b}.\bar{c}.\cdots$$

R11:
$(a.b)\prime = ?$
$(a.b).(\bar{a}+\bar{b}) = a\bar{a}b + ab\bar{b} = 0$
$(a.b) + \bar{a} + \bar{b} = (\bar{a}+\bar{b}+a).(\bar{a}+\bar{b}+b)$ (see 4.1)
$= 1.1 = 1.$
So $\bar{a}+\bar{b}$ is apparently the inverse of $a.b$: $\underline{(a.b)\prime = \bar{a}+\bar{b}}$
In general we have **R11a:**

$$(a.b.\cdots)\prime = \bar{a}+\bar{b}+\bar{c}+\cdots$$

R12:
$(\sum_{i=1}^{n} a_i b_i)\prime = ?$
$(\sum_i a_i b_i)\prime = (a_1b_1 + a_2b_2 + a_3b_3 + \cdots)\prime$
Take $a_1b_1 = x$, $a_2b_2 = y$, $a_3b_3 = z$, etc.
$(x+y+z+\cdots)\prime = \bar{x}.\bar{y}.\bar{z}.\cdots$ (see R10a)
$= (a_1b_1)\prime.(a_2b_2)\prime.(a_3b_3)\prime.\cdots$
$= (\bar{a}_1+\bar{b}_1).(\bar{a}_2+\bar{b}_2).(\bar{a}_3+\bar{b}_3).\cdots.$ (see R11a)
Hence

$$(\sum_i a_i b_i)\prime = \prod_i (\bar{a}_i + \bar{b}_i)$$

The inverse of a sum of products is the product of the sums of the inverse elements!

Similarly (prove yourself!):
R13:

$$[\prod_i (a_i + b_i)]\prime = \sum_i \bar{a}_i \bar{b}_i$$

The inverse of a product of sums is the sum of the products of the inverse elements. R12 and R13 are the theorems of *de Morgan* (1847).

R14:
• $ab + \bar{a}c + bc =$
$ab1 + \bar{a}c1 + bc1 =$
$ab1 + \bar{a}c1 + bc(a+\bar{a}) =$
$ab1 + \bar{a}c1 + bca + bc\bar{a} =$
$ab(1+c) + \bar{a}c(1+b) =$
$ab + \bar{a}c$
The term bc has been *absorbed; bc* is *redundant.*

• $(a+b)(\bar{a}+c)(b+c) =$
$(a+b+0)(\bar{a}+c+0)(b+c+a\bar{a}) =$
$(a+b+0)(\bar{a}+c+0)(a+b+c)(\bar{a}+b+c) =$

$[(a+b)+0.c].[(\bar{a}+c)+0.b] =$
$(a+b)(\bar{a}+c)$
The term $(b+c)$ has been absorbed. Hence, combining the last two results we have

$$(a+b)(\bar{a}+c) = a\bar{a} + ac + \bar{a}b + bc = ac + \bar{a}b + bc = ac + \bar{a}b$$

Duality
From all theory we notice that all rules come in pairs. Examples:

$a + 0 = a$
$a.1 = a$

and

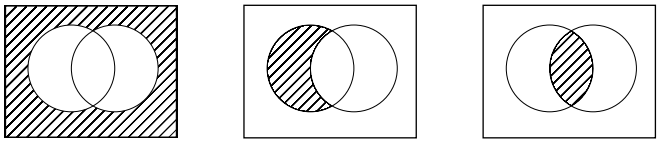
$a + (b.c) = (a+b).(a+c)$
$a.(b+c) = (a.b) + (a.c)$

Every theorem which can be proven for the Boole algebra has a *dual* form which also holds! The four conversion rules are:

$. \rightarrow +$
$+ \rightarrow .$
$1 \rightarrow 0$
$0 \rightarrow 1$

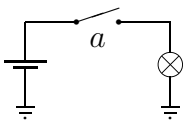
Finally, we will often work with the **binary set** $B = \{0, 1\}$.
$a.1 = a$ implies $1.1 = 1$ $(a = 1)$ and $0.1 = 0$ $(a = 0)$
$a + 0 = a$ implies $1 + 0 = 1$ $(a = 1)$ and $0 + 0 = 0$ $(a = 0)$

The distributive laws, especially 4.1, and other results, may be difficult to understand. We can check relations by using Venn diagrams:



$\bar{a} \cdot \bar{b}$ $\quad$ $a \cdot \bar{b}$ $\quad$ $a \cdot b$

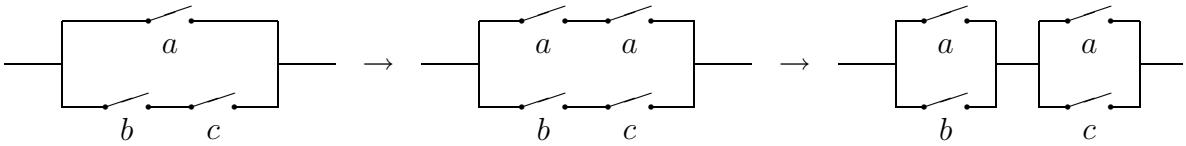
Another simple way is to use switching elements, such as the one shown below, in combination with a battery and a small lamp:



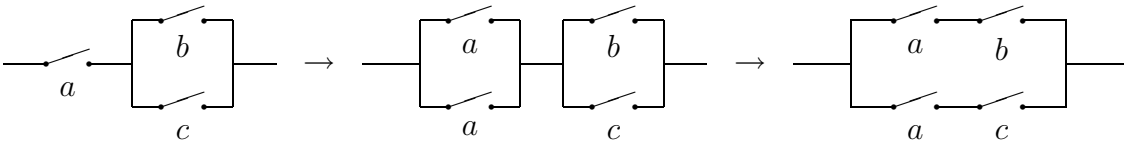
The switch a is open $(a = 0)$, but if we close it $(a = 1)$ the battery causes a current through the lamp which will light up. This is the simplest case, but we can combine two switches in series $(a.b)$ or in parallel $(a + b)$:



The left and right sides of postulate 4.1 become:



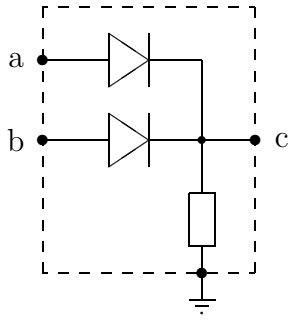
By checking all possible combinations we see that the left and right circuits are equivalent. In the same way we can illustrate 4.2:



4. Physical elements and logical functions

Take a box with three connectors a, b, and c, and a fourth one for *ground.* On a and on b we can put a voltage (positive) and measure the voltage on c. Call "voltage"=H and "no voltage"=L. H comes from high, L from low. Using simple electronic components like diodes[4], resistors, and transistors, we will introduce a few basic logical functions.

In the scheme below, a current can only flow from left to right. If the voltage at input a is positive, a current flows through the upper diode and through the resistor. If, at the same time, input b is connected to ground (0 V), the current that goes through the upper diode cannot flow through the lower diode. Hence, if a=H and b=L a current flows through the resistor and c=H. On the other hand, if a=L and b=L, no current flows through the resistor and c=L. So we can measure the output c for all possible combinations of H and L on the inputs a and b. We then get the left table:

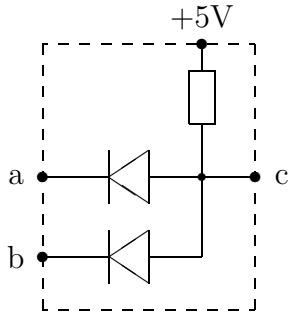


a	b	c
L	L	L
L	H	H
H	L	H
H	H	H

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

If we map {H,L} one-to-one on the set {1,0}, such that H $\rightarrow$ 1 and L $\rightarrow$ 0, we get the right table, which we call a **truth table**. From this we see that a+b=c. We call this the **OR function**.

Another box, but now with an input connected to + or H instead of to ground:



a	b	c
L	L	L
L	H	L
H	L	L
H	H	H

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

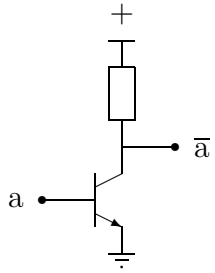
If both inputs a and b are H, there is no current through the resistor and c will be H. However, if one or both of the inputs a and b is L, there will be a current and the voltage on c drops to L (in this scheme, a current can only flow from right to left; take a look at the diode definition). From the right table we see that this circuit realises the function a.b=c. We call this the **AND function**. So we have seen that two postulates

[4]A diode is a nonlinear electronic component that can conduct current in one direction but not in the other. The symbol's arrow shows the way that the current can go.

of the Boole algebra can be realised physically.

Another postulate is $a.\bar{a} = 0$ together with $a + \bar{a} = 1$. We can realise $\bar{a}$ with a transistor (which is used here as a switching element, i.e. a controllable diode):

If a is L, the transistor does not conduct and $\bar{a}$, the output, is H.
If a is H, the transistor conducts, there is a voltage drop over the resistor and $\bar{a}$ is L.



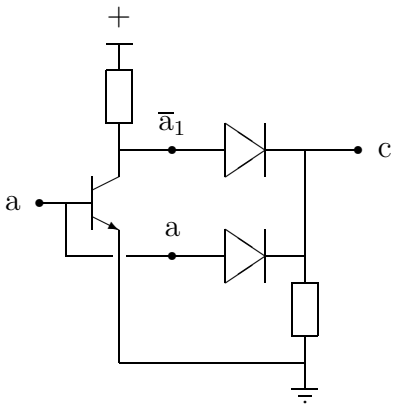
This gives the tables

a	$\bar{a}$
L	H
H	L

and

a	$\bar{a}$
0	1
1	0

If we connect a and $\bar{a}$ to a circuit that realises $a + b$, the OR circuit described above, we get:



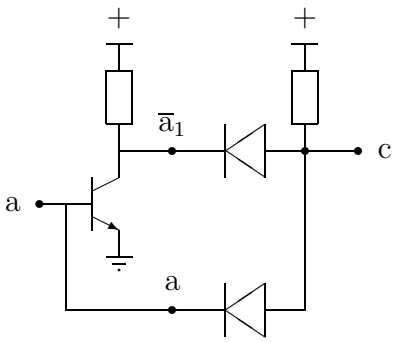
a	$\bar{a}_1$	c
L	H	H
H	L	H

LL and HH cannot occur!

a	$\bar{a}_1$	c
0	1	1
1	0	1

Hence $a + \bar{a}_1 = 1$

Similarly, the postulate $a\bar{a} = 0$ can be realised by taking the AND circuit:



a	$\bar{a}_1$	c
L	H	L
H	L	L

LL and HH cannot occur!

a	$\bar{a}_1$	c
0	1	0
1	0	0

Hence $a\bar{a}_1 = 0$

This proves that $a + \bar{a}_1 = 1$ and $a\bar{a}_1 = 0$. In other words, $\bar{a}_1$ exists and equals $\bar{a}$. In the same way the other postulates can be realised. For example, the associative property $(a + b) + c = a + (b + c)$ requires two OR circuits, normally called OR *gates,* in series. The distributive property $a + bc = (a + b)(a + c)$ requires one AND gate and one OR gate to realise the left term and two OR gates and one AND gate for the right term. In all cases the truth tables for the left and right terms must be equal.

4.1 Function overview

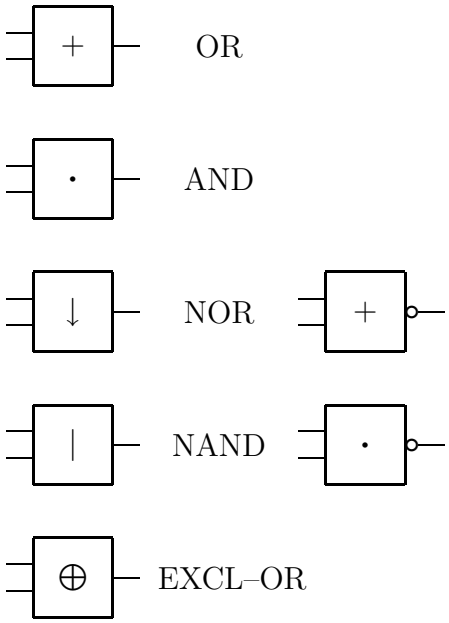
With two variables multiple functions can be realised, 16 in total. One of these is $f(a,b) = 0$, i.e. a circuit that is always open. Another one is $f(a,b) = 1$, i.e. a circuit that is always closed. Table 6 lists all functions of two variables. It consists of $2^2 = 4$ columns with the input combinations 00, 01, 10 and 11. In general, for n variables there are 2^n columns.
In the table we see that at first there are no 1s: C_0^4
then there is one 1 in 4 positions: C_1^4
then two 1s in 6 positions: C_2^4
then three 1s in 4 positions: C_3^4
then four 1s in 1 position: C_4^4
Together $C_0^4 + C_1^4 + C_2^4 + C_3^4 + C_4^4 = 2^4 = 16$ functions possible.

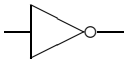
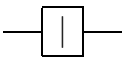
With n variables $\sum_{i=0}^{2^n} C_i^{2^n} = 2^{2^n}$.
For $n = 3 \rightarrow 256$
For $n = 4 \rightarrow 65{,}536$
For $n = 5 \rightarrow 4{,}294{,}967{,}296$
For $n = 6 \rightarrow \infty$

a	1	0	1	0		
b	1	1	0	0		
T_0	0	0	0	0	zero function (open, false)	$\bar{T}_{15}$
T_1	0	0	0	1	$\bar{a}\bar{b}$, $a \downarrow b$, $\overline{a+b}$, NOR, Peirce function	$\bar{T}_{14}$
T_2	0	0	1	0	$a.\bar{b}$, $a \not\leftarrow b$, inhibition, exclusion	$\bar{T}_{13}$
T_4	0	1	0	0	$\bar{a}.b$, $b \not\rightarrow a$, inhibition, exclusion	$\bar{T}_{11}$
T_8	1	0	0	0	$a.b$, AND, conjunction	$\bar{T}_7$
T_3	0	0	1	1	$\bar{b}$, $\neg b$, negation, inversion of b	$\bar{T}_{12}$
T_5	0	1	0	1	$\bar{a}$, $\neg a$, negation, inversion of a	$\bar{T}_{10}$
T_6	0	1	1	0	$a\bar{b} + \bar{a}b$, $a \oplus b$, $a \not\equiv b$, EXCLUSIVE–OR	$\bar{T}_9$
T_9	1	0	0	1	$ab + \bar{a}\bar{b}$, $a \odot b$, $a \equiv b$, equal	$\bar{T}_6$
T_{10}	1	0	1	0	a identity	$\bar{T}_5$
T_{12}	1	1	0	0	b identity	$\bar{T}_3$
T_7	0	1	1	1	$\bar{a} + \bar{b} = \overline{a.b}$, $a\|b$, NAND, Sheffer function (1913)	$\bar{T}_8$
T_{11}	1	0	1	1	$a + \bar{b} = \overline{\bar{a}.b}$, $b \rightarrow a$, implication	$\bar{T}_4$
T_{13}	1	1	0	1	$\bar{a} + b = \overline{a.\bar{b}}$, $a \rightarrow b$, implication	$\bar{T}_2$
T_{14}	1	1	1	0	$a + b$, OR, disjunction, INCLUSIVE–OR	$\bar{T}_1$
T_{15}	1	1	1	1	one function (closed, true)	$\bar{T}_0$

Table 6: All functions of 2 variables.

The most important functions are AND, NAND, OR, NOR and EXCLUSIVE–OR. Until now we have been using complete circuits. The symbols which are normally used are illustrated below:



We see that we can use a small circle for representing a negation. An inverter which generates the negation of the input signal is often symbolised by  or .

The above basic functions, and many more, are available in *hardware.* These are integrated circuits in plastic or ceramic packages: chips. For example, one can buy a chip with 14 pins containing 4 NAND gates (14 pins because 4x3=12 for the NANDs plus two for the power supply: ground and e.g. +5V).

5. Series developments

5.1 Disjunctive series development

THEOREM
Every Boolean function can be developed in a sum of products.

PROOF
By means of the first distributive property, all parentheses of the type $(a+b)(a+c) = a+bc$ can be removed [b and c may be complex functions].
By means of the second distributive property, all parentheses of the type $a.(b+c) = ab+ac$ can be removed.
Repeated application leads to a sum of products. This is called a **disjunctive** development.
Example: $abc + a\bar{c}d + \bar{b}d$

THEOREM
Any disjunctive form can be written as a sum of products in which in each product all variables occur.

PROOF
If in a form x the variable p is missing, then $x.1 = x.(p + \bar{p}) = xp + x\bar{p}$. So p has been added.
Example: suppose $\alpha = abc + acd + bd$
$abc = abc1 = abc(d + \bar{d}) = abcd + abc\bar{d}$
$acd = acd1 = acd(b + \bar{b}) = abcd + a\bar{b}cd$
$bd = bd1 = bd(ac + a\bar{c} + \bar{a}c + \bar{a}\bar{c}) = abcd + ab\bar{c}d + \bar{a}bcd + \bar{a}b\bar{c}d$
Hence, $\alpha = abcd + abc\bar{d} + ab\bar{c}d + a\bar{b}cd + \bar{a}bcd + \bar{a}b\bar{c}d$.
This form is called the **disjunctive normal form**.

If we write a function as a sum of products, this can be developed as follows:

$$T(a, b, c, \cdots) = (a + \bar{a})T = a\phi_1(b, c, \cdots) + \bar{a}\phi_2(b, c, \cdots)$$

ϕ_1 is a sum of products where a has been brought outside the parentheses.
ϕ_2 similar for $\bar{a}$.
Suppose $a = 1$, (i.e. $\bar{a} = 0$) and substitute this in $T(a, b, c, \cdots)$.

$$T(1, b, c, \cdots) = \phi_1(b, c, \cdots) \qquad \phi_1 \quad \text{is determined}$$

Now suppose $a = 0$, (i.e. $\bar{a} = 1$):

$$T(0, b, c, \cdots) = \phi_2(b, c, \cdots) \qquad \phi_2 \quad \text{is determined}$$

Hence:

$$T(a, b, c, \cdots) = aT(1, b, c, \cdots) + \bar{a}T(0, b, c, \cdots)$$

Repeated application gives

$$T(a,b,c,\cdots) = abT(1,1,c,\cdots) + a\bar{b}T(1,0,c,\cdots) + \bar{a}bT(0,1,c,\cdots) + \bar{a}\bar{b}T(0,0,c,\cdots)$$

If there are n variables a,b,c,...,n, the n-th variable can be assigned the weight $2^0 = 1$ and the variable a the weight 2^{n-1}. We can now write

$$T(a,b,\cdots,n) = \sum_{i=0}^{2^n-1} g_i m_i$$

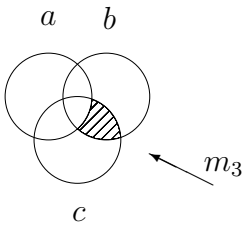
For three variables is $m_2 = \bar{a}b\bar{c}$: a has weight 4, b has weight 2 and c has weight 1. Inversion implies that a variable does not contribute to the sum. So $m_2 = \bar{a}b\bar{c} = 0.2^2 + 1.2^1 + 0.2^0 = 2$.

• The terms g_i denote a $T(1,0,1,\cdots)$. So is $g_2 = T(0,1,0)$, i.e. the value (0 or 1) of the function $T(a,b,c)$ for $a = 0$, $b = 1$, $c = 0$.

a	b	c	
0	0	0	g_0
0	0	1	g_1
0	1	0	g_2
0	1	1	g_3
1	0	0	g_4
1	0	1	g_5
1	1	0	g_6
1	1	1	g_7

• The terms m_i are called **min terms** and they represent in a Venn diagram <u>one</u> intersection:

$m_0 = \bar{a}\bar{b}\bar{c}$
$m_1 = \bar{a}\bar{b}c$
$m_2 = \bar{a}b\bar{c}$
$m_3 = \bar{a}bc$
$m_4 = a\bar{b}\bar{c}$
$m_5 = a\bar{b}c$
$m_6 = ab\bar{c}$
$m_7 = abc$



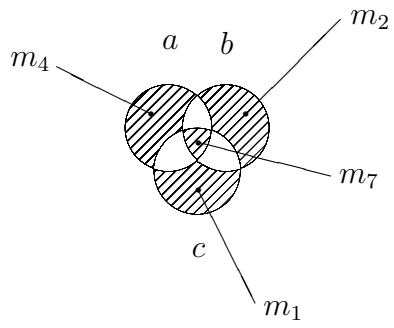
We see that $m_i.m_j = 0$ for $i \neq j$ and the min terms cover piece by piece the Venn diagram and represent a function.

• Relation between the function table and the series development:
The **inputs** of a function (truth) table correspond with the min terms.
The **outputs** are then the g terms of the series development.

Example:
$a \oplus (b \oplus c) =$
$a.\overline{(b \oplus c)} + \bar{a}.(b \oplus c) =$

$a(bc + \bar{b}\bar{c}) + \bar{a}(b\bar{c} + \bar{b}c) =$
$abc + a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c =$
$m_7 + m_4 + m_2 + m_1$.
This function is represented by the following Venn diagram and truth table:

a	b	c	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



5.2 Conjunctive series development

THEOREM
Every Boolean function can be developed in a product of sums.

PROOF
According to the distributive property, $a + bc = (a + b)(a + c)$. This can be done recursively. By adding e.g. $d\bar{d}$, the variable d can be introduced:
$(a + b)(a + c) + d\bar{d} = (d + (a + b)(a + c))(\bar{d} + (a + b)(a + c)) =$
$(d + a + b)(d + a + c)(\bar{d} + a + b)(\bar{d} + a + c)$.

Hence, any function can be written as a product of sums where in each sum all variables occur. This is called the **conjunctive normal form**.

Again, a and $\bar{a}$ can be brought outside parentheses (distributive property):

$$T(a, b, c) = [a + \psi_1(b, c)][\bar{a} + \psi_2(b, c)]$$

Suppose
$a = 0, \bar{a} = 1 \rightarrow T(0, b, c) = \psi_1(b, c)$ and
$a = 1, \bar{a} = 0 \rightarrow T(1, b, c) = \psi_2(b, c)$
such that $T(a, b, c) = [a + T(0, b, c)][\bar{a} + T(1, b, c)]$.
Repeated application for b and c gives:
$T(a, b, c) =$
$[a + b + c + T(0, 0, 0)] \cdot [a + b + \bar{c} + T(0, 0, 1)] \cdot$
$[a + \bar{b} + c + T(0, 1, 0)] \cdot [a + \bar{b} + \bar{c} + T(0, 1, 1)] \cdot$
$[\bar{a} + b + c + T(1, 0, 0)] \cdot [\bar{a} + b + \bar{c} + T(1, 0, 1)] \cdot$
$[\bar{a} + \bar{b} + c + T(1, 1, 0)] \cdot [\bar{a} + \bar{b} + \bar{c} + T(1, 1, 1)]$

In the same way as for a disjunctive series development we get

$$T(a, b, \cdots, n) = \prod_{i=0}^{2^n - 1} (G_i + M_i)$$

Alternative derivation:

$$T(a, b, \cdots, n) = \sum_{i=0}^{2^n-1} g_i m_i$$

$\bar{T}$ implies all $0 \to 1$ and all $1 \to 0$. So

$$\bar{T}(a, b, \cdots, n) = \sum_{i=0}^{2^n-1} \bar{g}_i m_i$$

According to de Morgan, $\overline{\sum \phi} = \prod \overline{\phi}$ and $\overline{\prod \phi} = \sum \overline{\phi}$. Hence

$$(\overline{T})\prime = T = \overline{\sum_{i=0}^{2^n-1} \bar{g}_i m_i} = \prod_{i=0}^{2^n-1} (g_i + \bar{m}_i).$$

$$\rightsquigarrow \quad \bar{m}_i = M_j \ \text{ with } \ j = 2^n - 1 - i.$$

- A term with $g_i = 1$ becomes 1 in the product (invisible).
- A term with $g_i = 0$ becomes M_j in the product (it exists!)
- The terms M_j are called **max terms** and they are the inverse of the min terms.

In the literature there exist two different notations:
1st: $a + b + c = M_7$
2nd: $a + b + c = M_0$
The 2nd takes over the notation for g_i, $[a + b + c + T(0, 0, 0)]$.
Here we use the 1st notation $a + b + c = M_7$, such that

$$T(a, b, c) = \prod_{i=0}^{2^n-1} (G_i + M_{2^n-1-i}).$$

Notation: instead of the complete min/max terms, only the subscripts are used

$$\sum m_1, m_3, m_7 \equiv \sum 1, 3, 7$$

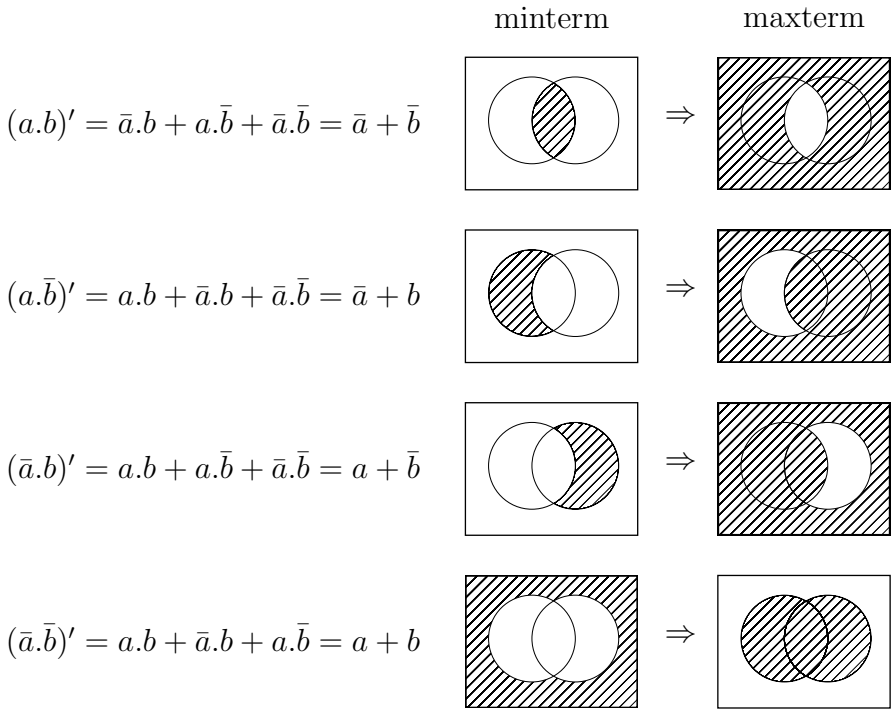
$$\prod M_1, M_3, M_7 \equiv \prod 1, 3, 7$$

Conversion:
To find the max terms from the min terms, we must determine the min terms for which $g_i = 0$ because for these the $M_j = M_{2^n-1-i}$ will exist.

Example:
We have 3 variables and $T(a, b, c) = \sum(1, 3, 7)$.
The missing min terms are 0, 2, 4, 5, 6. The max terms are $M_{7-0}, M_{7-2}, M_{7-4}, M_{7-5}, M_{7-6}$, or M_7, M_5, M_3, M_2, M_1
$T(a, b, c) = \sum(1, 3, 7) = \prod(1, 2, 3, 5, 7)$.

Verify: $\sum(0, 1, 3, 5) = \prod(0, 1, 3, 5)$

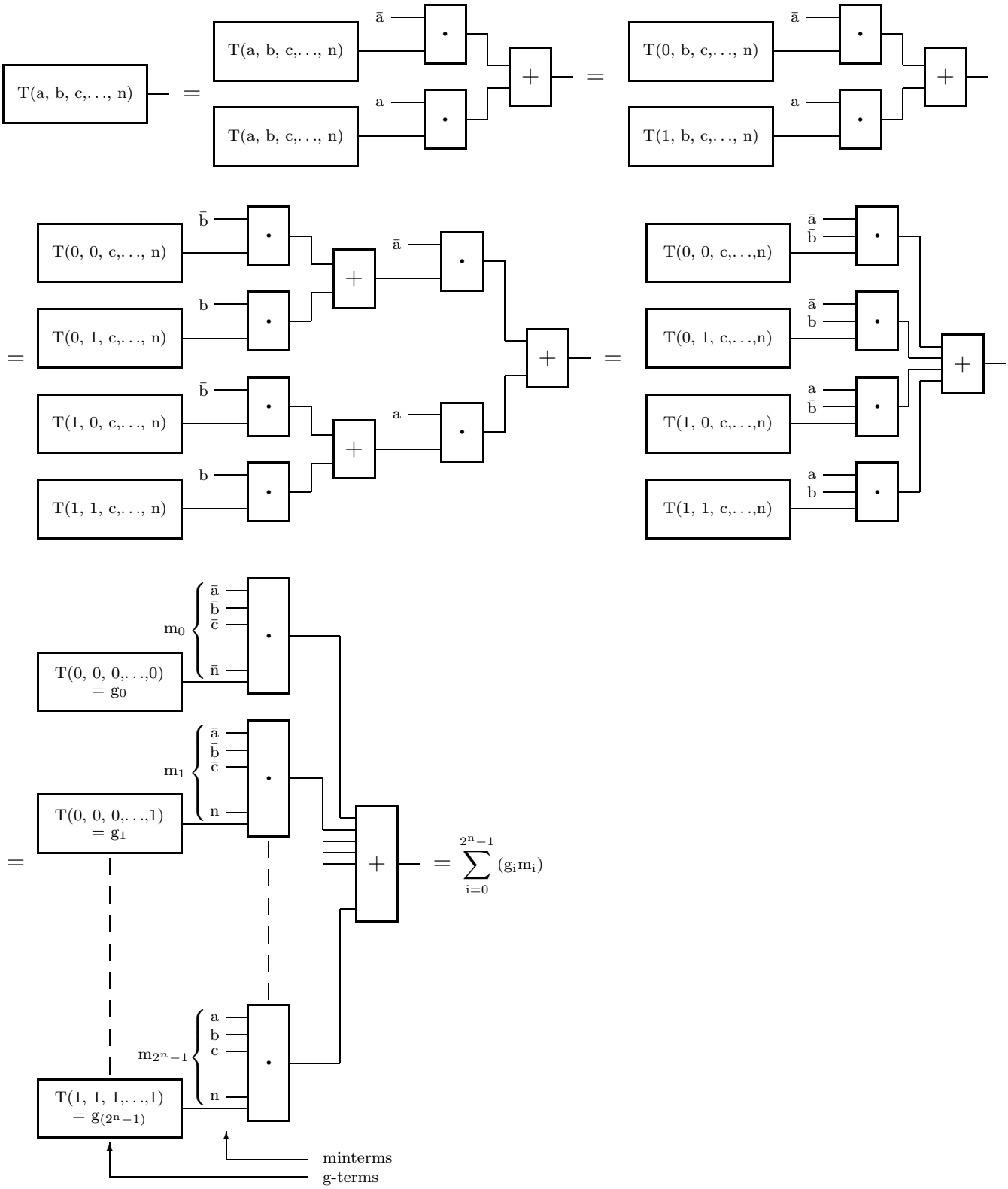
Summarising:
- The smallest sets (smallest areas in a Venn diagram) are min terms (min = minimal).
- The negation of the smallest gives the largest: max terms.



