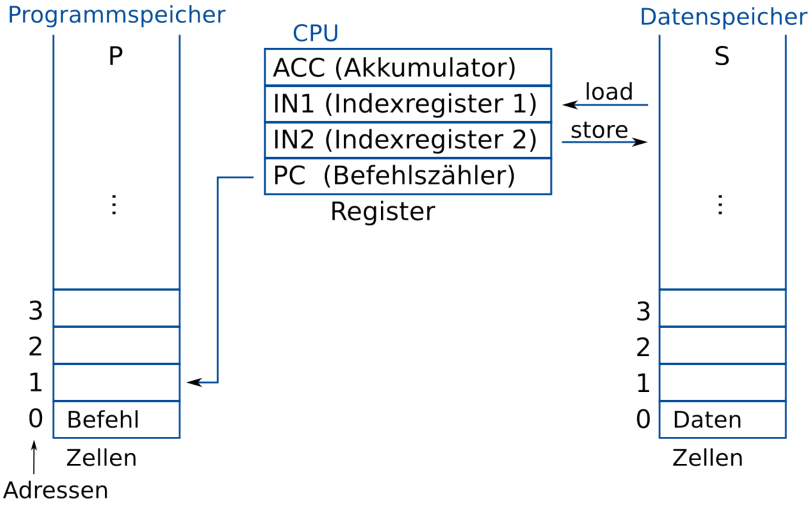
Abstrakte ReTI-Maschine

- Zwei unendlich große Speicher
  - Datenspeicher S für Daten (beliebig große Zahlen).
  - $S(i)$ = Inhalt von Zelle i des Datenspeichers, $i \in N$ Adresse.
  - Programmspeicher P für Maschinenbefehle.
  - Lade-/Speicher-, Rechen-, Sprungbefehle - siehe später.
  - $P(i)$ = Inhalt von Zelle i des Programmspeichers.
- Zentraleinheit CPU (Central Processing Unit)
  - Vier für Benutzer sichtbare Register.
  - PC = Befehlszähler (Program Counter).
  - ACC = Akkumulator.
  - $IN1$, $IN2$ = Indexregister 1 und 2.

Aufbau von ReTI



Programmablauf

- Programme bzw. Daten stehen beim Start der Maschine in P bzw. S.
- Programm beginnt bei Zelle 0 von P.
- Inhalt von P wird nicht geändert.
- Maschine arbeitet in Schritten $t = 1, 2, \ldots$
  In jedem Schritt t:
  - Ausführung eines Befehls: $P(PC)$ wird als Befehl interpretiert und in Schritt t ausgeführt.
  - PC erhält neuen Wert (abhängig von Befehl).
- Bei Programmstart ist $PC = 0$.

ReTI-Befehle und ihre Wirkung

- Load/Store: Laden von Werten aus dem Datenspeicher *S* bzw. Schreiben von Werten in *S*.
- Compute: Berechnungen (hier zunächst Addition und Subtraktion).
  - Mit Werten im Datenspeicher *S*.
  - Mit Absolutwerten (Immediate).
- Indexregister: Indirekte Speicheradressierung (siehe unten).
- Sprungbefehle: Bedingte und unbedingte Sprünge.

Load/Store

Transport von Daten zwischen *ACC* und Datenspeicher.

- LOAD *i*:
  Lädt Inhalt $S(i)$ von Speicherzelle *i* in Akkumulator *ACC* und erhöht *PC* um 1.

- STORE *i*:
  Speichert den Inhalt von *ACC* in $S(i)$ und erhöht *PC* um 1.

Load/Store: Übersicht

Befehl	Wirkung	
LOAD i	$ACC := S(i)$	$PC := PC + 1$
STORE i	$S(i) := ACC$	$PC := PC + 1$

Beispielprogramm

Ein Programm, das Inhalte von Speicherzelle $S(0)$ $(= x)$ und $S(1)$ $(= y)$ vertauscht.

0	LOAD 0;	$ACC := S(0) = x$
1	STORE 2;	$S(2) := ACC = x$
2	LOAD 1;	$ACC := S(1) = y$
3	STORE 0;	$S(0) := ACC = y$
4	LOAD 2;	$ACC := S(2) = x$
5	STORE 1;	$S(1) := ACC = x$

Compute-Befehle

Verknüpfe den Inhalt von ACC mit $S(i)$ oder mit einer Konstante und speichere das Ergebnis in ACC ab.

- ADD, SUB = Compute *memory*-Befehle
- $ADDI, SUBI$ = Compute *immediate*-Befehle
- Beides zusammen ergibt die Compute-Befehle.

Bei Compute memory: Interpretiere Parameter i direkt als Speicheradresse.

Befehl	Wirkung	
ADD i	$ACC := ACC + S(i)$	$PC := PC + 1$
SUB i	$ACC := ACC - S(i)$	$PC := PC + 1$

Immediate-Befehle

Interpretiere Parameter *i* direkt als Konstante.

Befehl	Wirkung	
LOADI *i*	$ACC := i$	$PC := PC + 1$
ADDI *i*	$ACC := ACC + i$	$PC := PC + 1$
SUBI *i*	$ACC := ACC - i$	$PC := PC + 1$

- Anmerkung: ADDI und SUBI sind Compute Befehle.
  LOADI ist den Load-/Store-Befehlen zuzuordnen.

Indexregister-Befehle

Befehl	Wirkung	
LOADINj i	$ACC := S(INj + i)$	$PC := PC + 1$
	$(j \in \{1,2\})$	
STOREINj i	$S(INj + i) := ACC$	$PC := PC + 1$
	$(j \in \{1,2\})$	
MOVE S D	$D := S$	$PC := PC + 1$
	$(D \in \{ACC, IN1, IN2\},$	
	$S \in \{ACC, IN1, IN2, PC\})$	
MOVE S PC	$PC := S$	
	$(S \in \{ACC, IN1, IN2\})$	

Beispielprogramm für Indexregister-Befehle

$S(0) = x$, $S(1) = y$
Kopiere y in Zelle $S(x)$:

0	LOAD 0;	$ACC := S(0) = x$
1	MOVE ACC $IN1$;	$IN1 := ACC = x$
2	LOAD 1;	$ACC := S(1) = y$
3	STOREIN1 0;	$S(x) = S(IN1 + 0) := ACC = y$

Sprung-Befehle

Manipulation des Befehlszählers.

