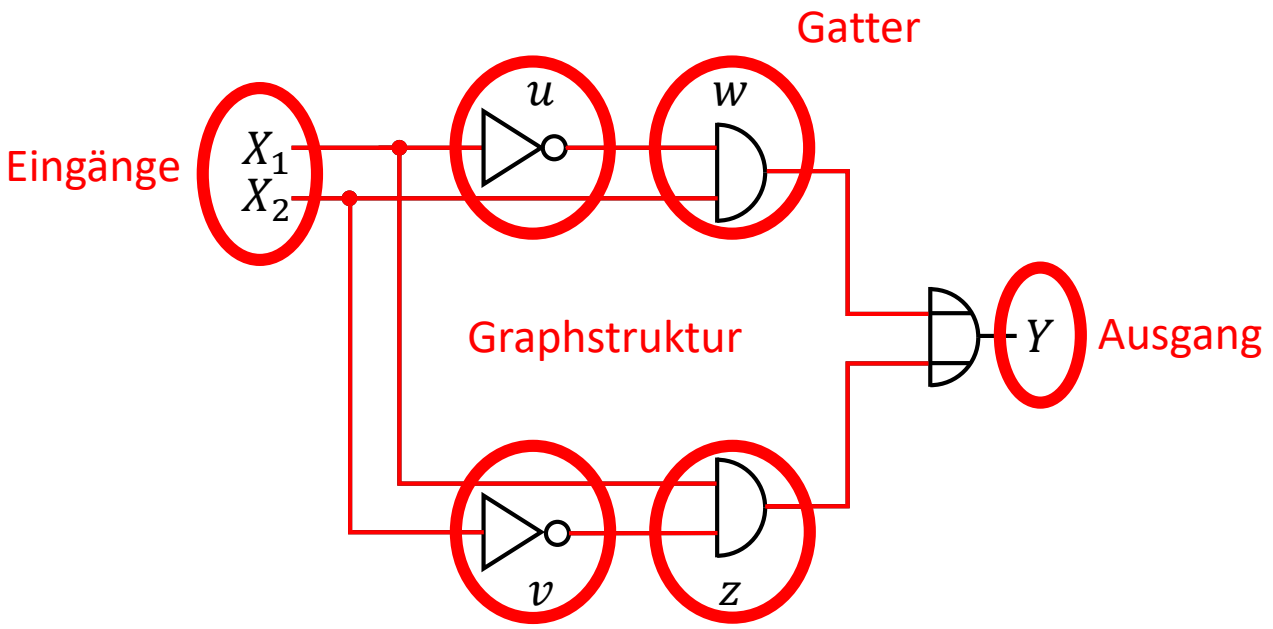
Schaltkreise und Schaltungen

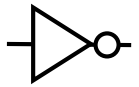
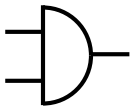
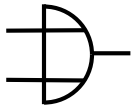
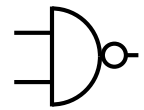
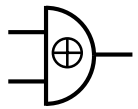
Schaltkreise

- Nach diesen schweren Vorarbeiten haben wir nun alle Zutaten beisammen, um über Schaltkreise zu reden.



Schaltsymbole

- Als Grundbausteine für den Aufbau seien folgende einfache Schaltungen gegeben, die zur Berechnung der genannten Funktionen dienen.

Schaltsymbol					
Berechnete Funktion	~	∧	∨	NAND	⊕

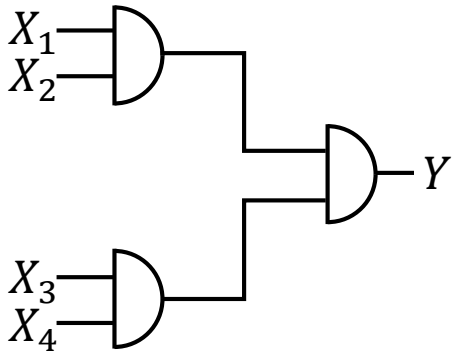
- Die Funktionen NAND und ⊕ („XOR“) sind beides 2-stellige Schaltfunktionen, die wie folgt definiert sind:

x_1	x_2	$\mathbf{NAND}(x_1, x_2)$	⊕
0	0	1	0
0	1	1	1
1	0	1	1
1	1	0	0

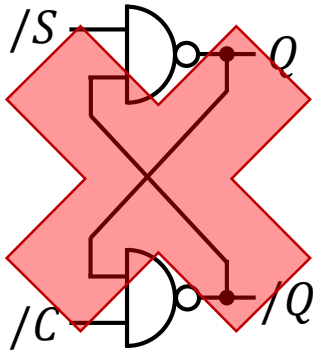
Schaltkreisdefinition

- **Definition.** Ein *Schaltkreis* ist ein 4-Tupel $S = (X, G, g, Y)$. Dabei ist:
  - $X = (X_1, \dots, X_n)$ eine Folge von Eingängen
  - $Y = (Y_1, \dots Y_m)$ eine Folge von Ausgängen
  - $G = (V, E)$ ein zykelfreier gerichteter Graph mit $\{0, 1\} \cup \{X_1, \dots, X_n\} \cup \{Y_1, \dots, Y_m\} \subseteq V$ und $I = V \setminus \{0, 1\} \cup \{X_1, \dots X_n\} \cup \{Y_1, \dots, Y_m\}$. I heißt die Menge der Gatter und jeder Knoten in I hat Ingrad 1 oder 2 und mindestens einen direkten Nachfolger.
  - $g: I \to F$ eine Abbildung, die jedem Gatter eine kommutative Schaltfunktion zuweist (F sei die Menge aller Schaltfunktionen). Dies erlaubt uns, G wie in dem Beispiel zu zeichnen, d.h. unter Verwendung von Schaltsymbolen. Dazu muss aber auch noch für alle $v \in \mathrm{V}$ gelten: ist $g(v)$ eine k-stellige Schaltfunktion, so hat v den Ingrad k.

Weitere Beispiele



Die Pfeilspitzen an den Kanten lässt man weg, die Kantenrichtungen ergeben sich aus den Schaltsymbolen der Gatter.



Kreuzende Linien symbolisieren Verbindungen nur dann, wenn sie durch einen ausgefüllten Kreis markiert sind.

Intuitiv würde man gerne so etwas aussagen (können) wie :

$$Y = X_1 \wedge X_2 \wedge X_3 \wedge X_4$$

Die Knoten 0 und 1 zeichnet man nur, wenn sie Nachfolger haben.

Informeller Vergleich: Schaltkreise und Boole’sche Ausdrücke

- Es gibt Parallelen zwischen Boole’sche Ausdrücken und Schaltkreisen
- Beide haben eine inhärente Struktur:
  - Boole’sche Ausdrücke durch die (implizite) Klammerung
  - Schaltkreise durch den verwendeten Graphen
- Beide Strukturen beinhalten die Konstanten 0, 1 und haben Variablen (Eingänge).
- Es gibt eine Verbindung zu Schaltfunktionen, but per se sind weder Boole’sche Ausdrücke noch Schaltkreise selbst Funktionen.

Belegung von Eingängen

- **Definition**. Sei $S = ((X_1, \ldots, X_n), G, g, Y)$ ein Schaltkreis. Eine *Eingangsbelegung* ist eine Abbildung $\beta: \{X_1, \ldots X_n\} \to \{0, 1\}$.

  Ist $a \in \{0,1\}^n$, dann bezeichnet β_a die Eingangsbelegung mit $X_i \mapsto a_i$ für alle $1 \leq i \leq n$.

Der berechnete Wert an einem Knoten

- Wir können nun die Erkenntnis ausnutzen, dass in einem zykelfreien gerichteten Graphen jeder Knoten eine Höhe hat:
- **Definition.** Die Abbildung $\phi_\beta: V \to \{0,1\}$ definiert für jeden Knoten v in einem Schaltkreis $S = ((X_1, \dots, X_n), (V, E), g, Y)$ den *Wert von v unter der Belegung β* wie folgt rekursiv über die Höhe von v**.**

  1. Folgende Fälle decken alle Knoten der Höhe 0 ab:

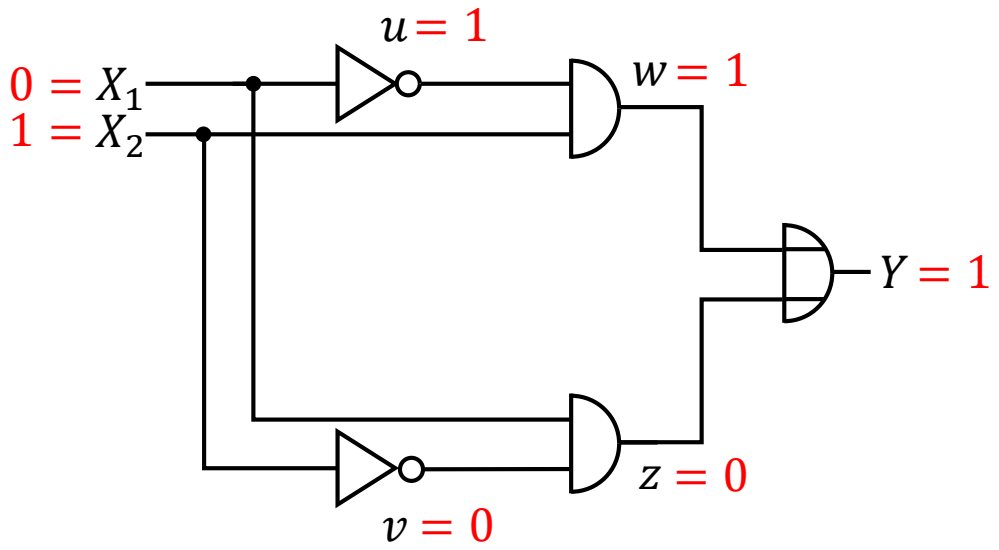
$$\phi_\beta(0) = 0$$
$$\phi_\beta(1) = 1$$
$$\phi_\beta(X_i) = \beta(X_i) \text{ für alle } 1 \leq i \leq n$$

  2. Für Knoten v mit einer Höhe > 0 und jeweils direkten Vorgängern $w_1, \dots, w_{k_v}$ gilt:

$$\phi_\beta(v) = g(v)\left(\phi_\beta(w_1), \dots, \phi_\beta\left(w_{k_v}\right)\right)$$

Beispiel

- Welchen Wert hat jeder Knoten in dem folgenden Schaltkreis unter der Belegung $X_1 \mapsto 0, X_2 \mapsto 1$?



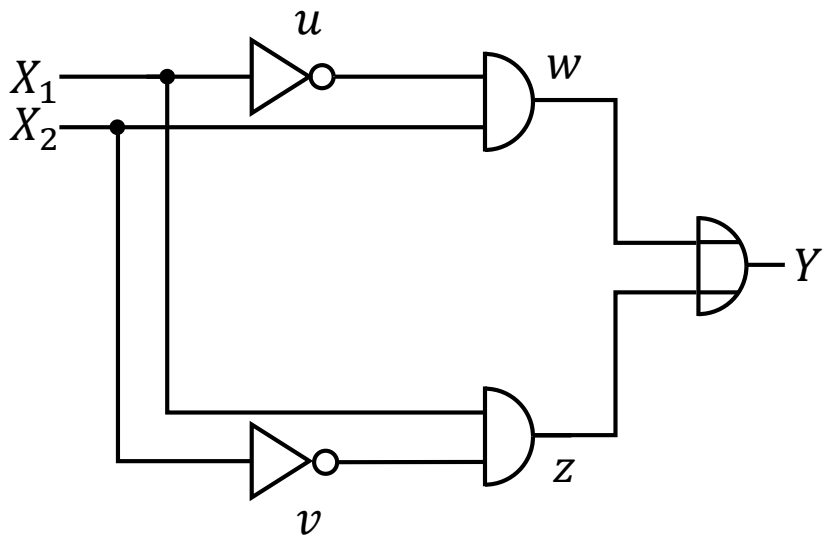
Durch einen Knoten berechnete Funktion

- Für unterschiedliche Belegungen β kann der durch ϕ_β berechnete Wert eines Knotens in einem Schaltkreis unterschiedlich sein.
- Abkürzung: Ist die Eingangsbelegung β mittels eines Vektors a definiert, schreiben wir statt ϕ_{β_a} kürzer ϕ_a.
- **Definition.** Sei v ein Knoten in einem Schaltkreis mit Eingängen $X_1, \dots, X_n$. Die Schaltfunktion $f_v: \{0,1\}^n \to \{0,1\},\ x \mapsto \phi_x(v)$ heißt *die durch den Knoten v berechnete Funktion.*

Beispiel

- Wir betrachten den Wert des Knotens Y unter allen möglichen Eingangs-belegungen β_a in dem bekannten Schaltkreis:

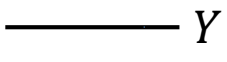
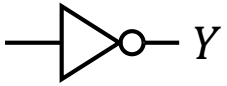
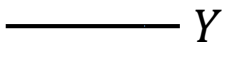
a	$\boldsymbol{\beta_a(X_1)}$	$\boldsymbol{\beta_a(X_2)}$	$\boldsymbol{\phi_{\beta_a}(Y)}$
$(0,0)$	0	0	0
$(0,1)$	0	1	1
$(1,0)$	1	0	1
$(1,1)$	1	1	0



- Es zeigt sich, dass die durch den Knoten Y berechnete Funktion f_Y die Schaltfunktion $\oplus$ ist.

Darstellungssatz

- **Satz (Darstellungssatz).** Zu jeder Schaltfunktion $f: \{0,1\}^n \to \{0,1\}$ gibt es einen Schaltkreis $S = ((X_1, \ldots, X_n), G, g, Y)$, sodass f die durch Y berechnete Funktion ist.
- Beweis per vollständiger Induktion über n.
  Induktionsanfang. Für $n = 1$ gibt es genau vier mögliche Schaltfunktionen f_1, f_2, f_3, f_4, wie in der folgenden Tabelle abgebildet. Diese werden jeweils durch die Schaltkreise in Spalte S_i berechnet. ■

i	$\boldsymbol{f_i(0)}$	$\boldsymbol{f_i(1)}$	$\mathbf{S_i}$
1	0	0	
2	0	1	
3	1	0	
4	1	1	

Darstellungssatz (2)

- Induktionsschritt. $n \to n+1$
  Sei f eine beliebige $n+1$-stellige Schaltfunktion. Wir definieren zwei Hilfsfunktionen g_0 und g_1 wie folgt:

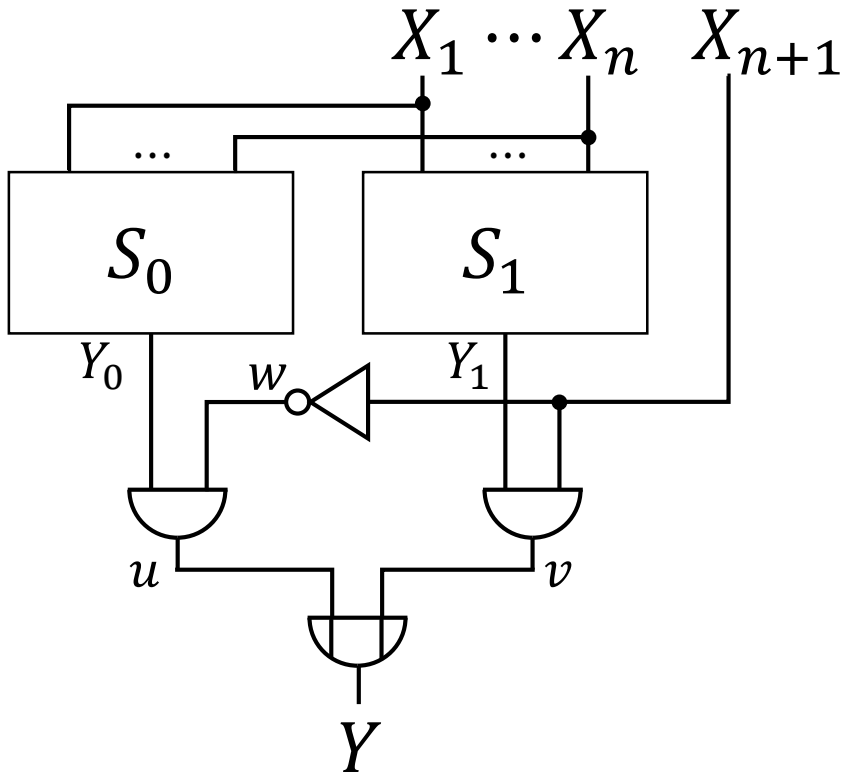
$$g_0(x_1, \dots, x_n) = f(x_1, \dots, x_n, 0)$$
$$g_1(x_1, \dots, x_n) = f(x_1, \dots, x_n, 1)$$

  d.h., g_0 und g_1 sind beide jeweils n-stellige Schaltfunktionen.
  Dann gibt es nach Induktionsvoraussetzung zwei Schaltkreise S_0 und S_1 mit jeweils einem Ausgang Y_0 bzw. Y_1, sodass g_0 die durch Y_0 und g_1 die durch Y_1 berechnete Funktion ist.

