

## 1. KOMBINATORISCHE LOGIK

Einfache logische Operationen ohne Speicher.

Die Ausgänge ändern sich nur in Abhängigkeit von den Eingängen. Jeder Ausgang  $y_i$  lässt sich durch eine boolesche Funktion  $f_i$  aller Eingänge beschreiben:  $y_i = f_i(x_0, x_1, \dots, x_n)$

Für  $N$  Eingänge gibt es  $2^N$  Eingangskombinationen.

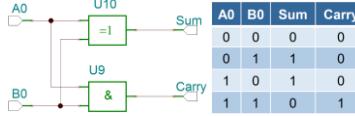
### Logische Operatoren

Function	Boolean Algebra	IEC 60617-12 since 1997
AND	$A \& B$	
OR	$A \# B$	
Buffer	$A$	
XOR	$A \$ B$	
NOT	$\text{!}A$	
NAND	$\text{!}(A \& B)$	
NOR	$\text{!}(A \# B)$	
XNOR	$\text{!}(A \$ B)$	

$A$	$B$	$\text{!}A$	$A \& B$	$A \# B$	$A \$ B$
0	0	1	0	0	1
0	1	1	0	1	0
1	0	0	0	1	0
1	1	0	1	1	1

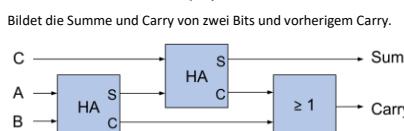
### 1-Bit Halb-Addierer (HA)

Bildet die Summe und Carry von zwei Bits.



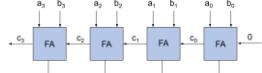
### 1-Bit Voll-Addierer (FA)

Bildet die Summe und Carry von zwei Bits und vorherigem Carry.



$C$	$A$	$B$	Sum	Carry
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

### 4-Bit Addierer

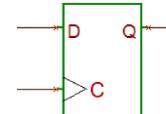


## 2. SEQUENTIELLE LOGIK

Sequentielle Logik hat gegenüber der Kombinatorischen Logik mehrere Zustände und enthält Speicher. Grundelement dafür sind D-Flip-Flops.

### D-Flip-Flop

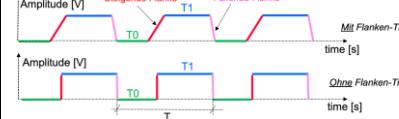
Wert am Eingang  $D$  wird gespeichert und an den Ausgang  $Q$  übertragen, wenn  $C$  von 0 auf 1 wechselt.



Ein Flip-Flop hat 2 Zustände.

$N$  Flip-Flops haben  $2^N$  Zustände.

### Clock Signal



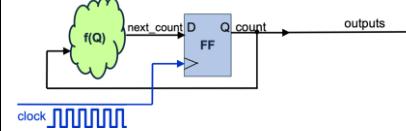
Periode  $T = T_0 + T_1[s]$

Frequent  $f = \frac{1}{T}[\text{Hz}]$

Duty Cycle  $\frac{T_1}{T}$

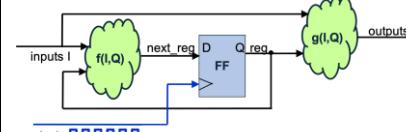
### Zähler (Counter)

Reihenfolge der Zustände und Zustand vom Ausgang hängt vom internen Zustand/Logik ab.



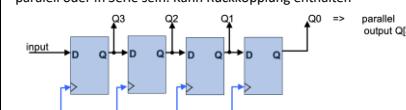
### Zustandsautomaten (Finite State Machine / FSM)

Der Ausgang ist abhängig vom Input und dem Status des Speichers. Der FF-Ausgangswert  $Q$  entspricht dem Zustand des Automaten.



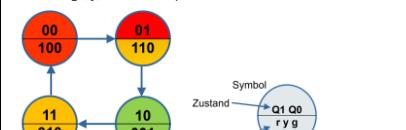
### Schieberegister

Jedes FF verzögert den Input um einen Takt. Schieberegister können parallel oder in Serie sein. Kann Rückkopplung enthalten.



### Zustandsdiagramm (Bsp. Ampel)

Um ein Zustandsautomat von einem Zustandsdiagramm zu bauen, muss man die Funktion  $f(Q)$  für den nächsten Zustand und die Funktion  $g(Q)$  für den Output ermitteln.



