Grundlagen der Digitaltechnik Foliensatz 4: Rechenwerke und Operationsabläufe

G. Kemnitz

Institut für Informatik, TU Clausthal (EDS_F4)

28. Juni 2023

Inhalt F4: Rechenwerke und Operationsabläufe

Rechenwerke
1.1 Addierer
1.2 Subtrahierer, Zähler etc.
1.3 Multiplizierer
1.4 Komparatoren
1.5 Block-Shifter
Automaten
2.1 Entwurf mit KV-Diagrammen
2.2 Beschreibung in VHDL
2.3 Redundante Zustände
2.4 Spezifikation und Entwurf
Operationsabläufe
3.1 Serielle Schnittstelle
3.2 Serieller Addierer
3.3 Dividierer

Die Zielfunktionen für Schaltungen aus Tausenden bis Millionen von Gattern werden als Programme beschrieben, simuliert und synthetisiert.

Programme zur Beschreibung digitaler Schaltungen nutzen neben logischen Ausdrücken und Fallunterscheidungen auch arithmetische Ausdrücke.

Arithmetische Ausdrücke werden durch Rechenwerke und taktgesteuerte Berechnungsabläufe nachgebildet.

Auch andere komplexe Verarbeitungsfunktionen werden sequentiell durch taktgesteuerte Abläufe nachgebildet.

Rechenwerke

Rechenwerke

Rechenwerke sind Schaltungen ohne Zwischenspeicher zur Ausführung arithmetischer (und bitscheibenorientierter Logik-) Operationen. Gebräuchliche Rechenwerke:

- Addierer,
- Subtrahierer (Addierer mit negiertem Subtrahenden),
- Multiplizierer,
- Vergleicher für gleich, größer, größer gleich, ...
- Shifter (Potenzierung mit Zweierpotenz),
- mehrfunktionale Einheiten, z.B. die ALU (**A**rithmetic **L**ogic **U**nit) eines Prozessors.

Kompliziertere arithmetische Funktionen (Division, Potenzierung, Winkelfunktionen, ...) werden in Schritten, gesteuert durch einen Automaten oder ein Programm, ausgeführt.

Beschreibung algorithmisch, parametriert. Schaltungsgeneratoren.

Addierer

Algorithmus der Addition

	0	1	1	0	0	1	0	1	$(-0 \cdot 2^8)$
+	1	0	0	0	0	0	1	1	$(-1 \cdot 2^8)$
	1	1	1	0	1 $^{(1)}$	0 $^{(1)}$	0 $^{(1)}$	0	$(-1 \cdot 2^8)$

Erweiterung zu vorzeichenbehafteten Zahlen

- Wiederhole für alle Stellen beginnend mit der niederwertigsten
  - Addition der Ziffern und des Übertrags der Bitstelle zuvor.

Für die Addition von drei Binärziffern $a_i + b_i + c_i$ gilt:

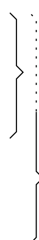
- Summe $s_i = 1$, wenn ein oder alle Summanden »1« sind, und
- Übertrag $c_{i+1} = 1$, wenn mindestens zwei der Summanden »1« sind

(a_i, b_i – Binärziffern; c_i – Übertrag).

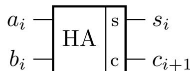
Voll- und Halbaddierer

Volladdierer addiert zwei Summanden- und ein Übertragsbit.
Halbaddierer addiert nur zwei Bit.

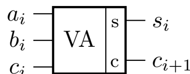
c_i	b_i	a_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



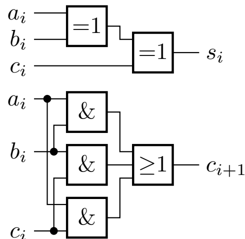
Halbaddierer



Volladdierer



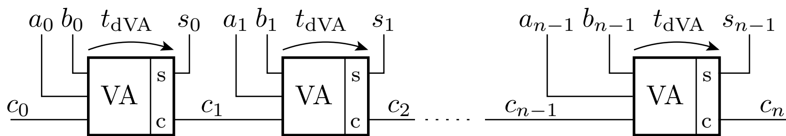
Schaltung Volladdierer



	Halbaddierer	Volladdierer
Summe s_i	$a_i \oplus b_i$	$a_i \oplus b_i \oplus c_i$
Übertrag c_{i+1}	$a_i b_i$	$a_i b_i \vee a_i c_i \vee b_i c_i$

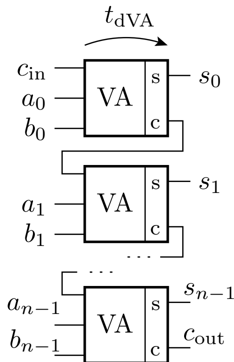
Ripple-Addierer

Der Ripple-Addierer ist der einfachste Addierer. Er besteht aus n über die Übertragssignale verketteten Volladdierern.



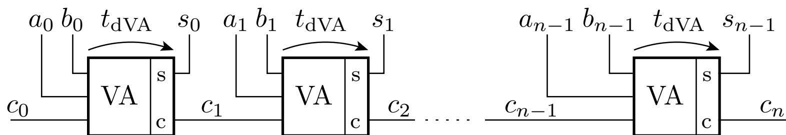
Beschreibung durch eine Generierungsschleife

```
signal a,b,s:std_logic_vector(n-1 downto 0);
signal cin, cout: std_logic;
...
process(a, b, cin)
 variable c: std_logic;
begin
 c:=cin;
 for i in 0 to n-1 loop
  s(i) <= a(i) xor b(i) xor c;
  c := (a(i) and b(i)) or(a(i) and c)
    or (b(i) and c);
 end loop;
 cout<=c;
end process;
```



c muss eine Variable sein, damit im Folgeschleifendurchlauf der zuletzt berechnete Wert weiterverarbeitet wird.

Verzögerungszeit eines Ripple-Addierers



- Die maximale Verzögerung ist die vom Übertragseingang c_0 bis zum Übertragsausgang c_n und beträgt

$$t_{\mathrm{dAdd}} = n \cdot t_{\mathrm{dVA}}$$

  (t_{dVA} – Verzögerung eines einzelnen Volladdierers).
- Verringerbar durch Minimierung der Verzögerung von c_{i-1} bis c_i für alle Volladdierer.

Schneller Übertragsdurchlauf

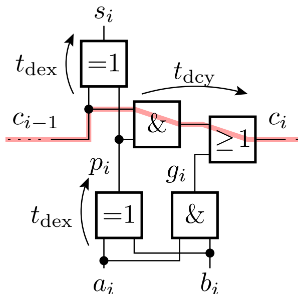
Zur Minimierung der Übertragsverzögerung werden bitweise zwei Hilfssignale gebildet:

- Übertrag weiterleiten (propagate):

$$p_i = a_i \oplus b_i$$

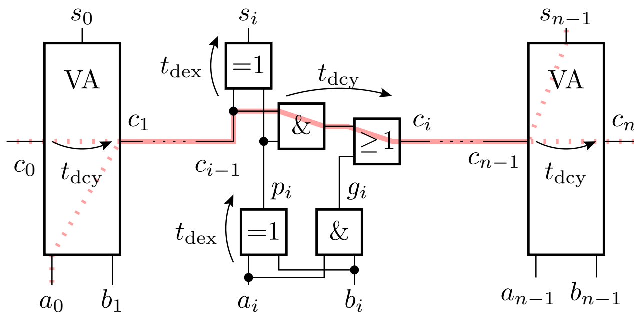
- Übertrag erzeugen (generate):

$$g_i = a_i b_i$$



Mit diesen Hilfssignalen vereinfacht sich der zeitkritische Berechnungsteil zu:

$$c_i = p_i c_{i-1} \vee g_i$$



Der Aufwand ist etwa derselbe wie für den Ripple-Addierer mit Volladdierern ohne schnellen Übertragsdurchlauf. Die Verzögerung ist nur etwa halb so groß.

In dem FPGA für die Übungen gibt es spezielle Verbindungen und Gatter für die schnelle Übertragsweitergabe zwischen benachbarten Zellen. Dadurch ist die Addition im Vergleich zu anderen gleichlangen Gatterverkettungen sehr schnell.

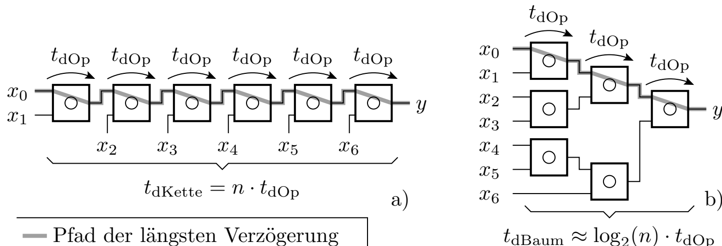
Hierarchische Addierer

In einem hierarchischen Addierer wird die Übertragsberechnung

- in eine assoziative Operation überführt und
- die Kettenstruktur durch eine Baumstruktur ersetzt:

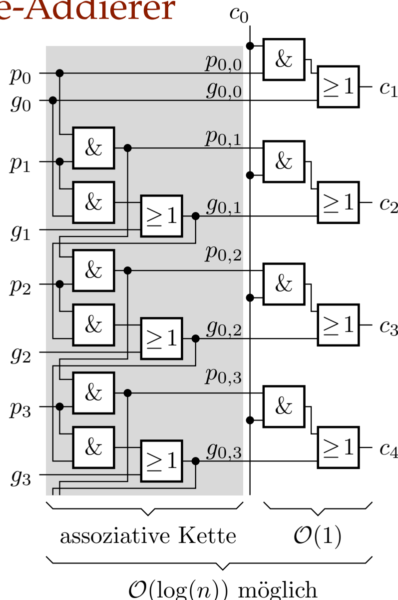
$$x_6 \circ \ldots \circ x_1 \circ x_0 = ((x_1 \circ x_0) \circ (x_3 \circ x_2)) \circ ((x_4 \circ x_5) \circ x_6)$$

Bei einem Baum nimmt die Verzögerung nur noch mit dem Logarithmus statt linear der Anzahl der Operationen zu.



Block-Generate-Propagate-Addierer

Aus den Generate- und Propagate-Signalen g_i und p_i der einzelnen Bitstellen wird für jede Bitstelle i ein Block-Generate-Signal $g_{0.i}$ und ein Block-Propagate-Signal $p_{0.i}$ gebildet. Diese beschreiben, ob die Summandenbits 0 bis i zusammen für Bit i einen Übertrag generieren oder den Eingangsübertrag c_0 bis Bit i weiterleiten. Die Bildung der Block-Generate- und Block-Propagate-Signale ist assoziativ.



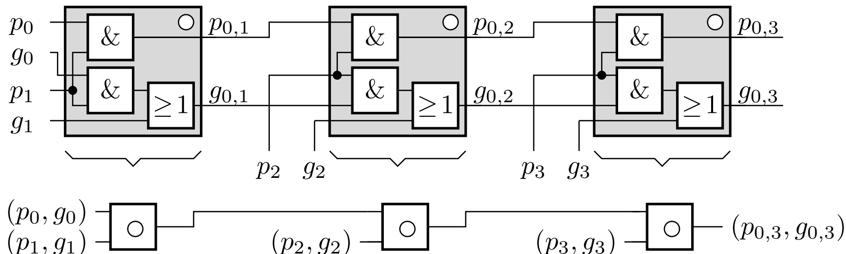
Die Kettenoperation zur Bildung der Block-Generate- und Block-Propagate-Signale für drei aufeinanderfolgende Bits:

$$(p_2, g_2) \circ [(p_1, g_1) \circ (p_0, g_0)] = (p_2 p_1 p_0,\ g_2 \vee g_1 p_2 \vee g_0 p_2 p_1)$$

führt bei Vertauschung der Berechnungsreihenfolgen auf dieselben logischen Operationen (Beweis der Assoziativität):

$$[(p_2, g_2) \circ (p_1, g_1)] \circ (p_0, g_0) = (p_2 p_1 p_0,\ g_2 \vee g_1 p_2 \vee g_0 p_2 p_1)$$

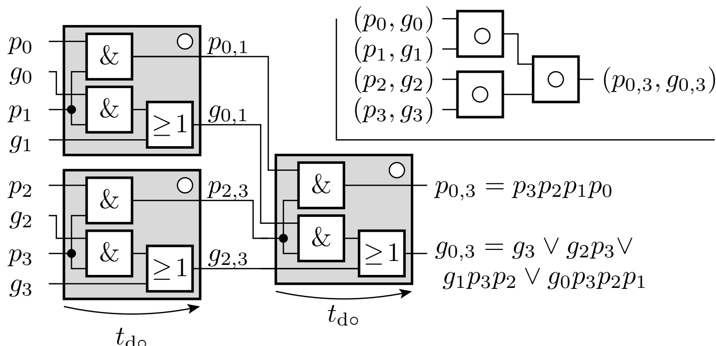
Für drei Bits besteht die Verarbeitungskette aus den Operationen:



Mit einer anderen Ausführungsreihenfolge (Klammersetzung)

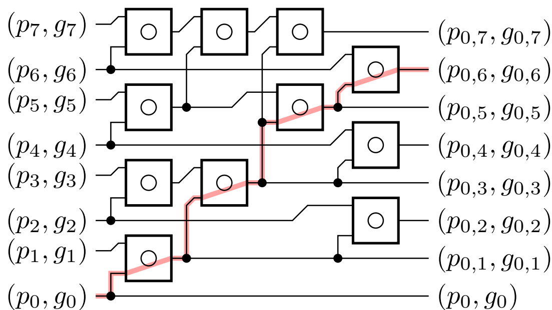
$$(p_3, g_3) \circ [(p_2, g_2) \circ [(p_1, g_1) \circ (p_0, g_0)]] = [(p_3, g_3) \circ (p_2, g_2)] \circ [(p_1, g_1) \circ (p_0, g_0)]$$

können die erste und die letzte Operation gleichzeitig erfolgen. Aus dem kettenartigen wird ein baumartiger Datenfluss:



($t_{d\circ}$ – max. Verzögerung der Operatorfunktion (entspricht der Verzögerung einer UND-ODER-Kette).

