11. Sequentielle Komponenten

Digitale Speicherelemente

- asynchrone Speicherelemente (asynchrone Latches)
  Können ihren Zustandsspeicher jederzeit ändern
  Bsp: asynchrones RS-Latch

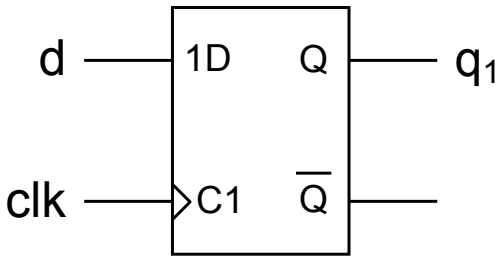
- synchrone Speicherelemente
  Ändern ihren Zustandsspeicher nur zu festgelegten Zeitpunkten, synchron mit einem Taktsignal

  • Taktzustandsgesteuerte Speicherelemente (Latches)
  Ändern ihren Zustandswert nur bei festgelegten Signalwert 1 (oder 0) des Taktsignals
  Bsp: RS-Latch, D-Latch

  • Taktflankengesteuerte Speicherelemente (FlipFlops)
  Ändern ihren Zustandswert nur bei Wechsel des Signalwerts des Taktsignals 0 → 1 (oder 1 → 0), sogenannte Taktflanke
  Bsp: D-FlipFlop, JK-FlipFlop, T-FlipFlop, RS-FlipFlop

Heute hauptsächlich FlipFlops benutzt! → Fokus auf diese Elemente

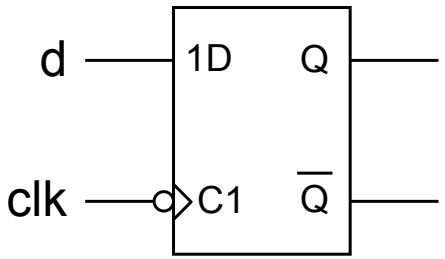
Synchrones D-FlipFlop



Funktionstabelle

clk	d	q^{t+1}
0/1/↓	–	
↑	0	
↑	1	

positiv taktflankengesteuert

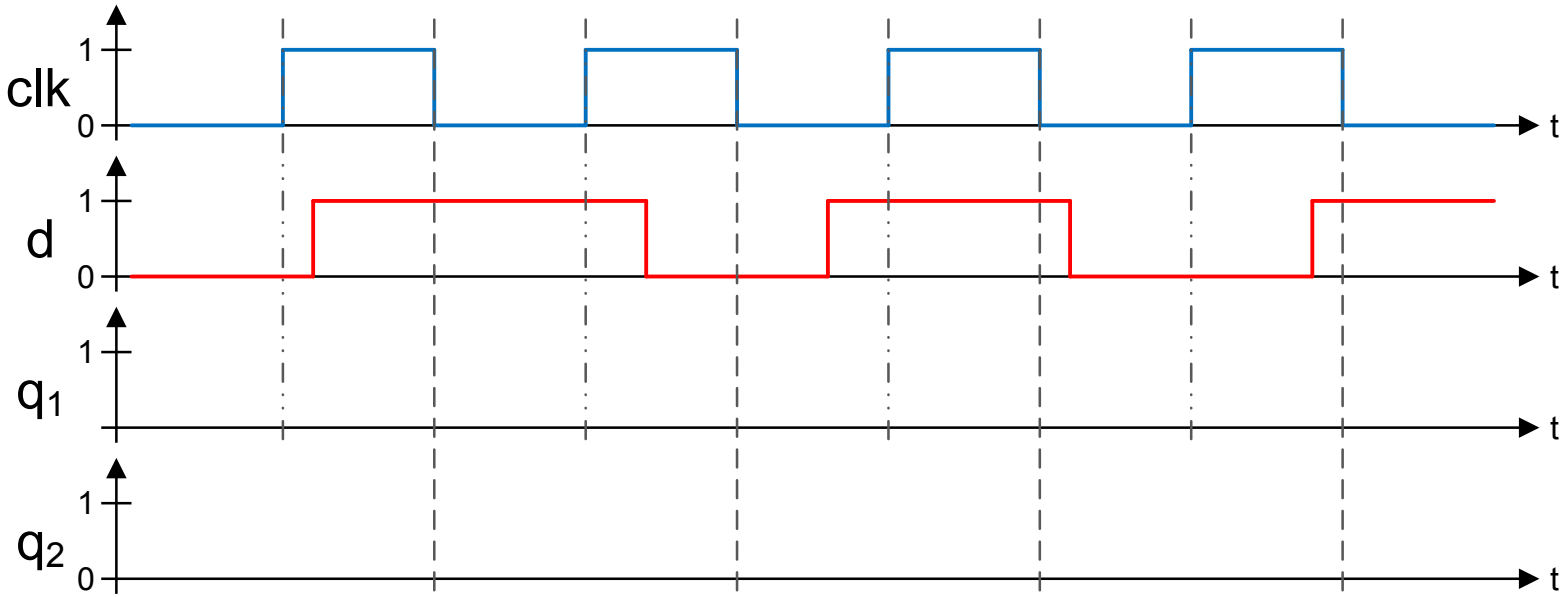


Funktionstabelle

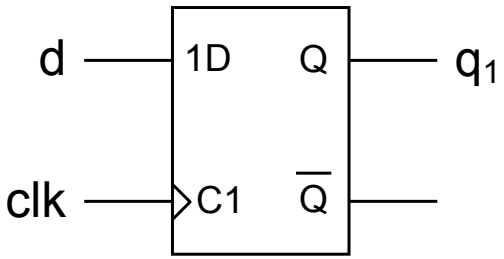
clk	d	q^{t+1}
0/1/↑	–	
↓	0	
↓	1	

negativ taktflankengesteuert

Zeitdiagramm:



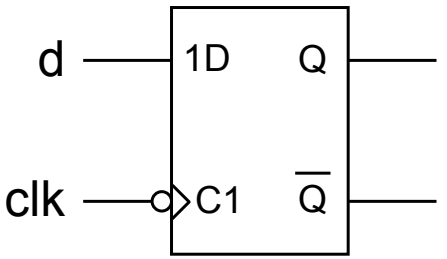
Synchrones D-FlipFlop



Funktionstabelle

clk	d	q^{t+1}
0/1/↓	–	q^t
↑	0	0
↑	1	1

positiv taktflankengesteuert

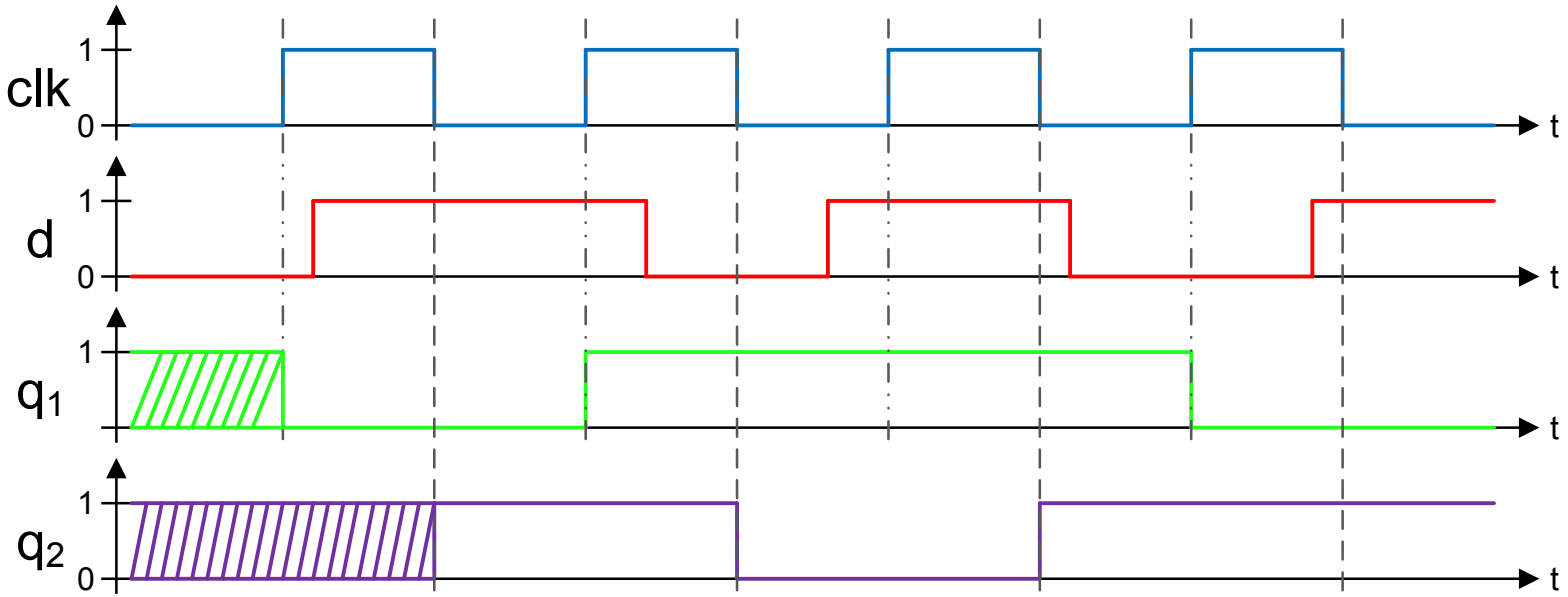


Funktionstabelle

clk	d	q^{t+1}
0/1/↑	–	q^t
↓	0	0
↓	1	1

negativ taktflankengesteuert

Zeitdiagramm:



Synchrones D-FlipFlop

Bevorrechtigte Eingänge:

Erweiterung des D-FlipFlops um weitere Eingänge, welche Vorrang vor den Standard-Eingängen haben. Diese Eingänge werden benutzt, um das FlipFlop in einen definierten Zustand zu bringen.

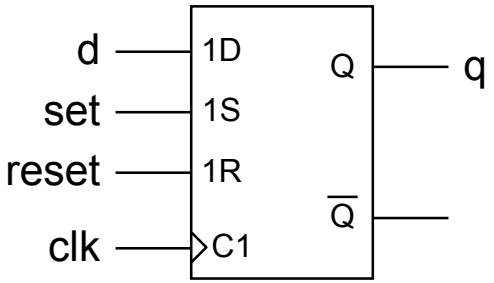
- synchrone bevorrechtigte Eingänge wirken sich nur bei der Taktflanke aus.

- asynchrone bevorrechtigte Eingänge wirken sich unmittelbar auf den Speicherzustand aus, unabhängig vom Taktsignal

Synchrones D-FlipFlop

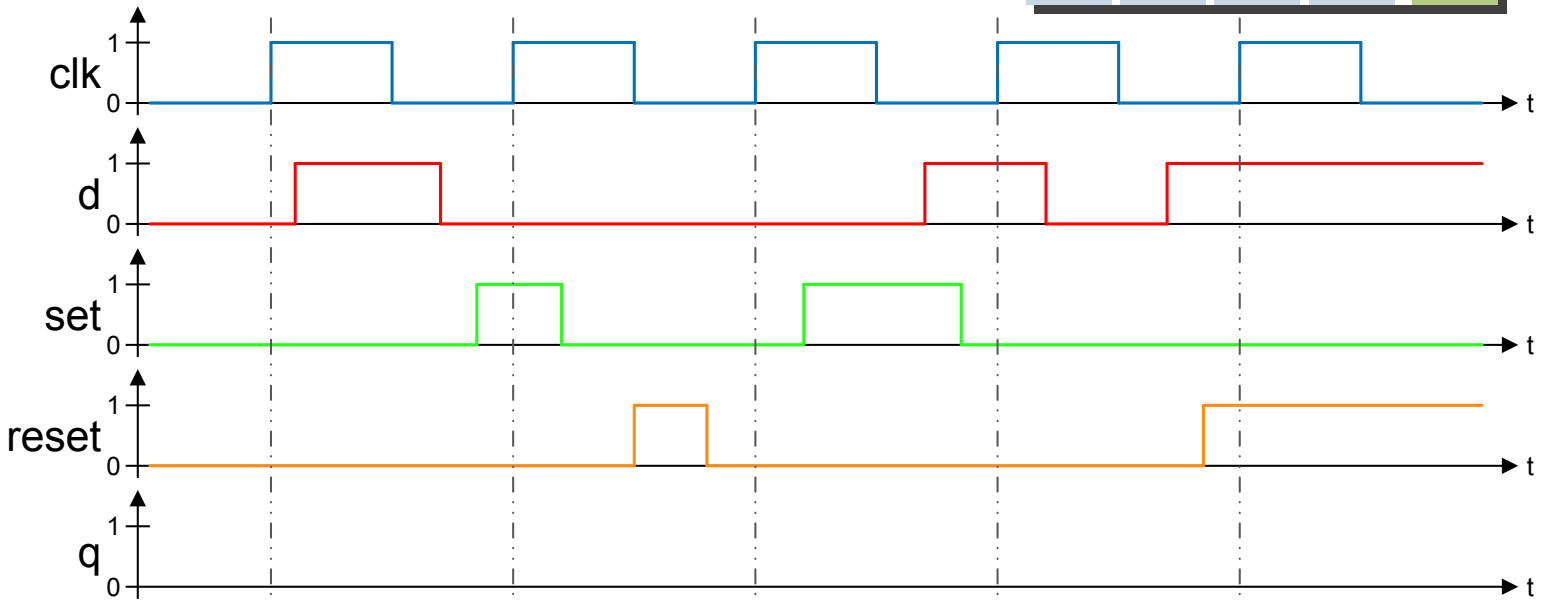
Synchrone Set und Reset Eingänge:

Die Setz- (Set) und Rücksetz- (Reset) Eingänge setzen den Speicherzustand des FlipFlops synchron mit der Taktflanke auf den Wert 1 bzw. 0, unabhängig vom d-Eingang.