Darstellungssatz (3)

Wir betrachten den nebenstehenden Schaltkreis und insbesondere den durch den Knoten Y berechneten Wert $\phi(Y)$.

Für diesen gilt:



Darstellungssatz (4)

$$\begin{aligned}\phi(Y) &= \phi(u) \vee \phi(v)\\ &= \big(\phi(Y_0) \wedge \phi(w)\big) \vee \big(\phi(Y_1) \wedge \phi(X_{n+1})\big)\\ &= \big(g_0(\beta(X_1), \dots, \beta(X_n)\,) \wedge \sim(\phi(X_{n+1}))\big) \vee \Big(g_1\big(\beta(X_1), \dots, \beta(X_n)\big) \wedge \phi(X_{n+1})\Big)\\ &= \big(f(\beta(X_1), \dots, \beta(X_n), 0) \wedge \sim(\beta(X_{n+1}\,)\big) \vee \big(f(\beta(X_1), \dots, \beta(X_n), 1) \wedge \beta(X_{n+1})\big)\end{aligned}$$

Falls $\beta(X_{n+1}) = 1$ gilt, ist die linke Hälfte dieses Ausdrucks (bis zum V) gleich 0 und die rechte Hälfte gleich $f\left(\beta\big(X_1), \dots, \beta(X_{n+1})\big)\right)$. Falls $\beta(X_{n+1}) = 0$ gilt, ist die rechte Hälfte gleich 0 und die linke Hälfte gleich $f\left(\beta\big(X_1), \dots, \beta(X_{n+1})\big)\right)$. Zusammen ergibt sich aufgrund der Disjunktion: $\phi(Y) = f\big(\beta(X_1), \dots, \beta(X_{n+1})\big)$. *qed.*

Schaltkreise zum Addieren

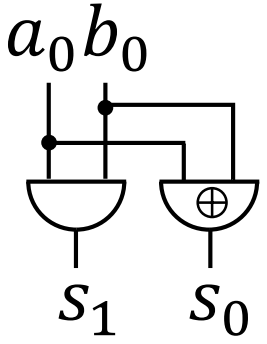
- Wir wollen uns nun der Konstruktion eines einfachen Schaltkreises zuwenden, mit dem man zwei Zahlen addieren kann.
- Wir beschränken uns dabei auf ganze Zahlen.
- Beginnen wollen wir mit der Addition zweier Bits.

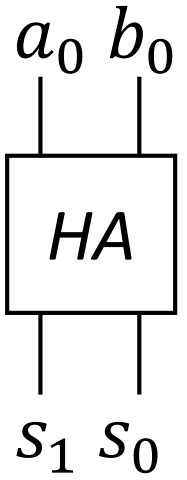
X_1	X_2	$X_1 + X_2$	$\langle X_1 + X_2 \rangle_2$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	2	10

Halbaddierer

- **Definition.** Ein Halbaddierer ist ein Schaltkreis, der die folgende Funktion $h: \{0,1\}^2 \to \{0,1\}^2$ berechnet: $h(a_0, b_0) = (s_1, s_0)$ mit $\langle s_1 s_0 \rangle_2 = a_0 + b_0$.

a_0	b_0	s_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0





Übertrag beim Addieren

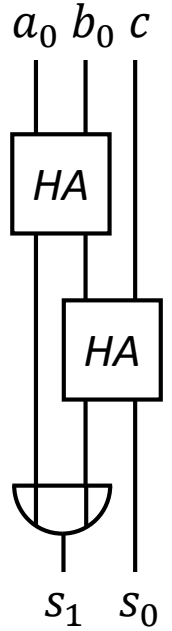
- Wir erinnern uns an die Schulmethode zum schriftlichen Addieren von Dezimalzahlen:

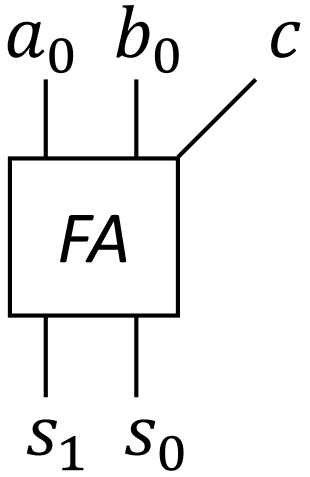
	8	7	2
	5_{1}	4	6
$_{1}$			
1	4	1	8

Volladdierer

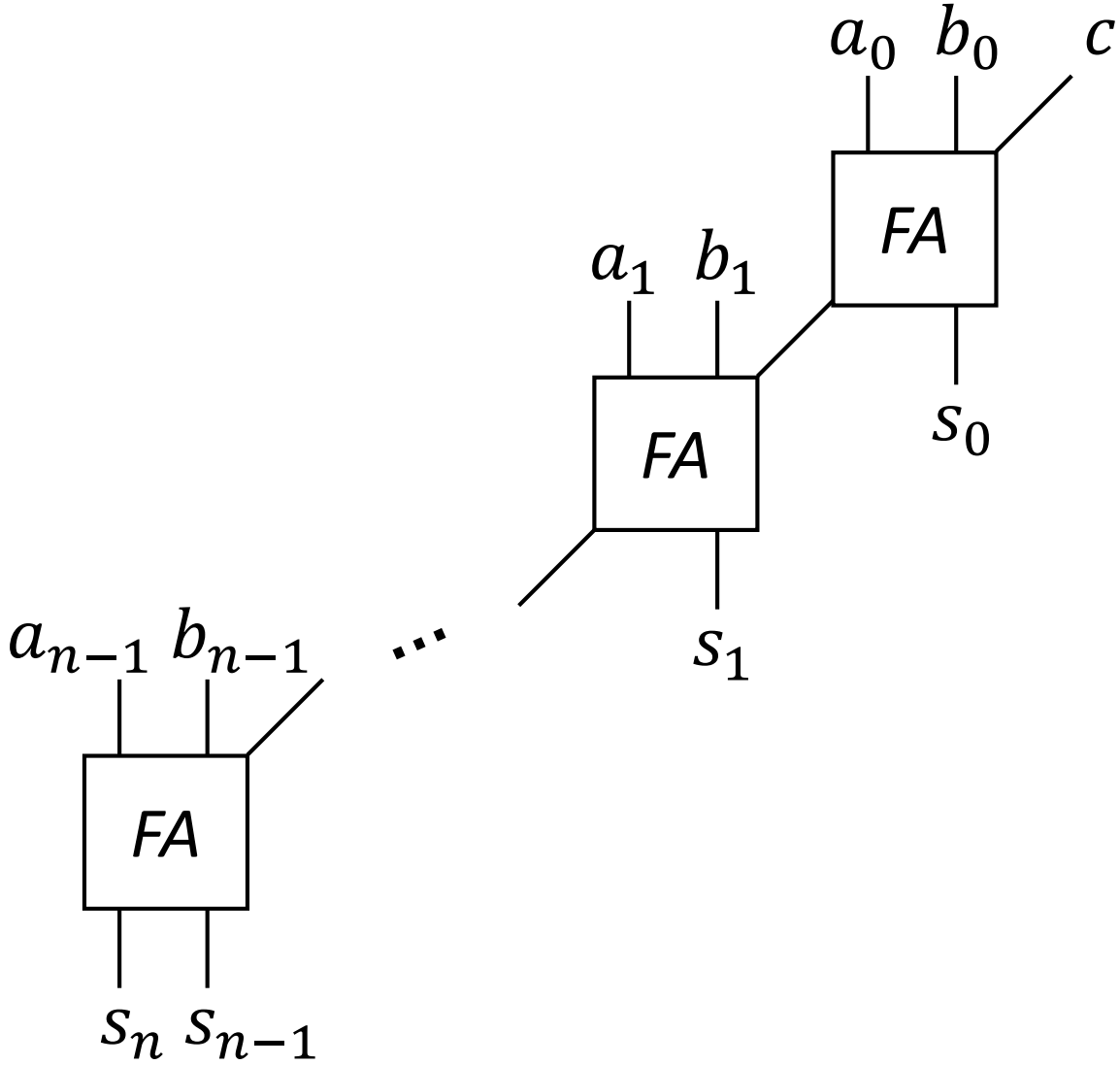
- Im Gegensatz zu einem Halbaddierer hat ein Volladdierer drei Eingänge a_0, b_0, c und berechnet deren Summe.

a_0	b_0	c	s_1	s_0
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



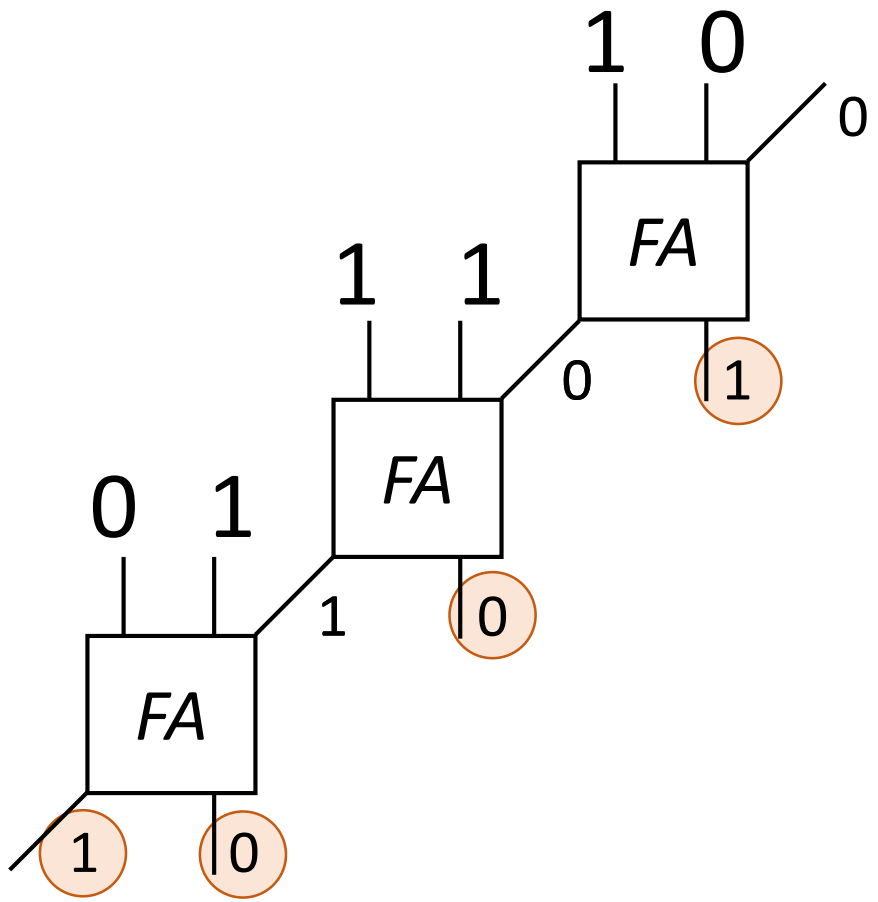


Carry-Chain-Addierer

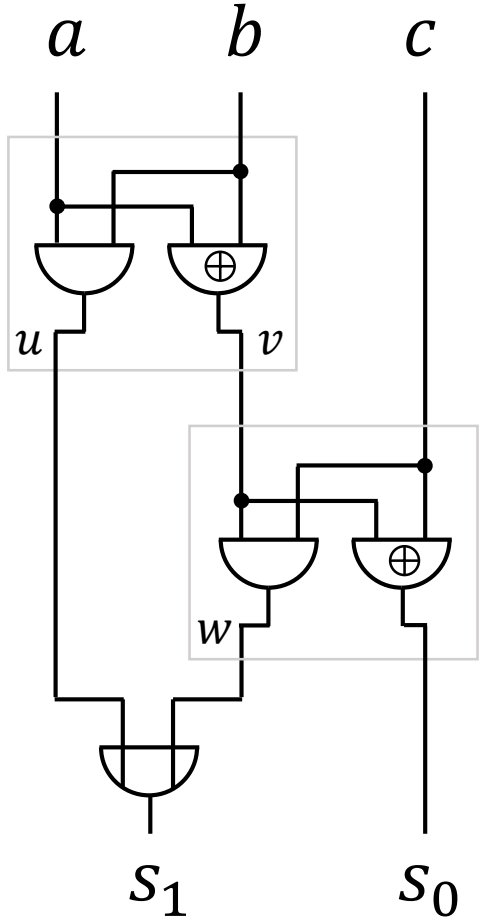


Beispiel

$$
\begin{array}{rrrr}
 & 0 & 1 & 1 \\
+ & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 1
\end{array}
$$



Was genau “rechnet” ein Volladdierer?