Erweiterung auf die Berechnung aller Block-Propagate- und Block-Generate-Signale für eine 8-Bit-Addition:



Die Verzögerungszeit wächst logarithmisch mit der Bitbreite n:

n	8	16	32	64
t_{dKette}	$\approx 8 \cdot t_{\mathrm{d\circ}}$	$\approx 16 \cdot t_{\mathrm{d\circ}}$	$\approx 32 \cdot t_{\mathrm{d\circ}}$	$\approx 64 \cdot t_{\mathrm{d\circ}}$
t_{dBaum}	$\approx 5 \cdot t_{\mathrm{d\circ}}$	$\approx 6 \cdot t_{\mathrm{d\circ}}$	$\approx 7 \cdot t_{\mathrm{d\circ}}$	$\approx 8 \cdot t_{\mathrm{d\circ}}$

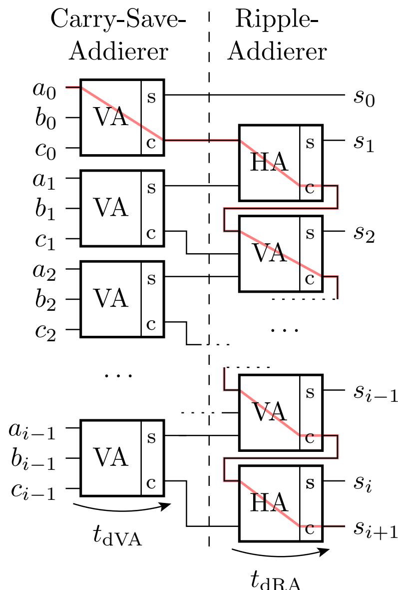
Preis: überproportional mit n wachsender Schaltungsaufwand.

Carry-Save-Addierer

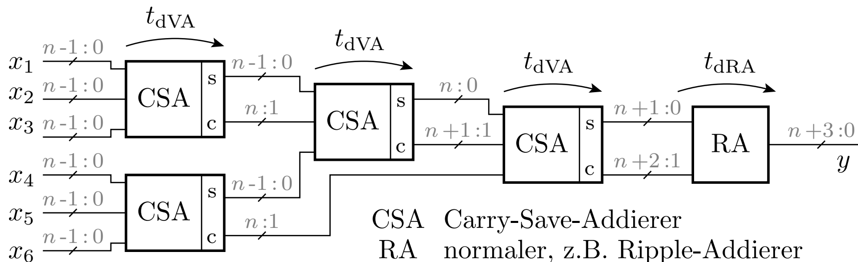
Ein Carry-Save-Addierer besteht je Bit aus einem Volladdierer, der drei Summandenbits zu einem Summen- und einem Übertragsbit zusammenfasst.

Abschließende Addition des Summen- und des Übertragsvektors mit einem normalen Addierer.

Die gesamte Additionsdauer für zwei Additionen ist nur um eine Volladdiererverzögerung länger als für eine Addition.



Für mehr als zwei Additionen erfolgt zuerst eine baumartige 3-zu-2-Reduktion und zum Abschluss wieder die Addition des Summen- und des Übertragsvektors mit einem normalen Addierer.



Beim Entwurf eines solchen Vektoraddierers ist zu beachten, dass die Bitanzahl der Zwischensummen mit der Summandenanzahl zunimmt, bei zwei Summanden um ein Bit, bei drei und vier Summanden um zwei Bit etc.

Zusammenfassung

- Ripple-Addierer: Kette von n Volladdierern. Zur Bitbreite n proportionale Verzögerungszeit.
- Schneller Übertragsdurchlauf: Halbierung der Additionsdauer ohne nennenswerten Zusatzaufwand.
- Hierarchische Addierer: Schnelle Addierer für große Bitbreiten. Nur logarithmisch mit der Bitbreite zunehmende Verzögerungszeit. Mehr Schaltungsaufwand.
- Carry-Save-Addierer: Addierer für mehrere Summanden, bei dem die Verzögerungszeit mit der Summandenanzahl nur logarithmisch zunimmt.

Subtrahierer, Zähler etc.

Subtraktion

Die Subtraktion ist die Addition mit dem negierten Summanden:

$$a - b = a + (-b)$$

Die Negation n-stelliger Zahlen ist »Stellenkomplement +1«:

$$-b = \bar{b} + 1$$

Zur Probe, die Summe aus einer Zahl und ihrem Stellenkomplement ist die größte darstellbare Zahl. Eins addiert im Zahlenkreis ergibt null. Beispiel:

$$\begin{aligned} 387 + \bar{3}\bar{8}\bar{7} + 1 &= 387 + 612 + 1 \\ &= 999 + 1 = (1^*)\,000 \end{aligned}$$

(1^* – nicht darstellbarer Übertrag). Das Stellenkomplement für Binärzahlen ist die bitweise Negation:

$$\bar{x} = \begin{cases} 1 & \text{für } x = 0 \\ 0 & \text{für } x = 1 \end{cases}$$

Subtraktionsbeispiel und Subtrahierer

bitweise Subtraktion

a		1	1	1	0	0	1	0	1
$-b$		1	0	1	0	1	0	1	0
c_i	(0)	(0)	(-1)	(-1)	(-1)	(0)	(-1)	(0)	(-1)
		0	0	1	1	1	0	1	0

(negierter) Eingangsübertrag

(negierter) Ausgangsübertrag

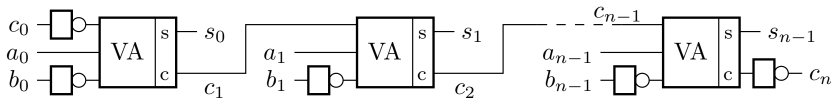
Addition des Stellenkomplements +1

a		1	1	1	0	0	1	0	1
$+\bar{b}$		0	1	0	1	0	1	0	1
									1
$\bar{c}_i$	(1)	(1)	(0)	(0)	(0)	(1)	(0)	(1)	(-1)
		0	0	1	1	1	0	1	0

Die Differenz »eins minus Eingangsübertrag« ist der negierte Eingangsübertrag:

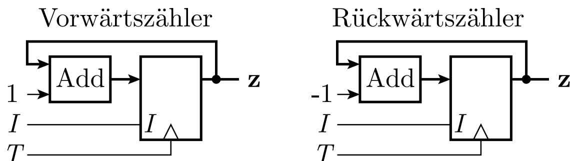
$$1 - c_0 = \bar{c}_0$$

Schaltung eines Subtrahierers:



Binärzähler

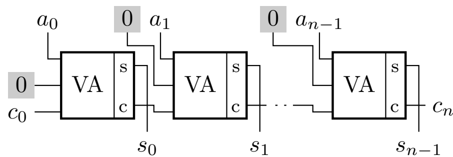
Binärzähler bestehen aus einem Register für den Zustand und einer Inkrement-Operation $(+1)$ oder der Dekrement-Operation (-1).



I Initialisierungssignal
T Takt
$\mathbf{z}$ Signal für den Zählstand

Inkrement-Rechenwerk

Das Rechenwerk für die Inkrement-Operation ist ein Addierer mit der Konstanten null und dem Eingabeübertrag.

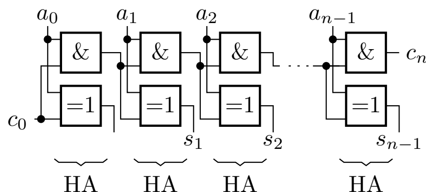


Konstanten-elimination

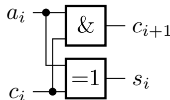
Halbaddierer

c_i	b_i	a_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
1	0	0	0	1
1	0	1	1	0

konstant



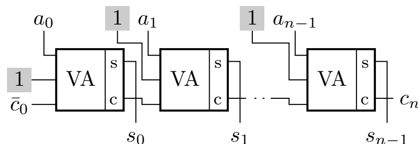
vereinfachte Schaltung



Ein Volladdierer mit einem konstanten Summanden null vereinfacht sich zu einem Halbaddierer.

Dekrement-Rechenwerk

Für die Dekrement-Operation werden die Zweierkomplementdarstellung von minus eins (11...1) und der negierte Eingabeübertrag $\bar{c}_0$ addiert. Das zweite Summandenbit der Volladdierer ist konstant eins. Die vereinfachte Schaltung ähnelt dem Halbaddierer für die Dekrement-Operation.

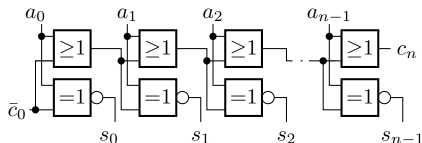


Konstanten-elimination

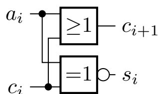
Wertetabelle eines Volladdierers mit $b_i = 1$

c_i	b_i	a_i	c_{i+1}	s_i
0	1	0	0	1
0	1	1	1	0
1	1	0	1	0
1	1	1	1	1

konstant

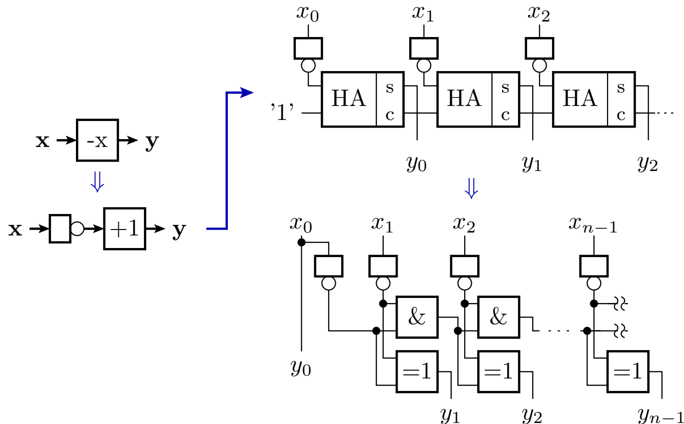


vereinfachte Schaltung



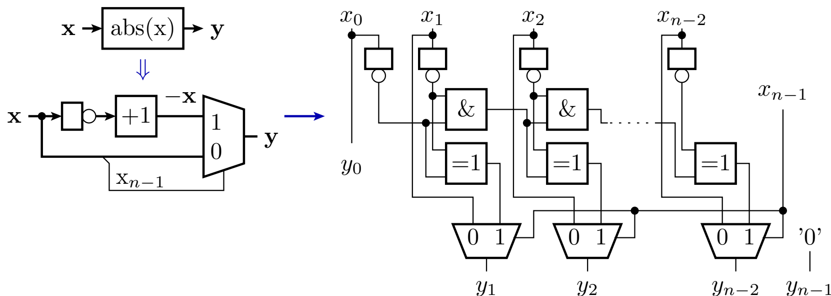
Rechenwerk für die Negation

Die Negation ist die Invertierung plus eins:



Rechenwerk zur Betragsbildung

Die Betragsbildung ist eine bedingte Negation:



Multiplizierer

Vorzeichenfreie Multiplikation

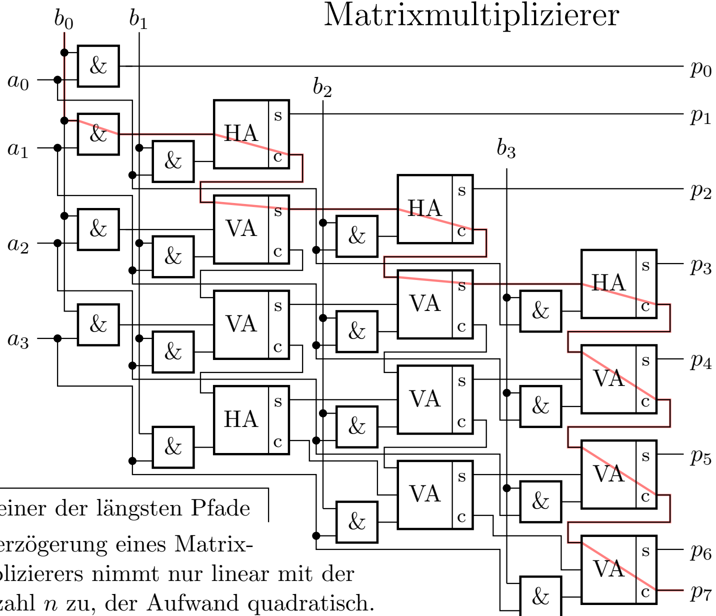
$$(a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0) \cdot (b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0) =$$

$$\begin{array}{r} a_3b_0 \cdot 2^3 + a_2b_0 \cdot 2^2 + a_1b_0 \cdot 2^1 + a_0b_0 \cdot 2^0 \\ a_3b_1 \cdot 2^4 + a_2b_1 \cdot 2^3 + a_1b_1 \cdot 2^2 + a_0b_1 \cdot 2^1 \qquad\qquad \\ a_3b_2 \cdot 2^5 + a_2b_2 \cdot 2^4 + a_1b_2 \cdot 2^3 + a_0b_2 \cdot 2^2 \qquad\qquad\qquad\qquad \\ a_3b_3 \cdot 2^6 + a_2b_3 \cdot 2^5 + a_1b_3 \cdot 2^4 + a_0b_3 \cdot 2^3 \qquad\qquad\qquad\qquad\qquad\qquad \\ \hline p_7 \; p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0 \end{array}$$

Die Multiplikation setzt sich zusammen aus

- 1-Bit-Multiplikationen (UND-Verknüpfungen).
- Zeilen- und spaltenweise Addition mit Halb- und Volladdierern.
- Anzahl der Produktbits gleich der Summe der Anzahl der Operandenbits, für das Beispiel:
  - Kleinstes Ergebnis: $0000_2 \cdot 0000_2 \mapsto 0000\,0000_2$
  - Größtes Ergebnis: $1111_2 \cdot 1111_2 \mapsto 1110\,0001_2$

Matrixmultiplizierer



Die Verzögerung eines Matrixmultiplizierers nimmt nur linear mit der Bitanzahl n zu, der Aufwand quadratisch.