clk	reset	set	d	q^{t+1}
0/1/↓	–	–	–	
↑	0			
↑	0			
↑	0			
↑	1			

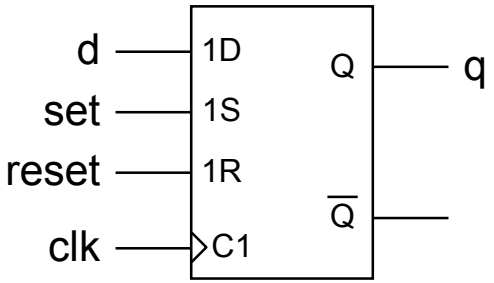
Zeitdiagramm:



Synchrones D-FlipFlop

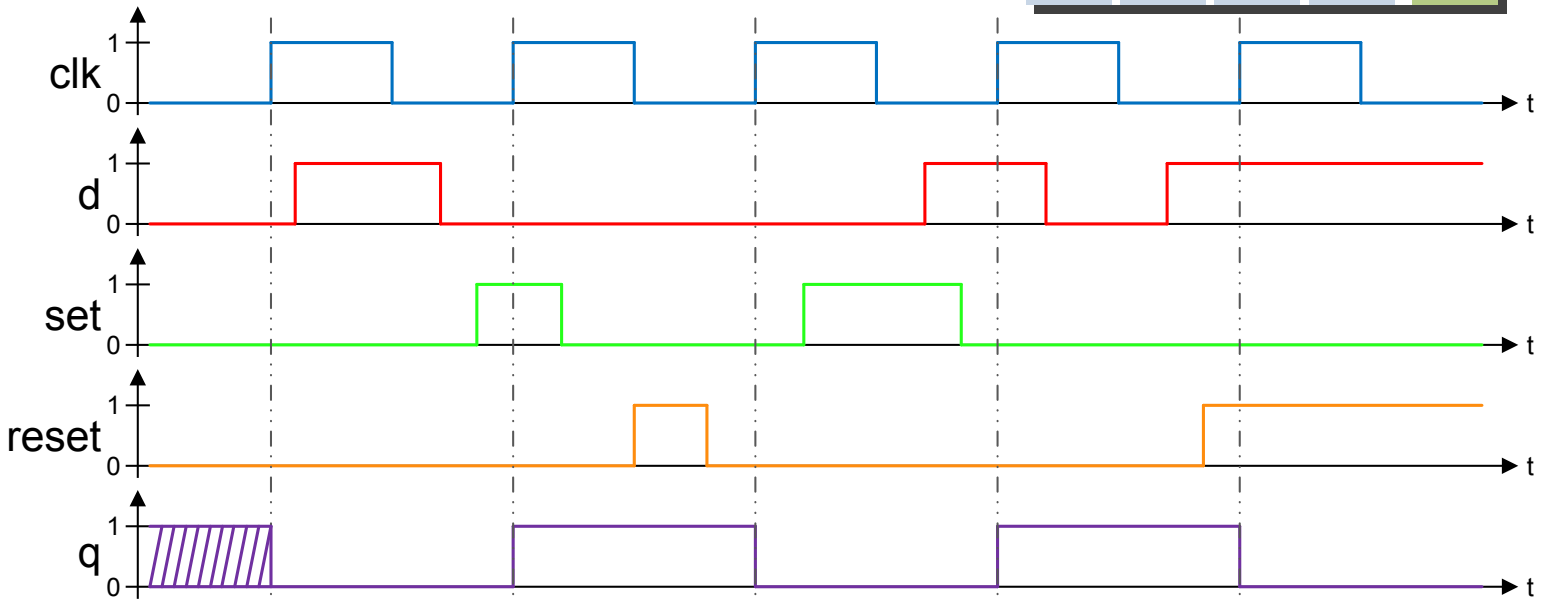
Synchrone Set und Reset Eingänge:

Die Setz- (Set) und Rücksetz- (Reset) Eingänge setzen den Speicherzustand des FlipFlops synchron mit der Taktflanke auf den Wert 1 bzw. 0, unabhängig vom d-Eingang.



clk	reset	set	d	q^{t+1}
0/1/↓	–	–	–	q^t
↑	0	0	0	0
↑	0	0	1	1
↑	0	1	–	1
↑	1	–	–	0

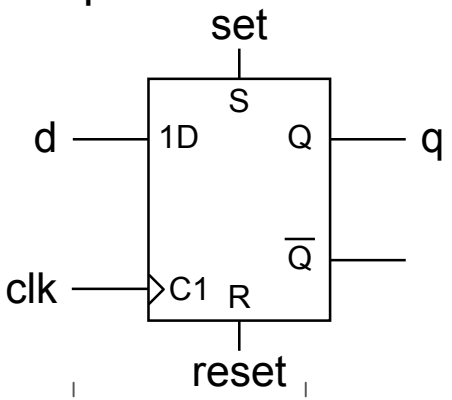
Zeitdiagramm:



Synchrones D-FlipFlop

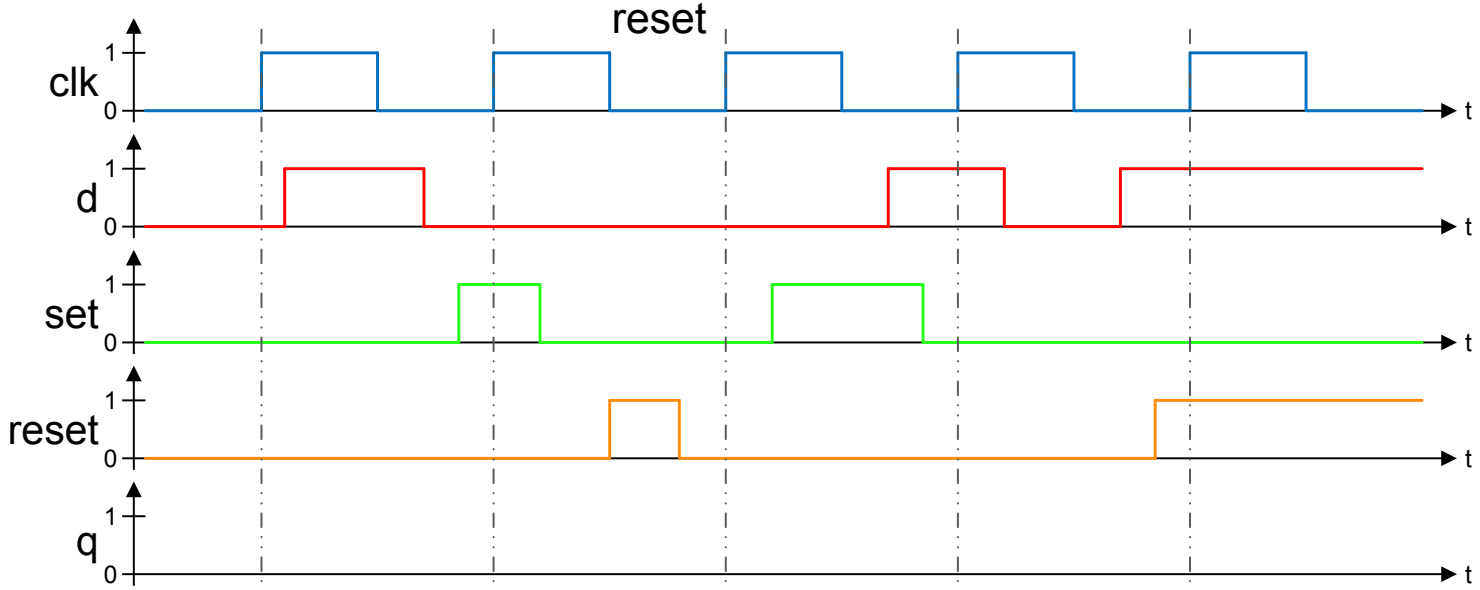
Asynchrone Set und Reset Eingänge:

Die asynchronen Setz- (Set) und Rücksetz- (Reset) Eingänge setzen den Speicherzustand des FlipFlops unmittelbar auf den Wert 1 bzw. 0, unabhängig vom Taktsignal.



clk	reset	set	d	q^{t+1}
0/1/↓	0	0	–	
↑	0			
↑	0			
–	0			
–	1			

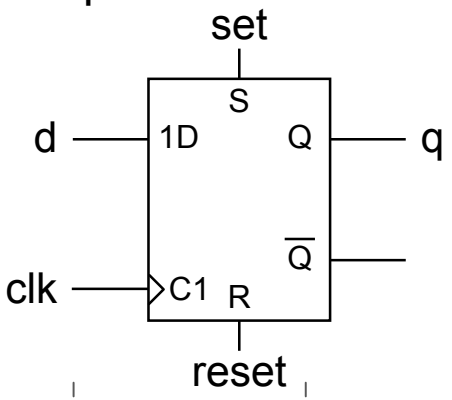
Zeitdiagramm:



Synchrones D-FlipFlop

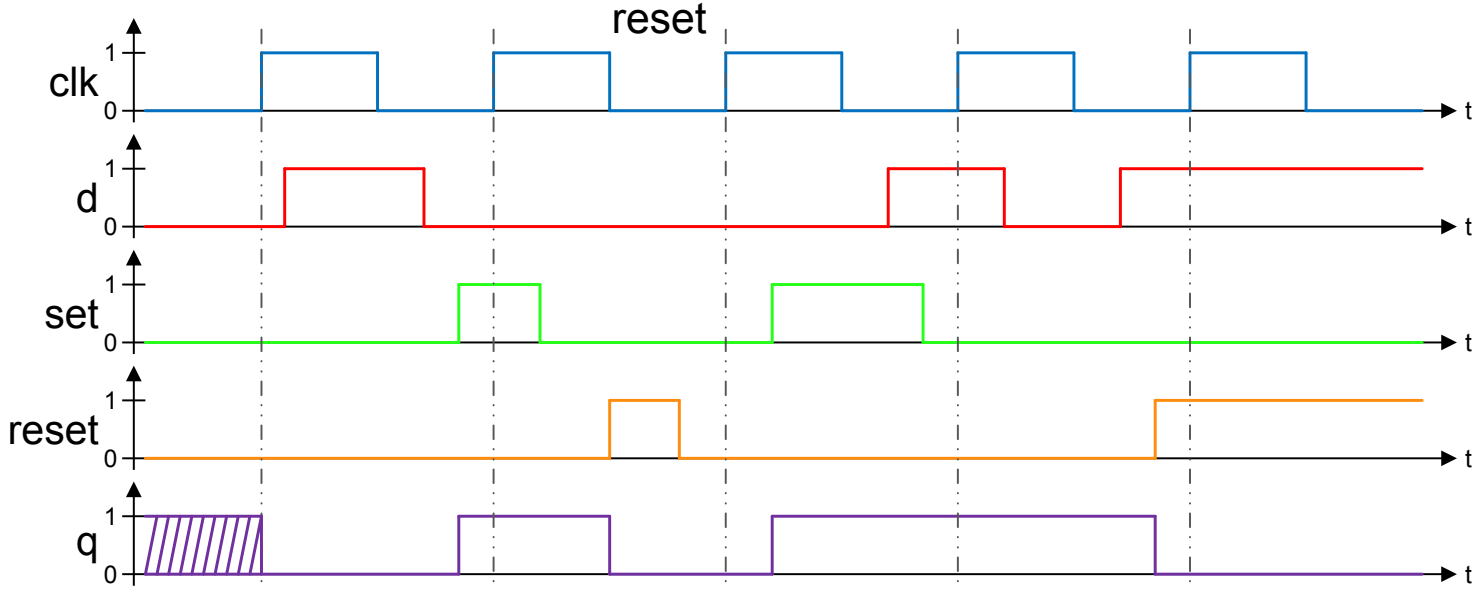
Asynchrone Set und Reset Eingänge:

Die asynchronen Setz- (Set) und Rücksetz- (Reset) Eingänge setzen den Speicherzustand des FlipFlops unmittelbar auf den Wert 1 bzw. 0, unabhängig vom Taktsignal.



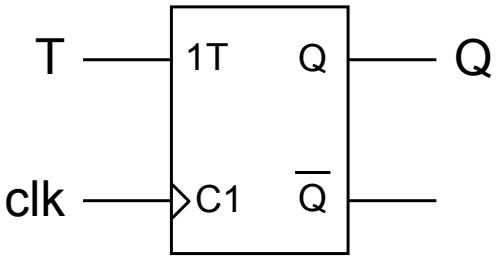
clk	reset	set	d	q^{t+1}
0/1/↓	0	0	–	q^t
↑	0	0	0	0
↑	0	0	1	1
–	0	1	–	1
–	1	–	–	0

Zeitdiagramm:



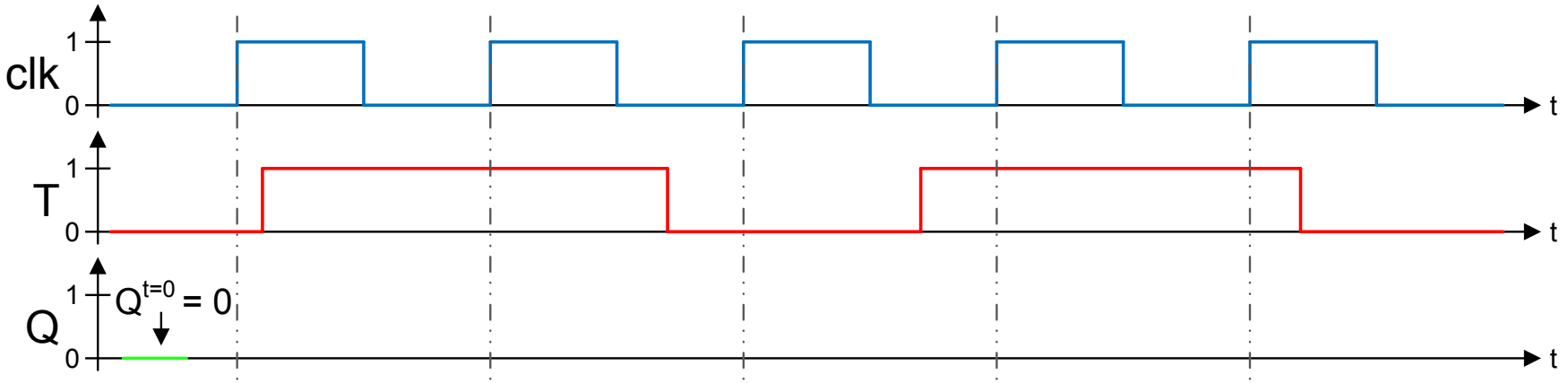
T-FlipFlop

Das Toggle-FlipFlop kann abhängig vom Eingang T seinen aktuellen Wert beibehalten, oder es wechselt mit jeder Taktflanke seinen Wert.



