University Bremerhaven

Course Worksheets

Digital Systems / Microprocessors
[PR–DIGSM]

- Part 1: Digital Systems Fundamentals
- Part 2: Dual Numbers and Arithmetics
- Part 3: Boolean Logic
- Part 4: Sequential Systems / State Machines
- Part 5: Microprocessor Architectures
- Part 6: Microprocessor Programming

Revision: V1.1 (2nd doc revision)

Release: November 2020

Prof. Dr.-Ing. Kai Mueller
University of Applied Sciences Bremerhaven
Institute for Automation and Electrical Engineering
An der Karlstadt 8

D–27568 Bremerhaven

Phone: +49 471 48 23 – 415
FAX: +49 471 48 23 – 555

Email: kmueller@hs–bremerhaven.de

I Introduction

I.I Course Documentation

See < http://www1.hs–bremerhaven.de/kmueller/> for updates.

I.II Digital Systems / Microprocessors

In billions of devices digital systems are almost everywhere. They have become part of many consumer products, industrial and scientific devices. Since the prices for digital hardware dropped dramatically over the years todays engineers can create almost everything that can be imagined.

This course teaches to design digital systems including microprocessors.

Bremerhaven, October 2014

Kai Müller
<kmueller@hs–bremerhaven.de>
Tel: (0471) 4823 – 415

II Contents

Section 1

1 Analog and Digital Systems

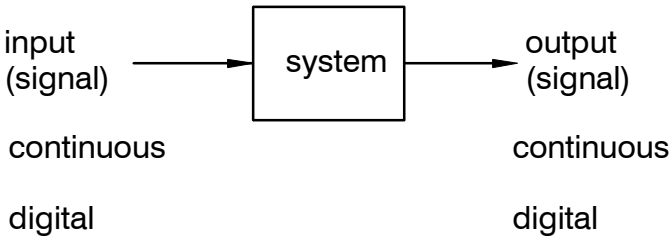


Figure 1.1: System (continuous or digital)

2 Number Systems

Polyadic numbers (positive) with *base* **B**.

$$n = \sum_{i=0}^{N-1} b_i B^i = b_N B^N + b_{N-1} B^{N-1} + \ldots + b_2 B^2 + b_1 B + b_0 . \qquad (1.1)$$

N is the *word size* of a number.

Horner scheme: $n = \Big(\Big(\ldots\Big(\big(b_{N-1} B + b_{N-2}\big)B + b_{N-3}\Big)B \ldots + b_2\Big)B + b_1\Big)B + b_0$

Example:

$$(1\,1\,1\,0\,1\,0\,1)_2 = 2^6 + 2^5 + 2^4 + 2^2 + 1 = (117)_{10}$$

2.1 Binary-, Octal- and Hex Numbers

Bit (binary digit) $b_i \in \{0, 1\}$

2.1.1 Binary Numbers

Base $B = 2$
Digits: $b_i \in \{0, 1\}$

Example:

$$(1\ 1\ 1\ 0\ 1\ 0\ 1)_2 = 2^6 + 2^5 + 2^4 + 2^2 + 1 = (117)_{10}$$

The rightmost bit is called ***LSB*** (least significant bit).

The leftmost bit is called ***MSB*** (most significant bit).

2.1.2 Octal Numbers

Base $B = 8$
digits: $b_i \in \{0, 1, \ldots, 7\}$

Example: number 117_{10}

$$165_8 = 1\ 8^2 + 6\ 8 + 5 = 64 + 48 + 5 = 117_{10}$$

Change to binary:

$$165_8 = (001\ \ 110\ \ 101)_2 = 1110101_2$$

In HL programing languages 165_8 is witten as “0o165”.

2.1.3 Hex (Hexadecimal) Numbers

Base $B = 16$
Digits: $b_i \in \{0, 1, \ldots, 7, 8, 9, A, B, C, D, E, F\}$

Example: number 117_{10}

$$75_{16} = 7\ 16 + 5 = 112 + 5 = 117_{10}$$

Change to binary number:

$$75_8 = (0111\ \ 0101)_2 = 1110101_2$$

Programing language: number 75_{16} is written as “0x75”.

Sometimes hex numbers are written as $75H = 75_{16}$.

2.1.4 Binary Coded Decimal Numbers (BCD)

4 binary digits for number range 0...9.

Example: number 117_{10} As BCD

$$117_{10} = (0001\ \ 0001\ \ 0111)_{BCD}$$

2.2 Conversion from Decimal to Binary Numbers

Subsequent Division by B results in digits b_i in the target system (as remainders)

If the division result is zero the algorithm has finished.

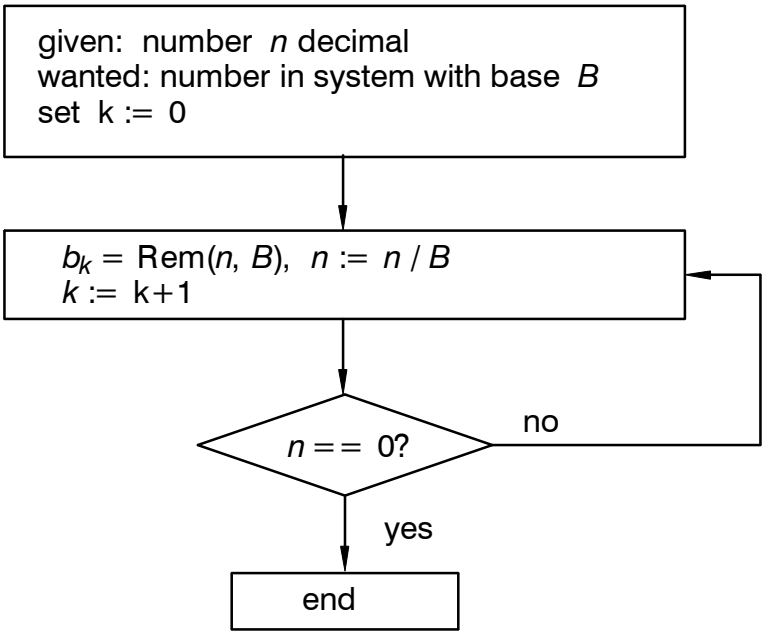


Figure 1.2: Number conversion (changing base B)

2.2.1 Example: Conversion of 102_{10} to binary [B = 2]

k	n	b_k (remainder)
0	102 / 2 = 51	0
1	51 / 2 = 25	1
2	25 / 2 = 12	1
3	12 / 2 = 6	0
4	6 / 2 = 3	0
5	3 / 2 = 1	1
6	1 / 2 = 0	1

[finished, $n = 0$.]

Result $b_k \ldots b_0$:

$n = 1100110_2$.

3 Arithmetic of Binary Numbers

Binary Addition (1 digit)

inputs			outputs	
$carry_{in}$	x_k	y_k	$x_k + y_k$	$carry_{out}$
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

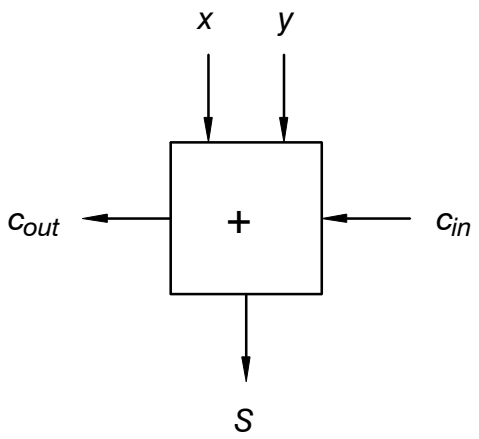


Figure 1.3: Symbol for Addition (one digit)

Example: Add of 110100111_2 and 101110_2

```
   1 1 0 1 0 0 1 1 1                  (423)10
+        1 0 1 1 1 0                   (46)10
--------------------
       1 0 1 1 1 0 0   (Carry Bits)
--------------------
   1 1 1 0 1 0 1 0 1                  (469)10
```

Subtract (one digit)

inputs			outputs	
$borrow_{in}$	x_k	y_k	$x_k - y_k$	$borrow_{out}$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

Example: Subtract 101110_2 from 110100111_2

```
   1 1 0 1 0 0 1 1 1                  (423)10
-        1 0 1 1 1 0                   (46)10
--------------------
-  0 1 1 1 1 0 0 0 0   (Borrow Bits)
--------------------
   1 0 1 1 1 1 0 0 1                  (377)10
```

last borrow "1" ==> $Y > X$ otherwise $Y \leq X$.

3.1 Negative Numbers and Complement

3.1.1 Negative Numbers

MSB = 0 → positive, MSB = 1 → negative

3.1.2 Complement of Numbers

Complement: negative number = C – positive number

2 complement types: one's complement two's complement.

one's complement: $C = 2^N - 1$, (general $C = B^N - 1$)

two's complement: $C = 2^N$, (general $C = B^N$)

3.1.3 Building Complements (8 Bit)

One's complement of 00110110

```
  0 1 1 1 1 1 1 1 1    (C = 2^8 - 1 = 255)
-   0 0 1 1 0 1 1 0    (54)
-------------------
    1 1 0 0 1 0 0 1
```

Two's complement of 00110110

```
  1 0 0 0 0 0 0 0 0    (C = 2^8 = 256)
-   0 0 1 1 0 1 1 0    (54)
-------------------
    1 1 0 0 1 0 1 0
```

Two's complement: invert all digits; then add 1.

Two's complement of 00110110

```
    0 0 1 1 0 1 1 0    (54)
-------------------
    1 1 0 0 1 0 0 1    (inversion)
+   0 0 0 0 0 0 0 1    (add of 1)
-------------------
    1 1 0 0 1 0 1 0    (two´s complement)
```

Signed numbers with N = 3 Bits

One's complement

```
decimal      -3   -2   -1   -0    0    1    2    3
binary      100  101  110  111  000  001  010  011
```

Two's complement

```
decima  l    -4   -3   -2   -1    0    1    2    3
binary      100  101  110  111  000  001  010  011
```

One's complement has two zeros (+0 and −0).

Ranges

One's complement: $n \in \{-2^{N-1}+1, ..., 2^{N-1}-1\}$

Two's complement: $n \in \{-2^{N-1}, ..., 2^{N-1}-1\}$, range not symmetric

3.1.4 Weights of Bits (Unsigned)

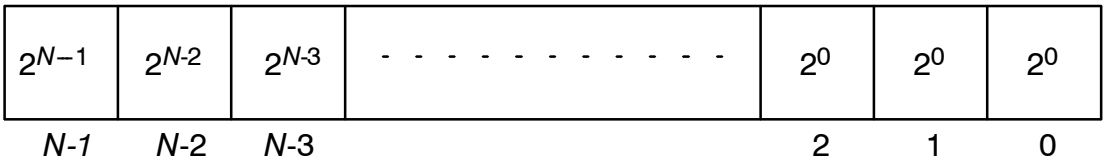


Figure 1.4: Unsigned (positive) numbers

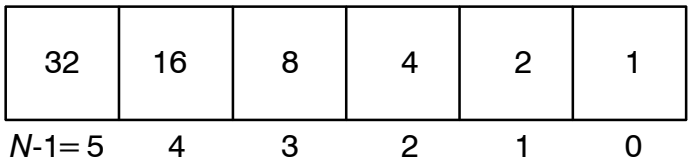


Figure 1.5: Unsigned number ($N = 6$, 6 Bits)

3.1.5 Weights of Bits (Signed)

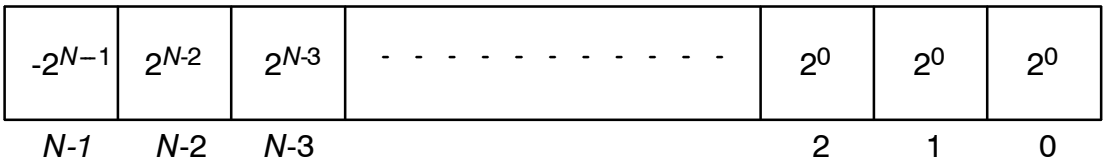


Figure 1.6: Signed numbers

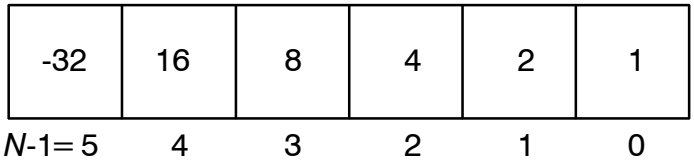


Figure 1.7: Signed number ($N = 6$, 6 Bits)

Range is

$$-2^{N-1} \le x \le 2^{N-1} - 1 \tag{1.2}$$

i.e. -32 ≤ x ≤ 31 for 6 Bit numbers.

