Technische Universität Braunschweig

Informatik für Ingenieure – VL 9

Übersicht

Letztes Mal:

- Endlich Automaten
- Frequenzteiler, Zähler
- Synchrone Schaltungen

Heute:

- Addierer und Multiplizierer
- ALU
- Speicher

Teile des heutigen Vortrags basiert auf der Vorlesungen von
Prof. H Michalik (TU Braunschweig)
Prof. J. Wawrzynek (UC Berkley)
Prof. M. Luisier (ETH Zurich)

4. Schnelle Arithmetik

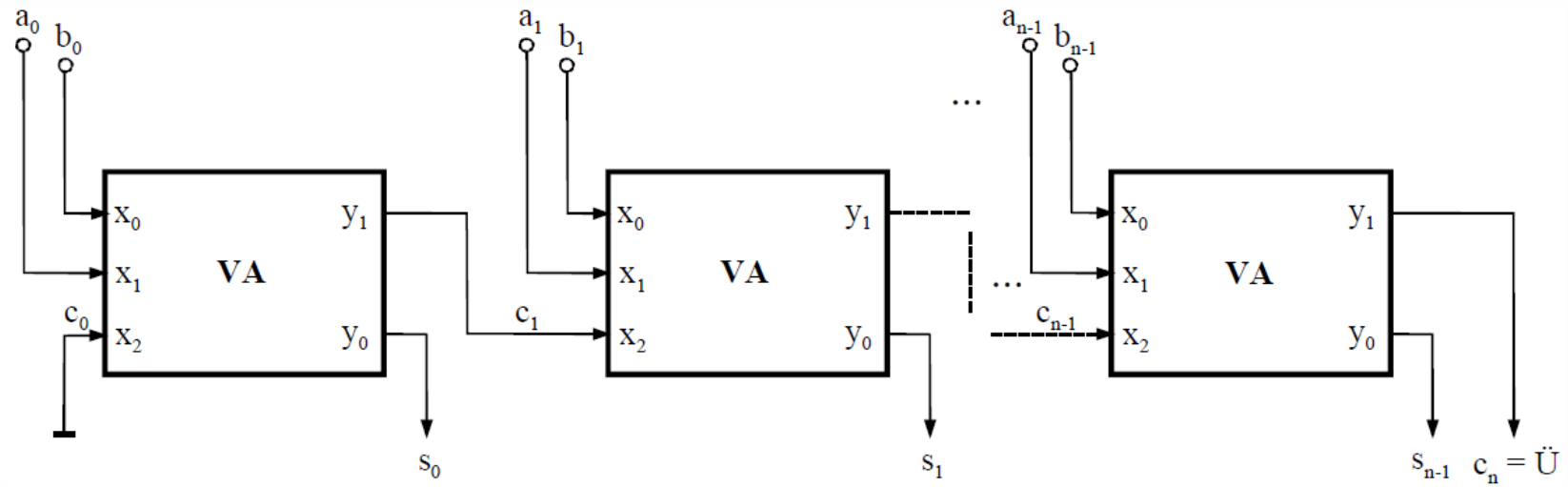
Addition

Ripple-Carry Addierer

Der längste Zeitaufwand ist die Anzahl der Stellen, durch die ein Übertrag hindurchwandern muss.

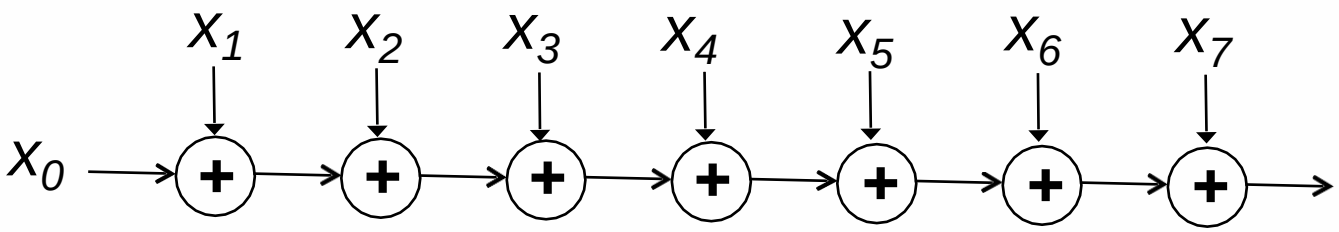
Wenn jeder Volladdierer die Zeit t_{add} benötigt, ist die Gesamtzeit also $n \cdot t_{add}$.

Im Folgendem sollen Lösungen betrachtet werden, um diese Zeit zu verringern.