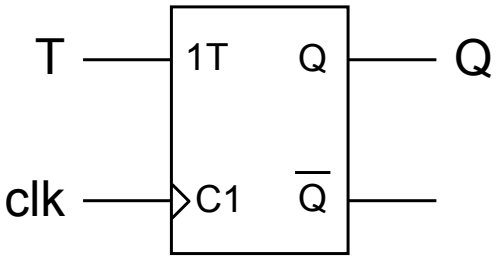
clk	T	Q^{t+1}
0/1/↓	–	
↑	0	
↑	1	

Zeitdiagramm:



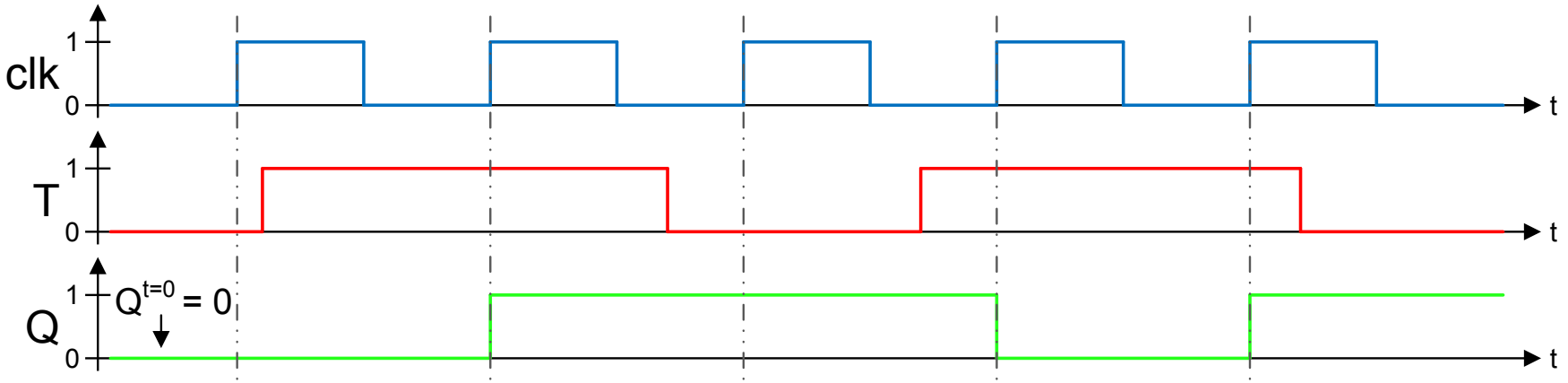
T-FlipFlop

Das Toggle-FlipFlop kann abhängig vom Eingang T seinen aktuellen Wert beibehalten, oder es wechselt mit jeder Taktflanke seinen Wert.



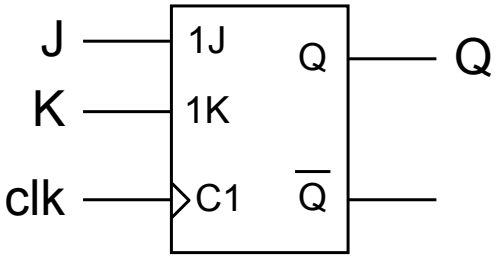
clk	T	Q^{t+1}
0/1/↓	–	Q^t
↑	0	Q^t
↑	1	$\overline{Q}^t$

Zeitdiagramm:



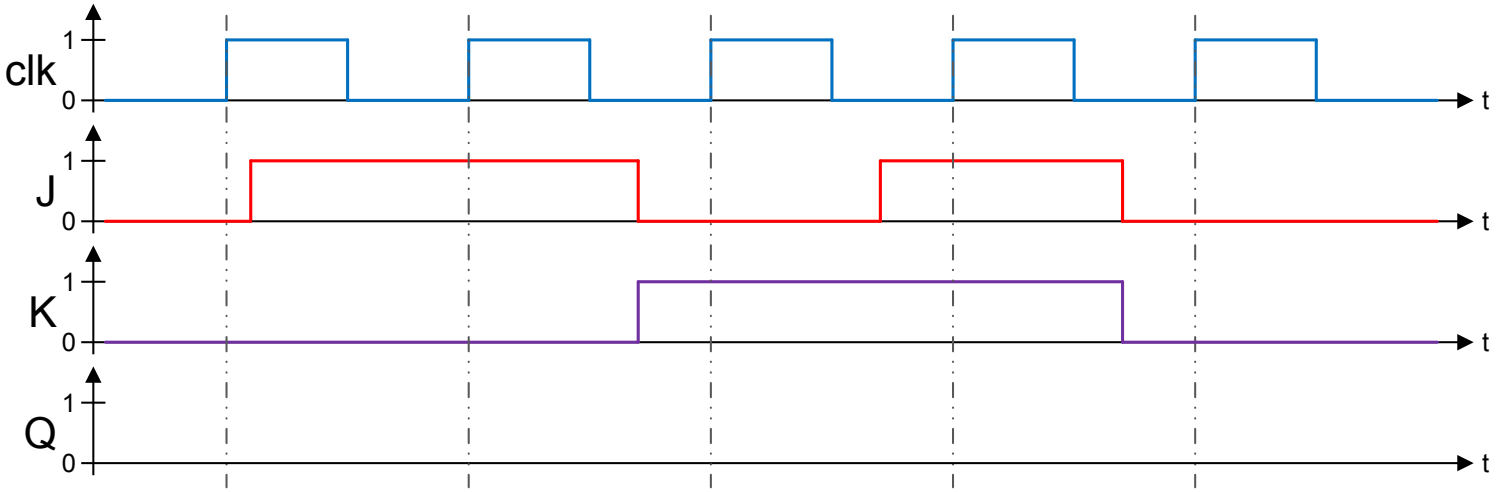
JK-FlipFlop

Das JK-FlipFlop kann abhängig von J, K seinen aktuellen Wert beibehalten, gesetz, gelöscht werden oder es wechselt mit jeder Taktflanke seinen Wert.



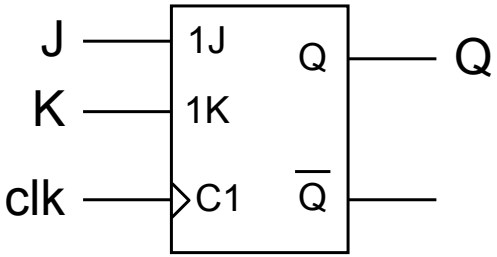
clk	J	K	Q^{t+1}
0/1/↓	–	–	
↑	0	0	
↑	0	1	
↑	1	0	
↑	1	1	

Zeitdiagramm:



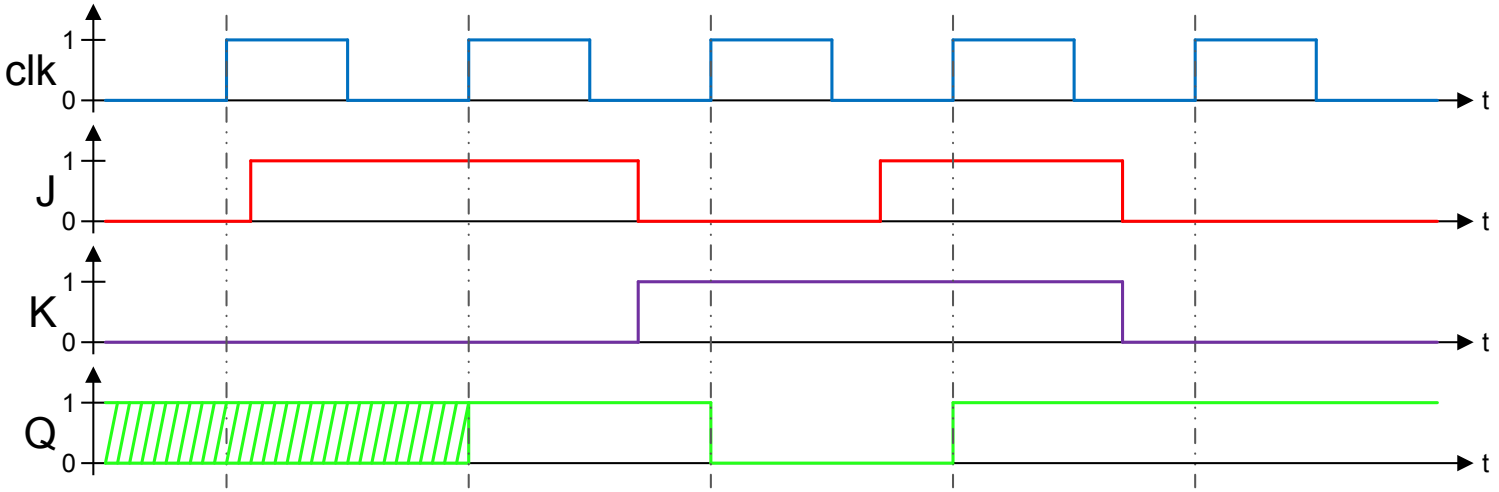
JK-FlipFlop

Das JK-FlipFlop kann abhängig von J, K seinen aktuellen Wert beibehalten, gesetz, gelöscht werden oder es wechselt mit jeder Taktflanke seinen Wert.



clk	J	K	Q^{t+1}
0/1/↓	–	–	Q^t
↑	0	0	Q^t
↑	0	1	0
↑	1	0	1
↑	1	1	$\overline{Q}^t$

Zeitdiagramm:

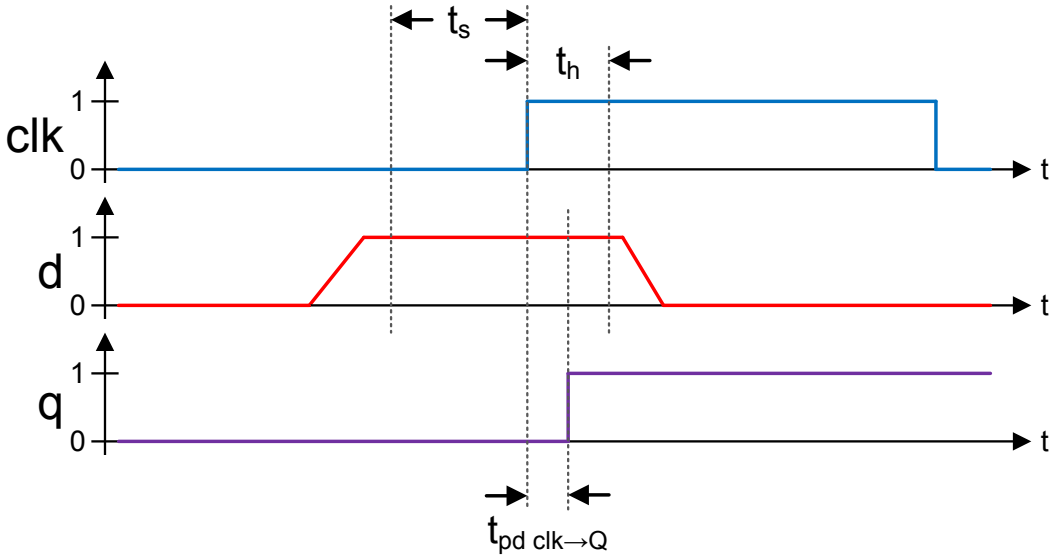


Zeitbedingungen

Für das korrekte Funktionieren der FlipFlops ist die Einhaltung von Zeitbedingungen erforderlich:

- Setup-Zeit t_s
  minimale Zeitdauer vor der Taktflanke, die das Signal am d-Eingang stabil anliegen muss
- Hold-Zeit t_h
  minimale Zeitdauer nach der Taktflanke, die das Eingangssignal stabil bleiben muss

Das Ausgangssignal nimmt erst nach der Zeit $t_{pdclk \rightarrow Q}$ seinen neuen Wert an.