Matrix-Multiplizierer für Vorzeichenzahlen

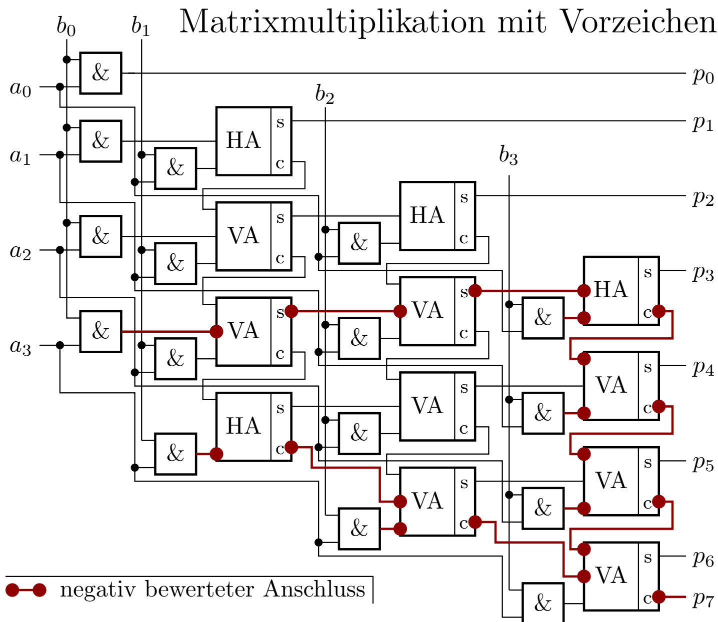
$$(-a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0) \cdot (-b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0) =$$

$$-a_0 b_3 \cdot 2^3 + a_0 b_2 \cdot 2^2 + a_0 b_1 \cdot 2^1 + a_0 b_0 \cdot 2^0$$
$$-a_1 b_3 \cdot 2^4 + a_1 b_2 \cdot 2^3 + a_1 b_1 \cdot 2^2 + a_1 b_0 \cdot 2^1$$
$$-a_2 b_3 \cdot 2^5 + a_2 b_2 \cdot 2^4 + a_2 b_1 \cdot 2^3 + a_2 b_0 \cdot 2^2$$
$$+a_3 b_3 \cdot 2^6 - a_3 b_2 \cdot 2^5 - a_3 b_1 \cdot 2^4 - a_3 b_0 \cdot 2^3$$

$$-p_7 \cdot 2^7 \quad +p_6 \cdot 2^6 \quad +p_5 \cdot 2^5 \quad +p_4 \cdot 2^4 \quad +p_3 \cdot 2^3 \quad +p_2 \cdot 2^2 \quad +p_1 \cdot 2^1 \quad +p_0 \cdot 2^0$$

Summanden			$a_i + b_i + c_i$			$a_i + b_i - c_i$			$a_i - b_i - c_i$			$-a_i - b_i - c_i$		
c_i	b_i	a_i	Wert	c_{i+1}	s_i	Wert	c_{i+1}	$-s_i$	Wert	$-c_{i+1}$	s_i	Wert	$-c_{i+1}$	$-s_i$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	1	1	1	0	1	-1	0	1
0	1	0	1	0	1	1	1	1	-1	1	1	-1	0	1
0	1	1	2	1	0	2	1	0	0	0	0	-2	1	0
1	0	0	1	0	1	-1	0	1	-1	1	1	-1	0	1
1	0	1	2	1	0	0	0	0	0	0	0	-2	1	0
1	1	0	2	1	0	0	0	0	-2	1	0	-2	1	0
1	1	1	3	1	1	1	1	1	-1	1	1	-3	1	1

Schaltung wie bisheriger Matrixmultiplizierer, nur mit zum Teil negativ bewerteten Zwischenergebnissen.

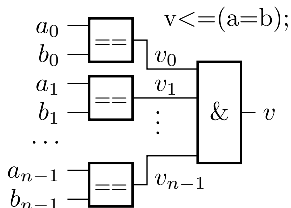
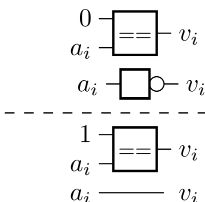
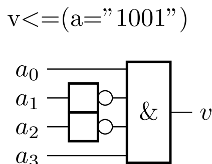


Matrixmultiplikation mit Vorzeichen
b_0
b_1
b_2
b_3
a_0
a_1
a_2
a_3
p_0
p_1
p_2
p_3
p_4
p_5
p_6
p_7
&
HA
VA
s
c
negativ bewerteter Anschluss

Komparatoren

Test auf Gleichheit oder Ungleichheit

```
signal a, b: std_logic_vector(n-1 downto 0);
signal v: std_logic;
```

Schaltung für einen bitweisen Vergleich	Regeln zur Konstanten-eliminierung	Vergleich mit einer Konstanten
		

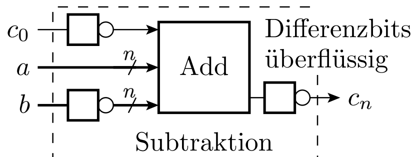
Test auf größer, größer oder gleich etc.

Ein Größenvergleich ist eine Subtraktion unter Auswertung des Vorzeichenbits:

- $a > b \Rightarrow a - b - 1 \geq 0$
- $a \geq b \Rightarrow a - b \geq 0$
- $a \leq b \Rightarrow a - b - 1 < 0$
- $a < b \Rightarrow a - b < 0$

Für vorzeichenfreie Zahlen:

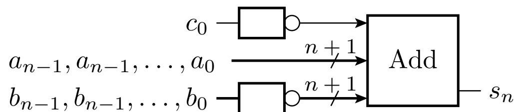
- nicht negative Differenz: kein Übertrag
- negative Differenz: Übertrag »eins borgen«



	$c_n = 0$	$c_n = 1$
$c_0 = 0$	$a \geq b$	$a < b$
$c_0 = 1$	$a > b$	$a \leq b$

Für vorzeichenbehaftete Zahlen

Differenzbildung verdoppelt den Ergebniswertebereich.
Vorzeichenbehaftete Erweiterung der Operanden um ein Bit und Auswertung des Ergebnisvorzeichenbits:



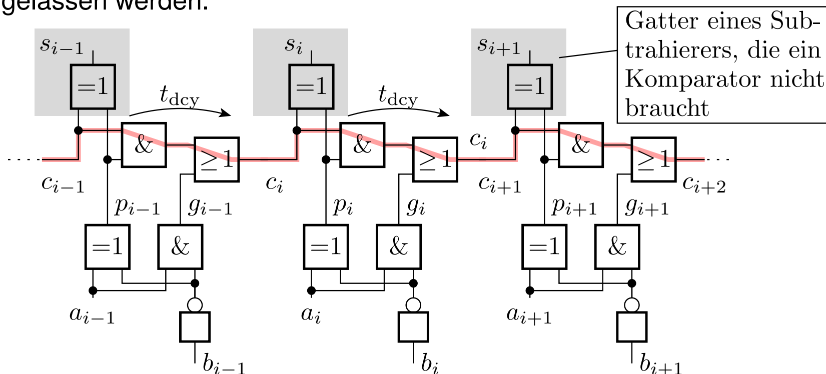
Summenbits s_0 bis s_{n-1} und Übertragsbit c_{n+1} nicht benötigt

s_n	c_0	
0	0	$a \geq b$
0	1	$a > b$
1	0	$a < b$
1	1	$a \leq b$

---

(0)011	(3)	(1)101	(-3)	(0)011	(3)
+(1)101	(-3)	+(1)101	(-3)	+(0)011	(3)
(0)000	(0)	(1)010	(-6)	(0)110	(6)

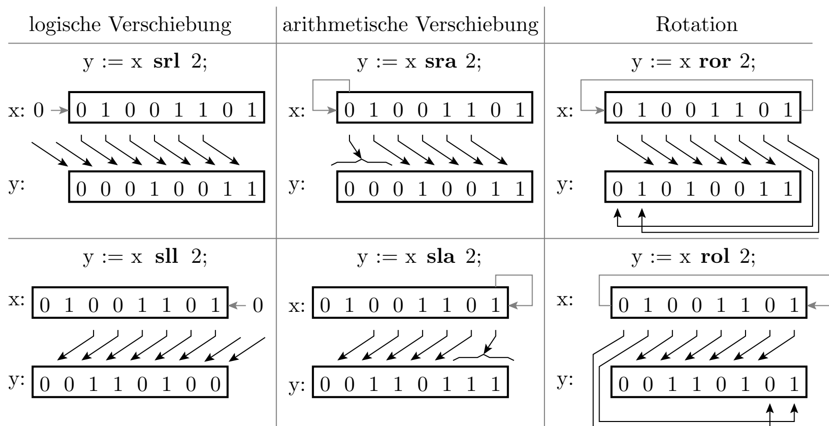
In der nachfolgenden Subtrahierschaltung mit schnellem Übertragsdurchlauf können die EXOR-Gatter für die Bildung der Summenbits weggelassen werden.



Schaltungsaufwand und Verzögerung sind ähnlich wie bei einem Subtrahierer. Wenn Schaltungsaufwand und Geschwindigkeit wichtige Optimierungsziele sind, ist der Test auf Gleichheit einem Größenvergleich in der Zielfunktion vorzuziehen.

Block-Shifter

Verschiebung und Rotation



- Operand 1: `std_logic_vector`, `unsigned`, `signed`
- Operand 2: `natural`

- Logische Verschiebung: Füllen der freiwerdenden Bits mit »0«; rechts $\mapsto$Halbierung, links $\mapsto$Verdopplung

$$\underbrace{1010\,1101_2}_{\mathrm{AD}_{16}} \text{ srl } 2 \quad\mapsto\quad \underbrace{0010\,1011_2}_{\mathrm{2B}_{16}}$$

$$\underbrace{0010\,1011_2}_{\mathrm{2B}_{16}} \text{ sll } 2 \quad\mapsto\quad \underbrace{1010\,1100_2}_{\mathrm{AC}_{16}}$$

- Arithmetische Verschiebung: Füllen der freiwerdenden Bits mit dem Vorzeichenbit; rechts $\mapsto$vorzeichenbehaftete Halbierung

$$\underbrace{0100\,1101_2}_{2^6+2^3+2^2+1=77} \quad \text{sra } 2 \quad\mapsto\quad \underbrace{0001\,0011_2}_{19\leq 77/4}$$

$$\underbrace{1011\,0010_2}_{-2^7+2^5+2^4+2=-78} \quad \text{sra } 2 \quad\mapsto\quad \underbrace{1110\,1100_2}_{-20\leq -78/4}$$

Auch bei negativen Ergebnissen wird abgerundet.

Block-Shifter

Eine konstante Verschiebung beschreibt Verbindungen ohne logische Verknüpfungen.

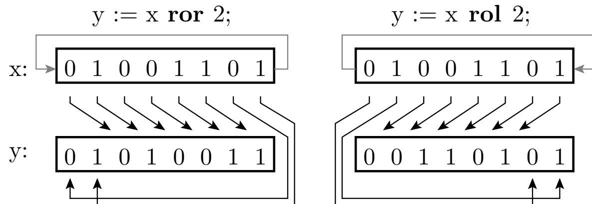
Eine variable Verschiebung beschreibt umschaltbare Verbindungen mit Multiplexern, realisiert als Block-Shifter.

Verschiebung 0 oder 1
Verschiebung 0 oder 2
Verschiebung 0 oder 4

x_0 x_1 x_2 x_3 x_4
y_0 y_1 y_2 y_3 y_4
s_0 s_1 s_2

Rotation

Bei der Rotation werden die Bits im Kreis verschoben.



Verschiebung und Rotation lassen sich auch mit Indexoperationen und Konkatenation beschreiben:

```
signal x, y: std_logic_vector(7 downto 0);
...
                                  -- identisch mit
y <= x(5 downto 0) & "00"  --  y <= x sll 2
y <= "00" & x(7 downto 2)  --  y <= x srl 2
```

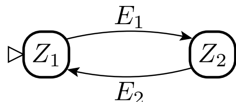
Automaten

Endliche Automaten

Endliche Automaten (FSM, Finite State Machine) beschreiben in der digitalen Schaltungstechnik Abläufe durch:

- eine Eingabemenge $\Sigma = \{E_1, E_2, \ldots\}$,
- eine Zustandsmenge $S = \{Z_1, Z_2, \ldots\}$ mit
- einem Anfangszustand $Z_1 \in S$,
- eine Ausgabemenge $\Pi = \{A_1, A_2, \ldots\}$,
- eine Übergangsfunktion $f_s : S \times \Sigma \to S$ und
- eine Ausgabefunktion $f_a : S \times \Sigma \to \Pi$.

Graphisch werden Zustände als Knoten und Zustandsübergänge als Kanten dargestellt. Die Kanten sind mit den Eingaben beschriftet, bei denen die Übergänge stattfinden.



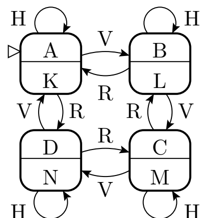
Zustand

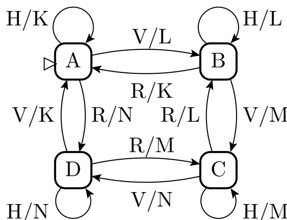
Zustandsübergang

Kennzeichnung des Startzustands

Transduktor, Moore- und Mealy-Automat

Die Automaten in der Digitaltechnik sind Transduktoren, die Ausgaben aus Zuständen und Eingaben generieren. Sie dienen zur Steuerung von Operationsabläufen.





Eingabemenge
$\Sigma = \{\mathrm{H}, \mathrm{V}, \mathrm{R}\}$
Zustandsmenge
$S = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$
Ausgabemenge
$\Pi = \{\mathrm{K}, \mathrm{L}, \mathrm{M}, \mathrm{N}\}$

- Moore-Automat: Ausgabe hängt nur vom Zustand ab.
- Mealy-Automat: Ausgabe hängt auch von der Eingabe ab und ist den Zustandsübergängen zugeordnet.

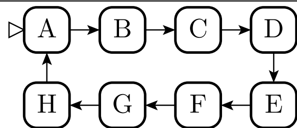
2. Automaten

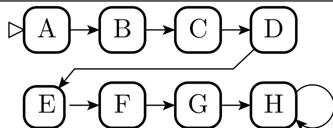
Mit beiden Automatentypen lassen sich dieselben Zielfunktionen beschreiben. Die Moore-Form braucht mehr Zustände, die Mealy-Form hat eine kompliziertere Ausgabefunktion.

Vereinbarungen:

- Wenn keine Ausgaben explizit zugeordnet sind, ist der Zustand die Ausgabe.
- Kanten mit demselben Anfangs- und Endknoten können weggelassen werden.

Automaten ohne Eingabe werden als autonome Automaten bezeichnet. Zyklische autonome Automaten dienen z.B. als Taktteiler und zyklenfreie zur Steuerung von Initialisierungsabläufen.