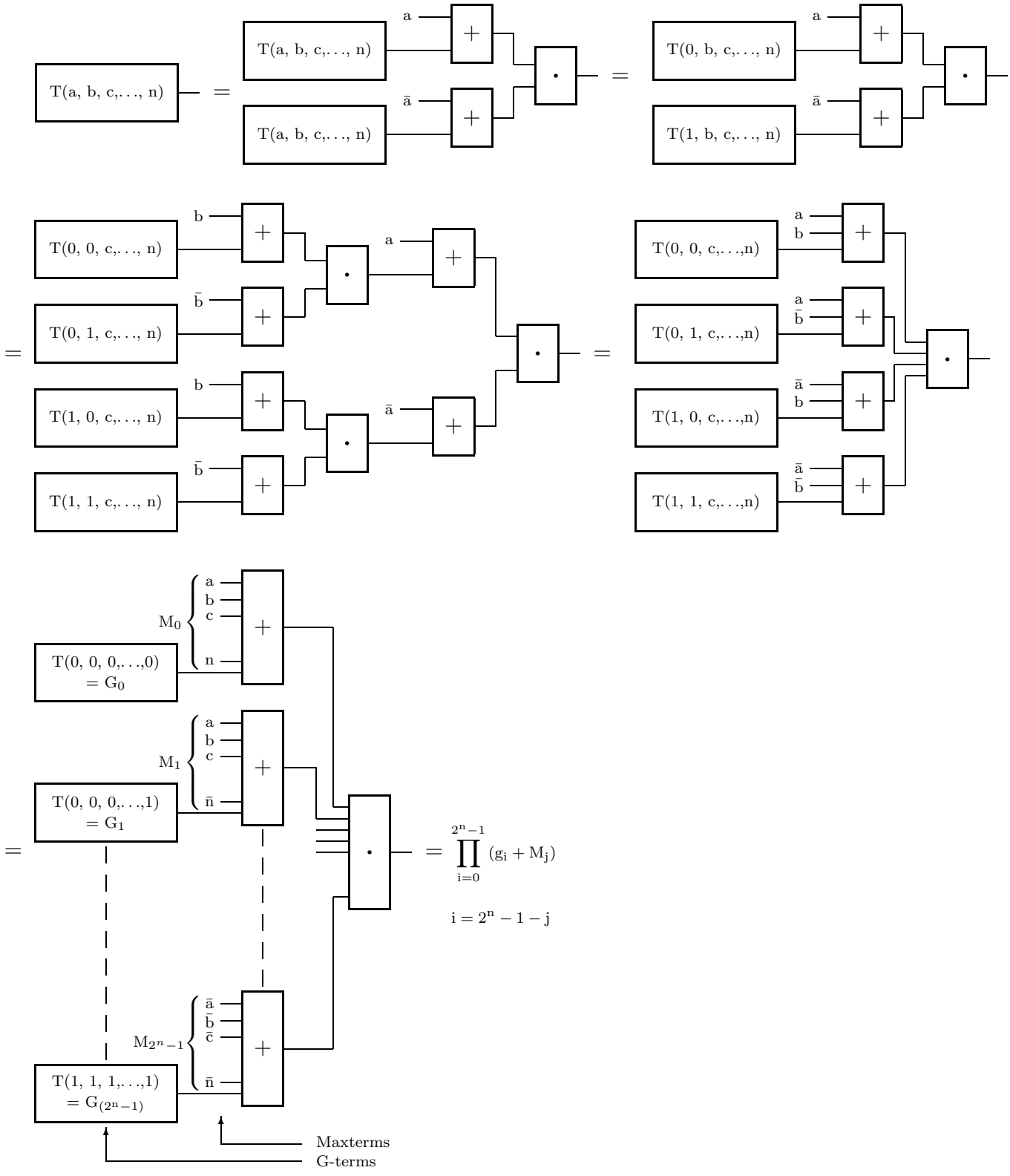
• Disjunctive series are mainly used to simplify circuits with AND, OR, and/or NAND gates.
• Conjunctive series are mainly used to simplify circuits with NOR gates.

Comparison/Summary:

Disjunctive	Conjunctive
$T(a, b, \cdots, n) = \sum_{i=0}^{2^n-1} g_i m_i$	$T(a, b, \cdots, n) = \prod_{i=0}^{2^n-1} (G_i + M_j)$ $j = 2^n - 1 - i$
A 0 in one of the g terms corresponds with a negation in the coupled m term: $g_3 = T(0, 1, 1) \Leftrightarrow m_3 = \bar{a}bc$	A 1 in one of the G terms corresponds with a negation in the coupled M term: $G_3 = T(0, 1, 1) \Leftrightarrow M_4 = a + \bar{b} + \bar{c}$
If the number of terms $g_i m_i$ with $g_i = 1$ equals k, the function can be written as the sum of k products.	If the number of terms $g_i m_i$ with $g_i = 1$ equals k, the function can be written as the product of $2^n - k$ factors.
For each combination of the variables, <u>at most</u> one of the k terms of the sum is equal to 1: $\Rightarrow m_p m_q = 0$ $(p \neq q)$.	For each combination of the variables, <u>at most</u> one of the $2^n - k$ factors of the product is equal to 0: $\Rightarrow M_p + M_q = 1$ $(p \neq q)$.



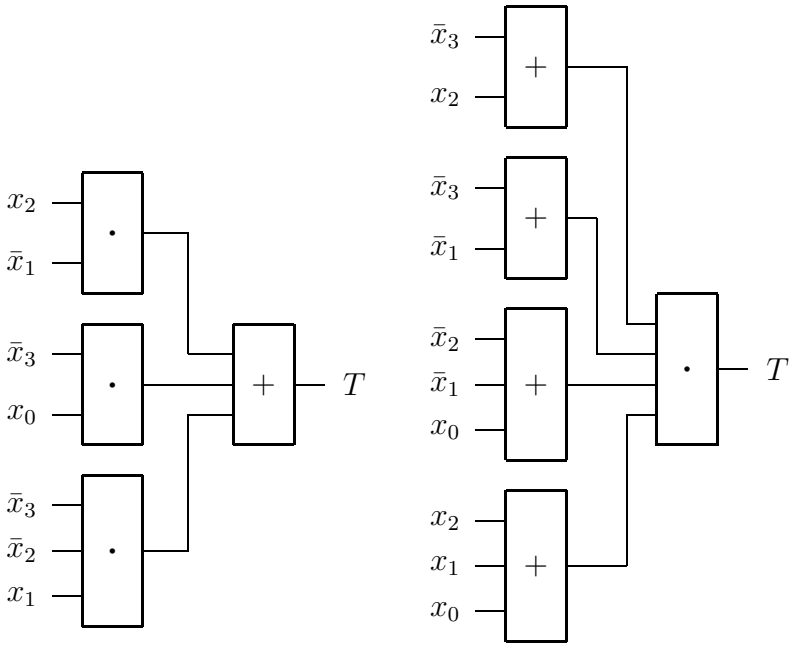
This scheme represents a disjunctive series development.



This scheme represents a conjunctive series development.

5.3 Conversion AND/OR to NAND/NOR circuits

Example to be used below:



This is the **regular form**: only ANDs and ORs:

$$T = \sum(1, 2, 3, 4, 5, 7, 12, 13) = x_2\bar{x}_1 + \bar{x}_3x_0 + \bar{x}_3\bar{x}_2x_1$$

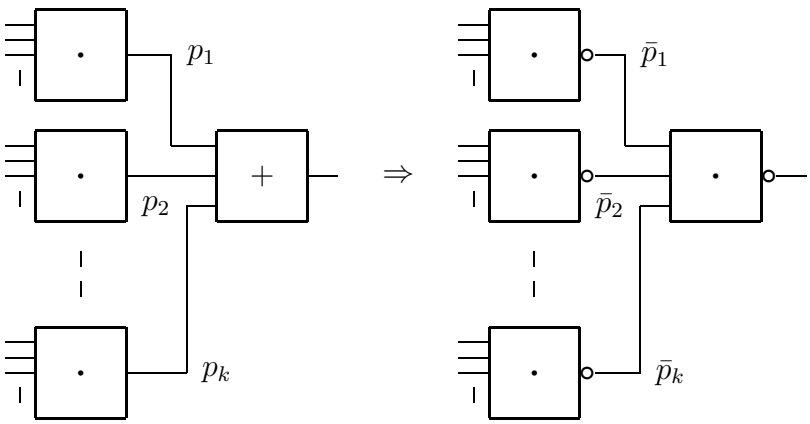
which equals

$$T = \prod(0, 1, 4, 5, 6, 7, 9, 15) = (\bar{x}_3 + x_2)(\bar{x}_3 + \bar{x}_1)(\bar{x}_2 + \bar{x}_1 + x_0)(x_2 + x_1 + x_0).$$

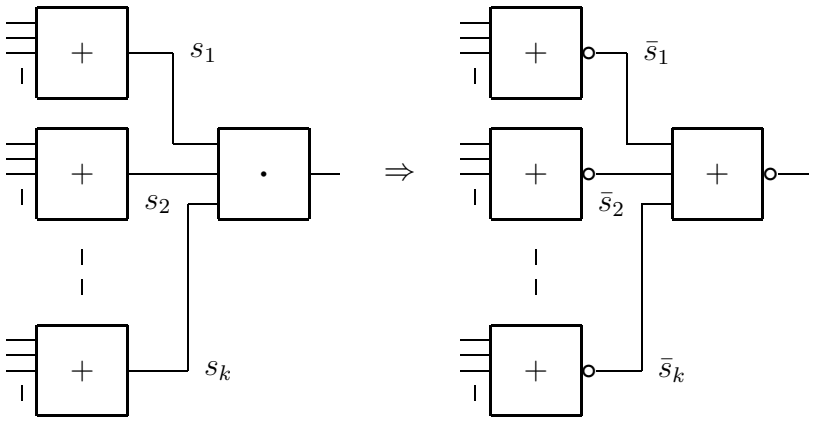
• A logical function T in disjunctive form can be realised with a 2–stage circuit consisting of NANDs only. Algebraically:

$$\sum g_i m_i = \overline{\prod \overline{g_i m_i}} = \overline{\prod(\bar{g}_i + \bar{m}_i)}$$

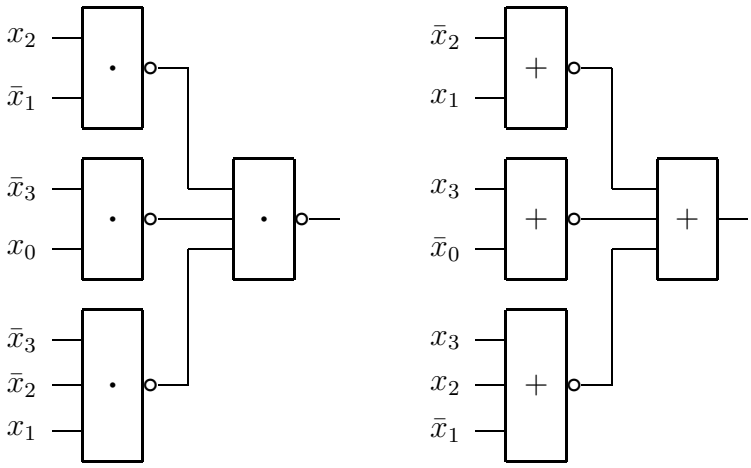
In the figure below $p_1 + p_2 + p_3 + \cdots p_k = \overline{\bar{p}_1\bar{p}_2\bar{p}_3 \cdots \bar{p}_k}$.



• A logical function T in conjunctive form can be realised with a 2–stage circuit consisting of NORs only:



The above example (top previous page) can be realised with 4 NAND gates (see below left), or with 5 NOR gates (try yourself). If this function must be realised with NOR gates, a more simple circuit is obtained by making $\bar{T}$ in conjunctive form and by using an OR instead of the last NOR:



In the left circuit we have

$$T = x_2\bar{x}_1 + \bar{x}_3x_0 + \bar{x}_3\bar{x}_2x_1$$

such that

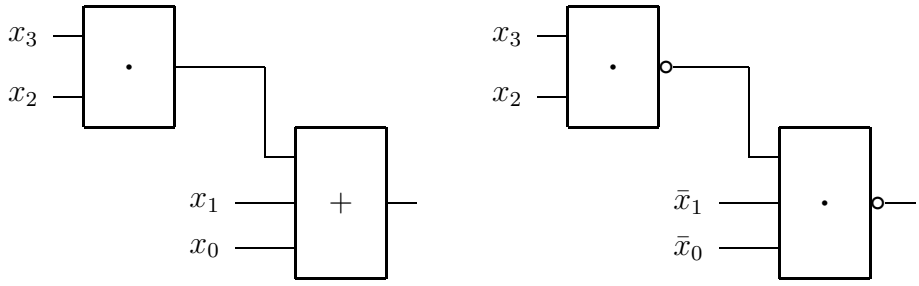
$$\bar{T} = (\bar{x}_2 + x_1)(x_3 + \bar{x}_0)(x_3 + x_2 + \bar{x}_1).$$

Taking the double negation of T, we obtain

$$T = \overline{(\bar{x}_2 + x_1)(x_3 + \bar{x}_0)(x_3 + x_2 + \bar{x}_1)},$$

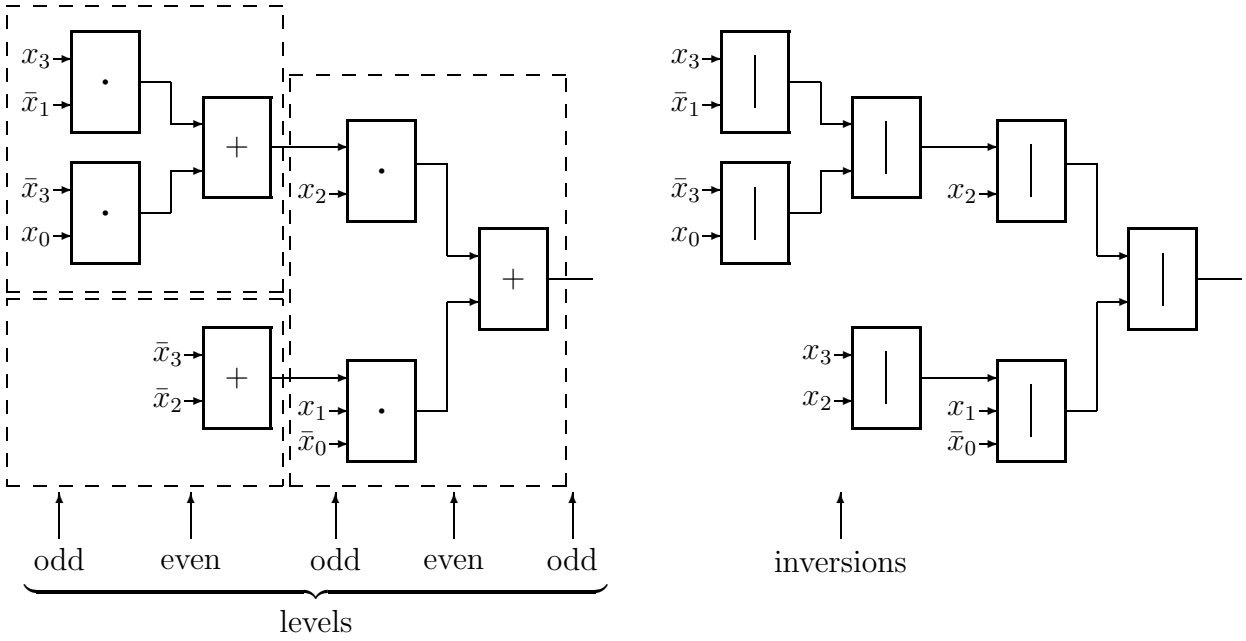
which corresponds to the right circuit.

• If one or more of the products in the disjunctive form consists of only one variable, then the corresponding AND gate is not necessary. Converting to NAND gates, this gate reduces to an inverter which can be omitted by taking the inverse of the variable (in practice: if the inverse variable is already available in a circuit):

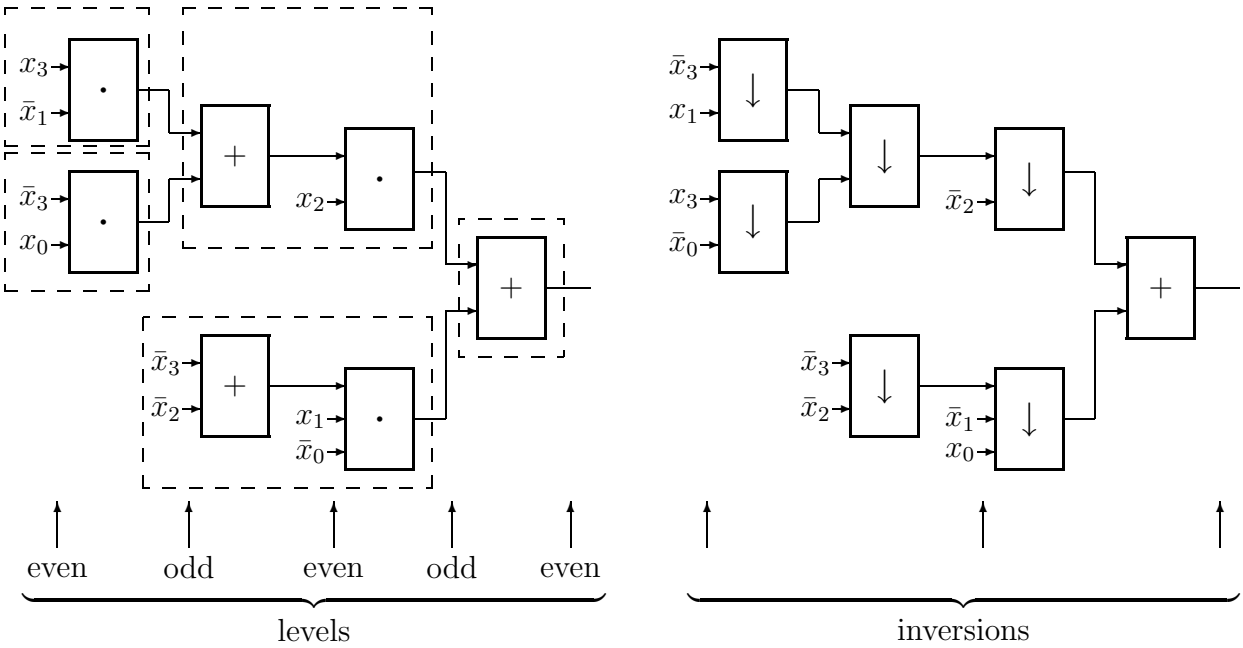


$$T = x_3x_2 + x_1 + x_0$$

• This inversion must also be applied in converting the conjunctive form to NOR gates, that is, if a *sum* consists of only one variable. See next page.



The output of the first circuit is given by $T = x_2(x_3\bar{x}_1 + \bar{x}_3x_0) + x_1\bar{x}_0(\bar{x}_3 + \bar{x}_2)$. Simplifying this expression, by increasing the "depth" of the circuit, we obtain an expression that allows to construct the circuit to the left. This circuit can be easily realised by using only NANDs, in which case the inputs of the even levels need to be inverted. It is also possible to realise the same function using only NORs and one OR. This is shown in the lower figure. Note the difference between the "numbering" of the levels (even and odd). *Check this trick!*



6. Simplification of logical functions

Logical functions may consist of many variables and may be quite complex. We may try to simplify functions on the basis of the postulates etc, but this is in many cases not an easy task. For this reason there exist systematic approaches, even CAD (computer aided design) programs. We will discuss only one method here. Arranging min (max) terms on a square lattice in a proper way, we obtain a so-called **Karnaugh diagram**:

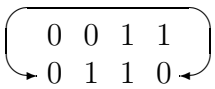
$\bar{c}$ $\Downarrow$ c

	$\bar{a}\bar{b}\bar{c}\bar{d}$ 0	$\bar{a}\bar{b}\bar{c}d$ 1	$\bar{a}\bar{b}cd$ 3	$\bar{a}\bar{b}c\bar{d}$ 2	$\bar{b}$
$\bar{a}$	$\bar{a}b\bar{c}\bar{d}$ 4	$\bar{a}b\bar{c}d$ 5	$\bar{a}bcd$ 7	$\bar{a}bc\bar{d}$ 6	$\Leftarrow$
$\Rightarrow$	$ab\bar{c}\bar{d}$ 12	$ab\bar{c}d$ 13	$abcd$ 15	$abc\bar{d}$ 14	b
a	$a\bar{b}\bar{c}\bar{d}$ 8	$a\bar{b}\bar{c}d$ 9	$a\bar{b}cd$ 11	$a\bar{b}c\bar{d}$ 10	$\Leftarrow$ $\bar{b}$

$\bar{d}$ $\Uparrow$ d $\Uparrow$ $\bar{d}$

We make the following observations:

- Going up, down, left, or right, only a single variable changes.
- Horizontally, the variables cd follow the values 00, 01, 11, 10.
- Vertically, the variables ab follow the values 00, 01, 11, 10.
- The single–variable change is also *cyclic* around the edges.



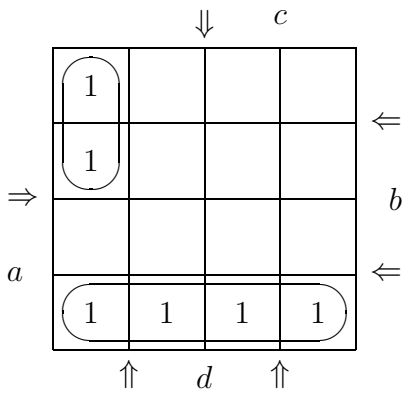
- We can consider the variables $abcd$, with a being the most significant bit, to represent a 16–counter $0, 1, 2, \cdots, 15$, therefore representing all the possible min terms.
- A function is represented by placing for each min term a 1 in the corresponding box.
- When two min terms occur in horizontally or vertically neighbouring boxes, a variable can be eliminated.

The Karnaugh diagram allows an easy simplification by visual inspection. By taking the **largest** blocks of 1s, most variables can be eliminated.

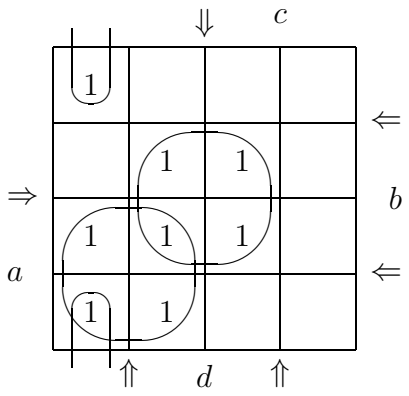
Example:
For the two blocks 5 and 7 in the centre we have $\bar{a}b\bar{c}d + \bar{a}bcd$. Only the c term differs, so $m_5 + m_7 = \bar{a}bd$. The term c has been eliminated.

Example:
$\sum 5, 7, 13, 15 = \bar{a}b\bar{c}d + \bar{a}bcd + ab\bar{c}d + abcd = bd$. The variables a and c are eliminated. Check this algebraically and in the Karnaugh diagram!

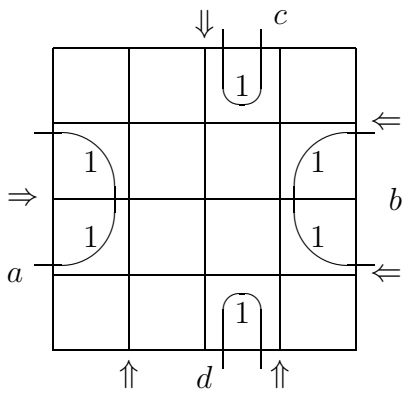
More examples:



$\sum 0, 4, 8, 9, 10, 11 = a\bar{b} + \bar{a}\bar{c}\bar{d}$.
A block of four 1s: 2 variables eliminated.
A block of two 1s: 1 variable eliminated.

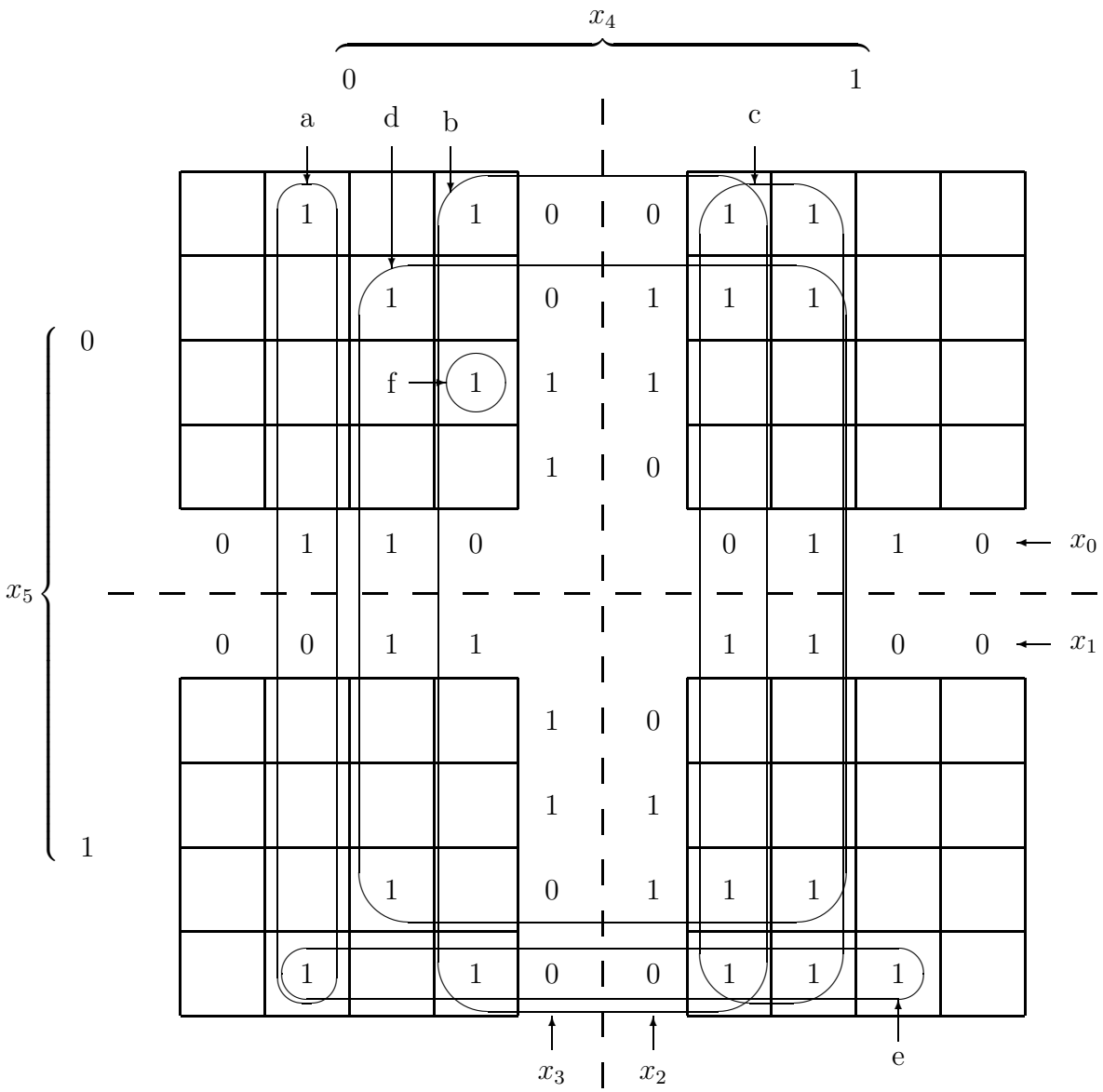


$\sum 0, 5, 7, 8, 9, 12, 13, 15 = a\bar{c} + bd + \bar{c}\bar{b}\bar{d}$.



$\sum 3, 4, 6, 11, 12, 14 = b\bar{d} + \bar{b}cd$.

As an example, a complex Karnaugh diagram with 6 variables is shown here.