Beispiel: Xilinx Virtex2 FPGA

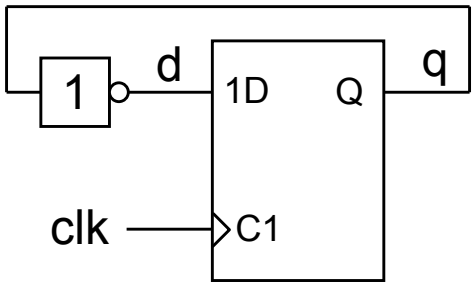
t_s: 0,208 ns

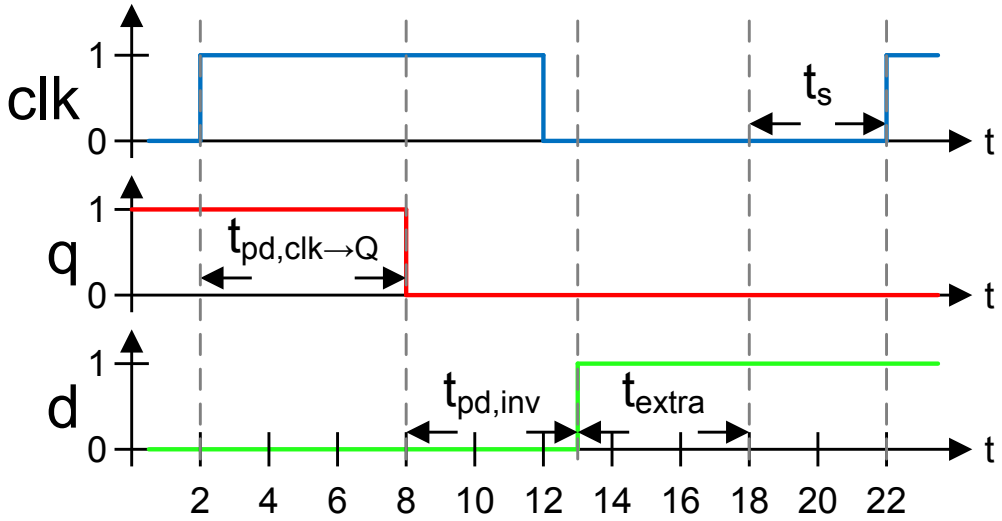
t_h: 0,12 ns

$t_{pdclk \rightarrow Q}$: 0,370 ns

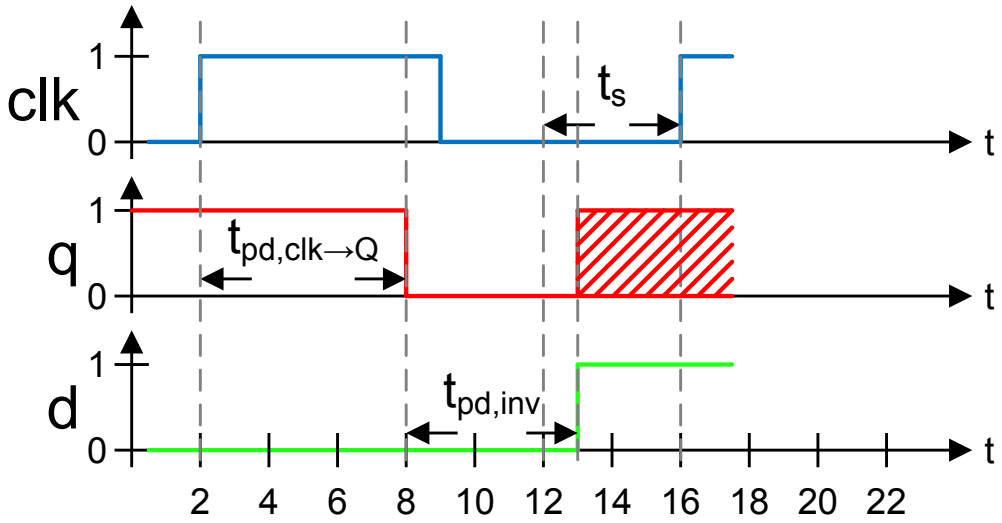
Zeitbedingungen

Beispiel:



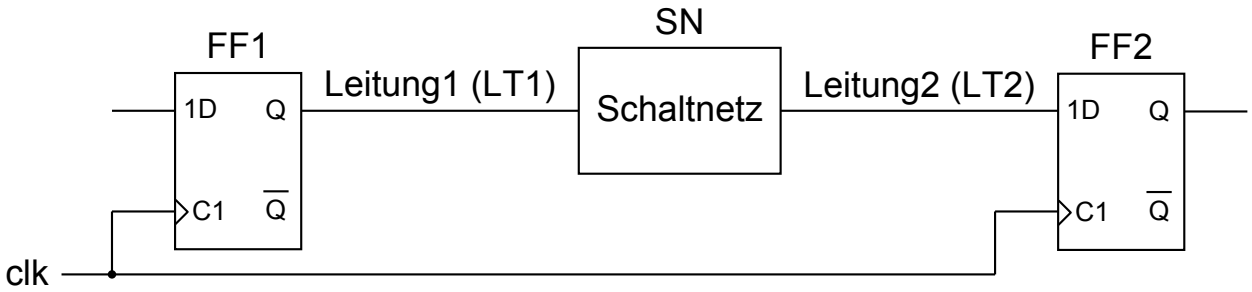


Setup-Zeit
Einhaltung



Setup-Zeit
Verletzung

Timing-Analyse



Bedingung für Setup-Zeit:

$$t_{pd,clk\to Q}^{FF1} + t_{pd}^{LT1} + t_{pd}^{SN} + t_{pd}^{LT2} + t_S^{FF2} \le T_{clk}$$

Daraus ergibt sich für die maximale Taktfrequenz:

$$f_{clk,\max} = \frac{1}{T_{clk,\min}} = \frac{1}{t_{pd,clk\to Q}^{FF1} + t_{pd}^{LT1} + t_{pd}^{SN} + t_{pd}^{LT2} + t_S^{FF2}}$$

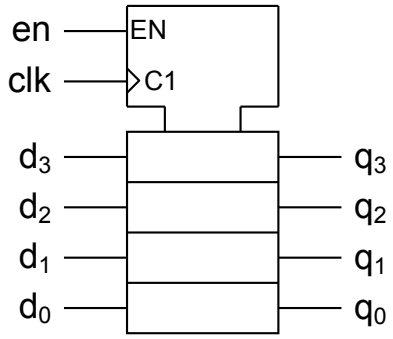
Bedingung für Hold-Zeit:

$$t_{pd,clk\to Q}^{FF1} + t_{pd}^{LT1} + t_{pd}^{SN} + t_{pd}^{LT2} > t_H^{FF2}$$

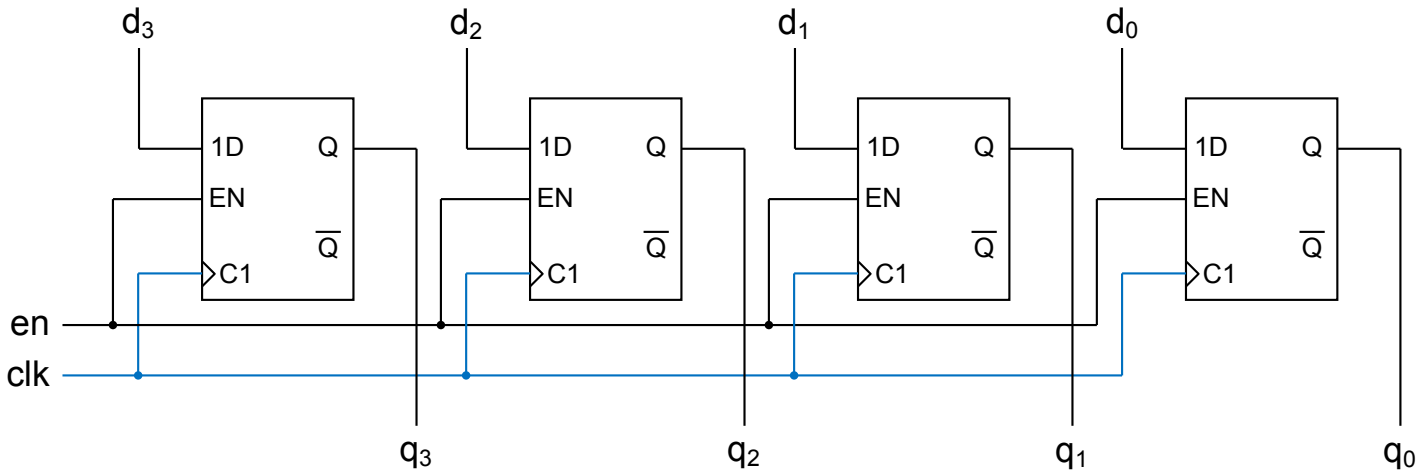
Register

Unter einem *Register* versteht man die parallele Anordnung von mehreren Speicherelementen, die von einem gemeinsamen Takt gespeist werden. Sie dienen dazu, vollständige Datenwörter zu speichern.

Beispiel: 4 Bit Register



clk	en	Funktion
0/1/↓	–	Speichern
–	0	Speichern
↑	1	Laden



Schieberegister - Unidirektional

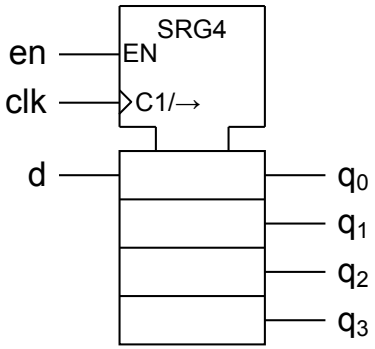
Beim *Schieberegister* wird das Datenwort nicht parallel eingelesen, sondern seriell über einen Eingang geladen. Bei einem n-Bit breiten Schieberegister ist das Datenwort nach n Takten vollständig eingelesen.

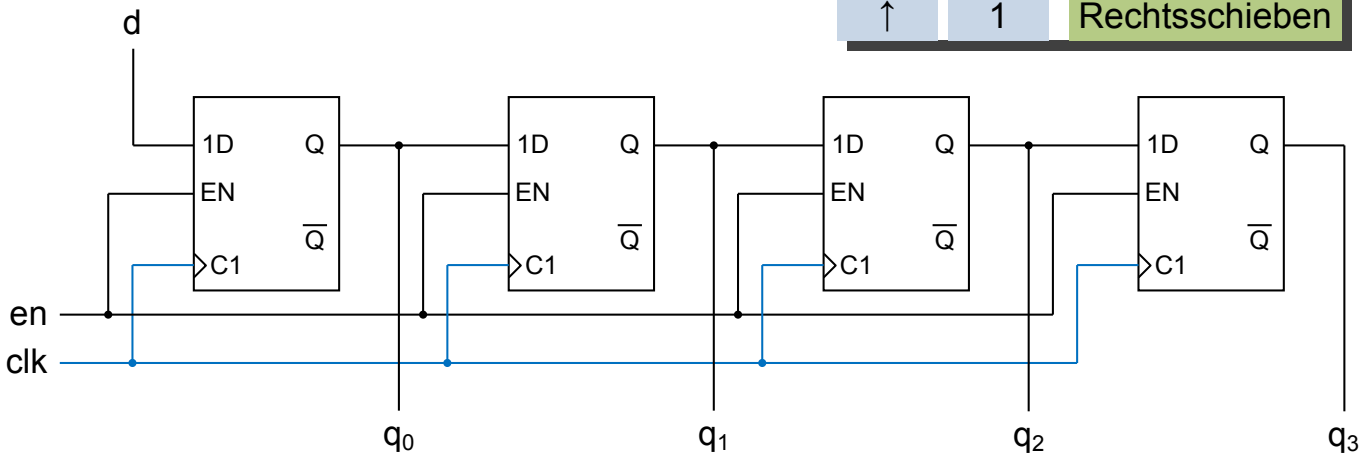
Anwendungen:

- Parallelisierung von seriellen Datenströmen
- bei vorzeichenlosen Zahlen entspricht Rechtsschieben (right shift) einer Multiplikation und Linksschieben (left shift) einer Division, wenn der Dateneingang auf 0 gesetzt wird.

clk	en	Funktion
0/1/↓	–	Speichern
–	0	Speichern
↑	1	Rechtsschieben

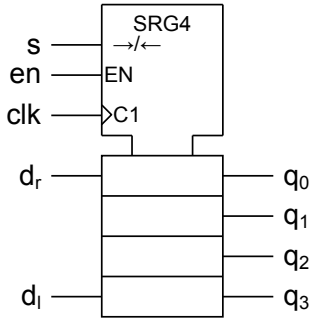
Beispiel: 4 Bit unidirektionales Schieberegister



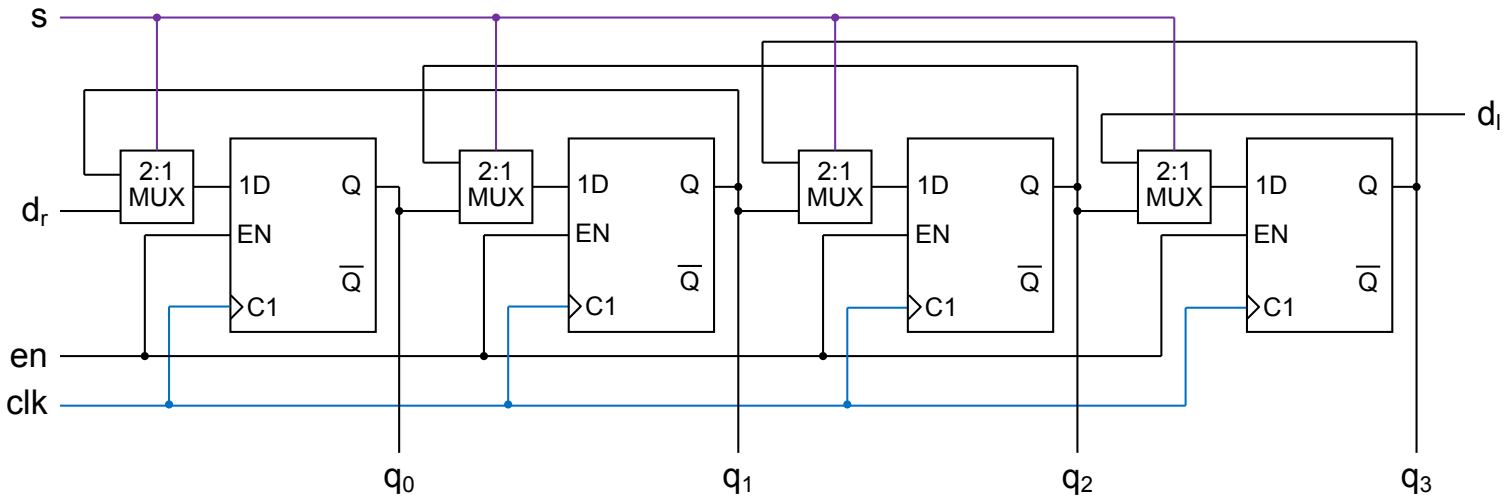


Schieberegister - Bidirektional

Beim *bidirektionalen Schieberegister* kann in 2 Richtungen geschoben werden. Dazu gibt es 2 Dateneingänge d_l und d_r. Die Auswahl der Schieberichtung wird über ein Steuersignal s durchgeführt.