- Sei β eine beliebige Eingangsbelegung.
- $\phi_\beta(s_0) = \phi_\beta(v) \oplus \phi_\beta(c)$
  $= \phi_\beta(a) \oplus \phi_\beta(b) \oplus \phi_\beta(c)$
- $\phi_\beta(s_1) = \phi_\beta(u) \vee \phi_\beta(w)$
  $= \big(\phi_\beta(a) \wedge \phi_\beta(b)\big) \vee \big(\phi_\beta(v) \wedge \phi_\beta(c)\big)$
  $= \big(\phi_\beta(a) \wedge \phi_\beta(b)\big) \vee$
  $\Big(\big(\phi_\beta(a) \oplus \phi_\beta(b)\big) \wedge \phi_\beta(c)\Big)$

Abkürzung

- Wie z.B. die vorherige Folie zeigt, müssen wir bei der Analyse des Werts eines Knotens sehr oft $\phi_\beta(\cdots)$ schreiben.
- Wir erlauben uns folgende Abkürzung, um Schreibarbeit zu sparen: *Ist aufgrund des jeweiligen Kontexts ersichtlich, was gemeint ist, schreiben wir statt $\phi_\beta(v)$ kurz v.*
- Die Einschränkung auf den jeweiligen Kontext ist dabei sehr wichtig, denn natürlich ist ein Knoten v nicht dasselbe wie die Funktion ϕ_β.
- Beispiel: aus $\phi_\beta(x) = \phi_\beta(y) \wedge \phi_\beta(z)$ wird dann kurz: $x = y \wedge z$

Der Wert von s_0

- Da $s_0 = a \oplus b \oplus c$ gilt, ist $\phi_\beta(s_0) = 1$ genau dann, wenn einer der folgenden vier Fälle vorliegt:

	$\beta(a)$	$\beta(b)$	$\beta(c)$
1.	1	0	0
2.	0	1	0
3.	0	0	1
4.	1	1	1

- Bezogen auf die Summe der Eingänge a, b, c unter der Belegung β bedeutet dies:

der von s_0 berechnete Wert ist $\begin{cases} 1, \text{falls } \beta(a) + \beta(b) + \beta(c) = 1 \text{ oder } 3 \text{ ist.} \\ 0, \text{falls } \beta(a) + \beta(b) + \beta(c) = 0 \text{ oder } 2 \text{ ist.} \end{cases}$

Der Wert von s_1

- $s_1 = (a \wedge b) \vee \big((a \oplus b) \wedge c\big)$
  $= (a \wedge b) \vee \big((a \vee b) \wedge c\big)$
  $= (a \wedge b) \vee \big((a \wedge c) \vee (b \wedge c)\big)$
  $= (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$

$\beta(a)$	$\beta(b)$	$a \oplus b$	$a \vee b$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

- Bezogen auf die Summe der Eingänge a, b, c unter der Belegung β bedeutet dies:

  der von s_1 berechnete Wert ist $\begin{cases} 1, \text{falls } \beta(a) + \beta(b) + \beta(c) = 2 \text{ oder } 3 \text{ ist.} \\ 0, \text{falls } \beta(a) + \beta(b) + \beta(c) = 0 \text{ oder } 1 \text{ ist.} \end{cases}$

Der Wert von $\langle s_1 s_0 \rangle_2$

- Fasst man die Ergebnisse der beiden vorherigen Folien in einer Tabelle zusammen, erhält man:

$\beta(a) + \beta(b) + \beta(c)$	$\phi_\beta(s_1)$	$\phi_\beta(s_0)$
0	0	0
1	0	1
2	1	0
3	1	1

- Das bedeutet, dass $\langle \phi_\beta(s_1)\phi_\beta(s_0) \rangle_2 = \beta(a) + \beta(b) + \beta(c)$
- Ein Volladdierer berechnet also tatsächlich die Summe seiner Eingänge.

Hardware in der Realität

- Unsere bisherigen Betrachtungen von Schaltkreisen haben sich auf deren *logische* Funktionalität konzentriert.
- In tatsächlicher Hardware müssen Gatter und Eingänge jedoch in irgendeiner Technologie *physikalisch* realisiert sein.
- Dadurch ergeben sich bestimmte Eigenschaften und Umstände, die wir bislang noch nicht betrachtet haben.

Mit Spannung erwartet

- Computer funktionieren mit Elektrizität. Die Darstellung der beiden binären Werte 0 und 1 erfolgt mittels unterschiedlicher Spannungen.
- Für Eingänge eines Schaltkreises oder eines Gatters unterscheiden wir folgende Spannungen:
  - VCC: die sog. “Versorgungsspannung”
  - V_{IL}: niedriger Inputpegel
  - V_{IH}: hoher Inputpegel
- Ob eine Eingangsspannung U den Wert 0 oder 1 repräsentiert, liefert uns eine Interpretationsfunktion I, die wie folgt definiert ist:

$$I(U) = \begin{cases} 0, & \text{falls } U \leq V_{IL} \\ 1, & \text{falls } U \geq V_{IH} \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

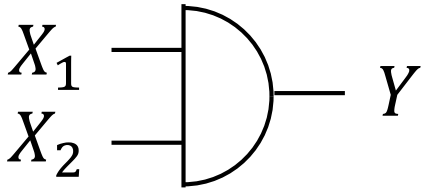
Ausgangssituation

- Für die Ausgänge eines Schaltkreises oder eines Gatters gibt es in jeder Technologie analog folgende Spannungen:
  - V_{OL}: niedriger Ausgangspegel
  - V_{OH}: hoher Ausgangspegel
- Die beiden logischen Werte 0 und 1 werden dann an einem Ausgang durch eine Spannung U wie folgt repräsentiert:

$$U \begin{cases} \leq V_{OL}, \text{ für den Wert 0} \\ \geq V_{OH}, \text{für den Wert 1} \end{cases}$$

- Damit dieser Ansatz Sinn hat, sollte gelten:
  - $V_{OL} \leq V_{IL} < V_{IH} \leq V_{OH}$
  - $0 \leq U \leq VCC$, für alle Spannungen U im Schaltkreis.

Quiz



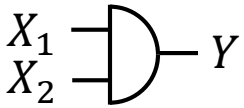
- Ein AND-Gatter sei in einer Standard Transistor-Transistor-Logik (TTL) Technologie realisiert, welche Charakteristika aufweist:

VCC	V_{IL}	V_{IH}	V_{OL}	V_{OH}
$5V$	$0.8V$	$2V$	$0.4V$	$2.4V$

- An den Eingängen X_1 und X_2 werden $0.7V$ bzw. $3.3V$ gemessen. Welcher Eingangsbelegung β entsprechen diese Spannungswerte?
- Welche Spannung U kann man am Ausgang Y erwarten?

Verzögerungszeiten von Gattern

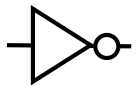
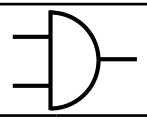
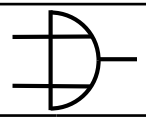
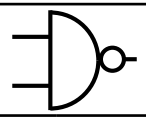
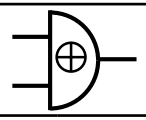
- Der Wert eines Gatters wird durch seine Eingangswerte bestimmt.
- Ändert sich einer dieser Eingangswerte, kann dies auch eine Änderung des Werts des Gatters zur Folge haben.
  - z.B.: Liegen bei einem AND-Gatter die Eingangswerte $X_1 = 0$ und $X_2 = 1$ an, so ist der Ausgangswert $Y = 0$.
    Ändert sich jedoch der Wert von X_1 auf 1, so ändert sich auch der Wert von Y auf 1.



- Aufgrund der physikalischen Eigenschaften passieren die entsprechenden Spannungsänderungen jedoch erst nach einer gewissen Verzögerungszeit.
- Chip-Hersteller geben diese Verzögerungszeiten an.

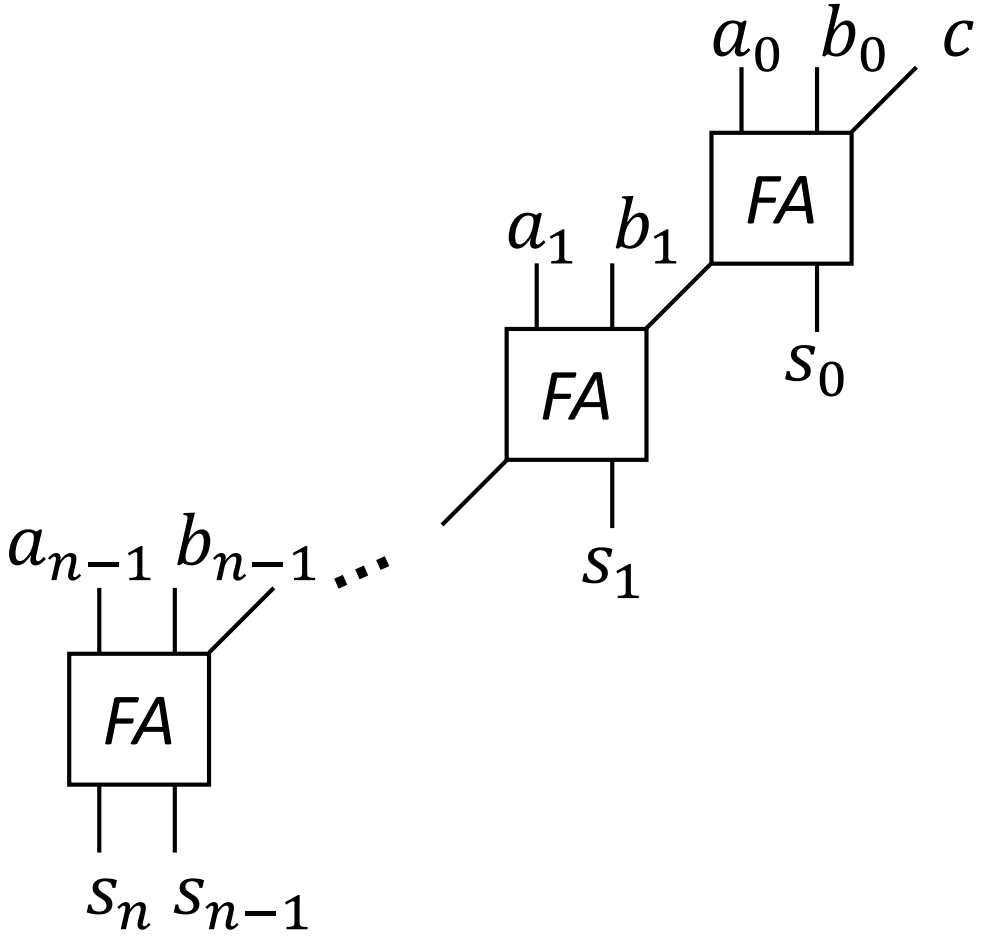
Beispiel

- Ein Hersteller von Gatter-Bausteinen spezifiert folgende Verzögerungszeiten:

										
	min	max	min	max	min	max	min	max	min	max
t_{PLH}	2.4	6.0	3.0	6.6	3.0	6.6	2.4	6.0	3.5	8.0
t_{PHL}	1.5	5.3	2.5	6.3	3.0	6.3	1.5	5.3	3.0	7.7

- t_{PLH} steht hierbei für die Zeit bei einer Änderung des Gatterwerts von 0 nach 1 und t_{PHL} für die Zeit einer Änderung von 1 nach 0.
- Wir bemerken, dass keine exakten Zeiten sondern Zeitintervalle der Form (min, max) angegeben werden.

Verzögerungszeiten in Schaltkreisen



- Sind in einem Schaltkreis mehrere Gatter hintereinander geschaltet, addieren sich deren Verzögerungszeiten.
- Die Verzögerungszeit des gesamten Schaltkreises wird bestimmt durch die maximale Höhe eines Ausgangsknoten in dem zugehörigen Graphen.

Höhe von Graphen und Schaltkreisen

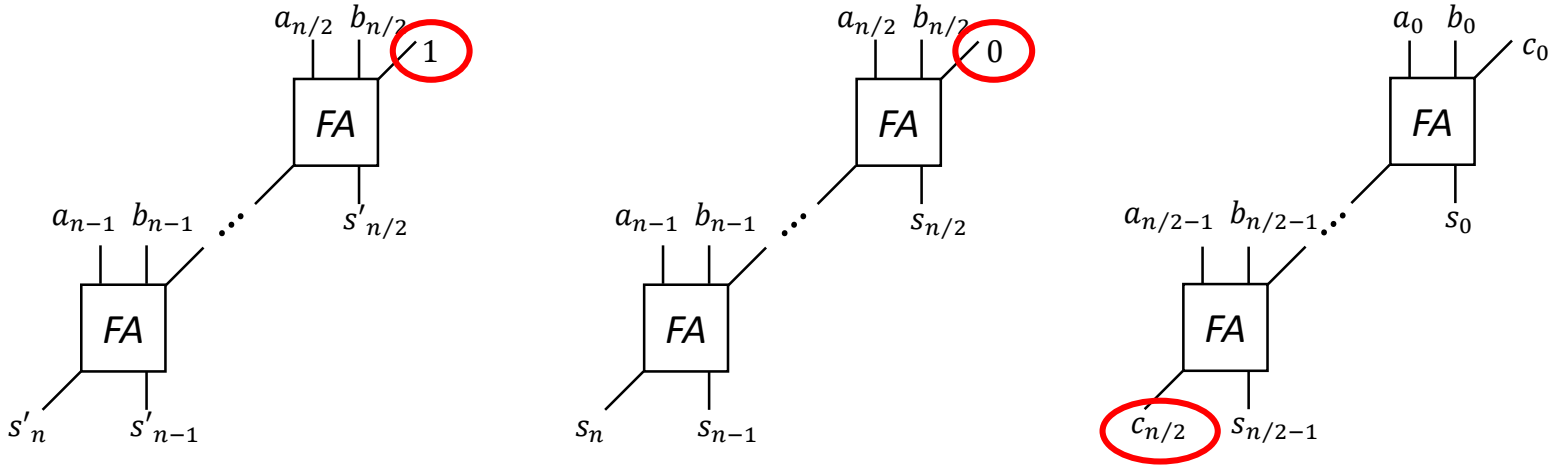
- Um im Folgenden umständliche Sprechweisen zu vermeiden, ist es sinnvoll, den Begriff der *Höhe* von Knoten auf komplette Graphen und dann auf Schaltkreise auszuweiten.
- **Definition.** Die Höhe eines Graphen $G = (V, E)$ ist die maximale Höhe eines Knotens $v \in V$.
- **Definition.** Die Höhe eines Schaltkreises $S = (X, G, g, Y)$ ist die Höhe des zugehörigen Graphen G.

Schnellere Schaltkreise

- Gibt es einen Schaltkreis, der zwei Binärzahlen schneller addiert als der Carry-Chain Addierer?
- Alternativ formuliert: gibt es einen Schaltkreis, der zwei Binärzahlen addiert, aber dessen Höhe geringer ist als die des Carry-Chain Addierers?
- Intuitiv würde man vielleicht sagen: „Nein, denn der Übertrag muss ja den gesamten Pfad von a_0, b_0, c_0 bis s_n, s_{n-1} durchlaufen.“

Verringerung der Höhe des Addierers

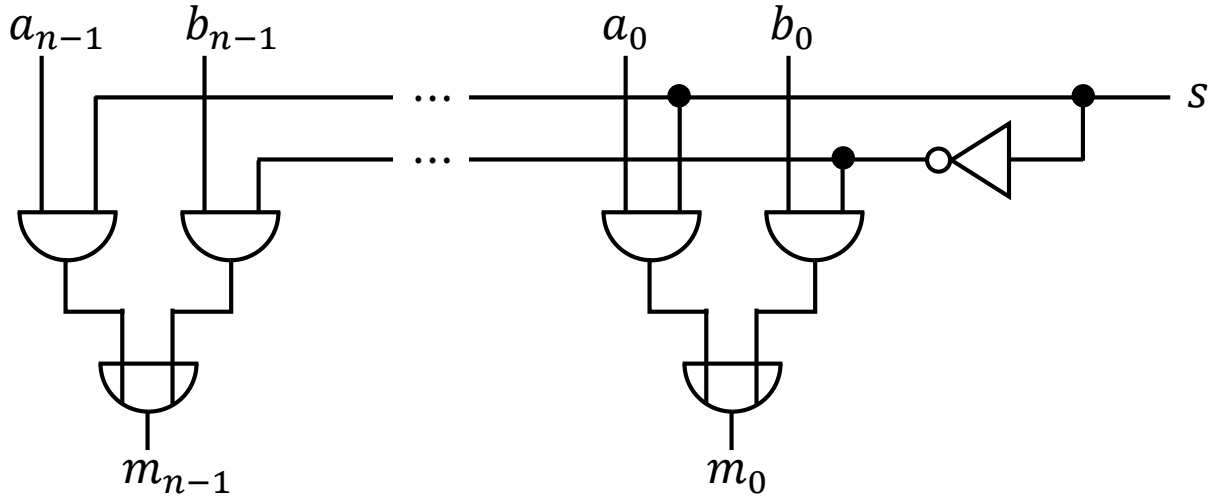
- Sei n eine gerade Zahl.



