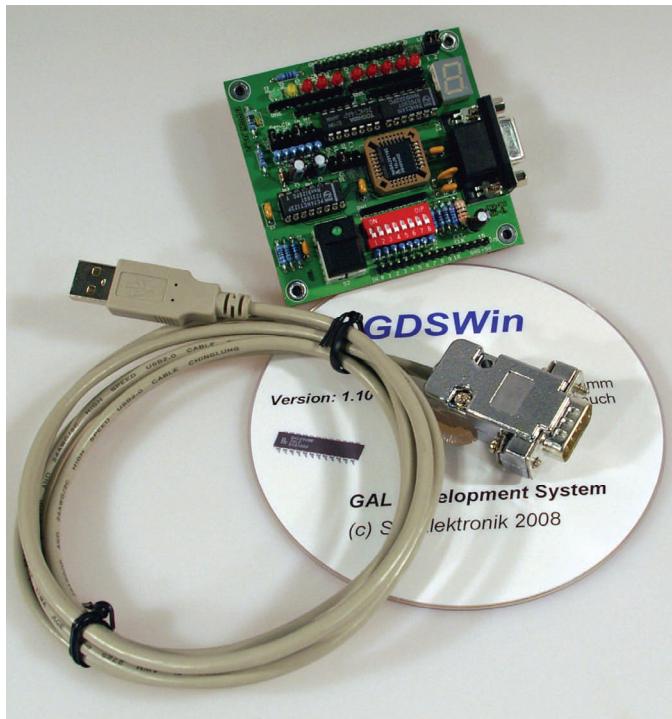


Gal-Tutorium

Einführung

in die programmierbare Logik



Kapitel 1: Vorbemerkungen

1.1. Einige Anmerkungen zum internen Aufbau von GALs

Schaut man sich (z.B. im Internet) den prinzipiellen inneren Aufbau des GAL22V10 (oder auch der ähnlichen Typen GAL18V10, GAL16V8 bzw. GAL20V8) an, so fällt zunächst eine Zweiteilung in einen großen kombinatorischen Bereich und einen Bereich für sequenzielle Logik auf, der in sogenannten Output-Logic-Macro-Cells (OLMCs) taktflanken-getriggerte D-Flip-Flops enthält.

Im kombinatorischen Bereich befindet sich eine große Verknüpfungsmatrix für Wired-And-Verknüpfungen für logische UND-Funktionen sowohl zwischen allen Eingängen als auch zusätzlich zwischen allen intern rückgeführten Ausgängen. Vor dem Dateneingang jeder OLMC befindet sich dann ein ODER-Gatter mit unterschiedlicher Anzahl der Eingänge, die vom jeweiligen Ausgangspin abhängt, welches für die Ausgänge der Wired-And-Matrix mögliche ODER-Verknüpfungen zur Verfügung stellt. Insofern ergibt sich im kombinatorischen Bereich durch den internen Aufbau des GALs eine Struktur, die ideal für die sogenannte disjunktive Normalform von Logikgleichungen geeignet ist. „Disjunktive Normalform“ bedeutet dabei, dass eine logische Verknüpfungsgleichung für mehrere Boolesche Eingangsvariablen die Struktur einer ODER-Verknüpfung von UND-Termen besitzt. (Durch die zusätzliche Möglichkeit von logischen Invertierungen an allen Ein- und Ausgängen wird prinzipiell auch die Umsetzung von Gleichungen in konjunktiver Normalform ermöglicht.)

Insofern wird innerhalb dieses Tutoriums bei der systematischen Erstellung von Logikgleichungen die disjunktive Normalform verwendet.

Bei der Realisierung von Schaltungen der sequenziellen Logik mit Hilfe eines GALs ergibt sich dann aufgrund der inneren Struktur immer ein sog. synchron getaktetes Schaltwerk, da die Takteingänge der D-Flip-Flops aller OLMCs intern zu einem gemeinsamen Takteingang verbunden sind; insofern erhalten alle verwendeten D-Flip-Flops innerhalb eines GALs zwangsweise ein synchrones Taktsignal. Dadurch werden von vorn herein jegliche Probleme, die sich aus Taktlaufzeiten bei asynchron getakteten Schaltwerken ergeben können, konsequent vermieden. Bei Einsatz des D-Flip-Flops innerhalb einer OLMC wird der Ausgang der kombinatorischen Matrix automatisch als Eingangssignal für den Dateneingang des Flip-Flops verwendet.

1.2. Systematisches Aufstellen von Logikgleichungen

a) kombinatorische Logik (bzw. Schaltnetze)

Im Rahmen der sog. kombinatorischen Logik stellt sich im Normalfall das Problem, dass man zur Festlegung der logischen Funktion eines Ausgangs die Abhängigkeit des logischen Ausgangszustands von allen relevanten logischen Eingangsvariablen in Form Boolescher Logikgleichungen angeben muss. Um hier zu einer systematischen Vorgehensweise zu gelangen, geht man am besten wie folgt vor: Zunächst fasst man in einer Tabelle alle auftretenden möglichen logischen Kombinationen der Eingangsvariablen zusammen und ordnet jeder Eingangskombination den gewünschten logischen Zustands der Ausgangsvariablen zu (siehe auch folgendes Beispiel). (Da eine boolesche Funktion endlich vieler Eingangsvariablen immer nur endlich viele Ausgangszustände besitzen kann, ist es immer möglich, diese Funktion in Form einer Tabelle eindeutig zu beschreiben.)

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Nachdem die Funktion in tabellierter Form vorliegt, sucht man diejenigen Zeilen auf, in denen die Ausgangsvariable eine logische 1 besitzt. Von jeder dieser Zeilen bildet man die UND-Verknüpfung aller Eingangsvariablen; und zwar verwendet man die Eingangsvariable für diese UND-Verknüpfung nicht invertiert, falls sie in der betreffenden Zeile den Wert 1 besitzt, bzw. invertiert, falls sie den Wert 0 besitzt. Anschließend werden dann alle UND-Terme der verschiedenen Zeilen durch eine ODER-Verknüpfung zusammengefasst. Für die nebenstehende Tabelle ergibt sich auf diese Weise folgende Logikgleichung für den Ausgang Q:

$$Q = \bar{A} * B * \bar{C} + A * \bar{B} * \bar{C} + A * B * \bar{C} .$$

Unter Umständen ist dann abschließend noch eine logische Vereinfachung der Gleichung z.B. unter Zuhilfenahme der Rechenregeln der Booleschen Algebra oder auch durch den Einsatz eines KV-Diagramms, welches sich hervorragend zur Vereinfachung von disjunktiven Normalformen eignet, möglich. Im obigen Beispiel fällt bei den zwei ersten UND-Termen die Verknüpfung mit \bar{A} bzw. A und bei den beiden letzten UND-Termen die Verknüpfung von B und \bar{B} auf, so dass die vereinfachte Gleichung für Q folgendermaßen lautet:

$$Q = B * \bar{C} + A * \bar{C} .$$

Alle folgenden Beispiele in diesem Tutorium gehen nach diesem Prinzip vor, um die Logikgleichungen aller Ausgänge des GALs systematisch aufzustellen.

Treten in der Logiktablette der booleschen Ausgangsfunktion mehr logische 1en als 0en auf, so kann man die Logikgleichung zunächst auch für die invertierte Ausgangsvariable nach dem oben beschriebenen Verfahren aufstellen und anschließend die Möglichkeit der logischen Ausgangsinvertierung innerhalb des GALs nutzen.

b) sequenzielle Logik (bzw. Schaltwerke)

Im Rahmen der sequenziellen Logik geht es um den Entwurf sog. Schaltwerke, d.h. es kommt nun die zeitliche Entwicklung der logischen Zustände nach Auftreten eines Taktimpulses ins Spiel. Hierzu ist es meist empfehlenswert, einen zeitlichen Ablaufplan der aufeinanderfolgenden logischen Zustände aufzustellen. Ähnlich wie bei der kombinatorischen Logik lässt sich dies in vielen Fällen ebenfalls in Form einer Tabelle realisieren, wenn man jeweils die Folgezeile als zeitliche Entwicklung der vorangehenden Zeile interpretiert.

So ergeben sich z.B. für die nachfolgende Tabelle die Logikgleichungen für die Ausgangsvariablen Q_0 , Q_1 und Q_2 wie folgt:

(Zur Vereinfachung sei davon ausgegangen, dass keine weiteren reinen Eingangsvariablen existieren.)

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0

Aus dem Zustand (0,0,0) soll nach dem nächsten Taktimpuls der Ausgang Q0 logisch 1 werden; dieses gilt ebenfalls für den Zustand (0,1,0). Daher lautet die Gleichung für Q0:

$$Q0 := \neg Q2 * \neg Q1 * \neg Q0 + \neg Q2 * Q1 * \neg Q0 ,$$

Der Ausgang Q1 soll nach den Zuständen (0,0,1) und (0,1,0) gesetzt werden:

$$Q1 := \neg Q2 * \neg Q1 * Q0 + \neg Q2 * Q1 * \neg Q0 .$$

Für den Ausgang Q2 ergibt sich nur eine logische 1, und zwar aus dem Vorgängerzustand (0,1,1):

$$Q2 := \neg Q2 * Q1 * Q0 .$$

Die Tabelle ist dann zyklisch zu verstehen, d.h. nach dem Zustand (1,0,0) folgt wieder der Zustand (0,0,0) in der ersten Zeile.

Da von den prinzipiell acht möglichen logischen Zuständen eines Systems mit drei Ausgangsvariablen in der obigen Tabelle nur fünf aufgelistet sind, kann es zu einem unerwünschten zeitlichen Ablauf kommen, falls sich das Schaltwerk aus irgendeinem Grunde – z.B. nach Einschalten der Betriebsspannung – in einem eigentlich nicht vorgesehenen Zustand befindet (siehe hierzu auch das Beispiel zum 4-Bit-BCD-Zähler).

Die Entwicklungsumgebung unterscheidet zwischen rein kombinatorischer Logik und sequenzieller Logik innerhalb einer Logikgleichung durch den Doppelpunkt vor dem Gleichheitszeichen; dabei steht ein einfaches Gleichheitszeichen ohne Doppelpunkt für die kombinatorische Verknüpfung, während ein Doppelpunkt direkt vor dem Gleichheitszeichen zum automatischen Einsatz des D-Flip-Flops der zugehörigen OLMC führt.

Kapitel 2: Kombinatorische Logik

Bei der sog. kombinatorischen Logik ist jeder Ausgang einer logischen Verknüpfung eine direkte Boolesche Funktion der Eingangsvariablen. Es kommen insbesondere keine Rückführungen von Ausgängen auf Eingänge von weiter vorn liegenden Gattern vor. Daher ergibt sich der logische Ausgangszustand unmittelbar aus dem aktuellen logischen Zustand der Eingänge. Die Schaltung zeigt infolgedessen keinerlei Speicherverhalten, oder anders ausgedrückt ist der Ausgangszustand unabhängig von der vorhergehenden zeitlichen Entwicklung der Ein- und Ausgänge. Schaltungen der kombinatorischen Logik werden oft auch „Schaltnetze“ (im Gegensatz zu „Schaltwerken“, siehe sequenzielle Logik) genannt.

In Bezug auf den GAL bedeutet dies, dass innerhalb des GALs zunächst nur die UND/ODER-Verknüpfungsmatrix nicht aber die D-Flip-Flops der OLMCs (Output Logic Macro Cell) genutzt werden.

2.1. Boolesche Funktionen

Aufgrund seiner inneren Struktur (ODER-Verknüpfung von UND-Termen plus zusätzliche Möglichkeit der logischen Negation an allen Ein- und Ausgängen) werden im GAL alle Booleschen Funktionen in der sog. disjunktiven Normalform realisiert.

Als erste Übung kann man sich daher einmal überlegen, wie viele verschiedene Boolesche Funktionen mit zwei Eingangsvariablen es prinzipiell überhaupt geben kann. Zwei Boolesche Eingangsvariablen können zusammen vier unterschiedliche Eingangskombinationen aufweisen (00, 01, 10, 11). Diesen vier Eingangskombinationen kann man auf 16 ($= 2^4$) unterschiedliche Weisen die Ausgangszustände 0 bzw. 1 zuordnen. Von diesen Zuordnungen sind allerdings sechs trivial, da sie entweder gar nicht vom Eingangszustand (0000, 1111) oder nur von einer der zwei Eingangsvariablen abhängig sind. Es verbleiben also die folgenden 10 unterschiedlichen, nicht trivialen Booleschen Funktionen von zwei Eingangsvariablen:

A	B	AND	NAND	OR	NOR	EXOR	EXNOR	A>B		B>A	
0	0	0	1	0	1	0	1	1	0	1	0
0	1	0	1	1	0	1	0	1	0	0	1
1	0	0	1	1	0	1	0	0	1	1	0
1	1	1	0	1	0	0	1	1	0	1	0

Die ersten vier (AND, NAND, OR, NOR) ergeben sich direkt aus der Tabelle in disjunktiver Normalform (ggf. mit Negation am Ausgang (NAND, NOR)).