After simplification, we get

$a = \bar{x}_4\bar{x}_3\bar{x}_2\bar{x}_1x_0$

$b = \bar{x}_3\bar{x}_2x_1\bar{x}_0$

$c = x_4\bar{x}_3x_1$

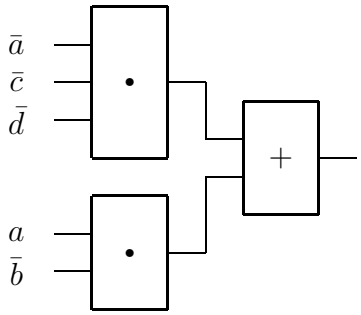
$d = \bar{x}_3x_2x_1x_0$

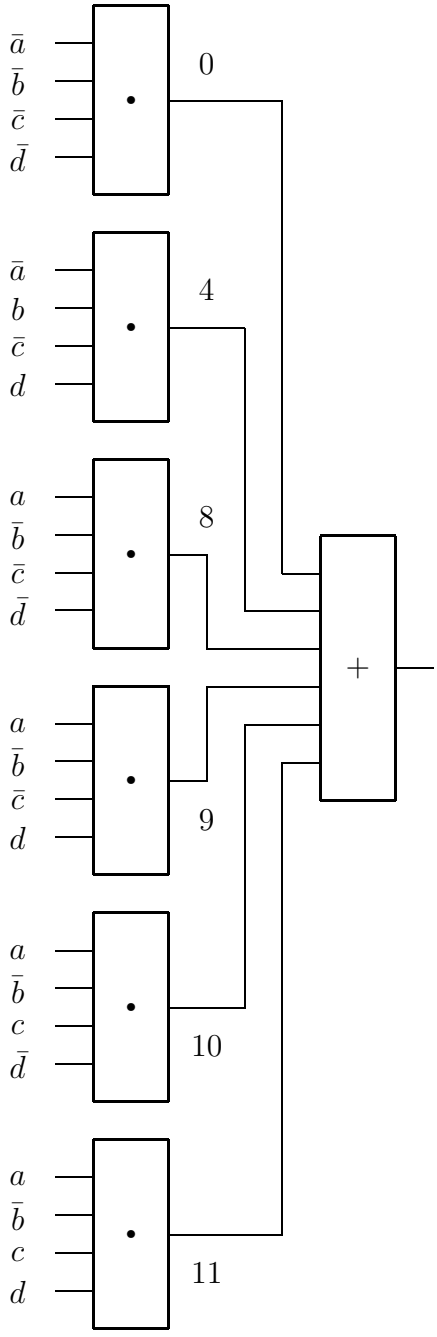
$e = x_5\bar{x}_3\bar{x}_2\bar{x}_1x_0$

$f = \bar{x}_5\bar{x}_4x_3x_2x_1\bar{x}_0$

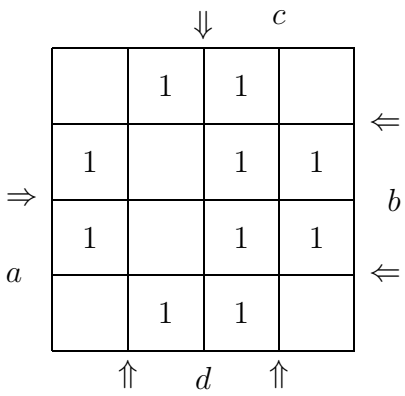
6.1 Technical realisations

The first example $(\sum 0, 4, 8, 9, 10, 11)$ requires many elements, namely 6 ANDs with 4 inputs each, i.e. 24 inputs, plus one OR with 6 inputs. Total: 7 gates and 30 inputs. The equivalent function $a\bar{b} + \bar{a}\bar{c}\bar{d}$ requires only three gates and 7 inputs. These circuits are shown to the right and below.



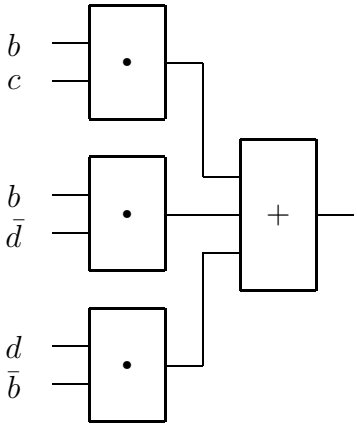


Let us use another example: $\sum(1, 3, 4, 6, 7, 9, 11, 12, 14, 15)$. Using the Karnaugh diagram

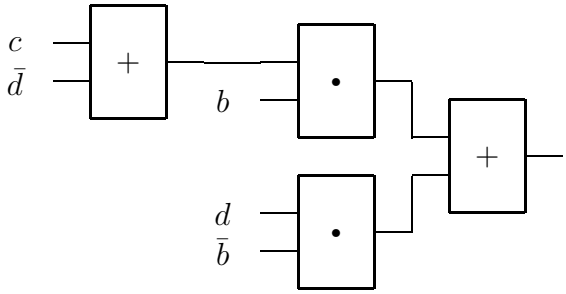


we can simplify this function to $bc + b\bar{d} + d\bar{b}$. This function can be realised in different ways.

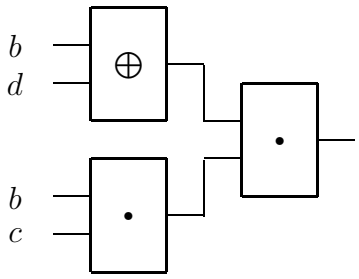
1) Straightforward, using two stages with 4 gates and 9 inputs:



2) By means of $b(c+\bar{d})+d\bar{b}$, which requires also 4 gates but only 8 inputs. However, this is a three-stage circuit. Because each stage has a certain delay, for example 10 ns, it is a little bit slower than the first solution. We speak of *ripple-through:* a change of c first changes the OR output, then the AND, and finally the output OR. This means 30ns in total, which may be too much in time-critical applications.



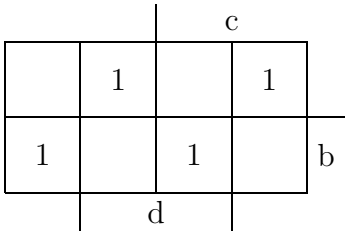
3) We can write $bc + b\bar{d} + d\bar{b} = bc + (b \oplus d)$. This solution requires only 3 gates with 6 inputs. However, now we need an EXCL-OR, which costs an extra chip if there is not one left over from a chip for some other circuit. One of the principles of hardware design is the optimisation of the use of available space on a printed circuit board by minimising the number of chips!



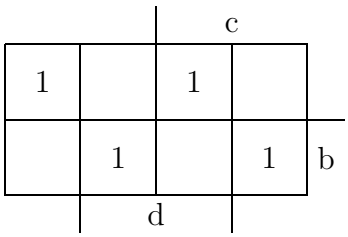
• An exclusive-OR combination is evident because of the ⊃⊂ structure in the Karnaugh diagram:

	1	1	
1			1

In general, an exclusive-OR can be recognised by checkerboard patterns:



$b \oplus c \oplus d$



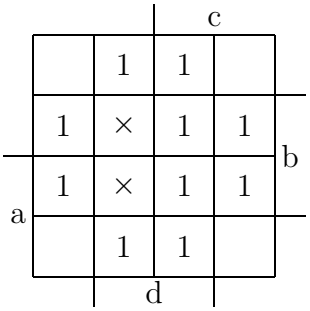
$$\begin{aligned}(b \oplus c \oplus d)' &= b \oplus c \oplus \bar{d} \\ &= \bar{b} \oplus c \oplus d \\ &= b \oplus \bar{c} \oplus d\end{aligned}$$

If we need to use an exclusive-OR gate with 4 inputs, for instance because it is available on a printed circuit board, the last example becomes $b \oplus c \oplus d \oplus 1$. Verify this!

6.2 Don't cares

With the definition of a logical function, all combinations of the input variables are assigned the two values *true* or *false* (1 or 0). However, it is possible that certain combinations can never occur, depending on other information or conditions. So one could assume arbitrarily a 0 or a 1 for these. This can lead to further simplifications! We speak of **don't cares**.

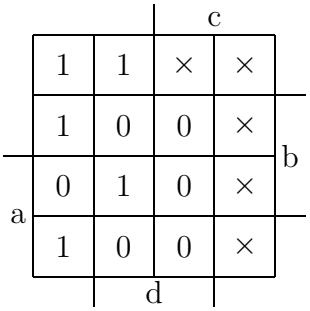
Example: $T = \sum(1, 3, 4, 6, 7, 9, 11, 12, 14, 15, \underline{5}, \underline{13})$.
This is the last example of the previous section, but extended with the don't cares 5 and 13 (we normally underline the minterms and represent the don't cares in the Karnaugh diagram with a × symbol:



If we assume that the don't cares are both 1,
we can take two blocks of 8 ones.
Hence $T = b + d$.

For the development of conjunctive forms, one can make use of $\bar{T}$ and invert the result. This implies collecting all 0s in the Karnaugh diagram: $\bar{T} = \bar{b}\bar{d} \rightsquigarrow T = b + d$. This example is not very illustrative. Let us take another

Example: $T = \sum(0, 1, \underline{2}, \underline{3}, 4, \underline{6}, 8, \underline{10}, 13, \underline{14})$.



$\bar{T} = c + da\bar{b} + ab\bar{d} + \bar{a}db$
$T_{\text{conj}} = \bar{c}.(\bar{d} + \bar{a} + b).(\bar{a} + \bar{b} + d).(a + \bar{d} + \bar{b})$
$T_{\text{disj}} = \bar{b}\bar{a} + \bar{b}\bar{d} + \bar{d}\bar{a} + ab\bar{c}d$

The conjunctive form requires 3 ORs and 1 AND with 13 inputs, whereas the disjunctive form requires 1 OR and 4 ANDs with 14 inputs. However, the disjunctive form is a little bit simpler because of three ANDs having only two inputs. This function can be further simplified by increasing the depth to three levels and employing one exclusive-OR:

$$T_{\text{conj}} = \bar{c}.[\bar{a} + (\bar{b} + d).(b + \bar{d})].(a + \bar{d} + \bar{b}).$$

This form requires only 4 elements with 9 inputs. *Check this!*

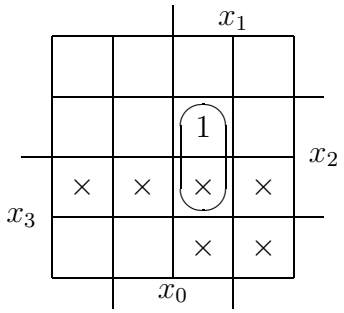
6.3 Application: code conversion

We will illustrate the application of don't cares by looking at the conversion between different codes.

6.3.1 BCD to DECIMAL

In this case we have 4 input lines x_3, x_2, x_1 and x_0, but 10 output lines labelled 0, 1, 2, ..., 9. The BDC codes go from 0000, 0001, 0010, to 1001, but the following binary combinations cannot occur. So we can use these as don't cares in the Karnaugh diagram:

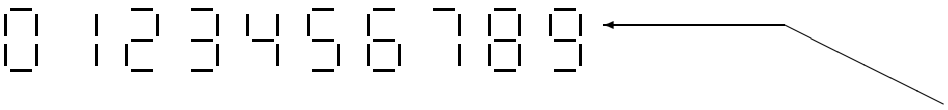
x_3	x_2	x_1	x_0	DEC
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	×
.	.	.	.	×
1	1	1	1	×



Example: $T_7 = x_2x_1x_0$.
It may be a good idea to try one other output yourself.

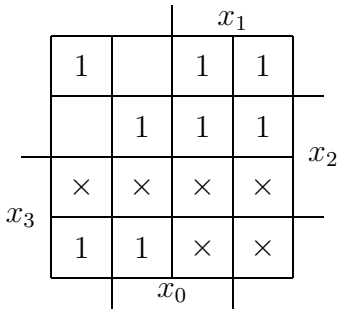
6.3.2 BCD to 7-SEGMENT

Some displays use seven line segments to show the numbers 0 to 9:



We must now make a table for each segment. We'll take the upper horizontal one:

x_3	x_2	x_1	x_0	$-$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	×
.	.	.	.	×
1	1	1	1	×



We can recognise two blocks of 8 ones (x_3 and x_1), plus two blocks of 4 ones (in the centre and the four corners).

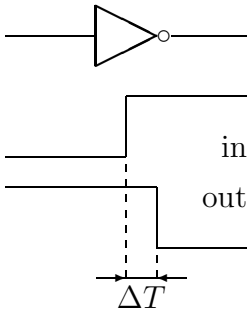
Hence:

$$\text{top} -= \bar{b}\bar{d} + a + c + bd = (b \oplus d)' + a + c$$

This can be done for all segments. *Try one!*

6.4 Hazards in combinatorial circuits

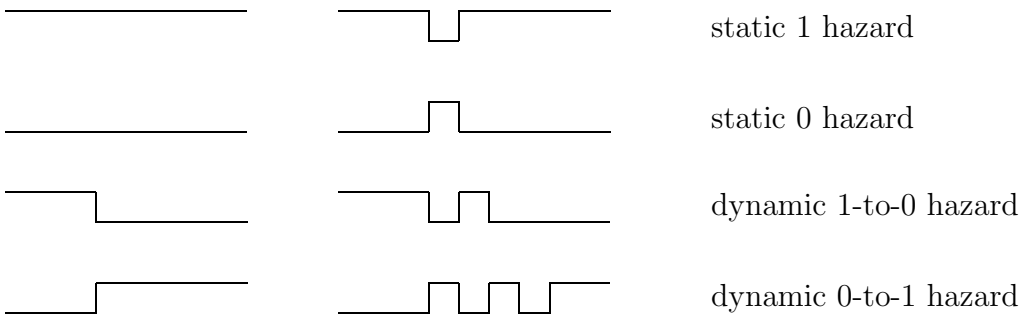
A **hazard** is a momentary error (a spurious signal or glitch) that may occur in an output of a combinatorial circuit, i.e. a circuit that consists only of normal gates without feedback loops. The circuit works correctly according to its specification, but due to different delay times some different output signals may occur before all outputs reach a steady state:



Taking an inverter (but the same holds for all different gates), if the input changes from 0 to 1, the output will go from 1 to 0, but with a delay of ΔT, normally around xx ns.

The delay ΔT may be about equal for different gates, but not exactly equal. Moreover, there may exist different paths with a different number of gates, in which case timing differences of $n\Delta T$ may occur. If a combinatorial circuit drives a sequential circuit which consists of flip-flops with edge- or transition-sensitive inputs, hazards may cause serious problems.

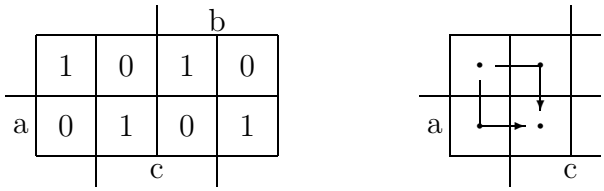
There are different hazard types. At the left we see stable signals or transitions without hazards and at the right with hazards:



In the case of a static 1 hazard, the signal must remain 1 but shows a small 0 glitch. In the case of a dynamic 1-to-0 hazard, the signal should go from 1 to zero, but it shows one logic 1 glitch. In the last case the signal should go from 0 to 1, but shows two logic 0 glitches. If a circuit is more complicated, the number of glitches can be bigger.

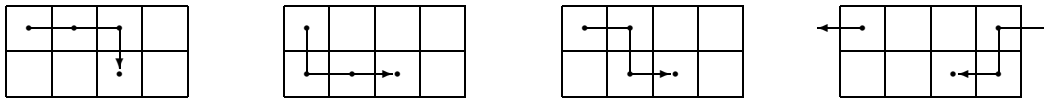
Now, how can hazards be detected and eliminated? They can be traced in a Karnaugh diagram when two or more input signals change and the function change cannot be covered by function combinations in the diagram. These are called **function hazards**.

Example:



If $b = 0$ and ac changes from 00 to 11, then depending on what input changes first, a or c, two different paths are followed which both include an output value 0, so the output goes from 1 via 0 to 1: ⊓⊔ This is a static 1 function hazard. *Check* $b = 0$ and $ac = 10 \Rightarrow ac = 01$.

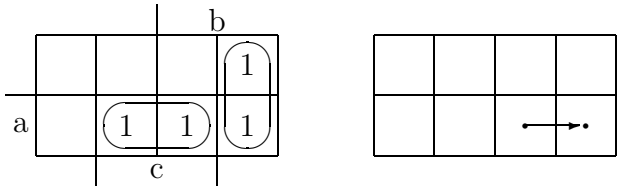
Taking the same Karnaugh diagram, but going from $abc = 000$ to $abc = 111$, we get in all cases ⊓⊔⊓ This is a dynamic 1-to-0 function hazard. Four different paths are shown in the diagrams below. *Check* $abc = 100 \Rightarrow 011$.



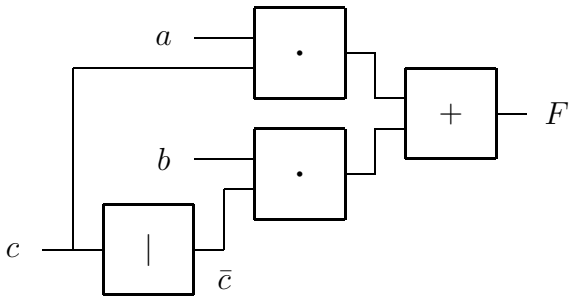
Function hazards in a combinatorial circuit cannot be controlled or avoided, unless exact timings of input signals and gate delays are respected. However, this is in practice impossible. In such cases we need to apply synchronous sequential circuits and flip-flops, which will be treated later on in Chapters 7 and 8.

A **logic hazard** is due to different delays when only one input signal changes. Logic hazards can be eliminated!

Example:



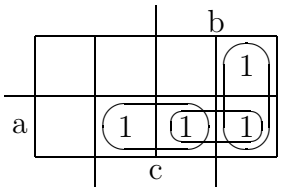
In this case we get the function $F = a.c + b.\bar{c}$. If $ab = 11$ and c changes from 1 to 0, as indicated by the arrow, we do not expect any problem because at both places we have a 1 without passing a 0. But in practice we can obtain ⊓⊔, which is a static 1 function hazard. In order to explain this, we need to look at the circuit:



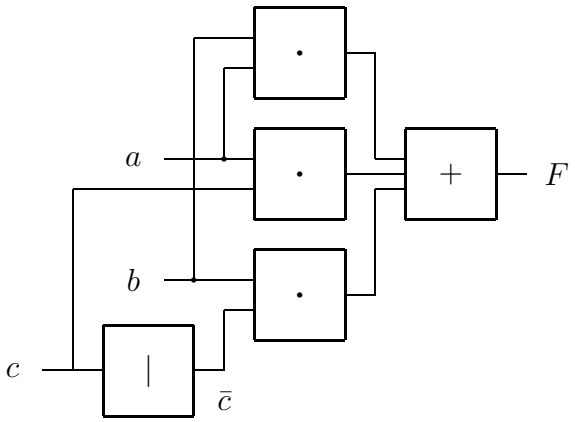
The delay of the inverter (the gate with the | symbol) causes different input combinations at the inputs of the two AND gates:

c	$\bar{c}$	
1	0	steady situation, $\bar{c} = \neg c$
0	0	input changed but output still 0, $\bar{c} \neq \neg c$
0	1	steady situation again, $\bar{c} = \neg c$

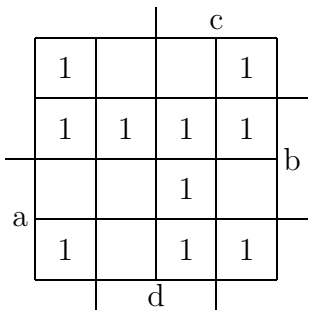
Solution: include an extra min-term that connects the subsets in the Karnaugh diagram:

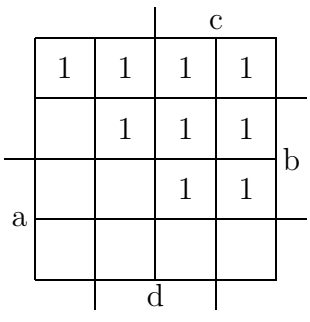


We obtain the function $F = a.c + b.\bar{c} + ab$.
Since a and b do not change, only c changes, the term ab keeps F at the level 1 when c changes. The drawback is that more gates are required, see the circuit below.



Below are two other examples. Do you know how many and which redundant terms are required to avoid hazards?





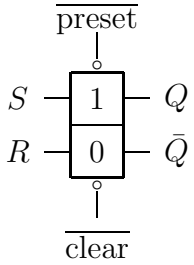
7. Memory elements

A combinatorial circuit consisting of logical functions (gates) without feedback loops, i.e. connections from outputs to previous inputs, cannot remember what happened before: the outputs depend only on the actual values of the inputs. In many cases we want a circuit to remember, for example if we want to make a counter that goes from state 3 to state 4 if we supply an input pulse. In such cases we need memory elements, which are normally called **flip-flops**.

A flip-flop is a *bistable* element: it has only two states called *on* (or 1) and *off* (or 0). Depending on the inputs, the flip-flop will remain in its old state, or it will change: $0 \rightarrow 1$ or $1 \rightarrow 0$. There are no other possibilities, although in many cases additional inputs like *preset* and *clear* are provided in order to initialise some or all flip-flops in a circuit. We will treat the most common flip-flop types below.

7.1 SR flip-flop

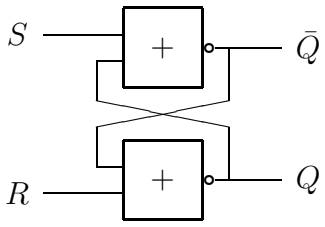
The SR or **Set-Reset** flip-flop has two inputs S and R and two outputs Q and $\bar{Q}$, sometimes also a $\overline{\text{preset}}$ and a $\overline{\text{clear}}$. A $\overline{\text{clear}}$ is a so-called *active-low* input: a 0 implies that the flip-flop goes to state 0, independent of the signal values on the S and R inputs. A 0 on the $\overline{\text{preset}}$ implies that the flip-flop goes to state 1. Below left is the symbol, below right the function table.



S	R	Q_{new}	
0	0	Q_{old}	stable
0	1	0	reset
1	0	1	set
1	1		illegal

In the table we can see that the SR combination 00 does not lead to a change: $Q_{\text{new}} = Q_{\text{old}}$. The combination 01 will always reset the flip-flop (if it was off it remains off, if it was on it will go to off) with the output $Q = 0$ and the output $\bar{Q} = 1$. The combination 10 will always set the flip-flop ($Q = 1$ and $\bar{Q} = 0$). Finally, the combination 11 is illegal. Why this is we will see by means of realisations.

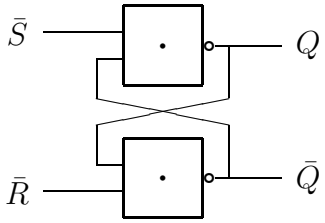
SR flip-flop with NORs



S	R	Q_{new}	$\bar{Q}_{\text{new}}$	
0	0	Q_{old}	$\bar{Q}_{\text{old}}$	stable
0	1	0	1	reset
1	0	1	0	set
1	1	0	0	illegal

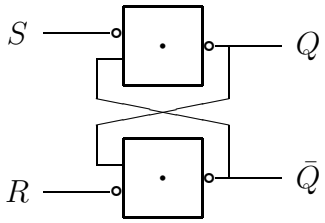
Now we can see what happens with the SR combination 11: each NOR will have a 0 output and therefore $\bar{Q}$ will not be the inverse of Q. You should familiarise yourself with this circuit, assuming values for Q and $\bar{Q}$ as well as S and R, following the connections.

SR flip-flop with NANDs



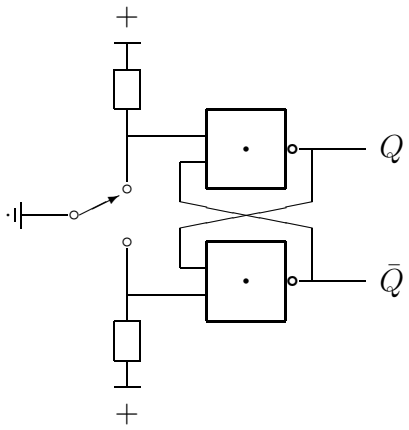
$\bar{S}$	$\bar{R}$	Q_{new}	$\bar{Q}_{\text{new}}$	
0	0	1	1	illegal
0	1	1	0	set
1	0	0	1	reset
1	1	Q_{old}	$\bar{Q}_{\text{old}}$	stable

Because a NAND always follows an input with level 0, we have to assume the *active-low* inputs $\bar{S}$ and $\bar{R}$. You should compare this circuit carefully with the NOR circuit! We can obtain the same functionality and table with NANDs if we introduce inverted inputs:



S	R	Q_{new}	$\bar{Q}_{\text{new}}$	
0	0	Q_{old}	$\bar{Q}_{\text{old}}$	stable
0	1	0	1	reset
1	0	1	0	set
1	1	1	1	illegal

This type of flip-flop is also called a *latch,* and can be used to *debounce* a switch: a switch between 0 and e.g. +5V always shows noise and this can cause problems in digital circuits. A solution is to apply a latch which provides clean Q and $\bar{Q}$ signals:

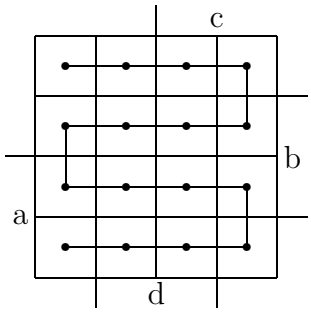


7.2 Application SR flip-flop: Gray code counter

Suppose we have a pulsed signal, p = 0 1 0 1 0 1 etc, periodic or not, and that we want to make a counter which counts 0 1 2 3 ... 15 0 1 2 ... etc. But instead of counting in binary code, we want to count in the Gray or mirror code. Counting from 0 to 15 means 16 steps, so we need 4 flip-flops ($2^4 = 16$). Call the flip-flops A, B, C, and D. They have outputs a, b, c, and d, as well as the inverted ones $\overline{\text{a}}$ etc. So in total we have 5 signals p, a, b, c, and d, and the question is: what functions are required for the S and R inputs of the flip-flops to count in the Gray code? From now on we will sometimes write FF instead of flip-flop. Let us first recall the Gray code:

step	a	b	c	d	p
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	1	0
3	0	0	1	0	1
4	0	1	1	0	0
5	0	1	1	1	1
6	0	1	0	1	0
7	0	1	0	0	1
8	1	1	0	0	0
9	1	1	0	1	1
10	1	1	1	1	0
11	1	1	1	0	1
12	1	0	1	0	0
13	1	0	1	1	1
14	1	0	0	1	0
15	1	0	0	0	1

Note: the Gray code is often used in position measurements because only 1 bit changes (when the binary code goes from 0111 to 1000 all 4 bits change). Contrary to the binary code, the Gray code follows a continuous path in the Karnaugh diagram:

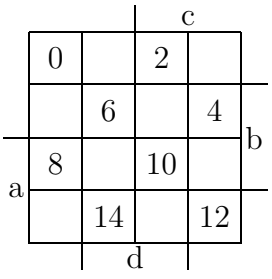


There are many codes which change only 1 bit at a time; just select a continuous path in the Karnaugh diagram!

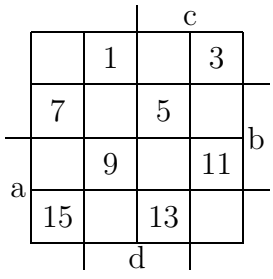
In step (or state) 0 the FFs for state 1 must be set and must remain. We must keep in mind that gates and FFs have a certain delay between inputs and outputs. It takes some time for a circuit to react, so there will be brief unstable states before the stable ones will be reached. This is shown in the table below, which is a copy of the top of the table above. Note that S means stable and U means unstable.

state	a	b	c	d	p	
0	0	0	0	0	0	S
	0	0	0	0	1	U
1	0	0	0	1	1	S
	0	0	0	1	0	U
2	0	0	1	1	0	S
	0	0	1	1	1	U
3	0	0	1	0	1	S
	0	0	1	0	0	U
4	0	1	1	0	0	S

We see that each abcd code has a stable and an unstable state; the last is caused by a transition in p, but the circuit did not have the time to react and to go directly to the stable new state. We can now make diagrams which show the stable and unstable states for p=0 and p=1. First the stable states:

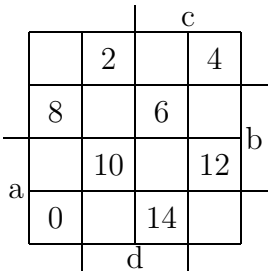


p=0

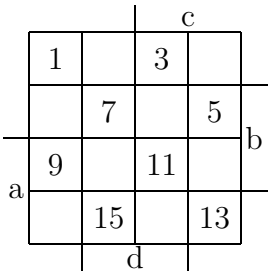


p=1

and now the same for the unstable states:

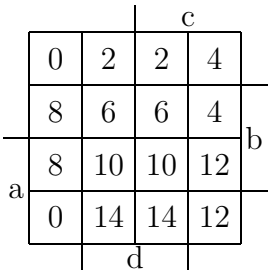


p=0

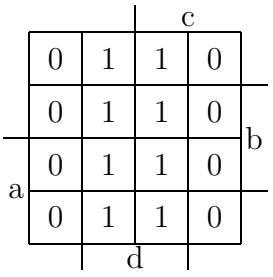


p=1

We can now combine the stable and unstable states for p=0 (left diagram below) and then fill the value that a FF should have in the table of the Gray code, for example FF D (right diagram):

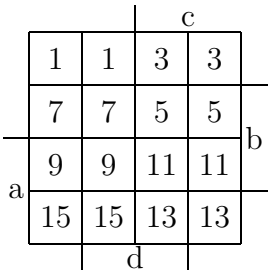


p=0

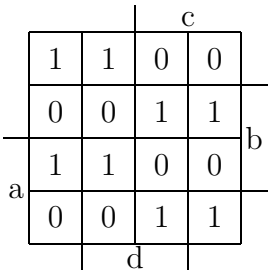


FF D for p=0

In the same way can we make the diagrams for p=1 and FF D:

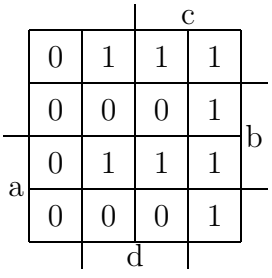


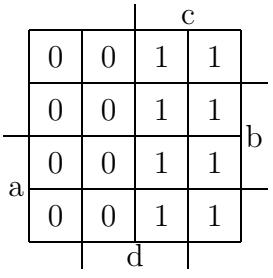
p=1



FF D for p=1

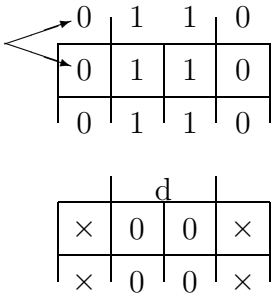
On the basis of the left diagrams (**p=0** and **p=1**) we can also fill the diagrams for the other FFs, for example FF C:





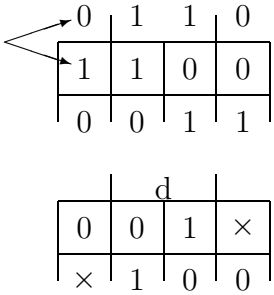
FF C for p=0 **FF C for p=1**

Until now we have seen how to make the Karnaugh diagrams for p=0 and 1 for the FFs. But now we should take look at how to generate the set and reset functions for the FFs. Let's take the diagram for p=0 and FF D. The top part is shown below left. Now we are going to compare (see the arrows) the values of D according to the position in the diagram (on top) and the actual values in the diagram. This comparison allows us to make a new diagram, one for generating the reset for FF D and p=0 which is shown below the diagram with the arrows:



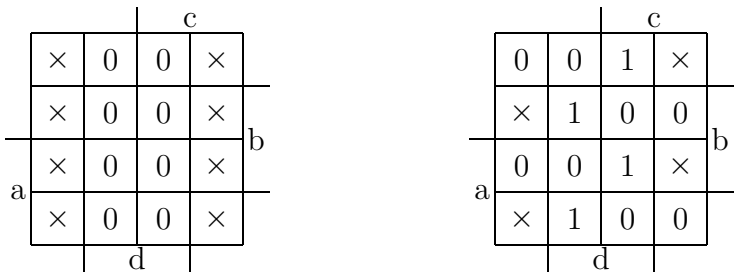
For reset D and p=0:
• If d=0 (on top) and d should be 0 (in diagram), the circuit may generate a reset, but it is not required because d is already 0. Conclusion: we can fill a don't care.
• If d=1 (on top) and d should be 1 (in diagram), the circuit may certainly NOT generate a reset. We need to fill a 0.