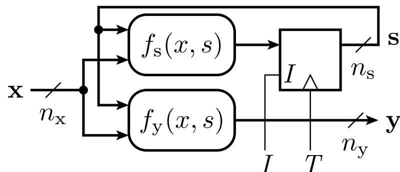


Automat als Schaltung

Bei der Realisierung als digitale Schaltung sind die Eingaben, Zustände und Ausgaben Bitvektoren. Die Darstellung von m Eingabe-, Zustands- oder Ausgabewerten verlangt Bitvektoren der Breite:

$$n \geq \log_2 (m)$$

Der Zustand wird in einem Register gespeichert. Anfangszustand ist der Initialisierungswert des Registers. Die Zustandsübergänge erfolgen mit der aktiven Taktflanke. Die Übergangs- und Ausgabefunktion wird mit Schaltungen für Verarbeitungsfunktionen realisiert:



x	Eingabe	**y**	Ausgabe
s	Zustand	T	Takt
I	Initialisierungssignal		

$f_s(x, s)$	Übergangsfunktion
$f_y(x, s)$	Ausgabefunktion

Kontrollfragen

Ein zu entwerfender Automat habe die Eingabemenge $\Sigma = \{E_1, E_2, E_3\}$, die Zustandsmenge $S = \{Z_1, Z_2, Z_3, Z_4, Z_5\}$ und die Ausgabemenge: $\Pi = \{A_1, A_2\}$.

1. Legen Sie eine Zustandscodierung mit minimaler Bitanzahl fest.
2. Legen Sie eine 1-Hot-Zustandscodierung fest[1].
3. Wie viele Möglichkeiten gibt es für beide Codierungen?

Zustand	Code min. Bitanzahl	Code 1-Hot
Z_1		
Z_2		
Z_3		
Z_4		
Z_5		

[1]Jeder Zustandswert enthält genau eine Eins und den Rest Nullen.

Zur Kontrolle

1. Die minimale Bitanzahl für $n_\mathrm{s} \geq \log_2(5)$ ist 3.
2. Ein 1-Hot-Code für 5 Werte verlangt 5 Bit.

Zustand	Code min. Bitanzahl	Code 1-Hot
Z_1	000	00001
Z_2	001	00010
Z_3	010	00100
Z_4	011	01000
Z_5	100	10000

3. Bei min. Bitanzahl sind 5 Codeworte auf $2^3 = 8$ Möglichkeiten anzuordnen:

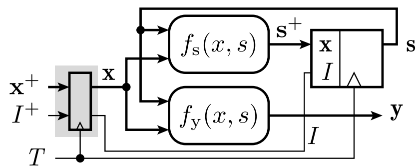
$$8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 = 6720$$

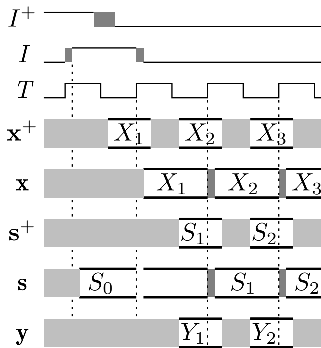
Für die 1-Hot-Codierung sind 5 Codeworte 5 Möglichkeiten zuzuordnen:

$$5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

Signalverläufe an Automaten

Alle Signaländerungen sind zeitlich am Takt ausgerichtet (RT-Funktionen). Für asynchrone Eingaben erfordert das ein zusätzliches Eingaberegister (grau unterlegt), das nicht zur Schaltung des Automaten zählt.

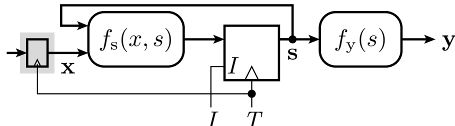




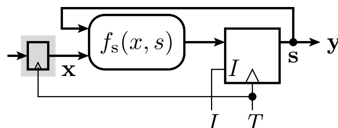
$\mathbf{x}^+$ Eingabe vor der Abtastung
I^+ Init.-Signal vor der Abtastung
y' abgetastetes Ausgabesignal
$\mathbf{s}^+$ Folgezustand
$\cdots$ Abtastzeitpunkt
▬ Wert ungültig / ohne Bedeutung

Vereinfachte Automaten

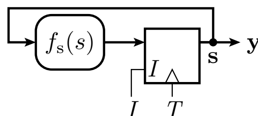
Bei einem Moore-Automaten ist die Ausgabe nicht von der Eingabe abhängig (**x** – Eingabe; **s** –Zustand; **y** – Ausgabe; I – Initialisierungssignal; T – Takt).



Ohne explizite Zuordnung von Ausgaben ist die Ausgabe der Zustand.



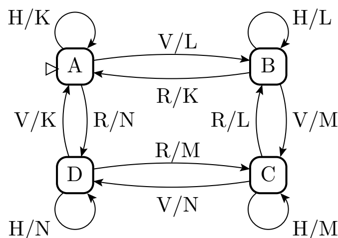
Bei einem autonomen Automaten ist der Folgezustand nur eine Funktion des Ist-Zustands.



Entwurf mit KV-Diagrammen

Vorwärts-/Rückwärtszähler

Die Funktion sei als Graph gegeben. Das Beispiel ist ein Zähler, der bei Eingabe »V« vorwärts, bei »R« rückwärts zählt und bei »H« seinen Zustand beibehält. Die Zählfolge ist »A«, »B«, ... Die Ausgabe ist den Übergängen zugeordnet (Mealy-Automat):



Ausgehend davon sind die Codierung der symbolischen Werte sowie die Größe des Zustandsregisters festzulegen und die Übergangs- und Ausgabefunktion zu entwerfen.

Aufstellen der Tabellen für die Übergangs- und Ausgabefunktion. Festlegung der Zustandscodierung.

symbolorientierte Übergangs und Ausgabefunktion ⇒ Zustandscodierung ⇒ bitvektororientierte Übergangs und Ausgabefunktion

	$s^+ = f_s(\mathbf{x}, \mathbf{s})$			$y = f_a(\mathbf{x}, \mathbf{s})$		
s	x : V	H	R	V	H	R
A	B	A	D	L	K	N
B	C	B	A	M	L	K
C	D	C	B	N	M	L
D	A	D	C	K	N	M

		C	10
		D	11
V	00	K	00
H	01	L	01
R	10	M	10
A	00	N	11
B	01		

	$\mathbf{s}^+ = f_s(\mathbf{x}, \mathbf{s})$			$\mathbf{y} = f_a(\mathbf{x}, \mathbf{s})$		
s	x : 00	01	10	00	01	10
00	01	00	11	01	00	11
01	10	01	00	10	01	00
10	11	10	01	11	10	01
11	00	11	10	00	11	10

Die drei bzw. vier symbolischen Werte für die Eingabe, die Ausgabe und den Zustand verlangen mindestens je zwei Bit. Die Eingabecodierung ist im Beispiel willkürlich gewählt. Die Ausgabecodierung ist so gewählt, das die Ausgabe gleich dem Folgezustand ist:

$$\mathbf{y} = \mathbf{s}^+$$

Das 2-Bit-Zustandsregister wird mit "00" initialisiert.

Aufstellen der KV-Diagramme

Für jedes Zustands- und jedes Ausgabebit ist ein KV-Diagramm aufzustellen. Wegen $\mathbf{y} = \mathbf{s}^+$ genügen im Beispiel zwei:

$s_1^+ = y_1$

	A	B	D	C
V	0	1	0	1
H	0	0	1	1
-	-	-	-	-
R	1	0	1	0

$s_0^+ = y_0$

	A	B	D	C
V	1	0	0	1
H	0	1	1	0
-	-	-	-	-
R	1	0	0	1

x_0, x_1, s_0, s_1; Blöcke: a, b, c, d, e

$$y_0 = s_0^+ = \bar{s}_0\bar{x}_0 \vee s_0 x_0$$

$$y_1 = s_1^+ = \bar{s}_1 s_0 \bar{x}_1 \bar{x}_0 \vee \bar{s}_1 \bar{s}_0 x_1 \vee s_1 \bar{s}_0 \bar{x}_1 \vee s_1 x_0 \vee s_1 s_0 x_1$$

Vom KV-Diagramm zur Schaltung

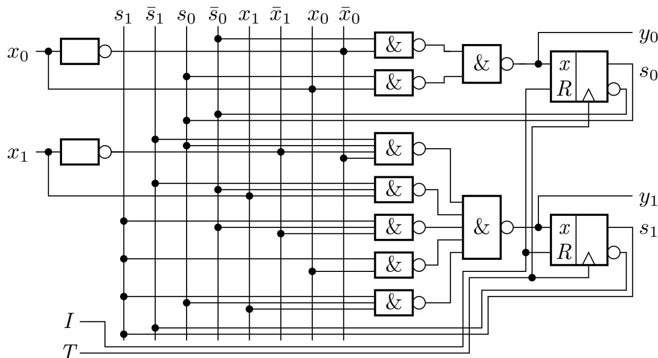
Im nächsten Schritt sind die minimierten logischen Funktionen durch verfügbare digitale Bausteine nachzubilden. Früher, als der digitale Schaltungsentwurf noch überwiegend Handarbeit war, wurden vorzugsweise Schaltkreise mit mehreren NAND-Gattern eingesetzt (vergl. Foliensatz F1). Die Gleichungsumformung aus der AND-OR-Form in die NAND-NAND-Form erfolgt mit Hilfe der De Morganschen Regeln:

$$y_0 = s_0^+ = \bar{s}_0\bar{x}_0 \vee s_0 x_0 = \overline{\left(\overline{\bar{s}_0\bar{x}_0}\right)\left(\overline{s_0 x_0}\right)}$$

$$\begin{aligned} y_1 = s_1^+ &= \bar{s}_1 s_0 \bar{x}_1 \bar{x}_0 \vee \bar{s}_1 \bar{s}_0 x_1 \vee s_1 \bar{s}_0 \bar{x}_1 \vee s_1 x_0 \vee s_1 s_0 x_1 \\ &= \overline{\left(\overline{\bar{s}_1 s_0 \bar{x}_1 \bar{x}_0}\right)\left(\overline{\bar{s}_1 \bar{s}_0 x_1}\right)\left(\overline{s_1 \bar{s}_0 \bar{x}_1}\right)\left(\overline{s_1 x_0}\right)\left(\overline{s_1 s_0 x_1}\right)} \end{aligned}$$

$$y_0 = s_0^+ = \overline{\left(\overline{\bar{s}_0\bar{x}_0}\right)\left(\overline{s_0 x_0}\right)}$$
$$y_1 = s_1^+ = \overline{\left(\overline{\bar{s}_1 s_0 \bar{x}_1 \bar{x}_0}\right)\left(\overline{s_1 s_0 x_1}\right)\left(\overline{s_1 \bar{s}_0 \bar{x}_1}\right)\left(\overline{s_1 x_0}\right)\left(\overline{\bar{s}_1 \bar{s}_0 x_1}\right)}$$

Umsetzung der logischen Funktionen in eine Gatterschaltung. Ergänzung der Speicherzellen des Zustandsregisters, des Takts und des Initialisierungssignals.



- Mit dem hier dargestellten Entwurfsfluss wurden bis vor wenigen Jahrzehnten die Ablaufsteuerungen für Rechner und andere digitale Schaltungen entworfen.
- Die Klausur zum Semesterende enthält traditionell eine Aufgabe von diesem Typ.
- Für Übergangs- und Ausgabefunktionen mit mehr als vier Eingabebits funktioniert der Bleistift-und-Papier-Entwurf nur noch begrenzt. Übergang zu rechnergestützten Verfahren wie Quine und McCluskey.
- Die zu entwerfenden Automaten in der Klausur werden nie mehr als $n_{\mathrm{x}} + n_{\mathrm{s}} \leq 4$ Eingabe- plus Zustandsbits haben.
- Heute werden die Übergangs- und Ausgabefunktionen mit Wertetabellen, Fallunterscheidungen und anderen algorithmischen Mitteln beschrieben und zu Schaltungen synthetisiert. Siehe Folgeabschnitt ...

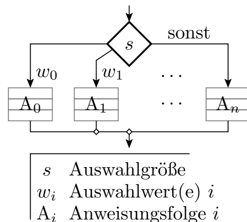
Beschreibung in VHDL

Übergangs- und Ausgabefunktionen

```
case Ausdruck is
  when Wert { |Wert } =>
      Anweisung { Anweisung}
  {when Wert { |Wert } =>
      Anweisung { Anweisung}}
  [when others =>
      Anweisung { Anweisung}]
  end case ;
```

s
w_0
A_0
w_1
A1

An

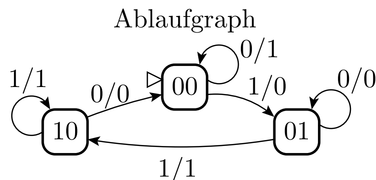


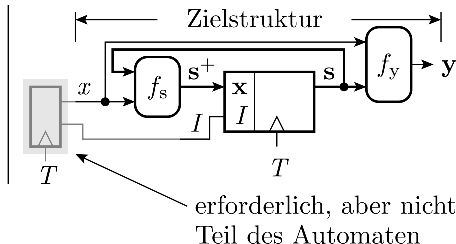
Die aus dem Zustandsgraphen ablesbaren Übergangs- und Ausgabefunktionen werden bevorzugt nach folgendem Schema beschrieben:

```
case Zustand is
 when ... => if Eingabe=... then Zustand <= ; ...
 ...
 when others => Zustand <= <Anfangswert²>; end case;
```

[2]Neuinitialisierung bei unzulässigen Zuständen ist eine vorbeugende Maßnahme gegen Abstürze (siehe nächster Abschnitt).

Beispielbeschreibung für einen Mealy-Automat



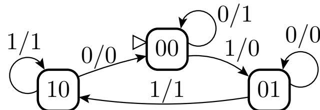


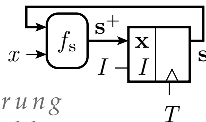
- Eingabemenge: $\{0, 1\} \Rightarrow$ Bit
- Zustandsmenge: $\{00, 01, 10\} \Rightarrow$ 2-Bit-Vektor
- Ausgabemenge: $\{0, 1\} \Rightarrow$ Bit

```
signal x, y, T, I: std_logic;
signal s: std_logic_vector(1 downto 0);
```

Übergangsfunktion als Abtastprozess

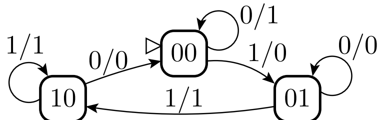
```
process(I, T)
 variable sx: std_logic_vector(2 downto 0);
begin
 if I='1' then
  s <= "00";
 elsif rising_edge(T) then
 sx := s & x;³
 case sx is
  when "00"&'0'|"10"&'0' => s<="00";
  when "01"&'0'|"00"&'1' => s<="01";
  when "10"&'1'|"01"&'1' => s<="10";
  when others => s <= "00"; -- Neuinitialisierung
 end case;                  -- bei Zustandsfehler
 end if;
end process;
```





[3] Auswahlwert ist die Konkatenation aus Zustand und Eingabe.