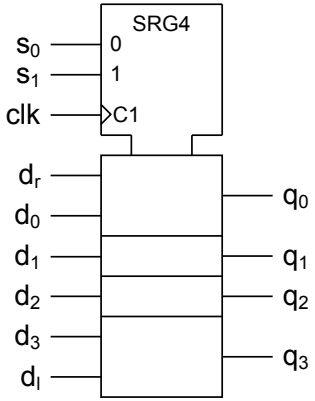


clk	en	s	Funktion
0/1/↓	–	–	Speichern
–	0	–	Speichern
↑	1	0	Rechtsschieben
↑	1	1	Linksschieben

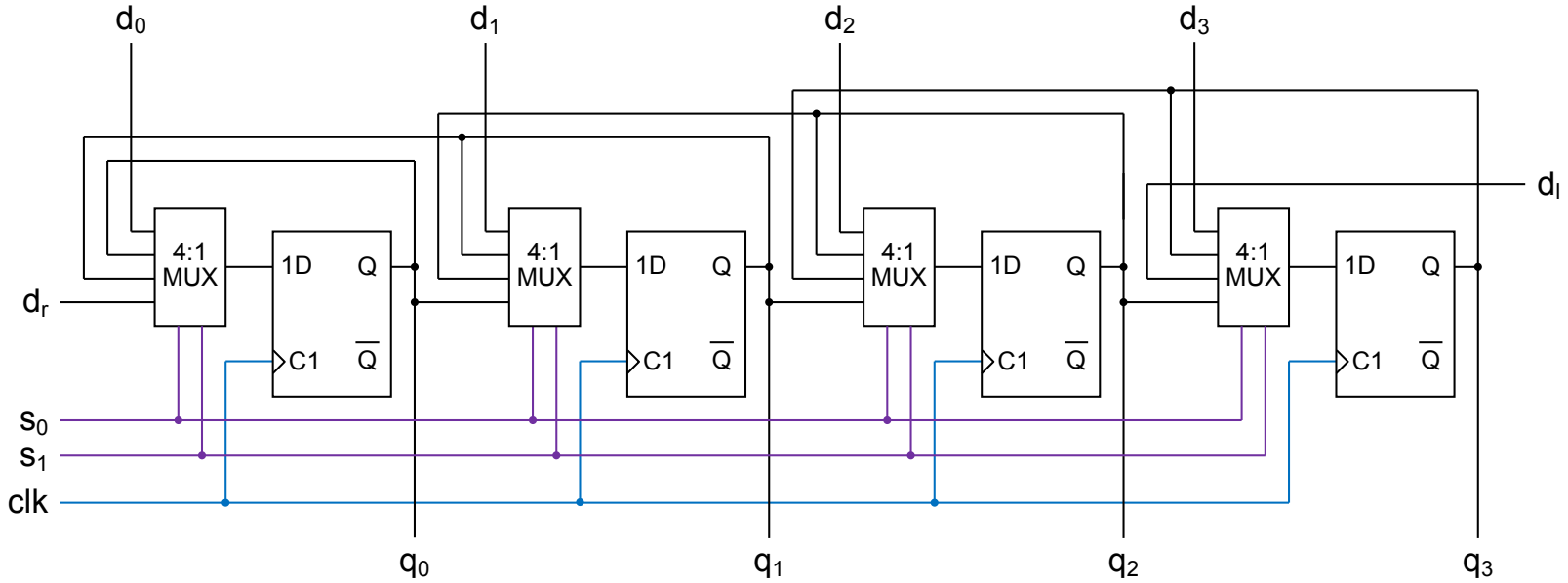


Universalregister

Das *Universalregister* kann sowohl parallel als auch seriell geladen werden.

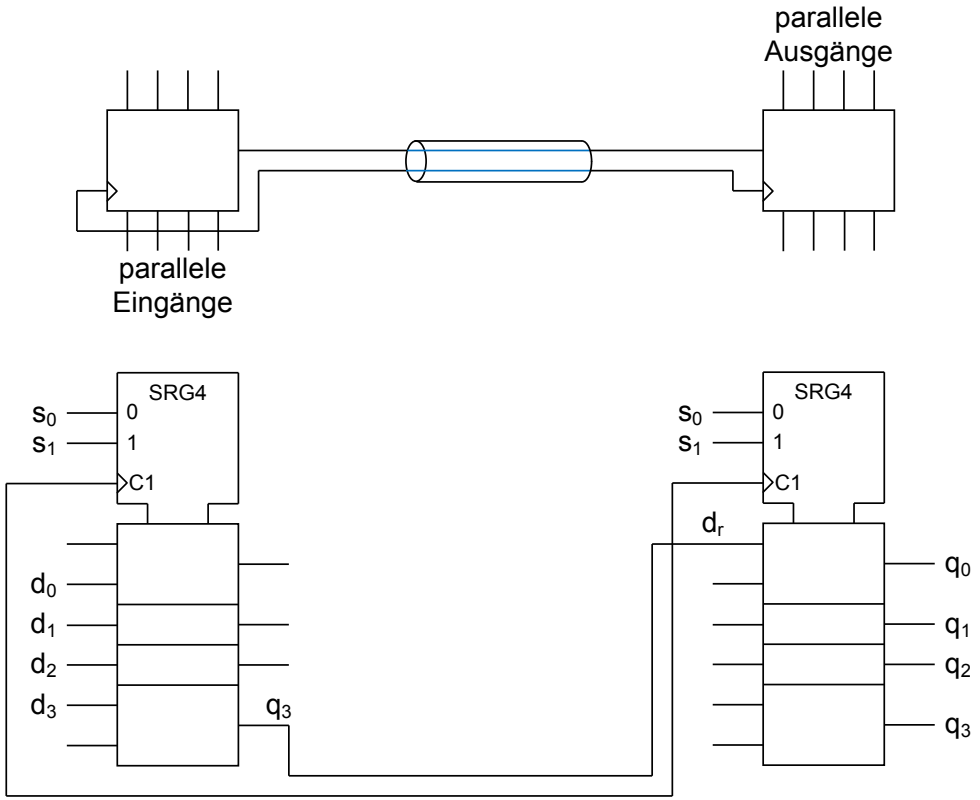


clk	s_1	s_0	Funktion
0/1/↓	–	–	Speichern
↑	0	0	Rechtsschieben
↑	0	1	Linksschieben
↑	1	0	Speichern
↑	1	1	Laden



Universalregister

Das Universalregister kann zur seriellen Übermittlung von Daten eingesetzt werden. Der Sender lädt das zu übertragende Datenwort parallel in sein Register ein und schiebt es dann nach rechts. Der Empfänger schiebt das empfangene Bit in sein Register und nach n – Takten kann er das Datenwort an seinen parallelen Ausgängen ablesen.



Zähler

Zähler durchlaufen nacheinander eine gewisse Wertefolge. Sie werden zumeist benutzt, um Zeitmessungen durchzuführen oder Zeitabläufe vorzugeben. Zähler unterscheiden sich in mehreren Eigenschaften.

Eigenschaften von Zählern:

- Synchrone / Asynchrone Realisierung
- Kodierungen (Binär, BCD, Gray-Code)
- Steuersignal für Zählrichtung
- Steuersignal für Zählen, Laden, Anhalten
- Synchroner / Asynchroner Set- oder Reset - Eingang

Binärzähler

Beispiel: 3 Bit Vorwärts Binär-Zähler

Durchläuft nacheinander die Zustände:
000 → 001 → 010 → 011 → 100 → 101 → 110 → 111 → 000

q_2^t	q_1^t	q_0^t	q_2^{t+1}	q_1^{t+1}	q_0^{t+1}
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Zustand $q_i^t \rightarrow q_i^{t+1}$

Binärzähler

Beispiel: 3 Bit Vorwärts Binär-Zähler

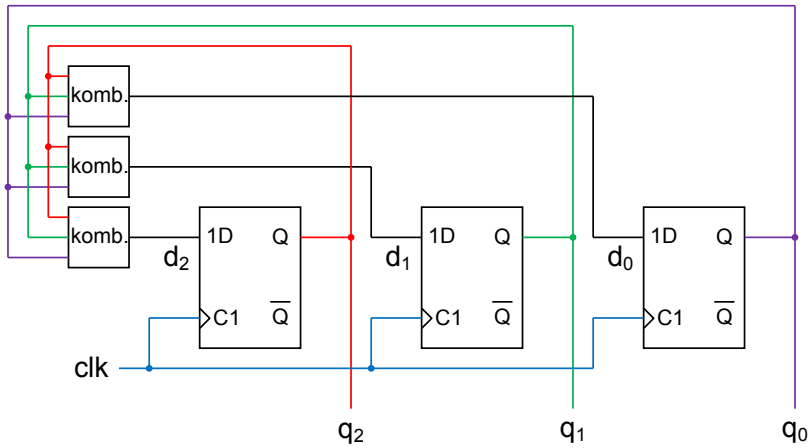
Durchläuft nacheinander die Zustände:
000 → 001 → 010 → 011 → 100 → 101 → 110 → 111 → 000

q_2^t	q_1^t	q_0^t	d_2^t	d_1^t	d_0^t
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Realisierung mit D-FlipFlops:

Wechsel der Zustände mit jeder positiven Taktflanke → neuer Wert q_i^{t+1} muss vorher an d_i-Eingang anliegen:

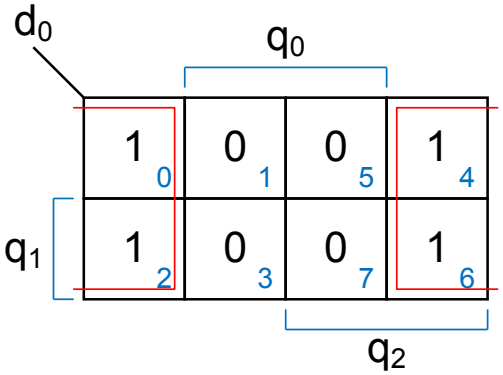
$$q_i^{t+1} = d_i^t = f(q_2^t, q_1^t, q_0^t)$$



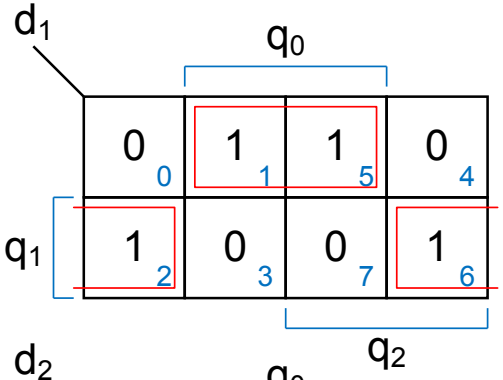
Binärzähler

Minimierung der Funktionen der d_i-Eingänge mit KV-Diagrammen

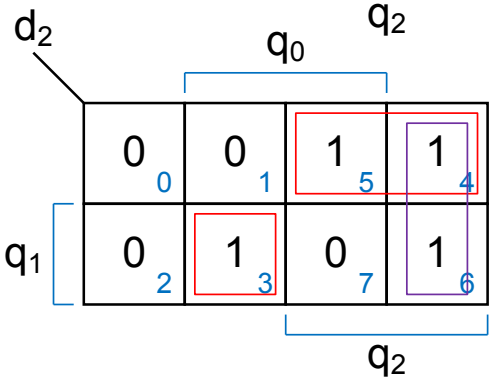
q_2^t	q_1^t	q_0^t	d_2^t	d_1^t	d_0^t
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0



$d_0 = q_0$ ‘



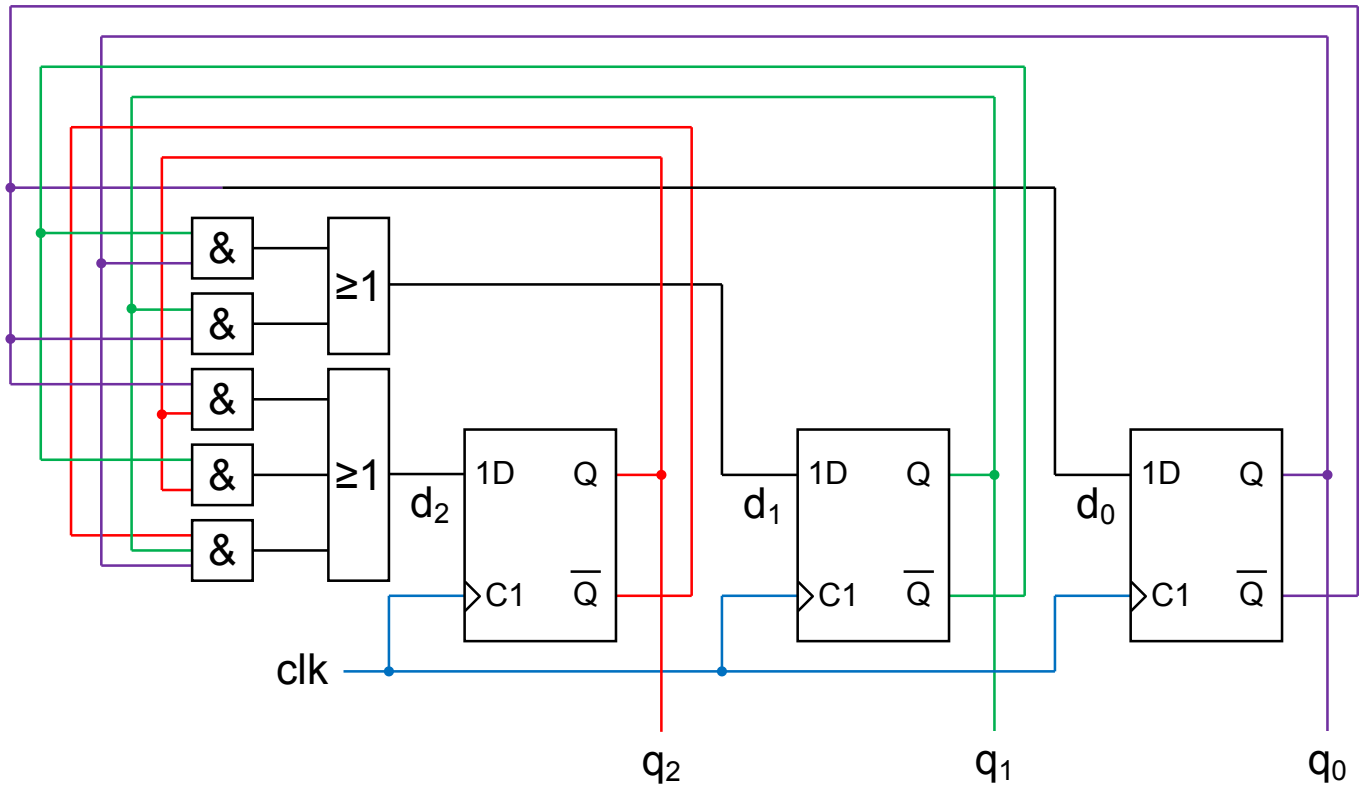
$d_1 = (q_0 * q_1$ ‘) + (q_0 ‘$* q_1$)