Bei der Antivalenzverknüpfung (EXOR) liest man aus der Tabelle ab:

$EXOR = \bar{A} * B + A * \bar{B}$, entsprechend für die Äquivalenzverknüpfung (EXNOR):

$EXNOR = \bar{A} * \bar{B} + A * B$.

In ähnlicher Form lassen sich die zwei Subjunktionen (auch als Implikationsverknüpfung bezeichnet) und die Negationen dazu aus der Tabelle ablesen. (Da die Subjunktion asymmetrisch in Bezug auf Vertauschung ihrer zwei Eingänge ist, ergeben sich hier zwei unterschiedliche Fälle, nämlich $A > B$ bzw. $B > A$; alle anderen Funktionen sind symmetrisch.)

Das Programm „Boolesche_Funktionen“ nutzt alle 10 Ausgänge des GALs zur Realisierung dieser 10 unterschiedlichen Booleschen Funktionen entsprechend der o.a. Tabelle.

Beispiel-File: Boolesche_Funktionen.gds

Einstellungen auf der Platine:

Schalter 1: Eingangsvariable A,

Schalter 2: Eingangsvariable B,

Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt.

2.2. BCD-7-Segment-Dekoder

Ein erstes Beispiel zur Demonstration der großen Leistungsfähigkeit eines GALs ist eine Dekoderschaltung, die eine Ziffer im BCD-Code zur Anzeige auf einer 7-Segmentanzeige umsetzt. An diesem Beispiel kann man weiterhin gut erlernen, wie eine systematische Schaltungsvereinfachung mit Hilfe der sog. Karnaugh-Veitch-Diagramme (kurz KV-Diagramme genannt) durchgeführt werden kann.

Zunächst sei hier die vollständige Verknüpfungstabelle angegeben:

BCD-Code				Dez.-wert	7-Segment-Code						
D	C	B	A		Qa	Qb	Qc	Qd	Qe	Qf	Qg
0	0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	1	0	1	1	0	0	0	0
0	0	1	0	2	1	1	0	1	1	0	1
0	0	1	1	3	1	1	1	1	0	0	1
0	1	0	0	4	0	1	1	0	0	1	1
0	1	0	1	5	1	0	1	1	0	1	1
0	1	1	0	6	1	0	1	1	1	1	1
0	1	1	1	7	1	1	1	0	0	0	0
1	0	0	0	8	1	1	1	1	1	1	1
1	0	0	1	9	1	1	1	1	0	1	1

KV-Diagramm für den Ausgang Qa (steuert das a-Segment der Anzeige):

Qa=	A		/A		
C	1	1	1	0	/D
					D
/C	1			1	
	0	1	1	1	/D
	/B	B	/B		

Die grau unterlegten Felder sind hier nicht relevant, da die zugehörigen Eingangskombinationen im BCD-Code nicht auftreten. (Erst beim Binärcode am Eingang werden diese Felder ebenfalls belegt; siehe nächstes Beispiel.)

Aus dem obigen KV-Diagramm liest man folgende Logikgleichung ab:

$$Qa = \underline{/D * B} + \underline{/C * /B * /A} + \underline{/D * C * A} + \underline{D * /C * /B}.$$

Bei Ausgängen, die weniger Nullen als Einsen aufweisen, kann man unter Ausnutzung der möglichen Ausgangsinvertierung auch die logischen Verknüpfungen für die Nullen aus der Tabelle ablesen bzw. auch unter Verwendung des KV-Diagramms vereinfachen. Der Ausgang für das c-Segment enthält z.B. nur eine einzige Null, daher ergibt sich direkt aus der Tabelle

$$/Q_c = /D * /C * B * /A .$$

Diese Vorgehensweise liefert ganz automatisch eine Minimierung der notwendigen ODER-Verknüpfungen. Da die Anzahl der UND-Terme in der kombinatorischen Matrix der GALs im Gegensatz zur Anzahl der jeweils vorhandenen Eingänge des ODER-Gatters unbeschränkt sind, sollte man vorrangig auf eine möglichst kleine Zahl notwendiger ODER-Verknüpfungen achten. Auch für negierte Ausgänge lässt sich dies natürlich wieder mit Hilfe der KV-Diagramme realisieren.

Dazu folgt hier noch abschließend ein Beispiel für den Ausgang Qf, der in der Tabelle insgesamt sechs Einsen und vier Nullen enthält:

Qf=	A	/A			
C	1	0	1	1	/D
/C	1			1	D
	0	0	0	1	/D
	/B	B	/B		

Die sechs Einsen lassen sich zwar zu vier Zweierblöcken zusammenfassen, aber während man die vier Nullen in nur drei Zweierblöcken zusammenfassen kann. Folglich benötigt man nur drei Eingänge des ODER-Gatters, wenn man die Möglichkeit der Ausgangsinvertierung nutzt und die Logikgleichung für den invertierten Ausgang programmiert:

$$/Q_f = /D * /C * B + /D * B * A + /D * /C * A .$$

$$Q_f = /D * C * /A + /C * /B * /A + /D * C * /B + D * /C * /B .$$

Selbstverständlich lassen sich die beiden Logikgleichungen für Qf und /Qf auch konventionell unter Verwendung der de-Morganschen-Gesetze und der allgemeinen Rechenregeln der Booleschen Algebra jeweils ineinander umrechnen. Der Rechenaufwand ist dabei allerdings nicht ganz unerheblich. Auch daran lässt sich der große Vorteil der KV-Diagramme erkennen.

Dieses Beispiel zeigt das große Potenzial, das in den GALs allein schon für die rein kombinatorische Logik steckt. Wollte man die für die Codeumsetzung notwendigen Logikgleichungen durch einzelne Gatter aus der TTL-Familie realisieren, würde das einen erheblichen Schaltungsaufwand bedeuten. Bei dem Beispiel des BCD-7-Segment-Dekoders mag man noch einwenden, dass für diesen Zweck innerhalb der Logikfamilien fertige Bausteine existieren (z.B. 7447, 7448, 74247 oder auch 40XX). Das folgende Beispiel eines Codeumsetzers von einem vollen 4-Bit-Binärcode auf eine hexadezimale Anzeige ist dann jedoch schon ein Einsatzbeispiel, für das kein fertiger Logikbaustein existiert.

Beispiel-File: BCD_7Segment_DEC.gds

Einstellungen auf der Platine: Schalter 5 bis 8: Eingänge für den BCD-Code, Schalter 5 (links) hat die höchste Wertigkeit (2^3), Schalter 8 (rechts) hat die niedrigste Wertigkeit (2^0); Jumper für 7-Segment-Anzeige gesetzt, Jumper für LED-Zeile optional zusätzlich.

2.3 4-Bit-Binär-Hexadezimal-Dekoder:

Bei diesem Beispiel geht es nun in Erweiterung zu dem vorangegangenen um eine Dekoderschaltung, die alle 16 möglichen Zustände im 4-Bit-Binär-Code als Hexadezimalzeichen auf der 7-Segmentanzeige ausgibt. Dabei werden die Zahlen 0 bis 9 wie üblich angezeigt, die Anzeige der Zahlen 10 bis 15 erfolgt durch die Buchstaben von A bis F. Um eine Darstellung auf einer konventionellen 7-Segmentanzeige zu ermöglichen, werden die Zahlen 11, 12 und 13 durch die Kleinbuchstaben b, c und d angezeigt. Für diese Dekodieraufgabe ergibt sich nachstehende Codetabelle:

Dez.-wert	4-Bit-Binär-Code				Hex.-anz.	7-Segment-Code						
	D	C	B	A		Qa	Qb	Qc	Qd	Qe	Qf	Qg
0	0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	1	0	1	1	0	0	0	0
2	0	0	1	0	2	1	1	0	1	1	0	1
3	0	0	1	1	3	1	1	1	1	0	0	1
4	0	1	0	0	4	0	1	1	0	0	1	1
5	0	1	0	1	5	1	0	1	1	0	1	1
6	0	1	1	0	6	1	0	1	1	1	1	1
7	0	1	1	1	7	1	1	1	0	0	0	0
8	1	0	0	0	8	1	1	1	1	1	1	1
9	1	0	0	1	9	1	1	1	1	0	1	1
10	1	0	1	0	A	1	1	1	0	1	1	1
11	1	0	1	1	B	0	0	1	1	1	1	1
12	1	1	0	0	c	1	0	0	1	1	1	0
13	1	1	0	1	d	0	1	1	1	1	0	1
14	1	1	1	0	e	1	0	0	1	1	1	1
15	1	1	1	1	F	1	0	0	0	1	1	1

Auf die Erstellung der KV-Diagramme wird hier bewusst verzichtet, da sich dies als eigenständige Übung anbietet. Die notwendigen, minimierten Logikgleichungen gehen aus dem Beispielfile hervor.

Beispiel-File: HEX_7Segment_DEC.gds

Einstellungen auf der Platine: Schalter 5 bis 8: Eingänge für den 4-Bit-Binär-Code, Schalter 5 (links) hat die höchste Wertigkeit (2^3), Schalter 8 (rechts) hat die niedrigste Wertigkeit (2^0); Jumper für 7-Segment-Anzeige gesetzt, Jumper für LED-Zeile optional zusätzlich.

2.4 Binär-/Gray-Code-Umsetzer:

Im Gegensatz zum Binär-Code, der wegen seiner einfachen logischen Abfolge ideal als Code für Digitalzähler geeignet ist, ändert sich beim Gray-Code beim Übergang von einer Zahl zur nächsten immer nur genau ein einziges Bit. Der Hauptvorteil liegt hier also in der Fehlererkennung bzw. Fehlervermeidung. Der Gray-Code wird z.B. für die Codierung bei absoluten Weg- bzw. Drehwinkelgebern verwendet, weil bei der Abtastung des Codelineals bzw. der Winkelcodescheibe auch bei nicht vollständig

exakter Lage der Abtasteinrichtung fehlerhafte Zwischenzustände ausgeschlossen werden können. Selbstverständlich lässt sich auch ein Zähler realisieren, der direkt im Gray-Code zählt (siehe Beispiele zur sequenziellen Logik), aber hier soll zunächst eine rein kombinatorische Schaltung zur Umsetzung zwischen Binär- und Gray-Code betrachtet werden. Für die Umsetzung vom Binär- zum Gray-Code gilt folgende Logiktablelle:

Dez.-wert	4-Bit-Binär-Code				Gray-Code			
	D	C	B	A	Q3	Q2	Q1	Q0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	0	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

Bei dieser Tabelle beginnt man mit der Aufstellung der Logikgleichungen am besten beim Ausgang Q3 und sieht sofort: $Q3 = D$.

Bei Q2 ist es schon ein wenig komplizierter: Q2 entspricht C, falls D = 0 ist, bzw. /C, falls D = 1 ist. Dies ergibt sich direkt aus einer Antivalenzverknüpfung von C und D: $Q2 = /D * C + D * /C$.

Für Q1 und Q0 sieht man sich am besten wieder die KV-Diagramme an:

Q1=	A	/A	
C	1	1	/D
/C	1	1	D
	/B	B	/B

Q0=	A	/A	
C	1	1	/D
/C	1	1	D
	/B	B	/B

Folglich gilt:

$$Q1 = /C * B + C * /B \text{ sowie } Q0 = /B * A + B * /A.$$

Beispiel-File: Bin_Gray_DEC.gds

Einstellungen auf der Platine: Schalter 5 bis 8: Eingänge für den 4-Bit-Binär-Code, Schalter 5 (links) hat die höchste Wertigkeit (2^3), Schalter 8 (rechts) hat die niedrigste Wertigkeit (2^0); Jumper für 7-Segment-Anzeige nicht gesetzt, Jumper für LED-Zeile gesetzt, Anzeige des Gray-Codes von LED0 (niedrigste Wertigkeit, rechts) bis D3 (höchste Wertigkeit, links)

Die Logiktable für die umgekehrte Umwandlungsrichtung vom Gray-Code zum Binär-Code sieht folgendermaßen aus:

Dez.-wert	Gray-Code				4-Bit-Binär-Code			
	D	C	B	A	Q3	Q2	Q1	Q0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

Auch hier ist wieder sofort ersichtlich:

$$Q3 = D .$$

Für Q2 gilt wie oben

$$Q2 = /D * C + D * /C .$$

Für die Ausgänge Q1 und Q0 ist die Sache etwas komplexer als im obigen Beispiel:

Q1=	A	/A		
C	1		1	/D
	1	1		D
/C	1		1	/D
	1	1		
	/B	B	/B	

Q0=	A	/A		
C		1	1	/D
	1		1	D
/C		1	1	/D
	1		1	D
	/B	B	/B	

Wie aus dem KV-Diagramm ersichtlich ist, lassen sich für die Ausgangsgleichung für Q1 noch jeweils zwei Felder zusammenfassen:

$$Q1 = /D * C * /B + D * C * B + D * /C * B + /D * /C * B .$$

Bei Q0 geht dies wegen der vollständig diagonalen Anordnung der Felder mit einer logischen 1 nicht mehr. Es ergibt sich hier keine weitere Minimierungsmöglichkeit; für Q0 werden hier insofern tatsächlich acht ODER-Terme benötigt:

$$Q0 = /D * C * B * A + /D * C * /B * /A + D * C * /B * A + D * C * B * /A + D * /C * B * A + D * /C * /B * /A + /D * /C * /B * A + /D * /C * B * /A .$$

Bei dieser Umsetzrichtung lässt sich nun allerdings eine erhebliche Vereinfachung der Logikgleichungen erreichen, indem man die Möglichkeit der Rückführung von Ausgängen in die kombinatorische Matrix nutzt. Bei Q3 bleibt alles gleich: $Q3 = D$. Q2 lässt sich auch folgendermaßen ausdrücken: $Q2 = \neg Q3 * C + Q3 * \neg C$. Hier handelt es sich eigentlich nur um eine formale Ersetzung aus Gründen der Einheitlichkeit, auf die Komplexität der Logikgleichung hat dies keinen Einfluss; bei Q1 und Q0 ist dies anders. Wie sich durch einfaches wenn auch etwas langwieriges Nachrechnen zeigen lässt, gilt: $Q1 = \neg Q2 * B + Q2 * \neg B$ und $Q0 = \neg Q1 * A + Q1 * \neg A$.