Ausgabefunktion als kombinatorischer Prozess



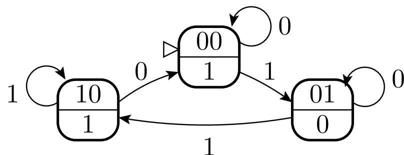

```
process(x, s)
 variable sx: std_logic_vector(2 downto 0);
begin
 sx := s & x;
 case sx is
   when "00"&'1' | "01"&'0' | "10"&'0' => y<='0';
   when others => y <= '1';
  end case;
end process;
```

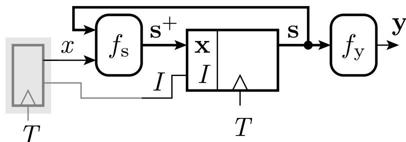
Auswahl ist wieder eine Konkatenation aus Zustand und Eingabe[4].

[4] Von unserem Entwurfssystem als schlechter Entwurfsstil bemängelt. Die empfohlene Beschreibung mit »case« für Zustands- und »if« für die Eingabeauswahl nach Folie 62 passt aber nicht auf die Folien.

Beispiel Moore-Automat

```
process(s)
begin
 case s is
  when "01"   => y<='0';
  when others => y<='1';
 end case;
end process;
```



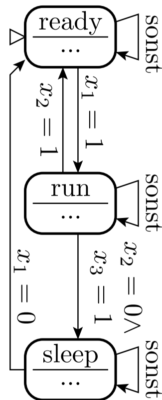


Die Eingabe-, Zustands- und Ausgabemenge sowie Übergangsfunktion und damit die Schnittstellenbeschreibung und der Abtastprozess sind wie im Beispiel zuvor. Die Ausgabe hängt vom Zustand ab und ist als kombinatorischer Prozess mit dem Zustand in der Weckliste beschrieben.

Symbolische Zustände

Zu Beschreibung eines Automaten mit symbolischen Zuständen ist für den Zustand ein Aufzählungstyp zu definieren. Die Synthese verwendet dann bevorzugt eine 1-Hot-Codierung.

```
type t_state is (ready, run, sleep);
signal s: t_state;
...
case s is
 when ready  =>
  if x1='1' then s <= run; end if;
 when run =>
  if    x2='1' then s <= ready;
  elsif x3='1' then s <= sleep;
  end if;
 when sleep =>
  if x1='0' then s <= ready; end if;
end case;
```



Redundante Zustände

Definition

Redundante Zustände sind Zustände eines Automaten, die er annehmen kann, die aber im spezifizierten Ablauf nicht enthalten sind.

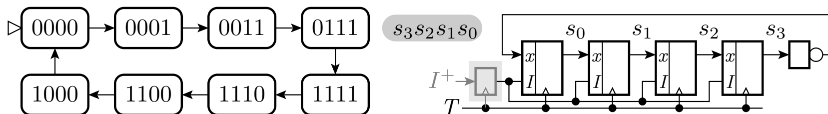
Die Zustandsanzahl einer digitalen Schaltung ist immer eine Zweierpotenz:

$$N_{\mathrm{Z}} = 2^n$$

(N_{Z} – Anzahl der Zustände; n – Anzahl der Speicherelemente). Die Anzahl der genutzten Zustände kann, aber muss keine Zweierpotenz sein. Außerdem ist eine Codierung mit minimaler Bitanzahl nicht unbedingt die günstigste Lösung. Automaten haben deshalb oft sehr viele redundante Zustände.

- Kann ein Automat redundante Zustände erreichen?
- Kann ein Automat seine redundanten Zustände wieder verlassen?

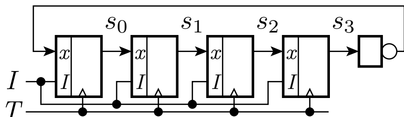
Beispiel Johnson-Zähler



Ein Johnson-Zähler ist ein mit einem Inverter rückgekoppeltes Schieberegister. Ausgehend vom Initialzustand "0000" läuft er zyklisch zuerst mit Einsen und dann wieder mit Nullen voll. Die Zykluslänge ist $2 \cdot n$ (n – Bitanzahl des Zustandsregisters). Die einfache Übergangsfunktion erlaubt hohe Taktfrequenzen.

Ein 4-Bit-Johnson-Zähler hat 16 Zustände. Davon werden 8 genutzt und 8 Zustände sind redundant.

Beschreibung in VHDL

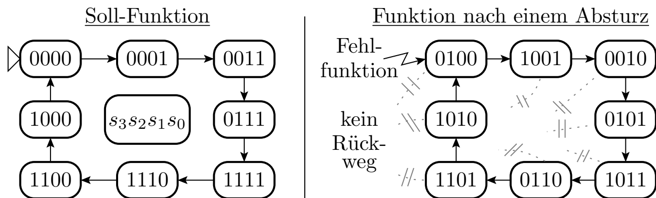



```
signal T, I: std_logic;
signal s: std_logic_vector(3 downto 0);
...
process(I, T)
begin
 if I='1' then s <= "0000";
 elsif rising_edge(T) then
  s <= s(2 downto 0) & (not s(3));
 end if;
end process;
```

Der Johnson-Zähler ist ein gutes Beispiel, dass sich Automaten oft viel einfacher als mit Fallunterscheidungen nach Zuständen und Eingabe beschreiben lassen.

Redundante Zustände und Systemabsturz

Viele Automaten nutzen nur einen kleinen Teil der 2^n (n – Bitanzahl des Zustandsregisters) Zustände, der 4-Bit-Johnson-Zähler z.B. nur 8 von 16 Zuständen. Was passiert, wenn der Automat in einen ungenutzten (unzulässigen) Zustand übergeht?

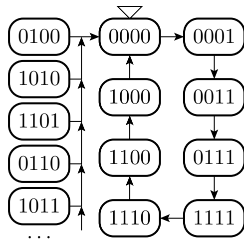


Der Johnson-Zähler durchläuft seine acht unzulässigen Zustände zyklisch. Bis zur Neuinitialisierung keine sinnvolle Reaktion mehr. Wenn ein System ohne Neuinitialisierung keinen zulässigen Zustand mehr erreicht, spricht man von einem Systemabsturz.

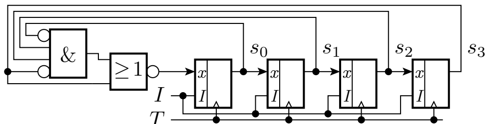
Neuinitialisierung in unzulässigen Zuständen

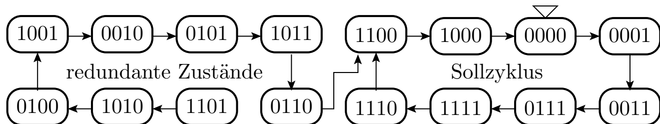
Zuweisung des Initialwertes im Others-Zweig. Verkompliziert und verlangsamt im Beispiel die Schaltung erheblich.

```
process(I, T)
begin
 if I='1' then s <= "0000";
 elsif rising_edge(T) then
  case s is
   when "0000"|"1000"|"1100"
       |"1110"|"1111"|"0111"
       |"0011"|"0001"=>
    s <= s(2 downto 0) & (not s(3));
   when others =>
    s <= "0000";
  end case;
 end if;
end process;
```



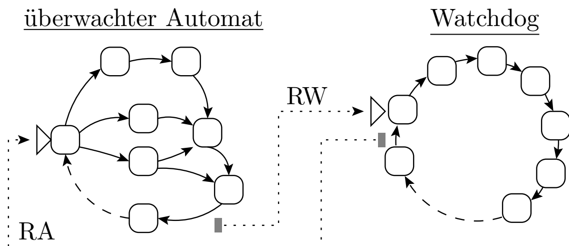
Neuinitialisierung aus nur einem redundanten Zustand je Zyklus






```
process(I, T)
begin
 if I='1' then s <= "0000";
 elsif rising_edge(T) then
  s <= s(2 downto 0) & not (s(3) or
       (not s(0) and s(1) and s(2) and not s(3)));
 end if;
end process;
```

Watchdog



RA Ein Überlauf des Watchdogs initialisiert den Automaten neu.

RW Bei einem bestimmten, regelmäßig stattfindenden Zustandsübergang wird der Watchdog neu initialisiert.

Bei großen Automaten mit praktisch unzählbar vielen redundanten Zuständen werden Abstürze durch Zeitüberwachung erkannt.

Der überwachte Automat setzt in periodisch zu erreichenden Sollzuständen einen Zeitzähler[5] zurück, der bei Überlauf den Automaten neu initialisiert.

[5]Bei Rechnern wird dieser Zeitzähler als Watchdog bezeichnet.

Spezifikation und Entwurf

Spezifikation und Entwurf

Das Automatenmodell ist ein Werkzeug zur Spezifikation und schrittweisen Verfeinerung der Zielfunktion. Das praktische Vorgehen dabei soll an zwei Beispielen vorgeführt werden:

- Spezifikation einer Fahrstuhltürsteuerung und
- Entwurf der Steuerung eines Zahlenschlosses.

Das erste Beispiel zeigt, dass eine Zielfunktion wahlweise als Moore- oder Mealy-Automat spezifiziert werden kann und dass nach den ersten Skizzen der Zielfunktion meist Nachbesserungen erforderlich werden (Widersprüche entfernen, Abläufe optimieren).

Das zweite Beispiel demonstriert, wie sich ein Automatenentwurf in einen Gesamtentwurf einbettet. Ausgehend von der Zielfunktion wird schrittweise die VHDL-Beschreibung für die Simulation und Synthese entwickelt.

Spezifikation einer Fahrstuhltürsteuerung

Die ersten Schritte der Spezifikation eines Automaten sind in der Regel die Zusammenstellung der Eingaben, Ausgaben und Zustände als Mengen.

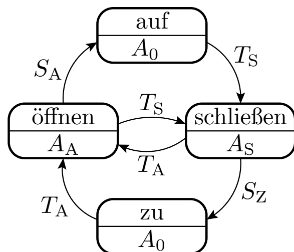
Eingangssignale der Fahrstuhltürsteuerung:

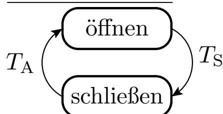
- Sensorsignal »Tür ist komplett auf« ($S_{\mathrm{A}} = 1$),
- Sensorsignal »Tür ist komplett zu« ($S_{\mathrm{Z}} = 1$),
- Tastereingabe »Tür öffnen« ($T_{\mathrm{A}} = 1$) und
- Tastereingabe »Tür schließen« ($T_{\mathrm{S}} = 1$).

Gesteuert wird der Schließmotor der Tür. Die Ausgaben sind:

- »Motor aus« (A_0),
- »Tür öffnen« (A_{A}) und
- »Tür schließen« (A_{S}).

Der Steuerablauf in der Moore-Form hat die Zustände »auf«, »schließen«, »zu« und »öffnen«. Bei den Tastereingaben wechselt der Automat in die Zustände »öffnen« bzw. »schließen« und bei Aktivierung der Endlagenschalter in die Zustände »auf« bzw. »zu«.



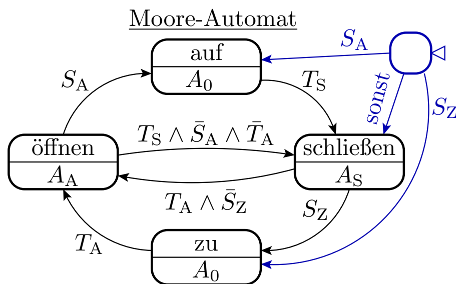


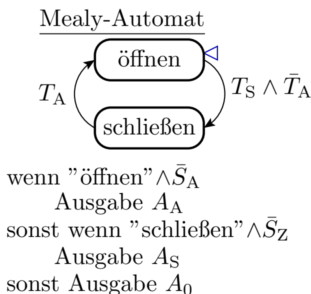
wenn "öffnen"$\wedge\bar{S}_A$
Ausgabe A_A
sonst wenn "schließen"$\wedge\bar{S}_Z$
Ausgabe A_S
sonst Ausgabe A_0

In der Mealy-Form werden nur die Zustände »öffnen« und »schließen« unterschieden. Das Abschalten des Motors über die Endlagenschalter ist in der Ausgabefunktion enthalten.

Kontrolle und Verfeinerung der Zielfunktion

Die Automatenbeschreibung des Moore-Automaten ist noch fehlerhaft/unvollständig. Sie beschreibt z.B. nicht, was in den Zuständen »öffnen« und »schließen« bei gleichzeitiger Aktivierung von Taster und Endlagenschalter passieren soll. Wenn beide Taster gedrückt sind, ist der permanente Wechsel zwischen »öffnen« und »schließen« sicher unerwünscht. Verbesserung:





Anhand der graphischen Beschreibung:

- alle Kantenübergänge, Übergangsbedingungen, ... kontrollieren,
- über mögliche Funktionserweiterungen, Verbesserungen nachdenken, ...

Ergebnis ist eine eindeutige, simulierbare in der zuvor beschriebenen Weise in eine Schaltung umsetzbare Beschreibung der Zielfunktion.

Entwurf der Steuerung eines Zahlenschlosses

Ein elektronisches Zahlenschloss wartet auf eine bestimmte Folge von Eingaben. Bei der richtigen Eingabefolge öffnet das Schloss. Im nachfolgenden Beispiel wird der Öffner durch eine LED nachgebildet:

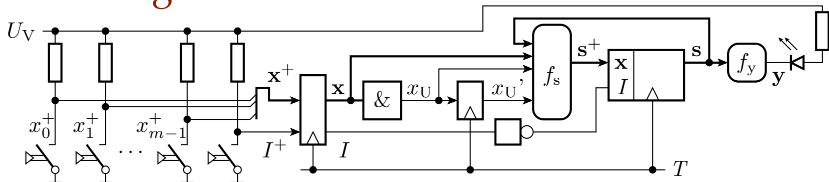
- Eingabefolge: Reset + richtige Zahlenfolge $\Rightarrow$ LED leuchtet
- Eingabefolge: Reset + falsche Zahlenfolge $\Rightarrow$ LED bleibt aus

---

Entwurfsablauf:

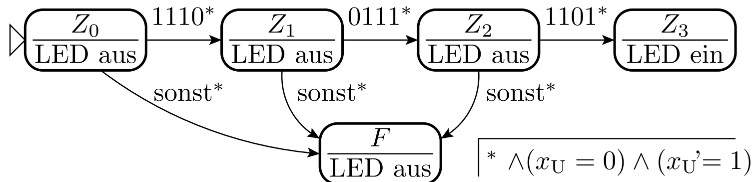
- Schaltungsskizze mit dem Automaten als Black-Box,
- Spezifikation des Zustandsgraphen für den Automaten,
- Beschreibung in VHDL,
- Simulation, Synthese, Test.