- JUMP für *unbedingte* Sprünge,
- JUMP_c mit $c \in \{<, =, >, \leq, \neq, \geq\}$ für *bedingte* Sprünge.
- Mit bedingten Sprüngen kann man *Programmschleifen* und *bedingte Anweisungen* realisieren!

Befehl	Wirkung
JUMP i	$PC := PC + i \quad (i \in \mathbb{Z})$
$\text{JUMP}_c\ i$	$PC := \begin{cases} PC + i, & \text{falls } ACC\ c\ 0 \\ PC + 1, & \text{sonst} \end{cases}$ $(i \in \mathbb{Z}, c \in \{<, =, >, \leq, \neq, \geq\})$

Beispielprogramm

$S(0) = x;\ S(1) = y,\ y \geq 0$

0	LOADI 0;	$ACC := 0$
1	STORE 2;	$S(2) := 0$
2	LOAD 1;	$ACC := S(1)$
3	SUBI 1;	$ACC := ACC - 1$
4	STORE 1;	$S(1) := ACC$
5	JUMP $<$ 5;	$PC := PC + 5$, falls $ACC < 0$
6	LOAD 2;	$ACC := S(2)$
7	ADD 0;	$ACC := ACC + S(0)$
8	STORE 2;	$S(2) := ACC$
9	JUMP -7;	$PC := PC - 7$

Boolesche Algebren - allgemein

- Es sei M eine Menge auf der zwei binäre Operationen $\cdot$ und $+$ und eine unäre Option $\sim$ definiert sind.
- Das Tupel $(M, \cdot, +, \sim)$ heißt boolesche Algebra, falls M eine nichtleere Menge ist und für alle $x, y, z \in M$ die folgenden Axiome gelten:

Kommutativität	$x + y = y + x$	$x \cdot y = y \cdot x$
Assoziativität	$x + (y + z) = (x + y) + z$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Absorption	$x + (x \cdot y) = x$	$x \cdot (x + y) = x$
Distributivität	$x + (y \cdot z) = (x + y) \cdot (x + z)$	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Komplement	$x + (y \cdot \sim y) = x$	$x \cdot (y + \sim y) = x$

Weitere, aus den Axiomen ableitbare Regeln:

- Existenz neutraler Elemente:
  $\exists \mathbf{0} : x + \mathbf{0} = x,\ x \cdot \mathbf{0} = \mathbf{0} \quad \exists \mathbf{1} : x \cdot \mathbf{1} = x,\ x + \mathbf{1} = \mathbf{1}$
- Doppeltes Komplement:
  $(\sim(\sim x)) = x$
- Eindeutigkeit des Komplements:
  $(x \cdot y = \mathbf{0}$ und $x + y = \mathbf{1}) \Rightarrow y = (\sim x)$
- Idempotenz:
  $x + x = x \quad x \cdot x = x$
- de Morgan-Regel:
  $\sim(x + y) = (\sim x) \cdot (\sim y) \quad \sim(x \cdot y) = (\sim x) + (\sim y)$
- Consensus-Regel:
  $(x \cdot y) + ((\sim x) \cdot z) = (x \cdot y) + ((\sim x) \cdot z) + (y \cdot z)$
  $(x + y) \cdot ((\sim x) + z) = (x + y) \cdot ((\sim x) + z) \cdot (y + z)$
- Diese Regeln gelten in allen booleschen Algebren!

Typ einer Instruktion

I[31, 30]	**Typ**
0 0	Compute
0 1	Load
1 0	Store, Move
1 1	Jump

31 30	29 ... 24	23 ... 0
Typ	Spezifikation	Parameter i

Load-Befehle: Kodierungsprinzip

31 30	29 28	27 26	25 24	23 ... 0
0 1	M	*	D	i

- M: Modus
- D: Vorerst irrelevant

Load-Befehle: Kodierung

Typ	**Modus**	**Befehl**	**Wirkung**	
0 1	0 0	LOAD i	$ACC := M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$
0 1	0 1	LOADIN1 i	$ACC := M(\langle IN1 \rangle + [i])$	$\langle PC \rangle := \langle PC \rangle + 1$
0 1	1 0	LOADIN2 i	$ACC := M(\langle IN2 \rangle + [i])$	$\langle PC \rangle := \langle PC \rangle + 1$
0 1	1 1	LOADI i	$ACC := 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$

Durchführung von Rechnungen $\langle x \rangle + [y]$ später.

Store-, Move-Befehle: Prinzip

31 30	29 28	27	26	25 24	23 ... 0
1 0	M	S		D	i

- M: Modus
- S: Source
- D: Destination

Kodierung S, D

S, D	**Register**
0 0	PC
0 1	IN1
1 0	IN2
1 1	ACC

UNI FREIBURG

Store-, Move-Befehle: Kodierung

Typ	**Modus**	**Befehl**	**Wirkung**	
1 0	0 0	STORE i	$M(\langle i \rangle) := ACC$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	0 1	STOREIN1 i	$M(\langle IN1 \rangle + [i]) := ACC$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	1 0	STOREIN2 i	$M(\langle IN2 \rangle + [i]) := ACC$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	1 1	MOVE S D	$D := S$	$\langle PC \rangle := \langle PC \rangle + 1$

außer bei $D = 00$ (PC)

Compute-Befehle: Prinzip

31 30	29	28 27 26	25 24	23 ... 0
0 0	MI	F	D	i

- MI: „compute **m**emory"/„compute **i**mmediate"
- F: Funktionsfeld
- D: Vorerst irrelevant