$d_2 = (q_0$ ‘$* q_2) + (q_1$ ‘$* q_2)$
$+ (q_0 * q_1 * q_2$ ‘)

Binärzähler

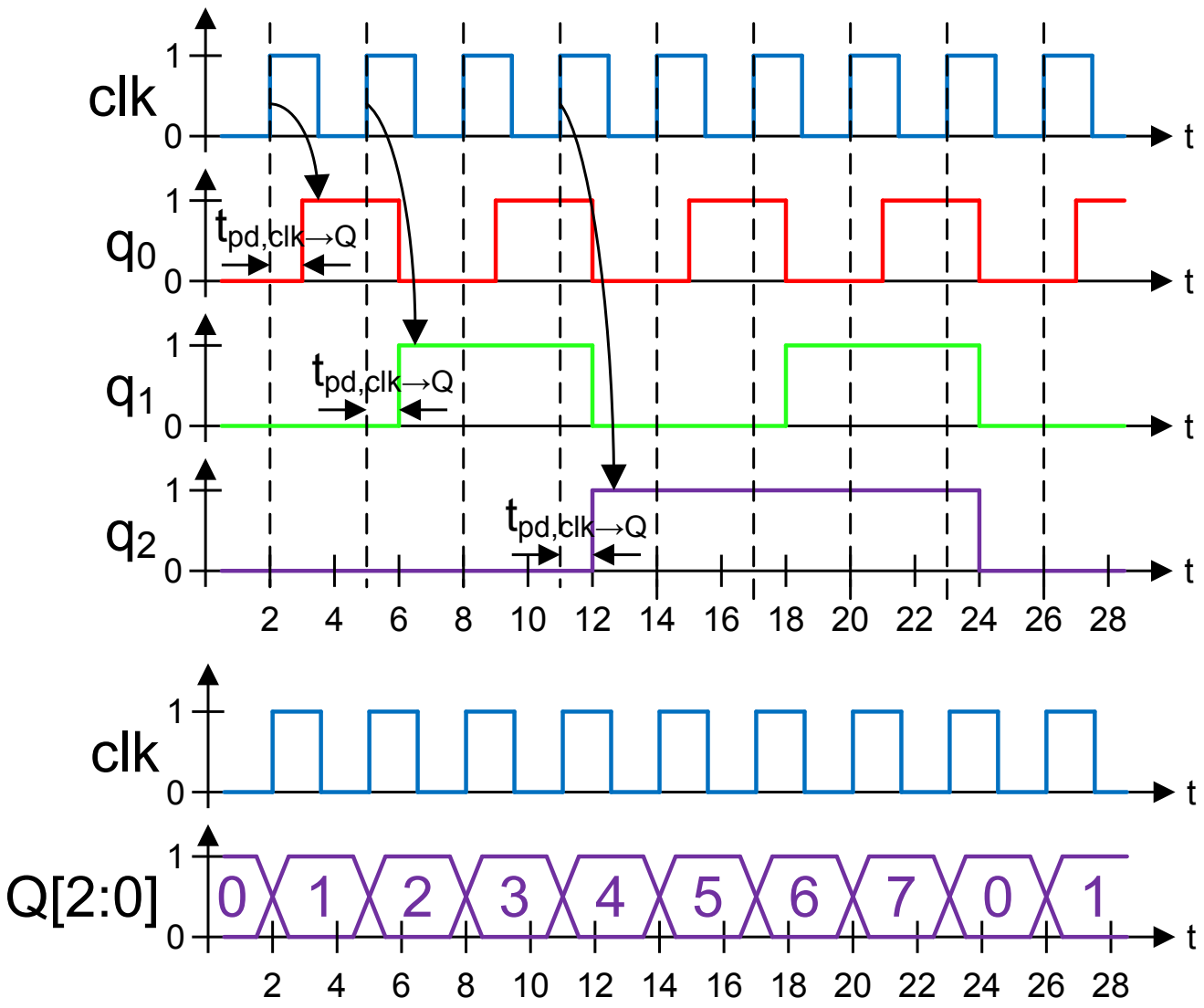
Realisierung:



Ist das Übergangsschaltnetz als zweistufiges Schaltnetz realisiert, so ist die Verzögerung in erster Näherung konstant und die Zählfrequenz somit unabhängig von der Bitbreite n.

Schaltungstiefe	Flächenbedarf
O(1)	$O(n^2)$

Binärzähler

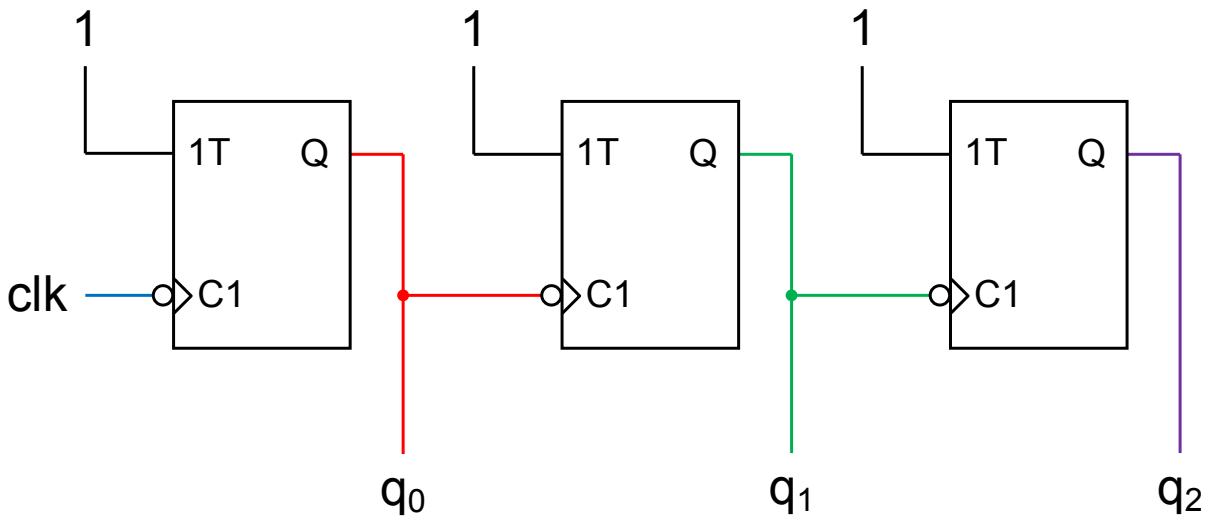


Alle FlipFlops ändern ihren Wert synchron zum gemeinsamen Taktsignal!

Alternative Darstellung im Zeitdiagramm!

Asynchroner Binärzähler

Binärzähler kann auch als asynchroner Zähler mit T-FlipFlops aufgebaut werden. Die Asynchronität kommt daher, da die Speicherelemente nicht mehr gleichzeitig schalten, sondern nacheinander.

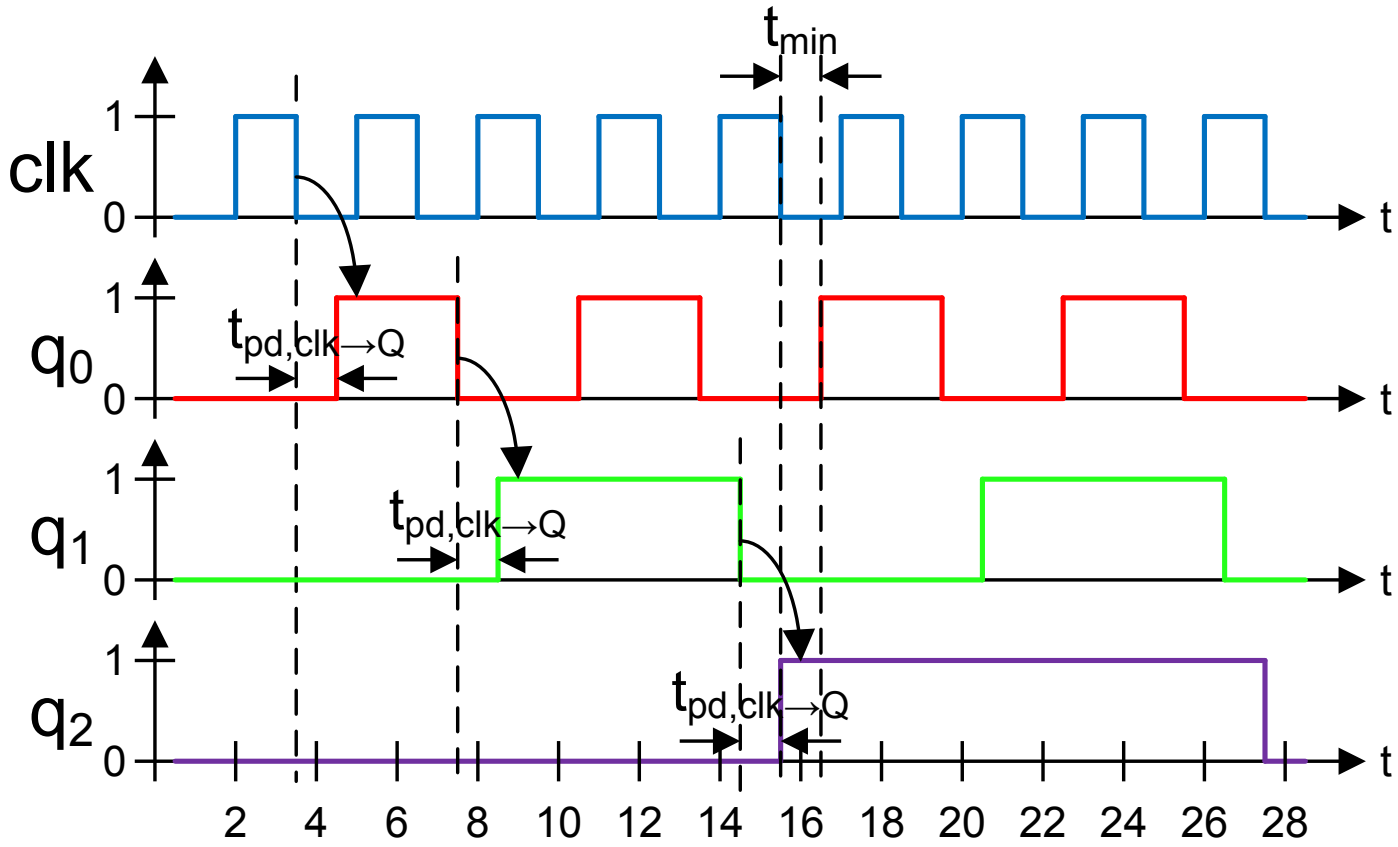


Der Vorteil des asynchronen Binärzählers ist der geringe Flächenbedarf, da für jedes dazukommende Bit nur ein zusätzliches T-FlipFlop benötigt wird.

Schaltungstiefe	Flächenbedarf
O(n)	O(n)

Asynchroner Binärzähler

Das verzögerte Schalten der einzelnen Stufen begrenzt die maximale Zählfrequenz.



Modulo-Zähler

Viele Codes benutzen nicht alle möglichen 2^n Codewörter von n-Bits, sondern eine Teilmenge davon. Entsprechend möchte man nicht alle Codewörter von 0 – 2^n-1 durchlaufen, sondern nur bis zu einer Obergrenze (N-1). Man spricht dann von einem sogenanten Modulo-N-Zähler.

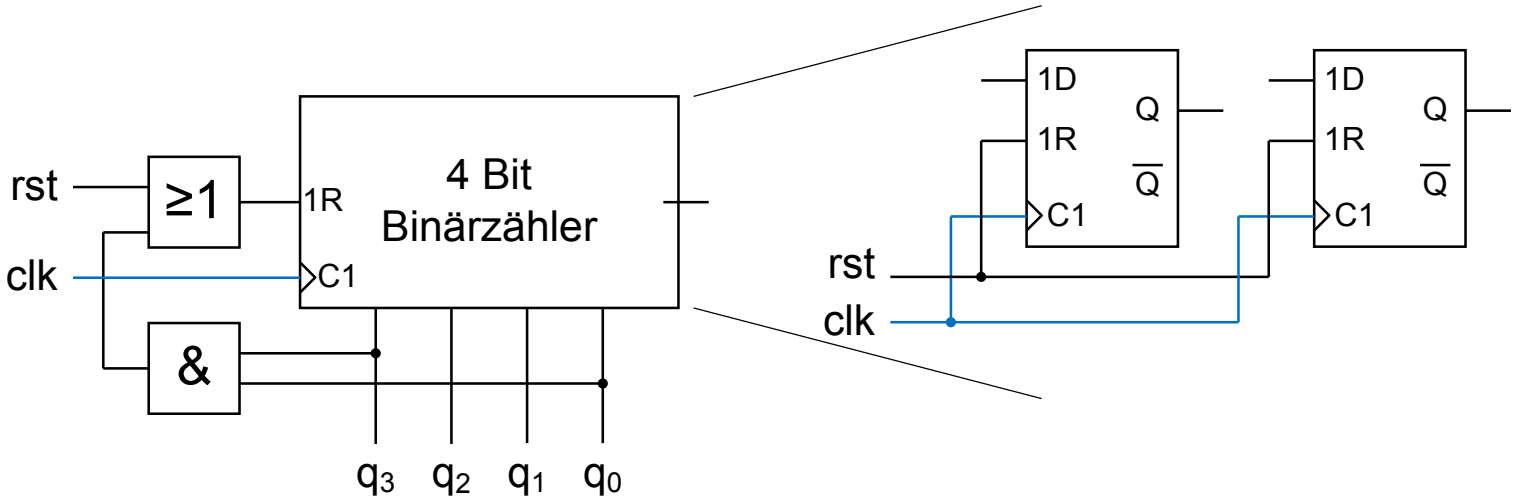
Beispiel: Der 4 Bit BCD-Code benutzt von den 16 möglichen Codewörter 10 Stück (0000 – 1001). Ein BCD-Zähler soll daher von 0000 bis 1001 zählen und dann wieder bei 0000 anfangen. Dies entspricht einem Modulo-10-Zähler.