Berechnung von Funktionen von n Variablen

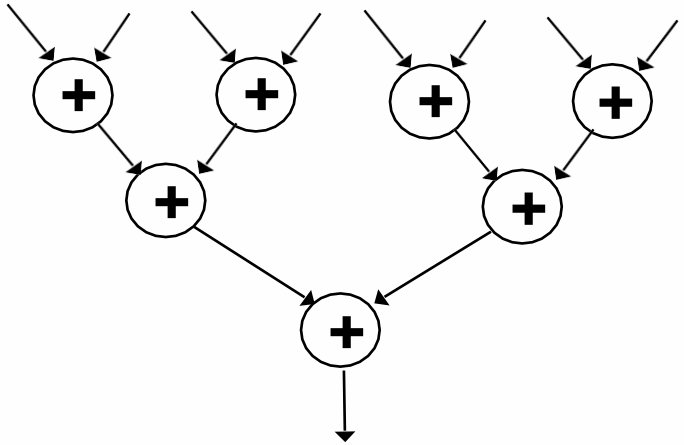
Startpunkt $((((((x_0 + x_1) + x_2) + x_3) + x_4) + x_5) + x_6) + x_7$



Kosten:

Zeit $\propto N*T_{add}$
Platz $\propto N*P_{add}$

Optimierung $((x_0 + x_1) + (x_2 + x_3)) + ((x_4 + x_5) + (x_6 + x_7))$



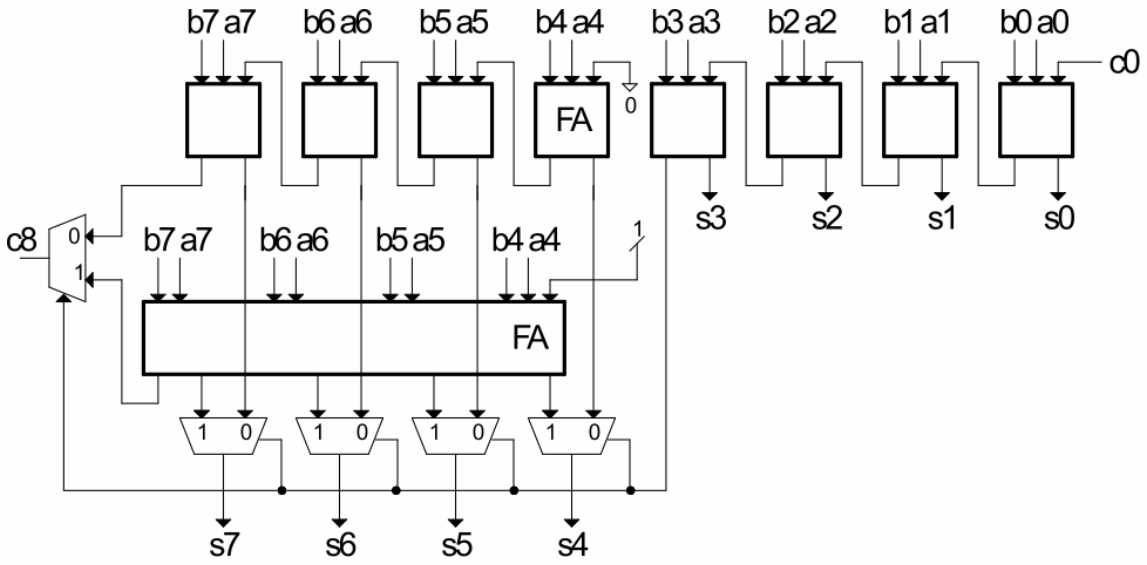
Kosten:

Zeit $\propto \log N * T_{add}$
Platz $\propto N * P_{add}$

Welche Eigenschaft von "+" nutzen wir aus?

Andere assoziative Operatoren? Boolesche Operationen?
Division? Min/Max?

Carry Select Addierer



Kosten:

$$\text{Zeit} \propto 0.5*N*T_{add} + T_{mux}$$
$$\text{Platz} \propto 1.5*P_{ripple} + (n/2+1)*P_{mux}$$

Der Addierer wird in zwei gleich lange Hälften unterteilt und für beide Hälften wird gleichzeitig mit der Addition begonnen.