Beispiel-File: Gray_Bin_DEC.gds und Gray_Bin_DEC_RF.gds

Einstellungen auf der Platine: Schalter 5 bis 8: Eingänge für den 4-Bit-Gray-Code, Schalter 5 (links) hat die höchste Wertigkeit (2^3), Schalter 8 (rechts) hat die niedrigste Wertigkeit (2^0); Jumper für 7-Segment-Anzeige nicht gesetzt, Jumper für LED-Zeile gesetzt, Anzeige des Binär-Codes von LED0 (niedrigste Wertigkeit, rechts) bis D3 (höchste Wertigkeit, links)

2.5. Addition zweier 4-Bit-Binärzahlen:

Die Addition zweier Binärzahlen lässt sich vergleichsweise einfach in Form kombinatorischer Logik realisieren. Beginnt man zunächst mit der Addition von zwei nur einstelligen Zahlen, ergibt sich die Schaltung eines sog. Halbaddierers. Da der Zahlenumfang bei zwei nur einstelligen Summanden extrem eingeschränkt ist, ergeben sich nur vier mögliche Fälle: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$ und $1 + 1 = 2$. Um die maximal mögliche Summe von 2 darstellen zu können, werden zwei binäre Ausgänge benötigt, die üblicherweise als Summe (S) und Übertrag (U) bezeichnet werden. Mit der Bezeichnung A und B für die zwei einstelligen Summanden ergibt sich folgende kleine Logiktable:

A	B	U	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Man erhält also folgende Logikgleichungen für die zwei Ausgänge:

$S = \neg A * B + A * \neg B$ sowie

$U = A * B$.

Problem des Halbaddierers ist, dass er zwar über einen Übertragsausgang für die nächst höhere Additionsstelle verfügt, jedoch keinen Übertragseingang besitzt, mit dem ein Übertrag aus einer im Stellenwert darunter liegenden Additionsstufe berücksichtigt werden könnte. Der Halbaddierer ist folglich nur für Addition der niedrigstwertigen Stelle geeignet. Für die Addition in allen weiteren Stellen wird ein Volladdierer benötigt, der genau diesen Übertragseingang (C) aufweist. Hier werden also letztendlich drei jeweils einstellige Binärzahlen addiert, der Zahlenbereich der Summation erstreckt sich hierbei bis 3 und ist daher genau wie beim Halbaddierer

durch zwei binäre Ausgänge realisierbar. Es gelten somit folgende Logiktablelle bzw. KV-Diagramme:

A	B	C	U	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

U=	A		/A	
C	1	1	1	0
/C	0	1	0	0
	/B	B		/B

S=	A		/A	
C		1		1
/C	1		1	
	/B	B		/B

Damit lauten die Logikgleichungen:

$$U = A * B + A * C + B * C$$

$$S = /A * /B * C + /A * B * /C + A * /B * /C + A * B * C .$$

Die Addition zweier Binärzahlen mit mehr als einer Stelle lässt sich nun durch eine entsprechende Anzahl von Volladdierern realisieren. Zur Variablenbezeichnung kommt dabei nun der jeweilige Stellenwert als Ziffer hinzu, das gilt analog für die Summe, wobei der Übertrag der letzten Additionsstelle als fünftes Summenbit aufgefasst werden kann. Die Überträge der unteren Stellen werden quasi nur intern benötigt, erfordern aber natürlich zunächst eigene Ausgänge am GAL, die dann in die Verknüpfungsmatrix zurückgeführt werden. Für die Addition zweier 4-Bit-Binärzahlen erhält man somit folgende Logikgleichungen:

$$S_0 = /A_0 * B_0 + A_0 * /B_0 , \quad U_0 = A_0 * B_0 ,$$

$$S_1 = /A_1 * /B_1 * U_0 + /A_1 * B_1 * /U_0 + A_1 * /B_1 * /U_0 + A_1 * B_1 * U_0 ,$$

$$U_1 = A_1 * B_1 + A_1 * U_0 + B_1 * U_0$$

$$S_2 = /A_2 * /B_2 * U_1 + /A_2 * B_2 * /U_1 + A_2 * /B_2 * /U_1 + A_2 * B_2 * U_1 ,$$

$$U_2 = A_2 * B_2 + A_2 * U_1 + B_2 * U_1$$

$$S_3 = /A_3 * /B_3 * U_2 + /A_3 * B_3 * /U_2 + A_3 * /B_3 * /U_2 + A_3 * B_3 * U_2 ,$$

$$S_4 = A_3 * B_3 + A_3 * U_2 + B_3 * U_2 \quad (= U_3) .$$

Neben den zwingend erforderlichen fünf Ausgängen zur Darstellung der vollständigen Summe der zwei 4-Bit-Summanden inkl. einem Übertragsbit werden also noch drei zusätzliche Hilfsausgänge für die Verarbeitung der internen Überträge benötigt. Der Übertrag der untersten Stelle (U₀) lässt sich allerdings noch direkt in die Gleichungen für S₁ und U₁ einsetzen, so dass man hierdurch einen Hilfsausgang einsparen kann:

$$S_1 = /A_1 * /B_1 * A_0 * B_0 + /A_1 * B_1 * /A_0 + /A_1 * B_1 * /B_0 + A_1 * /B_1 * /A_0$$

$$+ A_1 * /B_1 * /B_0 + A_1 * B_1 * A_0 * B_0 ,$$

$$U1 = A1 * B1 + A1 * A0 * B0 + B1 * A0 * B0 .$$

Für die zwei weiteren internen Überträge (U1 und U2) liefert dieses Verfahren dann allerdings wegen der größeren Anzahl von UND-Verknüpfungen beim Einsetzen in die Invertierungen zu viele ODER-Terme, um das mittels eines einzelnen GAL-Ausgangs zu realisieren.

Beispiel-File: Addierer_4_Bit.gds und Addierer_4_Bit_U0.gds

Einstellungen auf der Platine: Schalter 1 bis 4: Eingänge für den Summanden A im 4-Bit-Binär-Code, Schalter 1 (links) hat die höchste Wertigkeit (2^3), Schalter 4 (rechts) hat die niedrigste Wertigkeit (2^0), Schalter 5 bis 8: Eingänge für den Summanden B im 4-Bit-Binär-Code, Schalter 5 (links) hat die höchste Wertigkeit (2^3), Schalter 8 (rechts) hat die niedrigste Wertigkeit (2^0); Jumper für 7-Segment-Anzeige nicht gesetzt, Jumper für LED-Zeile gesetzt, Anzeige der Summe inkl. Übertrag von LED0 (niedrigste Wertigkeit, rechts) bis D4 (höchste Wertigkeit, links); LED8 und LED9 zeigen die zwei internen Überträge U1 und U2 an.

Kapitel 3: Grundlegende Aspekte von Flip-Flops

Dieses Kapitel soll veranschaulichen, wie durch Rückkopplung von Ausgängen der Übergang von der rein kombinatorischen Logik („Schaltnetze“) zur sequenziellen Logik („Schaltwerke“) gelingt. Ganz bewusst wird in diesem Abschnitt noch auf die Verwendung des D-Flip-Flops, welches in der OLMC eines jeden GAL-Ausgangs verfügbar ist, verzichtet. Insofern liegt der Schwerpunkt dieses Kapitels auf der Didaktik in Bezug auf den grundlegenden Aufbau der verschiedenen Flip-Flop-Typen. Beispiele zum eigentlichen Einsatz der sequentiellen Logik, die dann tatsächlich auf der Nutzung der internen D-Flip-Flops des GALs beruhen, folgen im nächsten Kapitel. Erst dort wird dann der volle Umfang der Leistungsfähigkeit der GALs ersichtlich.

3.1 RS-Flip-Flop

Die einfachste Schaltung, bei der es durch die Rückkopplung der Ausgänge auf diese Ausgänge beeinflussende Eingänge zu einem Schaltverhalten kommt, welches von der zeitlichen Entwicklung der Signale abhängt, ist das statische RS-Flip-Flop. Man gelangt dadurch also nun in den Bereich der sequenziellen Logik, bei der das zeitliche Verhalten der Schaltung in den Vordergrund rückt.

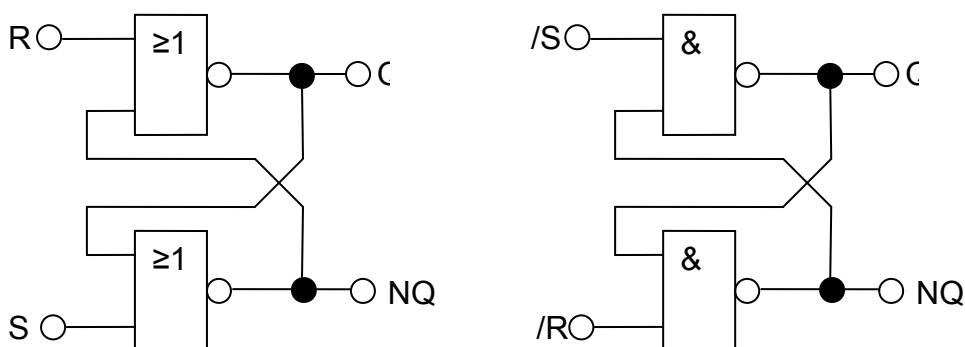
In der Literatur findet man üblicherweise zwei logisch grundsätzlich sehr ähnliche Realisierungen des statischen RS-Flip-Flops aufbauend entweder auf zwei NOR- oder zwei NAND-Gattern. Wie in der folgenden Abbildung dargestellt besitzt die Variante aus zwei NOR-Gattern jeweils high-aktive Rücksetz- bzw. Setzeingänge; der Setzeingang befindet sich an dem Gatter, das den invertierten Ausgang NQ zur Verfügung stellt. (Dieser Ausgang ist hier bewusst mit NQ und nicht etwa \bar{Q} benannt, um in den folgenden GAL-Gleichungen zwischen diesem zweiten Ausgang und der möglichen Ausgangsinvertierung des GALs für den Ausgang Q unterscheiden zu können.) Im Gegensatz zur NOR-Variante besitzt das RS-Flip-Flop aus NAND-Gattern low-aktive Rücksetz- und Setzeingänge; hier befindet sich der Setzeingang jedoch am Gatter für den Ausgang Q.

Beide Schaltungsvarianten lassen sich analog in GAL-Gleichungen übersetzen:

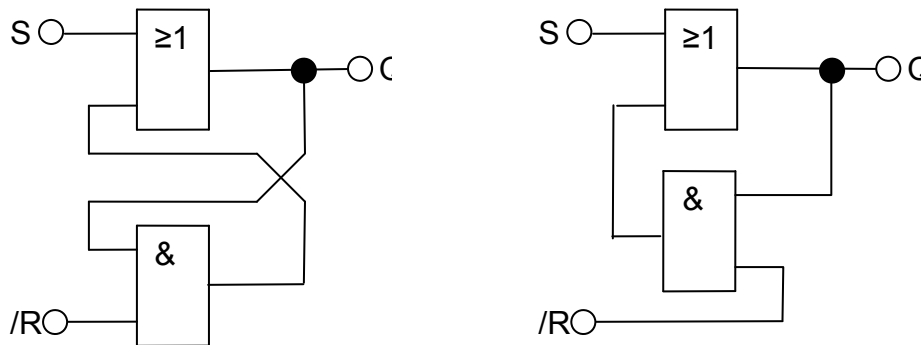
a) NOR-Variante: $\bar{Q} = R + NQ$ sowie $\bar{NQ} = S + Q$

b) NAND-Variante: $\bar{Q} = \bar{S} * NQ$ sowie $\bar{NQ} = \bar{R} * Q$

Der Nachteil beider Realisierungen besteht allerdings darin, dass für ein einziges RS-Flip-Flop schon zwei OLMCs des GALs benötigt werden.



Eine ganz interessante und wesentlich weniger bekannte Variante des RS-Flip-Flops ergibt sich, wenn man ausgehend von der NAND-Version auf das NAND-Gatter für den Ausgang Q das de-Morgansche-Gesetz anwendet. Es entfällt dann die Ausgangsinvertierung, die Funktion ändert sich zum ODER-Gatter, und abschließend müssen beide Eingänge invertiert werden. Man erhält daher folgende Schaltungsvariante mit gleichem logischen Verhalten:



Die beiden im obigen Bild dargestellten Schaltungen sind exakt identisch, nur tritt in der rechten Bildhälfte durch das gespiegelt dargestellte UND-Gatter die Selbsthalteschleife (in Analogie zur früheren Relais-technik) um das ODER-Gatter herum stärker hervor. Diese Selbsthalteschleife wird bei einer Null am /R-Eingang unterbrochen, was dann zum Rücksetzen des Flip-Flops führt.