Compute-Befehle: Kodierung

Typ	MI	F	Befehl	Wirkung	
0 0	0	0 1 0	SUBI i	$[ACC] := [ACC] - [i]$	$\langle PC\rangle := \langle PC\rangle + 1$
		0 1 1	ADDI i	$[ACC] := [ACC] + [i]$	$\langle PC\rangle := \langle PC\rangle + 1$
		1 0 0	OPLUSI i	$ACC := ACC \oplus 0^8 i$	$\langle PC\rangle := \langle PC\rangle + 1$
		1 0 1	ORI i	$ACC := ACC \vee 0^8 i$	$\langle PC\rangle := \langle PC\rangle + 1$
		1 1 0	ANDI i	$ACC := ACC \wedge 0^8 i$	$\langle PC\rangle := \langle PC\rangle + 1$
0 0	1	0 1 0	SUB i	$[ACC] := [ACC] - [M(\langle i\rangle)]$	$\langle PC\rangle := \langle PC\rangle + 1$
		0 1 1	ADD i	$[ACC] := [ACC] + [M(\langle i\rangle)]$	$\langle PC\rangle := \langle PC\rangle + 1$
		1 0 0	OPLUS i	$ACC := ACC \oplus M(\langle i\rangle)$	$\langle PC\rangle := \langle PC\rangle + 1$
		1 0 1	OR i	$ACC := ACC \vee M(\langle i\rangle)$	$\langle PC\rangle := \langle PC\rangle + 1$
		1 1 0	AND i	$ACC := ACC \wedge M(\langle i\rangle)$	$\langle PC\rangle := \langle PC\rangle + 1$

Load-Befehle		$I[25:24] = D$	
$I[31:28]$	Befehl	Wirkung	
0100	LOAD D i	$D := M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
0101	LOADIN1 D i	$D := M(\langle IN1 \rangle + [i])$	
0110	LOADIN2 D i	$D := M(\langle IN2 \rangle + [i])$	
0111	LOADI D i	$D := 0^8 i$	

Store-Befehle		MOVE: $I[27:24] = SD$	
$I[31:28]$	Befehl	Wirkung	
1000	STORE i	$M(\langle i \rangle) := ACC$	$\langle PC \rangle := \langle PC \rangle + 1$
1001	STOREIN1 i	$M(\langle IN1 \rangle + [i]) := ACC$	
1010	STOREIN2 i	$M(\langle IN2 \rangle + [i]) := ACC$	
1011	MOVE S D	$D := S$	$\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$

Compute-Befehle		$I[25:24] = D$	
$I[31:26]$	Befehl	Wirkung	
000010	SUBI D i	$[D] := [D] - [i]$	$\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
000011	ADDI D i	$[D] := [D] + [i]$	
000100	OPLUSI D i	$D := D \oplus 0^8 i$	
000101	ORI D i	$D := D \vee 0^8 i$	
000110	ANDI D i	$D := D \wedge 0^8 i$	
001010	SUB D i	$[D] := [D] - [M(\langle i \rangle)]$	$\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
001011	ADD D i	$[D] := [D] + [M(\langle i \rangle)]$	
001100	OPLUS D i	$D := D \oplus M(\langle i \rangle)$	
001101	OR D i	$D := D \vee M(\langle i \rangle)$	
001110	AND D i	$D := D \wedge M(\langle i \rangle)$	

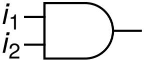
Jump-Befehle		
$I[31:27]$	Befehl	Wirkung
11000	NOP	$\langle PC \rangle := \langle PC \rangle + 1$
11001	JUMP$_{>}$ i	$\langle PC \rangle := \begin{cases} \langle PC \rangle + [i], & \text{falls } [ACC] \ c \ 0 \\ \langle PC \rangle + 1, & \text{sonst} \end{cases}$
11010	JUMP$_{=}$ i	
11010	JUMP$_{\geq}$ i	
11011	JUMP$_{<}$ i	
11100	JUMP$_{\neq}$ i	
11110	JUMP$_{\leq}$ i	
11111	JUMP i	$\langle PC \rangle := \langle PC \rangle + [i]$

Kodierung S,D

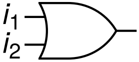
S, D	**Register**
0 0	PC
0 1	$IN1$
1 0	$IN2$
1 1	ACC

Einige wichtige Gatter

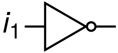
i_1	i_2	AND_2
0	0	0
0	1	0
1	0	0
1	1	1



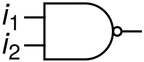
i_1	i_2	OR_2
0	0	0
0	1	1
1	0	1
1	1	1



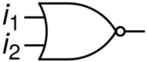
i_1	NOT_2
0	1
1	0



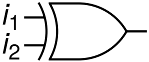
i_1	i_2	$NAND_2$
0	0	1
0	1	1
1	0	1
1	1	0



i_1	i_2	NOR_2
0	0	1
0	1	0
1	0	0
1	1	0

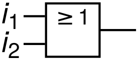
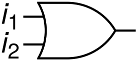
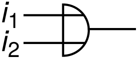
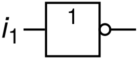
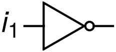
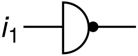


i_1	i_2	XOR_2
0	0	0
0	1	1
1	0	1
1	1	0



Logikgatter - verschiedene Notationen

Es gibt verschiedene Notationen für Logikgatter.

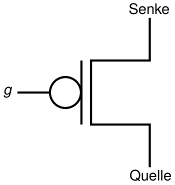
	IEC	ANSI	DIN
OR_2			
NOT			

Wir werden in dieser Vorlesung die ANSI-Notation verwenden.