3.1.6 Add and Subtract of Complement Numbers

Subtract is adding the complement

$$x - y = x - y + C - C = x + (C - y) - C = x + \bar{y} - C . \tag{1.3}$$

The number $\bar{y}$ is the complement with C.

$$C = 2^N \quad \text{(for twos' s complement)} \tag{1.4}$$

Examples (N = 4 bits)

1. $2 + 3 = 5$

```
  0 0 1 0   (2)
+ 0 0 1 1   (3)
-----------
  0 1 0 1   (5)
```

2. $2 - 3 = 2 + (-3) = -1$

```
  0 0 1 1     (3)
---------
  1 1 0 0     (inversion)
+ 0 0 0 1     (add of 1)
---------
  1 1 0 1     (-3)
```

add of $2 + (-3)$

```
  0 0 1 0     (2)
+ 1 1 0 1     (-3)
---------
  1 1 1 1     (-1)
```

3. $-2 -3 = (-2) + (-3) = -5$

```
    1 1 1 0   (-2)
+   1 1 0 1   (-3)
  ---------
(1) 1 0 1 1   (-5),   Carry Bit  ignored
```

4. $4 - 7 = 4 + (-7) = -3$

```
    0 1 0 0   (4)
+   1 0 0 1   (-7)
  ---------
    1 1 0 1   (-3)
```

5. $-3 - 6 = (-3) + (-6) = -9$?

```
    1 1 0 1    (-3)
+   1 0 1 0    (-6)
  ---------
  1 0 1 1 1    (+7)    (Overflow)
```

3.1.7 Number with N = 4 Bits

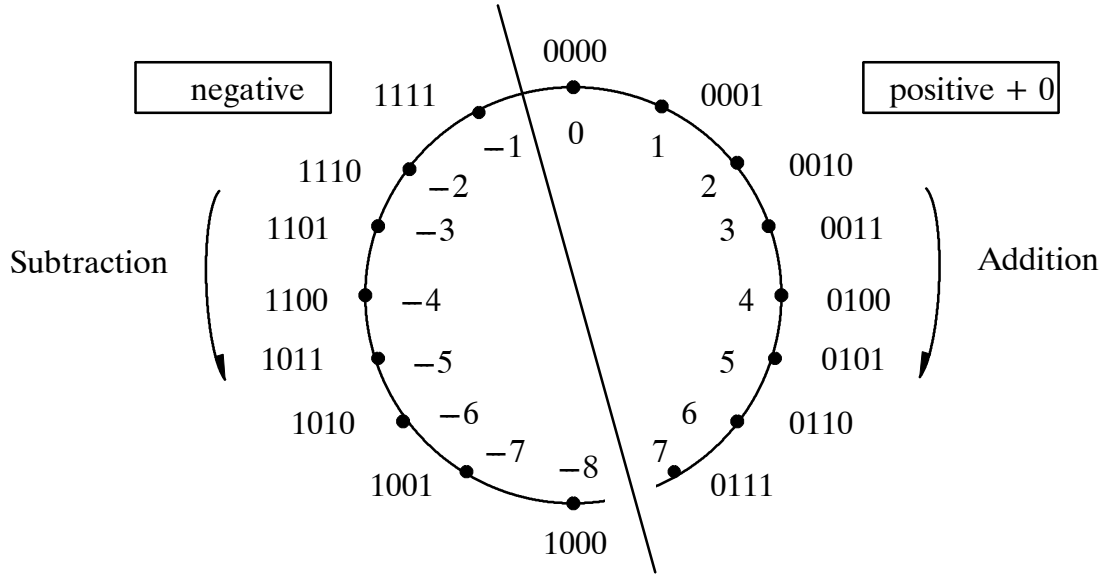


Figure 1.8: Number ring for two's complement

3.2 Binary Multiplication

Multiplication (1 bit)

inputs		outputs
x_k	y_k	$x_k \times y_k$
0	0	0
0	1	0
1	0	0
1	1	1

Multiplication 3 x 5

```
        0 0 1 1 x 0 1 0 1     3 x 5
-----------------------
        0 0 1 1
      0 0 0 0
    0 0 1 1
-----------------------
    0 0 1 1 1 1               15
```

```
0 0 1 1 x 0 1 0 1      3 x 5
-----------------------
          0 0 0 0
+         0 0 1 1
-----------------
          0 0 1 1
+       0 0 0 0         left shift
-----------------
        0 0 0 1 1
+     0 0 1 1           left shift
-----------------
      0 0 1 1 1 1
+   0 0 0 0             left shift
-----------------------
    0 0 0 1 1 1 1       result = 15
```

Word sizer of result is the sum of the word sizes of the inputs

3.2.1 Multiplication Two's Complement Numbers

Multiplication von (–3) x (–5)

```
1 1 0 1 x 1 0 1 1      (-3) x (-5)
-----------------------
          0 0 0 0
+         1 1 0 1
-----------------
        1 1 1 0 1
+       1 1 0 1         left shift
-----------------
      1 1 0 1 1 1       Zwischenergebnis = -9
+     0 0 0 0           left shift
-----------------
    1 1 1 0 1 1 1
+   0 0 1 1             left shift + negate!
-----------------------
(1) 0 0 0 1 1 1 1       result = 15,  Carry  ignored
```

***

Section 2

3.3 Division of Binary Numbers

A division can be reduced to s sequence of conditional subtract operations

Binary division of one bit:

inputs		outputs
x_k	y_k	x_k / y_k
0	0	undefined
0	1	0
1	0	undefined
1	1	1

The division x / y results in the quotient Q and the remainder R.

$$x = Q \cdot y + R . \tag{1.5}$$

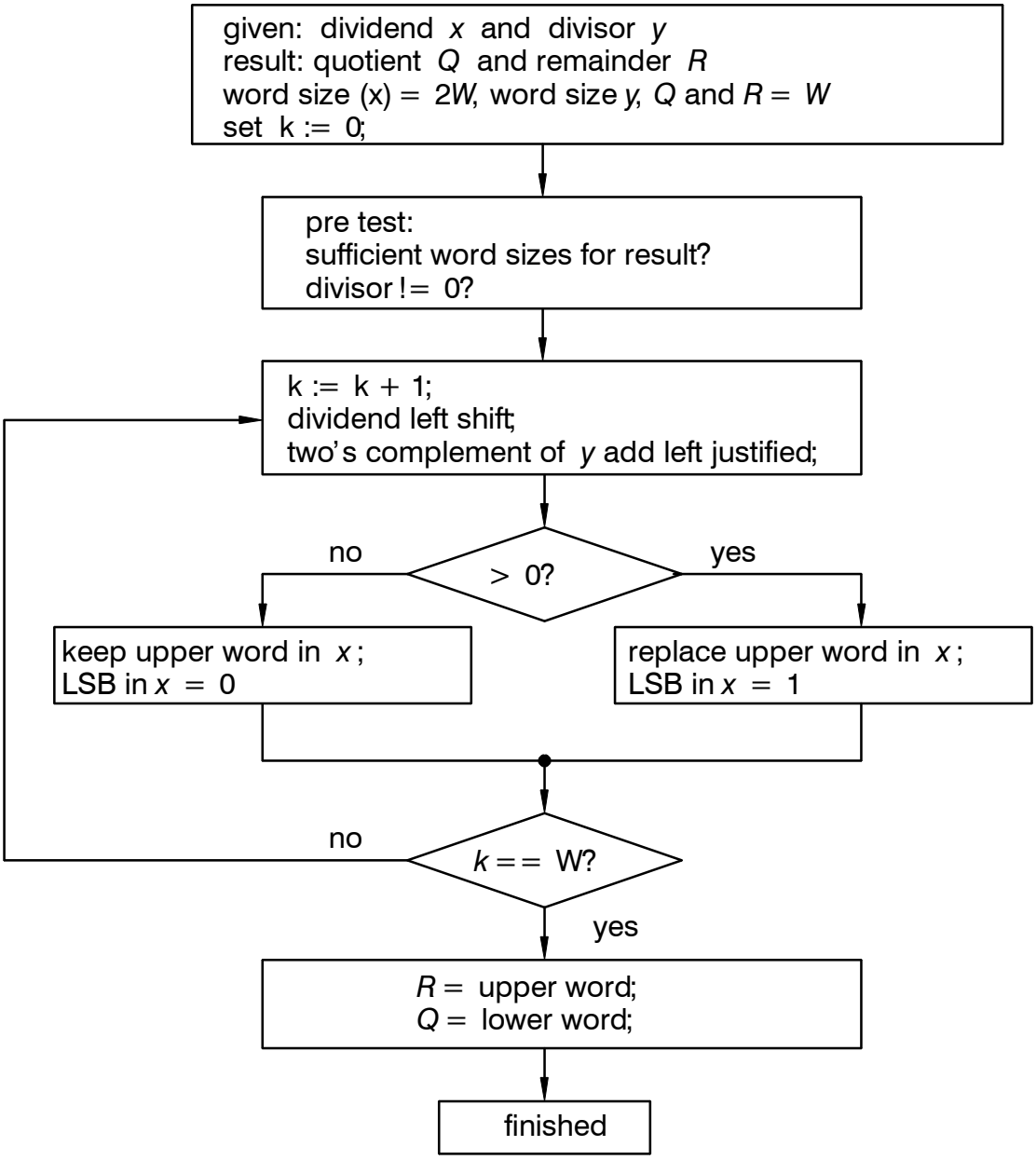


Figure 1.9: Division algorithm for unsigned numbers (shift-restore algorithm)

The following table shows a sample division (44 divided by 7)

```
     0 0 1 0  1 1 0 0    (44)
--------------------
     0 1 0 1  1 0 0 x    1. shift left of dividend
+    1 0 0 1             add of two´s complement of 7
 -------------------
     1 1 1 0             result negative => LSB = 0
--------------------
     0 1 0 1  1 0 0 0    => restore, LSB = 0
     1 0 1 1  0 0 0 x    2. shift left
+    1 0 0 1
--------------------
```

```
     0 1 0 0            result positive => LSB = 1
--------------------
     0 1 0 0  0 0 0 1   => replace, LSB = 1
     1 0 0 0  0 0 1 x   3. shift left
+    1 0 0 1
--------------------
     0 0 0 1            result positive  => LSB = 1
--------------------
     0 0 0 1  0 0 1 1   => replace, LSB = 1
     0 0 1 0  0 1 1 x   4. shift left
     1 0 0 1
--------------------
     1 0 1 1            result negative  => LSB = 0
--------------------
     0 0 1 0  0 1 1 0   final result => R = 2, Q = 6
====================
```

Microprocessors often support this algorithm by a special conditional subtract operation.

4 Floating Point Numbers – IEEE Single Precision Floating

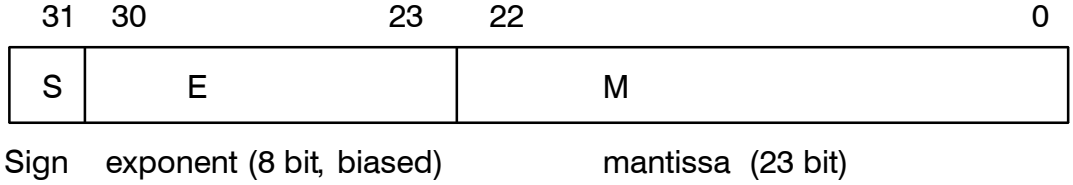


Figure 1.10: 32 Bit IEEE Floating Point Format (single precision)

The value can be computed as follows:

$$F = (1 - 2S) \cdot 2^{E-127} \cdot 1.\mathrm{M}\,. \qquad (1.6)$$

$E = 127$ means $2^0 = 1$. The mantissa has an implicit '1' before the dot (as long as $M \neq 0$).

Example: Floating point number $x = -6{,}5$

```
1  1 0 0 0 0 0 0 1  1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 =
0xC0D0000  (hexadecimal)
sign = 1  =>  negative number
exponent = 129  => 2^2
mantissa = 1.0 + 0.5 + 0.125 = 1.625
```

$$x = (-1) \cdot 2^2 \cdot 1.625 = -6.5 \tag{1.7}$$

Meaning of $E = 0$ und $E = 255$:

$E = 255$ and $F = 0$ and $S = 1 \rightarrow x = -\infty$
$E = 255$ and $F = 0$ and $S = 0 \rightarrow x = +\infty$
$E = 0$ and $F = 0$ and $S = 1 \rightarrow x = -0$
$E = 0$ and $F = 0$ and $S = 0 \rightarrow x = +0$

Single precision corresponds to approximately 7 decimal digits.

5 More Coding of Numbers and Characters

5.1 Gray-Code

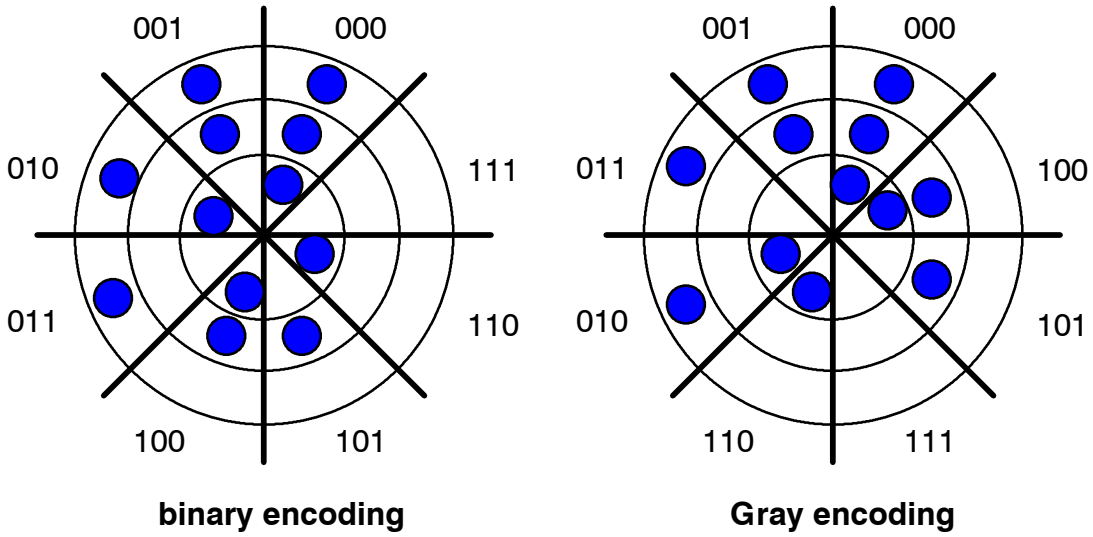


Figure 1.11: Binary and Gray encoded incremental sensors

Only one bit changes from one sector to an adjacent one in Gray code.