### Zustandslogik / Ausgangslogik

$Q_1$	$Q_0$	$D_1$	$D_0$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

state	$Q_1$	$Q_0$	red_on	yellow_on	green_on
0	0	0	1	0	0
1	0	1	0	1	0
2	1	0	0	0	1
3	1	1	1	1	0

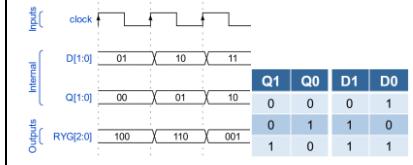
$DO = !Q_0$ ,  $D1 = Q_0 \# Q_1$

red =  $!Q_1$ , yellow =  $Q_0$ , green =  $!Q_0 \& Q_1$

### Zeitverlaufsdiagramme / Vektoren

Signal wechselt von 0 nach 1	
Signal wechselt von 1 nach 0	
Vektor (= Bus oder Signalgruppe) wechselt den Wert (MSB-LSB)	
Vektor wechselt von unbekanntem in definierten Wert	
Vektor wechselt von bekanntem in undefinierten Wert	

Bsp.:



### 3. ZAHLENSYSTEME

4 Bit -> Nibble

8 Bit -> Byte (Octet)

Dec	Bin	Hex	Oct
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

### Stellenwertsysteme

Um in Zahlen in dezimal umzurechnen, kann folgende formel verwendet werden, wobei  $i$  die Stelle (begonnen bei 0),  $b$  die Basis und  $a_i$  die Ziffer an Stelle  $i$ :

$$Z = \sum a_i \cdot b^i$$

Das Schieben um  $s$  Stellen wird wie folgt berechnet:

$$Z_{\text{neu}} = Z \cdot b^s$$

### Addition

$$\begin{array}{r} \text{Erster Summand} & 1 \ 1 \ 0 \ 1 \ . \ 0 \\ \text{Zweiter Summand} & + \ 1 \ 0 \ 1 \ 1 \ . \ 1 \\ \text{Carry} & \text{Sum} \\ \hline & 1 \ 1 \ 0 \ 0 \ . \ 1 \end{array}$$

### Subtraktion

$$\begin{array}{r} \text{Minuend} & 1 \ 1 \ 0 \ 1 \ . \ 0 \\ \text{Subtrahend} & - \ 1 \ 0 \ 1 \ 1 \ . \ 1 \\ \text{Carry} & \text{Sum} \\ \hline & 1 \ 1 \ 1 \ . \ 1 \end{array}$$

### Differenz

$$0 \ 0 \ 0 \ 1 \ . \ 1$$

### Multiplication

$$\begin{array}{r} \text{Faktoren} & 1 \ 0 \ 1 \times 1 \ 1 \ 1 \ 0 \\ & 1 \ 1 \ 1 \ 0 \\ & + 0 \ 0 \ 0 \ 0 \\ & + 1 \ 1 \ 1 \ 0 \\ \hline & 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \end{array}$$

### Division

$$\begin{array}{r} \text{Divisor} \div \text{Dividend} & 1 \ 1 \ 0 \ 1 \ 1 \ 0 \div 1 \ 0 \ 1 \ 0 = 1 \ 0 \ 1 \\ & - 1 \ 0 \ 1 \ 0 \\ & 0 \ 0 \ 1 \ 1 \ 1 \\ & - 0 \ 0 \ 0 \ 0 \\ & 0 \ 1 \ 1 \ 1 \ 0 \\ & - 1 \ 0 \ 1 \ 0 \\ \hline & 0 \ 1 \ 0 \ 0 \end{array}$$

### Darstellung negativer Zahlen

Binär	Dezimal	Sign+Magn.	Einerkompl.	Zweierkompl.	Excess-8
1111	-15	-7	0 - 0	+ -1	0 - 7
1110	14	-6	1 - 1	+ -2	+ 6
1101	13	-5	2 - 2	+ -3	+ 5
1100	12	-4	3 - 3	+ -4	+ 4
1011	11	-3	4 - 4	+ -5	+ 3
1010	10	-2	5 - 5	+ -6	+ 2
1001	9	-1	6 - 6	+ -7	+ 1
1000	8	0	7 - 7	+ -8	0
0111	7	+7	8 + 7	+ +7	-1
0110	6	+6	7 + 6	+ +6	-2
0101	5	+5	6 + 5	+ +5	-3
0100	4	+4	5 + 4	+ +4	-4
0011	3	+3	4 + 3	+ +3	-5
0010	2	+2	3 + 2	+ +2	-6
0001	1	+1	2 + 1	+ +1	-7
0000	0	+0	1 + 0	+ +0	-8

### Zeitverlaufsdiagramme (Vorzeichenwechsel)

#### Verfahren:

$$\begin{array}{l} +2 \rightarrow -2 \\ 00000010 \\ \bullet \text{ invertieren: } 11111110 \\ \bullet \text{ 1 addieren: } 00000001 \end{array}$$

$$\begin{array}{l} -2 \rightarrow +2 \\ 11111110 \\ \bullet \text{ invertieren: } 00000001 \\ \bullet \text{ 1 addieren: } 11111110 \end{array}$$