Zur Realisierung gibt es zwei Möglichkeiten:

- Benutzung eines n-Bit Zählers und Verwendung von synchronem Set oder Reset-Eingang
- Entwurf eines speziellen Modulo-N-Zählers

Modulo-Zähler

Beispiel: Verwendung eines 4-Bit Zählers zur Realisierung eines Modulo-10-Zählers



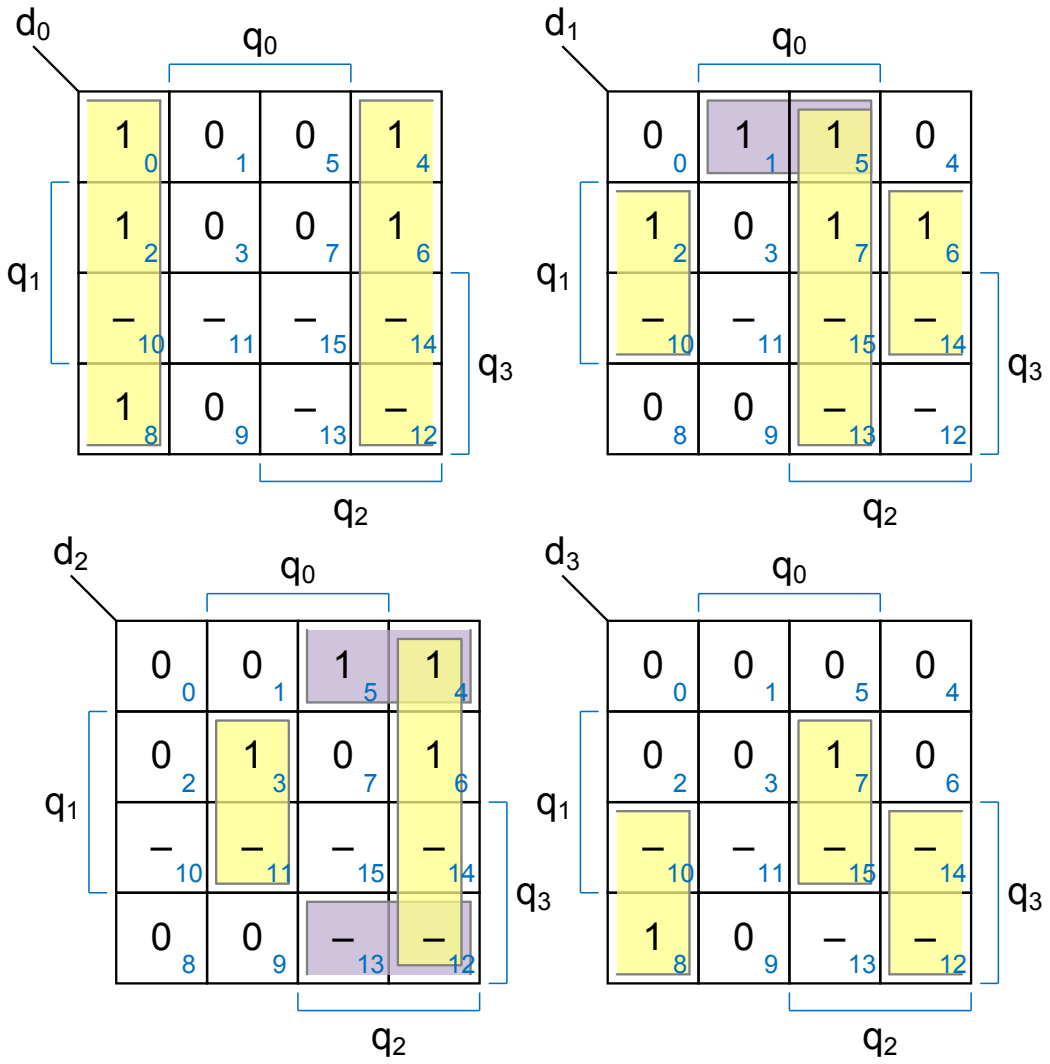
Kommt der Zähler auf den Wert 9 (1001), dann wird ein synchroner Reset des Zählers ausgelöst.

BCD-Zähler

Entwurf eines BCD-Zählers

q_3^t	q_2^t	q_1^t	q_0^t	q_3^{t+1}	q_2^{t+1}	q_1^{t+1}	q_0^{t+1}
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	–	–	–	–
1	0	1	1	–	–	–	–
1	1	0	0	–	–	–	–
1	1	0	1	–	–	–	–
1	1	1	0	–	–	–	–
1	1	1	1	–	–	–	–

BCD-Zähler



Realisierung des BCD-Zählers mit D-FlipFlops:

$d_0 = q_0$ ‘

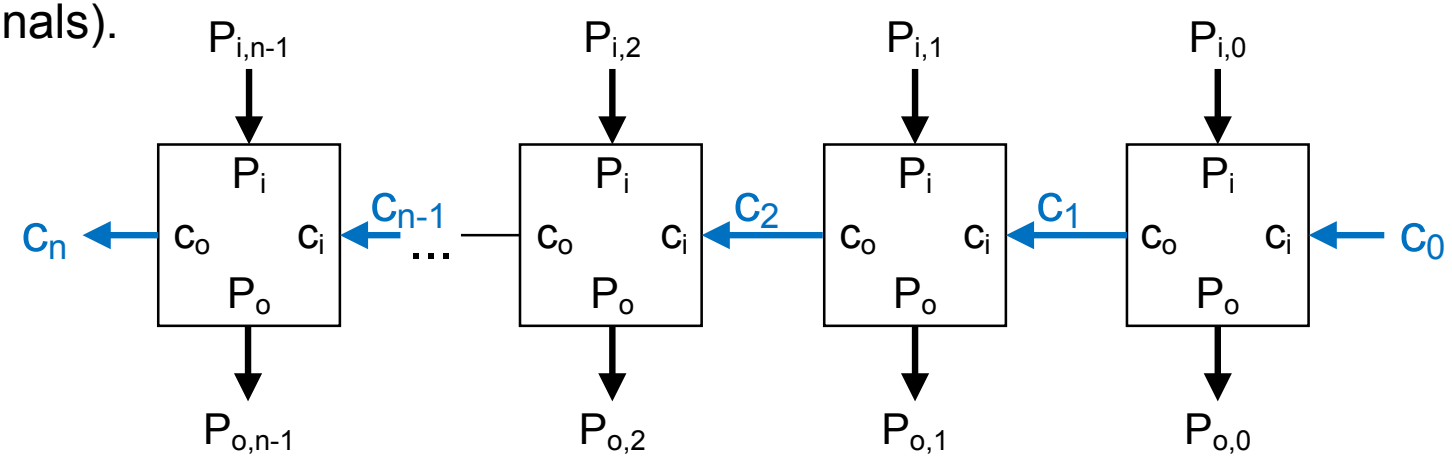
$d_1 = (q_0 * q_1' * q_3') + (q_0' * q_1) + (q_0 * q_2)$

$d_2 = (q_0 * q_1 * q_2') + (q_1' * q_2) + (q_0' * q_2)$

$d_3 = (q_0' * q_3) + (q_0 * q_1 * q_2)$

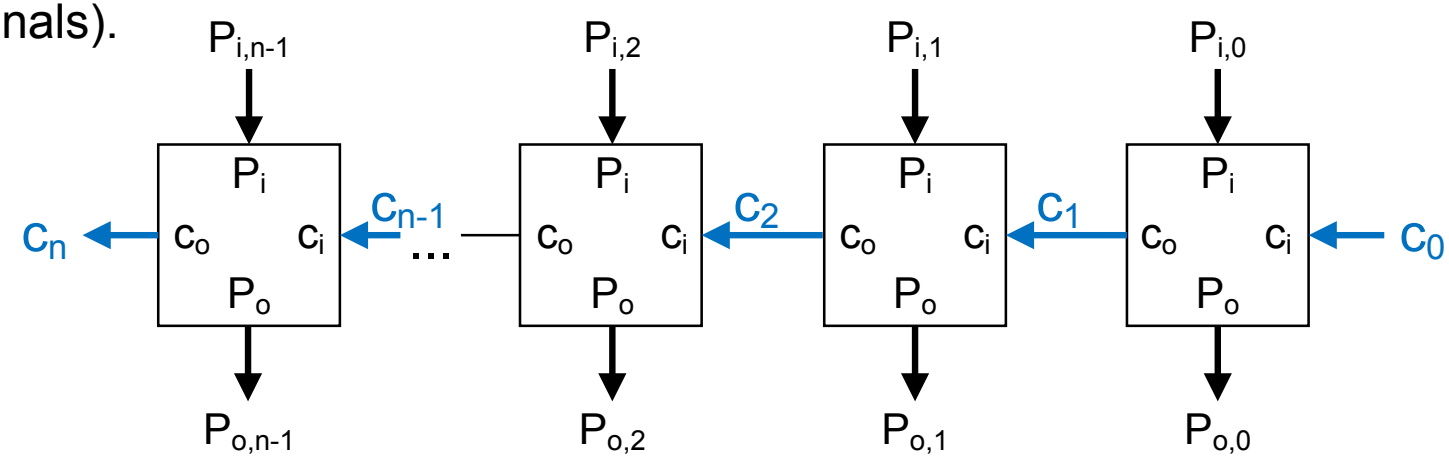
iterative Schaltung

Bei einer iterativen Schaltung werden zur Realisierung der Schaltung n gleiche Schaltungsblöcke verwendet. Jeder Block hat Eingangs- $\mathbf{P_i}$ und Ausgangssignale $\mathbf{P_o}$ (primary inputs, outputs). Die einzelnen Blöcke arbeiten alle parallel, wobei es Signale **c** gibt, welche durch die einzelnen Stufen propagieren müssen (cascading signals).

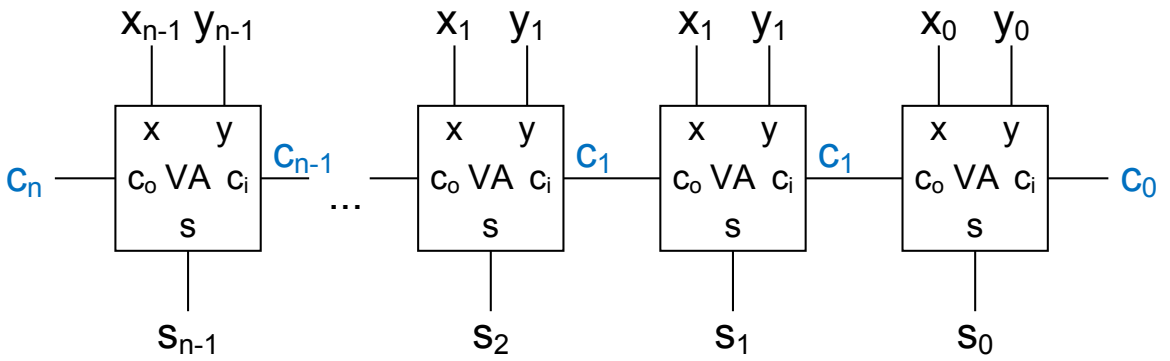


iterative Schaltung

Bei einer iterativen Schaltung werden zur Realisierung der Schaltung n gleiche Schaltungsblöcke verwendet. Jeder Block hat Eingangs- $\mathbf{P_i}$ und Ausgangssignale $\mathbf{P_o}$ (primary inputs, outputs). Die einzelnen Blöcke arbeiten alle parallel, wobei es Signale **c** gibt, welche durch die einzelnen Stufen propagieren müssen (cascading signals).

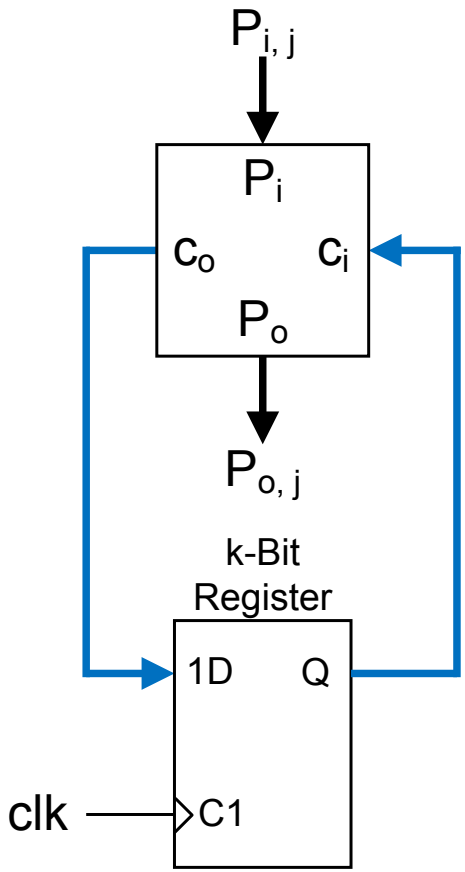


Beispiel:
In Kapitel 4 betrachtete Carry-Ripple-Addierer



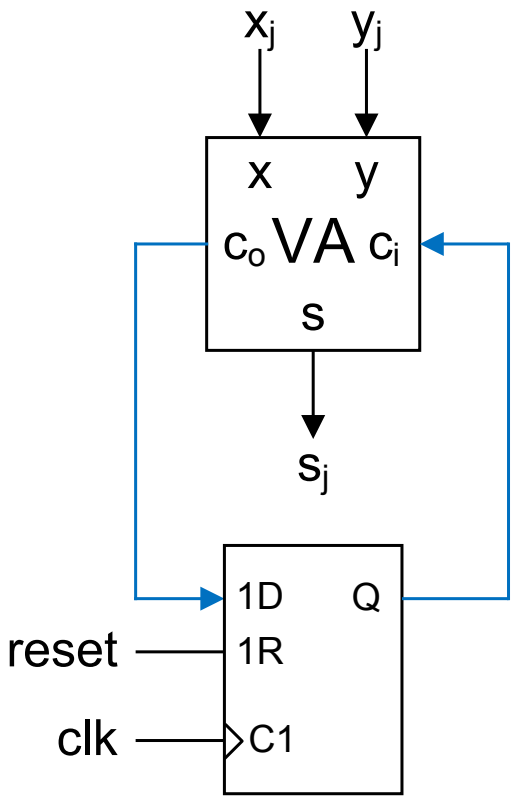
sequentielle Schaltung

Die Alternative zu einer iterativen Schaltungen ist die sequentielle Schaltung. Statt n Schaltungsblöcke parallel zu verwenden kann auch ein Block zeitlich nacheinander (sequentiell) benutzt werden. Die Eingangssignale werden nacheinander angelegt und die Ausgangssignale können nacheinander abgegriffen werden. Register speichern die cascading signals für die nächste Verarbeitung zwischen. Für die vollständige Verarbeitung werden dann n-Takte benötigt.