No erroneous readings with Gray code.

5.2 Character Encoding

Widely and simple encoding is ASCII (7 bit code, *American Standard Code for Information Interchange*). For more complex systems (Chinese characters) unicode is used. ASCII is unicode compatible.

Table of ASCII encoding

lower hex value \ upper hex value	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	´	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	–	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Example: character “m”

“m” => $6D_{16} = 0110\ 1101_2$.

The values 128 ... 255 can be used for internationalization (better: unicode)

Unicode

The international standard for any character in any language is Unicode. Unicode is organized in 16 planes; each plane owns $2^{16} = 65.536$ characters.

UTF

UTF is the coding standard for Unicode. The most common format is UTF-8 (especially in web applications).

UTF-8 is ASCII compatible in that the first 128 characters ($0...7F_H$) are identical.

Other characters require more bytes (multi-byte encoding). In this case the first byte starts with as many ’1’s as bytes are required (up to 6) followed by a ’0’ and payload information. All subsequent bytes start with “10” followed by the payload information. The payload bits are denoted as ’x’.

1 Byte (ASCII compatible)

```
0xxxxxxx
```

2 Bytes

```
110xxxxx 10xxxxxx
```

3 Bytes

```
1110xxxx 10xxxxxx 10xxxxxx
```

4 Bytes

```
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

5 Bytes

```
111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

6 Bytes

```
1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

Unicode requires only a maximum of 4 bytes (5 bytes and 6 bytes are an exception). The payload is 21 bits. Only the shortest form of coding is allowed, i.e. character 'a' is an ASCII character and must be coded in one byte!

***

Section 3

5.3 Minimum Distance

Minimum distance is equivalent to the number of bits that need to be changed get from on valid word to another valid word. See figure for a 3 bit example.

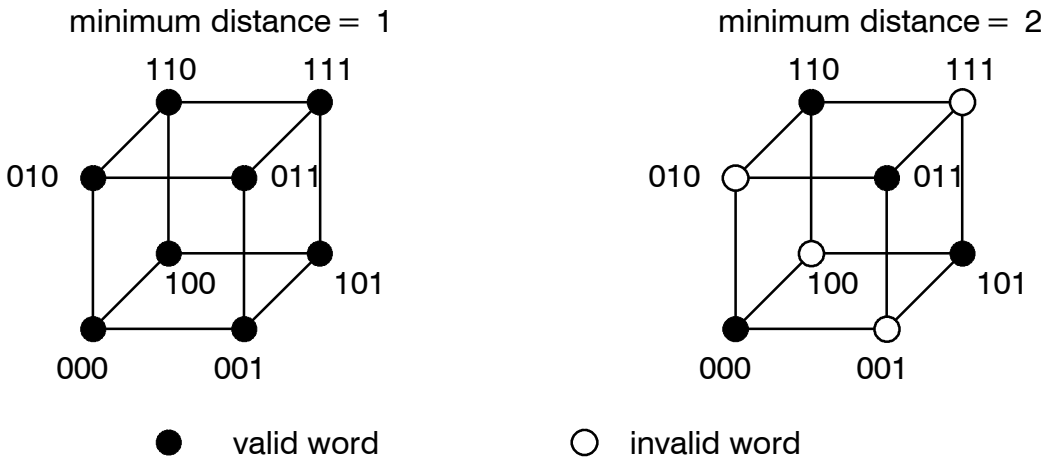


Figure 1.12: Minimum distance of 1 and 2 (hypercube)

To detect an error, a minimum distance of at least 2 is required.

5.3.1 Parity Bit

Minimum Distance = 2, even and odd parity for 3 information bits

information bits	even parity	odd parity
0 0 0	0 0 0 0	0 0 0 1
0 0 1	0 0 1 1	0 0 1 0
0 1 0	0 1 0 1	0 1 0 0
0 1 1	0 1 1 0	0 1 1 1
1 0 0	1 0 0 1	1 0 0 0
1 0 1	1 0 1 0	1 0 1 1
1 1 0	1 1 0 0	1 1 0 1
1 1 1	1 1 1 1	1 1 1 0

With the help of one parity bit *single bit errors* can be detected.

5.3.2 Error Correction (CRC)

Error correction requires several extra bits. The number of error correction bits depends on the length of the data bits which should be transmitted. If n_d is the number of data bits and n_c is the number of correction bits the following condition must be satisfied:

$$\mathrm{ld}\left(n_d + n_c\right) \leq \quad n_c \,. \tag{1.8}$$

Thus, no simple formula exist to determine the required number of correction bits.

An example with 8 data bits shall demonstrate the error correction process. Bit numbering starts with 1 instead of 0 as this is required for locating bit errors.

bit position	8	7	6	5	4	3	2	1
values	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**

The correction bits c_1 ... c_4 are inserted at positions which are powers of 2.

12	11	10	9	8	7	6	5	4	3	2	1
1	**1**	**0**	**0**	**c_4**	**1**	**0**	**0**	**c_3**	**1**	**c_2**	**c_1**

The correction bits are the logical XOR of all bit positions holding a '1':

```
          0011      ( 3)
xor       0111      ( 7)
xor       1011      (11)
xor       1100      (12)
--------------------
          0011      = c4 c3 c2 c1
```

Inserting these values into the transmission word gives:

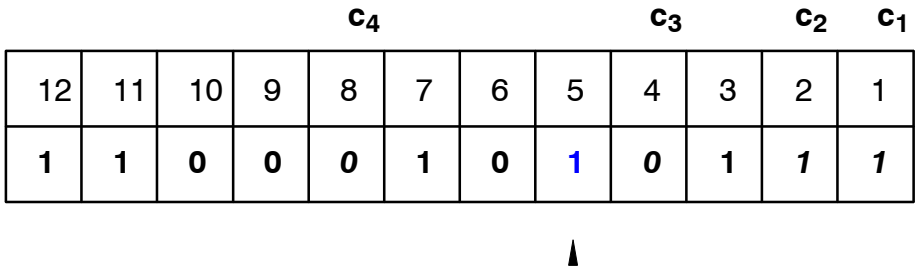
				c_4				**c_3**		**c_2**	**c_1**
12	11	10	9	8	7	6	5	4	3	2	1
1	**1**	**0**	**0**	***0***	**1**	**0**	**0**	***0***	**1**	***1***	***1***

If nor error occurs the logical sum (XOR) of all bit positions holding a '1' *and* the correction bits should be 0:

```
          0011      ( 3)
xor       0111      ( 7)
```

```
xor       1011      (11)
xor       1100      (12)
xor       0011      (corection bits)
--------------------
          0000      = result 0 (no error)
```

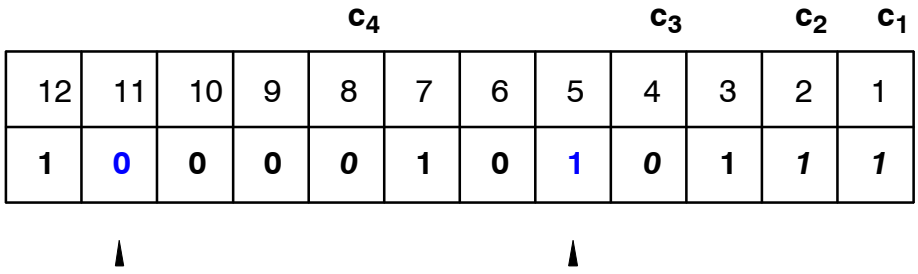
Now we assume the bit 5 was received as '1' (which is an error).



The same calculation results in the result 5. This means that bit 5 was received wrong and can be corrected.

```
          0011      ( 3)
xor       0101      ( 5)
xor       0111      ( 7)
xor       1011      (11)
xor       1100      (12)
xor       0011      (correction bits)
--------------------
          0101      = result 5 (bit 5 is wrong)
```

Multi bit errors (here bit 5 and bit 11) can also be detected. Correction, however, is with 4 correction bits not possible.



Checking the received word gives:

```
          0011      ( 3)
xor       0101      ( 5)
xor       0111      ( 7)
xor       1100      (12)
xor       0011      (correction bits)
--------------------
          1110      = result 14 (wrong,
                      no correction possible)
```

Section 4

5.4 Combinational Logic

In digital systems physical states are reduced to several discrete values (in VHDL `std_logic` type we have 9 different values). The boolean algebra knows only two values: '0' and '1'.

Boolean algebra, thus, can be used to describe the functional operation of a system.

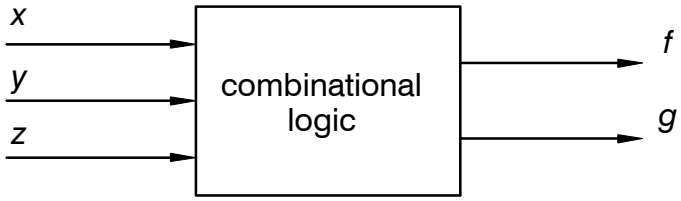
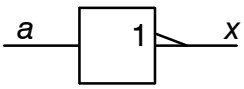


Figure 1.13: Combinational logic with 3 inputs and 2 outputs

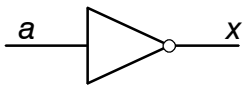
5.5 Logic Gates

5.5.1 Inverter

IEC symbol



int. symbol



truth table

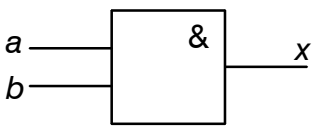
a	x
0	1
1	0

boolean equation

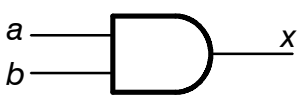
$x = a'$ ($x =$ not a)

5.5.2 AND

IEC symbol



int. symbol



truth table

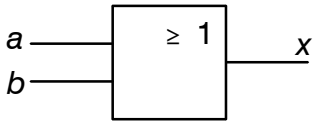
a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

boolean equation

$x = a \cdot b$ ($x =$ a and b)

5.5.3 OR

IEC symbol



int. symbol

truth table

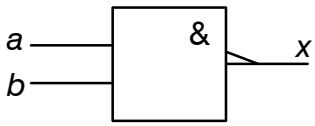
a	*b*	*x*
0	0	0
0	1	1
1	0	1
1	1	1

boolean equation

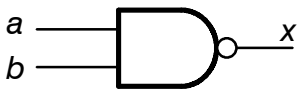
$x = a + b$ ($x =$ *a* or *b*)

5.5.4 NAND

IEC symbol



int. symbol



truth table

a	*b*	*x*
0	0	1
0	1	1
1	0	1
1	1	0

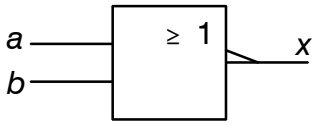
boolean equation

$x = (a \cdot b)'$ ($x =$ *a* nand *b*)

With NAND gates only any boolean logic becomes possible.

5.5.5 NOR

IEC symbol



int. symbol

truth table

a	*b*	*x*
0	0	1
0	1	0
1	0	0
1	1	0

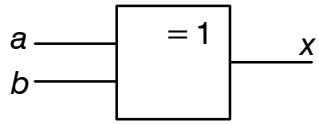
boolean equation

$x = (a + b)'$ ($x =$ *a* nor *b*)

With NOR gates only any boolean logic becomes possible.

5.5.6 XOR

IEC symbol



int. symbol

truth table

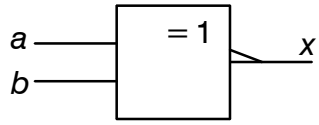
a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

boolean equation

$x = (a \oplus b)$ ($x = a$ xor b)

5.5.7 XNOR

IEC symbol



int. symbol

truth table

a	b	x
0	0	1
0	1	0
1	0	0
1	1	1

boolean logic

$x = (a \oplus b)'$ ($x = a$ xnor b)

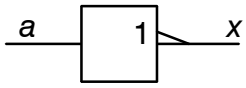
5.6 Priority Rules

1.	NOT	$\overline{a}$
2.	AND	$a \cdot b$
3.	OR	$a + b$
4.	NAND	$(a \cdot b)'$
5.	NOR	$(a + b)'$
6.	XNOR	$a \equiv b$
7.	XOR	$a \oplus b$

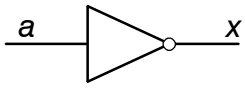
Brackets can be used to change the default priority rules. Often only priority rules 1.–3. are used; in all other cases brackets are recommended.

5.6.1 Unary Operators

symbol



intern. symbol



truth table

a	x
0	1
1	0

boolean function

german (DIN):	$x = \bar{y}$	($x =$ nicht a)
intern. form:	$x = a'$	($x =$ not a)

5.6.2 Binary Operators

inputs	
$a =$	1 0 1 0
$b =$	1 1 0 0

outputs x	boolean symbol	notation	elementary logic
0 0 0 0	$x = 0$	constant 0	
0 0 0 1	$x = (a + b)'$	NOR	•
0 0 1 0	$x = a \cdot b'$	inhibit	
0 0 1 1	$x = b'$	negate (b)	•
0 1 0 0	$x = a' \cdot b$	Inhibit	
0 1 0 1	$x = a'$	negate (a)	•
0 1 1 0	$x = a \oplus b$	XOR	•
0 1 1 1	$x = (a \cdot b)'$	NAND	•
1 0 0 0	$x = a \cdot b$	AND	•
1 0 0 1	$x = a \equiv b$	equivalence	•
1 0 1 0	$x = a$	identity (a)	
1 0 1 1	$x = a + b'$	implication	
1 1 0 0	$x = b$	identity (b)	
1 1 0 1	$x = a' + b$	implication	
1 1 1 0	$x = a + b$	OR	•
1 1 1 1	$x = 1$	constant 1	

5.6.3 Boolean Theorems for One Variable and Constants

The proof is straight forward by using the truth table.

$$x + 0 = x \tag{2.1}$$

$$x + 1 = 1 \tag{2.2}$$

$$x \cdot 0 = 0 \tag{2.3}$$

$$x \cdot 1 = x \tag{2.4}$$

$$x + x' = 1 \tag{2.5}$$

$$x \cdot x = x \tag{2.6}$$

$$x + x = x \tag{2.7}$$

$$x \cdot x = x \tag{2.8}$$

$$x'' = x \tag{2.9}$$

5.6.4 Boolean Theorems for Several Variables

The proof is also possible by using the truth table since the number of rows to check is limited. Also 5.6.3 can proof some results.

Commutativity

$$x + y = y + x, \quad x \cdot y = y \cdot x. \tag{2.10}$$

Associativity

$$(x + y) + z = x + (y + z), \quad (x \cdot y) \cdot z = x \cdot (y \cdot z). \tag{2.11}$$

Distributivity

$$x \cdot y + x \cdot z = x \cdot (y + z), \quad (x + y) \cdot (x + z) = x + y \cdot z. \tag{2.12}$$

Covering

$$x + x \cdot y = x, \quad x \cdot (x + y) = x. \tag{2.13}$$

Combining

$$x \cdot y + x \cdot y = x, \quad (x + y) \cdot (x + y) = x. \tag{2.14}$$

DeMorgan 1 (inverted AND)

$$(x_1 \cdot x_2 \cdot x_3 \cdot \cdots)' = x_1' + x_2' + x_3' + \cdots. \tag{2.15}$$

DeMorgan 2 (inverted OR)

$$(x_1 + x_2 + x_3 + \cdots)' = x_1' \cdot x_2' \cdot x_3' \cdot \cdots. \tag{2.16}$$

Duality (Metatheorem) non-trivial and important

Any theorem or identity remains valid if '0' and '1' are swapped and '+' and '' are swapped also. The '+' and '$\cdot$' operators exchange precedence after applying duality theorem as well. Example:

$$x + x \cdot y \cdot 1 = x \cdot (x + y + 0) = x. \tag{2.17}$$

5.6.5 Boolean Expression Types

The following types are just definitions. No boolean mathematics is involved with it.

Literal

Variable or complement of a variable: x, x'.

Product term

Logical product of one or more literals: x · y · z'.

Sum term

Logical sum of one or more literals: x + y' + z.

Sum of products

Logical sum product terms: x · y' · z' + x' · y · z + x' · y' · z.

Product of sums

Logical product of sum terms: (x + y' + z') · (x' + y + z) · (x' + y' + z).

Normal term

Logical term where no variable occurs more than once: x + y' · z'.
Any non-normal term can always converted to a normal term applying the previous theorems.

Minterm

Logical product as a normal term: x · y' · z'.

Maxterm

Logical sum as a normal term: x + y' + z'.

Canonical sum

Logical sum of minterms (corresponding to truth table rows which produce a '1').
This is denoted as $\Sigma_{x,y,z}$ for a three variable truth table.

Canonical product

Logical product of maxterms (corresponding to truth table rows which produce a '0').
This is denoted as $\Pi_{x,y,z}$ for a three variable truth table.

5.6.6 Circuit Minimization

Circuits based on canonical sums or canonical products are often expensive because they contain unnecessary terms. Circuit minimization is based on the combining theorem. If the