Diese Realisierung besitzt nun den großen Vorteil, dass eine einzige OLMC des GALs genügt, um ein RS-Flip-Flop aufzubauen. Nebenbei entfällt dabei die meist logisch unerwünschte Möglichkeit, dass beide Ausgänge des Flip-Flops, die eigentlich immer logisch invertiert zueinander sein sollen, bei gleichzeitig erfüllter Setz- und Rücksetzbedingung den gleichen logischen Wert annehmen; die obige Schaltungsvariante besitzt insofern eine inhärente Priorität des Setzeinganges. Es sei hier ausdrücklich betont, dass es sich nicht etwa um ein RS-Flip-Flop handelt, das nur aus einem einzigen Logikgatter aufgebaut ist – das ist prinzipiell unmöglich –, sondern dass sich das zweite notwendige Gatter hier innerhalb der UND-Matrix des GALs befindet und daher für das Flip-Flop nur eine OLMC belegt wird.

Die Logikgleichung für diesen einen Ausgang lautet wie folgt:

$$Q = S + \bar{R} * Q .$$

Beispiel-File: RS_FlipFlop.gds

Einstellungen auf der Platine:

Schalter 1: Setzeingang (S),

Schalter 2: Rücksetzeingang (R)

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,

LED: D9: Ausgang Q des auf die Galstruktur optimierten RS-Flip-Flops,

D8: Ausgang Q des RS-Flip-Flops basierend auf NOR-Verknüpfungen,

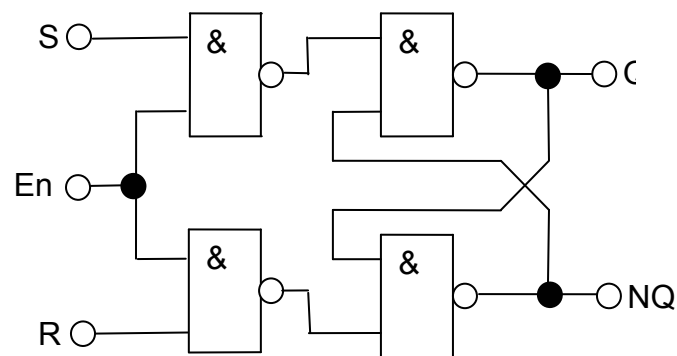
D7: Ausgang Q des RS-Flip-Flops basierend auf NAND-Verknüpfungen

(D0 und D1: invertierte Ausgänge für Standard-Realisierungen mit Nutzung von je zwei OLMCs)

(Einstellungen für den Takt nicht relevant)

3.2 Gegatetes RS-Flip-Flop

Das statische RS-Flip-Flop kann nun im nächsten Schritt zu einem getakteten - bzw. eigentlich besser ausgedrückt - zu einem gegateten RS-Flip-Flop erweitert werden, indem vor die Setz- und Rücksetzeingänge jeweils noch UND-Gatter geschaltet werden, deren zwei andere Eingänge zu einem gemeinsamen Freigabeeingang (Enable) verbunden werden. Dadurch ergibt sich zunächst nur eine eher geringfügig erscheinende Änderung des logischen Schaltverhaltens in der Form, dass nun bei diesem RS-Flip-Flop der Enable-Eingang logisch 1 sein muss, damit am Flip-Flop-Zustand durch den Setz- bzw. Rücksetzeingang überhaupt etwas verändert werden kann. Ist der Enable-Eingang logisch 0, so bleibt der im Flip-Flop gespeicherte Zustand unverändert, und zwar unabhängig davon, was an den anderen beiden Eingängen geschieht.



Obwohl im obigen Bild insgesamt vier NAND-Gatter eingesetzt sind, gibt es auch von diesem Flip-Flop eine Variante, die nur eine einzige OLMC des GALs nutzt. Die Logikgleichung in disjunktiver Normalform für diesen einen Ausgang Q lautet dann:

$$Q = S * En + Q * /R + Q * /En .$$

Diese Umwandlung erfolgt sehr ähnlich zu der ausführlich dargestellten Umwandlung des Standard-RS-Flip-Flops. Man versuche einmal selbst, aus dem obigen Schaltbild durch Umformung des NAND-Gatters für den Ausgang Q in ein ODER-Gatter nach de Morgan und anschließendes Umrechnen in die disjunktive Normalform die angegebene Logikgleichung zu erhalten.

Beispiel-File: RS_En_FlipFlop.gds

Einstellungen auf der Platine:

Schalter 1: Setzeingang,

Schalter 2: Rücksetzeingang,

Taster: Enable-Signal

Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,

LED: D9: Ausgang Q des auf die Galstruktur optimierten RS-Flip-Flops,

D8: Ausgang Q des RS-Flip-Flops basierend auf NOR-Verknüpfungen,

D7: Ausgang Q des RS-Flip-Flops basierend auf NAND-Verknüpfungen

(D0 und D1: invertierte Ausgänge fuer Standard-Realisierungen mit Nutzung von je zwei OLMCs)

(Einstellungen für den Takt nicht relevant)

3.3 D-Type-Latch und D-Master-Slave-Flip-Flop

Durch Verbindung der beiden Eingänge S und /R zu einem einzigen Dateneingang D gelangt man zum sog. D-Type-Latch; alternativ verwendete Bezeichnungen lauten taktpegel-gesteuertes D-Flip-Flop oder transparentes D-Flip-Flop. Dieses Latch verhält sich folgendermaßen: Solange der Enable-Eingang logisch 1 ist, ist der Ausgang Q gleich dem Logikpegel am Eingang D. (Das Latch verhält sich also quasi vom Eingang zum Ausgang transparent.) Fällt nun das Signal an Enable auf logisch 0, so wird der zu diesem Zeitpunkt aktuelle Logikpegel am D-Eingang im Latch gespeichert und steht am Ausgang Q solange zur Verfügung, bis Enable wieder auf 1 geht. Die Logikgleichung zur Realisierung eines D-Type-Latch im GAL lautet:

$$Q = D * En + Q * D + Q * /En .$$

Interessant wird das Verhalten nun, wenn man zwei dieser Latches hintereinander schaltet, wobei das Enable-Signal für das zweite Latch („Slave“) durch Invertierung aus dem Enable-Signal für das erste Latch („Master“) erzeugt wird. Man erhält dann ein Master-Slave-Flip-Flop (MSFF), dessen logisches Verhalten einem taktflanken-getriggerten Flip-Flop ähnelt. (Zur Erläuterung der geneuen Unterschiede siehe auch den Abschnitt zum JK_master-Slave-Flip-Flop). Erst mit diesem Flip-Flop-Typ sind dann z.B. Zähler und Schieberegister als typische Schaltungen der sequenziellen Logik realisierbar. Die Logikgleichungen für ein D-MSFF lauten:

$$QM = D * En + QM * D + QM * /En$$

$$QS = QM * /En + QS * QM + QS * En .$$

Beispiel-File: D_Latch.gds

Einstellungen auf der Platine:

Schalter 1: Dateneingang D,

Taster: Enable-Signal bzw. Clock-Signal

Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,

LED: D9: Ausgang Q des auf die Galstruktur optimierten D_Latches,

D8: Ausgang QS (Slave-Flip-Flop)

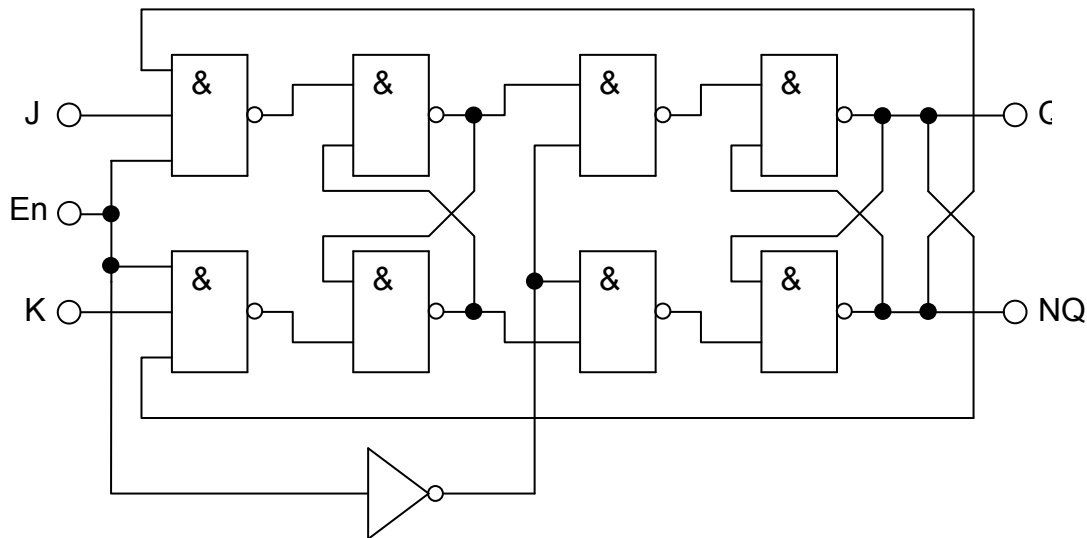
D7: Ausgang QM (Master-Flip-Flop)

(Einstellungen für den Takt nicht relevant)

3.4 JK-Master-Slave D-Flip-Flop

Bei allen Flip-Flop-Varianten, die nur einen einzelnen D-Eingang besitzen, wird auch ohne Integration einer gesonderten Prioritätsschaltung für den Setz- bzw. Rücksetzeingang der unterliegenden RS-Struktur verhindert, dass der unerwünschte Ausgangszustand $Q = /Q$ auftreten kann, da das Signal für den Rücksetzeingang durch die in das Flip-Flop integrierte Negation des Setzeinganges erzeugt wird. Diese Vorgehensweise schränkt allerdings den Funktionsumfang der D-Flip-Flop-Typen in gewisser Weise ein, so dass man auch ausgehend vom gegateteten RS-Latch auch ein RS-Master-Slave-Flip-Flop entwickeln kann. Um hierbei ebenfalls jegliche Probleme mit dem Eingangszustand $S = R = 1$ zu vermeiden, fügt man zusätzlich eine gekreuzte Rückkopplung der Slave-Ausgänge auf je einen zusätzlichen Eingang der UND-Gatter für die erste Gate-Stufe ein und gelangt so

zum JK-Master-Slave-Flip-Flop. (Wie beim D-MS-FF verhält sich das JK-MS-FF dann logisch ähnlich zum taktflankengesteuerten JK-Flip-Flop.)



Beispiel-File: JK_MS_FlipFlop.gds

Einstellungen auf der Platine:

Schalter 1: Eingang J,

Schalter 2: Eingang K,

Taster: Clk-Signal

Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,

LED: D9: Ausgang QM des Master-Flip-Flops (wechselt seinen Zustand bei Clk = 0),

D8: Ausgang QS des Slave-Flip-Flops (übernimmt den Masterzustand bei Clk = 1),

D0: Ausgang QJK des taktflanken-gesteuerten JK-Flip-Flops zum Vergleich,

Takt: intern (KEY)

3.5 3-Bit-Binär-Zähler

Nur um zu zeigen, dass aus rein kombinatorisch aufgebauten JK-Master-Slave-Flip-Flops Schaltungen realisiert werden können, deren Schaltverhalten mit Schaltwerken aus taktflankengesteuerten Flip-Flops vergleichbar ist, sei hier abschließend für dieses Kapitel ein Beispiel eines binären 3-Bit-Vorwärtszählers angeführt, der auf den im vorangehenden Abschnitt entwickelten JK-Master-Slave-Flip-Flops basiert.

Normalerweise wird man jedoch im Rahmen der sequenziellen Logik immer das innerhalb der OLMC verfügbare, taktflankengesteuerte D-Flip-Flop einsetzen. (Beispiele hierzu folgen im Kapitel 4.)

Beispiel-File: JK_MS_ZAEHLER.gds

Einstellungen auf der Platine:

Schalter 1: CEn: Freigabe des Zählers (CEn=1: Zähler zählt, CEn=0: Zähler gesperrt)

Taster: Clk-Signal

Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,

LED: D9: Ausgang Q2 des Zählers (Stellenwert 2^2 , höchstwertigstes Bit),

D8: Ausgang Q1 des Zählers (Stellenwert 2^1),

D7: Ausgang Q0 des Zählers (Stellenwert 2^0 , niedrigstwertigstes Bit)

D0...D2: Slave-Ausgänge (nicht relevant)

(Einstellungen für den Takt: intern (KEY))

Kapitel 4: Sequenzielle Logik

In diesem Kapitel soll gezeigt werden, wie sich durch Nutzung der internen D-Flip-Flops in den 10 OLMCs des GALs Schaltungen im Rahmen der sequenziellen Logik – also sog. Schaltwerke - realisieren lassen. Da durch die interne Verknüpfung aller Takteingänge der D-FlipFlops innerhalb eines GAL-Bausteins zu einer gemeinsamen Taktleitung alle Flip-Flops zwangsweise dasselbe Taktsignal erhalten, lassen sich prinzipbedingt nur synchron getaktete Schaltwerke realisieren. Diese Tatsache erweist sich aber eigentlich als sehr vorteilhaft, weil es bei asynchron getakteten Schaltwerken – hierzu zählen z.B. sog. „ripple counter“ – durch die Addition von Gatterlaufzeiten zum Auftreten unerwünschter Zwischenzustände bei Umschaltprozessen kommen kann; dies wird also bei synchron getakteten Schaltwerken prinzipiell vermieden.