#### Spezialfälle:

$$\begin{array}{l} 0 \rightarrow 0 \\ 00000000 \\ \bullet \text{ invertieren: } 11111111 \\ \bullet \text{ 1 addieren: } 00000001 \end{array}$$

$$\begin{array}{l} -128 \rightarrow \text{Überlauf} \\ 1000000000 \\ \bullet \text{ invertieren: } 0111111111 \\ \bullet \text{ 1 addieren: } 0000000001 \end{array}$$

$$\begin{array}{l} 0 \rightarrow 0 \\ 00000000 \\ \bullet \text{ invertieren: } 11111111 \\ \bullet \text{ 1 addieren: } 00000000 \end{array}$$

$$\begin{array}{l} 1000000000 \\ \bullet \text{ invertieren: } 0111111111 \\ \bullet \text{ 1 addieren: } 1000000000 \end{array}$$

$$\begin{array}{l} 0 \rightarrow 0 \\ 00000000 \\ \bullet \text{ invertieren: } 11111111 \\ \bullet \text{ 1 addieren: } 00000000 \end{array}$$

$$\begin{array}{l} 1000000000 \\ \bullet \text{ invertieren: } 0111111111 \\ \bullet \text{ 1 addieren: } 1000000000 \end{array}$$

$$\begin{array}{l} 0 \rightarrow 0 \\ 00000000 \\ \bullet \text{ invertieren: } 11111111 \\ \bullet \text{ 1 addieren: } 00000000 \end{array}$$

$$\begin{array}{l} 1000000000 \\ \bullet \text{ invertieren: } 0111111111 \\ \bullet \text{ 1 addieren: } 1000000000 \end{array}$$

$$\begin{array}{l} 0 \rightarrow 0 \\ 00000000 \\ \bullet \text{ invertieren: } 11111111 \\ \bullet \text{ 1 addieren: } 00000000 \end{array}$$

$$\begin{array}{l} 1000000000 \\ \bullet \text{ invertieren: } 0111111111 \\ \bullet \text{ 1 addieren: } 1000000000 \end{array}$$

$$\begin{array}{l} 0 \rightarrow 0 \\ 00000000 \\ \bullet \text{ invertieren: } 11111111 \\ \bullet \text{ 1 addieren: } 00000000 \end{array}$$

$$\begin{array}{l} 1000000000 \\ \bullet \text{ invertieren: } 0111111111 \\ \bullet \text{ 1 addieren: } 1000000000 \end{array}$$

$$\begin{array}{l} 0 \rightarrow 0 \\ 00000000 \\ \bullet \text{ invertieren: } 11111111 \\ \bullet \text{ 1 addieren: } 00000000 \end{array}$$

$$\begin{array}{l} 1000000000 \\ \bullet \text{ invertieren: } 0111111111 \\ \bullet \text{ 1 addieren: } 1000000000 \end{array}</math$$

## Redundanz von Codes

Der **durchschnittliche Codewortlänge**  $L$  eines Codes wird durch die Formel

$$L = \sum_{i=1}^N p(s_i) \cdot \ell_i \left[ \frac{\text{Bit}}{\text{Symbol}} \right]$$

bestimmt,  $\ell_i$  die Länge des entsprechenden Codeworts ist.

Die **Redundanz**  $R$  wird definiert durch die Formel

$$R = L - H(X) \left[ \frac{\text{Bit}}{\text{Symbol}} \right]$$

$R > 0$  Code kann noch verlustfrei komprimiert werden

$R = 0$  Code kann nicht mehr verlustfrei komprimiert werden

$R < 0$  Code wird verlustbehaftet komprimiert

## Kompressionsrate

Die Kompressionsrate  $CR$  eines Codierungsverfahrens wird definiert als das Verhältnis der Größe der ursprünglichen Daten  $D_{\text{orig}}$  zur Größe der komprimierten Daten  $D_{\text{comp}}$ :

$$CR = \frac{D_{\text{orig}}}{D_{\text{comp}}}$$

Eine höhere Kompressionsrate bedeutet eine stärkere Reduktion der Datensumme.

## Run Length Encoding (RLE)

Runs werden mit Tokens codiert: (Marker, Anzahl, Code). Als Marker wird ein wenig genutzter Code verwendet. Dieses muss dafür immer codiert werden.

Bsp: ...TERRRRRRRRMUGGWQCSSSSSSSSSL...

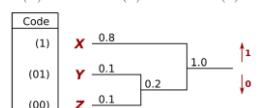
Komprimiert: ...TERA08RMA01AU02GWCA10SL...