Transistoren

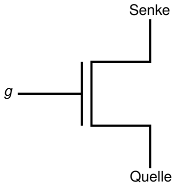
- Einen Transistor kann man vereinfacht als spannungsgesteuerten Schalter sehen:
  - Leitung *g* (gate) regelt Leitfähigkeit zwischen Quelle und Senke.

p-Kanal Transistor



- Leitet, wenn an *g* eine 0 anliegt.
- Sperrt, wenn an *g* eine 1 anliegt.

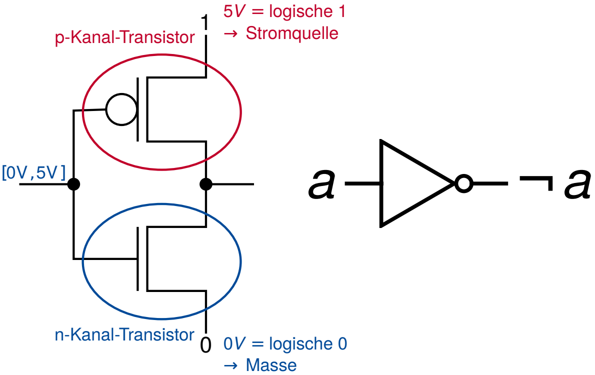
n-Kanal Transistor



- Leitet, wenn an *g* eine 1 anliegt.
- Sperrt, wenn an *g* eine 0 anliegt.

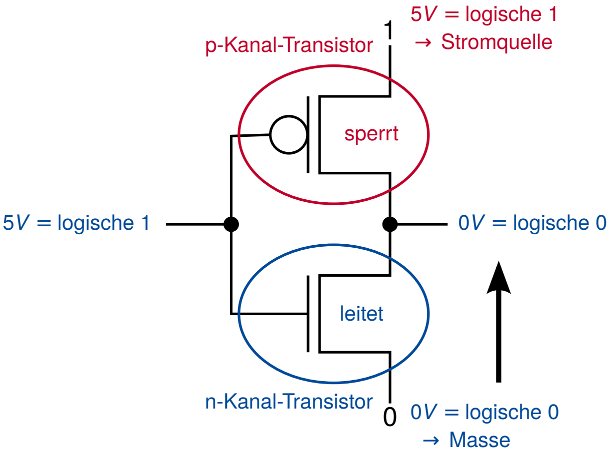
Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.



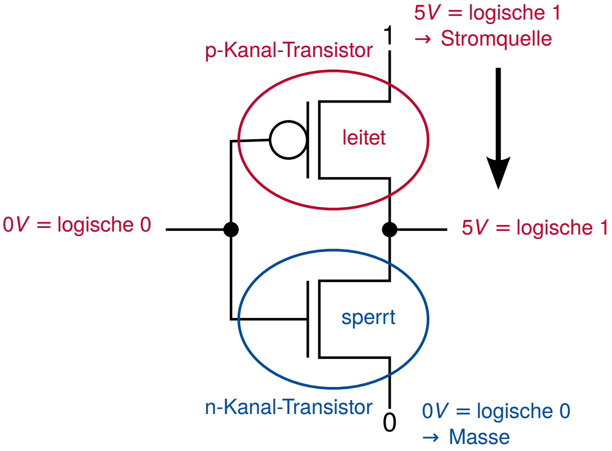
CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).

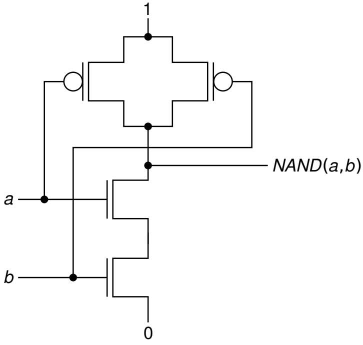


CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



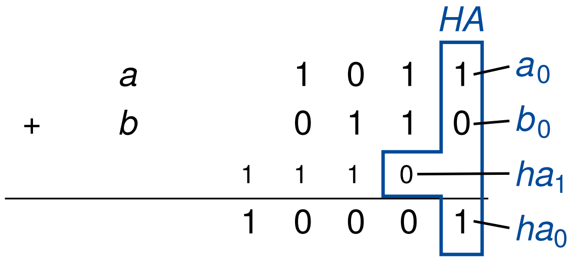
CMOS-NAND-Gatter



- Ausgang ist 0
  $\Leftrightarrow$ Es existiert ein leitender Pfad von 0 zum Ausgang
  $\Leftrightarrow$ beide n-Kanal-Transistoren leiten
  $\Leftrightarrow a = b = 1, a \wedge b = 1$
  $\Leftrightarrow NAND(a,b) = 0$

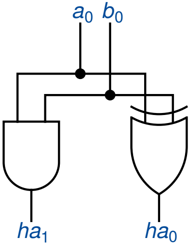
- Ausgang ist 1
  $\Leftrightarrow$ Es existiert ein leitender Pfad von 1 zum Ausgang
  $\Leftrightarrow$ einer der p-Kanal-Transistoren leitet
  $\Leftrightarrow a = 0$ oder $b = 0, \neg a \vee \neg b = 1$
  $\Leftrightarrow NAND(a,b) = 1$

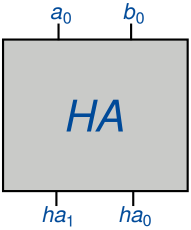
Addieren nach der Schulmethode (2/4)



a_0	b_0	ha_1	ha_0
0	0		
0	1		
1	0		
1	1		

Schaltkreis eines Halbaddierers





dabei gilt:

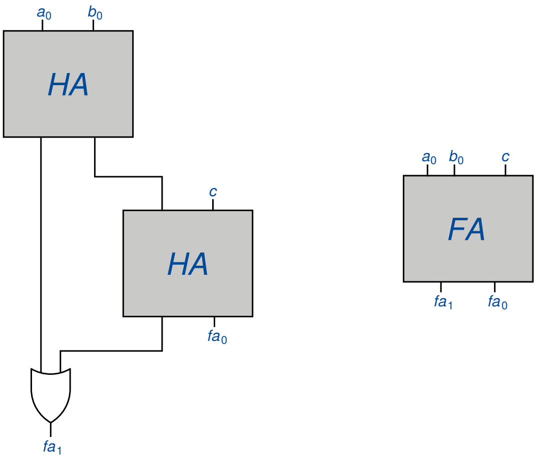
$C(HA) = 2, \quad depth(HA) = 1$

Volladierer (Full Adder, *FA*)

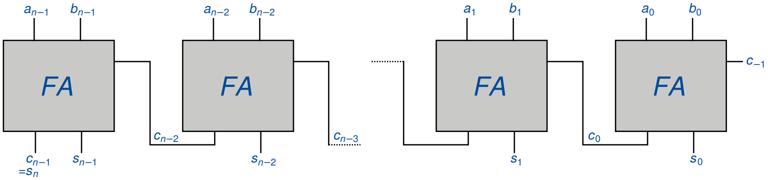
- Ein Volladdierer dient zur Addition zweier 1-Bit-Zahlen mit Eingangsübertrag.
- Er berechnet die Funktion $fa : \mathbb{B}^3 \to \mathbb{B}^2$ mit $fa(a_0, b_0, c) = (fa_1, fa_0)$ wobei $2fa_1 + fa_0 = a_0 + b_0 + c$

a_0	b_0	c	fa_1	fa_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

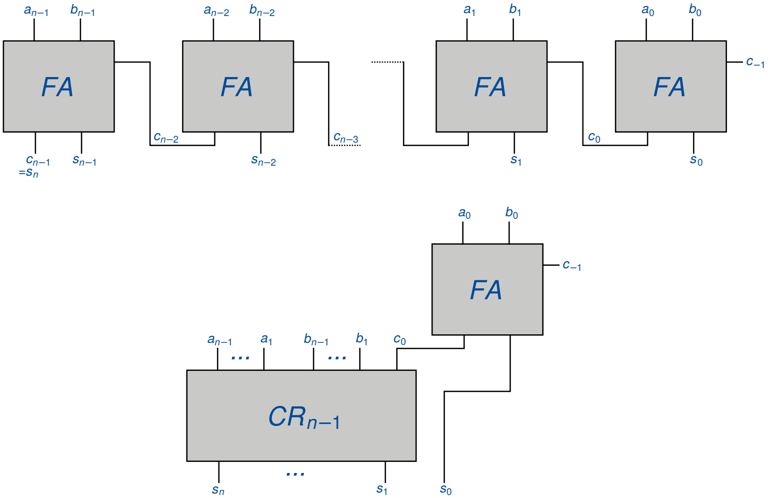
Schaltkreis eines Volladdierers



Aufbau eines Carry-Ripple-Addierers

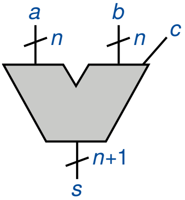


Zwei (identische) Darstellungen von CR



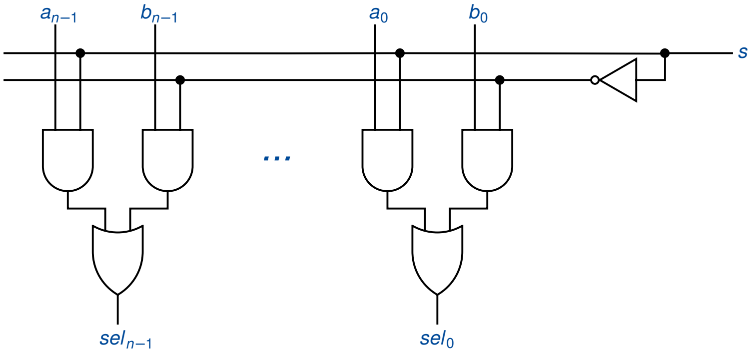
Schaltbild und Komplexität von *CR*

- $C(CR_n) = n \cdot C(FA) = 5n$.
- $depth(CR_n) = 3 + 2(n-1)$.
- Sowohl die Kosten als auch die Tiefe von *CR* sind somit linear in n.



- Es gibt (asymptotisch) bessere Addierer. Wir werden hier den Conditional-Sum-Addierer kennen lernen, für den wir wieder eine Hilfsschaltung (Multiplexer) benötigen.
- Eine weitere wichtige Schaltung ist der Inkrementer.

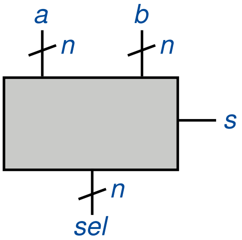
Aufbau von MUX_n



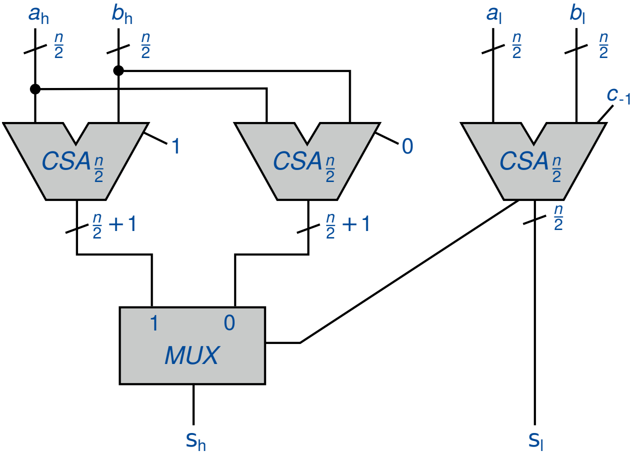
Schaltbild und Kosten MUX_n

Kosten und Tiefe:

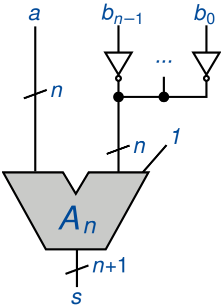
$C(MUX_n) = 3n + 1.$
$depth(MUX_n) = 3.$