4.1. 4-Bit-Binär-Zähler

Ein 4-Bit-Zähler, der im Binärcode vorwärts zählt, ist durch folgende logische Zustandstabelle gekennzeichnet:

Dez.-wert	4-Bit-Binär-Code			
	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Um aus dieser Tabelle die logischen Gleichungen für die vier Variablen Q0 bis Q3 ablesen zu können, geht man wie folgt vor: Da es im Gegensatz zur rein kombinatorischen Logik keine getrennten reinen Ein- bzw. Ausgänge gibt, stellt quasi der logische Ausgangszustand in einer Zeile gleichzeitig den logischen Eingangszustand der unmittelbar folgenden Zeile, die sich nach Auftreten des nächsten Taktsignals sozusagen automatisch aus der Vorgängerzeile ergeben soll, dar.

Beginnen wir zunächst mit Q0: Offensichtlich muss Q0 seinen logischen Ausgangszustand bei jedem Taktsignal ändern, und zwar unabhängig vom logischen Zustand der übrigen drei Variablen Q1 bis Q3. Insofern lautet die Logikgleichung für Q0:

$$Q0 := /Q0 .$$

Der Doppelpunkt vor dem Gleichheitszeichen stellt dabei innerhalb des GALs die Nutzung des D-Flip-Flops zur Generierung des Ausgangssignals sicher.

Bei Q1 sieht es schon etwas komplexer aus: Q1 muss auf 1 gesetzt werden, falls zuvor Q0 gesetzt, Q1 selbst aber rückgesetzt ist – z.B. beim Wechsel von Zustand 1 nach Zustand 2 – oder aber auch, wenn Q1 gesetzt, aber Q0 rückgesetzt ist – z.B. beim folgenden Wechsel von Zustand 2 nach 3; dies entspricht also einer Antivalenzverknüpfung von Q1 und Q0. Dieser Vorgang wiederholt sich entsprechend bei den anderen Zuständen, und zwar unabhängig von den Variablen Q2 und Q3. Daher lautet die Gleichung für Q1:

$$Q1 := Q0 * /Q1 + /Q0 * Q1 .$$

Entsprechend verhält es sich auch bei Q2, wobei nun neben Q2 selbst auch der logische Zustand der zwei „Vorgänger“-Bits Q1 und Q0 relevant ist:

$$Q2 := /Q2 * Q1 * Q0 + Q2 * /Q1 * /Q0 + Q2 * /Q1 * Q0 + Q2 * Q1 * /Q0 .$$

Auch ohne Einsatz eines KV-Diagramms lässt sich hier eine Vereinfachungsmöglichkeit bei den zwei mittleren Termen erkennen, da dort einmal eine UND-Verknüpfung mit /Q0 sowie mit Q0 auftritt. Vereinfacht erhält man also folgende Gleichung:

$$Q2 := /Q2 * Q1 * Q0 + Q2 * /Q1 + Q2 * Q1 * /Q0 .$$

Schließlich erhält man für Q3 folgende letzte Logikgleichung:

$$\begin{aligned} Q3 := & /Q3 * Q2 * Q1 * Q0 + Q3 * /Q2 * /Q1 * /Q0 + Q3 * /Q2 * /Q1 * Q0 \\ & + /Q3 * /Q2 * Q1 * /Q0 + /Q3 * /Q2 * Q1 * Q0 + /Q3 * Q2 * /Q1 * /Q0 \\ & + /Q3 * Q2 * /Q1 * Q0 + /Q3 * Q2 * Q1 + /Q0 . \end{aligned}$$

Man erkennt an der Vielzahl der auftretenden ODER-Terme, dass eine optimale Vereinfachung sinnvoll sein kann, um die Gleichung überhaupt innerhalb einer OLMC des GALs unterbringen zu können. Auch hierzu ist der Einsatz eines KV-Diagramms wieder hilfreich. Im folgenden KV-Diagramm sind die acht logischen 1en der obigen Zustandstabelle für Q3 entsprechend eingetragen.

Q3=	Q0	/Q0		
Q2		1		/Q3
	1		1	1
/Q2	1	1	1	1
	/Q1	Q1	/Q1	

Man liest für Q3 folgende vereinfachte Gleichung ab:

$$\begin{aligned} Q3 := & /Q3 * Q2 * Q1 * Q0 \\ & + Q3 * /Q2 + Q3 * /Q1 + Q3 * /Q0 \end{aligned}$$

Damit lautet der komplette Gleichungssatz für einen 4-Bit-Binär-Vorwärtzähler:

$$\begin{aligned} Q0 & := /Q0 \\ Q1 & := /Q1 * Q0 + Q1 * /Q0 \\ Q2 & := /Q2 * Q1 * Q0 + Q2 * /Q1 + Q2 * Q1 * /Q0 \\ Q3 & := /Q3 * Q2 * Q1 * Q0 + Q3 * /Q2 + Q3 * /Q1 + Q3 * /Q0 . \end{aligned}$$

Durchläuft man die obige Logiktablelle für die Zustände des 4-Bit-Zählers von unten, also beginnend bei 15, nach oben bis zur 0, so erhält man einen binären Rückwärtzähler. Dessen Logikgleichungen lauten wie folgt:

$$\begin{aligned} Q0 & := /Q0 \\ Q1 & := /Q1 * /Q0 + Q1 * Q0 \\ Q2 & := /Q2 * /Q1 * /Q0 + Q2 * Q1 + Q2 * Q0 \\ Q3 & := /Q3 * /Q2 * /Q1 * /Q0 + Q3 * Q2 + Q3 * Q1 + Q3 * Q0 . \end{aligned}$$

Durch Einführung einer zusätzlichen Eingangsvariablen „UP“ lassen sich nun beide Gleichungssätze auch zu einem 4-Bit-Zähler kombinieren, dessen Zählrichtung vom

Zustand dieser Variablen abhängt, so dass der Zähler vorwärts zählt, solange UP logisch 1 ist und rückwärts zählt, falls UP logisch 0 ist.

Weiterhin lässt sich durch eine Variable „Load“ auch eine taktabhängige Setzmöglichkeit des Zählers auf einen vorab eingestellten Binärwert realisieren (siehe Beispielfile).

**Beispiel-File: Bin_4Bit_Zaehler_VR.gds und BCD_4Bit_Zaehler_VR_CE_L.gds
Einstellungen auf der Platine:**

Schalter 1: Zählrichtung: UP=1: vorwärts, UP=0: rückwärts,
Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,
Taktestellung: zunächst Taster (KEY), später evtl. auch automatischer Takt (GEN),
Taktfrequenz nach Belieben.

4.2. 4-Bit-BCD-Zähler

Ein 4-Bit-Zähler, der binär codiert von 0 bis 9 vorwärts zählt, ist durch folgende logische Zustandstabelle gekennzeichnet:

Dez.-wert	4-Bit-Binär-Code			
	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Für Q0 liest man eine zum Binärzähler identische Gleichung ab:

$$Q0 := /Q0 .$$

Aber schon bei der Gleichung für Q1 muss man nun berücksichtigen, dass aus dem Zustand 9 der Übergang zur 0 erfolgen muss, d.h., Q1 darf bei diesem Übergang nicht gesetzt werden. Insofern spielt nun im Gegensatz zum Binärzähler auch der logische Zustand des höchstwertigsten Bits Q3 für die Zustandsänderung von Q1 eine Rolle.

Es ergibt sich folgende Gleichung für Q1:

$$Q1 := /Q1 * Q0 * /Q3 + Q1 * /Q0 * /Q3 .$$

Q2 ist dagegen wieder identisch zum Binärzähler, da auf den Zustand 9 ohnehin wieder Q2 = 0 folgt, was auch für den Zustand 0 richtig ist.

Bei Q3 werden nun hier nur noch zwei 1en benötigt, so dass diese Gleichung sehr einfach wird:

$$Q3 := /Q3 * Q2 * Q1 * Q0 + Q3 * /Q2 * /Q1 * /Q0 .$$

Damit lautet dann der vollständige Satz für den 4-Bit-BCD-Vorwärtszähler:

$$\begin{aligned} Q0 &:= /Q0 \\ Q1 &:= /Q1 * Q0 * /Q3 + Q1 * /Q0 * /Q3 \\ Q2 &:= /Q2 * Q1 * Q0 + Q2 * /Q1 + Q2 * Q1 * /Q0 \\ Q3 &:= /Q3 * Q2 * Q1 * Q0 + Q3 * /Q2 * /Q1 * /Q0 . \end{aligned}$$

Dadurch, dass in der obigen Zustandstabelle für den BCD-Zähler nicht alle prinzipiell möglichen Zustände für die vier binären Variablen Q0 bis Q3 enthalten sind, tritt ein zusätzliches Problem auf: In den obigen Gleichungssatz ist nicht explizit eingearbeitet, was passieren soll, wenn sich das Schaltwerk aus irgendeinem Grund nicht in einem der 10 gültigen Zustände befindet. Dies könnte z.B. nach dem Einschalten der Betriebsspannung auftreten oder u.U. auch während des Betriebs durch externe Störimpulse verursacht werden.

Beispiel-File: BCD_4Bit_Zaehler_VR.gds

Einstellungen auf der Platine:

Schalter 1: Zähler beim nächsten Taktimpuls auf Anfangswert setzen, falls logisch 1,
 Schalter 5 bis 8: Eingabe der 4-Bit-Binär-Zahl als Anfangswert
 (Stellenwert: Sch.5: 2^3 (links), Sch.6: 2^2 , Sch.7: 2^1 , Sch.8: 2^0 (rechts)),
 Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,
 Takteinstellung: zunächst Taster (KEY), später evtl. auch automatischer Takt (GEN),
 Taktfrequenz nach eigener Wahl.

4.3. 4-Bit-„Stunden“-Zähler

Ein 4-Bit-Zähler, der binär codiert von 1 bis 12 vorwärts zählt, ist durch nachstehende logische Zustandstabelle charakterisiert. Er könnte z.B. als Zähler in einer Uhr Verwendung finden, um die Stundenanzeige von 1 bis 12 zu realisieren.

Dez.-wert	4-Bit-Binär-Code			
	Q3	Q2	Q1	Q0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

Folgende Logikgleichungen liest man ab:

$$Q0 := /Q0$$

$$Q1 := /Q1 * Q0 + Q1 * /Q0$$

$$Q2 := /Q2 * Q1 * Q0 + Q2 * /Q1 * /Q3 + Q2 * /Q0 * /Q3$$

$$Q3 := /Q3 * Q2 * Q1 * Q0 + Q3 * /Q2 .$$

Beispiel-File: Stunden_4Bit_Zaehler.gds

Einstellungen auf der Platine:

Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,
 Takteinstellung: zunächst Taster (KEY), später evtl. auch automatischer Takt (GEN),
 Taktfrequenz nach eigener Wahl.

4.4. 4-Bit-Gray-Code-Zähler

Zum Abschluss der Zählerbeispiele soll noch ein 4-Bit-Gray-Code-Zähler vorgestellt werden. Der Gray-Code bietet bzgl. der Störsicherheit den Vorteil, dass bei jedem Übergang von einem Zählerzustand zum richtigen Folgezustand jeweils nur ein einziges Bit seinen Zustand ändert. Detektiert man also z.B. mit einer getrennten Überwachungsschaltung eine Zustandsänderung, bei der simultan zwei oder mehr Bits verändert werden, kann man sicher sein, dass eine nicht erlaubte Zustandsänderung innerhalb des Zählers – verursacht z.B. durch Störungen der Betriebsspannung – aufgetreten ist.

Dez.-wert	Gray-Code			
	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	0	1	0
4	0	1	1	0
5	0	1	1	1
6	0	1	0	1
7	0	1	0	0
8	1	1	0	0
9	1	1	0	1
10	1	1	1	1
11	1	1	1	0
12	1	0	1	0
13	1	0	1	1
14	1	0	0	1
15	1	0	0	0

Aus der Tabelle sind für die Vorwärtszählrichtung von 0 nach 15 folgende Logikgleichungen ablesbar:

$$Q0 := /Q3 * /Q2 * /Q1 + /Q3 * Q2 * Q1 + Q3 * Q2 * /Q1 + Q3 * /Q2 * Q1$$

$$Q1 := /Q3 * /Q2 * Q0 + /Q3 * Q1 * /Q0 + Q3 * Q2 * Q0 + Q3 * Q1 * /Q0$$

$$Q2 := /Q3 * Q1 * /Q0 + /Q3 * Q2 * Q0 + Q2 * /Q1 * /Q0 + Q3 * Q2 * Q0$$

$$Q3 := Q2 * /Q1 * /Q0 + Q3 * Q2 * Q0 + Q3 * Q1 * /Q0 + Q3 * /Q2 * Q0$$

Entsprechend lässt sich natürlich auch die Rückwärtszählrichtung von 15 nach 0 auswerten (siehe Beispielfile).