Schaltungsskizze



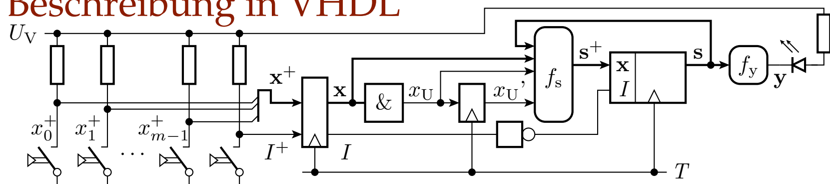
- m Zifferntasten + Reset-Taste. Asynchron prellend. Abtastung z.B. mit $T_P \approx 10$ ms. »0« wenn gedrückt.
- Gültige Eingabe: Abtastfolge keine Taste - eine Taste betätigt. Mehrere Tasten gilt als falsche Taste.
- Ausgabe-LED + Vorwiderstand. LED leuchtet bei $y = 0$.
- Moore-Automat (zustandszugeordnete Ausgabe).
- Neuinitialisierung mit abgetastetem Reset-Wert »0«.
- x_U – UND-Verknüpfung der Tasten: »1«, wenn im Abtastmoment keine Taste gedrückt, sonst »0«.

Spezifikation des Ablaufs



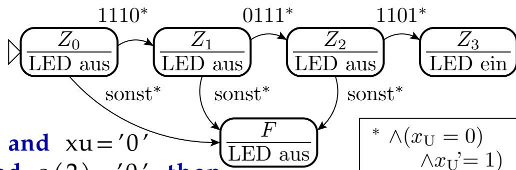
- Z_i Zustandsname; i – Nummer der nächsten Geheimzahl.
- Übergangsbedingung: aktive Taktflanke $\wedge\ x_{\mathrm{U}} = 0$ (Taste gedrückt) $\wedge\ x_{\mathrm{U}}'$ (Abtastschritt zuvor keine Taste gedrückt) $\wedge s \neq F \wedge s \neq Z_3$ (kein Endzustand).
- Richtige Zahlenfolge 0-3-1 ⇒ Eingabefolge 1110-0111-1101 ($m = 4$ Zifferntasten).
- Falsche Eingabe ⇒ Fehlerzustand F.
- Die Endzustände F und Z_3 werden nur durch Neuinitialisierung verlassen.

Beschreibung in VHDL




```
signal T, I_next, I, xu, xu_del, y: std_logic;
signal s: std_logic_vector(2 downto 0);
signal x_next, x: std_logic_vector(3 downto 0);
...
process(T) begin
 if rising_edge(T) then
  x <= x_next;
  I <= I_next;
  xu_del <= xu;
 end if;
end process;
xu <= x(0) and x(1) and x(2) and x(3);
```

```
process(I, T)
 variable v: std_logic_vector(6 downto 0);

begin
 if I='0' then
  s <= "000";
 elsif rising_edge(T) and xu='0'
    and xu_del ='1' and s(2)='0' then
  v:= s & x;
  case v is
   when "000" & "1110" => s<="001";
   when "001" & "0111" => s<="010";
   when "010" & "1101" => s<="100";
   when others => s <= "111";
  end case;
 end if;
end process;
-- Ausgabe: y<='0' (LED an), wenn s="100", sonst y<='0'
y <=not s(2) or s(1) or s(0);
```



Zustand	Codierung
Z_0	000
Z_1	001
Z_2	010
Z_3	100
F	111

Endzustände: 1--

Zusammenfassung

Ein Automat ist ein (mathematisches) Modell, das sich gut für die Spezifikation von Steuerabläufen in digitalen Schaltungen eignet.

Die Schaltung eines Automaten besteht aus einem Zustandsregister, der Übergangs- und der Ausgabefunktion. Anfangszustand ist der Initialwert des Registers. Die Übergangszeitpunkte legt der Takt fest.

Der Schaltungsentwurf mit Automaten besteht in der Regel aus

1. einem Schaltungsentwurf mit dem Automaten als Black-Box,
2. der Spezifikation des Zustandsgraphen und
3. der Beschreibung der Gesamtfunktion in VHDL.

Der kreative Aufgabenteil steckt in den beiden ersten Teilaufgaben. Die abschließende Beschreibung in VHDL ist nur noch Fleißarbeit.

Operationsabläufe

Steuerung von Operationsabläufen

Die Zustands- und Kantenanzahl eines Automaten wächst exponentiell mit der Anzahl der Eingabe- und Zustandsbits. Bei zu vielen Zuständen Aufteilung der Gesamtfunktionen in

- Operationen und
- einen Steuerablauf.

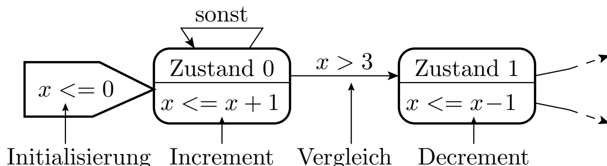
Die Steuerung erhält zusätzlich zum Zustandsregister Operandenregister und aufgabenspezifische RT-Funktionen.

Die Eingaben für den eingebetteten Automaten, die die Zustandsübergänge steuern, dürfen auch Ergebnisse von Vergleichs- und Logikoperationen sein.

Die den Zuständen und Kanten als Ausgaben zugeordneten Ausgaben dürfen auch Operationen bzw. Steuersignale für auszuführende Register-Transfer-Operationen sein.

Erweiterung der Graphendarstellung

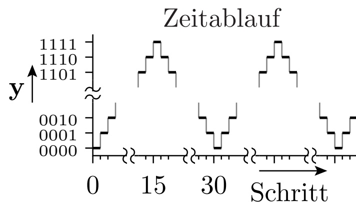
- Die Startkennung darf zusätzlich Anfangswertzuweisungen an Operandenregister enthalten.
- Übergangsbedingungen dürfen zusätzlich Vergleichsergebnisse und logische Ausdrücke sein.
- Die den Zuständen (Moore-Automat) oder den Kanten (Mealy-Automat) zugeordneten Ausgaben können auch gesteuerte Operationen sein.



Bereits ein einzelner Zähler kann die Anzahl der Ablaufzustände drastisch verringern.

Dreiecksignalgenerator

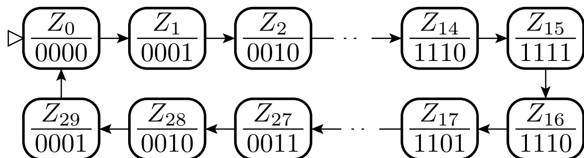
Aufgabe sei die zyklische Erzeugung des nachfolgenden Dreiecksignalverlaufs:



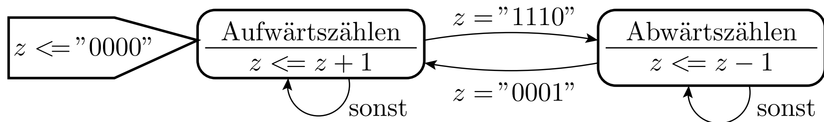
Ein bisheriger Automat für diese Aufgabe hat 30 Zustände, die zyklisch durchlaufen werden und denen Ausgaben zugeordnet sind.

3. Operationsabläufe

Ein bisheriger Automat für diese Aufgabe hat 30 Zustände, die zyklisch durchlaufen werden und denen Ausgaben zugeordnet sind.



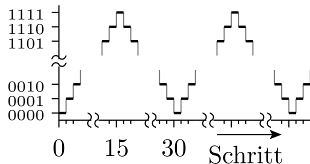
Mit einem Vor-/Rückwärtszähler als gesteuertes Rechenwerk genügen zwei Zustände.

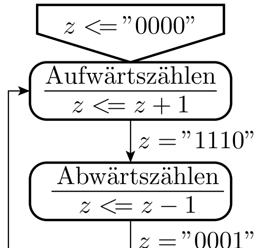


Die Beschreibung als Operationsablauf ist kleiner, skalierbar und einfacher zu programmieren.

Beschreibung in VHDL

```
signal T, I, S: std_logic;
signal z: unsigned(3 downto 0);
...
process(I, T)
begin
 if I='1' then
  S <= '0'; z <= "0000";
 elsif rising_edge(T) then
  case S is
   when '0' => z <= z + 1;
    if z="1110" then S<='1'; end if;
   when others => z <= z - 1;
    if z="0001" then S<='0'; end if;
  end case;
 end if;
end process;
```

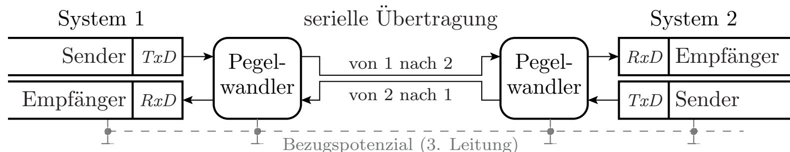


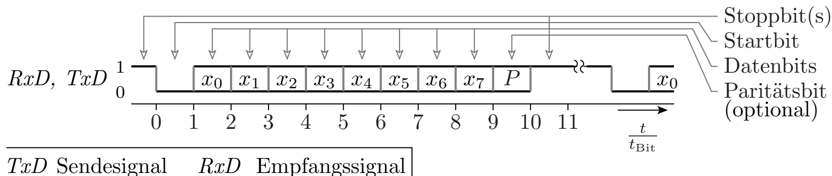


Serielle Schnittstelle

UART

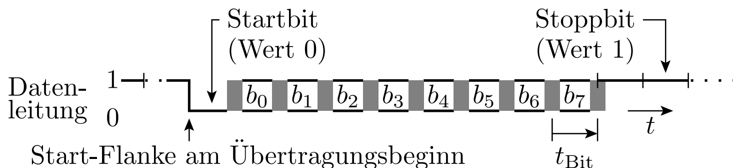
UART (**U**niversal **A**synchronous **R**eceiver **T**ransmitter) Sende- und Empfangseinheit für den Datenaustausch über 3 Leitungen, insbesondere zwischen Mikrorechnern und PC.





TxD Sendesignal *RxD* Empfangssignal

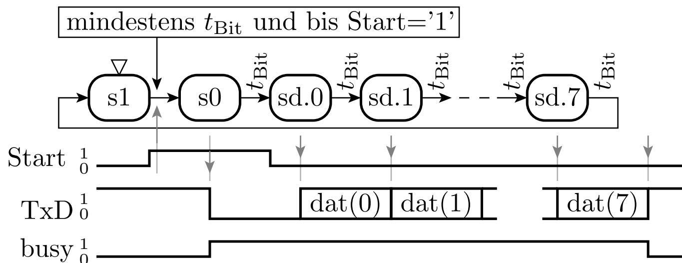
Asynchrone Übertragung ohne Takt



- Asynchrone Eingabesignale bitweise abtasten.
- Der Empfänger erkennt den Übertragungsbeginn an der Stopp-/ Start-Flanke und übernimmt die Werte nach 1,5, 2,5 etc. Bitzeiten.

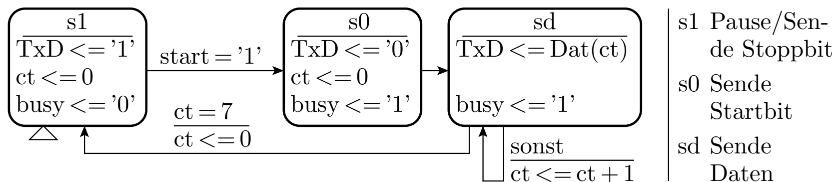
Voraussetzung: Gleich eingestellte Bitzeit, Bitanzahl, Stoppbitanzahl (und Parität) bei Sender und Empfänger. Der Kehrwert der Bitzeit ist die Baudrate b. Typische Baudraten: 4800 Bd, 9600 Bd, 19,2 kBd.

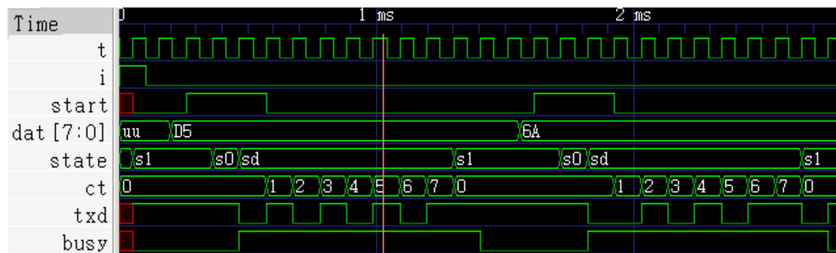
Entwicklung eines Operationsablaufs für den Sender



- Zustandsdauer mindestens/genau eine Bitzeit. Taktung des Automaten mit der Baudrate (9600 Bd = 9600 Hz).
- Zustandscodierung: s1, s0 und sd plus Bitzähler (ct).
- Startzustand: s1
- Datenausgabe an TxD und busy beim Verlassen des jeweiligen Zustands (einen Takt verzögert).