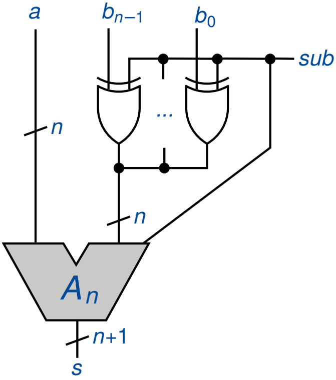


Aufbau von CSA_n



Subtrahierer





$b_i \oplus 0 = b_i$ $\quad$ $sub = 0 : [a] + [b] + 0$

$b_i \oplus 1 = \overline{b_i}$ $\quad$ $sub = 1 : [a] + [\overline{b}] + 1 = [a] - [b]$

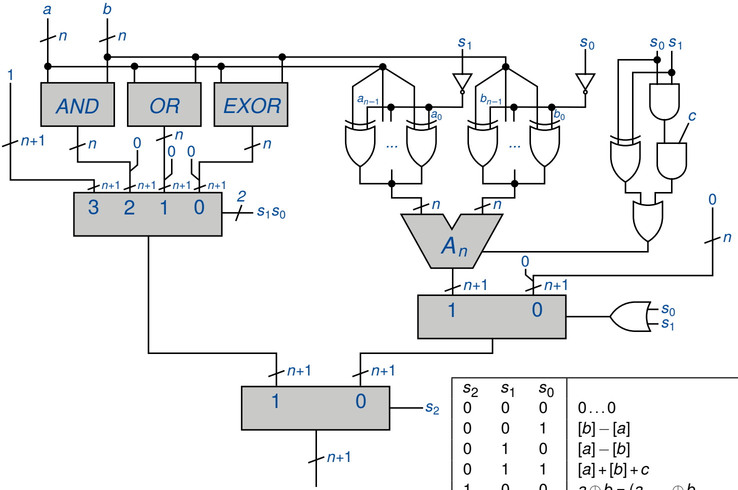
Compute-Befehle: Kodierung

Typ	MI	F	Befehl	Wirkung	
0 0	0	0 1 0	SUBI i	$[ACC] := [ACC] - [i]$	$\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADDI i	$[ACC] := [ACC] + [i]$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUSI i	$ACC := ACC \oplus 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	ORI i	$ACC := ACC \vee 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	ANDI i	$ACC := ACC \wedge 0^8 i$	$\langle PC \rangle := \langle PC \rangle + 1$
0 0	1	0 1 0	SUB i	$[ACC] := [ACC] - [M(\langle i \rangle)]$	$\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADD i	$[ACC] := [ACC] + [M(\langle i \rangle)]$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUS i	$ACC := ACC \oplus M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	OR i	$ACC := ACC \vee M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	AND i	$ACC := ACC \wedge M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$

Select-Eingang bei ReTI-ALU

Funktionsnummer			ALU-Funktion
s_2	s_1	s_0	
0	0	0	$0 \dots 0$
0	0	1	$[b] - [a]$
0	1	0	$[a] - [b]$
0	1	1	$[a] + [b] + c$
1	0	0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \dots, a_0 \oplus b_0)$
1	0	1	$a \vee b = (a_{n-1} \vee b_{n-1}, \dots, a_0 \vee b_0)$
1	1	0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \dots, a_0 \wedge b_0)$
1	1	1	$1 \dots 1$

Schaltrealisierung der ALU (1/2)

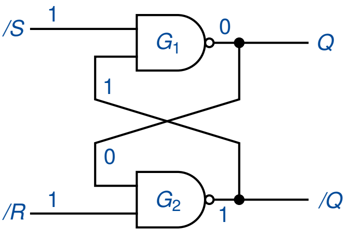


s_2	s_1	s_0	
0	0	0	$0\dots0$
0	0	1	$[b]-[a]$
0	1	0	$[a]-[b]$
0	1	1	$[a]+[b]+c$
1	0	0	$a\oplus b=(a_{n-1}\oplus b_{n-1},\dots,a_0\oplus b_0)$
1	0	1	$a\vee b=(a_{n-1}\vee b_{n-1},\dots,a_0\vee b_0)$
1	1	0	$a\wedge b=(a_{n-1}\wedge b_{n-1},\dots,a_0\wedge b_0)$
1	1	1	$1\dots1$

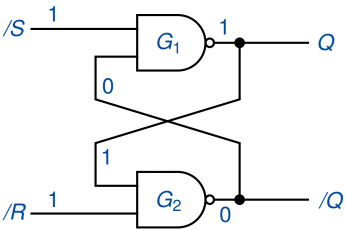
UNI FREIBURG

RS-Flipflop (RS-FF)

- Die vorherige Schaltung heißt RS-Flipflop.
- Sie hat mindestens zwei stabile Zustände.

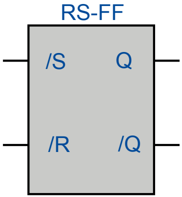


Zustand $Q = 0$

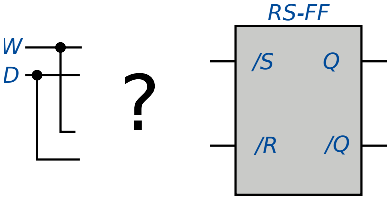


Zustand $Q = 1$

Schaltsymbol eines RS-FF



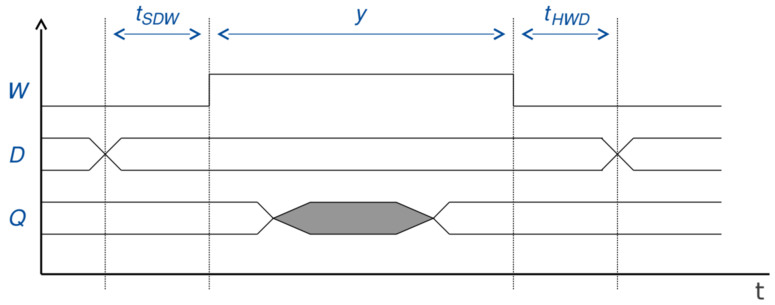
D-Latch (1/2)