In the same way we can make a Karnaugh diagram for reset D and p=1:



For reset D and p=1:
• If d=0 and should be 1: not a reset.
• If d=1 and should be 1: not a reset.
• If d=1 and should be 0: a reset!
• If d=0 and should be 0: a dont't care.

As a result, we obtain the following two Karnaugh diagrams:



Reset D for p=0 **Reset D for p=1**

We see that the reset FF D for p=0 is zero because we can replace all don't cares by 0s. The reset FF D for p=1 equals $(\bar{b}c\bar{a} + b\bar{c}\bar{a} + bca + \bar{b}\bar{c}a) \cdot p = (a \oplus b \oplus c) \cdot p$.

We can do the same for the set of FF D. We then get the following two Karnaugh diagrams:

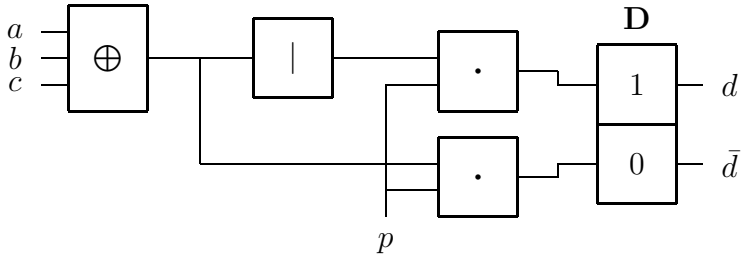
a \ b	cd=00	cd=01	cd=11	cd=10
00	0	×	×	0
01	0	×	×	0
11	0	×	×	0
10	0	×	×	0

Set D for p=0

a \ b	cd=00	cd=01	cd=11	cd=10
00	1	×	0	0
01	0	0	×	1
11	1	×	0	0
10	0	0	×	1

Set D for p=1

Again, the set D for p=0 is zero because we can replace all don't cares by 0s. The set D for p=1 equals $(\bar{b}\bar{a}\bar{c} + b\bar{a}c + ba\bar{c} + \bar{b}ac) \cdot p = (a \oplus b \oplus c)' \cdot p$. Summarising, we see that the output of FF D does not play any role in its set and reset inputs, that there is only a change when p=1 and that $D_{\text{reset}} = (a \oplus b \oplus c) \cdot p$ whereas $D_{\text{set}} = (a \oplus b \oplus c)' \cdot p$. We get the following circuit for FF D:



Now that we are ready with FF D, we can do the same for FF C. We then get the following four Karnaugh diagrams:

a \ b	cd=00	cd=01	cd=11	cd=10
00	×	0	0	0
01	×	×	1	0
11	×	0	0	0
10	×	×	1	0

C_{reset}; p=0

a \ b	cd=00	cd=01	cd=11	cd=10
00	×	×	0	0
01	×	×	0	0
11	×	×	0	0
10	×	×	0	0

C_{reset}; p=1

a \ b	cd=00	cd=01	cd=11	cd=10
00	0	1	×	×
01	0	0	0	×
11	0	1	×	×
10	0	0	0	×

C_{set}; p=0

a \ b	cd=00	cd=01	cd=11	cd=10
00	0	0	×	×
01	0	0	×	×
11	0	0	×	×
10	0	0	×	×

C_{set}; p=1

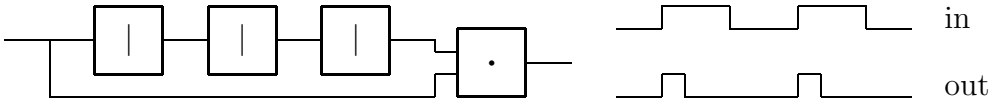
This results in the simplified functions $C_{\text{reset}} = (b\bar{a}d + a\bar{b}d) \cdot \bar{p} = d\bar{p}(a \oplus b)$ and $C_{\text{set}} = (\bar{b}d\bar{a} + bda) \cdot \bar{p} = d\bar{p}(a \oplus b)'$.

If you have verified the above examples, you can now do the analysis for flip-flops B and A. You should get $B_{\text{set}} = \bar{p}\bar{a}c\bar{d}$, $B_{\text{reset}} = \bar{p}ac\bar{d}$, $A_{\text{set}} = \bar{p}b\bar{c}\bar{d}$ and $A_{\text{reset}} = \bar{p}\bar{b}\bar{c}\bar{d}$. You should also be able to draw the circuits.

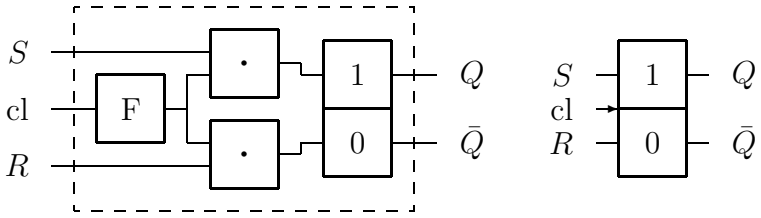
7.3 Clocked SR flip-flop

A circuit with Set-Reset flip-flops generally requires more components than are strictly required and the design of a counter is straightforward but takes a lot of work. In real-

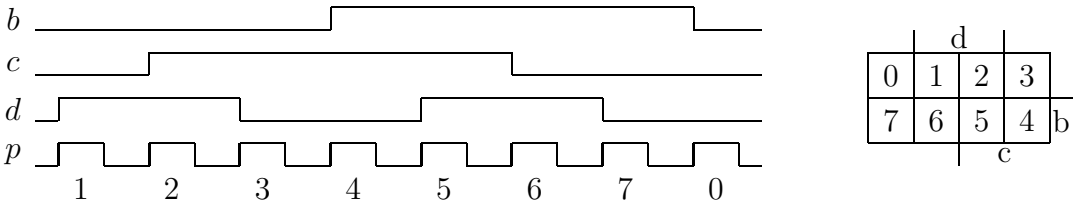
ising a counter, we may not be interested in the **levels** of the input signal, but only in the **positive flanks**. So we could combine a SR flip-flop with some circuit that generates a very brief pulse on the positive flank, having a duration that is just enough for the flip-flop to react. One simple circuit is:



The three inverters act as just a single one, but the total delay is three times that of a single inverter. This, together with the AND gate, will give a positive pulse at a positive flank of the input signal. If we connect this circuit F to a SR flip-flop (below left) we obtain a **clocked SR flip-flop**, the symbol of which is shown to the right:



The "cl" input is called the **clock input**. In order to illustrate the use of a clocked SR FF, we will design the Gray code counter but only for counting from 0 to 7 (8 pulses). Hence, we get the following timing diagram and count sequence in the Karnaugh diagram:



The circuit needs to react on positive flanks, so we don't have to make different tables for p=0 and p=1. Following the count sequence in the Karnaugh diagram (above right) and looking at the values of the flip-flops, we can directly fill Karnaugh diagrams for the set and reset inputs, for example for FF D:



D set **D reset**

Explanation: the count sequence goes 01234567 012... and the value that FF D should have is 01100110 011... Hence, in the first position (0) D is 0. In the second position D must be 1, so we need to generate a set in the first position (set = 1, reset = 0). In the second position (1) D is 1 and in the third position (2) as well: so the circuit could generate a set but not necessarily. We can enter in the second position set = don't care,

reset = 0. In the third position (2) D=1, but in the fourth (3) D=0: we need to generate a reset (set = 0, reset = 1). Doing this for the entire count cycle, we obtain the above two diagrams from which we get the functions

$$D_{\text{set}} = (\bar{c}\bar{b} + cb)p = (b \oplus c)'p$$

and

$$D_{\text{reset}} = (c\bar{b} + \bar{c}b)p = (b \oplus c)p$$

Exactly the same way, but looking at the values that the other flip-flops should have, we obtain for FF C

	d			
0	1	×	×	
0	0	0	×	b
		c		

$C_{\text{set}} = d\bar{b}p$

	d			
×	0	0	0	
×	×	1	0	b
		c		

$C_{\text{reset}} = bdp$

and for FF B

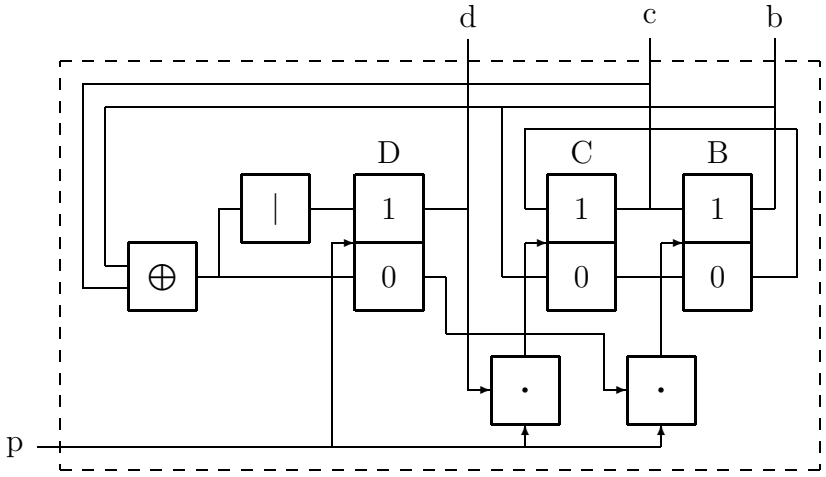
	d			
0	0	0	1	
0	×	×	×	b
		c		

$B_{\text{set}} = \bar{d}cp$

	d			
×	×	×	0	
1	0	0	0	b
		c		

$B_{\text{reset}} = \bar{c}\bar{d}p$

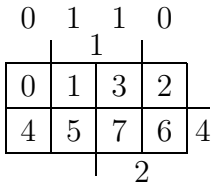
The resulting circuit is much simpler than in the case of a normal SR flip-flop, because it requires less gates:



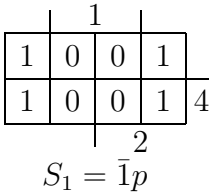
Another example: a binary counter

Gray counters are not very common, but binary counters are. The design is exactly the same as in the previous example. Let us design a binary counter mod 8 with clocked pulses. Let us call the FFs 1, 2, and 4, which corresponds to the weight of the bits in the code. We can actually follow two design strategies: we can make a table and compare the bits, or we can follow the count sequence in the Karnaugh diagram and look at the bit values. Both are shown below:

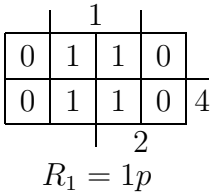
before					after		
4	2	1			4	2	1
0	0	0	0	1	0	0	1
0	0	1	1	2	0	1	0
0	1	0	2	3	0	1	1
0	1	1	3	4	1	0	0
1	0	0	4	5	1	0	1
1	0	1	5	6	1	1	0
1	1	0	6	7	1	1	1
1	1	1	7	0	0	0	0



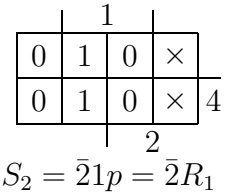
The table shows the 421 bit codes for the count sequence 0,1,2,...7, with the bit codes before the positive flank of the count pulse as well as after. This allows to compare the bits directly and filling the Karnaugh diagrams for the set and reset inputs. In the Karnaugh diagram with the count sequence we always must look at the bit values corresponding with the position. As an example, we'll take a look at the "1" flip-flop. The code starts with state 0 and goes to state 1, so bit "1" goes from 0 to 1. In this case we need a set, so we fill a 1 in the set diagram and a 0 in the reset diagram (shown below). Then the count goes from 1 to 2 and bit "1" goes from 1 to 0. We need a reset (set 0, reset 1). Etc. The complete Karnaugh diagrams, the functions, as well as the circuit are shown below:



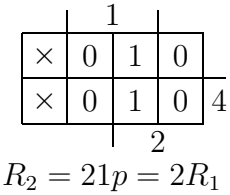
$S_1 = \bar{1}p$



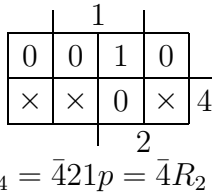
$R_1 = 1p$



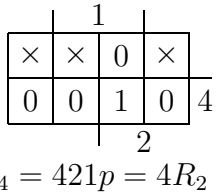
$S_2 = \bar{2}1p = \bar{2}R_1$



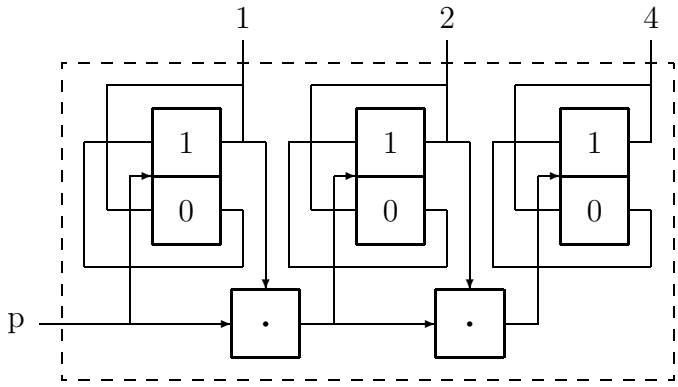
$R_2 = 21p = 2R_1$



$S_4 = \bar{4}21p = \bar{4}R_2$



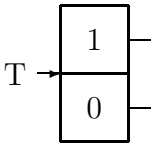
$R_4 = 421p = 4R_2$



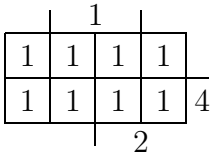
Note: The use of R_1 in S_2 and of R_2 in S_4 is a simplification which is not strictly required. The advantage is that less elements are required. The disadvantage is a *ripple-through* of the counting flank (from FF 1 to FF 2 to FF 4); it takes some time to reach a stable state. You can draw the circuit with the direct combinations of the input signals. On a positive flank of p, all FFs can change at the same time (synchronous counter). This type of counter, which counts on a positive flank, is also called a *coherent* counter.

7.4 T flip-flop

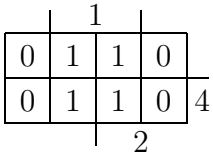
In the last example, the binary counter, we have seen that the FF 1 counts on every input pulse, i.e. its switching is given by $p \cdot \bar{Q}$. In fact all three FFs are acting like this, which can be seen from the feedback loops around each FF. There exist flip-flops which do this automatically, the so-called T or **trigger** FFs. Each time the T FF sees a positive flank on its input, it will change from 1 to 0 or from 0 to 1.



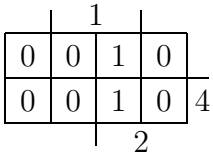
For a binary counter, we must again indicate in Karnaugh diagrams when must be switched. There are **no** don't cares!



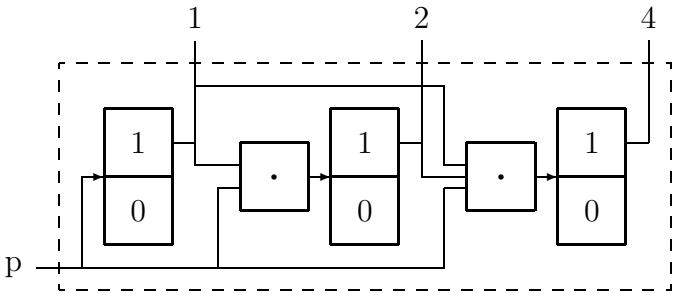
$T_1 = p$



$T_2 = 1p$

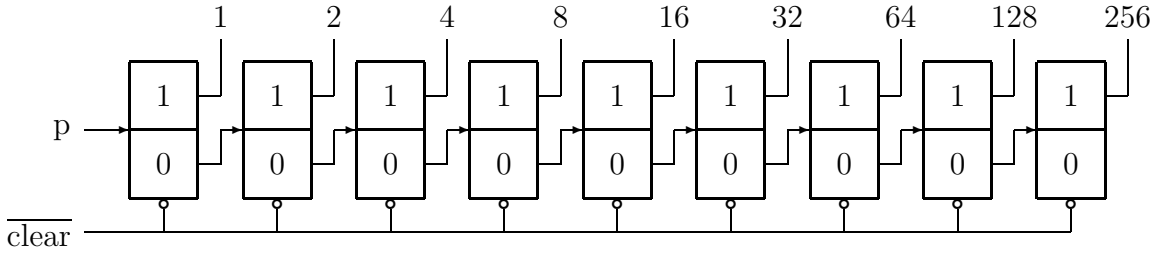


$T_4 = 21p = 2T_2$



This is a synchronous counter, because all FFs can change at about the same time (but

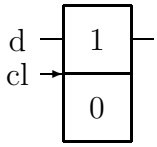
not quite; do you know how to change the above circuit such that all FFs can change at *exactly* the same time?). If a ripple-through is allowed, for example in non time-critical applications, it is even easier to use T flip-flops for making an asynchronous counter:



This is a mod 512 counter or an *up* counter: after a 0 pulse on the $\overline{\text{clear}}$ input, all FFs are 0. At each positive flank on the p input, the counter increments by one. If the delay of one FF is 10 ns, it may take 90 ns to reach a stable state.
Question 1: Do you know how to change this circuit to get a *down* or decremental counter?
Question 2: Do you know how to make an up-down counter which has one extra input, being 1 for up and 0 for down?

7.5 D flip-flop

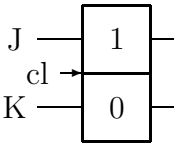
The **D flip-flop** is often used in registers or latches to store some bits, bytes or words temporarily. Its function is simple: it stores the level at the d input when it gets a positive flank at the cl or clock input. In other words: it copies the d input to the Q output. Making a counter with D FFs is also simple: just fill a Karnaugh diagram with the next level of each bit. Check this for a mod 8 binary counter.



7.6 JK flip-flop

The **JK flip-flop** is the most versatile one: it combines the functions of the SR and T flip-flops. The function table and symbol are show to the right. The inputs J and K can assume any value as long as the p input does not get a positive flank: the FF stays in its old state, $Q_{n+1} = Q_n$.

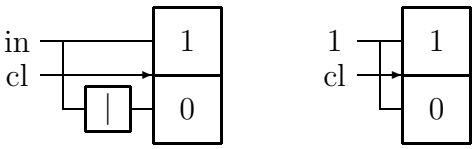
J	K	p	Q_{n+1}	
0	0	0	Q_n	stable
0	1	0	Q_n	stable
1	0	0	Q_n	stable
1	1	0	Q_n	stable
0	0	1	Q_n	stable
0	1	1	0	reset
1	0	1	1	set
1	1	1	$\bar{Q}_n$	T function



If we make a Karnaugh diagram for Q_{n+1},

Q \ JK	00	01	11	10
0	0	0	1	1
1	1	0	0	1

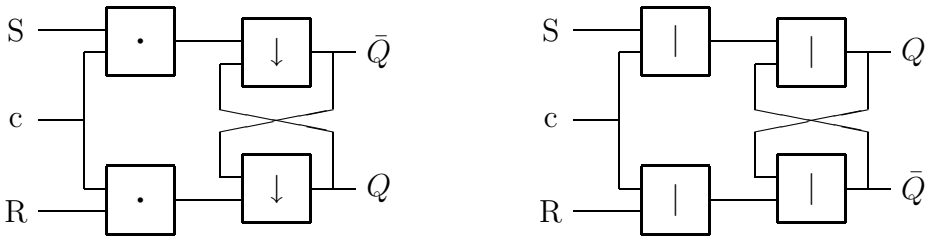
we can see that $Q_{n+1} = J \cdot \bar{Q}_n + \bar{K} \cdot Q_n$. A JK flip-flop can be used to make different functions, like a D or a T FF:



7.7 Other FF types

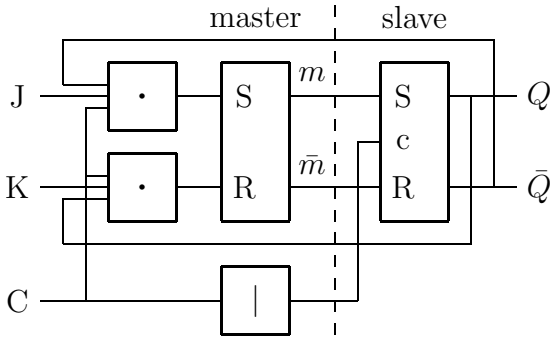
The flip-flops with a clock input which is sensitive to a positive (or negative) flank are often called *edge-triggered* FFs: the inputs, like S and R and D, must have a stable and valid value for a very brief time after the cl input goes high. There exist a few alternatives:

Gated latches. Examples:



These are the gated SR NOR latch (left) and the gated SR NAND latches. If the c input is 0, the S and R inputs are *disabled* and the outputs will remain stable. A 1 on the c input (or a brief positive pulse) *enables* the S and R inputs.

Pulse-triggered or master-slave FFs
Gated latches can be used in sequential circuits with feedback loops if the control signal c consists of a *very* brief pulse. If such a sequential circuit is driven by a periodic clock signal that controls all flip-flops, a method which is used very often, then problems may arise because delays and feedback loops may cause a sequence of transitions, even oscillations. For this reason **master-slave** flip-flops have been introduced. Example:



This is a master-slave JK flip-flop, which is a cascade of two gated SR latches such that the first latch (the master) drives the second (the slave). It works as follows:

• If C=1: the slave is disabled because of the inverter. The master is enabled, that is, the outputs m and $\bar{m}$ follow the inputs $J \cdot \bar{Q}$ and $K \cdot Q$. In other words, the master is *transparent.*

• If C makes a transition from 1 to 0, a negative flank: The master is disabled, that is, m and $\bar{m}$ will assume stable values depending on the previous values of $J \cdot \bar{Q}$ and $K \cdot Q$. Now the slave is enabled and will reflect the stable state of the master after a brief delay.

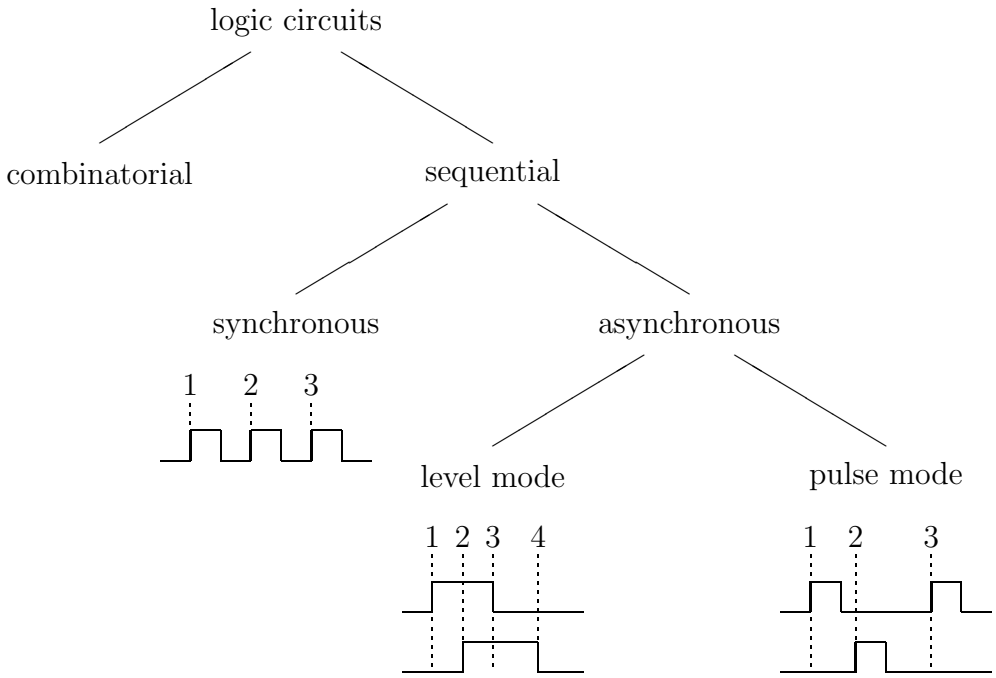
• If C=0: Now the slave is transparent, but the outputs cannot change (they are frozen) because the master provides it with stable input signals.

NOTE: All or most FFs (SR, JK, D, T) are available in three types: gated/basic, edge-triggered and master-slave. Edge-triggered are available for both positive (0 to 1) and negative (1 to 0) edges.

8. Sequential circuits

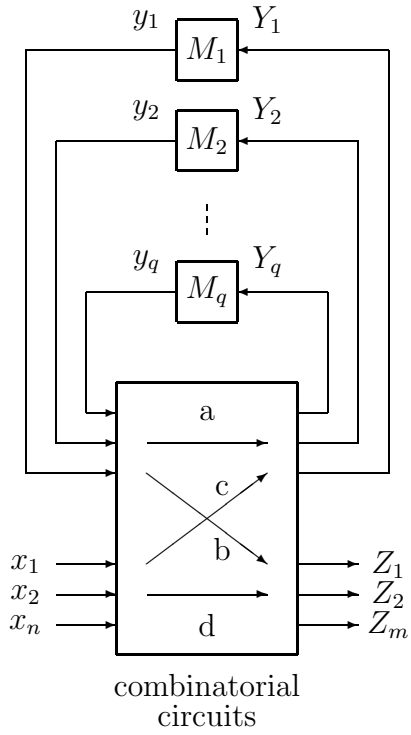
Circuits are divided between **combinatorial** and **sequential**. As already remarked before, a combinatorial circuit consists of only ordinary gates and has no feedback loops. Always: inputs → outputs, also in all subcircuits. If feedback loops are present, or if FFs are used, we speak of a sequential circuit because the next state will depend on the previous states. Note: a FF is a combinatorial circuit with two feedback loops.

Sequential circuits are further divided between **synchronous** and **asynchronous**. Synchronous means: all FFs are controlled by one common clock signal, which is mostly a periodic squarewave signal with a fixed frequency and all FFs may change at the same time, i.e. if they *must* change. This clock is absent with asynchronous circuits, so FFs always act according to their input signals. This can be on the basis of the signal levels (level mode) or on the basis of flanks or edges (pulse mode).



We have already seen examples of sequential circuits: the counters on page 47 (asynchronous level mode), and on pages 49 and 51 (asynchronous pulse mode, but can be easily made synchronous). Different types of counters are obvious examples of sequential circuits. However, they are quite common, also in the kitchen: think of your laundry washing machine. In computers they can be very complex, especially if they need to work at the highest clock speeds: memory and bus controllers in multiprocessor systems for example.

8.1 Huffman model; Mealy and Moore machines



At the right is shown the general scheme of sequential circuits, the **Huffman model** (D.A. Huffman, 1954). We will call:

$x_1 \ldots x_n$: primary variables
$y_1 \ldots y_q$: secondary variables
$Y_1 \ldots Y_q$: secondary functions
$Z_1 \ldots Z_m$: primary or output functions

- The Ms can be flip-flops
- The Ms can be simple connections (feedback loops)

The general form shown here is also called a **Mealy machine** (G.H. Mealy, 1955). If one or more of the combinatorial circuits a,b,c,d is missing, different simplifications are obtained. One very common simplification occurs when the circuit **d** is missing: the so-called **Moore machine** (E.F. Moore, 1956).

The most important simplifications are:
- There is no **d** circuit, the Zs depend only on y. This can be applied when the outputs are not important during changes of the inputs.
- If the Ms are memory elements and there is no **a** circuit, then the Ys are controlled only by xs. The memory elements can "remember" only one previous state.
- If the Ms are simple feedback loops and there is no **a** circuit, the memory function is lost and the circuit has become a combinatorial one.
- If circuit **b** is missing, the memory elements cannot influence the outputs. The entire circuit can be replaced by circuit **d**.
- If **c** is missing and the circuit is in a stable state, then **b** cannot change anymore. The circuit is combinatorial.
- If **c** is missing and the circuit is in an unstable state, then eventually a stable state can be reached. See previous point. However, if no stable state can be reached, the M–**a** closed loop will cause an oscillation. This is no more a sequential circuit with outputs that depend on the inputs and the previous state.

8.2 State or flow diagram

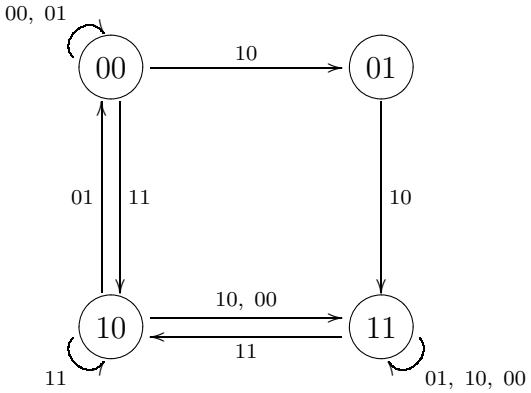
Design problems may be so complex that a simple analysis, as was done for the counters in the previous chapter, is impossible. In such cases a more systematic approach is required. In this approach we split the design problem in two steps: *analysis* and *synthesis*.