Beispiel-File: Gray_4Bit_Zaehler.gds

Einstellungen auf der Platine:

Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt, Takteinstellung: zunächst Taster (KEY), später evtl. auch automatischer Takt (GEN), Taktfrequenz nach eigener Wahl.

4.5. Schieberegister

Bei einem sog. Schieberegister soll der im Register gespeicherte Zustand bei jedem Taktimpuls um ein Bit nach rechts bzw. links verschoben werden. Solche Schieberegister werden u.a. bei der seriellen Übertragung von ursprünglich parallel codierter Information als Umsetzer verwendet. So wird z.B. bei der Datenübertragung über die serielle Schnittstelle eines PCs das zu übertragene Byte zunächst parallel in ein 8-Bit-Schieberegister geladen und dann Bit für Bit über nur eine Datenleitung ausgegeben. Auf der Empfängerseite arbeitet ein ähnliches Schieberegister, welches die Daten nach erfolgter bitweiser Übertragung wieder parallel codiert verfügbar macht.

Als Beispiel für ein komplexes Schieberegister soll hier ein 10-Bit-Register mit umkehrbarer Schieberichtung und Ringspeichermodus entwickelt werden. Ring-

speichermodus bedeutet dabei, dass das Bit, welches am einen Ende des Registers hinausgeschoben wird, am anderen Ende wieder eingefügt werden kann. Bezeichnet man die 10 Ausgänge des Registers mit Q0 bis Q9 und führt eine Variable „DataInput“ für die serielle Dateneingabe ein, so lauten die Logikgleichungen für das einfache Schieberegister zunächst:

$$\begin{aligned} Q0 &:= \text{DataInput} \\ Q1 &:= Q0 \\ Q2 &:= Q1 \\ &\dots \\ Q9 &:= Q8 . \end{aligned}$$

Soll zusätzlich die Schieberichtung durch die Variable SD gesteuert werden können, so wird daraus folgender Satz von Gleichungen:

$$\begin{aligned} Q0 &:= /SD * Q1 + SD * \text{DataInput} \\ Q1 &:= /SD * Q2 + SD * Q0 \\ Q2 &:= /SD * Q3 + SD * Q1 \\ &\dots \\ Q9 &:= /SD * \text{DataInput} + SD * Q8 . \end{aligned}$$

Schließlich lässt sich der Ringspeichermodus durch eine weitere Eingangsvariable RM steuern:

$$\begin{aligned} Q0 &:= /SD * Q1 + SD * RM * Q9 + SD * /RM * \text{DataInput} \\ Q1 &:= /SD * Q2 + SD * Q0 \\ Q2 &:= /SD * Q3 + SD * Q1 \\ &\dots \\ Q9 &:= /SD * /RM * \text{DataInput} + /SD * RM * Q0 + SD * Q8 . \end{aligned}$$

Beispiel-File: Schieberegister.gds

Einstellungen auf der Platine:

Schalter 1: serieller Dateneingang,
 Schalter 2: Schieberichtung,
 Schalter 3: Ringspeichermodus,
 Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,
 Takteinstellung: zunächst Taster (KEY), später evtl. auch automatischer Takt (GEN),
 Taktfrequenz nach eigener Wahl.

4.6. Binärer Pseudo-Zufallsbit-Generator

Ein Schieberegister lässt sich auch als binärer Pseudozufallsgenerator einsetzen, wenn im Ringspeichermodus das letzte Bit vor der Rückführung auf den Dateneingang des ersten Flip-Flops mit einem der dazwischen liegenden Bits über ein Exklusiv-ODER-Gatter verknüpft wird.

Als Beispiel hierfür wird ein 5-Bit-Schieberegister gezeigt, bei dem durch zwei Variablen ausgewählt werden kann, welcher der ersten vier Flip-Flop-Ausgänge zur EXOR-Bildung Verwendung finden soll. Außerdem kann das Register über den DIP-Schalter parallel auf einen Anfangswert geladen werden. Dadurch ist es möglich, die jeweilige Zyklenlänge der Pseudo-Zufalls-Zustände experimentell zu ermitteln.

Beispiel-File: Zufall_Bit_Gen.gds

Einstellungen auf der Platine:

Schalter 1 und 2: variable Rückkopplung von Q0 bis Q3 für EXOR-Verknüpfung mit Q4,

Schalter 3: paralleles Laden des Registers,

Schalter 4 bis 8: Dateneingabe zum parallelen Laden eines Registerzustands.

Jumper für LED-Zeile gesetzt, Jumper für 7-Segmentanzeige nicht gesetzt,

Takteinstellung: zunächst Taster (KEY), später evtl. auch automatischer Takt (GEN),

Taktfrequenz nach eigener Wahl.

Anhang: Quellcode der Beispieldateien

Beispiel-File: Boolesche_Funktionen.gds

Schalter 1: Eingangsvariable A, Schalter 2: Eingangsvariable B,
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt
(Einstellungen fuer den Takt nicht relevant)

CHIP Boolesche_Funktionen ispGAL22V10

```
SCLK CLK A      B      NC      NC      NC      MODE NC  NC
NC  NC  NC  GND
SDI  NC  NSUB_BA SUB_BA NSUB_AB SUB_AB EXNOR SDO  EXOR
NOR  OR  NAND  AND  VCC
```

// Konjunktion (UND-Verknüpfung):

AND = A * B;

/NAND = A * B;

// Disjunktion (ODER-Verknüpfung):

OR = A + B;

/NOR = A + B;

// Bijunktion (Äquivalenz-Verknüpfung):

EXOR = /A * B + A * /B;

/EXNOR = /A * B + A * /B;

// Subjunktion (Implikations-Verknüpfung):

SUB_AB = /A + A * B;

/NSUB_AB = /A + A * B;

SUB_BA = /B + A * B;

/NSUB_BA = /B + A * B;

Beispiel-File: BCD_7Segment_DEC.gds

Schalter 5 bis 8: Eingabe der 4-Bit-BCD-Zahl (gueltige Zahlen: 0 bis 9!)
(Stellenwert: Sch.5: 2^3 (links), Sch.6: 2^2 , Sch.7: 2^1 , Sch.8: 2^0 (rechts)),
Jumper fuer 7-Segmentanzeige gesetzt, Jumper fuer LED-Zeile optional
(Einstellungen fuer den Takt nicht relevant)

CHIP BCD_7SEGM_DEC ispGAL22V10

SCLK CLK NC NC NC NC D MODE C B A NC NC GND
SDI NC Qa Qb Qc Qd Qe SDO Qf Qg Q7 Q8 Q9 VCC

$$/Qa = /D * /C * /B * A + C * /B * /A;$$

$$/Qb = C * /B * A + C * B * /A;$$

$$/Qc = /C * B * /A;$$

$$/Qd = /D * /C * /B * A + C * /B * /A + C * B * A;$$

$$Qe = /D * /C * /A + C * B * /A + D * /A;$$

$$Qf = /C * /B * /A + D + C * /B + C * B * /A;$$

$$/Qg = /D * /C * /B + C * B * A;$$

/* ungenutzte Ausgaenge ausgeschaltet: */

Q7 = GND;

Q8 = GND;

Q9 = GND;

Beispiel-File: HEX_7Segment_DEC.gds

Schalter 5 bis 8: Eingabe der 4-Bit-Binaer-Zahl
(Stellenwert: Sch.5: 2^3 (links), Sch.6: 2^2 , Sch.7: 2^1 , Sch.8: 2^0 (rechts)),
Jumper fuer 7-Segmentanzeige gesetzt, Jumper fuer LED-Zeile optional
(Einstellungen fuer den Takt nicht relevant)

CHIP HEX_7SEGM_DEC ispGAL22V10

SCLK CLK NC NC NC NC D MODE C B A NC NC GND
SDI NC Qa Qb Qc Qd Qe SDO Qf Qg Q7 Q8 Q9 VCC

$$/Qa = /D * /C * /B * A + /D * C * /B * /A + D * /C * B * A + D * C * /B * A;$$

$$/Qb = /D * C * /B * A + /D * C * B * /A + D * /C * B * A + D * C * /B * /A + D * C * B;$$

$$/Qc = /D * /C * B * /A + D * C * /B * /A + D * C * B;$$

$$/Qd = /D * /C * /B * A + /D * C * /B * /A + C * B * A + D * /C * B * /A;$$

$$/Qe = /D * /C * /B * A + /D * /C * B * A + /D * C * /B + /D * C * B * A + D * /C * /B * A;$$

$$/Qf = /D * /C * /B * A + /D * /C * B + /D * C * B * A + D * C * /B * A;$$

$$/Qg = /D * /C * /B + /D * C * B * A + D * C * /B * /A;$$

/* ungenutzte Ausgaenge ausgeschaltet */

Q7 = GND;

Q8 = GND;

Q9 = GND;

Beispiel-File: Bin_Gray_DEC.gds

Schalter 5 bis 8: Eingabe der 4-Bit-Binaer-Zahl
(Stellenwert: Sch.5: 2^3 (links), Sch.6: 2^2 , Sch.7: 2^1 , Sch.8: 2^0 (rechts)),
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
LED: Gray-Code D3: hoch (links), ..., D0: niedrig (rechts)
(Einstellungen fuer den Takt nicht relevant)

CHIP BIN_GRAY_DEC ispGAL22V10

SCLK CLK NC NC NC NC D MODE C B A NC NC GND
SDI NC Q0 Q1 Q2 Q3 Q4 SDO Q5 Q6 Q7 Q8 Q9 VCC

$Q3 = D ;$

$Q2 = /D * C + D * /C ;$

$Q1 = /C * B + C * /B ;$

$Q0 = /B * A + B * /A ;$

/ ungenutzte Ausgaenge ausgeschaltet */*

$Q4 = GND;$

$Q5 = GND;$

$Q6 = GND;$

$Q7 = GND;$

$Q8 = GND;$

$Q9 = GND;$

Beispiel-File: Gray_Bin_DEC.gds und Gray_Bin_DEC_RF.gds

Schalter 5 bis 8: Eingabe der 4-Bit-Binaer-Zahl
(Stellenwert: Sch.5: 2^3 (links), Sch.6: 2^2 , Sch.7: 2^1 , Sch.8: 2^0 (rechts)),
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
LED: Gray-Code D3: hoch (links), ..., D0: niedrig (rechts)
(Einstellungen fuer den Takt nicht relevant)
CHIP BIN_GRAY_DEC ispGAL22V10

```
SCLK CLK NC NC NC NC D  MODE C  B  A  NC NC GND
SDI  NC  Q0 Q1 Q2  Q3 Q4  SDO Q5 Q6 Q7 Q8 Q9  VCC
```

```
Q3 = D ;
Q2 = /D * C + D * /C ;
Q1 = /C * B + C * /B ;
Q0 = /B * A + B * /A ;
/* ungenutzte Ausgaenge ausgeschaltet */
Q4 = GND;
Q5 = GND;
Q6 = GND;
Q7 = GND;
Q8 = GND;
Q9 = GND;
```

Schalter 5 bis 8: Eingabe der 4-Bit-Zahl im Gray-Code
(Stellenwert: Sch.5: hoch (links), ..., Sch.8: niedrig (rechts)),
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
LED: Binaer-Code D3: 2^3 , D2: 2^2 , D1 : 2^1 , D0: 2^0
(Einstellungen fuer den Takt nicht relevant)

Dieses Beispiel nutzt die Rueckfuehrung von Ausgaengen in die Verknuepfungsmatrix;
die Funktion ist identisch zu GRAY_BIN_DEC, aber die Logikgleichungen sind einfacher.
CHIP GRAY_BIN_DEC_RF ispGAL22V10

```
SCLK CLK NC NC NC NC D  MODE C  B  A  NC NC GND
SDI  NC  Q0 Q1 Q2  Q3 Q4  SDO Q5 Q6 Q7 Q8 Q9  VCC
```

```
Q3 = D ;
Q2 = /Q3 * C + Q3 * /C ;
Q1 = /Q2 * B + Q2 * /B ;
Q0 = /Q1 * A + Q1 * /A ;
/* ungenutzte Ausgaenge ausgeschaltet */
Q4 = GND;
Q5 = GND;
Q6 = GND;
Q7 = GND;
Q8 = GND;
Q9 = GND;
```

Beispiel-File: Addierer_4_Bit.gds und Addierer_4_Bit_U0.gds

Schalter 1 bis 4: Eingabe Summand A, 4-Bit-Binaer-Code,
Schalter 5 bis 8: Eingabe Summand B, 4-Bit-Binaer-Code
(Stellenwert: Sch.1+5: hoch (links), ..., Sch.4+8: niedrig (rechts)),
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
LED: Binaer-Code der Summe mit Uebertrag: D4: 2^4 , D3: 2^3 , D2: 2^2 , D1 : 2^1 ,
D0: 2^0 ,
D9: U2, D8: U1, D7: U0 (interne Uebertraege)
(Einstellungen fuer den Takt nicht relevant)