- Ist $c_{n/2} = 1$, wählen wir das linke Ergebnis ($s'_n \dots s'_{n/2}$), andernfalls das rechte ($s_n \dots s_{n/2}$), und fügen es mit $s_{n/2-1} \dots s_0$ zusammen.

n-Bit Multiplexer

- Beim Beweis des Darstellungssatzes haben wir in dem Schaltkreis auf Folie 167 im unteren Teil einen Mechanismus gesehen, der es erlaubt, zwischen zwei Signalen mittels eines Steuersignals s zu wählen.
- Setzt man diesen Mechanismus n-mal nebeneinander, ergibt sich folgender Schaltkreis:

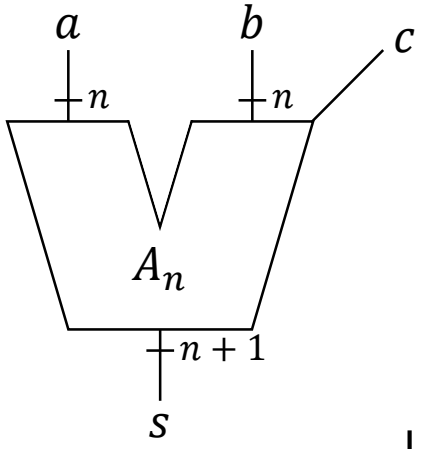


- An den Eingänge liegen zwei Bitfolgen $a = a_{n-1} \ldots a_0$ und $b = b_{n-1} \ldots b_0$ an, sowie das Steuerbit s.
- Die Ausgänge $m_{n-1} \ldots m_0$ bilden zusammen eine Bitfolge m.
- Ist $s = 1$, dann ist $m = a$; ist $s = 0$, dann ist $m = b$.

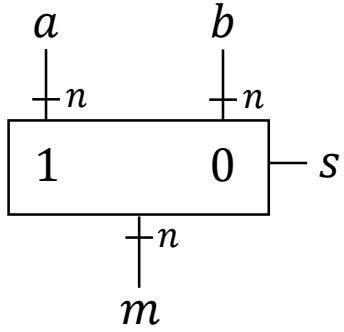
Symbolvereinbarungen

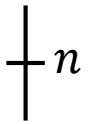
- Um das Zeichnen etwas zu vereinfachen, verwenden wir ähnlich wie bei Halb- und Volladdierer graphische Symbole für bereits bekannte Schaltkreise:

n-Bit Addierer

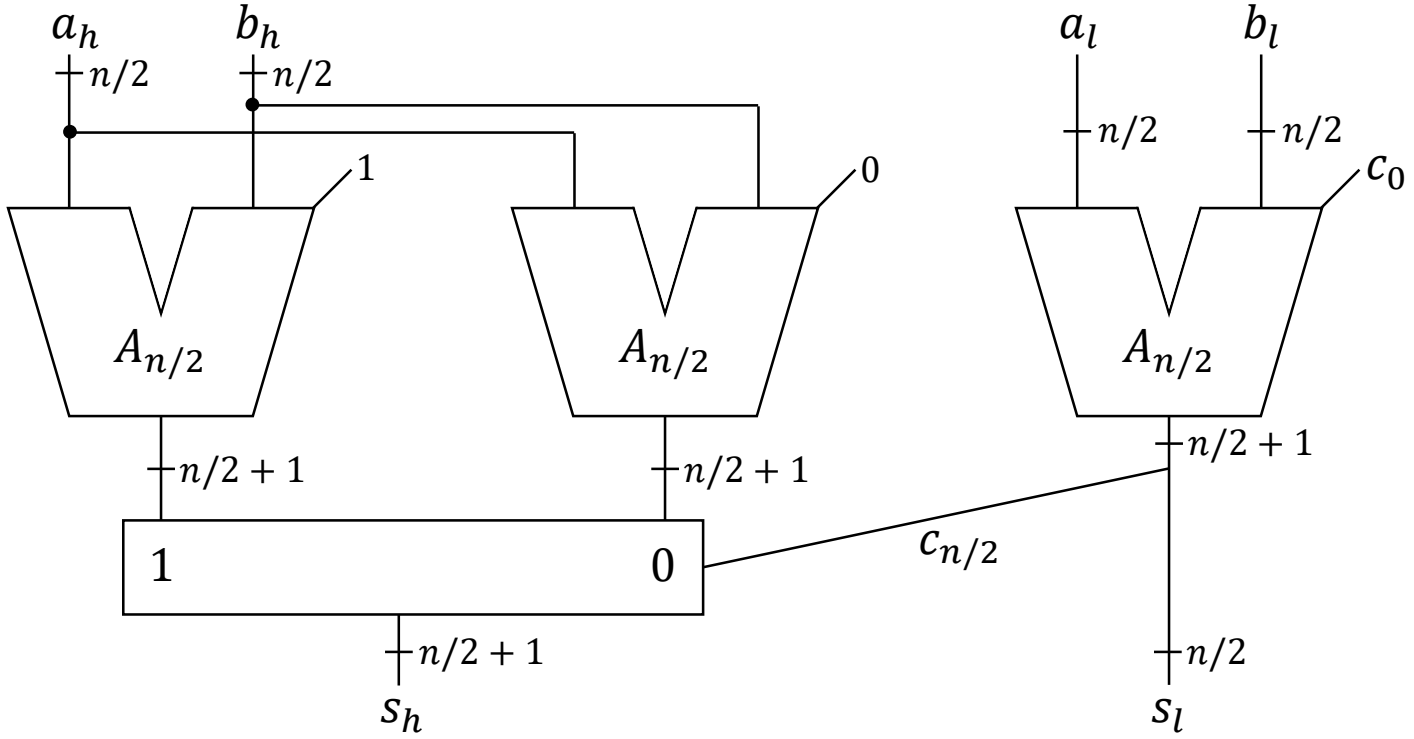


n-Bit Multiplexer



- Dabei steht die Darstellung  für n parallel verlaufende Verbindungen.

Conditional-Sum Addierer



- Hierbei seien $a_h = a_{n-1} \dots a_{n/2}$ und $a_l = a_{n/2-1} \dots a_0$ (b_h, b_l analog).

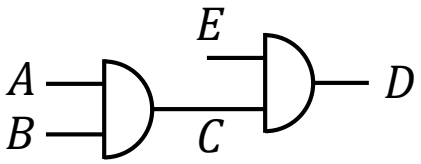
Betrachtung

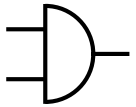
- Gegenüber dem Carry-Chain Addierer ist es uns mit dem Conditional-Sum Addierer gelungen, die Höhe des Schaltkreises etwa zu halbieren.
- Das hatte aber auch einen Preis: wir benötigen jetzt anderthalb mal so viele Gatter für die Addition und brauchen sogar noch weitere Gatter für den Multiplexer.
- Der Ausdruck “Preis” ist hierbei wörtlich gemeint: Gatter haben reale Kosten.

Timing-Analysen

- Die Verzögerungszeiten von Gattern werden von externen Faktoren beeinflusst, z.B.:
  - Fertigungsprozess des Chips, auf dem sich das Gatter befindet
  - Anzahl der Nachfolger im Graphen des Schaltkreises (kapazitive Last)
  - Betriebstemperatur
- Da wir keine exakten Verzögerungszeiten, sondern nur Minimal- und Maximalwerte des Herstellers kennen, repräsentieren wir Zeiten als Intervalle, z.B. $(2.4, 6)$.
  - Ein *exakter* Zeitpunkt t wird durch ein Interval der Form (x, x) dargestellt.
- **Definition**: Ein *Zeitinterval* ist ein Vektor $(a, b) \in \mathbb{R}^2$ mit $a \leq b$. Die Funktionen $\min, \max: \mathbb{R}^2 \to \mathbb{R}$ sind wie folgt definiert: $\min(a, b) = a$ und $\max(a, b) = b$.

Beispiel

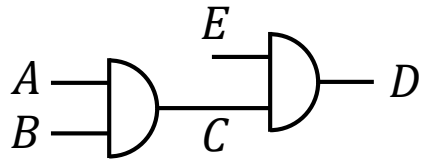


		
	min	max
t_{PLH}	3.0	6.6
t_{PHL}	2.5	6.3

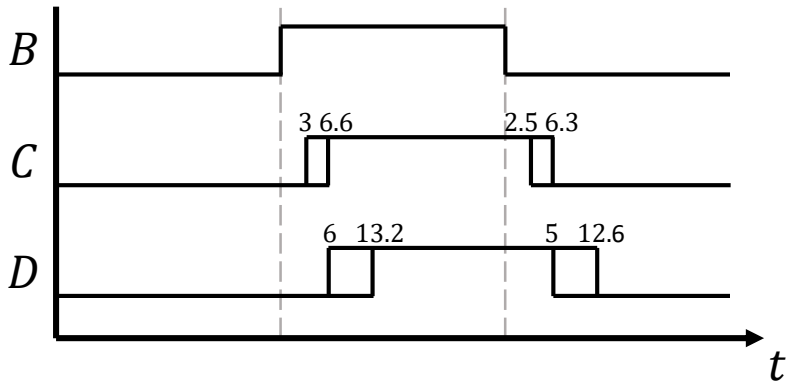
- In diesem Schaltkreis seien die Eingänge A und E beide mit 1 und Eingang B mit 0 belegt.
- Wir betrachten folgende Zeitpunkte:

Zeitpunkt	Ereignis	Verzögerungszeiten
t_0	Signal B ändert sich von 0 zu 1.	t_0
t_1	Signal C ändert sich von 0 zu 1.	$t_1 = t_0 + (3, 6.6)$
t_2	Signal D ändert sich von 0 zu 1.	$t_2 = t_1 + (3, 6.6) = t_0 + (6, 13.2)$

Detaillierte Timing-Diagramme

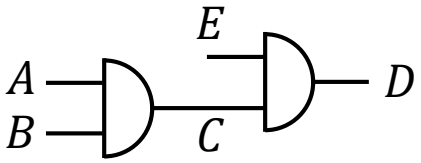


	min	max
t_{PLH}	3.0	6.6
t_{PHL}	2.5	6.3



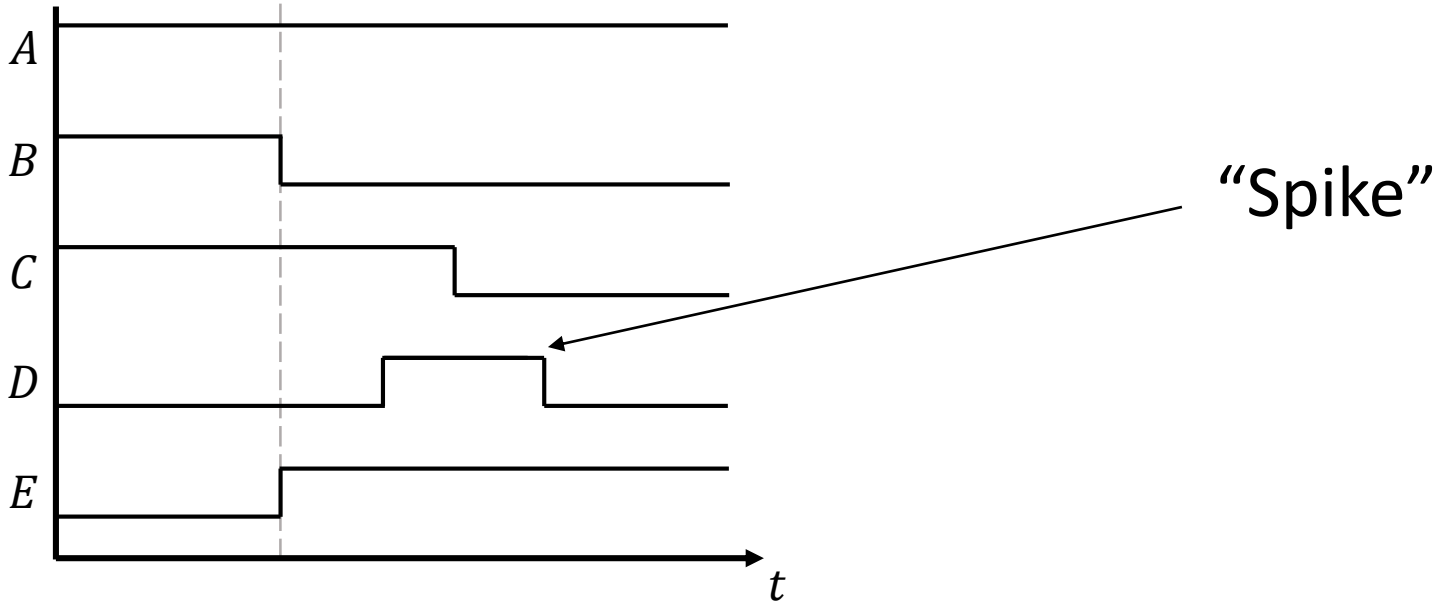
- Analysen wie die der vorherigen Folie lassen sich durch Diagramme veranschaulichen, in denen jedes (relevante) Signal auf der vertikalen Achse eingetragen wird, und die Zeit von links nach rechts verläuft.
- Bei jedem Signal steht eine unmittelbar unter dem Namen verlaufende horizontale Linie für den Wert 0, eine unmittelbar darüber verlaufende Linie für den Wert 1.

Vergleich logische / physikalische Gatter



	A	B	C	D	E
β_1	1	1	1	0	0
β_2	1	0	0	0	1

Übergang $\beta_1 \rightarrow \beta_2$

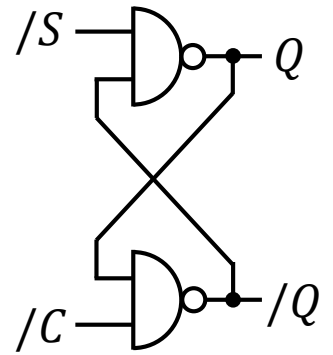


Spike-freies Umschalten von Gattern

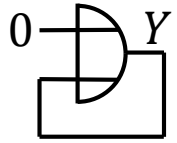
- Spikes sind ungewollte Nebeneffekte, die aufgrund der Verzögerungszeiten von Gattern beim Ändern mehrerer Eingangssignale entstehen können.
- Da sie keine Entsprechung auf logischer Seite haben, will man sie vermeiden, damit die am Ausgang eines physikalischen Gatters gemessene Spannung auch tatsächlich immer der assoziierten Schaltfunktion entspricht.
- Dies gelingt, indem man die die Eingangssignale nicht parallel sondern sequenziell mit genügend zeitlichem Abstand ändert.
- Spätere Signale werden hierbei erst dann geändert, wenn das Gatter auf alle früheren Signale bereits reagiert hat.
- Dazu muss man jeweils $\max(t_{PLH})$ bzw. $\max(t_{PHL})$ sowie die Zeiten, die zur Änderung der Signale selbst benötigt werden, berücksichtigen.

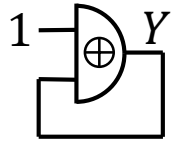
Schaltungen

- **Definition.** Eine *Schaltung* ist genauso definiert wie ein Schaltkreis, jedoch ohne das Wort “zykelfreier”.