A state diagram is based on the combinations of the secondary functions $Y_1 - Y_q$. In the analysis these combinations are represented by a binary number and in the synthesis by a sequence or state number.

Example of analysis:

• There are two secondary functions Y_1 and Y_0. ⓪① means $Y_1 = 0$ and $Y_0 = 1$.

• There are two primary variables x_1 and x_0. $\xrightarrow{10}$ means inputs $x_1 = 1$ and $x_0 = 0$.

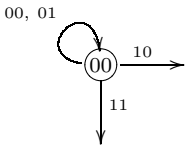


• If a combination of the secondary functions is stable, then *at least one* combination of the primary variables exists for which this state is maintained:

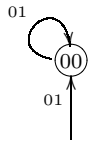


This means: combination $Y_1 = 1$ and $Y_0 = 0$ is stable for $x_1 = 1$ and $x_0 = 0$, so $Y_1 = y_1$ and $Y_0 = y_0$.

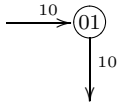
• To make a *complete* analysis, all x_1x_0 arrows for a stable state must be specified (00, 01, 10, 11).



• Furthermore, an arrow pointing at a stable state implies that the corresponding input variables are keeping that stable state.

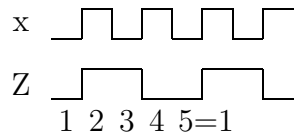


• If ⓪① belongs to an unstable state, this implies that the input combination leads further to a stable or another unstable state:

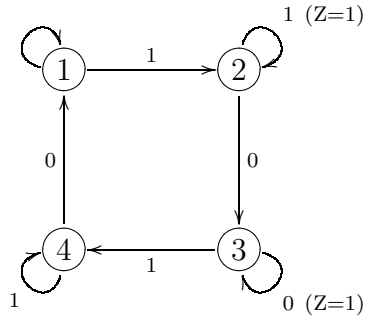


In circuit *synthesis* the Ys are not yet known at the start. Known are the stable states, because these are required. Stable states are given consecutive numbers.

Example of synthesis: design a divide-by-two counter



state	x	Z
1	0	0
2	1	1
3	0	1
4	1	0
1	0	0



If the number of states is large, a state diagram may be complicated and less suitable for simplification methods. It is for example difficult to see whether states are equivalent or not. More suitable are tables: **state-table** or flow-table. In the case of the divide-by-two counter:

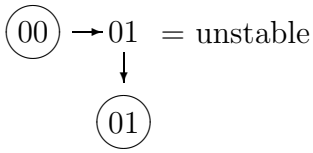
y_1y_0 (secondary variables)	primary variables x = 0	primary variables x = 1
00	(00)	01
01	11	(01)
11	(11)	10
10	00	(10)
	Y_1Y_0 (secondary functions)	

Analysis

	x = 0	x = 1
1	(1)	2
2	3	(2)
3	(3)	4
4	1	(4)

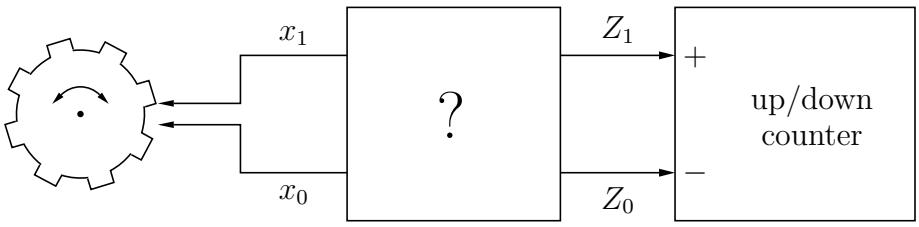
Synthesis

We see that for stable states $Y_i = y_i$. In general, a change of the primary variables leads first to another position in the same **row**, and then to another position in that **column**. The intermediate state is unstable. Exceptions will be treated later (races).

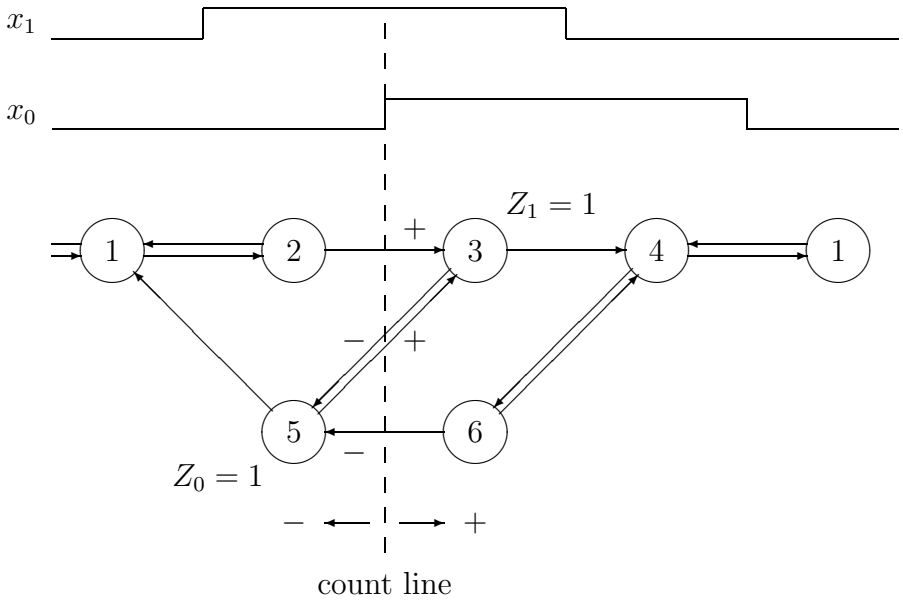


• If there is only one stable state in each row, we speak of a *primitive* state table.

Example to be used further in the text: make a circuit that allows to read the position of an axis by means of a cam-disc and an up/down counter:



The principle is very simple. At the input we have a cam disc with two microswitches. If the disc rotates, the switches will be opened and closed in certain sequences. Depending on these sequences, the circuit must give a pulse to the counter: either increment (+ input) or decrement (– input). The circuit must determine the rotation direction of the disc, e.g.
rotation to the right: x_1 becomes 1 first, then x_0
rotation to the left: x_0 becomes 1 first, then x_1.
However, there is the possibility that the rotation orientation changes before both inputs are 1, so we first start with making a diagram with all possibilities and by making a flow diagram:



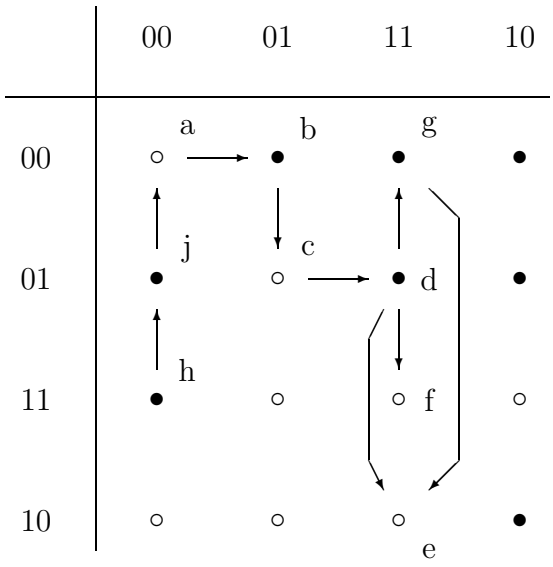
Explanation: After drawing the signals x, we start with drawing circles from left to right where the signals have the combinations 00, 10, 11, and 01, filling arbitrary numbers 1, 2, etc. Then we define a count line where both signals become 1. If we pass this line going to the right, we need to increment the counter (+), if we pass going to the left we need to decrement (–). From state 3 going to the left, we cannot pass via state 2 because the disc could stop and turn right again: we need another state 5. Try to understand the entire diagram. Once the diagram is complete, we can fill the state table:

	x_1x_0 00	01	11	10	Z_1	Z_0	
1	(1)	4	–	2	0	0	start state; no count information
2	1	–	3	(2)	0	0	ready for increment
3	–	4	(3)	5	1	0	passed count line: increment
4	1	(4)	6	–	0	0	avoid state 3 in rotating left
5	1	–	3	(5)	0	1	passed count line: decrement
6	–	4	(6)	5	0	0	ready for decrement

8.3 Races

The figure below shows the state table and the output table of a Mealy machine with two inputs, two secondary functions and one output function. Also shown is a schematic representation of the state table in order to illustrate state transitions. Stable states ($Y_i = y_i$) are shown by ∘s, unstable states by •s.

	x_1x_0				x_1x_0			
y_1y_0	00	01	11	10	00	01	11	10
00	00	01	10	11	0	0	0	0
01	00	01	10	11	0	0	0	0
11	01	11	11	11	0	0	0	0
10	10	10	10	11	1	0	0	0
	Y_1Y_0				Z			



Let's see what can happen. Suppose the circuit is in state **a**. The input combination 01 leads to state **b** (unstable) and then to **c** (stable).

If now the input becomes 11, the Y_1Y_0 need to change from 01 to 10, and the circuit goes to **d** (unstable). For flip-flops this means that Y_1 must be set and that Y_0 must be reset, but the outputs y_1 and y_0 are still, for a short time, 0 and 1. There are now three options:

• If both flip-flops change at *exactly* the same time, the circuit will go to state **e** (stable). However, *exactly* the same time is almost impossible in practice because of small differences in delays....

• If Y_1 changes first, we will have for a short time $y_1y_0 = 11$ and the circuit will go to state **f** (stable).

• If Y_0 changes first, we will have for a short time $y_1y_0 = 00$ and the circuit will go to state **g** (unstable), with $Y_1Y_0 = 10$ and this leads to the stable state **e**.

Hence, depending on the difference in speed between Y_1 and Y_0, either the stable state **f** or the stable state **e** is reached.

The situation in which more than one secondary variable must change is called a **race**. If the final state depends on the order in which the secondary variables change, this is called a **critical race**. If there are two secondary variables, we need to check all transitions $00 \leftrightarrow 11$ and all $01 \leftrightarrow 10$.

One solution for solving critical races is to select flip-flops with exactly the same delay, but even then, after using circuits for a certain time the characteristics may change. It is therefore better to avoid critical races. If impossible, unstable states can be added such that races become uncritical. This will be explained later on.

In the above figure: state **f** with inputs 00 leads to the unstable states **h** and **j** to **a**. No critical races are possible because for each change only one Y changes.

Check: What happens if in state **a** the inputs become 10? Is this race critical?

8.4 Synthesis of sequential circuits

With primary variables and required stable states, a **primary state table** is made first. This was already illustrated on page 59. States are defined by the state *numbers* as well as by the *row* and *column* in which they appear. Below: if we speak about a state, both state number and row/column position are meant.

8.4.1 Equivalences

An incomplete definition of the behaviour of a circuit can lead to equivalent stable states in the primary state table. **Equivalent states** have the same input combination, the same output functions and go to the same next states for all new input combinations. Equivalent states can be detected by comparing rows in the primary state table.
If for two rows:
(1) the stable states are in the same column,
(2) the output combinations are equal, and
(3) all other columns have the same numbers,
then the two rows indicate equivalent states.

Theorem: equivalent rows can be reduced to a single row.
Proof: trivial, rename one stable state.

Example: the Moore machine shown to the right.
• In this table rows 2 and 4 can be unified by changing all 4s to 2s. Row 4 can be deleted.
• If all 4s have been changed to 2s, we see that now rows 1 and 3 are also equivalent: change all 3s to 1s and delete row 3.
As a result, we have eliminated 2 rows out of the seven and we obtain the reduced table shown to the right.

This analysis is simple and straightforward for small tables, but in bigger tables it may be difficult to see what happens. We then need a more systematic approach. One such approach will be studied using another example.

	x_1x_0 00	01	11	10	Z_1	Z_0
1	(1)	2	–	6	0	0
2	1	(2)	5	–	1	0
3	(3)	4	–	6	0	0
4	1	(4)	5	–	1	0
5	–	7	(5)	6	0	1
6	3	–	5	(6)	1	1
7	3	(7)	5	–	1	1

	x_1x_0 00	01	11	10	Z_1	Z_0
1	(1)	2	–	6	0	0
2	1	(2)	5	–	1	0
5	–	7	(5)	6	0	1
6	1	–	5	(6)	1	1
7	1	(7)	5	–	1	1

	x_1x_0 00	01	11	10	Z_1	Z_0
1	(1)	2	–	3	0	0
4	(4)	7	–	8	0	0
12	(12)	2	–	8	0	0
3	1	–	6	(3)	0	1
8	12	–	11	(8)	0	1
5	–	9	(5)	3	1	0
6	–	2	(6)	10	1	0
11	–	2	(11)	10	1	0
2	4	(2)	5	–	1	1
7	12	(7)	6	–	1	1
9	1	(9)	11	–	1	1
10	1	–	6	(10)	1	1

This is a Moore machine and the outputs are already sorted together because this is easier to see. From this table we first construct another table which is shown below.

Horizontal: make three columns of which the first two are marked "done" and $\not\equiv$ (not equivalent). Then make columns for all row combinations that *could be* equivalent because the stable states are in the same column and the outputs are the same. E.g., for outputs 00 we have 1-4, 1-12 and 4-12.

Vertical: make rows with the required equivalences for making a combination equivalent, looking at all stable states in the columns. E.g., all combinations between (1,4,12), then (3,8,10), then (5,6,11), etc.

We can now start filling the table.

done	$\not\equiv$		1-4	1-12	4-12	3-8	5-6	5-11	6-11	2-7	2-9	7-9
√	x	1-4									x	
		1-12				x						x
√	x	4-12								x		
		3-8	x	x								
√	x	5-6								x		
√	x	5-11									x	
		6-11				x						x
√	x	2-7	x		x							
√	x	2-9					x	x				
		7-9										
√	x	3-10					x	x				

1) Indicate, with a x, in each column which combinations must be equivalent to make the column combination equivalent. In the state table we can see that for combination 1-4 these are 2-7 and 3-8.

2) Start the search with a combination which is certainly *not* equivalent, e.g. because of unequal outputs. In this example we can start with 3-10. Because 3-10 is not equivalent, the combinations 5-6 and 5-11 are also not equivalent. Put a **x** in the $\not\equiv$ column in row 3-10 and then in rows 5-6 and 5-11.

3) We are ready with row 3-10, so put a $\surd$ in the "done" column.

4) Proceed with another combination which is not equivalent, e.g. 5-6 or 5-11. If 5-6 not equivalent, then 2-7 neither. And so on.

5) The rows which are empty in the "done" and $\not\equiv$ columns, after completing the procedure, indicate equivalent rows of the state table. In this example the equivalent states

are 1-12, 3-8, 6-11 and 7-9. Hence, $12 \rightarrow 1$, $8 \rightarrow 3$, $11 \rightarrow 6$ and $9 \rightarrow 7$. We can delete the rows 12, 8, 11 and 9, obtaining a reduction from 12 to 8 rows:

	x_1x_0 00	01	11	10	Z_1	Z_0
1	(1)	2	–	3	0	0
4	(4)	7	–	3	0	0
3	1	–	6	(3)	0	1
5	–	7	(5)	3	1	0
6	–	2	(6)	10	1	0
2	4	(2)	5	–	1	1
7	1	(7)	6	–	1	1
10	1	–	6	(10)	1	1

8.4.2 Pseudo equivalences

Often certain changes of the primary variables are impossible because of the conditions of the problem. These are indicated by a – (don't care) in the state table. Again, a don't care can be replaced by any combination of the secondary functions. By assuming clever states, some rows can be made equivalent with other rows. This is called a **pseudo equivalence**. This can also be done for rows which are equivalent except for output values, if the latter contain don't cares. Two examples:

(1)	3	–	–	0	0	}	(1)	3	–	4	0	0
(2)	3	–	4	0	0							

(1)	3	–	4	0	0	}	(1)	3	–	4	0	0
(2)	3	–	4	0	–							

Often, the reduction of the number of rows by means of pseudo equivalences is not trivial because one can choose between multiple solutions.

8.4.3 Mixing of rows

The **mixing of rows** is in fact the same procedure as pseudo equivalence, but now by merging unstable and stable states on the basis of don't cares. In mixing a stable with an unstable state, the stable one has priority. The outputs need not be equal, but: a Moore machine turns into a Mealy machine! For a Mealy machine, mixing is only possible with the same outputs and taking don't cares into account. Example: the axis problem on pages 59-60. At the left is the original state table, at the right the result of mixing rows

1-2, 3-5 and 4-6:

	x_1x_0 00	01	11	10	Z_1	Z_0
1	(1)	4	–	2	0	0
2	1	–	3	(2)	0	0
3	–	4	(3)	5	1	0
4	1	(4)	6	–	0	0
5	1	–	3	(5)	0	1
6	–	4	(6)	5	0	0

	x_1x_0 00	01	11	10	x_1x_0 00	01	11	10
1	(1)	4	3	(1)	00	- -	- -	00
3	1	4	(3)	(3)	- -	- -	10	01
4	1	(4)	(4)	3	- -	00	00	- -
		Y_1Y_0				Z_1Z_0		

You can try yourself another "mix" by merging rows 1-4 and rows 3-5. Generally, multiple mixing possibilities occur. It is not easy to select the easiest one in advance. In the further synthesis *all* possibilities must be examined, finally selecting the most simple circuit.

8.4.4 Assigning the secondary functions

By means of equivalence and pseudo equivalence the number of states has been reduced. By means of mixing rows the number of combinations of the secondary functions has been reduced. For realising the sequential circuit, we must now assign flip-flops to the Ys.

Suppose there are left n different combinations of the secondary functions (n is the number of rows in the resulting state table). These n combinations can be realised with $k \geq [\log_2 n]$ functions Y_i. If $n = 7$, then $k \geq [\log_2 7] = 3$. In the case of our example "axis position" we have $n = 3$ and therefore $k = 2$. Hence, we need two flip-flops.

We can make different assignments for the states 1, 3, and 4:

AS1	Y_0 0	1
Y_1 0	1	3
Y_1 1	4	a

AS2	Y_0 0	1
Y_1 0	1	3
Y_1 1	b	4

AS3	Y_0 0	1
Y_1 0	1	4
Y_1 1	c	3

We will start with studying assignment **AS1**. The transitions between state 1 ($Y_1 = Y_0 = 0$) and 3 ($Y_1 = 0, Y_0 = 1$), as well as between 1 and 4 ($Y_1 = 1, Y_0 = 0$) are *uni-variant.* I.e. only one secondary function changes. In these cases no critical races can occur, so we don't need to study these races in the state table. However, in the table also transitions between states 3 and 4 occur, so the Ys go from 01 to 10, and vice versa. These transitions are *bi-variant,* the two Ys change which often goes via state 1 (00) or via state a (11).

• Transition 3 → 4 occurs for input 01, the circuit follows the **cycle** (3) → 1 → (4). This is shown in the table below at the left. If the transition 3 → 4 goes via a (01 → 11 → 10), a similar cycle occurs if we add row a: in row a we place state 4 in column 01.

This is shown in the table below at the right:

		00	01	11	10
00	1	①	4	3	①
01	3	1	4	③	③
10	4	1	④	④	3

		00	01	11	10
00	1	①	4	3	①
01	3	1	4	③	③
10	4	1	④	④	3
11	a	–	4	–	

Both cycles $3 \to 1 \to 4$ and $3 \to a \to 4$ constitute an *uncritical* race, because they end both in the same state.

• Going from 4 to 3 if the input is 10, the cycle ④ → 1 → ③ or $10 \to 00 \to 01$ could be followed (below left). However, 1 is a stable state, so this cannot be allowed. This can be solved by *forcing* the circuit to go always via state a (see below right): put in row 4 an a for input 10 and in row a state 3.

		00	01	11	10
00	1	1	4	3	①
01	3	1	4	3	③
10	4	1	4	④	3
11	a	–	4	–	

		00	01	11	10
00	1	1	4	3	①
01	3	1	4	3	③
10	4	1	4	④	a
11	a	–	4	–	3

Now all problems have been solved and the assignment can be included in the state table. Note that y_1y_0 has been re-ordered to make directly a Karnaugh diagram:

		x_1x_0				x_1x_0			
		00	01	11	10	00	01	11	10
y_1y_0	00	00	10	01	00	00	- -	- -	00
	01	00	10	01	01	- -	- -	10	01
	11	- -	10	- -	01	- -	- -	- -	- -
	10	00	10	10	11	- -	00	00	- -
		Y_1Y_0				Z_1Z_0			

Solution for **AS1**:
$Y_1 = y_1\bar{y}_0x_1 + \bar{x}_1x_0$
$Y_0 = y_0x_1 + \bar{y}_1x_1x_0 + y_1x_1\bar{x}_0$
$Z_1 = y_0x_0$
$Z_0 = y_0\bar{x}_0$

This solution is not necessarily the most simple one. We need to check also assignments AS2 and AS3.

For **AS2**:

	x_1x_0				x_1x_0			
	00	01	11	10	00	01	11	10
1	①	4	3	①	00	- -	- -	00
3	1	4	③	③	- -	- -	10	01
4	1	④	④	3	- -	00	00	- -
b	1	4	–	–	- -	- -	- -	- -

AS2	Y_0	
	0	1
Y_1 0	1	3
Y_1 1	b	4

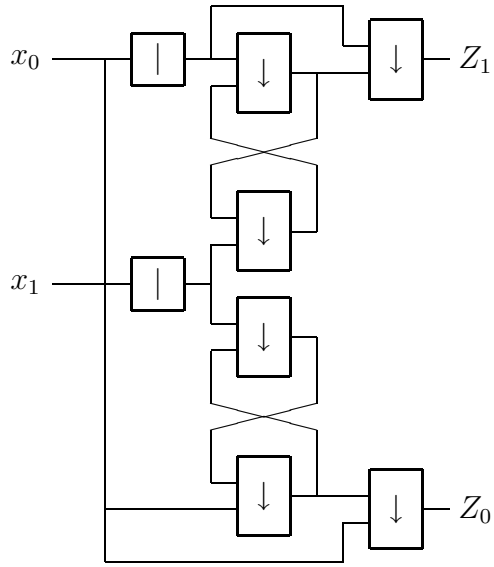
• $1 \rightarrow 3$ is uni-variant.
• $1 \rightarrow 4$ is bi-variant: either $1 \rightarrow 3 \rightarrow 4$ or $1 \rightarrow b \rightarrow 4$. By putting in row b column 01 a 4, a race is created. However, this is uncritical.
• From 3 all changes are uni-variant.
• From 4 to 3 is uni-variant.
• $4 \rightarrow 1$ via 3 or via b. In row b column 00 we put a 1.
We obtain the assignment:

		x_1x_0				x_1x_0			
		00	01	11	10	00	01	11	10
y_1y_0	00	00	11	01	00	00	- -	- -	00
	01	00	11	01	01	- -	- -	10	01
	11	00	11	11	01	- -	00	00	- -
	10	00	11	–	–	- -	- -	- -	- -
			Y_1Y_0				Z_1Z_0		

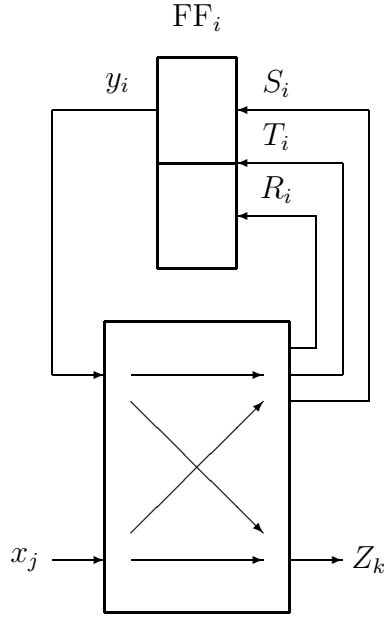
Solution for **AS2**:
$Y_1 = y_1x_0 + \bar{x}_1x_0$
$Y_0 = y_0x_1 + x_0$
$Z_1 = \bar{y}_1x_0$
$Z_0 = y_0\bar{x}_0$

We see that the AS2 solution is more simple than the AS1 solution.
Check yourself: the AS3 assignment leads to the solution
$Y_1 = y_1x_1 + y_0x_1\bar{x}_0 + \bar{y}_0x_1x_0$
$Y_0 = y_0x_1 + y_1x_1 + \bar{x}_1x_0$
$Z_1 = y_1x_0$
$Z_0 = y_1\bar{x}_0$

Because the AS3 solution is also more complex than the AS2 one, we will realise the AS2 solution by means of NOR gates:



In this realisation, the M elements in the general Huffman model are feedback loops from gate outputs to inputs. This results in a memory function. Recall that gates plus feedback form SR flip-flops. If the memory function is incorporated into the M elements, i.e. M = SR flip-flop, this results in the same circuit. But in this case an alternative approach is possible:



The FFs can be of any type, i.e. SR, SR with T, T, etc. We will take a look at realising the "axis-position" circuit by means of SR and T flip-flops.

• Using SR flip-flops, the set and reset functions can be derived directly from the state table. Applying the AS2 assignment, we get the following table for Y_1:

y_1y_0 \ x_1x_0	00	01	11	10
00	0	<u>1</u>	0	0
01	0	<u>1</u>	0	0
11	<u>0</u>	1	1	<u>0</u>
10	<u>0</u>	1	–	–

FF1 *must* get a set if FF1=0 and $Y_1 \rightarrow 1$.
FF1 *may* get a set if FF1=1 and $Y_1 \rightarrow 1$.
If in a row with $y_1 = 0$ a $Y_1 = 1$ occurs, this means that FF1 must get a set for the corresponding input combination. This is indicated by underlining the 1 in the state table. In the rows with $y_1 = 1$ a set may be given for $Y_1 = 1$, but not necessarily: a don't care. Hence, $S_1 = \bar{x}_1 x_0$.

Similar for the reset function:
FF1 *must* get a reset if $y_1 = 1$ and $Y_1 = 0$; underline the 0s.
FF1 *may* get a reset if $y_1 = 0$ and $Y_1 = 0$. Hence $R_1 = \bar{x}_0$. The analysis for FF0 is exactly the same. This procedure is much more simple than the one followed in the chapter "memory elements."

• Using the trigger function or T flip-flops; the FF always changes its state. This implies that all underlined 1s and 0s in the above table must be taken together. Now the not-underlined 1s and 0s may not be taken as don't cares, only the original don't cares at the bottom line to the right! We then get the following Karnaugh diagram for FF1, from which we obtain the function $T_1 = \bar{y}_1\bar{x}_1x_0 + y_1\bar{x}_0$.

	1		
	1		
1			1
1			1

Check yourself: for FF0 we get $S_0 = x_0$, $R_0 = \bar{x}_1\bar{x}_0$ and $T_0 = y_0\bar{x}_1\bar{x}_0 + \bar{y}_0x_0$. In this example the set and reset functions are more simple than the trigger functions, so it is better to use SR FFs instead of T FFs. Finally, the output functions Z_1 and Z_0 must be realised, but these have already been determined.

8.4.5 Increasing the number of secondary functions

Shown below is a state table which, after assigning two secondary functions, shows critical races. The states marked with an $*$ lead to critical races.

	x_1x_0 00	01	11	10
1	①	2	3	3
2	②	②	3	4
3	③	4	③	③
4	1	④	3	④

Y_1 \ Y_0	0	1
0	1	2
1	4	3

y_1y_0 \ x_1x_0	00	01	11	10
00	(00)	01	11*	11*
01	(01)	(01)	11	10*
11	(11)	10	(11)	(11)
10	00	(10)	11	(10)

Y_1Y_0

In row 00, from stable state 00 with input 10 the state 11^* is reached and than the stable state 11 if both Ys change at the same time. If not at the same time, the stable state 10 is reached. So this is a critical race. From 00 with input 11 a race is also entered,

but this one is uncritical. There is no Y combination free which could be used to avoid the critical race. The same problems arise for all other assignments!

By assuming an additional secondary function, the number of Y combinations is doubled, so we get more possibilities to create uni-variant changes and to convert races into cycles. A new assignment with three Ys:

Y_2 \ Y_1Y_0	00	01	11	10
0	1	2	c	3
1	4	a	d	b

	x_1x_0 00	01	11	10
1	1	2	3	3
2	2	2	3	a
3	3	b	3	3
4	1	4	3	4
a	–	–	–	4
b	–	4	–	–
c	–	–	–	–
d	–	–	–	–

	x_1x_0 00	01	11	10
000	000	001	010	010
001	001	001	010	101
010	010	110	010	010
100	000	100	010	100
101	—	—	—	100
110	—	100	—	—
011	—	—	—	—
111	—	—	—	—

• The column with input 11 can never give critical races because it contains only 3s. So stable state 3 is reached directly or via a cycle. From 2 and input 11 the state 3 must be reached. This can via 1 because input 11 in state 1 leads also to 3. This transition can also go via c. The table is not changed.
• $1 \rightarrow 2$ and $1 \rightarrow 3$ are uni-variant.
• From 2 to 4 with input 10 is not allowed via 1 because 1 with input 10 leads to stable state 3. The transition from 2 to 4 must go via a. So, row 2 column 10 gets an a and in row a comes a 4 in the same column.
• $3 \rightarrow 4$ gives a critical race, the circuit may end up in stable state 2. This is replaced by a cycle by including state b.
After substituting the assignment we obtain the rightmost table, from which with Karnaugh the Y functions can be determined.

NOTE: This chapter is <u>not</u> the complete theory behind sequential circuits. There are more complications, like noise sensitivity and power-on problems (if the power is switched on, flip-flops may get a random state, and these may lead to a stable state which is not prescribed). Furthermore, sometimes a circuit should be as fast as possible. In such cases cycles must be avoided or minimised.