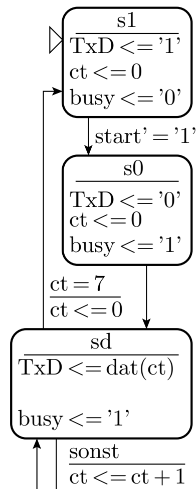
Operationsablaufgraph und Simulation





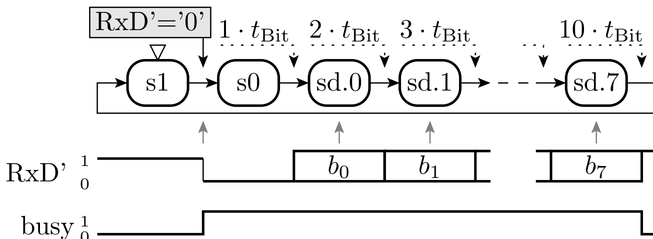
```
entity uart_sender is -- Schnittstellenbeschreibung
 port(start, T, I: in std_logic;
  dat: in std_logic_vector(7 downto 0);
  TxD, busy: out std_logic);
end entity;
architecture a of uart_sender is
 type t_state is (s1, s0, sd);
 signal state: t_state;
 signal I_del, start_del: std_logic;
 signal ct: natural range 0 to 7;
begin
 process(T) -- Abtasten der asynchronen Eingaben
 begin
  if rising_edge(T) then
   I_del <= I; start_del<= start;
  end if;
 end process;
 <Ablaufbeschreibung des Senders>
end architecture;
```

```
process(I_del, T) -- Ablaufbeschreibung
begin             -- des Senders
 if I_del='1' then
  state <= s1;
 elsif rising_edge(T) then
  case state is
   when s1 => TxD<='1';
    busy<='0'; ct<=0;
    if start_del='1' then state<=s0;
    end if;
   when s0 => TxD<='0';
    busy<='1'; ct<=0; state<=sd;
   when sd => TxD<=dat(ct);
    busy<='1';
    if ct=7 then state<=s1; ct<=0;
    else ct<=ct+1;
    end if;
  end case;
 end if;
end process;
```



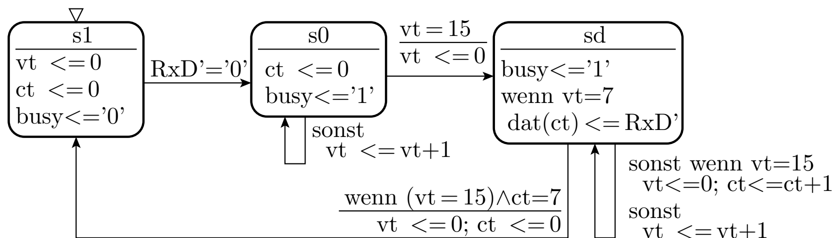
s1 Pause/Sende Stoppbit
s0 Sende Startbit
sd Sende Daten

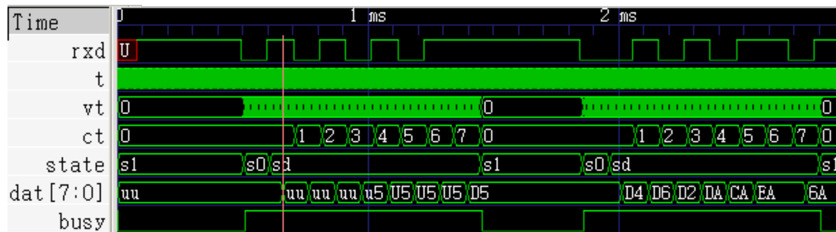
Operationsablauf für den Empfänger



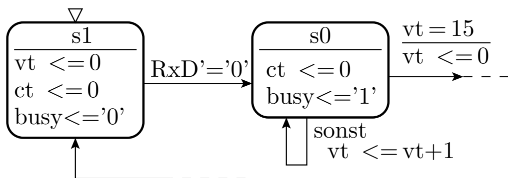
- RxD' ist das abgetastete asynchrone RxD-Signal.
- Die Startflankenabfrage »RxD'='0'« muss mehrfach je Bitzeit erfolgen. Festlegung 16×. Taktung mit der 16-fachen Baudrate.
- Messung der Bitzeit mit einem zusätzlichen 4-Bit-Zähler »vt«, der bei Erkennung der Startflanke rückgesetzt wird.
- Lesen der b_i-Werte jeweils bei »vt=7« (halbe Bitzeit).
- Ansonsten Zustandscodierung wie beim Sendeautomat.

Operationsablaufgraph und Simulation

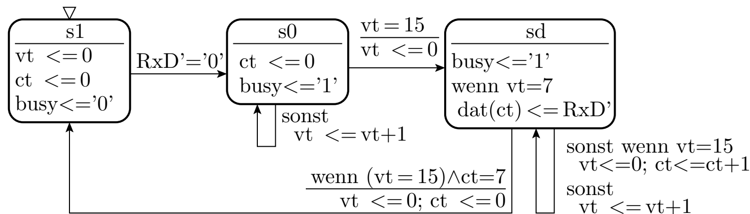




```
entity uart_receiver is
 port(T, I, RxD: in std_logic;
 dat:  out std_logic_vector(7 downto 0);
 busy: out std_logic);
end entity;

architecture a of uart_receiver is
 type t_state is (s1, s0, sd);
 signal state: t_state;
 signal vt: natural range 0 to 15;
 signal ct: natural range 0 to 7;
 signal RxD_del, I_del: std_logic;
begin

-- Abtastprozess für RxD und I
 process(T)  ...  end process;

-- Zustandsautomat
 process(I_del, T)
 begin ... end process;
end architecture;
```



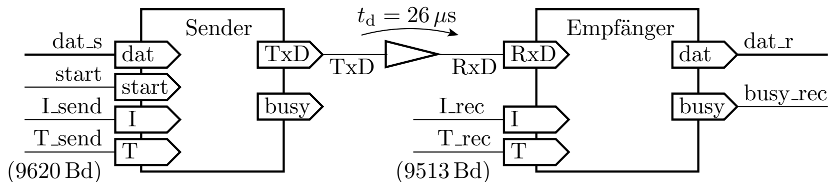

```
process(I_del, T)
begin
 if I_del='1' then
  state <= s1;
 elsif rising_edge(T) then
  case state is
   when s1 =>
    ct <= 0; vt <= 0; busy<='0';
    if RxD_del='0' then state <= s0; end if;
   when s0 =>
    ct <= 0; busy<='1';
    if vt=15 then vt <= 0; state <= sd;
    else vt <= vt +1;
    end if;
   when sd =>
    ...
```



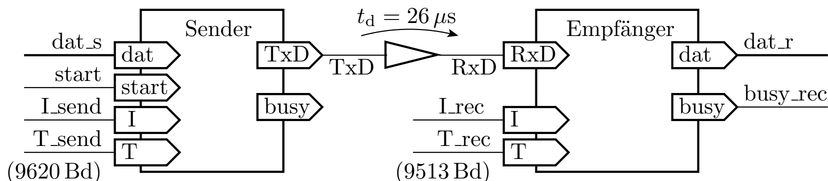

```
  when sd =>
   busy<='1';
   if vt=7 then dat(ct)<=RxD_del; end if;
   if vt = 15 then vt <= 0;
    if ct=7 then state <= s1; ct<=0;
    else ct <= ct +1;
    end if;
   else vt <= vt +1;
   end if;
  end case;
 end if;
end process;
```

Testrahmen

```
entity test_uart is end entity;
architecture a of test_uart is -- Testrahmen
 constant tb_send: delay_length := 1 sec/9620;
 constant tb_rec:  delay_length := 1 sec/9513;
  <Vereinbarung der Anschlusssignale>
begin
 <Instanziierung von Sender und Empfänger>
 <2 Prozesse zur Erzeugung der Takt und Init.-Signale>
 <Prozess zur Datenbereitstellung und zum Übertragungsstart>
end architecture;
```

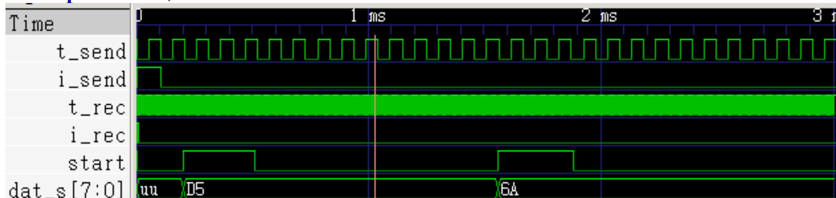


```
-- Anschlusssignale
 signal dat_s, dat_r: std_logic_vector(7 downto 0);
 signal T_send, I_send, T_rec, I_rec: std_logic;
 signal start,TxD, RxD,busy_send,busy_rec: std_logic;
begin
-- Instanziierung von Sender und Empfänger
sender: entity work.uart_sender
 port map (start=>start, T=>T_send, I=>I_send,
          dat=>dat_s, TxD=>TxD, busy=>busy_send);
receiver: entity work.uart_receiver
 port map (T=>T_rec, I=>I_rec, RxD=>RxD, dat=>dat_r,
          busy=>busy_rec);
```



```
-- Simulation einer verzögerten Datenübertragung
RxD <= transport TxD after 26 us;

-- Generierung von Sendetakt und -initialisierung
process
begin
 T_send <= '0'; I_send<='1'; wait for tb_send/2;
 T_send <= '1';              wait for tb_send/2;
 T_send <= '0'; I_send<='0'; wait for tb_send/2;
 while now < 3 ms loop
  T_send <= not T_send;      wait for tb_send/2;
 end loop;
 wait;
end process;

-- Für den Empfänger mit ca. 16-facher Frequenz
process
begin
 T_rec <= '0'; I_rec<='1'; wait for tb_rec/32;
  ...
end process;
```

```
process      -- Startsignalerzeugung und
 begin       -- Bereitstellung der Sendedaten
  start <= '0'; wait for 200 us;
  start <= '1'; dat_s <= x"D5";
  wait until busy_send = '1'; wait for 40 us;
  start <= '0';
  wait until busy_send = '0'; wait for 150 us;
  start <= '1'; dat_s <= x"6A";
  wait until busy_send = '1'; wait for 63 us;
  start <= '0';
  wait;
end process;
```



Simulationsergebnis

Time 0 1 ms 2 ms 3

Sender

t

i

start

start_del

dat_s[7:0] uu D5 6A

state s1 s0 sd s1 s0 sd s1

ct 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0

txd

busy_send

Empfänger

rxd U

t

vt 0 0 0

ct 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0

state s1 s0 sd s1 s0 sd s1

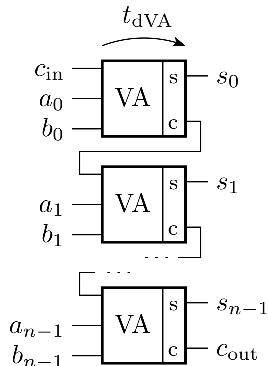
dat_r[7:0] uu uu uu uu u5 U5 U5 U5 D5 D4 D6 D2 DA CA EA 6A

busy_rec

Serieller Addierer

Generierungsschleife für einen Ripple-Addierer

```
signal a,b,s:std_logic_vector
        (n-1 downto 0);
signal cin, cout: std_logic;
...
process(a, b, cin)
 variable c: std_logic;
 begin
 c:=cin;
 for i in 0 to n-1 loop
  s(i)<=a(i) xor b(i) xor c;
  c:=(a(i) and b(i)) or
     ((a(i) or b(i)) and c);
 end loop;
 cout<=c;
end process;
```



Maximale Verzögerung: $t_{dRAdd} \approx n \cdot t_{dVA}$ (vergl. Folie 10)

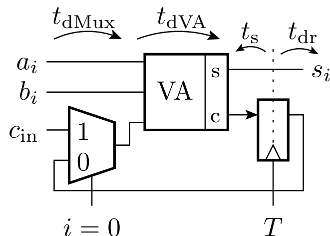
Volladdierergleichungen als Funktionen

```
-- Berechnung des Summenbits
function vas(a, b, c: std_logic) return std_logic is
begin
  return a xor b xor c;
end function;

-- Berechnung des Übertragsbits
function vac(a, b, c: std_logic) return std_logic is
 begin
  return (a and b) or ((a or b) and c);
 end function;
```

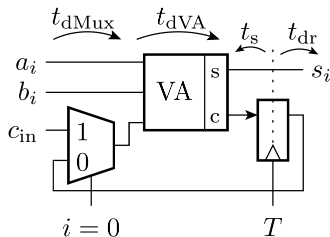
Serieller Addierer

Addition der Bits in aufeinanderfolgenden Takten. Erfordert ein Register für den Übertrag. Dafür genügt ein Volladdierer.

```
process
 variable c: std_logic;
begin
 <warte auf Operanden und Additionsauftrag>
 c:=cin;
 for i in 0 to n-1 loop
  s(i)<= vas(a(i), b(i), c);
  c:= vac(a(i), b(i), c);
  -- warte einen Takt
  wait on rising_edge(T);
 end loop;
 cout<=c;
end process;
```



Additionsdauer



Ripple-Addierer ohne schnellen Übertragsdurchlauf:

$$t_{\mathrm{dRAdd}} \geq n \cdot t_{\mathrm{dVA}}$$

Serieller Addierer:

$$t_{\mathrm{dSAdd}} \geq n \cdot (t_{\mathrm{dMux}} + t_{\mathrm{dVA}} + t_{\mathrm{s}} + t_{\mathrm{dr}})$$

Etwa doppelt so lange, wie ein Ripple-Addierer ohne und viermal so lange wie ein Ripple-Addierer mit schnellem Übertragsdurchlauf.

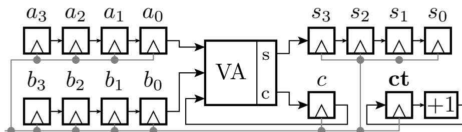
Erweiterung um eine Ablaufsteuerung

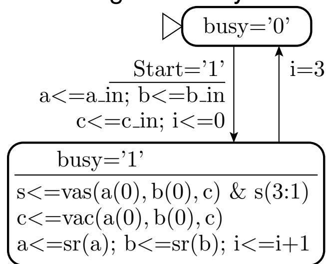
Zusätzlich erforderliche Hardware

- Schieberegister für die Summanden und das Ergebnis.
- Bitzähler.

Geplanter Ablauf:

- Warte auf ein Starteingabesignal.
- Übernehme Summanden und c_{in}. Aktiviere Ausgabe »busy«.
- Addiere nacheinander die $n = 4$ Bits.
- Deaktiviere »busy«.

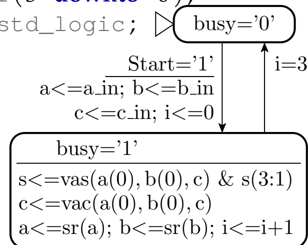




(vas(...), vac(...) – Funktionen zur Berechnung des Summen- bzw. des Übertragasbits; sr – logische Rechtsverschiebung).

Schaltungsbeschreibung in VHDL

```
signal a,a_in,b,b_in,s:std_logic_vector(3 downto 0);
signal init, T, c, c_in, Start, busy: std_logic;
signal i: natural range 0 to 3;
process(init,T)
begin
 if init='1' then busy<='0';
 elsif rising_edge(T) then
  if busy='0' and Start='1' then
   busy<='1'; i<=0;
   a<=a_in; b<=b_in; c<=c_in;
  elsif busy='1' then
   s<=vas(a(0),b(0),c) & s(3 downto 1);
   c<=vac(a(0),b(0),c); i<= i+1;
   a<='0' & a(3 downto 1); b<='0' & b(3 downto 1);
   if i=3 then busy<='0'; end if;
  end if; end if;
end process;
```



Simulationsergebnis

Signals	Waves									
Time	0					100 ns				
init	U									
t										
start										
busy	U									
a_in[3:0]	uuuu	1011				0111				
b_in[3:0]	uuuu	0110				0101				
c_in	U									
a[3:0]	uuuu	1011	0101	0010	0001	0000	0111	0011	0001	0000
b[3:0]	uuuu	0110	0011	0001	0000		0101	0010	0001	0000
c	U									
i	0	1	2	3	4	0	1	2	3	4
s[3:0]	uuuu	0uuu	10uu	010u	0010		0001	0000	1000	1100

Zusammenfassung

Der hier entworfene serielle Addierer besteht aus:

- einem Volladdierer + 1-Bit-Übertragsregister,
- 3 n-Bit Schieberegistern für Summanden und Summe,
- einem Zähler von 0 bis mindestens $n-1$ und
- einer Ablaufsteuerung mit zwei Zuständen.