W	D	$/S$	$/R$
0	0		
0	1		
1	0		
1	1		

- W ist active high.
  - $W = 0 \Rightarrow /S, /R$ inaktiv
  - $W = 1 \Rightarrow \begin{cases} /S \text{ aktiv, falls } D = 1 \\ /R \text{ aktiv, falls } D = 0 \end{cases}$

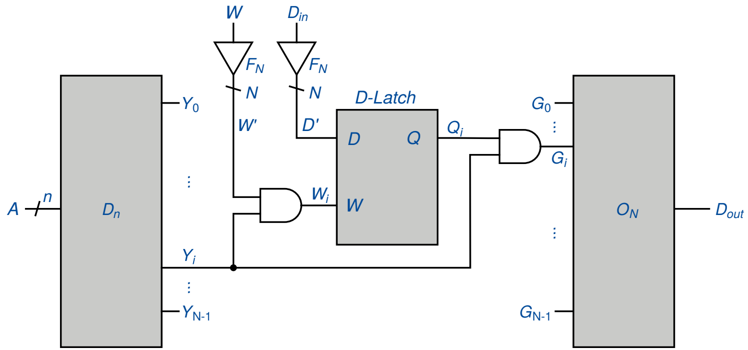
Timing-Diagramm



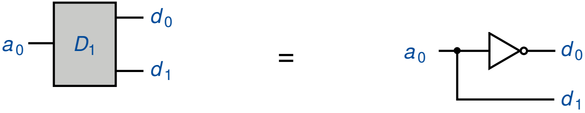
Wie lange müssen die einzelnen Signale aktiv sein, damit der Schreibvorgang reibungslos abläuft?

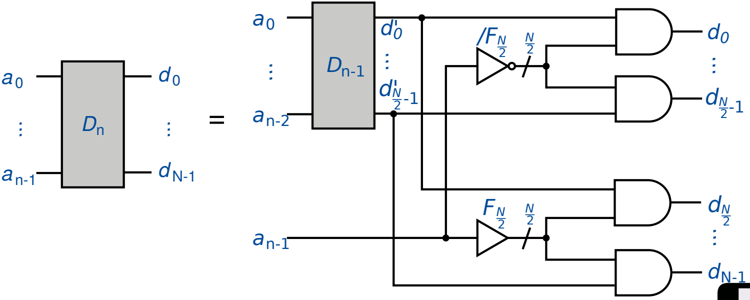
⇒ Siehe nächstes Kapitel (Timing).

SRAM: Schaltbild



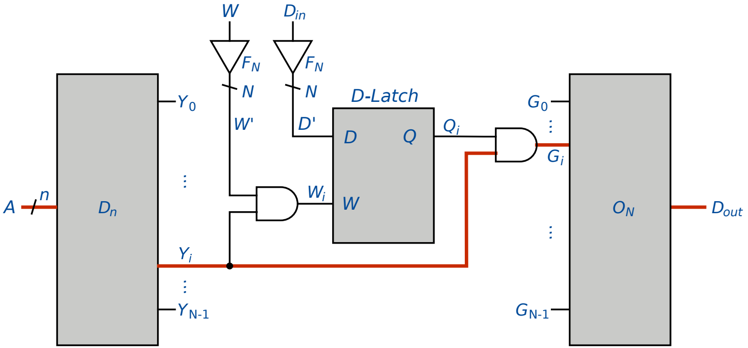
Dekodierer: Rekursiver Aufbau





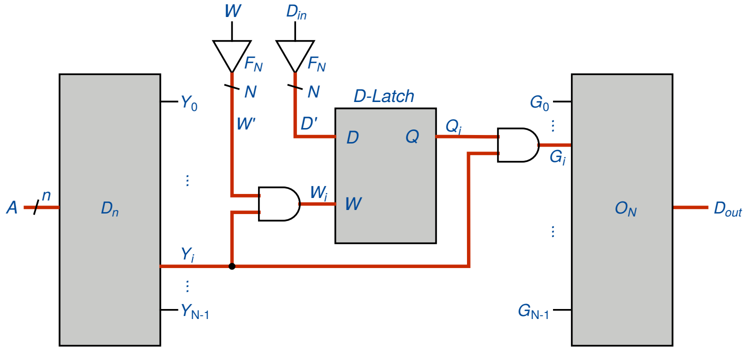
SRAM: Lesevorgang ($W = 0$)

- D_n setzt $Y_i = 1$ für $i = \langle A \rangle$, $Y_j = 0$ für $j \neq i$.
- Der Inhalt der i-ten Zelle L_i steht an G_i, für alle $j \neq i$ steht an G_j der Wert 0.



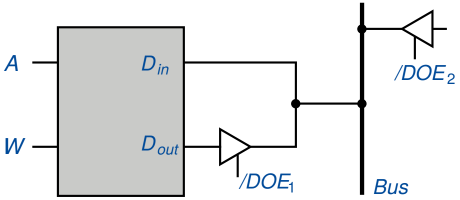
SRAM: Schreibvorgang (Puls auf W)

- D_{in} an D-Eingänge sämtlicher Latches angelegt.
- Schreibpuls nur am W-Eingang von L_i (da $Y_i = 1$).



Bus zur Kommunikation mit SRAM

- SRAM mit gemeinsamem Datenein- und -ausgang.



- Lesezugriff auf den Speicher: $/DOE_1$ enabled, alle anderen Treiber, z.B. $/DOE_2$, disabled.

- Schreibzugriff: D_{in} nimmt den Wert vom Bus, $/DOE_1$ disabled.