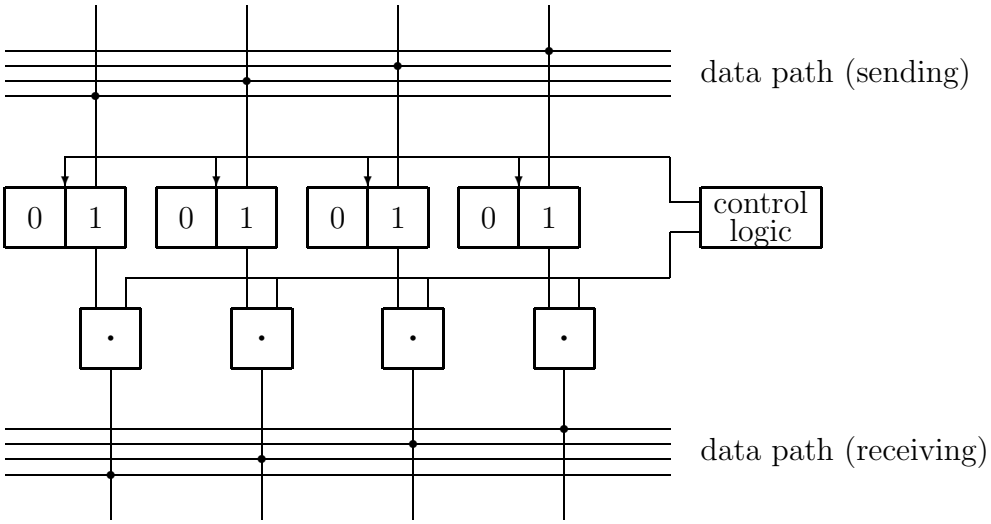
9. Registers and functional elements

In this chapter we will take a brief look at a few different circuits that are available. These are registers and bus drivers, comparators, de- and encoders, as well as (de)multiplexers. Such circuits are quite common in microprocessor systems and available in MSI (medium scale integration) and LSI (large scale integration) chips. We cannot deal with all different forms, because this is just an introduction which should allow to understand the terminology and functionality. The interested reader should take a look at some data books, for example the *Texas Instruments* Data Book.

9.1 Registers and buses

As remarked in Chapter 7, the D flip-flop is one of the simplest FF types, and mainly used to store some bits temporarily. As an example, we can have a few parallel input lines and the same number of output lines. These may be the data or the address buses of a microprocessor, of a complex CPU (central processing unit) printed circuit board, or all the connections of all boards in a big *mainframe* computer.

One of the simplest forms to store data temporarily is taking a few D flip-flops in parallel, perhaps with a few ANDs if we want to control not only the input but also the output:

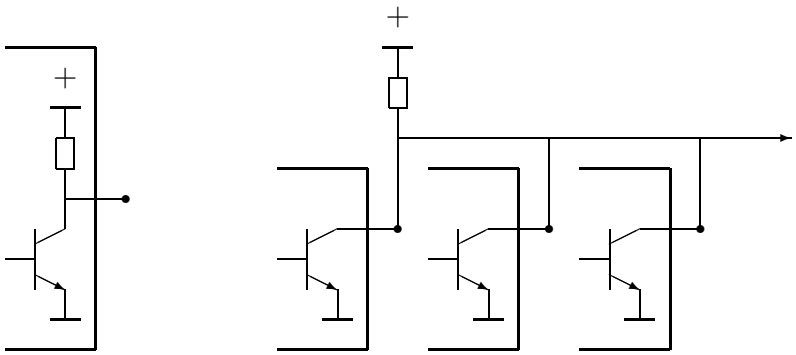


The sending data path is also called **input bus** or simply inbus, the receiving data path is then called **output bus** or outbus. The signals from the control logic are normally called **input strobe** and **output strobe**. The **register** shown above has 4 bits, but this could be 8, 16, 32, or in the case of modern RISC[5] machines 64 bits.

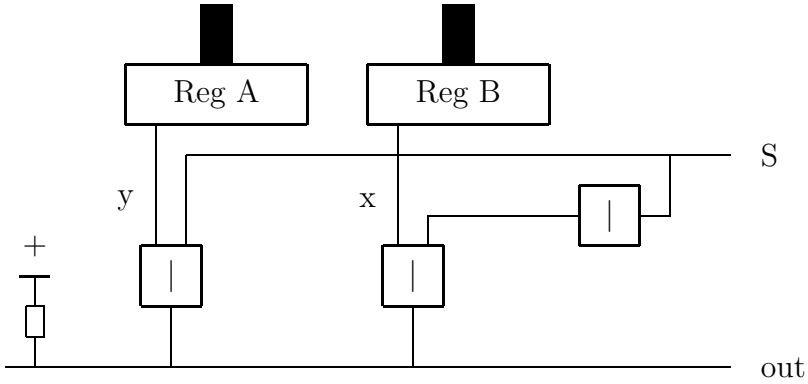
If we want to copy e.g. one byte from one input bus to two output buses, we can take two registers in parallel and control the data flow by means of two signals: an input strobe for each register. However, if we want to copy data from two input buses to one central output bus, we need other solutions. One such solution is based on *open collector outputs.* Any gate has a transistor at its output and this is normally already connected to a resistor and e.g. +5V inside the chip (below left). If the collector is not connected to a resistor

[5]RISC = reduced instruction set computer

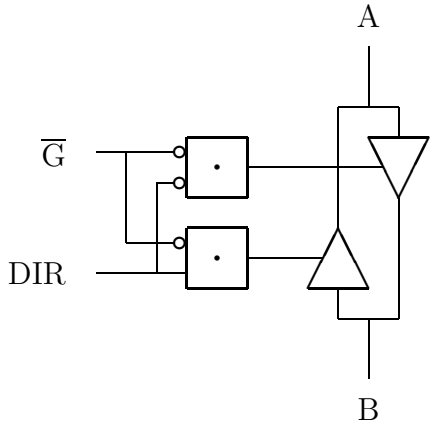
inside the chip, i.e. the collector is "open," we need to take a resistor outside the chip, in which case we may connect multiple open collector outputs to a single line (below right).

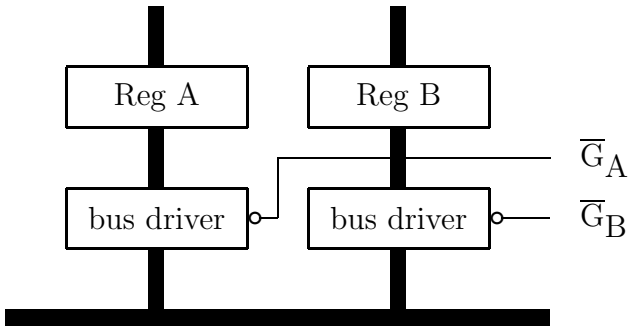


The external resistor is called *pull-up* resistor because it "pulls" the outputs to e.g. +5V. If one or more transistors is conducting, the output line will go to 0. For this reason we speak of a *wire-AND* connection, for example to select one register to a common output bus (we show only one bit of the registers and the output bus):



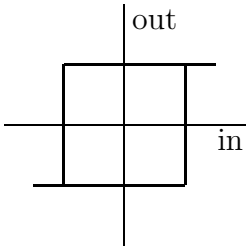
In this example there are two input buses and two registers of N bits. If the control signal S (from select) is 0, the output of the left NAND will be 1, but the output of the right NAND and therefore the output line, will be $\overline{1 \cdot x} = \bar{x}$. In the case that S=1 we will have $\bar{y}$. In this example we used NANDs, but we could have used ANDs as well. Bus buffering is so common in computer systems that special circuits are available. Even bi-directional *bus transceivers* exist which allow to send data from bus A to B or from B to A (figure to the right). Such a transceiver has two control inputs: DIR for defining the direction and the active-low $\overline{\text{G}}$ which is an *enable* input. If $\overline{\text{G}} = 1$ the outputs have such a high impedance that they virtually do not exist (3-state outputs; 0, 1, or high impedance). Such 3-state transceivers allow to connect different registers or buses together, see the next figure.





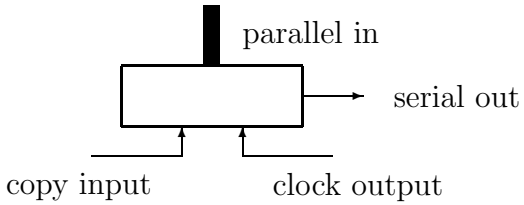
In this case there are no bi-directional bus transceivers but one-directional bus drivers. Care must be taken that the two enable signals $\overline{G}_A$ and $\overline{G}_B$ are not both 0 at the same time.

If the connections are long, which means that noise from neighbouring connections can be induced, transceivers can be selected with hysteresis. This hysteresis means that the input signal must exceed a certain voltage difference before the output will change, see figure to the right. Note that this figure is an idealisation, the hysteresis function does not need to be an ideal square.

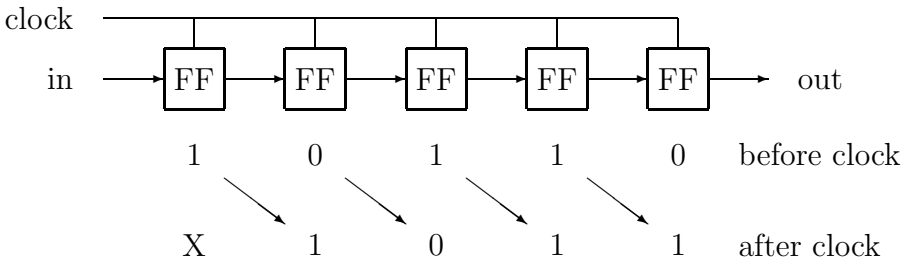


Shift registers

Sometimes data is passed in a serial way instead of in parallel. Examples are a terminal link type "RS232" and the MIDI interface for connecting synthesisers (or synthesisers with computers). In such cases the data need to be converted from parallel to serial, which can be done by means of **shift registers**.



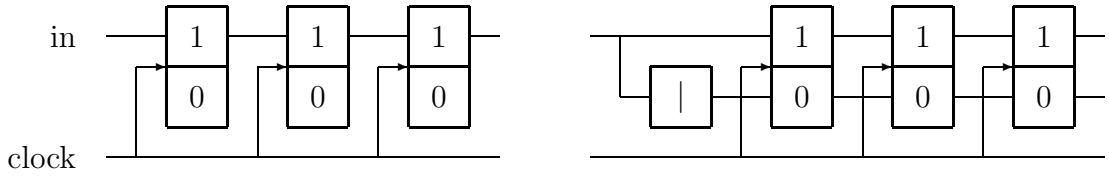
A shift register contains a number of flip-flops which can pass their information to their left or right neighbour. Passing = shifting. There exist different types: serial in, serial out; parallel in, serial out; and serial in, parallel out. An example, serial in serial out is shown below:



• The *rightmost* bit is lost after shifting to the right, unless it is stored somewhere else or transmitted.

• The leftmost bit can be filled in three ways: (a) always a 0, (b) always a 1, or (c) with other information. An example of the last option is to make a feedback loop from output to input, in which case we will get a *cyclic* shift.

Shift registers are also used for hardware multiplication and division, and the shift function is therefore often incorporated into an "accumulator" or ALU (arithmetic logical unit) of a CPU (central processing unit). Shift registers can be realised by means of cascades of D (left) or JK/SR (right) flip-flops:



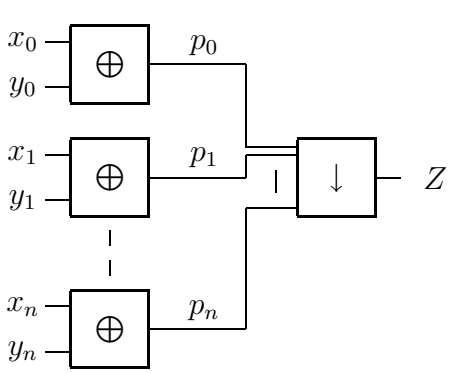
Apart from simple gates like ANDs, ORs, counters, etc, there are many MSI (medium scale integration) circuits that perform specific tasks. These are often used in computer circuits: comparators, encoders, decoders, multiplexers, demultiplexers.

9.2 Comparator

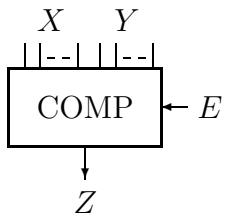
Purpose: compare two binary numbers X and Y.

- For many applications only one output is necessary, i.e. $Z = 1$ if $X = Y$ and $Z = 0$ if $X \neq Y$.

  Example



$$p_0 = x_0 \oplus y_0 = \begin{cases} 1 & \text{if } x_0 \neq y_0 \\ 0 & \text{if } x_0 = y_0 \end{cases}$$

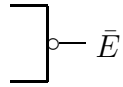


  From the above scheme, we can observe that

$$Z = \overline{p_0 + p_1 + \cdots + p_n} = \overline{p_0} \cdot \overline{p_1} \cdots \overline{p_n}$$

  Often there is an enable (E) input:

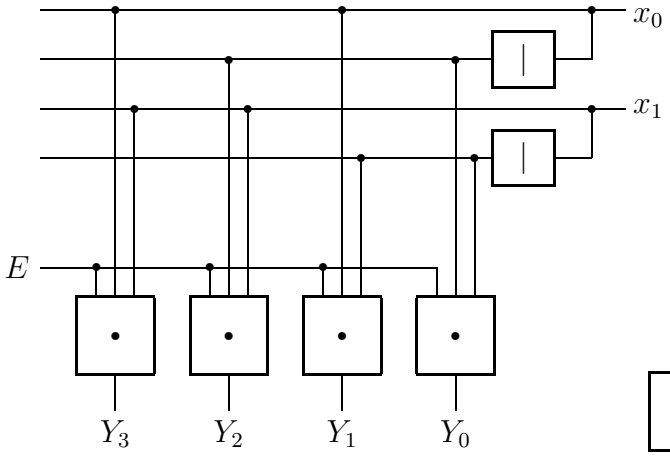
  $Z =$ invalid if $E = 0$, valid if $E = 1$

  Another option is to use $\bar{E}$ (valid if $\bar{E} = 0$), indicated by 

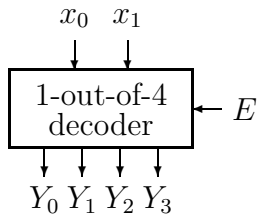
- If additional information is required, i.e. $X < Y$ and/or $X > Y$, relatively complex circuits with many gates are required, especially if the number of bits is large. It is then simpler to subtract X and Y and to examine the sign bit.

9.3 Decoder

We have seen examples of decoders on page 31, i.e. BCD and 7-segment. A normal decoder (*binary*) is not based on exploiting don't cares in the function table.
Purpose: a binary 1-out-of-2^n decoder converts a code to one bit line, i.e. exactly 1 output bit may be one.



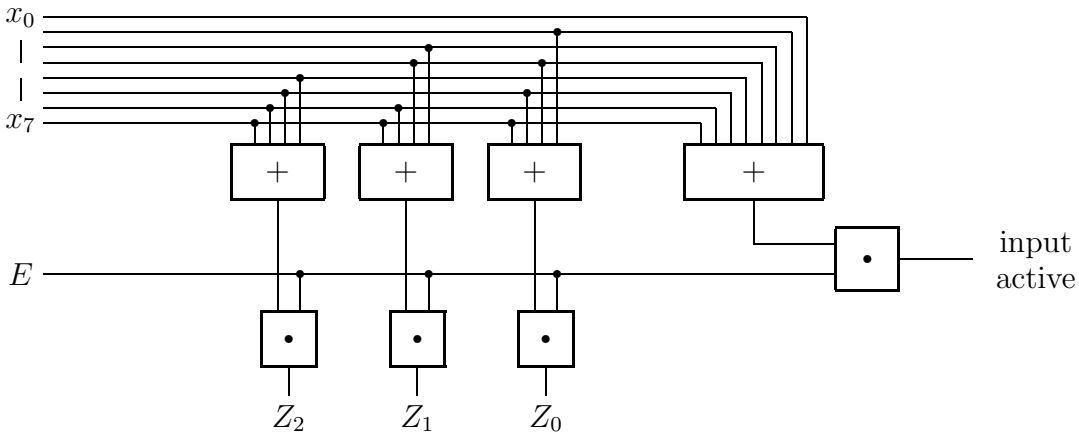
in		out			
x_1	x_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

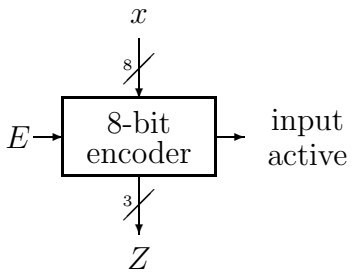


9.4 Encoder

An encoder does exactly the reverse, namely it generates the binary code of an active input line. In general, 2^k inputs give k outputs. An extra output line may indicate whether there is one input line active (to detect the case that *no* input line is active). It may also be necessary to detect the case that multiple lines are active and an invalid code is generated.
A solution to the latter is a so-called *priority encoder*: if more than one input line is active, the output code corresponds to the highest-numbered active input line.

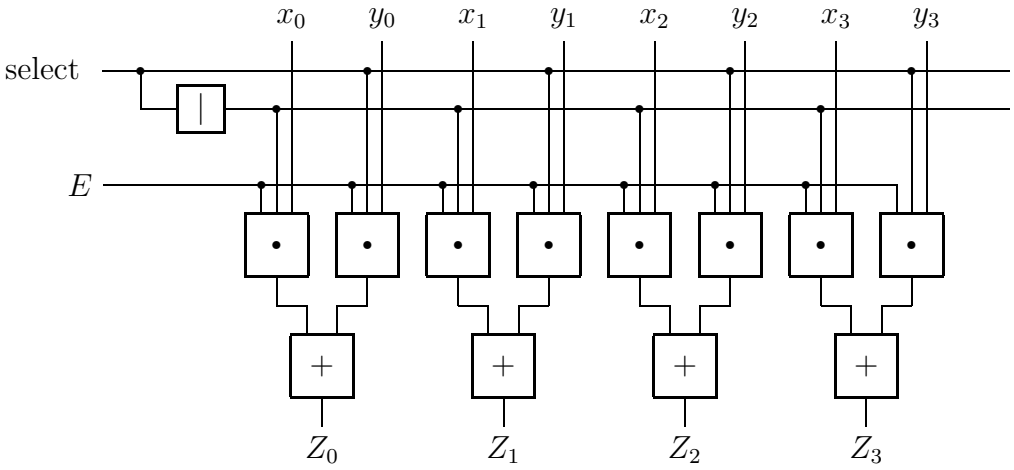
x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	Z_2	Z_1	Z_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

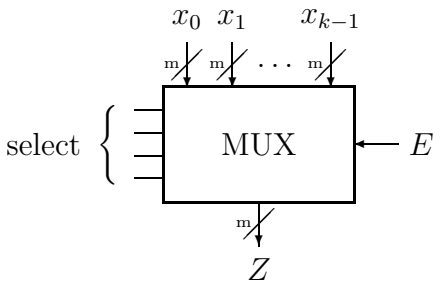




9.5 Multiplexer

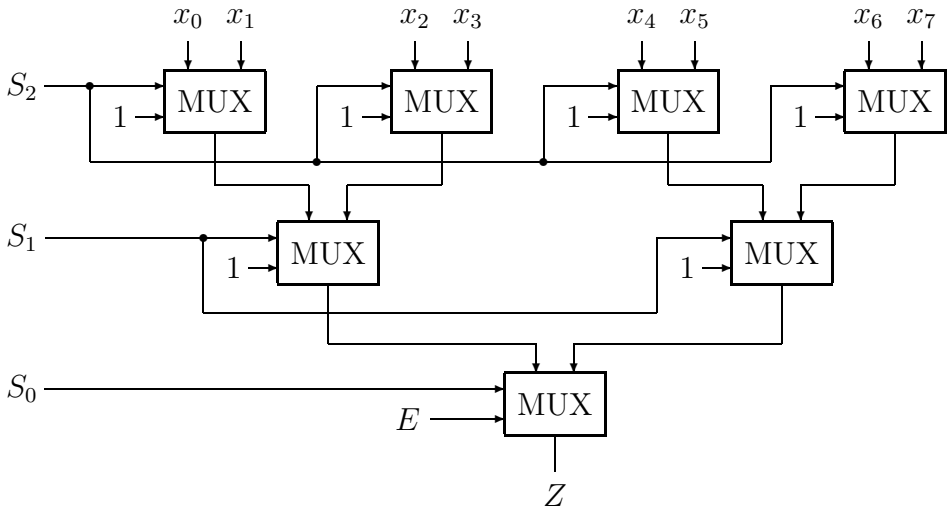
Purpose: to route data from one of several sources to a common destination.
The source is selected by applying appropriate selection signals. If the maximum number of data sources is k and each source has m bits, the multiplexer is called a k-input m-bit multiplexer. It is convenient to choose $k = 2^p$. It is convenient to make $k = 2^p$ such that the source is encoded by p bits.
A multiplexer can be used to generate any combinational function.





k-input m-bit multiplexer

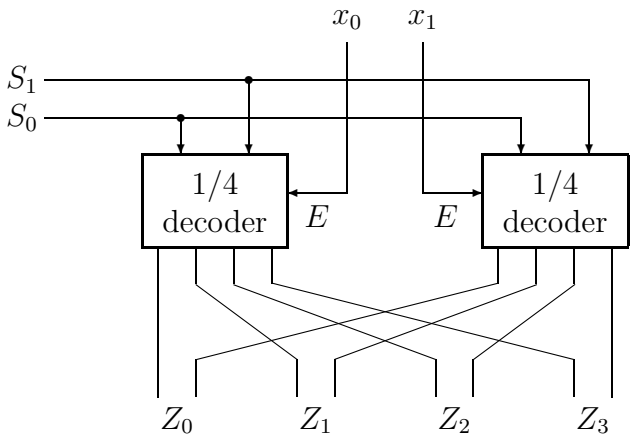
In the above symbol the input x_0 corresponds to the two signals x_0 and y_0, as shown in the circuit, for the case that $m = 2$. Multiplexers can be applied in a tree-like structure, e.g. we can construct an 8-input 1-bit multiplexer from seven 2-input 1-bit multiplexers:



In the above figure, the S_i are the *select* signals. In this circuit only the last multiplexers have been connected to the enable input; all others are always enabled (1, i.e. connected with a resistor to +5V). This has the advantage that the circuit which generates the enable signal needs to drive only one input.

9.6 Demultiplexer

Purpose: to route data from a common source to one of 2^n destinations. This is exactly the reverse function of a multiplexer.
This can be done by employing *decoders*: the enable (control) input of each decoder is a 1-bit source, the destination is the selection address applied to the decoder:



4-output
2-bit
demultiplexer

A k-output m-bit multiplexer can be constructed from m $1/k$ decoders. In general, $k = 2^p$.

10. ALU - Arithmetic and Logical Unit

With the present knowledge we will construct an ALU that can perform a limited number of functions.

10.1 Binary addition $a_k 2^k + b_k 2^k = (a_k + b_k)2^k$

a_k	b_k	S	
0	0	0	
0	1	1	
1	0	1	
1	1	1	+ 1 carry $(2 \cdot 2^k = 2^{k+1})$

The "1 carry" means that there must be a 1 from a neighbouring addition, i.e. from $a_{k-1}2^{k-1} + b_{k-1}2^{k-1}$:

c_{k-1}	a_k	b_k	S_k	C_k
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

S_k:

	a_k			
	0	1	0	1
c_{k-1}	1	0	1	0
			b_k	

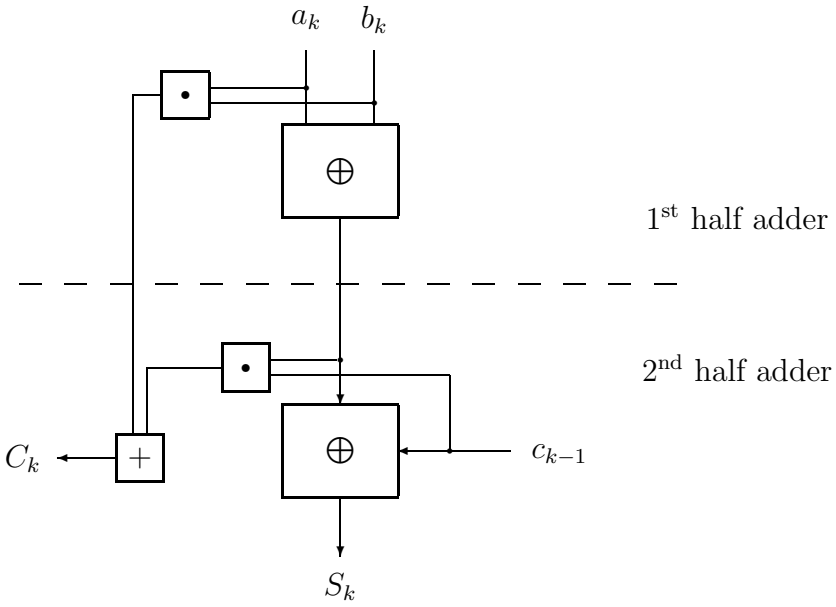
$$S_k = a_k \bigoplus b_k \bigoplus c_{k-1}$$

C_k:

	a_k			
	0	0	1	0
c_{k-1}	0	1	1	1
			b_k	

$$C_k = a_k b_k + a_k c_{k-1} + b_k c_{k-1}$$

From this, we could directly construct a circuit, using these Karnaugh maps, but usually an adder is constructed by connecting two half adders:



The function table can be divided into two parts:

a_k	b_k	S_k	C_k
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

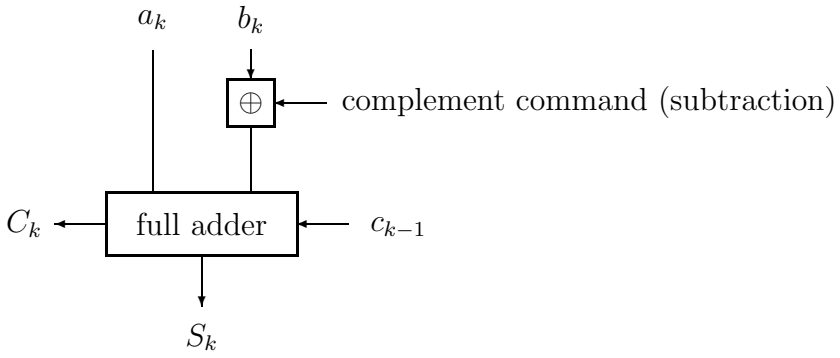
$S_k = a_k \oplus b_k$ = sub-sum
$C_k = a_k \cdot b_k$

S_k	c_{k-1}	S_k	C_k
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S_k = S_k \oplus c_{k-1}$ = output sum
$C_k = S_k \cdot c_{k-1}$

Both circuits are identical; they are called "half adder".
Together with an OR gate they form a "full adder".

- If we want to compute the <u>difference</u> between a_k and b_k (subtraction), we need to take the complement of b_k. A simple way is to use an extra exclusive-OR:



The adder needs a 1 on the least significant bit because the "2-complement code" equals the "1-complement code + 1."

10.2 High-speed adders

In chaining single-bit full adders together in order to sum words of n bits, the addition of the corresponding bits is fast but a carry generated at the least significant bits may cause a change up to the most significant bits. This is called *ripple-through* or *carry propagation.* To make the addition faster, we can compute the input carry at bit k directly from all inputs of preceding stages $k-1$, $k-2$, ... Such adders are called carry-lookahead adders.

To this purpose we compute the carry C_k at stage k by using two other functions: g_k = carry generate signal and p_k = carry propagate signal.

So for stage k these are defined as $g_k = a_k b_k$ and $p_k = a_k + b_k$.
The carry signal to be propagated to the next stage $k+1$ is defined by $C_k = g_k + p_k c_{k-1}$.
In other words: C_k is a combination of the carry generated at stage k and that propagated from stage $k-1$.
Substituting g_k and p_k in C_k, we obtain $C_k = a_k b_k + (a_k + b_k)c_{k-1} = a_k b_k + a_k c_{k-1} + b_k c_{k-1}$, which equals the expression obtained from the second Karnaugh diagram above.

Similarly, c_{k-1} can be expressed in terms of g_{k-1}, p_{k-1} and c_{k-2}:

$$c_{k-1} = g_{k-1} + p_{k-1} c_{k-2}$$

Substituting this in the previous equation, we get

$$C_k = g_k + p_k g_{k-1} + p_k p_{k-1} C_{k-2}$$

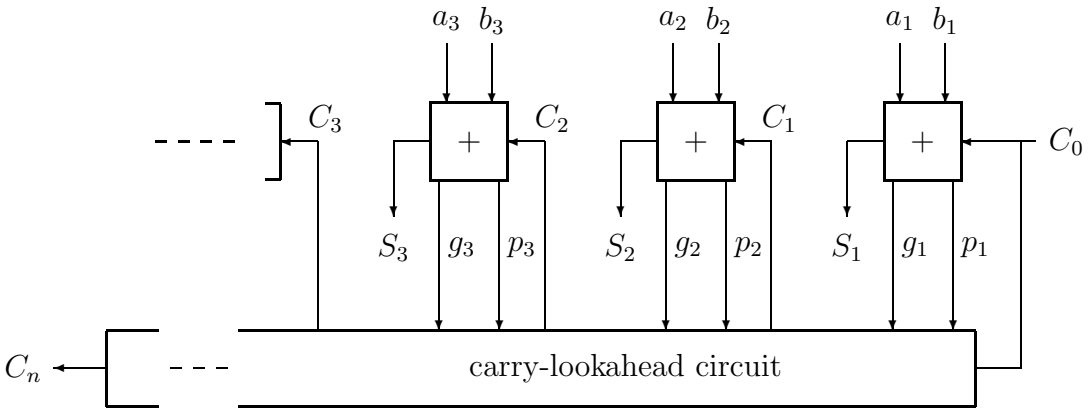
This can be repeated, so

$$C_1 = g_1 + p_1 C_0$$

$$C_2 = g_2 + p_2 g_1 + p_2 p_1 C_0$$

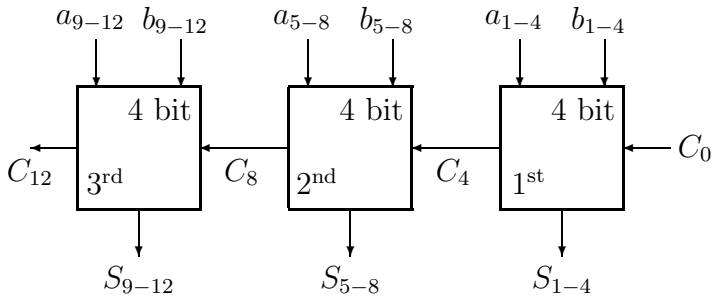
$$C_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 C_0$$

$$C_4 = g_4 + p_4 g_3 + p_4 p_3 g_2 + p_4 p_3 p_2 g_1 + p_4 p_3 p_2 p_1 C_0$$



The lookahead circuit is a two-level circuit consisting of ANDs and ORs. If the delay of a gate is d, the total delay of the carry lookahead is $\boxed{3d}$ (d for g and p plus AND plus OR).