CHIP GRAY_BIN_DEC_RF ispGAL22V10

SCLK CLK A3 A2 A1 A0 B3 MODE B2 B1 B0 NC NC GND
SDI NC S0 S1 S2 S3 U3 SDO Q5 Q6 U0 U1 U2 VCC

$S0 = /A0 * B0 + A0 * /B0 ;$
 $U0 = A0 * B0 ;$

$S1 = /A1 * /B1 * U0 + /A1 * B1 * /U0 + A1 * /B1 * /U0 + A1 * B1 * U0 ;$
 $U1 = A1 * B1 + A1 * U0 + B1 * U0 ;$

$S2 = /A2 * /B2 * U1 + /A2 * B2 * /U1 + A2 * /B2 * /U1 + A2 * B2 * U1 ;$
 $U2 = A2 * B2 + A2 * U1 + B2 * U1 ;$

$S3 = /A3 * /B3 * U2 + /A3 * B3 * /U2 + A3 * /B3 * /U2 + A3 * B3 * U2 ;$
 $U3 = A3 * B3 + A3 * U2 + B3 * U2 ;$

/* ungenutzte Ausgaenge ausgeschaltet */

Q5 = GND;

Q6 = GND;

Schalter 1 bis 4: Eingabe Summand A, 4-Bit-Binaer-Code,
 Schalter 5 bis 8: Eingabe Summand B, 4-Bit-Binaer-Code
 (Stellenwert: Sch.1+5: hoch (links), ..., Sch.4+8: niedrig (rechts)),
 Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
 LED: Binaer-Code der Summe mit Uebertrag: D4: 2⁴, D3: 2³, D2: 2², D1 : 2¹,
 D0: 2⁰,
 D9: U2, D8: U1 (interne Uebertraege)
 (Einstellungen fuer den Takt nicht relevant)

CHIP GRAY_BIN_DEC_RF ispGAL22V10

SCLK CLK A3 A2 A1 A0 B3 MODE B2 B1 B0 NC NC GND
 SDI NC S0 S1 S2 S3 U3 SDO Q5 Q6 Q7 U1 U2 VCC

$$S0 = /A0 * B0 + A0 * /B0 ;$$

$$S1 = /A1 * /B1 * A0 * B0 + /A1 * B1 * /A0 + /A1 * B1 * /B0 + A1 * /B1 * /A0 \\ + A1 * /B1 * /B0 + A1 * B1 * A0 * B0 ;$$

$$U1 = A1 * B1 + A1 * A0 * B0 + B1 * A0 * B0 ;$$

$$S2 = /A2 * /B2 * U1 + /A2 * B2 * /U1 + A2 * /B2 * /U1 + A2 * B2 * U1 ;$$

$$U2 = A2 * B2 + A2 * U1 + B2 * U1 ;$$

$$S3 = /A3 * /B3 * U2 + /A3 * B3 * /U2 + A3 * /B3 * /U2 + A3 * B3 * U2 ;$$

$$U3 = A3 * B3 + A3 * U2 + B3 * U2 ;$$

/* ungenutzte Ausgaenge ausgeschaltet */

Q5 = GND;

Q6 = GND;

Q7 = GND;

Beispiel-File: RS_FlipFlop.gds

Verschiedene Realisierungen eines statischen RS-Flip-Flops:

Schalter 1: Setzeingang,
Schalter 2: Ruecksetzeingang

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
LED: D9: Ausgang Q des auf die Galstruktur optimierten RS-Flip-Flops,
D8: Ausgang Q des RS-Flip-Flops basierend auf NOR-Verknuepfungen,
D7: Ausgang Q des RS-Flip-Flops basierend auf NAND-Verknuepfungen
(D0 und D1: invertierte Ausgaenge fuer Standard-Realisierungen mit Nutzung
von je zwei OLMCs)
(Einstellungen fuer den Takt nicht relevant)

Beobachtung:

Alle Flip-Flops verhalten sich logisch gleichwertig, solange nicht
die unerwuenschte Eingangskombination $R = S = 1$ anliegt.

CHIP RS_FlipFlop ispGAL22V10

```
SCLK CLK S          R          NC NC NC MODE NC NC NC          NC
NC GND
SDI  NC  NQ_NAND NQ_NOR Q2 Q3 Q4 SDO  Q5 Q6  Q_NAND Q_NOR
Q   VCC
```

/ statisches RS-Flip-Flop basierend auf der NOR-Verknuepfung */*

/Q_NOR = R + NQ_NOR ;

/NQ_NOR = S + Q_NOR ;

/ statisches RS-Flip-Flop basierend auf der NAND-Verknuepfung */*

*/Q_NAND = /S * NQ_NAND ;*

*/NQ_NAND = /R * Q_NAND ;*

/ statisches RS-Flip-Flop optimiert fuer disjunktive Normalform */*

*Q = Q * /R + S ;*

/ nicht genutzte Ausgaenge ausgeschaltet */*

Q2 = GND;

Q3 = GND;

Q4 = GND;

Q5 = GND;

Q6 = GND;

Beispiel-File: RS_En_FlipFlop.gds

Verschiedene Realisierungen eines gegateteten RS-Flip-Flops:

Schalter 1: Setzeingang,

Schalter 2: Ruecksetzeingang,

Taster: Enable-Signal

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,

LED: D9: Ausgang Q des auf die Galstruktur optimierten RS-Flip-Flops,

D8: Ausgang Q des RS-Flip-Flops basierend auf NOR-Verknuepfungen,

D7: Ausgang Q des RS-Flip-Flops basierend auf NAND-Verknuepfungen

(D0 und D1: invertierte Ausgaenge fuer Standard-Realisierungen mit Nutzung von je zwei OLMCs)

(Einstellungen fuer den Takt nicht relevant)

Beobachtung:

Alle Flip-Flops verhalten sich logisch gleichwertig, solange nicht die unerwuenschte Eingangskombination $R = S = 1$ anliegt.

CHIP RS_En_FlipFlop ispGAL22V10

```
SCLK En S          R          NC NC NC MODE NC NC NC          NC
NC GND
SDI   NC NQ_NAND NQ_NOR Q2 Q3 Q4 SDO   Q5 Q6 Q_NAND Q_NOR
Q    VCC
```

/ statisches RS-Flip-Flop mit zusaetzlichem Enable-Eingang basierend auf der NOR-Verknuepfung */*

*/Q_NOR = R * En + NQ_NOR ;*

*/NQ_NOR = S * En + Q_NOR ;*

/ statisches RS-Flip-Flop mit zusaetzlichem Enable-Eingang basierend auf der NAND-Verknuepfung */*

*/Q_NAND = /S * NQ_NAND + /En * NQ_NAND ;*

*/NQ_NAND = /R * Q_NAND + /En * Q_NAND ;*

/ statisches RS-Flip-Flop mit zusaetzlichem Enable-Eingang optimiert fuer disjunktive Normalform */*

*Q = Q * /R + Q * /En + S * En ;*

/ nicht genutzte Ausgaenge ausgeschaltet */*

Q2 = GND;

Q3 = GND;

Q4 = GND;

Q5 = GND;

Q6 = GND;

Beispiel-File: D_Latch.gds

Realisierung eines transparenten D-Type-Latches und eines D-Master-Slave-Flip-Flops im Vergleich:

Schalter 1: Dateneingang D,
Taster: Enable-Signal bzw. Clock-Signal

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
LED: D9: Ausgang Q des auf die Galstruktur optimierten D_Latches,
D8: Ausgang QS (Slave-Flip-Flop)
D7: Ausgang QM (Master-Flip-Flop)
(Einstellungen fuer den Takt nicht relevant)

CHIP D_Latch ispGAL22V10

SCLK En D NC NC NC NC MODE NC NC NC NC NC GND
SDI NC Q0 Q1 Q2 Q3 Q4 SDO Q5 Q6 QM QS Q VCC

/* transparentes D-Type-Latch optimiert fuer disjunktive Normalform (nur eine OLMC genutzt) */

$$Q = D * En + Q * D + Q * /En ;$$

/* D-Master-Slave-Flip-Flop optimiert fuer disjunktive Normalform
(Zum besseren Vergleich ist der Slave bei En = 1 aktiviert.) */

$$QM = D * /En + QM * D + QM * En ;$$

$$QS = QM * En + QS * QM + QS * /En ;$$

/* nicht genutzte Ausgaenge ausgeschaltet */

$$Q0 = GND;$$

$$Q1 = GND;$$

$$Q2 = GND;$$

$$Q3 = GND;$$

$$Q4 = GND;$$

$$Q5 = GND;$$

$$Q6 = GND;$$

Beispiel-File: JK_MS_FlipFlop.gds

Realisierung eines JK-Master-Slave-Flip-Flops
im Vergleich zu einem echten taktflanken-gesteuerten JK-Flip-Flop:

Schalter 1: Eingang J,
Schalter 2: Eingang K,
Taster: Clk-Signal

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
LED: D9: Ausgang QM des Master-Flip-Flops (wechselt seinen Zustand bei Clk = 0),
D8: Ausgang QS des Slave-Flip-Flops (uebernimmt den Masterzustand bei Clk = 1),
D0: Ausgang QJK des taktflanken-gesteuerten JK-Flip-Flops zum Vergleich,
(Einstellungen fuer den Takt: intern (KEY))

Beobachtung:

Im Gegensatz zu einem taktflanken-gesteuerten JK-Flip-Flop kann das Master-Flip-Flop über J und K nicht nur zum Zeitpunkt der entsprechenden Taktflanke sondern während des gesamten aktiven Taktimpulses gesetzt bzw. rückgesetzt werden.

(Um das Verhalten beider Flip-Flops besser vergleichen zu können, ist das JK-Master-Slave-Flip-Flop so aufgebaut, dass es genau wie das interne D-Flip-Flop der OLMC seinen Ausgang bei der steigenden Taktflanke (also beim Drücken des Tasters) aktualisiert. Infolgedessen ist bei nicht gedrücktem Taster das Enable-Signal fuer das Master-Flip-Flop aktiv, so dass eine Änderung an J bzw. K zum spontanen Setzen bzw. Rücksetzen des Masters führen kann.)

CHIP JK_MS_FlipFlop ispGAL22V10

SCLK Clk J K NC NC NC MODE NC NC NC NC NC GND
SDI NC QJK Q1 Q2 Q3 Q4 SDO Q5 Q6 Q7 QS QM VCC

/* Realisierung des JK-Master-Slave-Flip-Flops mit je einer OLMC fuer das Master- sowie das Slave-Flip-Flop:*/

$QM = J * /QS * /Clk + QM * /K + QM * Clk + QM * /QS ;$

$QS = QM * Clk + QS * QM + QS * /Clk ;$

/* Realisierung des taktflanken-gesteuerten JK-Flip-Flops:

(Da die OLMCs des GALs nur D-Flip-Flops enthalten, wird das JK-Verhalten durch entsprechende Kombinatorik nachgebildet.) */

$QJK := QJK * /J * /K + /QJK * J * K + J * /K ;$

/* nicht genutzte Ausgaenge ausgeschaltet */

Q1 = GND;

Q2 = GND;

Q3 = GND;

Q4 = GND;

Q5 = GND;

Q6 = GND;

Q7 = GND;

Beispiel-File: JK_MS_ZAEHLER.gds

Realisierung eines asynchronen 3-Bit-Binaer-Zaehlers aus
durch reine Kombinatorik aufgebaute JK-Master-Slave-Flip-Flops:

Schalter 1: CEn: Freigabe des Zaehlers (CEn = 1 : Zaehler zaehlt, CEn = 0 : Zaehler gesperrt)

Taster: Clk-Signal

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,

LED: D9: Ausgang Q2 des Zaehlers (Stellenwert 2^2 , hoechstwertigstes Bit),

D8: Ausgang Q1 des Zaehlers (Stellenwert 2^1),

D7: Ausgang Q0 des Zaehlers (Stellenwert 2^0 , niedrigstwertigstes Bit)

D0...D2: Slave-Ausgaenge (nicht relevant)

(Einstellungen fuer den Takt: intern (KEY))

Beobachtung:

Auch mit JK-Master-Slave-Flip-Flops laesst sich ein Zaehler realisieren. In der vorliegenden

Version ist dieser Zaehler allerdings asynchron getaktet, da der Ausgang des Flip-Flops mit

der niedrigeren Wertigkeit als Taktsignal fuer das naechst hoehere Flip-Flop verwendet wird.

CHIP JK_MS_Zaehler ispGAL22V10

SCLK Clk CEn NC NC NC NC MODE NC NC NC NC NC GND
SDI NC QM0 QM1 QM2 Q3 Q4 SDO Q5 Q6 QS0 QS1 QS2 VCC

/* 1. JK-Master-Slave-Flip-Flop (J und K sind zu CEn zusammengefasst) */

$$QM0 = CEn * /QS0 * Clk + QM0 * /CEn + QM0 * /Clk + QM0 * /QS0 ;$$

$$QS0 = QM0 * /Clk + QS0 * QM0 + QS0 * Clk ;$$

/* 2. JK-Master-Slave-Flip-Flop (J und K sind staendig logisch 1, QS0 wird als Takt verwendet) */

$$QM1 = /QS1 * QS0 + QM1 * /QS0 + QM1 * /QS1 ;$$

$$QS1 = QM1 * /QS0 + QS1 * QM1 + QS1 * QS0 ;$$

Beispiel-File: Bin_4Bit_Zaehler_VR.gds und BCD_4Bit_Zaehler_VR_CE_L.gds

4-Bit-Binaer-Zaehler mit umkehrbarer Zaehrichtung (up / down)

Schalter 1: Zaehrichtung: UP=1: vorwärts, UP=0: rueckwaerts,
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
Takteinstellung: zunaechst Taster (KEY), spaeter evtl. auch automatischer Takt (GEN),
Taktfrequenz nach Belieben.