number of variables is not greater than 4 the minimization can be carried out graphically by **Karnaugh maps**.

x2	x1	y
0	0	0
0	1	1
1	0	0
1	1	1

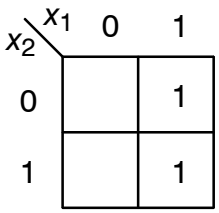


Figure 1.14: 2-variable truth table and Karnaugh map

x3	x2	x1	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

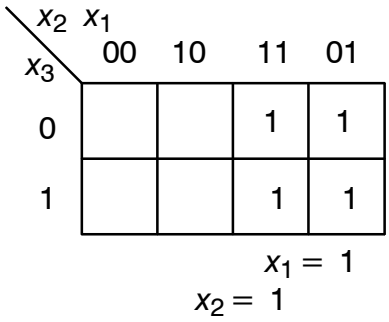


Figure 1.15: 3-variable truth table and Karnaugh map

For any adjacent cell only one input variable changes it's value. This is the reason for the ordering of rows and columns in the Karnaugh map.

Karnaugh maps have no start and end column or row. It can be any column or row.

x4	x3	x2	x1	y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

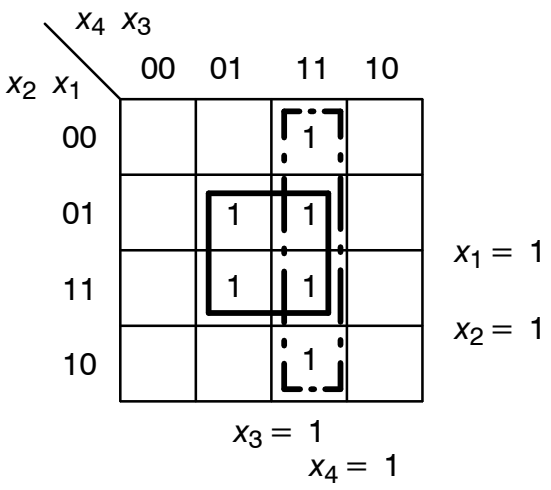


Figure 1.16: 4-variable truth table and Karnaugh map

Any set of adjacent ’0’ belongs to a minimal maxterm; a set of adjacent ’1’ form a minimal minterm. A large “area” corresponds to a simple term and vice versa. Thus, it is desirable to find the largest possible areas.

We will focus the theory to ’1’. The same theory applies to ’0’ by the duality theorem.

Prime Implicant

A prime implicant a maximum area of ’1’ (see areas in fig. 1.16). Of course smaller areas exist but they are no prime implicants.

A minimal sum is a sum of prime implicants.

The sum of all prime implicants is the *complete sum*. This is not necessarily minimal.

Distinguished 1-cells

A cell which is covered by only *one* prime implicant.

Essential prime implicant

A prime implicant that covers at least *one* distinguished 1-cell.

Every essential prime implicant must be included in a minimal logic function.

After removing all ’1’ from the essential prime implicants the remaining ’1’ need to be covered by additional prime implicants. If there is no distinguished 1-cell, any single cell can be treated as a distinguished 1-cell. Finding the truly minimized logical function is sometimes a non-trivial task.

***

Lab #01

6 Lab #01: Bargraph

A bargraph display is widely used to visualize a measurement value. Amplitude and increasing or decreasing values are easier to read than on a pure numeric display.

Such a display should be created for a three bit unsigned integer value. The limited number of available gates require optimal solutions as minimized sum of products or minimized product of sums.

The inputs (3 bit value) is taken from switches, the outputs are LEDs. Fig. shown the input-output map of the bargraph display. The red circles can be regarded as ’1’ and the white circles are ’0’s.

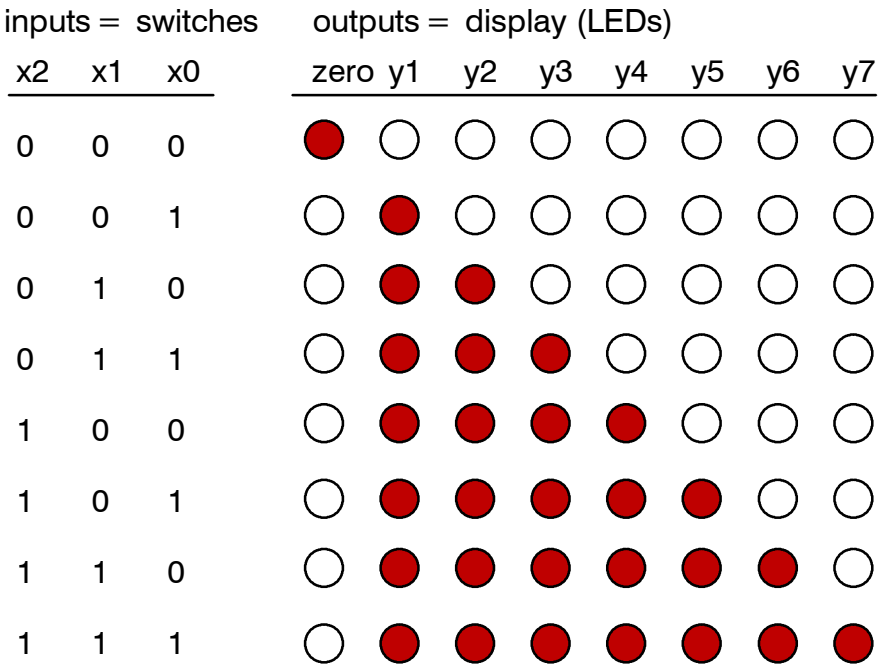
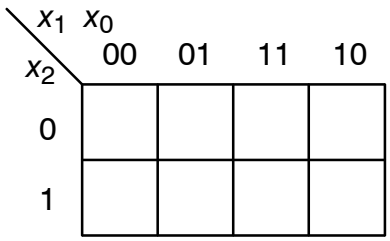


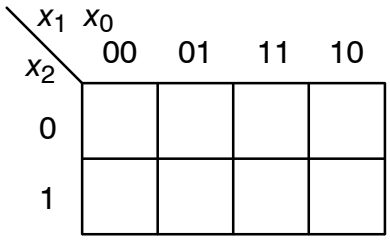
Figure 1.17: Bargraph for 3 bit

- Fill in '0's and '1's in the Karnaugh maps for every output. You need one Karnaugh diagram for every output.
- Choose between product of sums and sum of products, whatever is easier.
- Draw *all required prime implicants* for a solution. Write down the corresponding minterms or maxterms.
- Write down the boolean expression for every output.
- You might take advantage of the DeMorgan theorem to obtain a simpler solution.



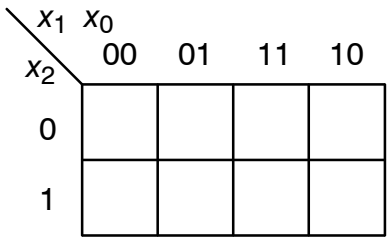
zero =

Figure 1.18: Karnaugh map for zero



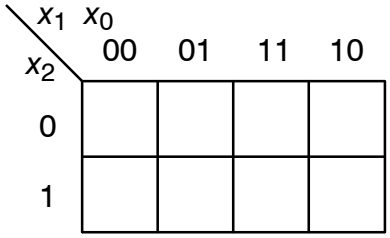
y_1 =

Figure 1.19: Karnaugh map for y_1



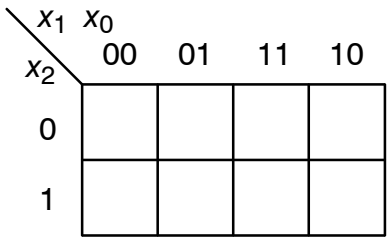
$y_2 =$

Figure 1.20: Karnaugh map for y_2



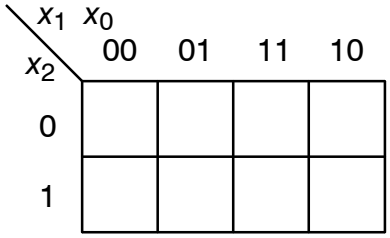
$y_3 =$

Figure 1.21: Karnaugh map for y_3



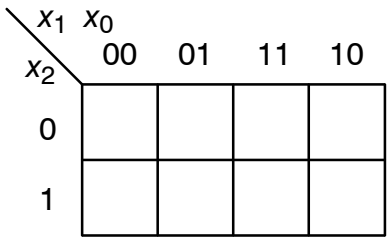
$y_4 =$

Figure 1.22: Karnaugh map for y_4



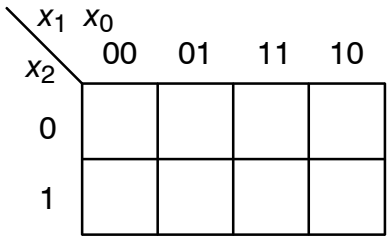
$y_5 =$

Figure 1.23: Karnaugh map for y_5



$y_6 =$

Figure 1.24: Karnaugh map for y_6



$y_7 =$

Figure 1.25: Karnaugh map for y_7

- Build the complete circuits in gate logic and verify proper operation. Change the switches to represent numbers from 0...7.

***

Section 5

7 Sequential Logic

7.1 Storage Element

The storage element in fig. 1.26 keeps its information for an infinite time (or until power is cycled down). This is the basic structure for static memory.

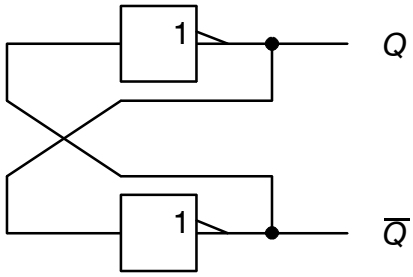


Figure 1.26: Memory (register) with combinational components

On power-on the circuit will store Q='0' and $\overline{Q}$='1' or Q='1' and $\overline{Q}$='0'. This is called *meta stable* circuit.

7.1.1 RS-Latch

> Latch = asnychonous storage element, can be cleared (reset) or set.
> It does not depend on a clock signal.

In contrast to a latch a Flip-Flop store the information with a clock signal (see section 7.2).

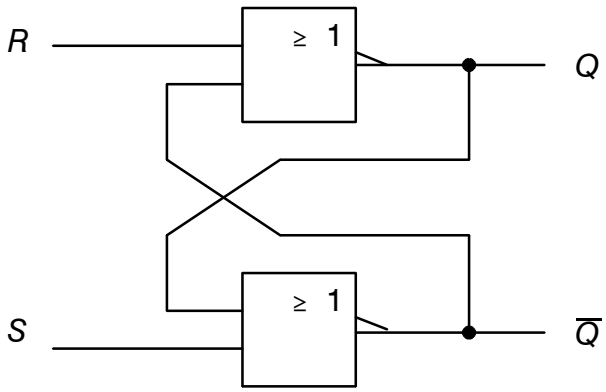


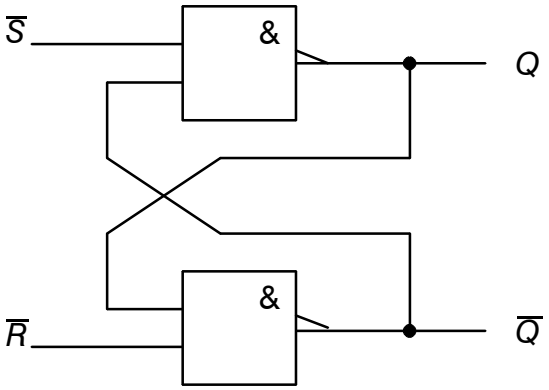
Figure 1.27: RS-Latch with NOR gates

The function can be expressed by the following table. Note that this is a not a truth table as it is used for combinational logic.

S	R	Q	$\overline{Q}$	
0	0	store		
0	1	0	1	(clear or reset)
1	0	1	0	(set)
1	1	0	0	(meta stable)

In practical application the last row (meta stable) should be avoided.

RS-Latches can be created with NOR or NAND gates.



$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$	
0	0	1	1	(meta stable)
0	1	1	0	(set)
1	0	0	1	(reset)
1	1	store		

Figure 1.28: RS-Latch with NAND gates (sometime denoted as $\overline{RS}$-Latch)

$S = 1,\ R = 1$ is used to store data.

Latches have their own symbols.

Figure 1.29: RS-Latch and $\overline{RS}$-Latch

7.1.2 RS-Latch with Enable

An extension to the RS-Latch is the enable input. The solution with only NAND gates is shown in fig. 1.30.

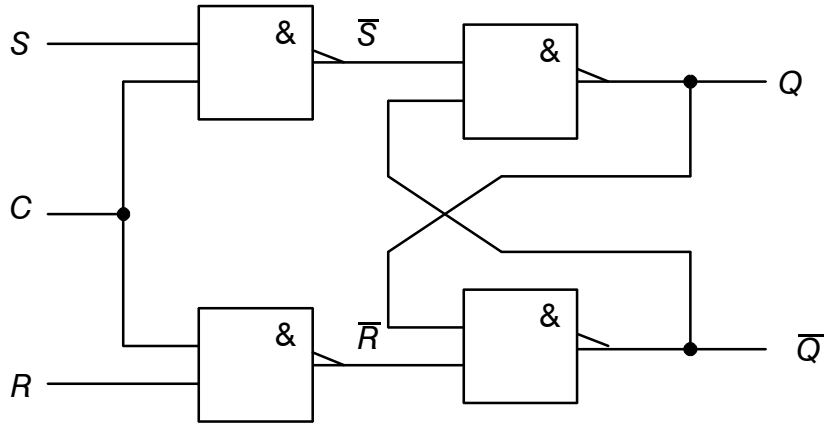


Figure 1.30: RS-Latch with enable input (C = Enable)

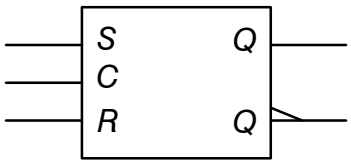


Figure 1.31: Symbol for an RS-Latch with enable

S	R	C	Q	$\overline{Q}$	
0	0	1	store		
0	1	1	0	1	(reset)
1	0	1	1	0	(set)
1	1	1	1	1	(meta stable)
x	x	0	store		

7.1.3 D-Latch

The D-Latch allows to store data on its D input. It avoids the meta stable state.

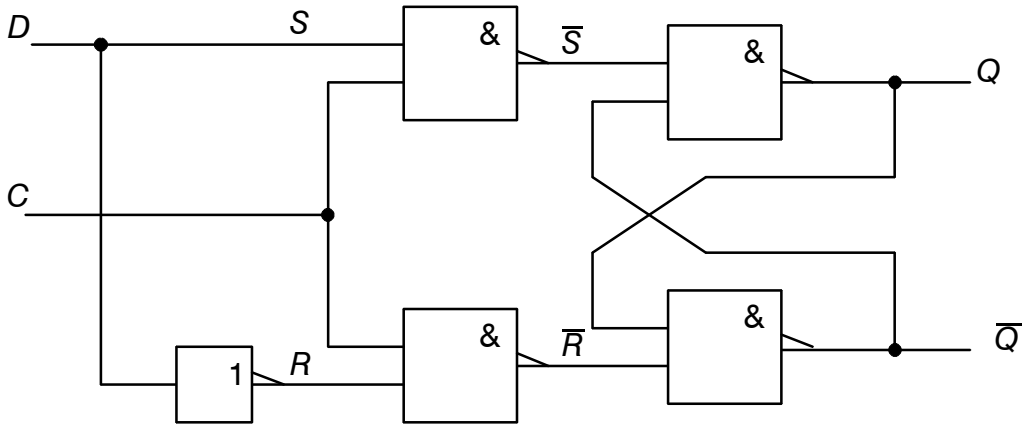


Figure 1.32: D-Latch with enable (C = Enable)

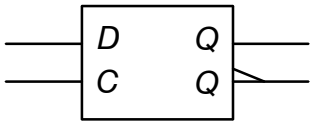


Figure 1.33: Symbol for D-Latch

D	C	Q	$\overline{Q}$	
0	1	0	1	(reset)
1	1	1	0	(set)
x	0	store		

7.2 Flip-Flop

Data is store with the positive or negative edge of a "clock" signal.