UNI FREIBURG

Zur Erinnerung: ReTI-Befehle im Überblick

	31 30	29 28	27 26	25 24	23 ... 0
Load	0 1	M	*	D	i

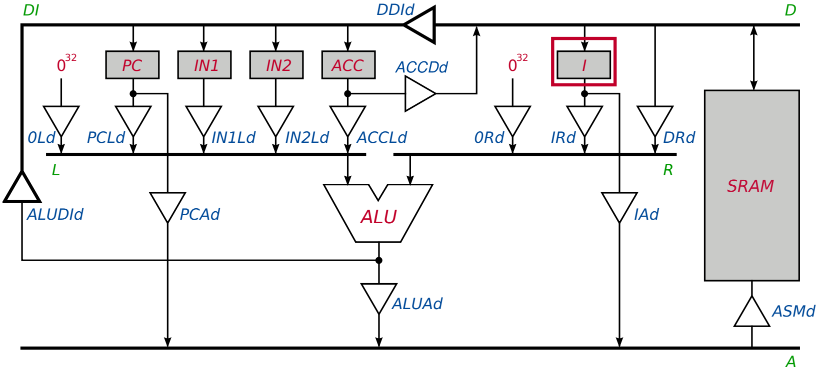
	31 30	29 28	27 26	25 24	23 ... 0
Store	1 0	M	S	D	i

	31 30	29	28 27 26	25 24	23 ... 0
Compute	0 0	MI	F	D	i

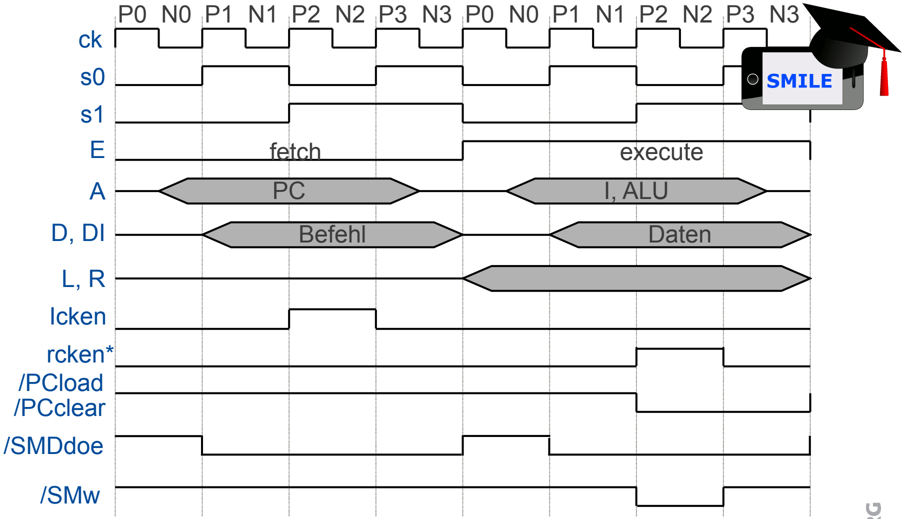
	31 30	29 28 27	26 25 24	23 ... 0
Jump	1 1	C	*	i

M - Modus ; S - Source ; D - Destination ; MI - memory/immediate ;
F - Function ; C - Condition

Verfeinerung des Schaltbilds

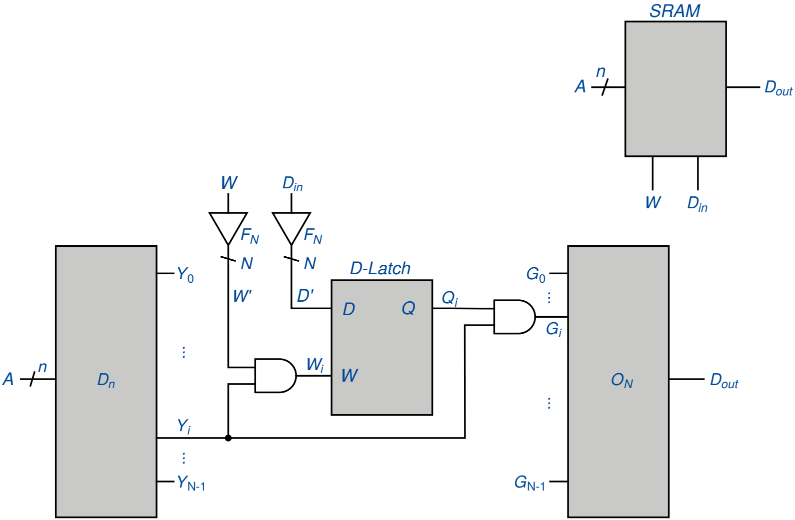


Idealisiertes Timing-Diagramm



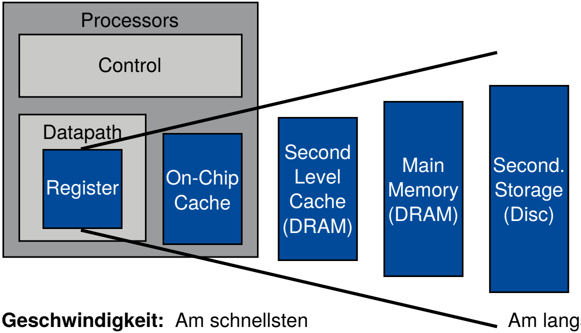
(* r = PC, IN1, IN2, ACC)

SRAM



SRAM
A
n
D_{out}
W
D_{in}
W
D_{in}
F_N
F_N
N
N
W'
D'
D-Latch
D
Q
W
W_i
Q_i
Y_0
Y_i
Y_{N-1}
A
n
D_n
G_0
G_i
G_{N-1}
O_N
D_{out}

Speicherhierarchie (2/3)



Geschwindigkeit:	Am schnellsten	Am langsamsten
Größe:	Am kleinsten	Am größten
Kosten:	Am teuersten	Am billigsten