Der Entwurf erfolgte in den Schritten:

- Operationsablauf ohne Steuerautomat.
- Ergänzung von RT-Funktionen (Schieberegister und Zähler) und Spezifikation des Steuerablaufs.
- Beschreibung in VHDL. Simulation. Synthese. Test. ...

Ab einer Entwurfskomplexität wie im Beispiel ist eine Simulation des Entwurfs dringend zu empfehlen.

Ein serieller Addierer ist ähnlich aufwändig wie ein Ripple-Addierer und langsamer. Für den praktischen Einsatz uninteressant.

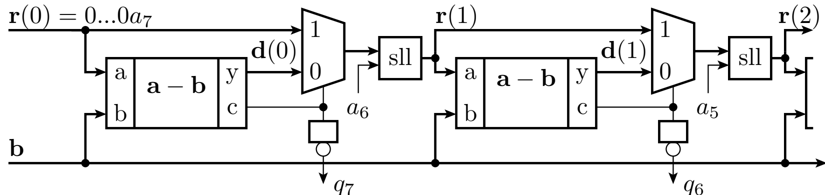
Dividierer

Division von Binärzahlen

$$\frac{\mathbf{a}}{\mathbf{b}} = \mathbf{q} + \frac{\mathbf{r}}{\mathbf{b}}$$

- Berechnung des Quotienten **q** und des Divisionsrest **r** für $a = 11$ und $b = 3$:

Bitnummer	3	2	1	0	Ergebnis
Rest	1011	1011	1011	0101	$\mathbf{r} = 0010\ (2)$
Subtrahend	− 0011	− 0011	− 0011	− 0011	
Differenz	negativ	negativ	0101	0010	
Quotient	$q_3 = 0$	$q_2 = 0$	$q_1 = 1$	$q_0 = 1$	$\mathbf{q} = 0011\ (3)$

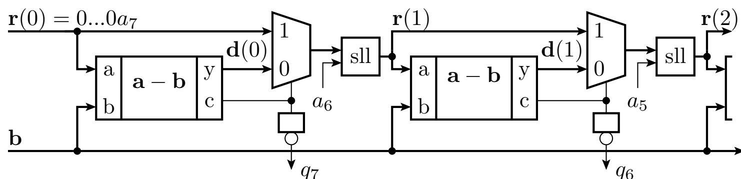


Simulation mit $a = "1010\,0011"$ und $b = "0010\,0111"$:

```
process
  variable a: unsigned(7 downto 0) := x"A3";
  variable b: unsigned(7 downto 0) := x"27";
  variable r: unsigned(7 downto 0) := x"00";
  variable q: unsigned(7 downto 0);
  variable d: unsigned(8 downto 0);
 begin
  report("a/b=q+r/b: a=10100011 b=00100111"));
  for i in 0 to 7 loop
   r := r(6 downto 0) & a(7-i);
   d := '0'&r - b;
   q := q(6 downto 0) & not d(8);
   if d(8)='0' then r := d(7 downto 0); end if;
   report(<Ausgabe Zwischenergebnisse>);
  end loop;
  report(<Ausgabe der Sollwerte zum Vergleich>);
  wait;
 end process;
```

Textausgabe der Simulation mit $a =$ ”1010 0011”, $b =$ ”0010 0111”:

```
div_alg.vhd:30:3:@0ms:(report note):
...  :30:3:  ...  : a/b=q+r/b: a=10100011 b=00100111
...  :31:3:  ...  : i|rest (r)|diff. (d)|quot.(q)
...  :37:4:  ...  : 0|00000001|111011010|UUUUUUU0
...  :37:4:  ...  : 1|00000010|111011011|UUUUUU00
...  :37:4:  ...  : 2|00000101|111011110|UUUUU000
...  :37:4:  ...  : 3|00001010|111100011|UUUU0000
...  :37:4:  ...  : 4|00010100|111101101|UUU00000
...  :37:4:  ...  : 5|00000001|000000001|UU000001
...  :37:4:  ...  : 6|00000011|111011100|U0000010
...  :37:4:  ...  : 7|00000111|111100000|00000100
...  :39:3:  ...  : soll: r=00000111 q=00000100
```



Erzeugung der Textausgaben

Die Programmierung formatierter Debug-Ausgaben ist in VHDL aufwändiger als in anderen Sprachen. Verwendete Konvertierfunktion für die Bitvektoren vom Typ `unsigned` in Ausgabetexte :

```
function str(w: unsigned) return string is
 variable tmp: string(1 to 3);
 variable s: string(1 to w'length);
 variable i: positive:=1;
begin
 for j in w'range loop
  tmp := std_logic'image(w(j));
  s(i) := tmp(2);
  i := i +1;
 end loop;
 return s;
end function;
```

Ausgabe einer Textzeile mit Zwischenergebnissen:

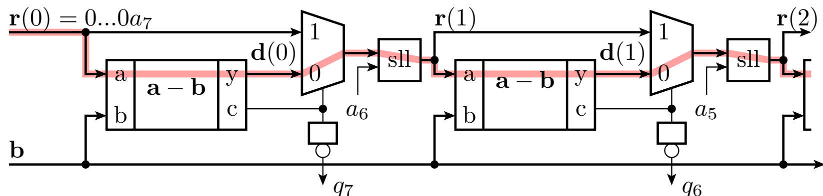
```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
...
report(integer'image(i) &"|"& str(r) &"|"& str(d)  &
    "|"& str(q));
```

(report – Funktion für die Textausgabe; *<Typ>*'image – Attribut für elementare Datentypen zur Konvertierung in eine Textdarstellung; & – Konkatenationsoperator zur Zusammenfassung von Feldern und Feldelementen gleichen Typs).

Formulieren Sie eine Anweisung für die Ergebnisausgabe:

```
soll: r=00000111 q=00000100
```

Dauer einer Hardware-Division

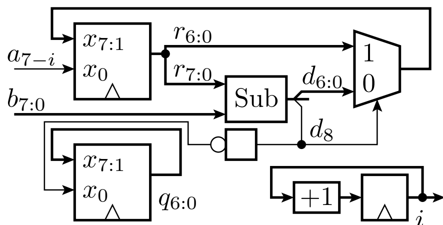


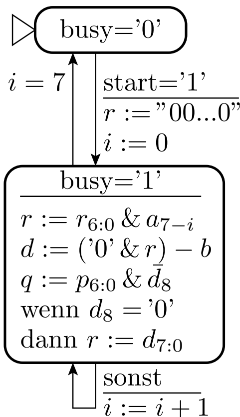
sll Linksverschiebung (Verdopplung) — längster Pfad

- Der längste Pfad verläuft durch alle Subtrahierer. Quadratische Zunahme der Laufzeit mit der Bitanzahl[6].
- Die Division wird deshalb in der Regel sequentiell in n Takten mit einer n-Bit-Subtraktion pro Takt ausgeführt.
- Vergleichbare Ausführungszeit bei deutlich geringerem Hardware-Aufwand und Stromverbrauch.

[6] Hardware-Multiplizierer nur lineare Laufzeitzunahme mit der Bitbreite.

Serieller Dividierer



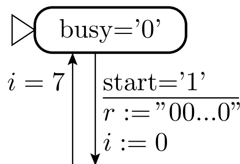


- 2 Zustände: busy∈{'0', '1'}
- Register für den Rest **r** und den Quotienten **q**
- n-Bit-Subtrahierer, Zähler von 0 bis $n-1$

Schaltungsbeschreibung in VHDL

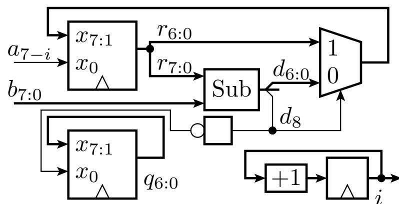
```
entity ser_div is
 generic (constant n: positive:=8);
 port (
   T, I, start: in std_logic;
   sa, sb: in unsigned(n-1 downto 0);
   busy: out std_logic;
   sr, sq: out unsigned(n-1 downto 0));
end entity;

architecture a of ser_div is
 signal si: natural range 0 to n-1;
 signal sd: unsigned(n downto 0);
begin
 process(T, I)
  <Beschreibung der Zielfunktion>
 end process;
end architecture;
```

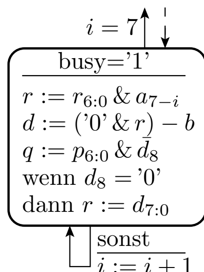
```
variable vi: natural range 0 to n-1;
variable vBusy: std_logic;
variable vr: unsigned(n-1 downto 0);
variable vq: unsigned(n-1 downto 0);
variable vd: unsigned(n downto 0);
begin
 if I='1' then vBusy := '0';
 elsif rising_edge(T) then
  if vBusy='0' and start='1' then
   vBusy :='1';
   vi :=0;
   vr := (others=> '0');
  end if;
  <Berechnungen im Zustand busy='1'>
  si <= vi; sr<= vr; sq <= vq;
  sd <= vd; busy <= vBusy;
 end if;
```



Berechnung in Variablen mit abschließender Zuweisung an Signale.

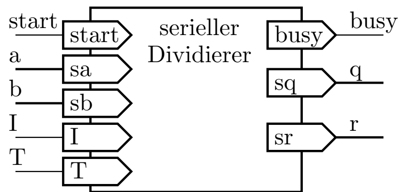
```
-- Berechnungen im Zustand busy='1'
if vBusy = '1' then
 vr := vr(6 downto 0) & sa(n-1-vi);
 vd := ('0' & vr) - sb;
 vq := vq(6 downto 0) & not vd(8);
 if vd(8)='0' then vr := vd(7 downto 0); end if;
 report(<Ausgabe Zwischenergebnisse>);
 if vi<n-1 then vi := vi + 1;
 else
  vBusy := '0';
  report(<Ausgabe Endergebnisse>);
 end if;
end if;
```





Testrahmen

```
entity test_ser_div is end entity;
architecture aaa of test_ser_div is
 signal a, b, r, q: unsigned(7 downto 0);
 signal I, T, Start, busy: std_logic;
begin
 TObj: entity work.ser_div generic map(n=>8)
  port map(T=>T, I=>I, Start=>Start, sa=>a,
           sb=>b, busy=>busy, sr=>r, sq=>q);
 <Prozess zur Takt- und Init.-Erzeugung>
 <Prozess zur Eingabeerzeugung>
end architecture;
```

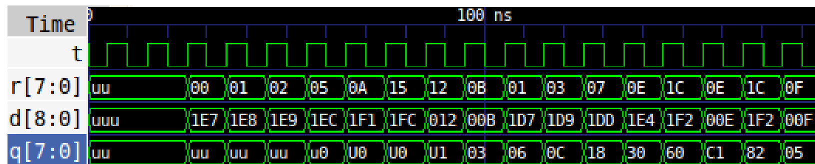


```
ClkProc: process -- Takt- und Init.-Erzeugung
  begin
   T <= '0'; I<='1'; wait for 5 ns;
   T <= '1'; wait for 5 ns;
   T <= '0'; I<='0'; wait for 5 ns;
   while now < 200 ns loop
    T <= not T; wait for 5 ns;
   end loop;
   wait;
  end process;

InpProc: process -- Eingabeerzeugung
  begin
   wait until busy = '0'; wait for 3 ns;
   a <= x"56"; b <= x"19"; Start <='1';
   wait until busy = '1'; wait for 3 ns;
   Start <='0';
   <dasselbe mit a=E1 und b=2A>
   wait;
 end process;
```

Signal- und Textausgaben zur Kontrolle

```
Time     0                                 100 ns
t
i
busy     U
start    U
a[7:0]   uu   56                              E1
b[7:0]   uu   19                              2A
r[7:0]   uu   00  01  02  05  0A  15  12  0B  01  03  07  0E  1C  0E  1C  0F
d[8:0]   uuu  1E7 1E8 1E9 1EC 1F1 1FC 012 00B 1D7 1D9 1DD 1E4 1F2 00E 1F2 00F
q[7:0]   uu   uu  uu  uu  u0  U0  U0  U1  03  06  0C  18  30  60  C1  82  05
```

..:@25ns:(**report** note): 0|00000000|111100111|UUUUUUU0
..:@35ns:(**report** note): 1|00000001|111101000|UUUUUU00
..:@45ns:(**report** note): 2|00000010|111101001|UUUUU000
..:@55ns:(**report** note): 3|00000101|111101100|UUUU0000
..:@65ns:(**report** note): 4|00001010|111110001|UUU00000




```
..:@75ns:(report note): 5|00010101|111111100|UU000000
..:@85ns:(report note): 6|00010010|000010010|U0000001
..:@95ns:(report note): 7|00001011|000001011|00000011
..:@95ns:(report note): soll: r=00001011 q=00000011

..:@105ns:(report note): 0|00000001|111010111|00000110
..:@115ns:(report note): 1|00000011|111011001|00001100
..:@125ns:(report note): 2|00000111|111011101|00011000
..:@135ns:(report note): 3|00001110|111100100|00110000
..:@145ns:(report note): 4|00011100|111110010|01100000
..:@155ns:(report note): 5|00001110|000001110|11000001
..:@165ns:(report note): 6|00011100|111110010|10000010
..:@175ns:(report note): 7|00001111|000001111|00000101
..:@175ns:(report note): soll: r=00001111 q=00000101
```

Zusammenfassung

Auch der Dividierer wurde in Schritten entworfen:

- Formulierung des Algorithmus mit einem Zahlenbeispiel.
- Programmierung und Test des Algorithmus als imperatives Programm nach der »printf«-Methode.
- Spezifikation des Steuerablaufs und aller Operationen.
- Programmierung und Simulation. Kontrolle der Signalverläufe.
- Kontrolle der Textausgaben (Regressionstest).

Tatsache

Je komplizierter die Entwurfsaufgabe desto wichtiger ist ein strukturiertes testgetriebenes Herangehen.

Der nächste Schritt wäre die Überführung in eine synthesefähige VHDL-Beschreibung + Testrahmen, Regressionstest, ..., Test in der Anwendungsumgebung.