Flip-Flop = *edge triggered* Latch

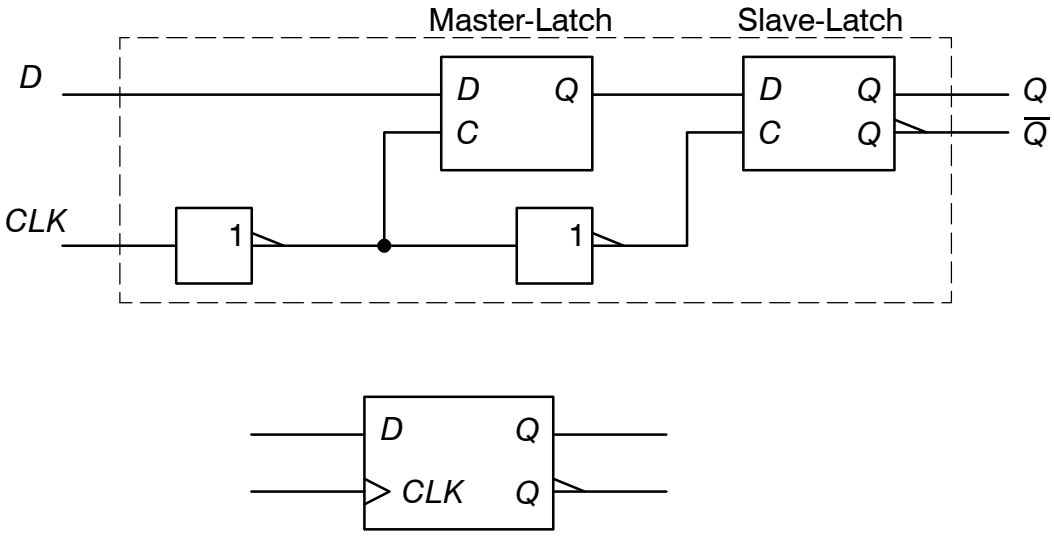


Figure 1.34: Flip-Flop made from two two D-Latches (triggered on positive edge3) and Symbol

The functional table contains the new "edge" symbol.

D	C	Q	$\overline{Q}$	
0	↓	0	1	(clear)
1	↓	1	0	(set)
x	0	store		
x	1	store		

If $CLK = 0$ the master Latch stores D input (slave is still disabled). The positive edge of CLK "closes" the master latch and enables the slave latch. This is the point where data is stored and becomes visible on the output Q.

If the clock signal becomes '0' again the slave latch will keep it's value (the output does not change).

Negative edge triggered Flip-Flops us an inverted *CLK s*ignal.

Some D-Flip-Flops have *asynchronous* set and reset inputs denoted as *PRESET* and *CLEAR*.

PRESET = asynchronous set of Flip-Flop.

CLEAR = asynchronous clear of Flip-Flop.

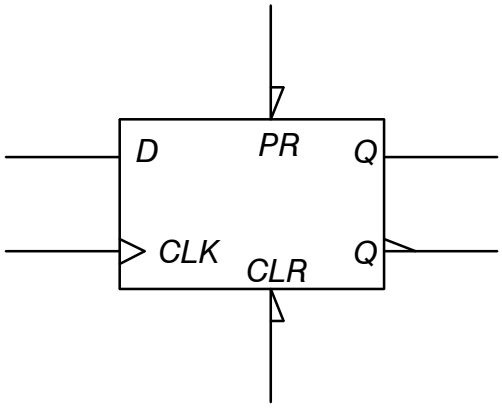


Figure 1.35: D-Flip-Flop with *PRESET* and *CLEAR*

All other Flip-Flop types can be derived easily from D-Flip-Flops.

7.3 State-Machines

State machines (FSM = finite state machine) generate output signals from inputs, internal states and a clock signal. This is denoted as sequential logic.

Complex digital systems like microprocessors or control units are alway state machines.

The basic structures are *Moore* or *Mealy* machines.

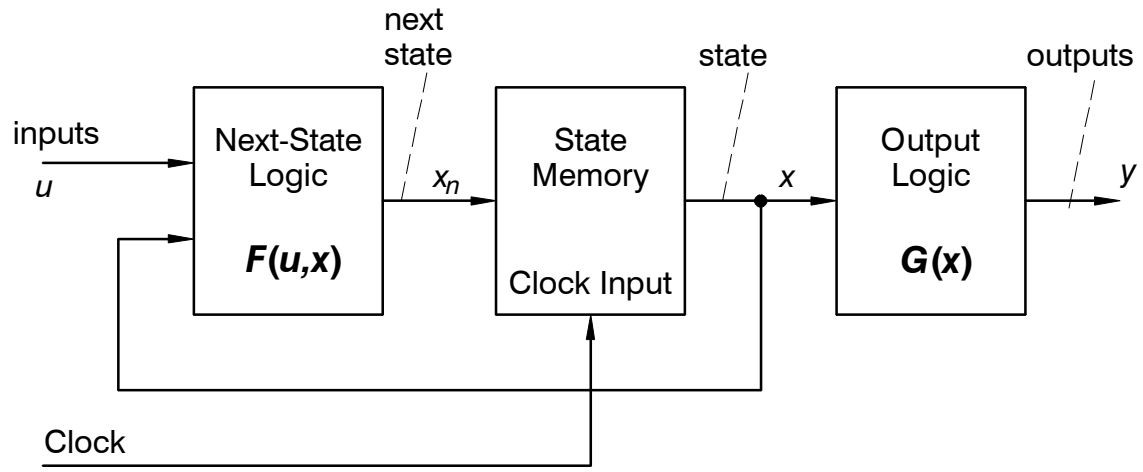


Figure 1.36: Moore machine

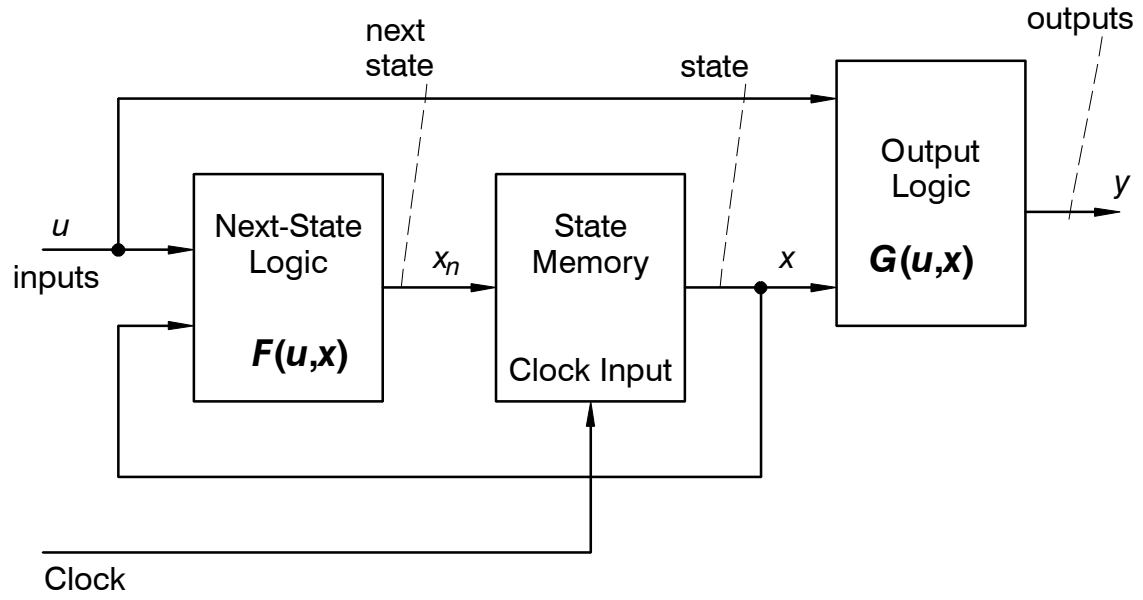


Figure 1.37: Mealy machine

The logical functions

$$F(u,x)\,, \quad G(x) \tag{8.1}$$

of the Moore machine or

$$F(u,x)\,, \quad G(u,x) \tag{8.2}$$

of the Mealy machine are combinational logic functions.

7.4 Example: Traffic Light Controller

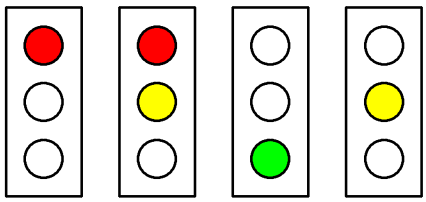


Figure 1.38: Minimum set of states for a traffic light controller

INPUT *u*: Reset signal, sets traffic light to red".
(active low, i.e. '0' means "Reset").

STATES *x*: 4 states / 2 bits if binary encoded. This means two Flip-Flops.

OUTPUTS *y*: red, yellow, green

Moore machine is sufficient.

7.4.1 Next-State Logic *F*(*u*, *x*)

If u = '0' the next state will always state 0. Otherwise the next-state logic generates the following state:

			next state	
u	x1	x0	x1n	x0n
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

***F*(*u*, *x*)**

The minimal solution (by the help of Karnaugh maps) is shown in fig. 1.39.

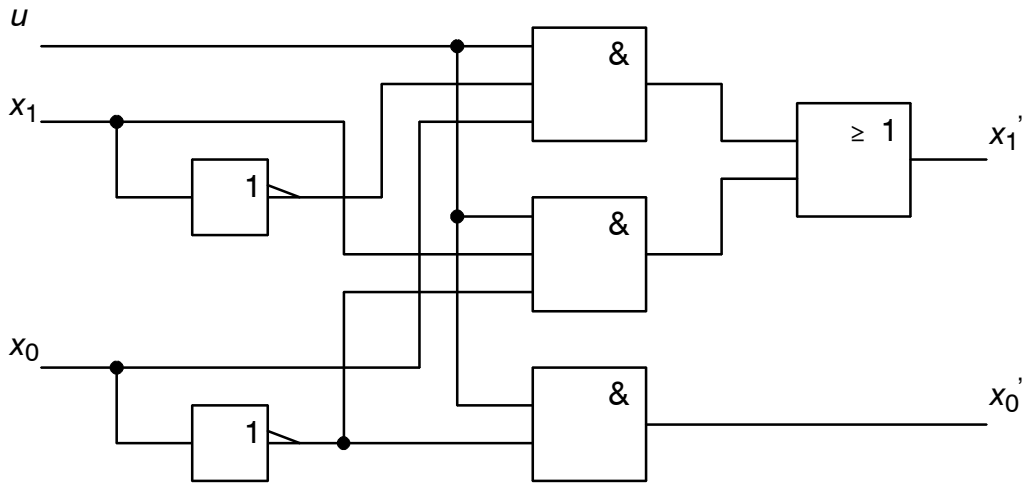


Figure 1.39: Minimal solution of $F(u, x)$

7.4.2 Output-Logic *G*(x) (for a Moore Machine)

The truth table shows the connections between states and outputs:

		lights		
x1	x0	red	yel	green
0	0	1	0	0
0	1	1	1	0
1	0	0	0	1
1	1	0	1	0

G(x)

The minimal solution is shown in fig. 1.40.

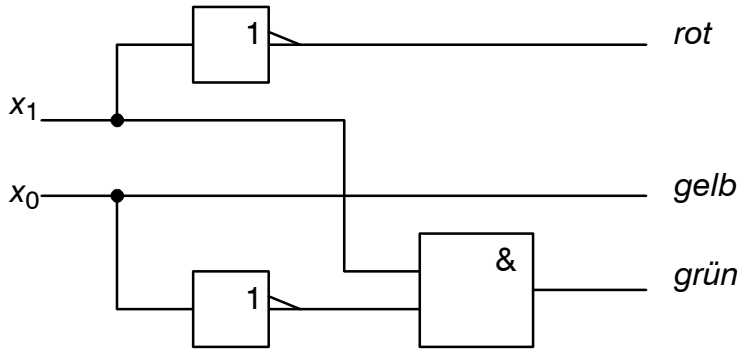


Figure 1.40: Minimal solution of $G(x)$

7.4.3 State-Memory

The state memory is the simplest component of a state machine. It consist aof the required number of Flip-Flops. In this case we have x_0 and x_1.

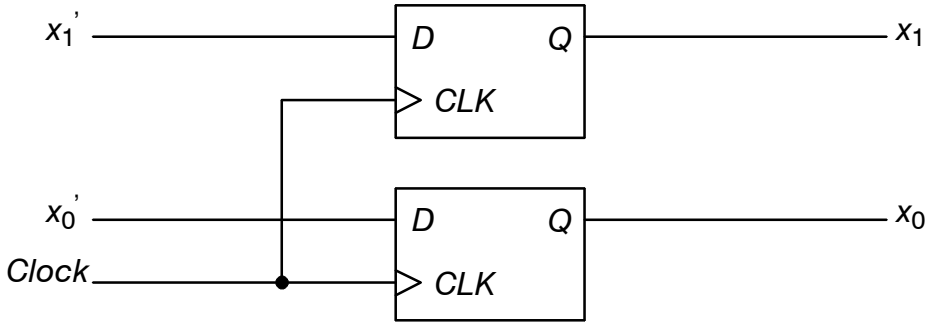


Figure 1.41: State memory

7.4.4 Complete Traffic Light Controller

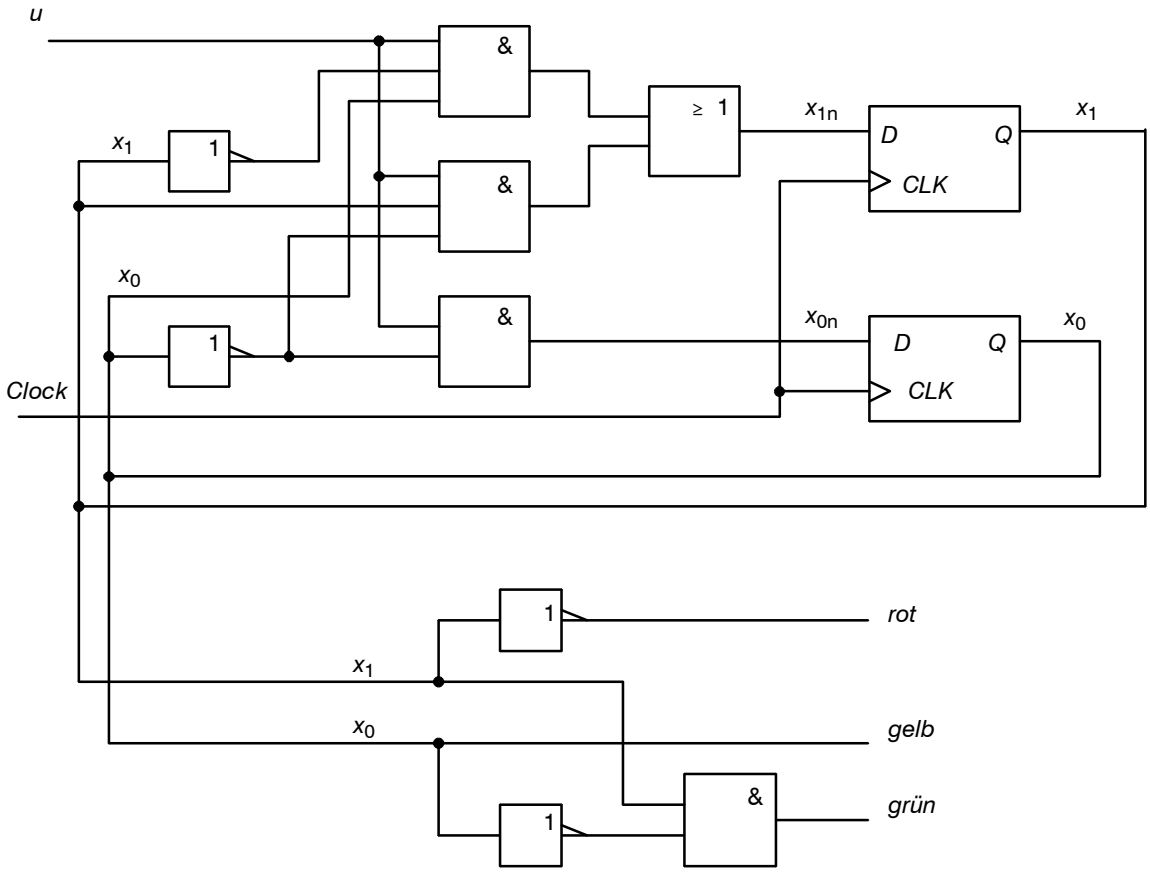


Figure 1.42: Traffic light controller circuit

7.5 State Diagram

Any state machine should be designed by a state diagram. This diagram is a precise description of a state machine.

state ⇔ circle
arrow ⇔ transition (with condition for “firing” the transition)

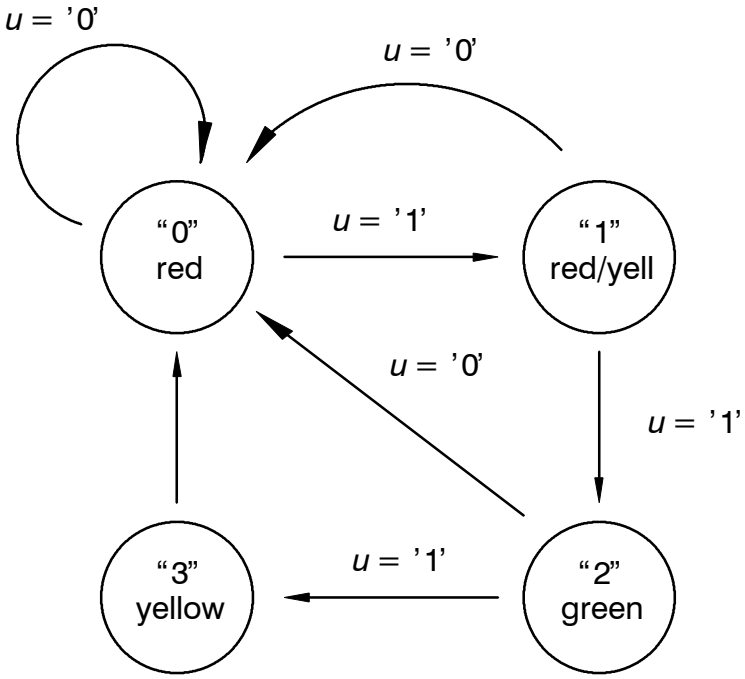


Figure 1.43: State diagram for the traffic light controller

***

Lab #02: Down Counter

7.6 Down Counter (3 Bit)

Digital control systems consist of sequential logic. A clock signal is used to advance from one state to the next.

This lab requires minimized logic for the combinational logic blocks of the state machine (Moore machine).

A three bit down counter operates as follows:

$$000 - 111 - 110 - 101 - 100 - 011 - 010 - 001 - 000 - 111 - \ldots$$
$$0_{10} - 7_{10} - 6_{10} - 5_{10} - 4_{10} - 3_{10} - 2_{10} - 1_{10} - 0_{10} - 7_{10}$$

This is called a free-running counter. After 000 the counter wraps to 111.

The state diagram is shown in fig. 1.17.

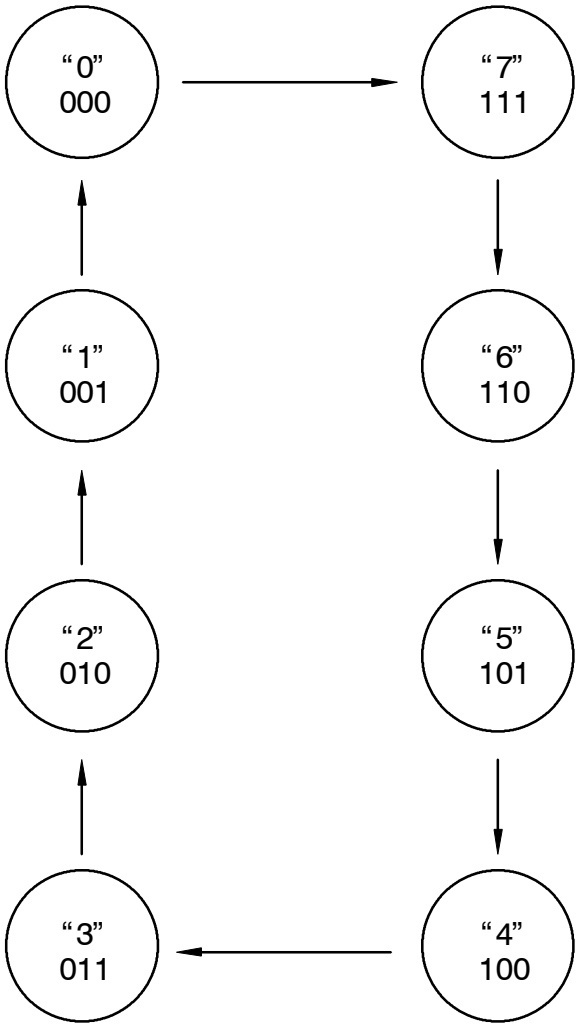
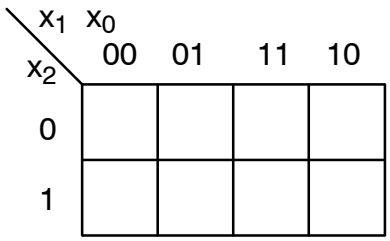


Figure 1.44: State diagram for the 3 bit down counter

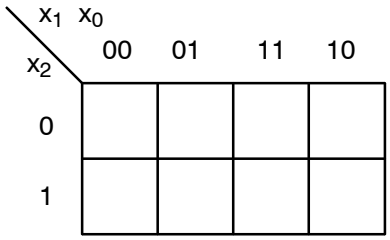
The combinational block $F(x)$ (Next-State-Logic) has to be designed and to verified in hardware. The output logic block $G(x)$ is not required since the FF outputs are identical to the outputs of the state machine.

- Write down the truth table for x2n, x1n and x0n.
- Minimize the functions by using Karnaugh maps. Write down the boolean equations for all outputs.



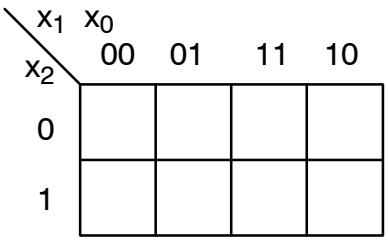
$x_{2n} =$

Figure 1.45: Karnaugh map for x_{2n}



$x_{1n} =$

Figure 1.46: Karnaugh map for x_{1n}

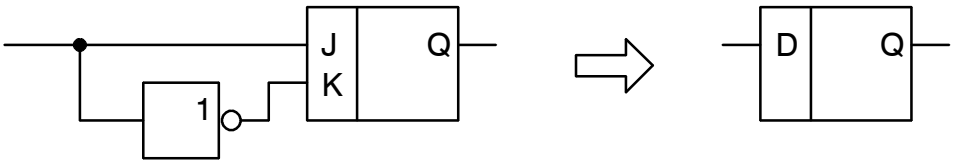


$x_{0n} =$

Figure 1.47: Karnaugh map for x_{0n}

- Draw the complete circuit.
- Write the complete circuit in hardware and verify proper operation.
  The push button will be the clock source. Use LEDs for the outputs.

Hint 1: Instead of D-FFs only JK-FFs are available. The following circuit converts a JK-FF into a D-FF.:



Hint 2: The JK-FF changes its output if J and K are left open (toggle mode).

***

Section 6

8 PicoBlaze Microcontroller

The PicoBlaze™ cpu will be served as an example to integrate an 8 bit microcontroller into an application. PicoBlaze is a highly optimized design. It is property of Xilinx Inc. but it is open source and “zero cost”, which means it can be included freely into any system. Since it takes advantage of LUT primitives (look-up tables) and BRAM (block RAM) it should be targeted to the FPGA families it was designed for. The required resources are very low, several microcontrollers can be included even in small devices.

PicoBlaze is a modern RISC type cpu with a symmetric instruction set. Symmetric means that every instruction can be applied to all registers. All operations arithmetic and logic operations can be carried out within 2 x 16 registers (register bank A and register bank B). The remaining instructions involve flow control and simple LOAD/STORE to and from memory and IO ports.