- Problem: in Graphen mit Zyklen hat nicht jeder Knoten eine Höhe. Bei Schaltkreisen haben wir jedoch ϕ_β rekursiv über die Höhe der Knoten definiert.
- Es ist i.A. auch unklar, was der Wert eines Knotens in einem Zyklus überhaupt sinnvoll sein sollte. Beispiele:





Stabile Zustände von Schaltungen

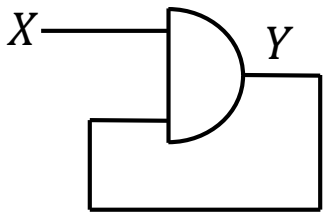
- **Definition.** Sei $S = (X, G, g, Y)$ eine Schaltung mit $G = (V, E)$ und β eine Eingangsbelegung von S. Eine Abbildung $\phi_\beta : V \to \{0, 1\}$ heißt stabiler Zustand von S, falls gilt:
  - $\phi_\beta(0) = 0$ und $\phi_\beta(1) = 1$
  - $\phi_\beta(v) = \beta(v)$ für alle Eingänge v von S.
  - Für jeden Knoten $v \in \mathrm{V}$ mit genau einem direkten Vorgänger u gilt:

$$\phi_\beta(v) = \sim\left(\phi_\beta(u)\right)$$

  - Für jeden Knoten $v \in V$ mit genau zwei direkten Vorgängern u_1, u_2 gilt:

$$\phi_\beta(v) = g(v)\left(\phi_\beta(u_1), \phi_\beta(u_2)\right)$$

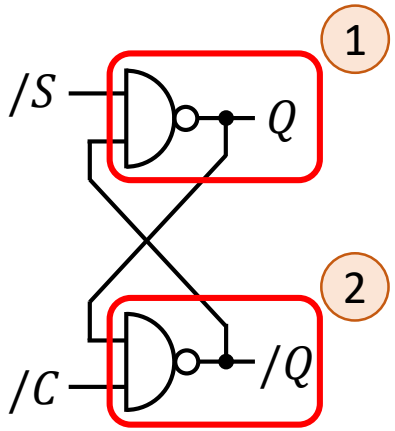
Beispiel



$\phi_\beta(X)$	$\phi_\beta(Y)$	$\phi_\beta(X) \wedge \phi_\beta(Y)$	
0	0	0	✓
0	1	0	✗
1	0	0	✓
1	1	1	✓

- Betrachten wir X und Y, gibt es genau vier mögliche Abbildungen ϕ_β.
- Da der Ausgang des AND-Gatters gleichzeitig einer seiner Eingänge ist, kann ϕ_β nur dann ein stabiler Zustand der Schaltung sein, wenn die Einträge in den letzten beiden Spalten identisch sind.
- Die zweite Zeile stellte keinen stabilen Zustand dar: nach einer gewissen Verzögerung würde sie in Zeile 1 übergehen.

R/S-Flipflop



(A) Liegt an einem Eingang eines NAND-Gatters das Signal 0 an, so ist sein Output 1.

(B) Liegt an beiden Eingängen eines NAND-Gatters 1 an, so ist sein Output 0.

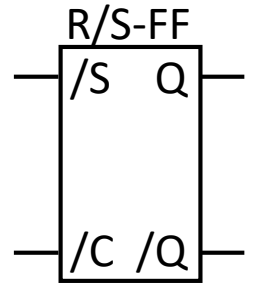
Wir betrachten alle möglichen Abbildungen ϕ_β: von diesen kommen nur diejenigen als stabile Zustände in Frage, die den Gatter-Eigenschaften (A) und (B) nicht widersprechen.

Tabelle aller möglichen Abbildungen ϕ_β.

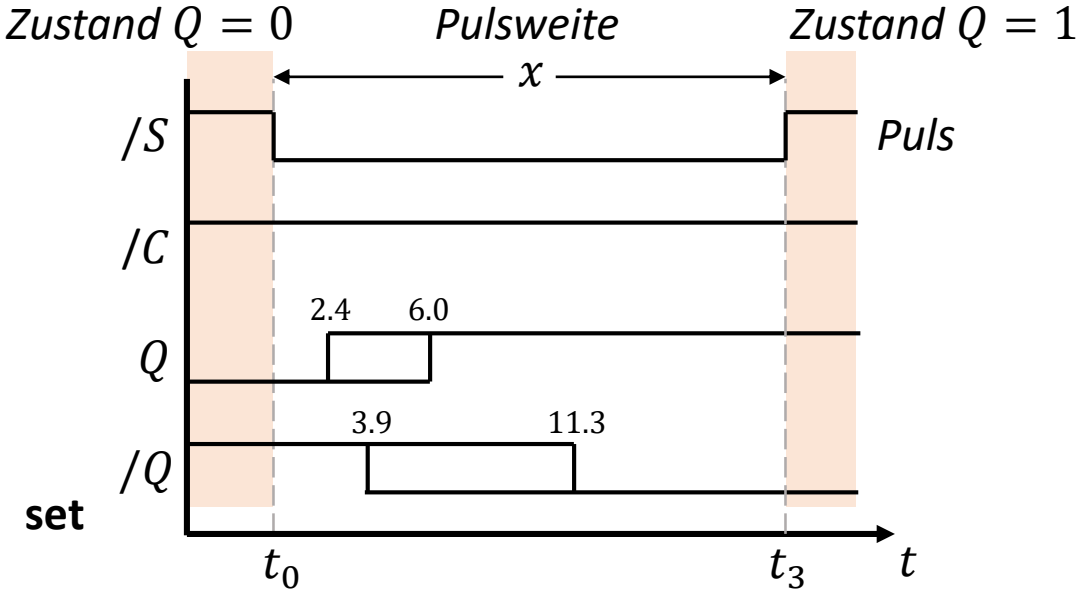
$\phi_\beta(/S)$	$\phi_\beta(/C)$	$\phi_\beta(Q)$	$\phi_\beta(/Q)$	Widerspruch zu:
0	0	0	0	✗ 1A 2A
0	0	0	1	✗ 1A
0	0	1	0	✗ 2A
0	0	1	1	✓
0	1	0	0	✗ 1A 2A
0	1	0	1	✗ 1A
0	1	1	0	✓
0	1	1	1	✗ 2B
1	0	0	0	✗ 1A 2A
1	0	0	1	✓
1	0	1	0	✗ 2A
1	0	1	1	✗ 1B
1	1	0	0	✗ 1A 2A
1	1	0	1	✓
1	1	1	0	✓
1	1	1	1	✗ 1B 2B

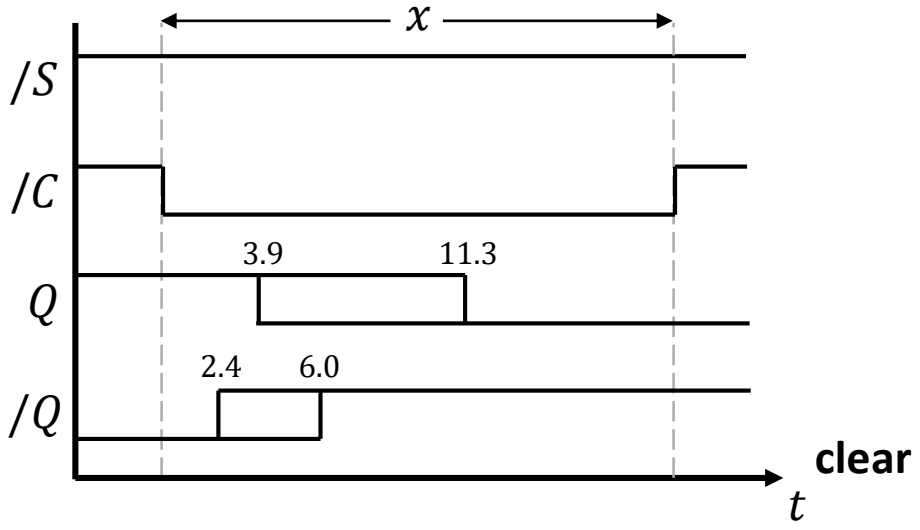
Zustandswechsel beim R/S-Flipflop

- Wir betrachten zwei der möglichen stabilen Zustände näher:
  - "Zustand $Q = 0$": $/S = 1,\ /C = 1,\ Q = 0,\ /Q = 1$
  - "Zustand $Q = 1$": $/S = 1,\ /C = 1,\ Q = 1,\ /Q = 0$
- Wie kann man einen dieser Zustände in den anderen überführen?



Schaltsymbol





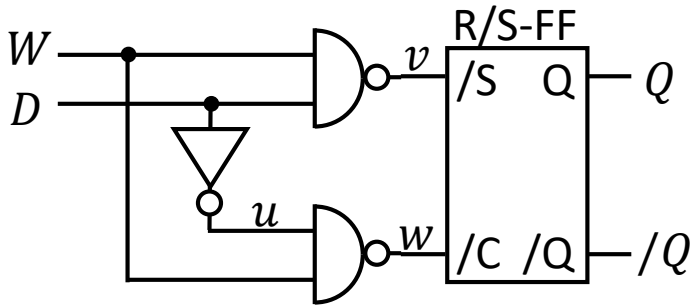
Betrachtung

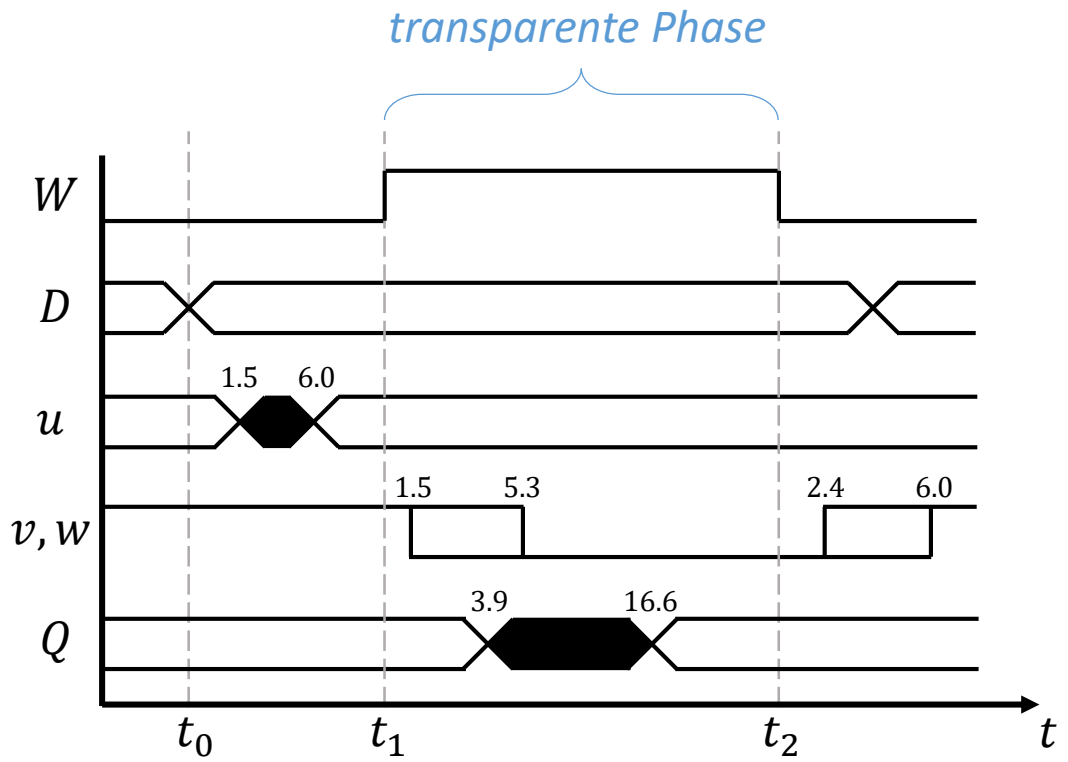
- Das R/S-Flipflop stellt bereits eine einfache Speicherzelle dar.
- In ihm kann man ein Bit speichern:
  - 1 mit set
  - 0 mit clear
- Den gespeicherten Wert kann man anschließend am Ausgang Q auslesen.
- In vielen Situationen ergibt sich der zu speichernde Wert aber erst in Abhängigkeit von anderen Operationen.
  - z.B. die Ausgänge eines Volladdierers nach einer Addition.
- Es wäre also wünschenswert, R/S-Flipflops so zu erweitern, dass automatisch entweder set oder clear ausgeführt wird, je nachdem ob der zu speichernde Wert 1 oder 0 ist.

Erweiterung des R/S-Flipflops

- Idee: wir benötigen einen vorgeschalteten Schaltkreis, der dafür sorgt, dass ein *set* ausgeführt wird, falls der zu speichernde Wert 1 ist, und ein *clear*, falls 0 ist.
  - Wir nennen den Wert ab sofort D für „Datum“ (Singular von „Daten“).
- In beiden Fällen benötigen wir dazu einen Puls: entweder auf $/S$ oder auf $/C$.
- Das motiviert die folgende Konstruktion: wir entwerfen einen neuen Baustein mit zwei Eingängen – einen für den Puls, und einen weiteren, der diesen Puls je nach Wert zu $/S$ oder zu $/C$ lenkt.

D-Latch





- Die Verzögerungszeiten t_{PHL} und t_{PLH} eines Gatters sind nicht unbedingt identisch.
- Um nicht immer zwei verschiedene Timing-Diagramme zeichnen zu müssen, kann man beide Zeiten zu einem einzigen Intervall $(\min(t_{PHL}, t_{PLH}), \max(t_{PHL}, t_{PLH}))$ kombinieren.
  - z.B. AND-Gatter: $(2.5, 6.6)$, Inverter: $(1.5, 6.0)$, …

Vergleich R/S-Flipflop und D-Latch

- R/S-Flipflop und D-Latch sind nicht vollkommen unterschiedliche Dinge, denn das D-Latch baut ja auf dem R/S-Flipflop auf.
- Beide dienen dazu, ein Bit zu speichern.
- Beim R/S-Flipflop muss je nachdem, ob eine 0 oder eine 1 gespeichert werden soll, eine andere Aktion ausgeführt werden (clear oder set)
- Beim D-Latch ist das nicht der Fall: wir müssen lediglich einen Puls auf Eingang W senden, dann wird der aktuelle Wert auf D gespeichert.
- Technischer Unterschied:
  - die Eingangssignale beim R/S-Flipflop sind *active low*, d.h. die Signale werden durch *Absenken* ($1 \rightarrow 0$) aktiviert (Konvention: Signalname beginnt mit Schrägstrich).
  - das Schreibsignal W beim D-Latch ist hingegen *active high*.