The number of required gate inputs increases with n, the word length, so for practical reasons the word length is limited. For larger word lengths a number of limited carry-lookahead adders is then used:



4-bit carry-lookahead adders

For this 12-bit adder with group carry-lookahead and ripple carry between groups the delay is

$$\left.\begin{array}{lcl} 1^{\text{st}} & : & 3d \\ 2^{\text{nd}} & : & 2d \quad (p\text{s and } g\text{s already available, no delay}) \\ 3^{\text{rd}} & : & 2d \end{array}\right\} \boxed{7d}$$

For a ripple carry adder of 12 bits (12 full adders) this is in the worst case $12 \times 2 = \boxed{24d,}$ and 0 when no carries occur. Hence, on the average the delay will be $12d$.

$$\left.\begin{array}{ll} \text{Full carry} & 3d \\ \text{Group carry} & 7d \\ \text{Ripple carry} & 24d \end{array}\right\} \text{worst case}$$

10.3 Overflow

For a 2-complement adder we need an overflow detection. How can this be achieved? Suppose

$$\begin{array}{rcl} S_A & = & \text{sign of A} \\ S_B & = & \text{sign of B} \\ S_R & = & \text{sign of result} \\ C & = & \text{carry out} \end{array}$$

All bits (variables) are at the following positions:

	S_A	$\times$	$\times$	$\ldots$
	S_B	$\times$	$\times$	$\ldots$
C	S_R	$\times$	$\times$	$\ldots$

in which the $\times$ bits can be 0 or 1.

- First adding two positive numbers:

  If we look at one specific combination

  $S_A = S_B = 0$

	0	1
	0	1
0	1	0

  we see that the most significant bits have caused a change of the sign bit. This is impossible because the sum of two positive numbers must always be positive. Hence, OV=1!

S_A	S_B	S_R	C	OV		
0	0	0	0	0	$\longleftarrow$	new sign=0, no overflow
0	0	1	0	1	$\longleftarrow$	$S_R = 1$, so there must be an overflow because $S_A + S_B = 0$ is required

  The other combinations with $S_A = S_B = 0$ cannot occur and C can never be 1, so <u>0001 and 0011 are don't cares</u>.

- Now A positive and B negative:

S_A	S_B	S_R	C	OV				
0	1	0	1	0				
					$\longleftarrow$	$A \geq \lvert B \rvert$	$\rightarrow$	S_R becomes 0, but this can only be if a one is added and then we
0	1	1	0	0	$\longleftarrow$	$A > B$	$\rightarrow$	get $S_B + 1 = C$ (because $S_B = 1$) we get a negative number, sign remains 1, no carry

  0100 cannot occur (for $S_R = 0$, adding a 1 is required so $C = 1$).
  0111 cannot occur (S_B would have been added a 2; this is impossible because the carry from the adders to the right can only be ≤ 1).

S_A	S_B	S_R	C	OV		
1	0	1	0	0	$\lvert A \rvert > \lvert B \rvert$	How about don't cares here?
1	0	0	1	0	$\lvert A \rvert \leq \lvert B \rvert$	

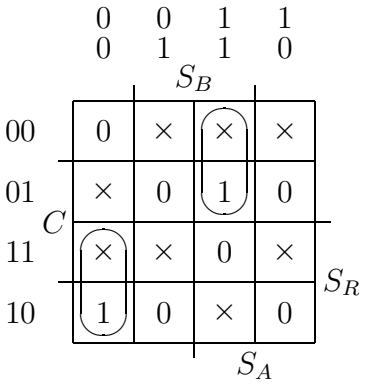
- For two negative numbers:

S_A	S_B	S_R	C	OV		
1	1	1	1	0	$\longleftarrow$	the sum is negative and delivers a carry because $S_A + S_B$ result in a carry
1	1	0	1	1	$\longleftarrow$	now there is no carry from the adders to the right, but this implies that there is an overflow: sum=positive

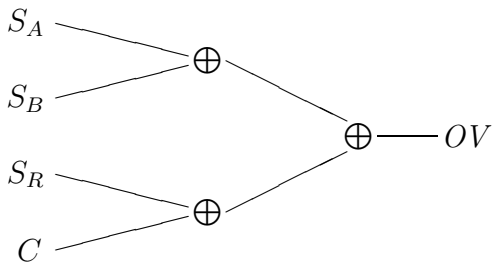
  The don't cares are 1100 and 1110, but the latter is impossible because $S_A = S_B = 1$ gave already a carry.

- Table for addition (without don't cares) and two solutions:

S_A	S_B	S_R	C	OV
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	1	0	0
1	0	0	1	0
1	1	1	1	0
1	1	0	1	1



$OV = S_A S_B \bar{S}_R + \bar{S}_A \bar{S}_B S_R$



	1		1
1		1	
	1		1
1		1	

$OV = S_A \oplus S_B \oplus S_R \oplus C$

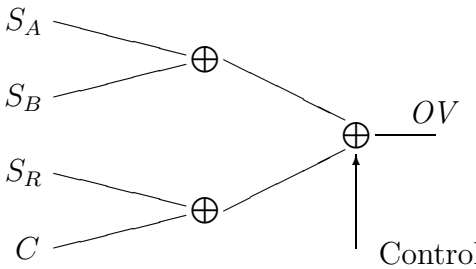
Subtraction, i.e. $A - B$, is obtained by inverting B and adding a 1 to the least significant bit. We obtain a similar table (*try this!*) where the S_B terms must be inverted.

S_A	S_B	S_R	C	OV	
0	0	0	1	0	$A \geq B$ } (B = positive)
0	0	1	0	0	$A < B$ }
0	1	0	0	0	$A + B < 2^{n-1}$
0	1	1	0	1	$A + B > 2^{n-1}$
1	0	1	1	0	
1	0	0	1	1	
1	1	1	0	0	
1	1	0	1	0	

However, this is the table for addition, but now with $\bar{S}_B$ instead of S_B, so

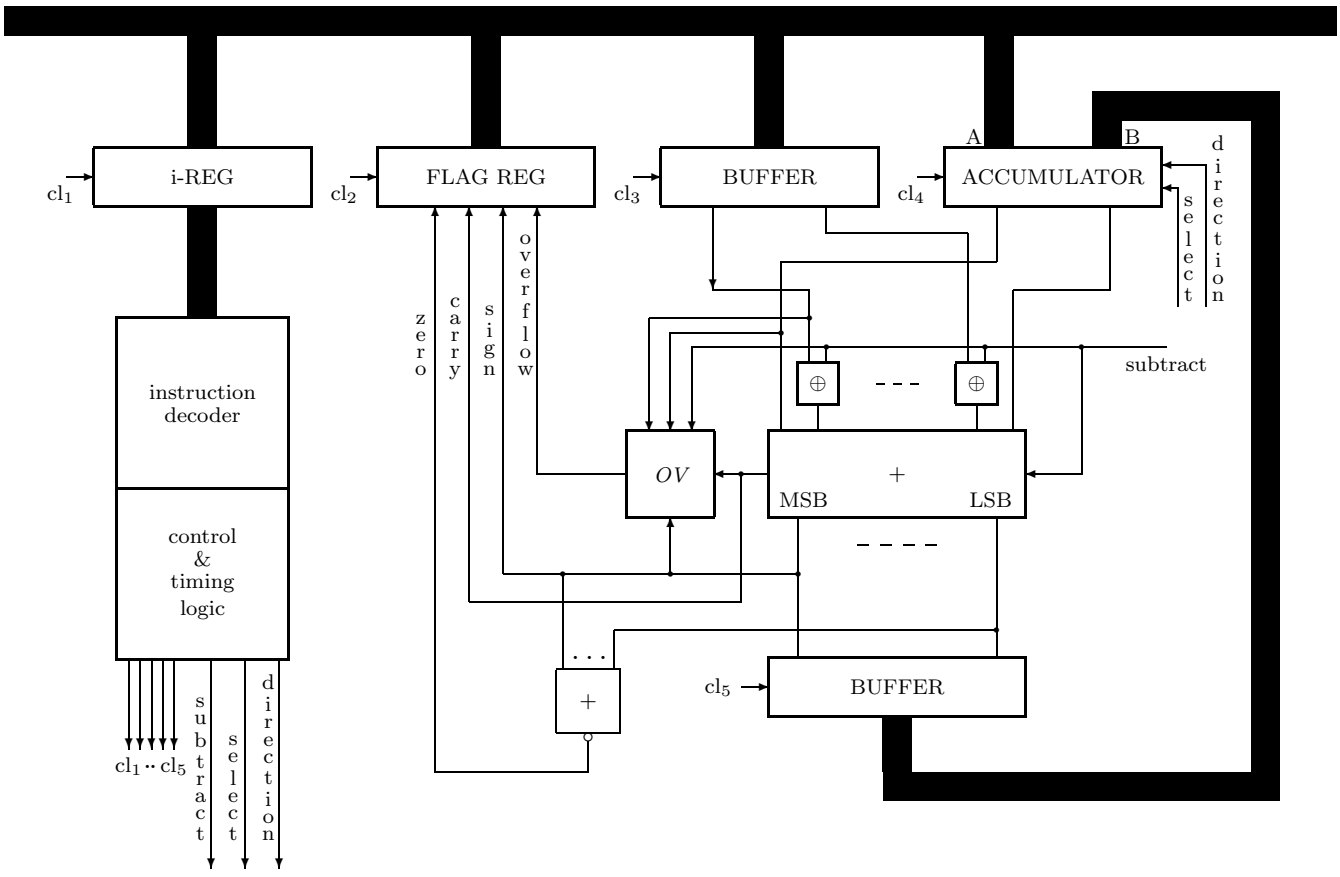
$$OV = S_A \oplus \bar{S}_B \oplus S_R \oplus C = S_A \oplus S_B \oplus S_R \oplus C \oplus 1$$

The circuit becomes:



(which must also generate an input carry for the least significant bit, 2-complement!)

10.4 Parts of a simple μprocessor ALU with + and −

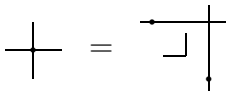


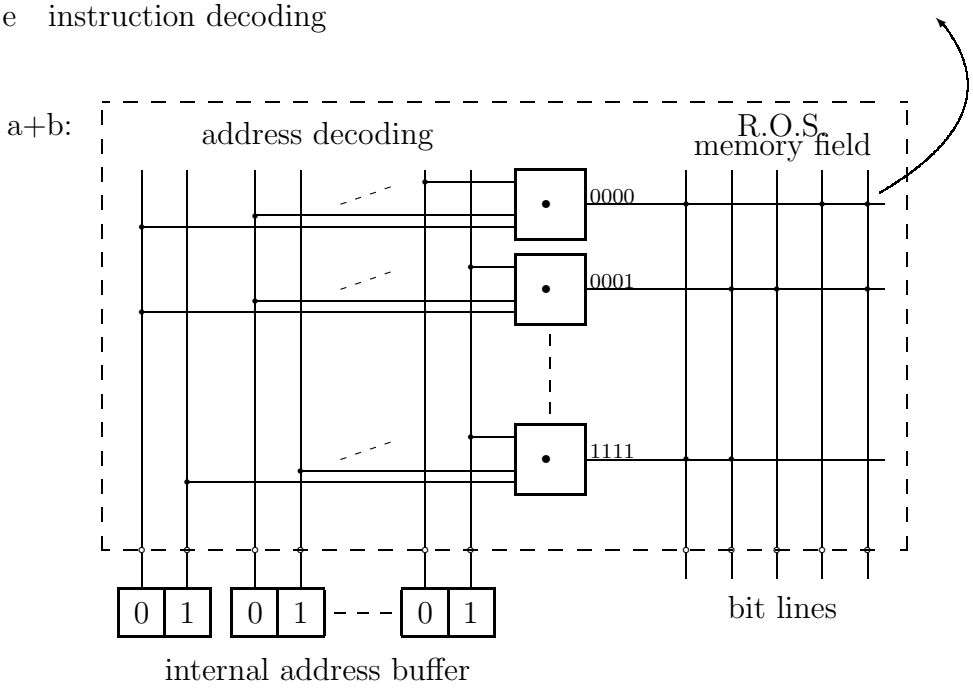
11. μ-programs

We have seen one function of a μ-computer, namely addition and subtraction in the ALU. This is done in hardware because of the speed. Other functions are multiply and shift, which can be used in combined forms (multiply = shift + add operations; note that all arithmetic functions can be done by using only the + function). We also but briefly discussed registers/buffers.

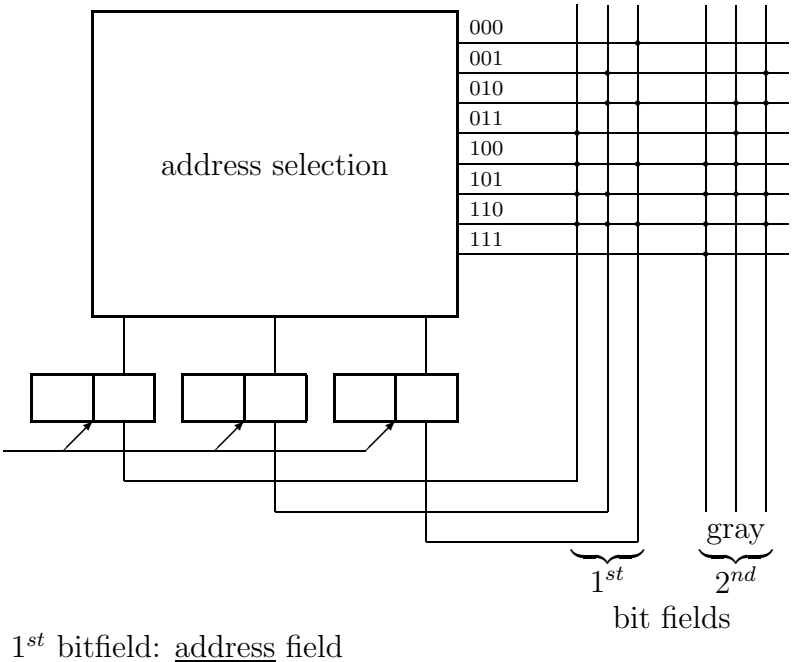
A μ-computer (processor itself, no peripherals) consists (or may consist) of only a few elements: ALU, registers, and control logic.
The control logic takes care of data flow and function commands. This is defined in so-called μ-programs which define the μ-orders for performing basic tasks.
Control logic consists of/does:

a non-volatile memory (ROS = read only store)
b address selection
c address modification
d address buffer
e instruction decoding





11.1 Applications of the addressing mechanism



- feedback of bitlines to address buffer generates a new address
- other bitlines can be used for generating a specific code (like gray-mirror code)

  NOTE We also could count directly in the gray code!

1^{st} bitfield: address field
2^{nd} bitfield: code field

An address field can be complete (all lines are used for address) or incomplete (some address lines may be stored in a register which can sometimes be changed)

- Here only complete address fields are considered for simplicity.

Depending on a function to be performed, a μ-program can be modified.
Example: multiplying words of 32 bits. The signs give four possibilities: $++$, $+-$, $-+$, $--$.
If both numbers are made positive first, the same multiplication method (μ-program routine) can be used in all 4 cases:
1) examine signs
2) multiply
3) put sign bit

In such a case, some parts of a μ-program can be skipped. This is done by jumping to another address. This is called "address modification."

One could use the two least significant bits for jumps.

Example	`JOHN`	~~`xxxxx xx`~~	the next address will
		`PETER 00`	be `PETER 00`

We can now change the two 0s (depending on the problem) such that the following addresses are

```
PETER 00 - no modification
PETER 01
PETER 10
PETER 11
```

On these 4 addresses, we can put a jump instruction or a jump command, such that the μ-program goes to a specific piece of code.

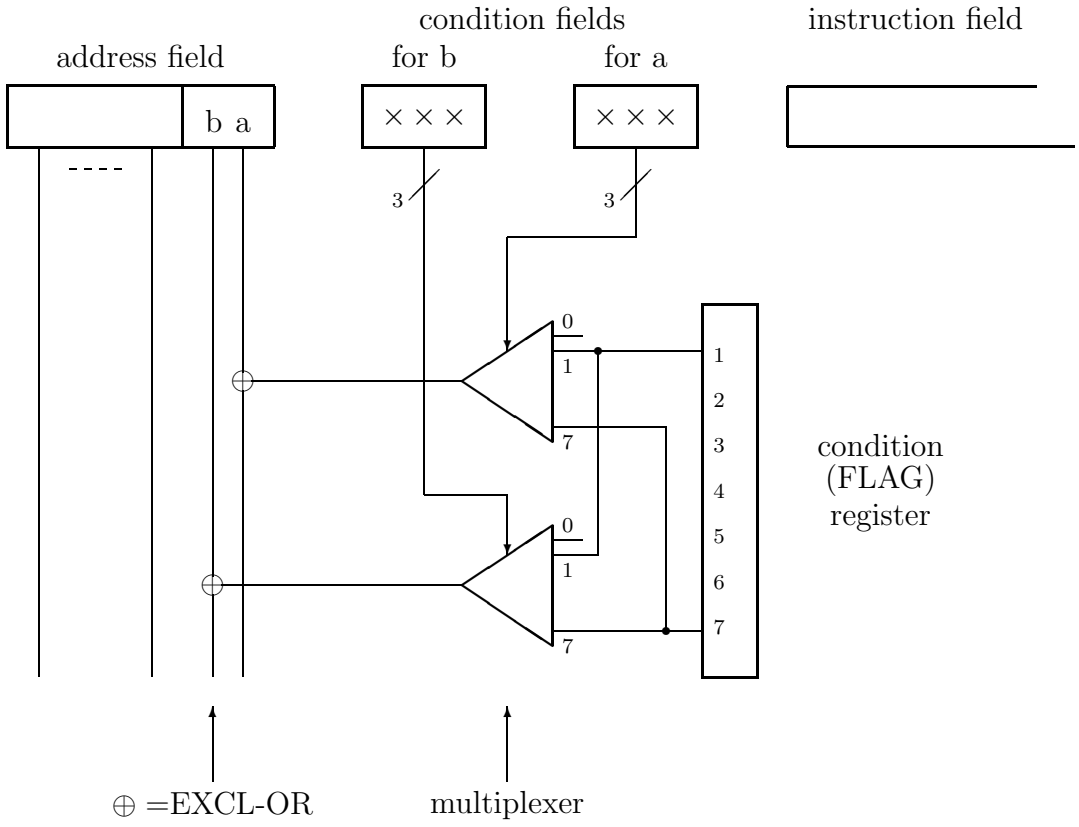
Example

LABEL	address	instruction
JOHN	PETER 00	
JOHN+1	⋮ rest μ-program	
	⋮ part	
PETER 00	MARY	
PETER 01	ANN	
MARY	MARY+1	
	+2	
	⋮	
MARY+m	JOHN+1	
ANN	ANN+1	
	+2	
	+3	
	⋮	
ANN+n	MARY	

Address modifications are often made on the basis of the contents of a condition register (or FLAG register)

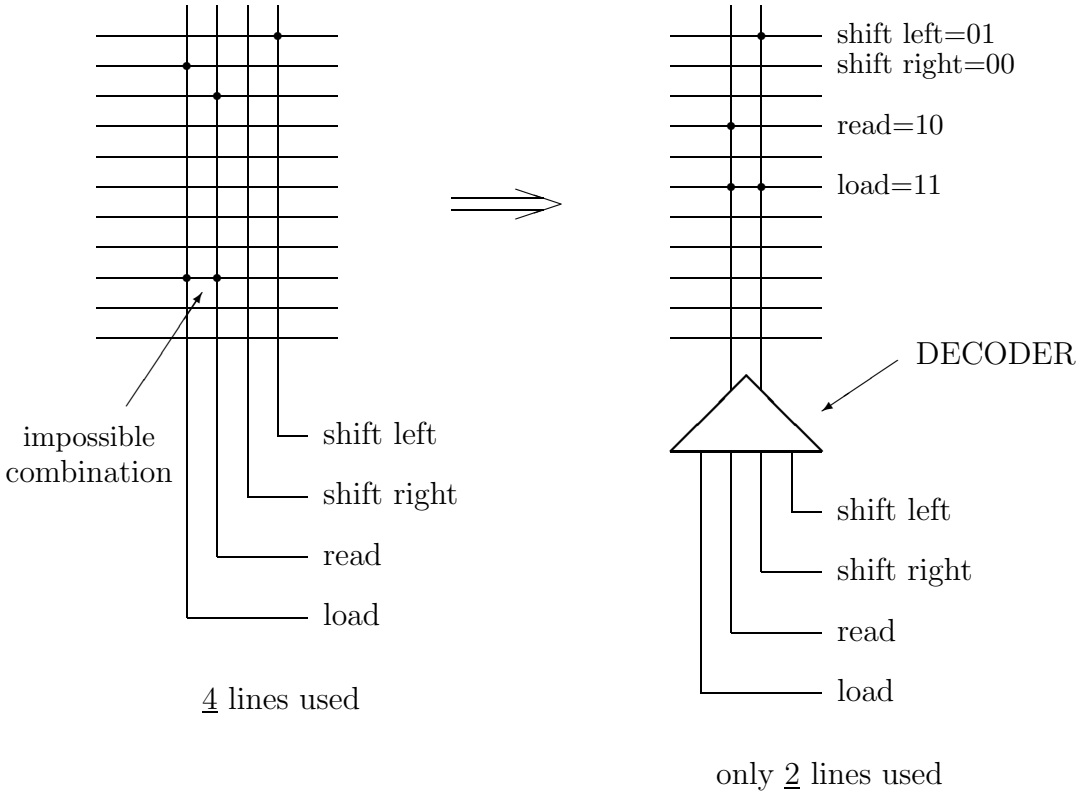
carry, sign, zero, parity, overflow — modified upon certain instructions like additions

A condition field is used to define which bit(s) of the condition register is (are) used in the address modification:



This instruction field has often groups of instructions.
Some instructions are mutually exclusive, like shift-left and shift-right.
Instructions can be coded; later on again decoded.

Example



- Designers collect mutually exclusive instructions. These are combined in specific instruction sub-fields. In this way, the total field becomes smaller and cheaper. However, to be added are decoding/demultiplexing costs, such that an economic minimum must be found.

12. Mass memory

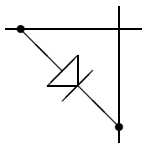
Any computer, from μ to super (vector or other architectures), contains from few registers to many hundreds. These registers store data temporarily, to transfer information between internal functional blocks.
To use a computer efficiently, having some sort of application, we need to store the program itself and data going into or coming out of the program. The principle may be the same (e.g. use of D flip-flops, ...), but now we need many "registers": mass memory. We will concentrate on so-called main memory. We will not discuss serial devices such as disks and tapes.
There are 4 items to be discussed: memory type, organisation, speed, and usage (read, write, address decoding).

12.1 Memory types

RAM random access memory - any word can be read/written.

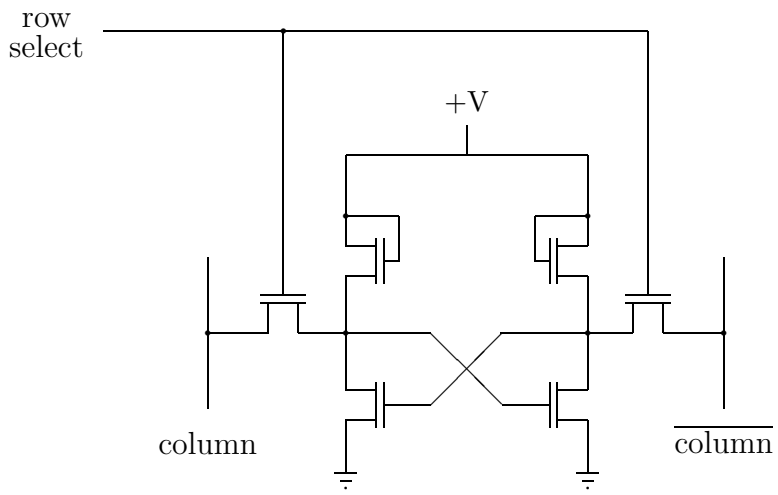
ROM read only memory - can only be read, so must be "programmed" in the factory (one can order these).

PROM programmable read only - can be programmed by the user

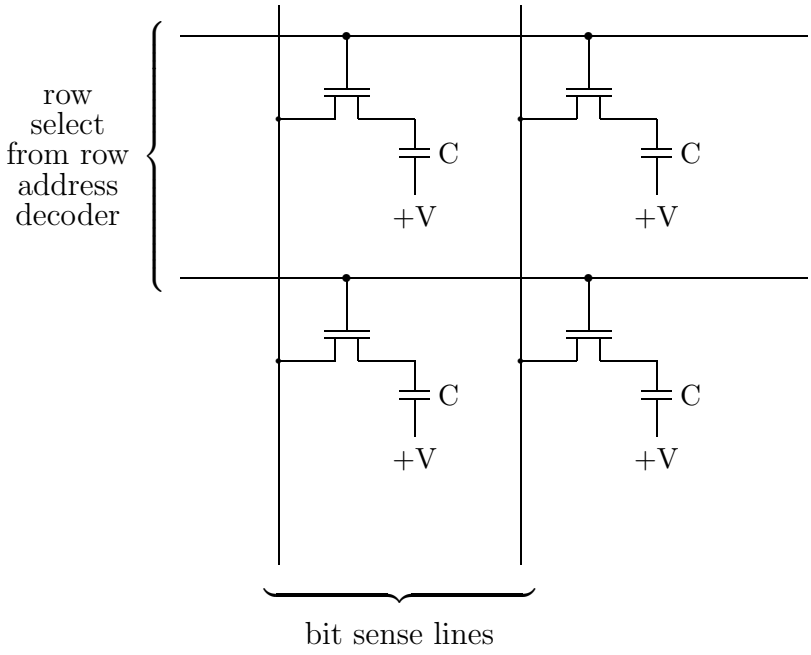


- fusible-link PROM: each matrix element (a diode in a line matrix) has a low current fuse that can be melted by applying a pulse which exceeds a critical current.
- EPROM: UV-light-erasable PROM is most common, can be reprogrammed many times. **Drawbacks**: chip must be removed from board to apply UV light; byte erasure is impossible - all information is lost; quartz windows make production more expensive.
- E^2PROM (electrically-erasable PROM): a newer development which eliminates the disadvantages of EPROMs.
  NOTE: for all these types special circuits are required.

STATIC RAM - each cell consists of a simple FF like a SR one, so each bit requires 4–6 transistors. Each bit is accessed by a row and column select signal, generated by the address decoder. It looks like:



DYNAMIC RAM - each cell consists of a simple select MOS (metal oxide semiconductor) transistor plus a small capacitor:



Advantage: very high density; more bits/chip than static RAM's.
Disadvantage: C leaks, so periodically each bit must be refreshed and this every 2 ms!

Refreshing is not difficult: each column has its own sense amplifier and this can be used to recharge a cell in a given row. Row-select sequencing is applied: first select row 1, then row 2, and so on. Because of the extra circuitry required, special chips were

made to do this job (DRAM controllers). Furthermore, the chip layout is designed such that different blocks can be accessed at the same time, for example in 128 × 128 bit-cell quadrants with common row addresses. But nowadays all necessary circuitry is integrated in the memory chips

Static and dynamic RAM are both volatile. When the power is switched off, all data is lost. CMOS static RAMs consume very little power in "stand-by" mode. In some systems CMOS-SRAM is used in combination with a battery (battery-backed memory) in order to keep all information while the computer is switched off. For IBM compatibles one can buy such systems as a plug-in board (which are even made such that they behave as a hard disk - one can boot from a memory board!).

12.2 Memory organisation

This refers to the number of bits in an input/output word. Important is the total number of bits/chip, but also the word length. Most static RAMs are available with a word length of 8 bits, so a data bus and all bus drivers, etc. may also be 8 bits wide. If a 16-bit μ-processor is used (an 8086 from Intel© for example), two "banks" of 8 bits must be used in parallel, one for the high-order byte and one for the low-order byte (well, not really - there are controllers that can transfer a 16-bit processor into a machine that accesses 8-bit wide memory).

Because of the 8-bit wide word length, less IC pins are available for the address lines. This is not a problem for STATIC RAMs because they have a smaller capacity relative to dynamic RAMs.
For dynamic RAMs, the capacity is higher and the refresh must be dealt with. For these reasons the number of address lines is greater so that, for the same number of IC pins of a static RAM, fewer data pins are used. Often only 1! This means that an 8-bit system requires 8 DRAM chips in parallel.

In summary, for static RAMs organisations of 2K×8 to 64K×8 are common, whereas dynamic RAMs organisations of 16K×1 to 4M×1 are also found.
However, there exist DRAMs of e.g. 16K×8 which have all refresh control logic "on board". EPROMs have always 8 bit words because of the required external programming equipment (which is standard...).

12.3 Memory speed

There is a tendency towards larger capacity as well as speed, both for SRAMs and DRAMs. SRAM design is concerned with capacity and speed, whereas DRAM design is mainly concerned with capacity.
For this reason, the capacity of DRAMs is about 16× the capacity of SRAMs and the speed of SRAMs is 8–16× the speed of DRAMs. For fast applications SRAMs are always used.