PicoBlaze has been developed in a number of versions. The latest version is KCPMS6 which stands for

(K)Constant Coded Programmable State Machine version 6.

This indicates that PicoBlaze is a finite state machine. The developer of the architecture, coding, and software tools is Ken Chapman, so the name can also be interpreted as

Ken Chapman’s Programmable State Machine version 6.

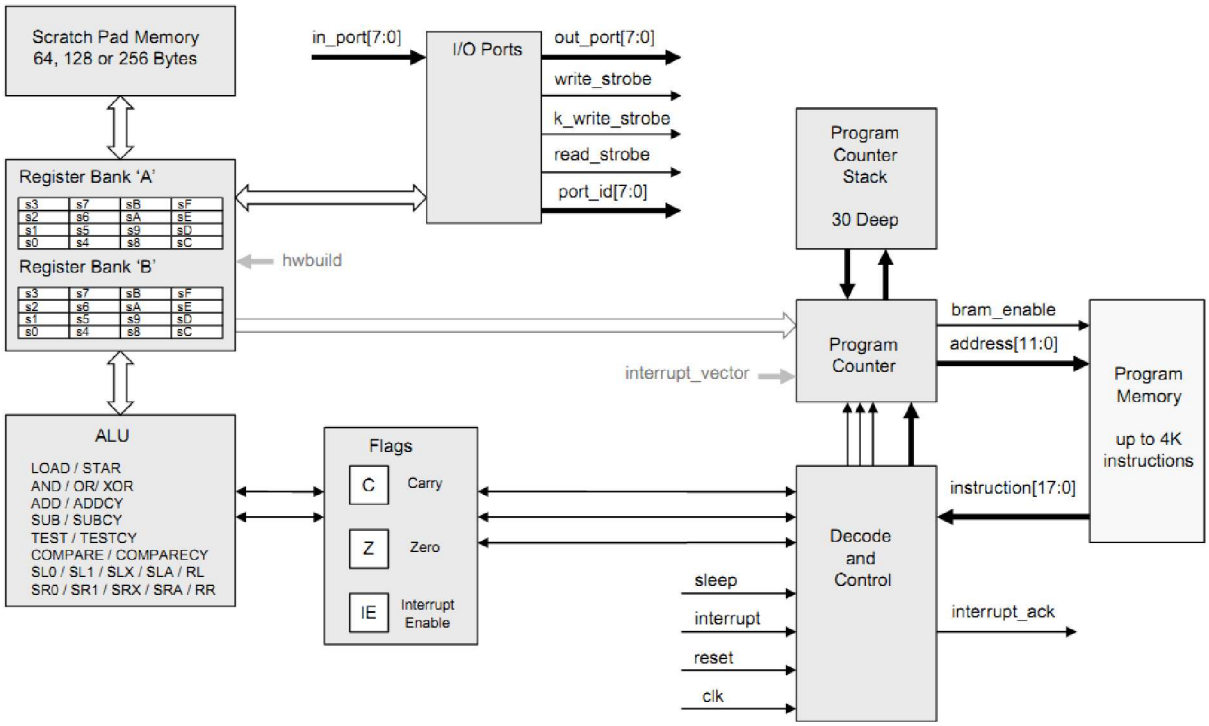


Figure 1.48: PicoBlaze KCPMS6 architecture and IO (© Xilinx)

The features can be explained by the blocks in fig. 1.48. New features in version 6 are marked in **bold** letters.

- The central component is the "Decode and Control" block (state machine). It can operate at high clock rates (**100 MHz and above**). The PicoBlaze execution time for all instructions is 2 clock cycles. With a 100 MHz *clk* signal it executes 50 million instructions per second.
  The signals of the "Decode and Control" block:

reset	clears registers and flags and starts execution at the address 0x000.
interrupt/ *interrupt_ack*	interrupts execution and starts at a predefined address which can be specified during instantiation of the processor component. The output interrupt_ack indicates that the interrupt has been serviced. External logic should reset interrupt if interrupt_ack is set.
sleep	The sleep signal stops program execution. The duration is unlimited.
instruction	18 bit instruction code from the block RAM (program memory). Every instruction is 18 bits wide.

- Program Counter
  The program counter is 10, **11, or 12 bits** wide, thus allowing to interface 1K, **2K, or 4K**

instruction (18 b its wide). Not all FPGA families are suited for all sizes. On Spartan-6 the number of instructions should be limited to 2K (11 bits).

- Program memory
  Program memory can be up to **4K instructions**. Memory blocks are in most devices organized as 2KByte memory. Since the BRAM supports a parity bit and it can be organized as 1024 x 18 (instead of 2048 x 8 or 2048 x 9) the 18 bit instruction word fits perfectly into a BRAM. More than 1K instruction require more BRAM blocks. Small size FPGA have approximately 20 BRAM blocks.

- Program counter stack
  The stack captures 30 return addresses. This means that the number of calls within calls must not exceed 30. Please note that an interrupt requires also one stack word. Interrupts should never allowed within interrupts. This is one of the best methods to crash a microcontroller program!

- Register banks
  PicoBlaze contains now **two independent register banks** of 16 registers each. Transfer of registers between banks is possible with the **new STAR (store alternate register) instruction**.

- Scratch pad memory
  This is the RAM area for PicoBlaze. The size is now configurable from 64 **up to 256 bytes**.

- ALU
  The arithmetic logic unit carries out all logic and numeric operations. **Some new instructions have been added to improve usability (e.g. COMPARECY).**

- FLAGS
  Flags are used for program flow control (conditional jumps, calls and returns). **The new HWBUILD instructions allows to set the C (Carry) flag (not the only function of HWBUILD).**

- IO ports
  Up to 256 ports (input and output) allow interfacing to external logic or pins without extra logic (multiplexer or demultiplexer). Strobe signals are provided to indicate that data is written or read (important for some external interfaces). **The new k_write_strobe signal indicates a write cycle of a constant.**

The microcontroller hardware is programmed into a Spartan-6 FPGA (equivalent to approx. 200,000 gates) or Zynq-7000 FPGA The target boards are shown in fig. 1.49. and fig. 1.50.

Figure 1.49: Spartan-6 FPGA board (FPGA is located in the center of the board)

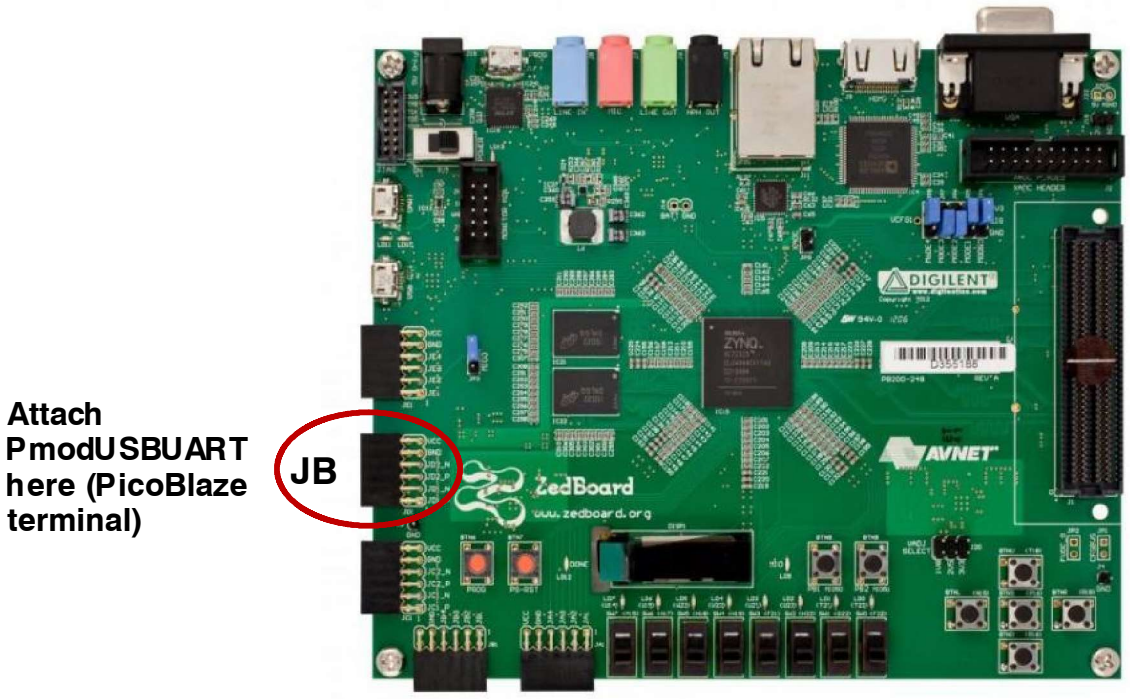


Figure 1.50: Zynq-7000 FPGA board (FPGA is located in the center of the board)

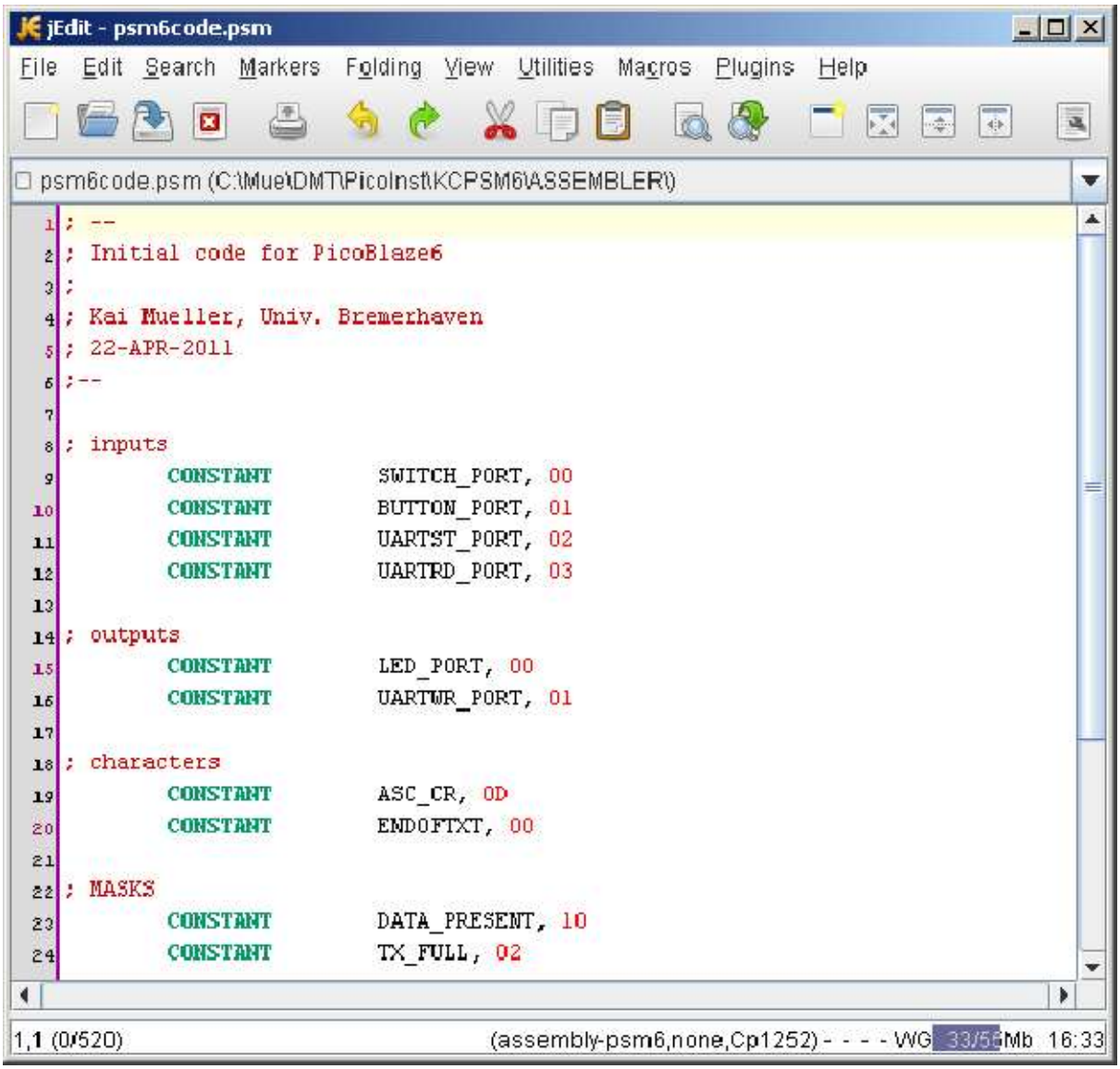


Figure 1.51: Editing an assembly file (.psm) with Jedit

According to CONSTANT definitions the following ports have been hard-coded in VHDL:

Adresse [hex]	INPUT	OUTPUT	OUTPUT (const.) OUTPUTK
00	UART_STAT_port	– –	– –
01	UART_RX6_port	UART_TX6_port	RESET_UART_port
02	SWITCH_port	LED_port	– –
03	PUSHBUT_port	– –	– –
04	DIN4_port	DOUT4_port	– –

Port bit assignments:

UART_STAT_port (RD, ADDRESS=**00**):

7	6	5	4	3	2	1	0
0	0	RX full	RX half full	RX data present	TX full	TX half full	TX data present

UART_RX6_port (RD, ADDRESS=**01**):

7	6	5	4	3	2	1	0
RX7	RX6	RX5	RX4	RX3	RX2	RX1	RX0

SWITCH_port (RD, ADDRESS=**02**):

7	6	5	4	3	2	1	0
SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0

PUSHBUT_port (RD, ADDRESS=**03**): (updated, **PBU is PicoBlaze reset**)

7	6	5	4	3	2	1	0
0	0	0	0	PBL	PNC	PBR	PBD

DIN4_port (RD, ADDRESS=**04**, see fig. 1.52):

7	6	5	4	3	2	1	0
0	0	0	0	JB8	JB7	JB2	JB1

UART_TX6_port (WR, ADDRESS=**01**):

7	6	5	4	3	2	1	0
TX7	TX6	TX5	TX4	TX3	TX2	TX1	TX0

LED_port (WR, ADDRESS=**02**):

7	6	5	4	3	2	1	0
LED7	LED6	LED5	LED4	LED3	LED2	LED1	LED0

DOUT4_port routed through JB3, JB4, JB9, JB10 (WR, ADDRESS=**04**, see fig. 1.52):

7	6	5	4	3	2	1	0
X	X	X	X	X	b2h	b1h	ena

RESET_UART_port (WR CONSTANT, ADDRESS=**01**):

7	6	5	4	3	2	1	0
X	X	X	X	X	X	RX reset	TX reset

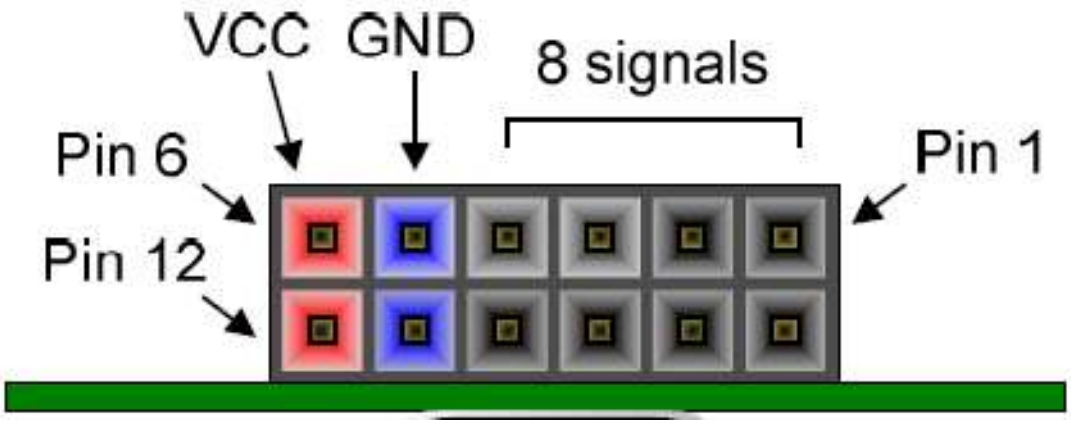


Figure 1.52: PMOD connector pin assignments

Lab #03: Programming of logical functions (part of exam)

8.1 Programming logical functions: NOT, ON, OFF, FLIP

Write a PicoBlaze™ program in assembly language which performs basis logical functions. An 8 bit value from the switches (port address 02) should be read into the controller.

The logical functions are determined by push buttons (port address 03).

The result is the functions should be written to the LEDs (port address 02).

Logical functions:

- no button pressed ⇒ LEDs show switch values (1 to 1 copy),
- Button "right" pressed (PBR) ⇒ invert byte (1's complement),
- Button "left" pressed (PBL) ⇒ all LED's off,
- Button "down" pressed (PBD) ⇒ all LED's on,
- Button "center" pressed (PBC) ⇒ "Flip" operation (sequence of bits reversed)

bit in		bit out:
0	→	7
1	→	6
2	→	5
3	→	4
4	→	3
5	→	2
6	→	1
7	→	0

Obviously "flip" is the hardest thing to code. A suitable structure for this is the use of a subprogram:

```
                ...
                CALL        flip
                ...
                JUMP        somewhere

; subprogram to flip byte in s2
flip:           ...
                RETURN
```

***

Lab #04: Moving Light (part of exam)

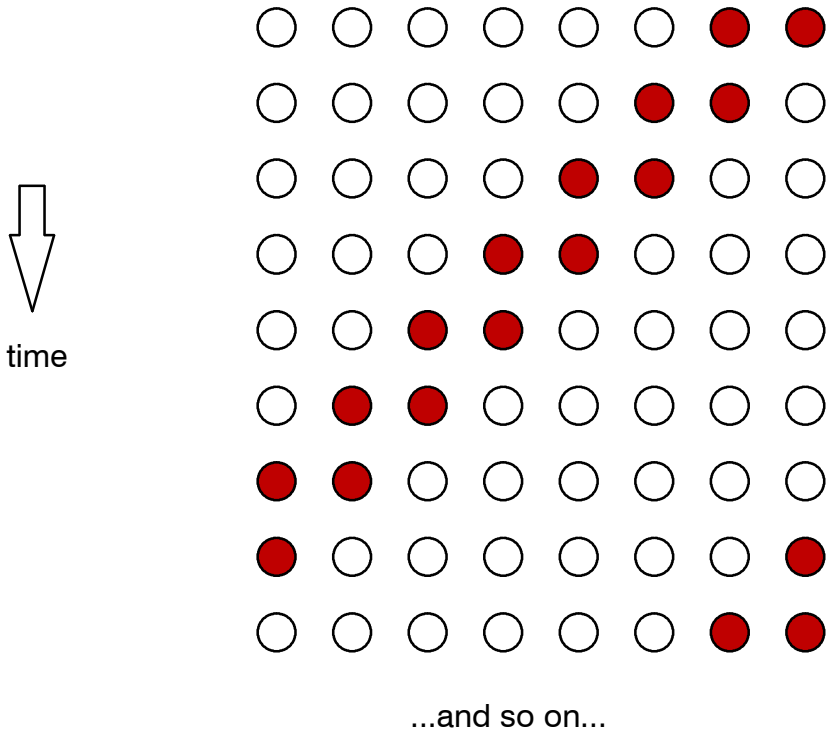
8.2 Moving LEDs With Different Patterns

Write a program for the PicoBlaze microcontroller that implements a moving light pattern. The pattern should be send to the LEDs (post address 02).

The pattern is loaded from the switches (port address 02) by activating pushbutton “center” (PBC in port address 03).

- Frequency of pattern change: 1-2Hz
- Possible new feature: Reverse the pattern movement

Example:



Please draw a **Nassi-Shneiderman-Diagram**, to describe your algorithm. This ensures a clean software design process.

***

9 Bibliography

[1] Peter J. Ashenden: The Designer's Guide to VHDL, 3rd. Ed.
Morgan Kaufmann, 2008

[2] John F. Wakerly: Digital Design, Principles & Practices.
Prentice Hall, 2001

[3] Volnei A. Predroni: Circuit Design and Simulation with VHDL, 2nd. Ed.
MIT Press, 2010

[4] Roger Lipsett, Carls Schaefer, Cary Ussery: VHDL Hardware Description and Design.
Kluwer Academic 1990

[5] Sudhakar Yalamanchili: VHDL Starter's Guide.
Prantice Hall, 1998

[6] J. Reichardt, B. Schwarz: VHDL-Synthese.
Oldenbourg, 2001

***