Speichern ohne Puls

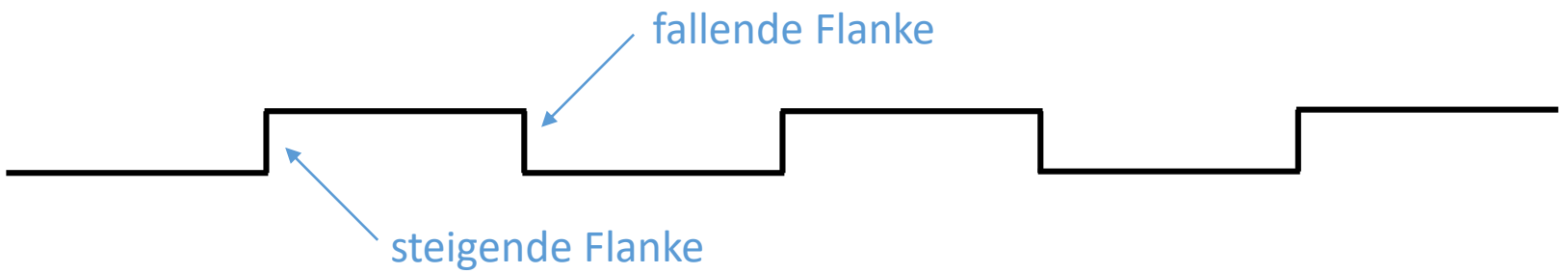
- Beim Senden eines Puls an ein D-Latch ist das Timing wichtig, um spikefrei umzuschalten.
- Wie sendet man eigentlich einen Puls im Nanosekundenbereich?
- Idealerweise sollte sich der Benutzer eines Speicherbausteins nicht darum zu kümmern brauchen.
- Idee: Pulse automatisch erzeugen.
- Problem: man bräuchte einen Puls immer dann, wenn sich das Signal D verändert hat – aber woher soll man wissen, wann dies der Fall ist?
- Ansatz: Pulse ständig und in regelmäßigen Abständen erzeugen

Puls und Taktsignal (clock signal)

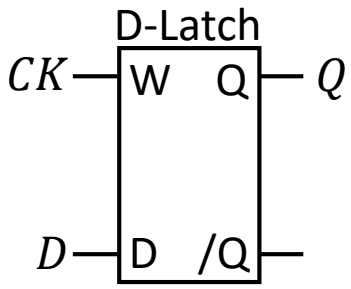
- Active high-Puls

- Active low-Puls

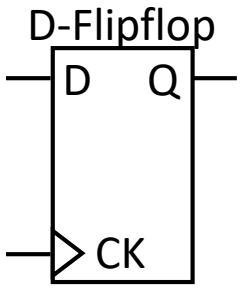
- Taktsignal



D-Flipflop



Aufbau

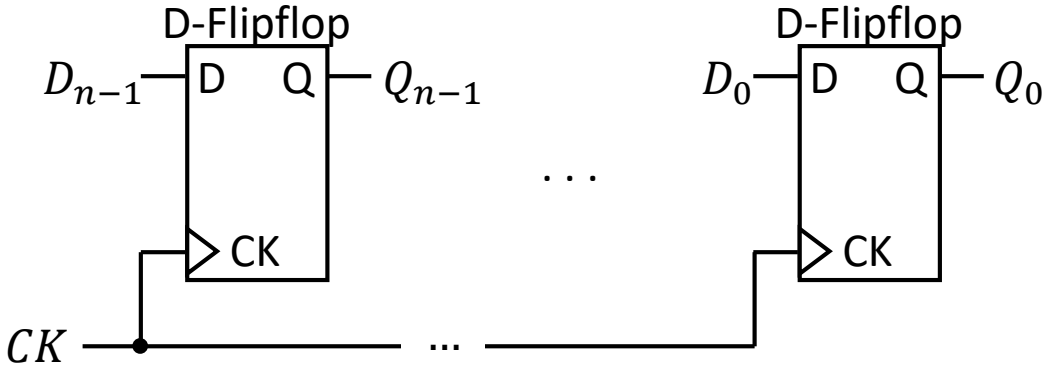


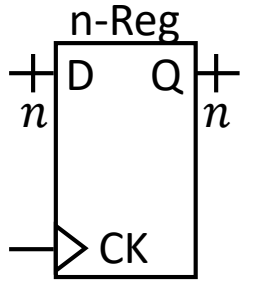
Schaltsymbol

- Im Gegensatz zum D-Latch hat das D-Flipflop keine transparente Phase: der Wert D wird immer nur zum Zeitpunkt einer steigenden Flanke geschrieben.

Register

- Alle obigen Speicherzellen speichern jeweils nur ein einzelnes Bit.
- Wollen wir eine n-Bit-Zahl speichern, hindert uns aber niemand daran, n gleiche Speicherzellen parallel einzusetzen.
- Bei D-Flipflops verwenden wir dabei sinnvollerweise bei allen dasselbe Taktsignal:



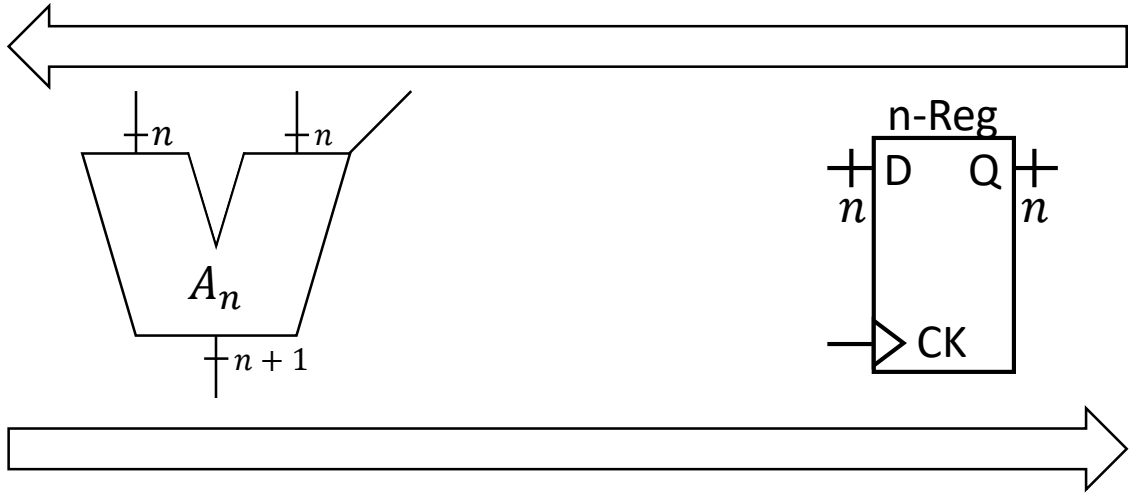


Schaltsymbol

- Diese Konstruktion nennt man ein n-Bit Register.

Betrachtung

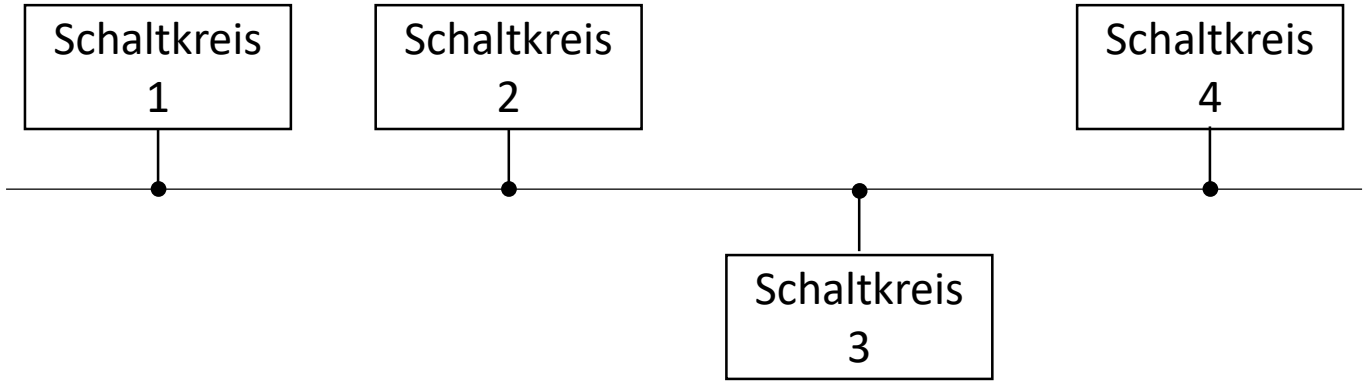
- Wir haben nun Schaltungen zum Rechnen (Addierer) und zum Speichern (Register) von Daten kennengelernt.



- Ein sinnvoller nächster Schritt wäre, beide Techniken zu verbinden, um mit zuvor gespeicherten Daten rechnen und um Ergebnisse von Berechnungen speichern zu können.

Bus

- Ein Bus ist eine Datenleitung, die Ein- oder Ausgänge mehrerer Schaltkreise miteinander verbindet, z.B.:



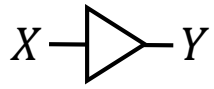
- Ist der Ausgang eines Schaltkreises mit dem Bus verbunden, sagt man: „der Schaltkreis schreibt auf den Bus“. Ist der Eingang verbunden, sagt man: „der Schaltkreis liest von dem Bus.“
- Ohne Weiteres würde dies im Allgemeinen aber zu einem Kurzschluss führen, den man auch als „bus contention“ bezeichnet.

Bus Contention vermeiden

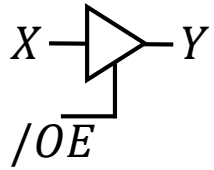
- Die Kurzschlussproblematik kann vermieden werden, indem man dafür sorgt, dass nie mehr als ein Schaltkreis zum selben Zeitpunkt Daten auf den Bus schreibt oder von ihm liest.
- Zu jedem Zeitpunkt darf also höchstens ein Schaltkreis schreiben und ein Schaltkreis lesen.
- Um dies zu realisieren, benötigt man eine Art „Schalter“ zwischen Schaltkreis und Bus.

Treiber

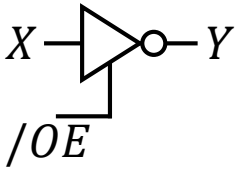
- Ein *Treiber* ist eine Schaltung, die das Eingangssignal verstärkt.
- Ein *Tristate-Treiber* ist ein Treiber, der neben dem Eingangssignal noch über ein Steuersignal $/OE$ verfügt, das den Treiber in einen von zwei möglichen Zuständen schaltet:
  - $/OE = 0$: am Ausgang liegt das verstärkte Eingangssignal an
  - $/OE = 1$: Ausgang und Eingang sind voneinander isoliert



Schaltsymbol Treiber



Schaltsymbol Tristate-Treiber



Schaltsymbol invertierender Tristate-Treiber

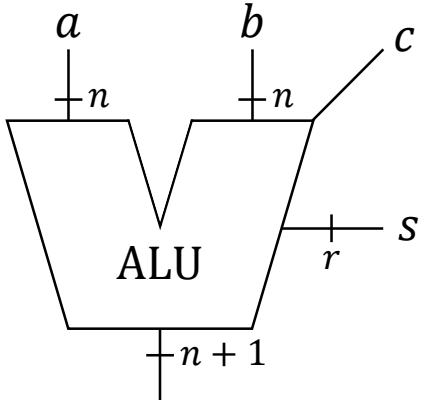
Arithmetic Logic Unit (ALU)

- Ein Informatiksystem kann typischerweise nicht nur addieren, sondern noch eine Reihe anderer arithmetischer oder logischer Operationen durchführen, z.B.:
  - Subtraktion
  - Multiplikation
  - Shifts (Verschieben von Bitfolgen nach links oder rechts)
  - Bitweise logische Operationen ($\vee$, $\wedge$, $\neg$, $\oplus$, etc.)
- Man fasst die entsprechenden Schaltkreise gerne zusammen und bezeichnet sie als ALUs.

Beispiel

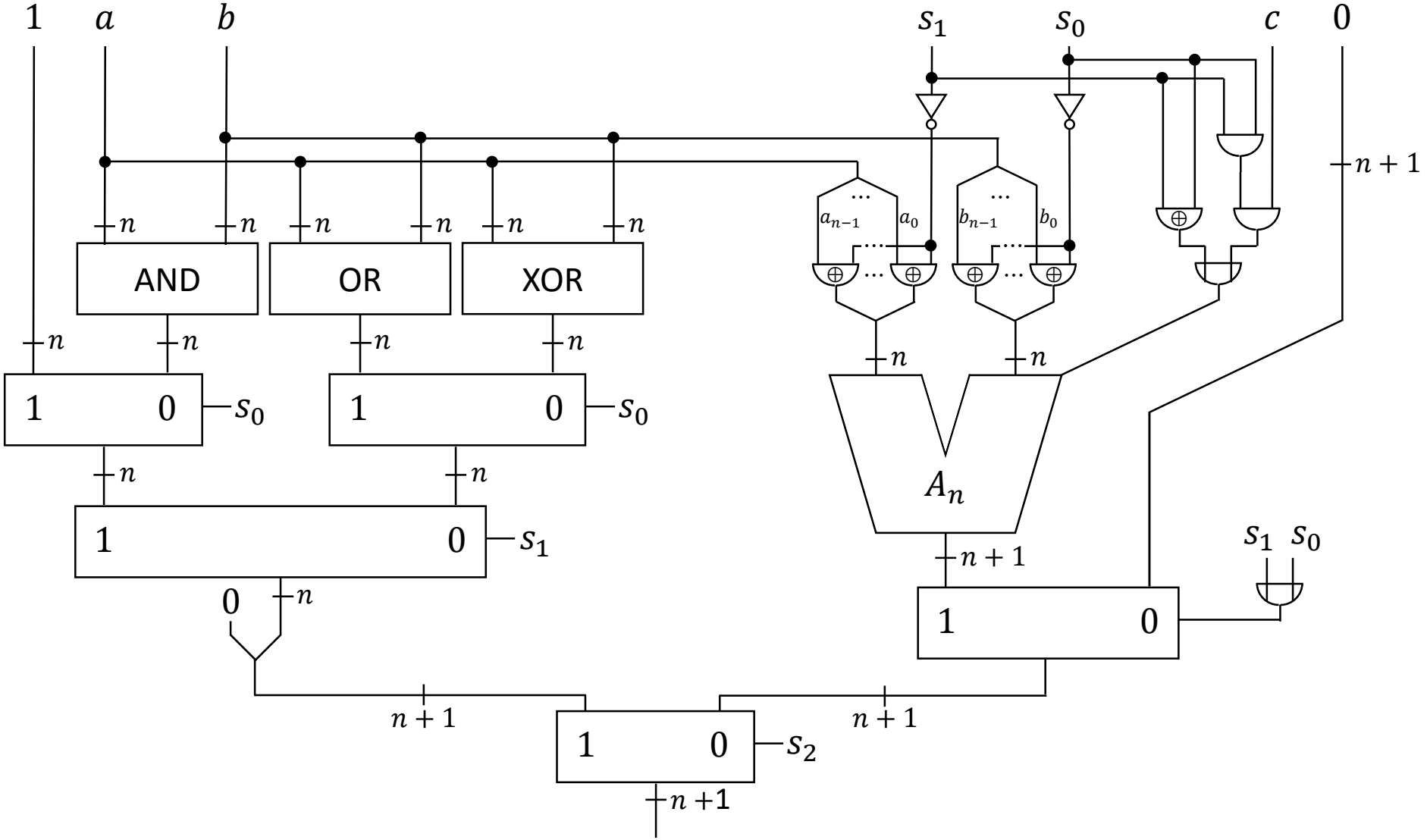
- Eine einfache ALU berechne die folgenden acht Funktionen:

$f_{\langle s \rangle}$	
$0 \ldots 0$ (n-mal)	Konstante
$b - a$	Arithmetische Operationen
$a - b$	
$a + b$	
$a \oplus b$	Bitweise logische Operationen
$a \vee b$	
$a \wedge b$	
$1 \ldots 1$ (n-mal)	Konstante



Schaltsymbol einer *Arithmetic Logic Unit*.