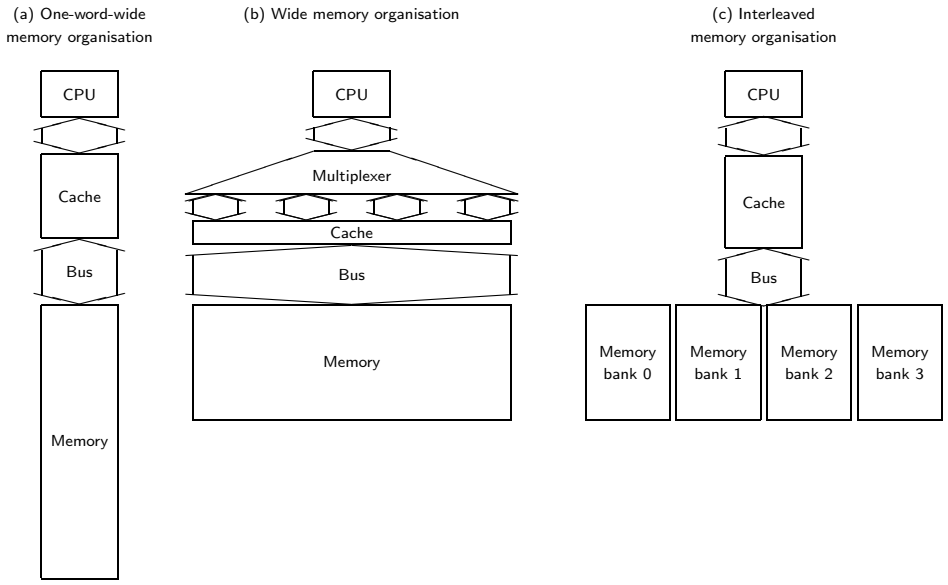
There are additional techniques to speed-up performance of a system, apart from selecting SRAMs instead of DRAMs: caching and interleaving.

Caching means that a part of a program/data in memory is placed in a special buffer; the same principle as virtual memory (a page can be on disk or in main memory).

A few typical numbers:

	CPU registers	cache	main memory	disk
typical size	< 1Kb	< 512Kb	< 512Mb	> 100Mb
access time (ns)	10	20	100	10–20,000,000
bandwidth (Mb/sec)	800	200	133	4
controlled by	compiler	hardware	operating system	op-syst/hdware

Interleaving means that memory chips are organised in banks which are 1-word wide, but the banks are addressed in parallel, such that multiple words can be read/written at a higher speed.



In this figure:

(a) the same word length for CPU, cache and memory. The fast cache allows fast transfers, but transfers between cache and memory are relatively slow.

(b) the cache and memory are broader than the CPU word length. Because of the cache and memory width, the effective memory transfer is faster.

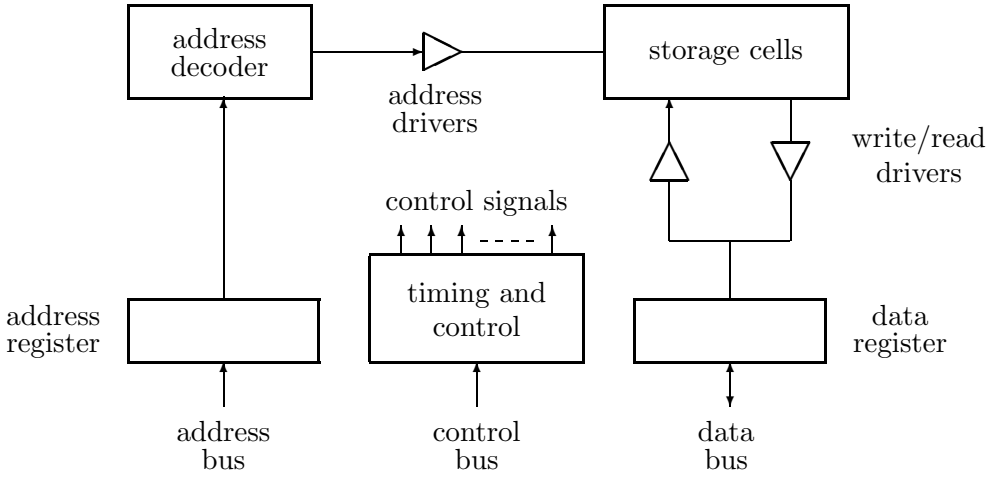
(c) Interleaving: 4 words can be read/written from/to memory at a higher speed.

(b) and (c) are more or less equivalent: special hardware is required such as a fast multiplexer and a bank-select mechanism. This is done by generating extra address codes (b) or by using the two least significant address bits for selecting the memory bank (c). Address decoding is discussed below.

Caching and interleaving requires optimisation: if the CPU wants to access code/data which is not in cache memory, the specific part containing the code/data must be loaded from main memory. The question is then how big the parts must be.

12.4 Memory usage (read, write, address decoding)

Memory chips are functional blocks which can be seen as a black-box, the user does not need to know how exactly it is organised. A simplified scheme generally looks like this for a RAM chip:

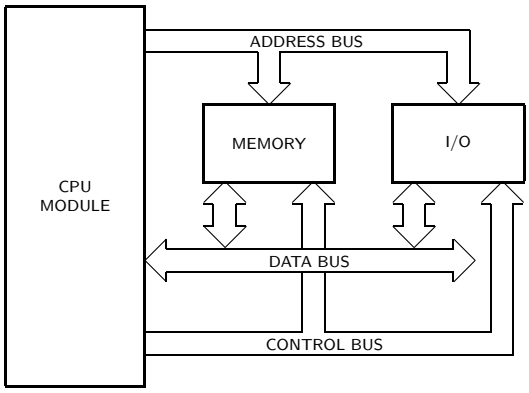


Data bus: a number of lines on which data is transferred into or output of the memory chip.

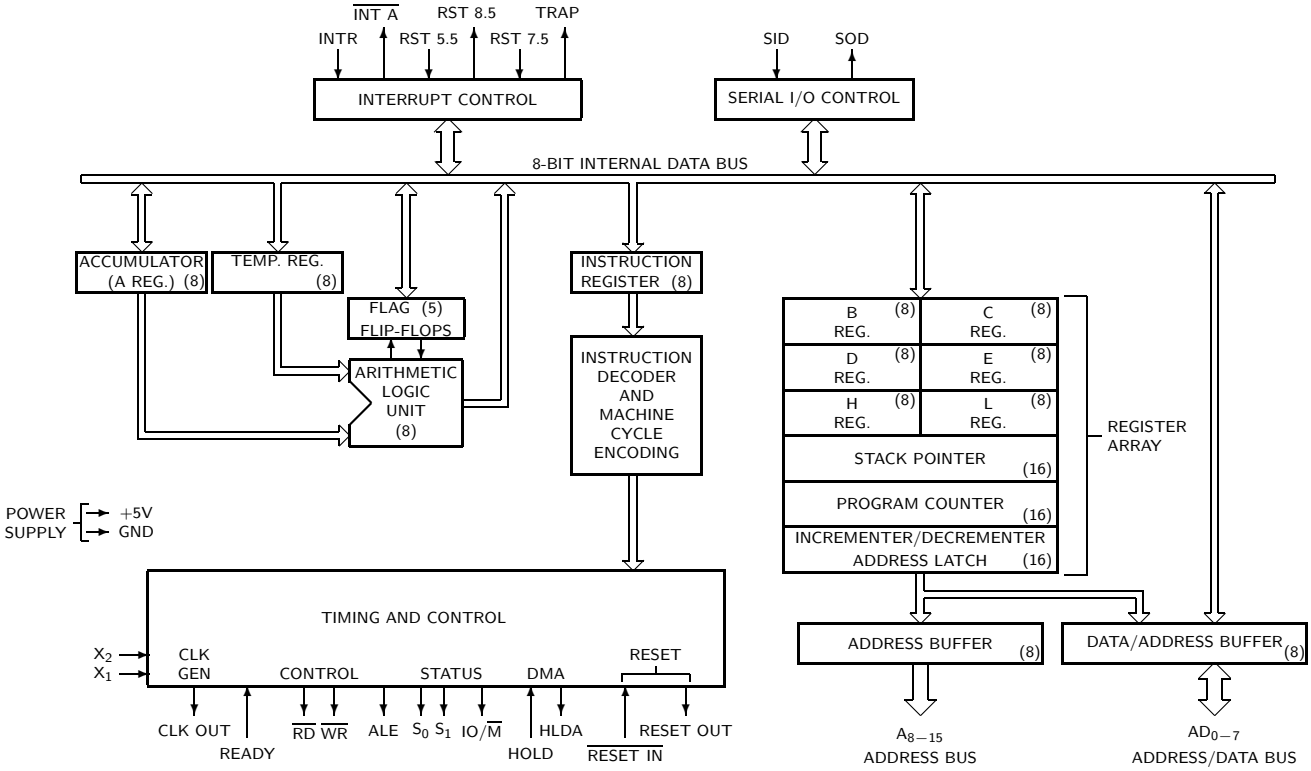
Address bus: a number of lines which give the encoded address of a data word in the storage cells.

Control bus: a few lines which indicate what has to be done, generally read, write, output enable, and chip select.

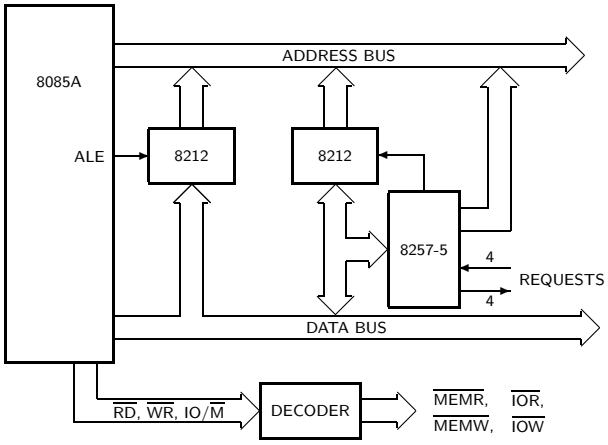
On the other hand, a CPU is also a functional block which uses data, address, and control buses to communicate with memory and peripherals:



An example is the Intel 8085A processor:



For now we are only interested in a few control signals and the data and address buses:



The 8257-5 is a DMA controller (direct memory access); this will not be explained in this course.

The 8085A is an 8-bit processor, so the data bus is 8-bits wide. Because of the large amount of other control signals, not all 16 data addressing bits could be assigned an output pin. This was solved by multiplexing the data bus and the lower 8 address bits.

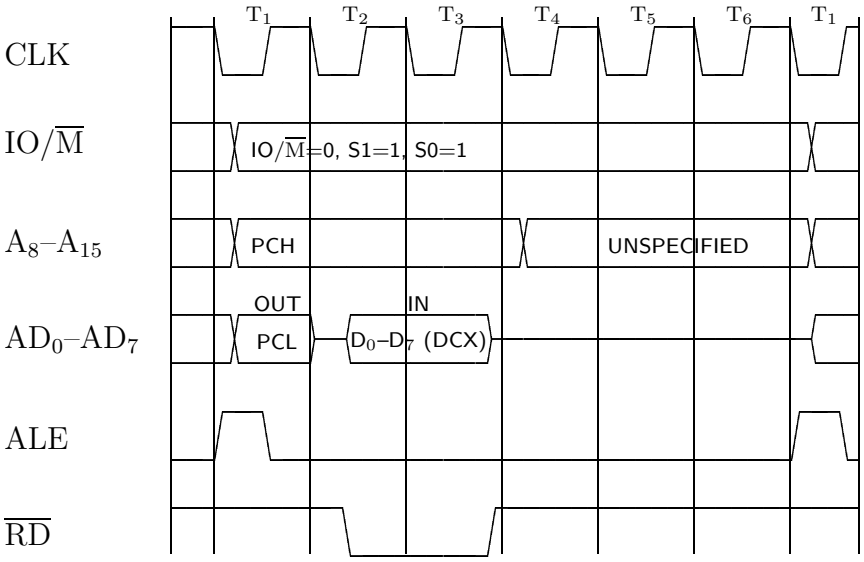
So, AD_{0-7} are combined address and data lines and A_{8-15} are address lines. Before a word in memory can be read or written, the CPU puts the low address byte (A_{0-7}) onto the AD_{0-7} lines and generates then a special signal such that the low-order address byte can be clocked into an 8-bit register or latch. In the previous figure, the latch is indicated

by the block called "8212" (this is one chip) and the special control signal to store the low-order address byte is ALE (address latch enable).

The other signals that we are interested in are $\overline{\text{RD}}$, $\overline{\text{WR}}$ and IO/$\overline{\text{M}}$.

- IO/$\overline{\text{M}}$ indicates whether transfer is going to be to memory (IO/$\overline{\text{M}}$=0) or to other input/output devices (IO/$\overline{\text{M}}$=1).
- $\overline{\text{RD}}$ is a read signal, so if the CPU wants to read it makes $\overline{\text{RD}}$ low
- $\overline{\text{WR}}$ is a write signal, so if the CPU wants to write it makes $\overline{\text{WR}}$ low.

Normally, a control bus has slightly different control signals: $\overline{\text{MEMR}}$, $\overline{\text{MEMW}}$, $\overline{\text{IOR}}$ and $\overline{\text{IOW}}$. We can use a very simple decoder to generate these signals.
How does it work? Let us take a look at a memory read cycle (in this case of an OF - opcode fetch).



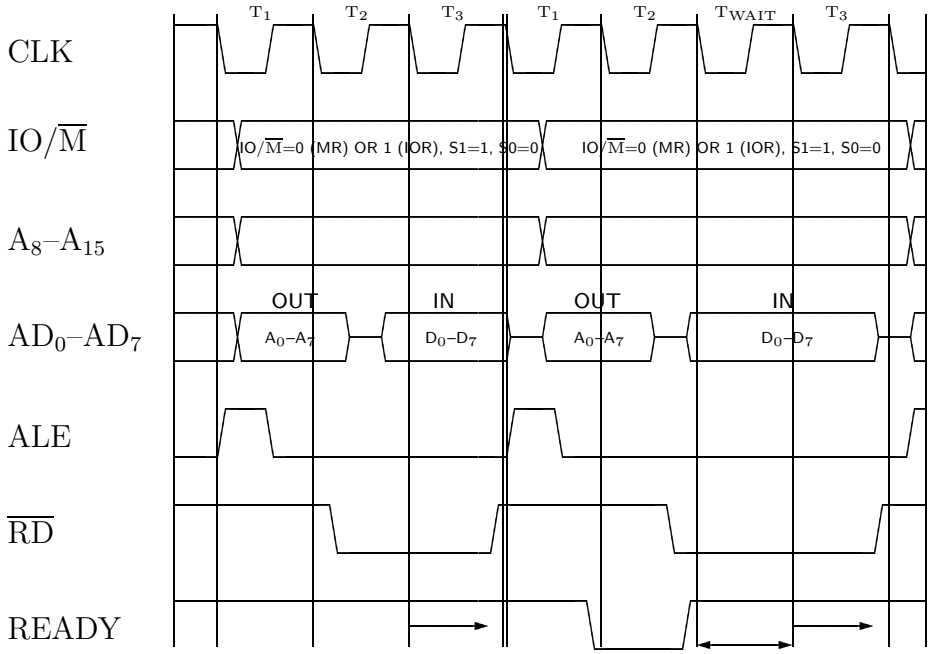
PCH=program counter high
PCL=program counter low

CLK is a periodic system clock and a memory read cycle is defined by a number of states T. Reading a data byte from memory takes normally (if the memory is fast enough) three states: T_1, T_2 and T_3.
In the figure above an OF (opcode fetch) is shown. This is an instruction which takes 6 states: T_1, T_2 and T_3 are concerned with reading an instruction from memory, whereas T_4, T_5 and T_6 are concerned with the interpretation (decoding) of the opcode.

- In T_1, IO/$\overline{\text{M}}$ goes low (memory access), A_8–A_{15} are put on the bus, AD_0–AD_7 which contains the low-order address byte as well and the negative flank of the ALE signal can be used to clock this byte into a latch. After T_1, the address bus is complete!

- In T_2, the $\overline{RD}$ signal goes low, which means that all data bus buffers or transceivers must be directed toward the CPU and the memory must provide the data word with the address indicated.
- In T_3, the $\overline{RD}$ signal goes high, which means that the CPU has read the word into an internal register and the read cycle is finished.

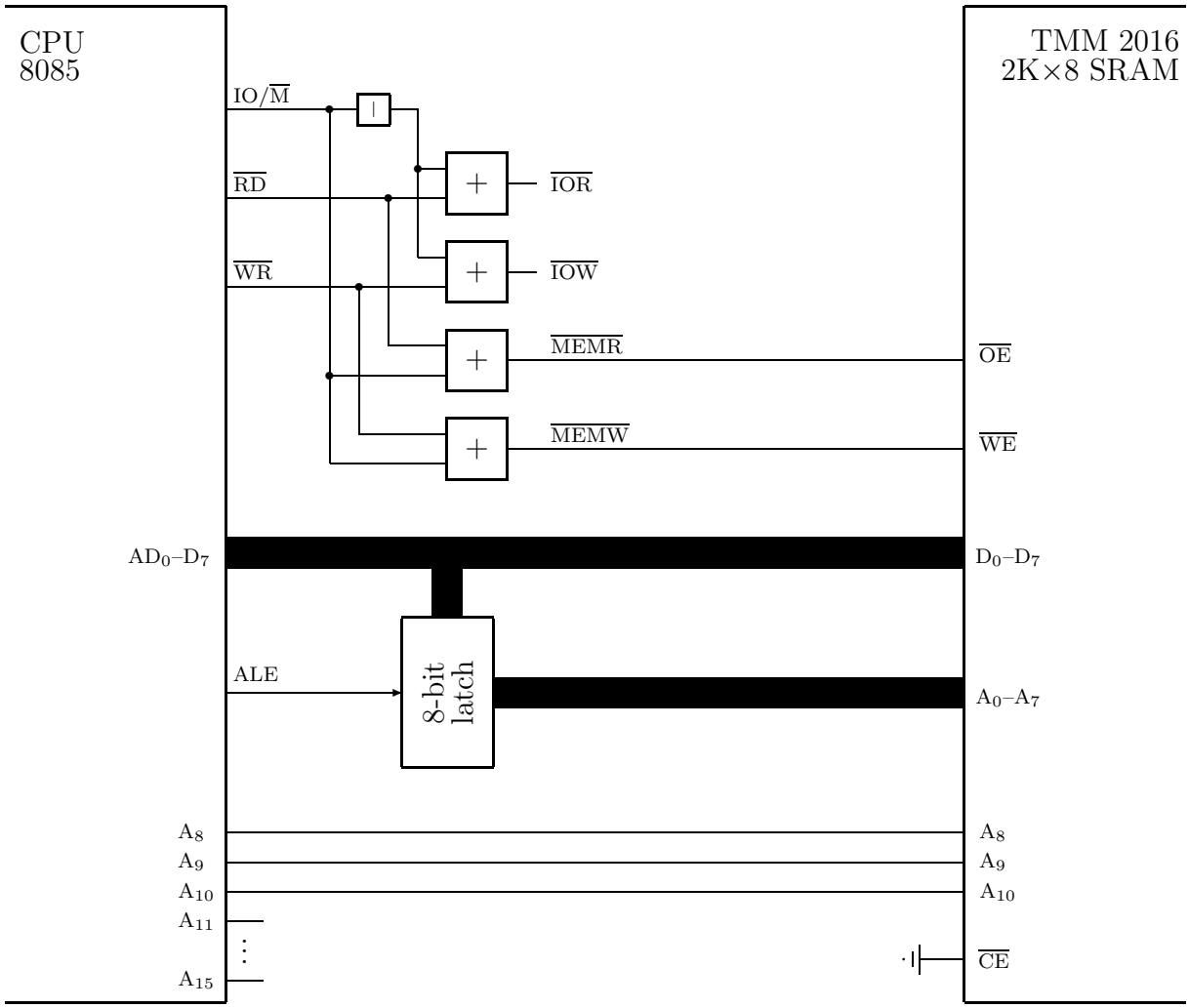
A normal memory (data) read cycle requires only 3 states T_1, T_2 and T_3 (see below left). If the memory is slow or there are many buffers in a system (CPU on a CPU board with its own data/address buffers, memory on another board with its own buffers, ...), the memory board must generate a "ready" signal in order to indicate that the CPU must wait an extra cycle before clocking-in the data on the data bus: memory with 1 wait state.



NOTE: the 8086 processor, which will be discussed in detail in the next semester, is a 16-bit processor but it uses the same principle of address-data multiplexing with an ALE signal.

12.5 CPU-memory interfacing

In this section we will briefly discuss a few examples and explain a few address decoding methods.
The simplest interface is shown below:



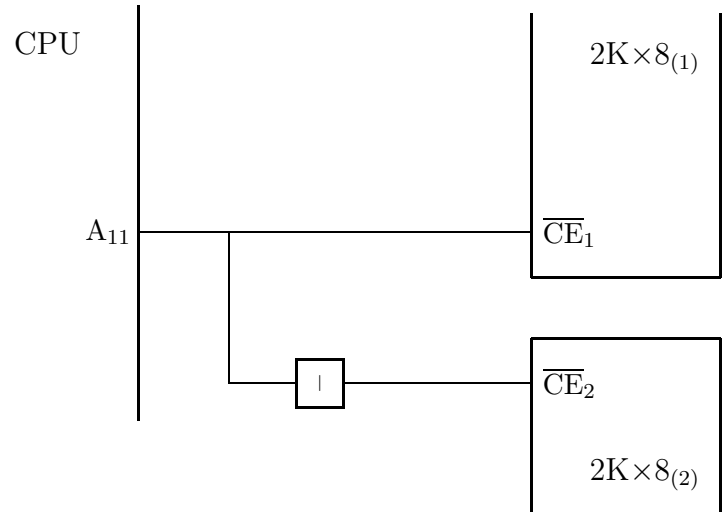
NOTES:

- $\overline{\text{MEMR}}$ goes to $\overline{\text{OE}}$ (=output enable).
- $\overline{\text{MEMW}}$ goes to $\overline{\text{WE}}$ (=write enable).
- $\overline{\text{IOR}}$ and $\overline{\text{IOW}}$ are not used, so $\overline{\text{RD}}$ and $\overline{\text{WR}}$ could be used but such a system cannot communicate with peripherals.
- 2K words means 10 address lines A_0–A_{10}. In this minimum system the entire address decoding is done by the 2016.
- A_{11}–A_{15} are not used at all (the programmer must be sure not to include jump instructions beyond 2K).
- the $\overline{\text{CE}}$ (chip enable) input of the 2016 is always low.

Of course, such a minimum system is not realistic unless we need to design a small controller board for some application, but even then the use of only RAM means *no program to execute*! At least a PROM or EPROM should be added. For other applications we will need to use more RAM and to combine more memory chips.

12.6 Address decoding

In the above example we used a 2K RAM at addresses from 00000–003FFH (hexadecimal) or 0-2047D (decimal). We could use A_{11} to A_{15} to put this memory block at higher addresses. Or, to add a second chip in order to increase the memory from 2K to 4K, occupying 00000–004FFH. To do this, we need to exploit address bit A_{11}:

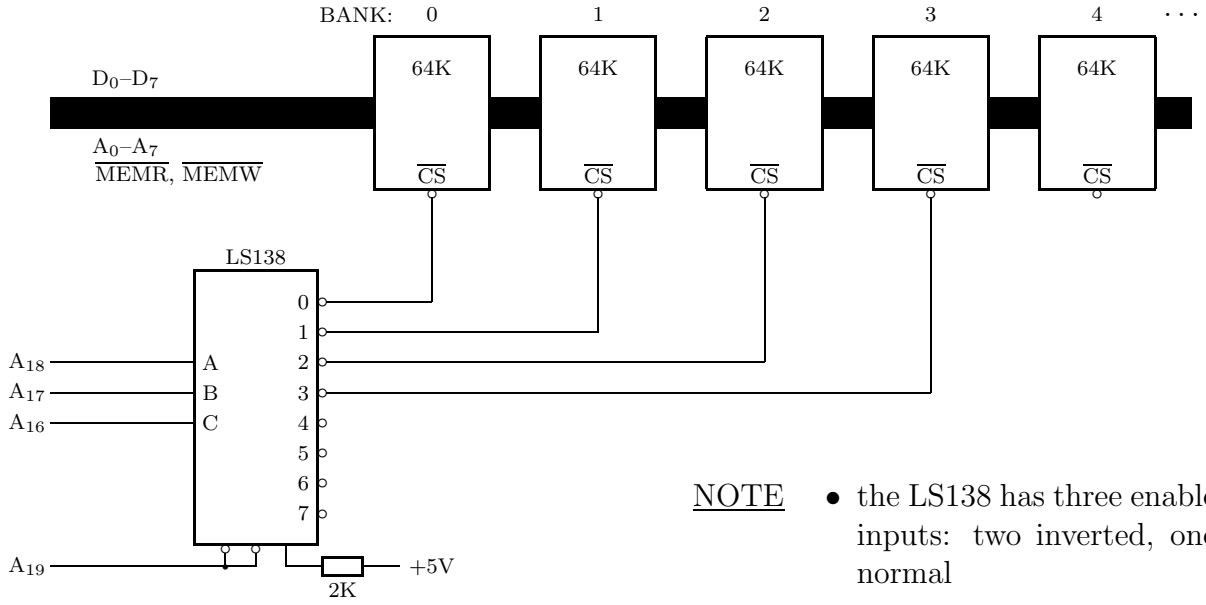


NOTES:
- all other lines go to both chip 1 and chip 2, to the same inputs and outputs.
- A_{11} to $\overline{CE}_1$ means chip 1 contains addresses 00000-003FFH.
  $\overline{A_{11}}$ to $\overline{CE}_2$ means chip 2 contains addresses 00400-004FFH.

  In other words: A_{11} has been decoded in order to address more memory and to address more memory chips.

Again, this example is not quite realistic because we still need more memory! So we also need to use A_{12} and A_{13} and... and each chip needs its specific $\overline{CE}$ signal in the memory space that it will occupy: address decoding.
Instead of using simple circuits consisting of a few gates, special circuits are then employed.

Example
- use 64K chips (64K requires 16 address lines: A_0–A_{15}.
  Hence, A_{16} and A_{17} must be decoded to use 4×64K
- use a 74LS138 (3-to-8 line decoder)



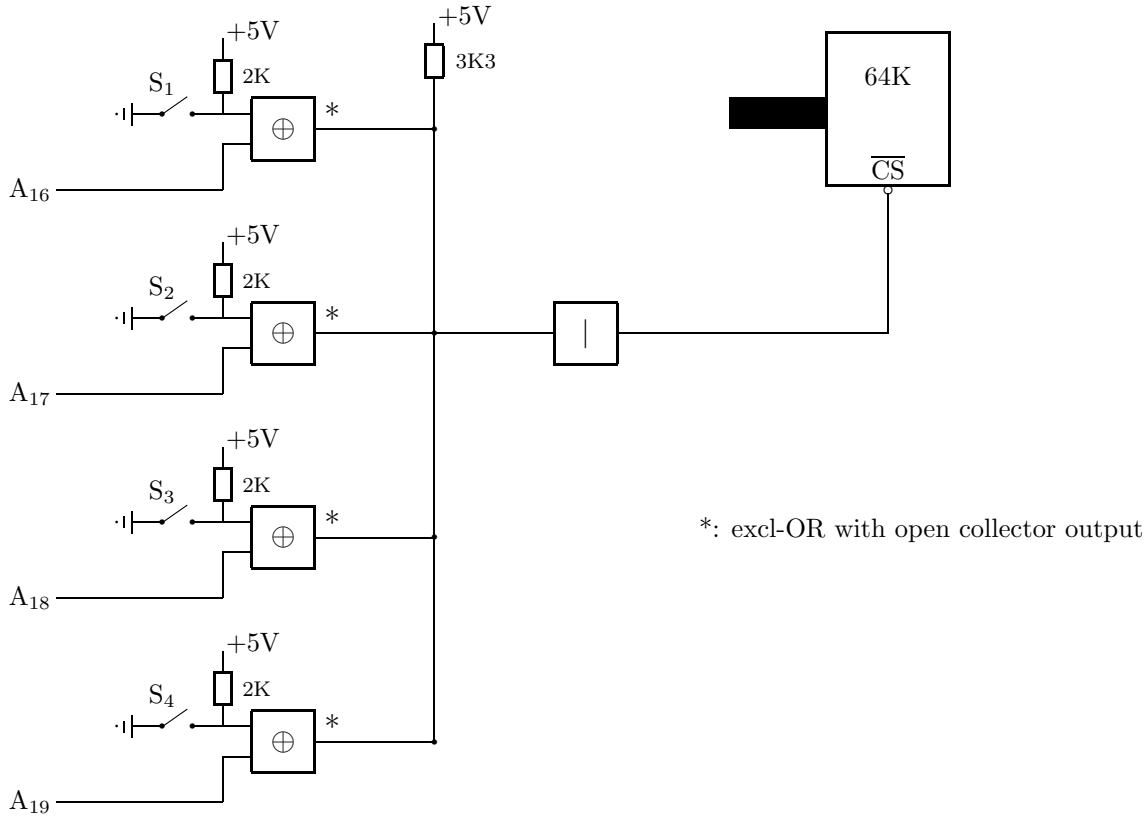
NOTE

- the LS138 has three enable inputs: two inverted, one normal
- the 8086 has 20 address lines, so A_{19} is the highest

Bank 0: 00000–0FFFFH
Bank 1: 10000–1FFFFH
Bank 2: 20000–2FFFFH
⋮
Bank 7: 70000–7FFFFH

Using one LS138 and 64K chips without using A_{19} allows to address 80000H words (512Kb). Using also A_{19}, the 8086 can address 1Mb at most.

Sometimes we may need to put a chip in a specific memory range: for example when we make a PC board which contains battery-backed CMOS SRAMS (memory chips with a very low power consumption; a small battery allows to keep the information when the computer is shut down). If we want to put this board in different PCs with certain amounts of memory, we might adapt the board address space to the host address space. One simple way to do so is by using exclusive-ORs:



When all four switches are open, A_{16}–A_{19} must equal 0000 to cause all four excl-OR open collector outputs to be high, i.e. $\overline{CS}$ becomes 1: disabled. This decoder (actually comparator) allows to put the 64K block to any of the 16 possible 64K slots.

To put the block in the space beginning at C0000H=$\overbrace{1100}^{\text{CH}}.\overbrace{0000}^{\text{0H}}\ldots$ $S_4S_3S_2S_1$ must be 1100 (only S_4 and S_3 closed).

Instead of using separate switches and exclusive-ORs we can use "dip-switches" and a comparator.