## Huffman Code

Die Huffman-Codierung ist ein Verfahren zur optimalen Quellencodierung, das darauf abzielt, die durchschnittliche Codewortlänge zu minimieren. Der Algorithmus funktioniert wie folgt:

1. Liste aller Symbole mit ihren Wahrscheinlichkeiten erstellen.
2. Die beiden Symbole mit den kleinsten Wahrscheinlichkeiten auswählen und zu einem neuen Symbol zusammenfassen, dessen Wahrscheinlichkeit die Summe der beiden ist.
3. Diesen Vorgang wiederholen, bis nur noch ein Symbol übrig ist.
4. Den Baum von unten nach oben durchgehen und jedem Symbol einen Code zuweisen, wobei links eine 0 und rechts eine 1 hinzugefügt wird.

$$P(X) = 0.80 \quad P(Y) = 0.10 \quad P(Z) = 0.10$$

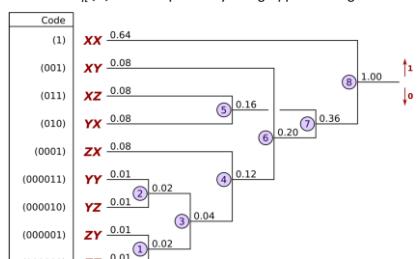


Um die Redundanz weiter zu reduzieren, kann die Huffman-Codierung auf Symbolgruppen (z.B. Paare oder Tripel von Symbolen) angewendet werden, anstatt auf einzelne Symbole. Dafür gilt:

$$p(x_1, x_2, \dots, x_n) = p(x_1) \cdot p(x_2) \cdot \dots \cdot p(x_n)$$

$$H_n(X) = n \cdot H(X) \left[ \frac{\text{Bit}}{\text{n Symbole}} \right]$$

Hierbei ist  $H_n(X)$  die Entropie der Symbolgruppe der Länge  $n$ .

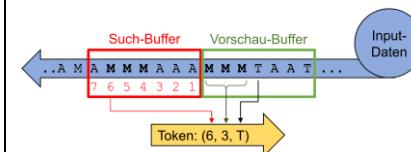


$H, L, R$  sind in diesem Bsp. nun Bit/2 Symbole.

## LZ77 (Lempel Ziv)

Die LZ77-Codierung ist ein verlustfreies Kompressionsverfahren, das auf der Ersetzung von wiederkehrenden Mustern durch Verweise basiert:

1. Suche im Suchbuffer nach der längsten Übereinstimmung mit dem Anfang des Lookahead-Buffers.
2. Erstelle ein Tripel (Offset, Länge, Nächstes Symbol), wobei:
  - a. **Offset:** die Position der Übereinstimmung im Suchbuffer ist,
  - b. **Länge:** die Länge der Übereinstimmung ist,
  - c. **Nächstes Symbol:** das erste Symbol im Lookahead-Buffer nach der Übereinstimmung ist.
  - d. Bei keiner Übereinstimmung: (0, 0, Nächstes Symbol)
3. Verschiebe den Suchbuffer und den Lookahead-Buffer entsprechend der Länge der Übereinstimmung und des nächsten Symbols.
4. Wiederhole den Vorgang, bis alle Daten codiert sind.



Bsp:

Such-Buffer		Vorschau-Buffer
10	9	A M A M M M A A A
8	7	M M M T A A T ...
7	6	5 4 3 2 1
6	5	4 3 2 1
5	4	3 2 1
4	3	2 1
3	2	1
2	1	
1		

Offset	Länge	Symbol
0	0	A
0	0	M
2	2	M
4	2	A
6	4	T

Bit ganz rechts im Vorschau-Buffer dürfen nicht codiert werden.

Für den Abschluss des Codes muss eine Lösung definiert werden (Token / Größe im Header)

## LZW (Lempel Ziv Welch)

Die LZW-Codierung (Lempel-Ziv-Welch) ist ein weiteres verlustfreies Kompressionsverfahren, das auf der Erstellung eines Wörterbuchs basiert:

1. Initialisiere das Wörterbuch mit allen möglichen Symbolen der Eingabedaten oder andere Charsets (ASCII).
2. Lese die Eingabedaten und finde die längste Zeichenkette, die im Wörterbuch vorhanden ist.
3. Gib den Index dieser Zeichenkette im Wörterbuch aus.
4. Füge die längste Zeichenkette plus das nächste Symbol zur Eingabe zum Wörterbuch hinzu.
5. Wiederhole den Vorgang, bis alle Daten codiert sind.

Bsp: AMAMMMAAAAMMMTAAT

Index	Eintrag	Token
0...255	ASCII	
256	AM	65 (A)
257	MA	77 (M)
258	AMM	256 (AM)
259	MM	77 (M)
260	MAA	257 (MA)
261	AA	65 (A)
262	AMMM	258 (AMM)
...	...	...