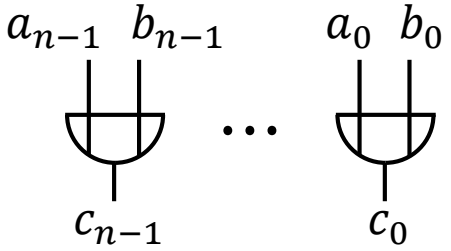
Realisierung der Beispiel-ALU

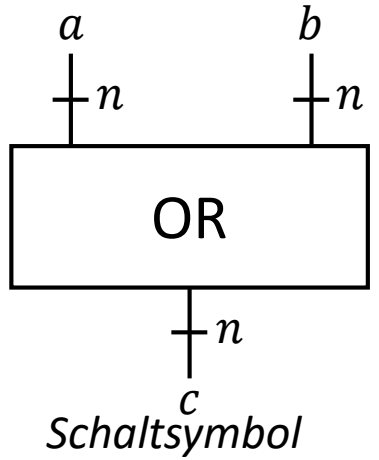


$s_2 s_1 s_0$	$f_{\langle s \rangle}$
000	$0 \ldots 0$
001	$b - a$
010	$a - b$
011	$a + b$
100	$a \oplus b$
101	$a \vee b$
110	$a \wedge b$
111	$1 \ldots 1$

n-fache bitweise logische Operationen

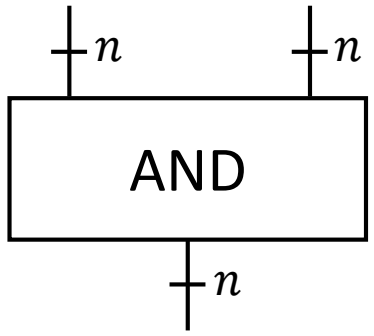
- Der nebenstehende Schaltkreis berechnet für zwei Eingabefolgen aus n Bits, die wir mit $a = a_{n-1} \dots a_0$ und $b = b_{n-1} \dots b_0$ bezeichnen, eine *bitweise Disjunktion*, d.h. eine Ausgabefolge $c = c_{n-1} \dots c_0$, bei der gilt: $c_i = a_i \vee b_i$ für alle $0 \leq i < n$.

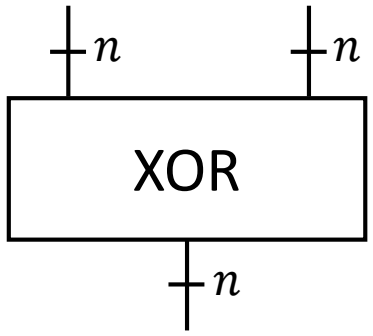


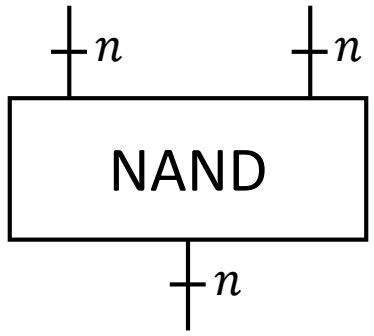


Schaltsymbol

- Analog konstruiert man Schaltkreise zur Berechnung anderer n-fachen bitweisen Operationen:







Komplexere Berechnungen

- Mit einer ALU können wir also verschiedene Operationen auf zwei Operanden ausführen.
- Um eine komplexe Berechnung wie z.B. $(4 + 2) - (1 + 8)$ zu durchzuführen, könnten wir diese also in mehrere Teiloperationen aufspalten (im Beispiel: zwei Additionen, eine Subtraktion).
- Zwischenergebnisse (hier: $(4 + 2)$ und $(1 + 8)$) könnten wir Speicherzellen, z.B. Registern ablegen und von dort wiederverwenden.
- Uns fehlt aber noch eine Möglichkeit, mehrere Rechnungen automatisch hintereinander auszuführen zu lassen.

Prozessoren

- Prozessoren bilden das Herzstück von Informatiksystemen. Sie ermöglichen dem Benutzer eine gezielte Informationsverarbeitung durch Interaktion mit Datenspeichern.
  - Umgang mit Daten: Speichern, Zugreifen, Löschen
  - Operationen mit Daten: Rechnen, Transformieren, Kombinieren, Extrahieren
- Die gewünschte Arbeitsweise eines Prozessors wird diesem in Form von Befehlsfolgen einprogrammiert.
- Man unterscheidet zwischen spezialisierten Prozessoren (z.B. Graphikprozessoren, GPU) und Hauptprozessoren (CPU).

Befehle / Instruktionen

- Die verschiedenen Komponenten der CPU bieten eine Reihe unterschiedlicher Funktionalitäten an.
  - In Speicherzellen können Daten gespeichert werden.
  - Über Busse können Daten transportiert werden.
  - Mit einer ALU können Operationen auf Daten ausgeführt werden.
- Durch die Kombination dieser Funktionalitäten ergeben sich verschiedene Operationen, die mittels der CPU ausgeführt werden können, z.B. Daten kopieren, addieren, vergleichen, etc.
- Ein *Befehl* kodiert eine konkrete solche Operation, die von der CPU ausgeführt werden soll, in Form einer Bitfolge fester Länge.

Programmspeicher

- Bislang haben wir uns mit Datenspeichern beschäftigt, wobei Daten Bitfolgen waren, die entweder als Binär- oder Zweierkomplementzahlen interpretiert wurden.
- Beobachtung: Befehle sind ebenfalls Bitfolgen, also Daten.
- Um einem Prozessor also Befehlsfolgen zur Verfügung zu stellen, benötigen wir dann lediglich:
  - einen ausreichend großen Datenspeicher, um alle gewünschten Befehle dort zu speichern → *Programmspeicher*
  - die Information, welcher Befehl aus dem Programmspeicher ausgeführt werden soll
  - eine Möglichkeit, den aktuellen Befehl in eine konkrete Eingangsbelegung der CPU zu übersetzen

Beispiel Programmspeicher

Adressen	Befehle
0	001001
1	000100
2	000101
3	001001

- Die Speicherzellen des Programmspeichers werden einfach durchnummeriert.
- Man nennt die Nummer einer Speicherzelle dessen *Adresse*.

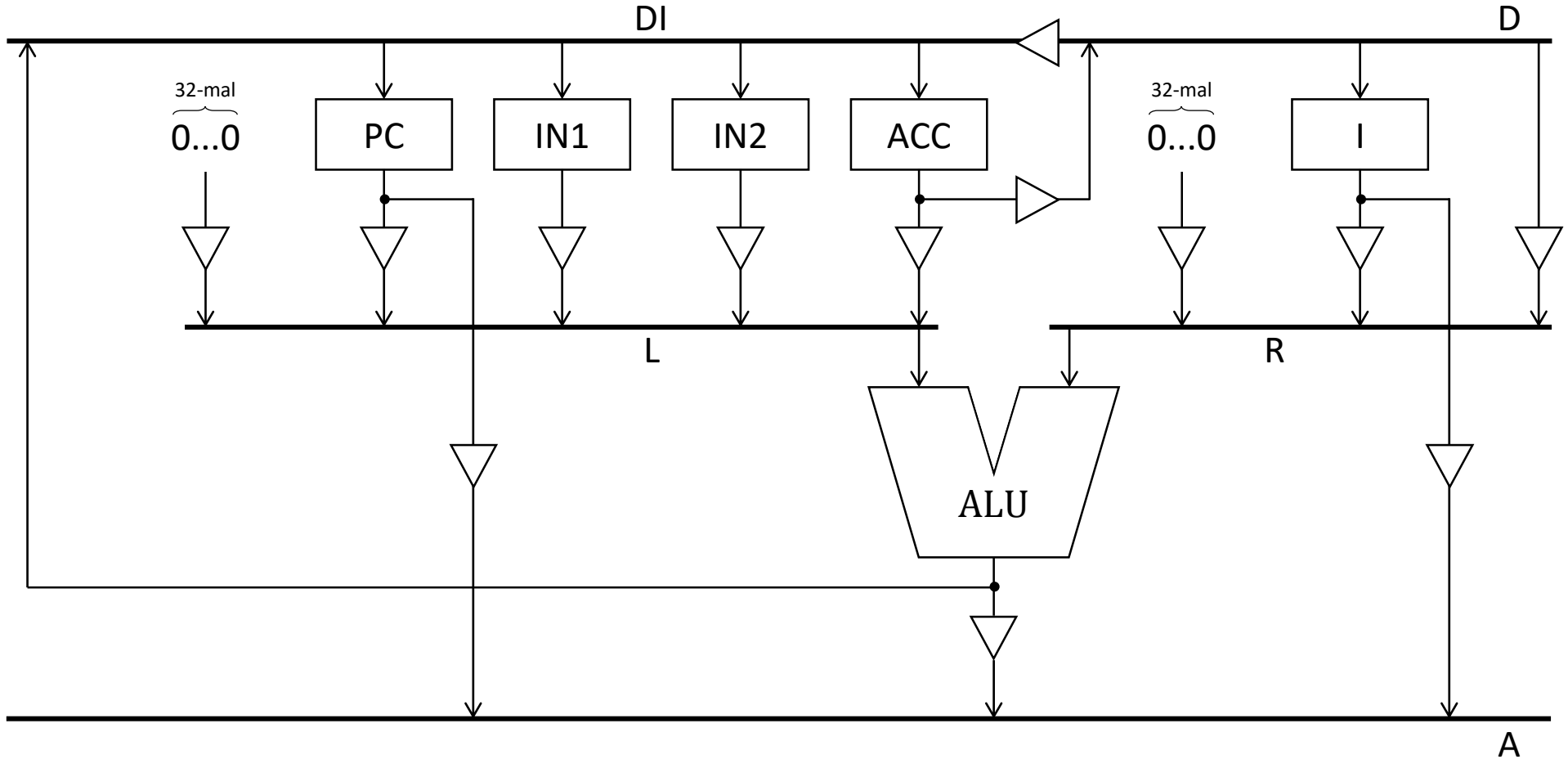
Datenspeicher

- In Informatiksystemen kommen verschiedene Arten von Speichern zum Einsatz.
- Register: direkt im Prozessor verbaut, feste Bitgröße, geringe Anzahl, sehr schnell im Zugriff (auslesen, schreiben)
- SRAM (static random access memory), Stromverbrauch abhängig vom Einsatz, Herstellung relativ teuer, Verwendung in schnellen Pufferspeichern (Caches)
- DRAM (dynamic random access memory), günstiger in der Herstellung, benötigen aber in der Anwendung kontinuierliche Wiederauffrischung (ansonsten Datenverlust), die den Zugriff langsamer macht
- Massenspeicher (z.B. Festplatten, SSD), im Ggs. zu obigen Datenspeichern nichtflüchtig, d.h. Daten bleiben auch ohne Stromversorgung erhalten
- ROM (read-only-memory)

Von Neumann-Architektur

- Daten- und Programmspeicher sind prinzipiell gleicher Art
  - Sie speichern beide Bitfolgen einer bestimmten Länge in Speicherzellen.
  - Der Zugriff erfolgt über einen Adressbus (zur Auswahl der gewünschten Zelle) und über einen Datenbus (zum Auslesen/Schreiben der Daten dieser Zelle).
- Es ist daher naheliegend, sie in der Praxis als Teile eines einzigen größeren Speichers zu realisieren.
- Dennoch waren beide in frühen Rechnerarchitekturen getrennt.
- Die Von-Neumann-Architektur (nach John von Neumann), in der beide Speicher zusammengeführt wurden, wurde als großer Fortschritt aufgenommen.

Beispiel: eine einfache CPU



Busse

- D: Datenbus (verbunden mit CPU-externem RAM-Speicher S)
- A: Adressbus (verbunden mit CPU-externem RAM-Speicher S)
- DI: Interner Datenbus
- L: linker Operand der ALU
- R: rechter Operand der ALU

Programmzähler PC

- Um zu wissen, welcher Befehl als nächstes ausgeführt werden soll, speichert man dessen Adresse im Programmspeicher in einem speziellen Register (*PC = Program Counter*).
- Alle Register unserer CPU speichern Bitfolgen der Länge 32.
- In der Regel soll ein Befehl nach dem anderen ausgeführt werden: dies entspricht dann einfach einem Hochzählen des PC-Werts.
  - Dazu könnte man einfach einen Addierer nehmen, bei dem Eingang fest mit der Zahl 1 verbunden ist, und dessen andere Eingang aus dem PC gelesen wird. Ist diese Addition durchgeführt, kann der neue Wert wieder in den PC geschrieben werden.

Clock

- Wie wir wissen, spielt das Timing bei Schaltungen eine wichtige Rolle.
- Aufgrund der Verzögerungszeiten und um Spikes zu vermeiden, dauert es immer eine gewisse Zeit, bis das Ergebnis einer Berechnung oder eines Lese-/Schreibvorgangs vorliegt.
- Beim Hochzählen des PCs muss dies berücksichtig werden: wir dürfen nicht zum nächsten Befehl übergehen, bevor der vorherige vollständig beendet ist.
- Zur zeitlichen Synchronisation verwenden wir das Taktsignal, das bereits bei Registern zum Einsatz kommt.
- Schaltzeiten von Befehlen werden auf eine ganzzahlige Anzahl von *Taktzyklen* (die Zeit von einer steigenden Flanke des Taktsignals bis zur nächsten) aufgerundet.

Indexregister IN1 und IN2

- Wir nehmen an, dass alle Daten, die mit der CPU verarbeitet werden sollen, sich in einem externen RAM-Speicher S befinden.
- Mittels Adress- und Datenbus können die Daten aus bestimmten Zellen in die Register der CPU geladen werden.
- Die Adresse der jeweiligen Zelle kann als Teil des Lade-Befehls spezifiert oder den Registern *IN1* oder *IN2* entnommen werden.
  - Das ist praktisch, wenn die genaue Adresse z.B. erst durch die CPU berechnet werden muss.
- Wie diese Mechanismen im Detail aussehen, folgt später.

Akkumulator ACC

- Das Akkumulator-Register *ACC* ist ein zentrales Register bei Berechnungen mit der ALU.
- Es kann sowohl Eingabedaten (linker Operand) enthalten, als auch das Register sein, in dem das Ergebnis der ALU gespeichert wird.
- Es dient auch zur Übertragung von Daten in den externe RAM-Speicher S.
  - Das heißt, Daten die in das RAM übertragen werden sollen, müssen dazu erst in den Akkumulator geschrieben werden.

Instruktionsregister I

- Im Instruktionsregister *I* befindet sich der aktuell auszuführende Befehl. Ist dieser verarbeitet, wird der nächste[1)] Befehl geladen.
- Wir können also zwei prinzipielle Phasen unterscheiden :
  - „Fetch“: In dieser Phase wird der nächste Befehl aus dem Programmspeicher in das Instruktionsregister geladen.
  - „Execute“: In dieser Phase wird der Befehl, der sich zu diesem Zeitpunkt im Instruktionsregister befindet, ausgeführt.