CHIP Bin_4Bit_Zaehler_VR ispGAL22V10

SCLK CLK UP NC NC NC NC MODE NC NC NC NC NC NC GND
SDI NC Q0 Q1 Q2 Q3 Q4 SDO Q5 Q6 Q7 Q8 Q9 VCC

Q0 := /Q0 ;

Q1 := /Q1 * Q0 * UP + Q1 * /Q0 * UP
+ /Q1 * /Q0 * /UP + Q1 * Q0 * /UP ;

Q2 := /Q2 * Q1 * Q0 * UP + Q2 * /Q1 * UP + Q2 * Q1 * /Q0 * UP
+ /Q2 * /Q1 * /Q0 * /UP + Q2 * Q1 * /UP + Q2 * Q0 * /UP ;

Q3 := /Q3 * Q2 * Q1 * Q0 * UP + Q3 * /Q2 * UP + Q3 * /Q1 * UP + Q3 * /Q0 * UP
+ /Q3 * /Q2 * /Q1 * /Q0 * /UP + Q3 * Q2 * /UP + Q3 * Q1 * /UP + Q3 * Q0 * /UP ;

/* ungenutzte Ausgaenge ausgeschaltet */

Q4 = GND;
Q5 = GND;
Q6 = GND;
Q7 = GND;
Q8 = GND;
Q9 = GND;

4-Bit-Binaer-Zaehler (0...15) mit variabler Zaehrichtung (UP), Count-Enable-Eingang (CE)
und Moeglichkeit des Parallel-Ladens eines Zaehlerstandes (Load)

Schalter 1: Count-Enable (Zaehlerfreigabe: Zaehler zaehlt bei CE=1, Zaehlerstand eingefroren bei CE=0),
Schalter 2: Zaehrichtung: UP=1: vorwärts, UP=0: rueckwaerts,
Schalter 3: Zaehler auf Anfangswert setzen, falls Load=1,
Schalter 5 bis 8: Eingabe der 4-Bit-Binaer-Zahl als Anfangswert (Stellenwert: Sch.5: 2³ (links), Sch.6: 2², Sch.7: 2¹, Sch.8: 2⁰ (rechts)),
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,

Takteinstellung: interner Takt, zunaechst Taster (KEY), spaeter evtl. auch automatischer Takt (GEN),
Taktfrequenz nach Belieben.

CHIP Bin_4Bit_Zaehler_VR_CE_L ispGAL22V10

SCLK CLK CE UP Load NC D MODE C B A NC NC GND
SDI NC Q0 Q1 Q2 Q3 Q4 SDO Q5 Q6 Q7 Q8 Q9 VCC

$Q0 := CE * /Load * /Q0 + /CE * Q0 + Load * A ;$

$Q1 := CE * /Load * /Q1 * Q0 * UP + CE * /Load * Q1 * /Q0 * UP$
 $+ CE * /Load * /Q1 * /Q0 * /UP + CE * /Load * Q1 * Q0 * /UP + /CE * Q1 + Load * B ;$

$Q2 := CE * /Load * /Q2 * Q1 * Q0 * UP + CE * /Load * Q2 * /Q1 * UP + CE * /Load * Q2 * Q1 * /Q0 * UP$
 $+ CE * /Load * /Q2 * /Q1 * /Q0 * /UP + CE * /Load * Q2 * Q1 * /UP + CE * /Load * Q2 * Q0 * /UP + /CE * Q2 + Load * C ;$

$Q3 := CE * /Load * /Q3 * Q2 * Q1 * Q0 * UP + CE * /Load * Q3 * /Q2 * UP + CE * /Load * Q3 * /Q1 * UP + CE * /Load * Q3 * /Q0 * UP$
 $+ CE * /Load * /Q3 * /Q2 * /Q1 * /Q0 * /UP + CE * /Load * Q3 * Q2 * /UP + CE * /Load * Q3 * Q1 * /UP + CE * /Load * Q3 * Q0 * /UP$
 $+ /CE * Q3 + Load * D ;$

/ ungenutzte Ausgaenge ausgeschaltet */*

Q4 = GND;

Q5 = GND;

Q6 = GND;

Q7 = GND;

Q8 = GND;

Q9 = GND;

Beispiel-File: BCD_4Bit_Zaehler_VR.gds

4-Bit-Binaer-Zaehler mit umkehrbarer Zaehlrichtung (up / down)

Schalter 1: Zaehlrichtung: UP=1: vorwaerts, UP=0: rueckwaerts,
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
Takteinstellung: zunaechst Taster (KEY), spaeter evtl. auch automatischer Takt (GEN),
Taktfrequenz nach Belieben.

CHIP Bin_4Bit_Zaehler_VR ispGAL22V10

SCLK CLK UP NC NC NC NC MODE NC NC NC NC NC GND
SDI NC Q0 Q1 Q2 Q3 Q4 SDO Q5 Q6 Q7 Q8 Q9 VCC

Q0 := /Q0 ;

Q1 := /Q1 * Q0 * UP + Q1 * /Q0 * UP
+ /Q1 * /Q0 * /UP + Q1 * Q0 * /UP ;

Q2 := /Q2 * Q1 * Q0 * UP + Q2 * /Q1 * UP + Q2 * Q1 * /Q0 * UP
+ /Q2 * /Q1 * /Q0 * /UP + Q2 * Q1 * /UP + Q2 * Q0 * /UP ;

Q3 := /Q3 * Q2 * Q1 * Q0 * UP + Q3 * /Q2 * UP + Q3 * /Q1 * UP + Q3 * /Q0 * UP
+ /Q3 * /Q2 * /Q1 * /Q0 * /UP + Q3 * Q2 * /UP + Q3 * Q1 * /UP + Q3 * Q0 * /UP ;

/* ungenutzte Ausgaenge ausgeschaltet */

Q4 = GND;

Q5 = GND;

Q6 = GND;

Q7 = GND;

Q8 = GND;

Q9 = GND;

Beispiel-File: Stunden_4Bit_Zaehler.gds

4-Bit-Zaehler, der im Binaercode von 1 bis 12 zaehlt (z.B. als Zaehler fuer die Stundenanzeige bei einer Uhr mit 12-Stunden-Anzeige)

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
Takteinstellung: zunaechst Taster (KEY), spaeter evtl. auch automatischer Takt (GEN),
Clock-Generator nach Belieben.

CHIP Stunden_4Bit_Zaehler ispGAL22V10

SCLK CLK NC NC NC NC NC MODE NC NC NC NC NC GND
SDI NC Q0 Q1 Q2 Q3 Q4 SDO Q5 Q6 Q7 Q8 Q9 VCC

Q0 := /Q0 ;

Q1 := /Q1 * Q0 + Q1 * /Q0 ;

Q2 := /Q2 * Q1 * Q0 + Q2 * /Q1 * /Q3 + Q2 * /Q0 * /Q3 ;

Q3 := /Q3 * Q2 * Q1 * Q0 + Q3 * /Q2 ;

/* ungenutzte Ausgaenge ausgeschaltet */

Q4 = GND;

Q5 = GND;

Q6 = GND;

Q7 = GND;

Q8 = GND;

Q9 = GND;

Beispiel-File: Gray_4Bit_Zaehler.gds

4-Bit-Gray-Vorwaertszaehler

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
Takteeinstellung: zunaechst Taster (KEY), spaeter evtl. auch automatischer Takt (GEN),
Taktfrequenz nach Belieben.

CHIP Gray_4Bit_Zaehler ispGAL22V10

SCLK CLK NC NC NC NC NC MODE NC NC NC NC NC GND
SDI NC Q0 Q1 Q2 Q3 Q4 SDO Q5 Q6 Q7 Q8 Q9 VCC

$Q0 := /Q3 * /Q2 * /Q1 + /Q3 * Q2 * Q1 + Q3 * Q2 * /Q1 + Q3 * /Q2 * Q1 ;$

$Q1 := /Q3 * /Q2 * Q0 + /Q3 * Q1 * /Q0 + Q3 * Q2 * Q0 + Q3 * Q1 * /Q0 ;$

$Q2 := /Q3 * Q1 * /Q0 + /Q3 * Q2 * Q0 + Q2 * /Q1 * /Q0 + Q3 * Q2 * Q0 ;$

$Q3 := Q2 * /Q1 * /Q0 + Q3 * Q2 * Q0 + Q3 * Q1 * /Q0 + Q3 * /Q2 * Q0 ;$

/* ungenutzte Ausgaenge ausgeschaltet */

Q4 = GND;

Q5 = GND;

Q6 = GND;

Q7 = GND;

Q8 = GND;

Q9 = GND;

Beispiel-File: Schieberegister.gds

10-Bit-Schieberegister mit variabler Schieberichtung, seriellem Dateneingang und wahlbarem Ringspeichermodus

Schalter 1: serieller Dateneingang,
Schalter 2: Schieberichtung,
Schalter 3: Ringspeichermodus,
Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,
Takteinstellung: interner Takt, zunaechst Taster (KEY), spaeter evtl. auch automatischer Takt (GEN),
Taktfrequenz nach Belieben.

CHIP Schieberegister ispGAL22V10

SCLK CLK DI SD RM NC NC MODE NC NC NC NC NC GND
SDI NC Q0 Q1 Q2 Q3 Q4 SDO Q5 Q6 Q7 Q8 Q9 VCC

$Q0 := \text{/SD} * Q1 + \text{SD} * \text{RM} * Q9 + \text{SD} * \text{/RM} * \text{DI} ;$

$Q1 := \text{/SD} * Q2 + \text{SD} * Q0 ;$

$Q2 := \text{/SD} * Q3 + \text{SD} * Q1 ;$

$Q3 := \text{/SD} * Q4 + \text{SD} * Q2 ;$

$Q4 := \text{/SD} * Q5 + \text{SD} * Q3 ;$

$Q5 := \text{/SD} * Q6 + \text{SD} * Q4 ;$

$Q6 := \text{/SD} * Q7 + \text{SD} * Q5 ;$

$Q7 := \text{/SD} * Q8 + \text{SD} * Q6 ;$

$Q8 := \text{/SD} * Q9 + \text{SD} * Q7 ;$

$Q9 := \text{/SD} * \text{/RM} * \text{DI} + \text{/SD} * \text{RM} * Q0 + \text{SD} * Q8 ;$

Beispiel-File: Zufall_Bit_Gen.gds

Pseudo-Zufalls-Bit-Generator auf Basis eines 5-Bit-Schieberegisters

Abhaengig vom gewaehlten rueckgekoppelten Ausgang laesst sich die jeweilige Laenge eines Zyklusses bis zum Wiederauftreten eines Zustands untersuchen.

Schalter 1 und 2: variable Rueckkopplung von Q0 bis Q3 fuer EXOR-Verknuepfung mit Q4,

Schalter 3: paralleles Laden des Registers,

Schalter 4 bis 8: Dateneingabe zum parallelen Laden eines Registerzustands.

Jumper fuer LED-Zeile gesetzt, Jumper fuer 7-Segmentanzeige nicht gesetzt,

Takteinstellung: zunaechst Taster (KEY), spaeter evtl. auch automatischer Takt (GEN),

Taktfrequenz nach eigener Wahl.

CHIP Zufall_Bit_Gen ispGAL22V10

```
SCLK CLK RQ0 RQ1 SET E D MODE C B A NC NC GND
SDI NC Q4 Q3 Q2 Q1 Q0 SDO Q5 Q6 Q7 Q8 Q9 VCC
```

```
Q0 := /SET * Q4 * /Q2 * RQ1 * RQ0 + /SET * /Q4 * Q2 * RQ1 * RQ0
      + /SET * Q3 * /Q2 * RQ1 * /RQ0 + /SET * /Q3 * Q2 * RQ1 * /RQ0
      + /SET * Q3 * /Q1 * /RQ1 * RQ0 + /SET * /Q3 * Q1 * /RQ1 * RQ0
      + /SET * Q3 * /Q0 * /RQ1 * /RQ0 + /SET * /Q3 * Q0 * /RQ1 * /RQ0
      + SET * E ;
```

```
Q1 := /SET * Q0 + SET * D ;
```

```
Q2 := /SET * Q1 + SET * C ;
```

```
Q3 := /SET * Q2 + SET * B ;
```

```
Q4 := /SET * Q3 + SET * A ;
```

```
/* unbenutzte Ausgaenge ausgeschaltet */
```

```
Q5 = GND;
```

```
Q6 = GND;
```

```
Q7 = GND;
```

```
Q8 = GND;
```

```
Q9 = GND;
```