serieller Addierer

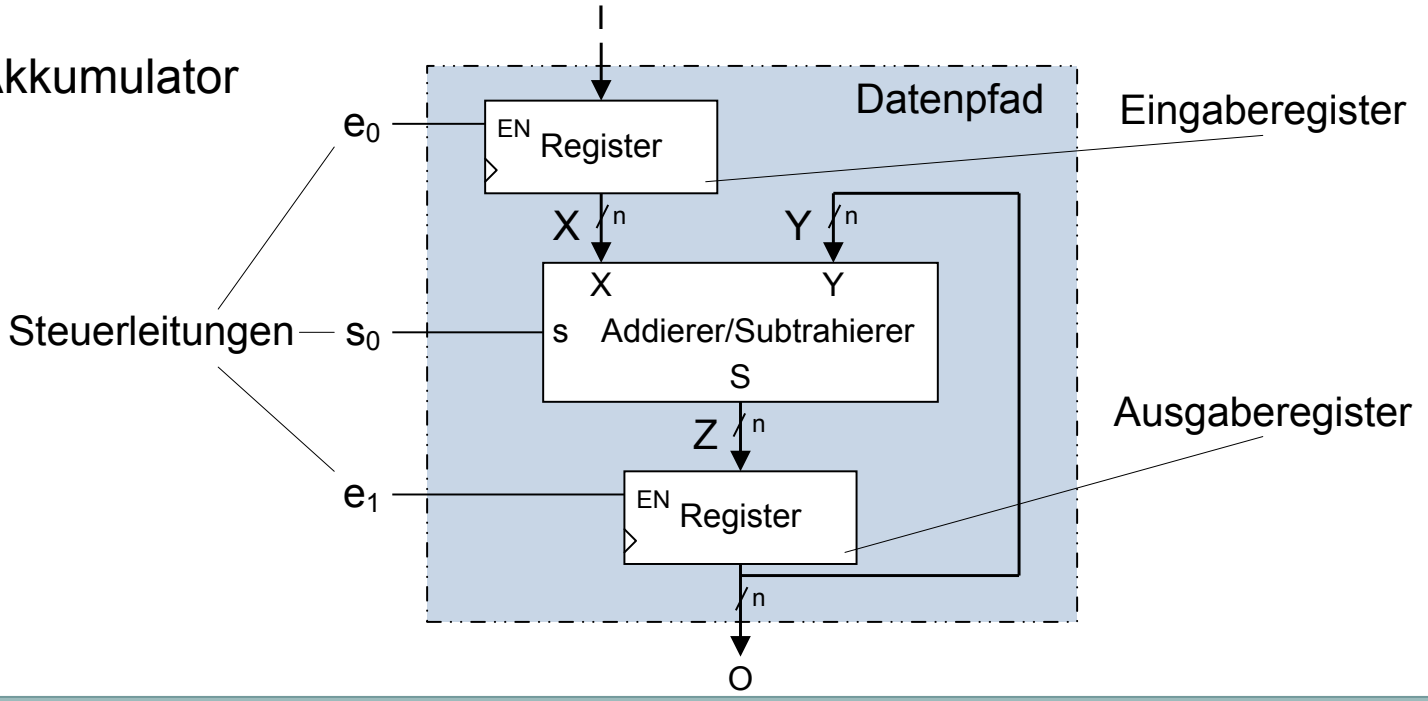
Beispiel:



Register-Transfer-Ebene

Die Register-Transfer-Ebene (engl. Register Transfer Level, RTL) stellt eine höhere Entwurfssicht auf digitale Schaltungen dar. Dabei werden nicht mehr die Schaltungsdetails (Gatter, FlipFlops), sondern vollständige Schaltungskomponenten und deren Kommunikation betrachtet.
Der Hauptaugenmerk liegt auf den Signalpfaden von Speicherelementen (Register) über funktionale Einheiten (z.B. ALU) bis zu anderen Speicherelementen, womit ein Datenpfad beschrieben wird.

Beispiel: Akkumulator



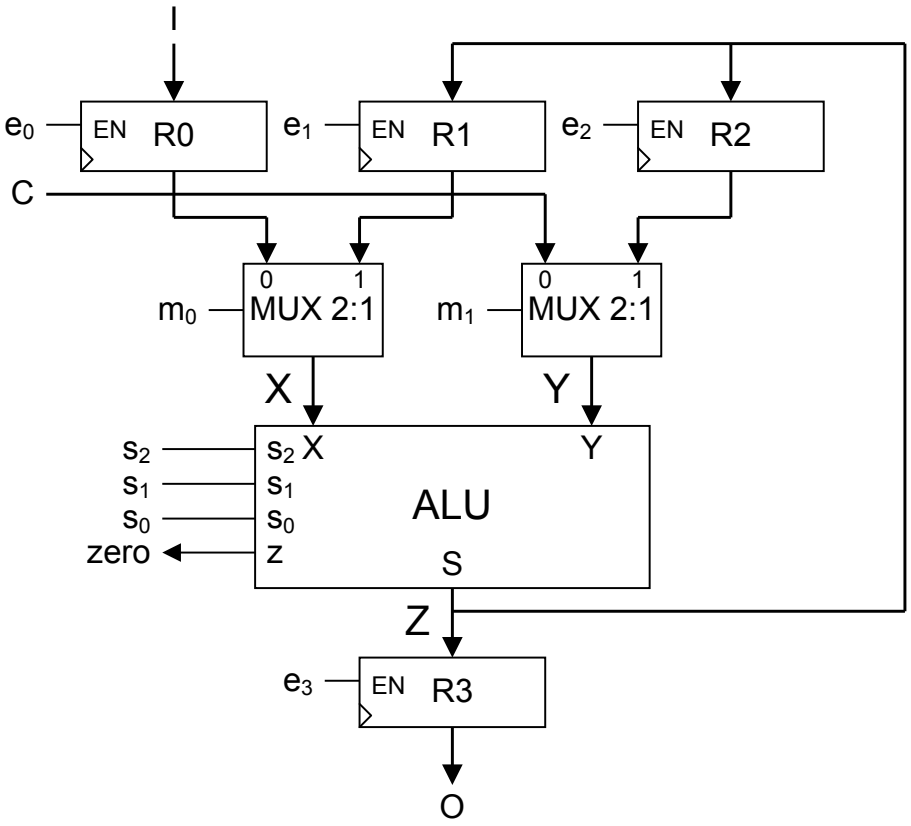
Register-Transfer-Ebene

Die Beschreibung auf der Register-Transfer-Ebene erfolgt mit RTL-Anweisungen. Dabei werden Datentransfers von der Quelle (-Register) zum Ziel (-Register) beschrieben.

Symbol	Beschreibung	Beispiel
Buchstaben und Ziffern	Register	AR, R0
runde Klammern	Teile eines Registers	R1(2), R2(2:4)
Pfeil (←)	Datentransfer	R1 ← R2
Komma (,)	gleichzeitige Transfers	R1←R2, R3←R4
2 senkrechte Striche (||)	Verknüpfung	R1(4:2) || R2(1:0)
Plus (+)	Addition	R2 + R3

Ein Semikolon steht zwischen Transfers, die nacheinander ausgeführt werden sollen (sequentiell).
Ein Komma steht zwischen Transfers, die gleichzeitig ausgeführt werden sollen (parallel).

Datenpfad



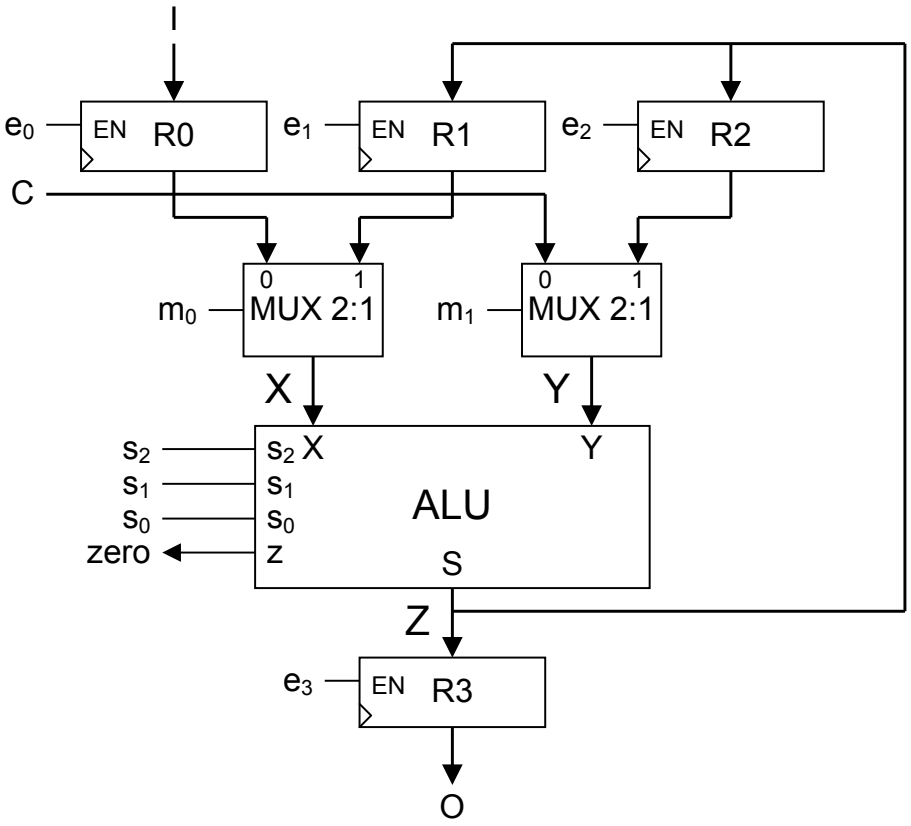
ALU Funktionen:

s_2	s_1	s_0	Funktion
–	0	0	Z := X XOR Y
–	0	1	Z := X OR Y
–	1	0	Z := X AND Y
0	1	1	Z := X + Y
1	1	1	Z := X - Y

Ansteuerung des Datenpfads:

e_3	e_2	e_1	e_0	m_1	m_0	s_2	s_1	s_0	Transfer

Datenpfad



ALU Funktionen:

s_2	s_1	s_0	Funktion
–	0	0	Z := X XOR Y
–	0	1	Z := X OR Y
–	1	0	Z := X AND Y
0	1	1	Z := X + Y
1	1	1	Z := X - Y

Ansteuerung des Datenpfads:

e_3	e_2	e_1	e_0	m_1	m_0	s_2	s_1	s_0	Transfer
0	0	1	0	1	0	0	1	1	R1←R0 + R2
0	1	0	0	1	0	–	0	1	R2←R0 or R2
0	0	1	1	0	0	1	1	1	R1←R0 - C, R0←I
1	0	0	0	1	1	–	0	0	R3←R1 xor R2

Kontrollfragen

- Bei Speicherelementen wird in mehrerlei Weise zwischen synchron und asynchron unterschieden. Was ist jeweils synchron und asynchron?
- Wie unterscheiden sich D-, T-, RS- und JK-Flipflops?
- Welche besonderen Zeitbedingungen sind bei Flipflops einzuhalten?
- Wie wirken sich diese auf die maximal mögliche Taktfrequenz aus?
- Was unterscheidet ein Register von einem Flipflop?
- Welche Arten von Registern (mit zusätzlichen Funktionalitäten) gibt es?
- Wie würden Sie einen Zähler für den One-Hot-Kode aufbauen? Welche Vor-/Nachteile hätte solch ein Zähler gegenüber Zählern für die in diesem Kapitel behandelten Kodierungen?
- Worin ähneln und unterscheiden sich iterative und sequentielle Schaltungen?
- Nennen Sie die Grundelemente der Syntax der RTL-Anweisungen.

ANHANG

D-FlipFlop

```
entity flipflop is
port(clk   : in   bit;
     rst   : in   bit;
     d     : in   bit;
     q     : out  bit);
end entity flipflop;

architecture behaviour of flipflop is
begin
     process(clk, rst)
     begin
          if (rst = '1') then
               q <= '0';
          elsif (clk'event and clk = '1') then
               q <= d;
          end if;
     end process;
end architecture behaviour;
```

n-Bit Register

```
entity nregister is
generic( n     : integer := 4);
port(clk, rst  : in  bit;
     D         : in  bit_vector(n-1 downto 0);
     Q         : out bit_vector(n-1 downto 0));
end entity nregister;

architecture behaviour of nregister is
begin
     process(clk)
     begin
          if (clk'event and clk = '1') then
               if (rst = '1') then
                    Q <= (others => '0');
               else
                    Q <= D;
               end if;
          end if;
     end process;
end architecture behaviour;
```

Binärzähler

```
entity counter is
generic(  N     : integer := 10);
port( clk, rst : in    std_logic;
      Q        : out   std_logic_vector(3 downto 0));
end entity counter;

architecture behaviour of counter is
signal count  : integer range 0 to N-1;
begin
      Q <= conv_std_logic_vector(count, 4);
      process(clk, rst)
      begin
          if (rst = '1') then
              count <= 0;
          elsif rising_edge(clk) then
              if (count = N-1) then
                  count <= 0;
              else
                  count <= count + 1;
              end if;
          end if;
      end process;
end architecture behaviour;
```