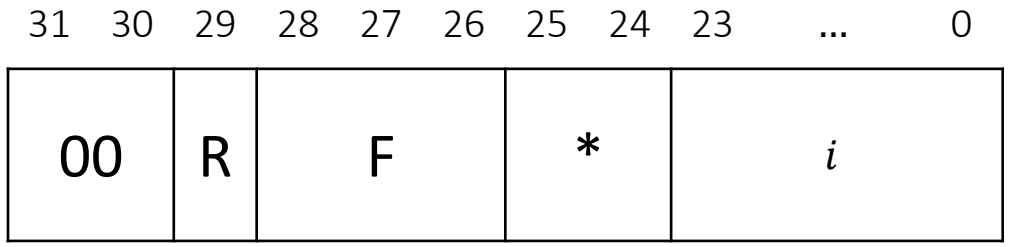
1) siehe Programmzähler PC

Instruktionssatz

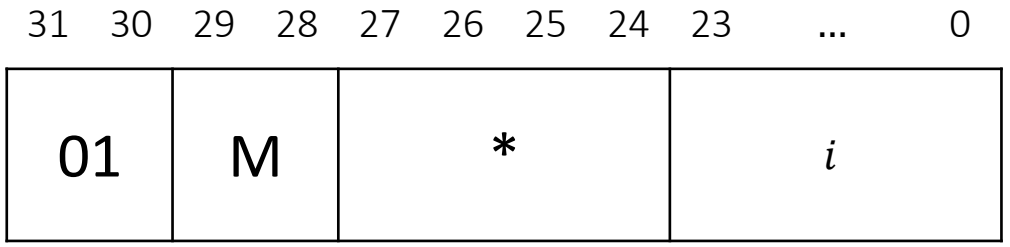
- Unsere CPU unterstützt vier Arten von Befehlen:
  1. Ladebefehle zum Transfer von Daten aus dem RAM in den ACC
  2. Speicherbefehle zum Transfer von Daten aus dem ACC in das RAM
  3. Befehle zum Durchführen von ALU-Operationen
- All diese Instruktionen erhöhen außerdem den PC um 1.
- Um flexibel zwischen verschiedenen Programmteilen hin- und herwechseln zu können gibt es außerdem noch:
  4. Sprungbefehle, um den PC anderweitig zu setzen, entweder unbedingt oder in Abhängigkeit von dem Zustand des ACC.

Instruktionsformat

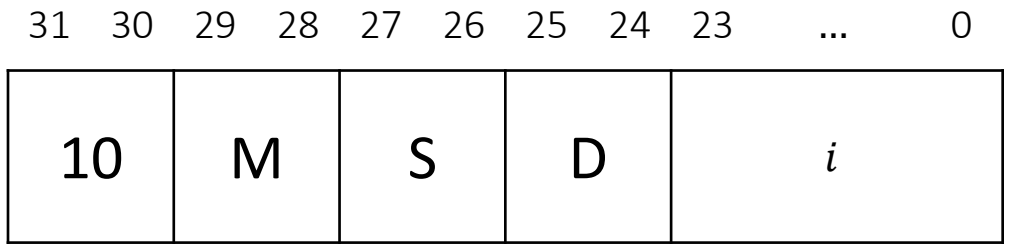
- Da das Instruktionsregister wie die anderen Register der CPU 32 Bits speichern kann, werden alle Befehle durch Folgen von 32 Bits repräsentiert.



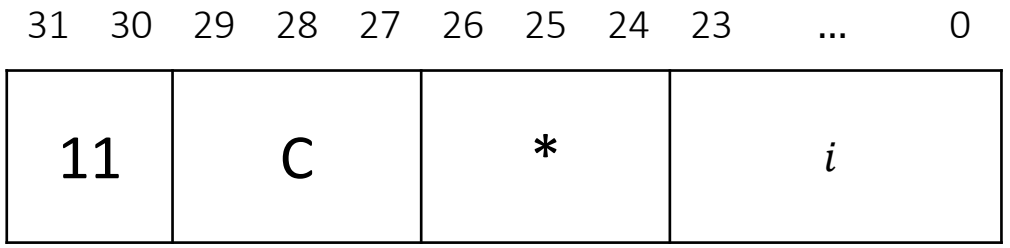
ALU-Operationen



Ladebefehle



Speicherbefehle



Sprungbefehle

Transfer von Daten in die CPU

M	**Befehl**	**Effekt**
00	`LOAD` i	$ACC := S(i)$ $PC := \langle PC \rangle_2 + 1$
Der Wert der i-ten Speicherzelle des RAM wird in den Akkumulator kopiert.		
01	`LOADIN1` i	$ACC := S(\langle IN1 \rangle_2 + i)$ $PC := \langle PC \rangle_2 + 1$
Der Wert der RAM-Speicherzelle mit der Adresse $\langle IN1 \rangle_2 + i$ wird in den Akkumulator kopiert.		
10	`LOADIN2` i	$ACC := S(\langle IN2 \rangle_2 + i)$ $PC := \langle PC \rangle_2 + 1$
Der Wert der RAM-Speicherzelle mit der Adresse $\langle IN2 \rangle_2 + i$ wird in den Akkumulator kopiert.		
11	`LOADI` i	$ACC := i$
Der Wert i wird in den Akkumulator geschrieben.		

- Befehle, deren Parameter nicht aus dem RAM gelesen, sondern direkt Teil des Instruktionsformats sind, nenne man *Immediate-Befehle* (z.B. LOADI)

Notation in den Instruktionstabellen

- In den Tabellen finden sich viele Ausdrücke der Form $A := B$
- Dabei kommen in A und B oft die Namen der CPU-Register $(PC, IN1, IN2, ACC)$ sowie der des RAM-Speichers $\big(S(\dots)\big)$ vor.
- Die Bedeutung eines solchen Ausdrucks ist, dass *nach* dem Ausführen des Befehls der Wert von A derselbe ist wie der von B *vor* dem Ausführen des Befehls war.

Transfer von Daten aus der und innerhalb der CPU

M	**Befehl**	**Effekt**
00	STORE i	$S(i) \coloneqq ACC$ $PC \coloneqq \langle PC \rangle_2 + 1$
Der Wert des Akkumulators wird in die i-te Speicherzelle des RAM kopiert.		
01	STOREIN1 i	$S(\langle IN1 \rangle_2 + i) \coloneqq ACC$ $PC \coloneqq \langle PC \rangle_2 + 1$
Der Wert des Akkumulators wrd in die Speicherzelle mit der Adresse $\langle IN1 \rangle_2 + i$ kopiert.		
10	STOREIN2 i	$S(\langle IN2 \rangle_2 + i) \coloneqq ACC$ $PC \coloneqq \langle PC \rangle_2 + 1$
Der Wert des Akkumulators wrd in die Speicherzelle mit der Adresse $\langle IN1 \rangle_2 + i$ kopiert.		
11	MOVE S D	$D \coloneqq S$
Der Wert des Registers S wird in das Register D kopiert.		

Beispiel: ein einfaches Programm

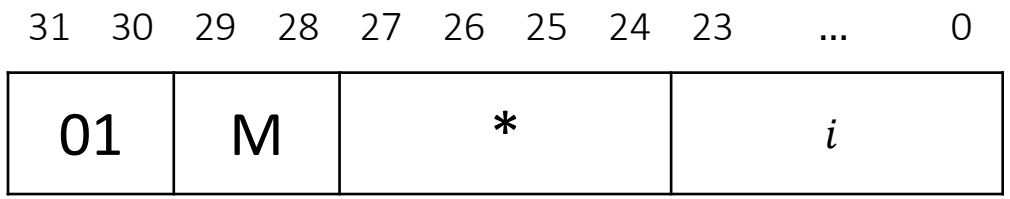
- Das folgende Programm vertauscht die Inhalte der RAM-Speicherzellen mit den Adressen 0 und 1.

```
0    LOAD 0     ACC:=S(0)
1    STORE 2    S(2):=ACC
2    LOAD 1     ACC:=S(1)
3    STORE 0    S(0) := ACC
4    LOAD 2     ACC:=S(2)
5    STORE 1    S(1): ACC
```

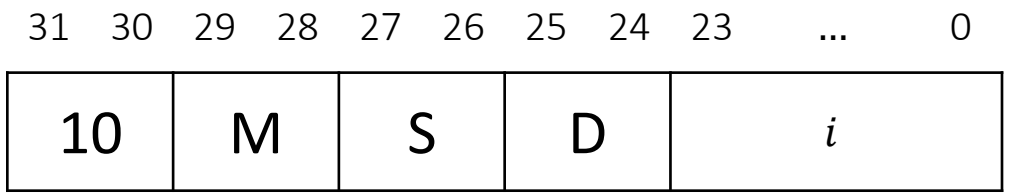
Werte des Akkumulators und der Speicherzellen 0, 1, 2 vor bzw. nach der Ausführung der Programmbefehle:

ACC	$S(0)$	$S(1)$	$S(2)$	
0	42	12	0	← bevor Befehl 0
42	42	12	0	← nach Befehl 0, bevor Befehl 1
42	42	12	42	← nach Befehl 1, bevor Befehl 2
12	42	12	42	← nach Befehl 2, bevor Befehl 3
12	12	12	42	← nach Befehl 3, bevor Befehl 4
42	12	12	42	← nach Befehl 4, bevor Befehl 5
42	12	42	42	← nach Befehl 5

Repräsentation im Programmspeicher



Ladebefehle



Speicherbefehle

```
0    LOAD 0
1    STORE 2
2    LOAD 1
3    STORE 0
4    LOAD 2
5    STORE 1
```

Adresse	Wert
0	01 00 0000 00000000000000000000000
1	10 00 00 00 00000000000000000000010
2	01 00 0000 00000000000000000000001
3	10 00 00 00 00000000000000000000000
4	01 00 0000 00000000000000000000010
5	10 00 00 00 00000000000000000000001

ALU-Operationen

R	F	Befehl	Effekt	
0	010	SUBI i	$\langle\langle ACC\rangle\rangle_2 \coloneqq \langle\langle ACC\rangle\rangle_2 - \langle\langle i\rangle\rangle_2$	$PC \coloneqq \langle PC\rangle_2 + 1$
	011	ADDI i	$\langle\langle ACC\rangle\rangle_2 \coloneqq \langle\langle ACC\rangle\rangle_2 + \langle\langle i\rangle\rangle_2$	$PC \coloneqq \langle PC\rangle_2 + 1$
	100	XORI i	$ACC \coloneqq ACC \oplus 00000000i$	$PC \coloneqq \langle PC\rangle_2 + 1$
	101	ORI i	$ACC \coloneqq ACC \vee 00000000i$	$PC \coloneqq \langle PC\rangle_2 + 1$
	111	ANDI i	$ACC \coloneqq ACC \wedge 00000000i$	$PC \coloneqq \langle PC\rangle_2 + 1$
1	010	SUB i	$\langle\langle ACC\rangle\rangle_2 \coloneqq \langle\langle ACC\rangle\rangle_2 - S(\langle i\rangle_2)$	$PC \coloneqq \langle PC\rangle_2 + 1$
	011	ADD i	$\langle\langle ACC\rangle\rangle_2 \coloneqq \langle\langle ACC\rangle\rangle_2 + S(\langle i\rangle_2)$	$PC \coloneqq \langle PC\rangle_2 + 1$
	100	XOR i	$ACC \coloneqq ACC \oplus S(\langle i\rangle_2)$	$PC \coloneqq \langle PC\rangle_2 + 1$
	101	OR i	$ACC \coloneqq ACC \vee S(\langle i\rangle_2)$	$PC \coloneqq \langle PC\rangle_2 + 1$
	111	AND i	$ACC \coloneqq ACC \wedge S(\langle i\rangle_2)$	$PC \coloneqq \langle PC\rangle_2 + 1$

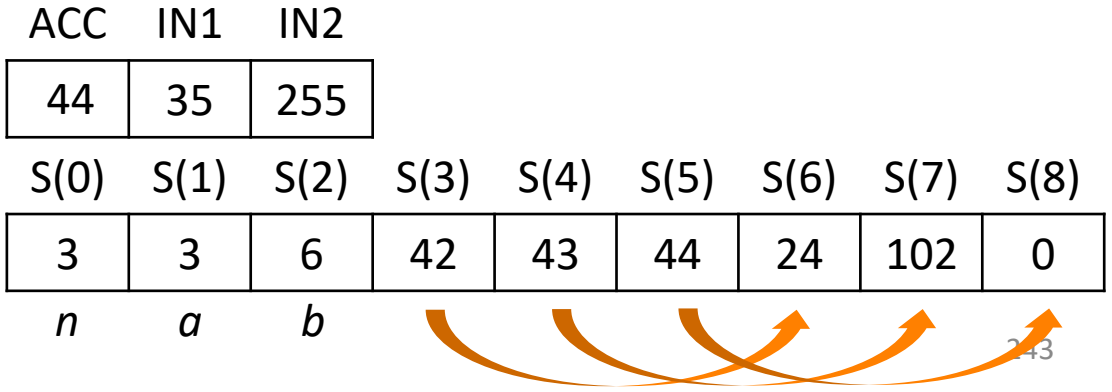
Da die Bitfolge i nur 24 Bits lang ist, die Operanden der bitweisen logischen Operationen aber gleiche Länge haben müssen, stellen wir i dort bei R=0 jeweils noch 8 Nullen voran.

Sprungbefehle

C	c	Befehl	Effekt
000		NOP	$PC := \langle PC \rangle_2 + 1$
001	$>$	$\text{JUMP}_c\ i$	$PC := \begin{cases} \langle PC \rangle_2 + \langle\langle i \rangle\rangle_2, & falls\ \langle\langle ACC \rangle\rangle_2\ c\ 0 \\ \langle PC \rangle_2 + 1, & sonst \end{cases}$
010	$=$		
011	$\geq$		
100	$<$		
101	$\neq$		
110	$\leq$		
111		JUMP i	$PC := \langle PC \rangle_2 + \langle\langle i \rangle\rangle_2$

Beispiel: Kopieren von Daten im Speicher

```
0   LOAD 1          ACC := a
1   MOVE ACC IN1    IN1 := a
2   LOAD 2          ACC := b
3   MOVE ACC IN2    IN2 := b
4   LOAD 0          ACC := n
5   JUMP= 13        falls n=0, stopp
6   LOADIN1 0       ACC := S(IN1)
7   STOREIN2 0      S(IN2) := ACC
8   MOVE IN1 ACC
9   ADDI 1          IN1 := IN1 + 1
10  MOVE ACC IN1
11  MOVE IN2 ACC
12  ADDI 1          IN2 := IN2 + 1
13  MOVE ACC IN2
14  LOAD 0
15  SUBI 1          S(0) := S(0) - 1
16  STORE 0
17  JUMP -12        springe nach 5
18                  stopp
```



Was fehlt noch zur Umsetzung?

- Wir haben Komponenten (Schaltkreise, Schaltungen) definiert.
- Wir haben Busse eingeführt.
- Wir haben die Beispielarchitektur einer CPU vorgestellt.
- Wir haben einen Instruktionssatz festgelegt.

➢ Was noch fehlt, ist die konkrete Ansteuerung der Eingänge des Schaltkreises je nach Befehl.

- Dazu dient eine sog. Kontrollogik. Sie setzt z.B. Steuerbits oder enabled die richtigen Bustreiber für jeden Befehl.

Moderne CPUs

- Moderne CPU sind um ein Vielfaches komplexer und leistungsfähiger.
- Einige Schlagwörter zur Unterstützung eines freiwilligen, weiterführenden Selbststudium:
  - Caches
  - Prefetch
  - Out-of-order execution
  - Multicore

Programmierung

- In den frühen Jahren gab es ausschließlich Programmierung auf diesem maschinennahen Level
  - Maschinensprachen
  - Assembler-Sprachen
- Zur Lösung komplexerer Probleme ist eine höhere Abstraktionsebene förderlich
  - Höhere Programmiersprachen
- Auch heute noch ist Assembler-Programmierung relevant, zum Beispiel im Beispiel „Embedded Computing“.

Beispiel für Programmcode in einer höheren Programmiersprache

```
i = 0;
while (i < n) {
    b[i] = a[i];
    i++;
}
```

Wie funktionieren Höhere Programmiersprachen?

- Benötigt man dafür völlig andere Rechnerarchitekturen?
  - Nein
  - Stattdessen werden Höhere Programmiersprachen letztendlich wieder in Maschinensprache übersetzt
- Compilersprachen
- Interpreter
- Mischformen
- Mehr dazu im nächsten Semester!