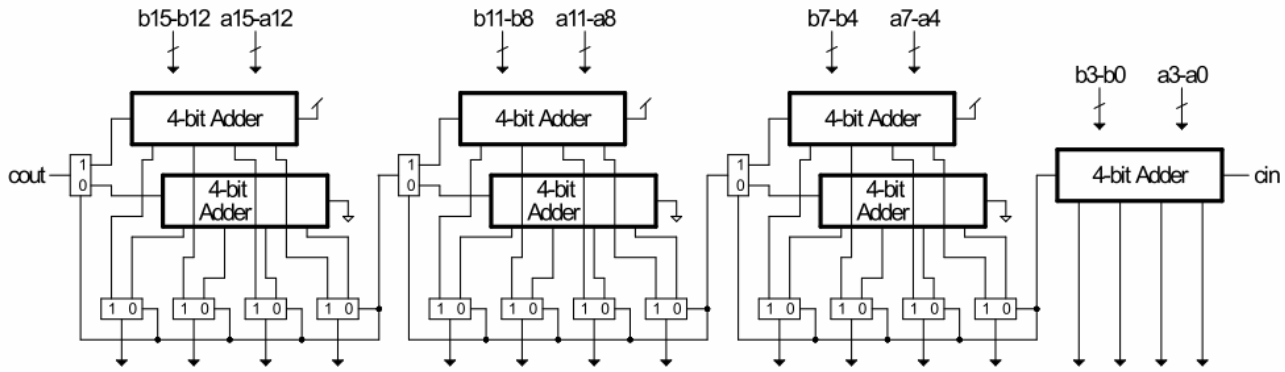
Bei der linken (höher signifikanten) Hälfte ist aber nicht bekannt, ob am Carry-Eingang des rechtesten Volladdierers eine 1 oder eine 0 ankommt.

Wenn die rechte Hälfte mit ihrer Addition fertig ist, ist das eingehende Carry der linken Hälfte bekannt.

Somit kann das richtige Ergebnis ausgewählt werden (select). Das andere (falsche) Ergebnis wird einfach verworfen.

Erweiterte Carry-Select-Addierer

16-bit CSA Addierer:



Was ist die optimale Anzahl von Blöcken und Bits/Block?

Viele kleine Blöcke, die Verzögerung wird durch die gesamte Mux-Verzögerung bestimmt.

Wenige grosse Blöcke, die Verzögerung wird von der Addierer dominiert.

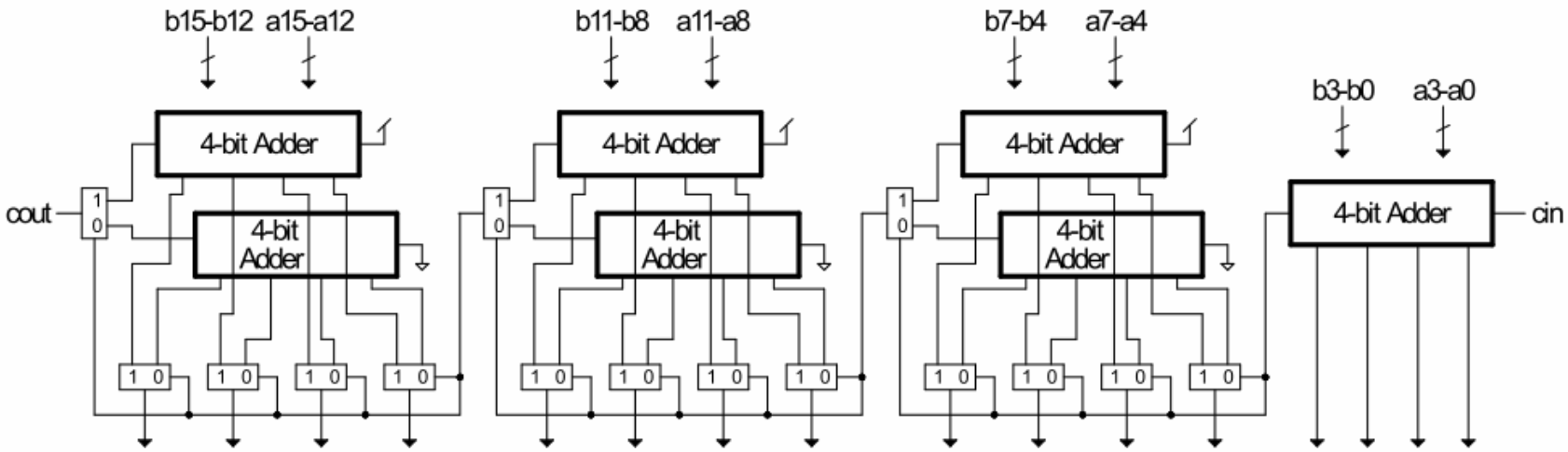
$\sqrt{N}$ Stufen von $\sqrt{N}$ bits

(Vereinfachter Fall mit einheitlicher Größe)

Kosten: Zeit $\propto \sqrt{N}$

Platz $\propto 2*P_{ripple}$ + mux

Carry-Select-Addierer Vergleich



Vergleich zu Ripple Addierer:

$T_{total} = 2*\sqrt{N}\ T_{add} - T_{add}$, mit der Annahme $T_{add} = T_{mux}$

Für Ripple-Addierer $T_{total} = N*T_{add}$

Carry-Select schneller für jeden Wert von N>3.

Carry Look-Ahead Addierer

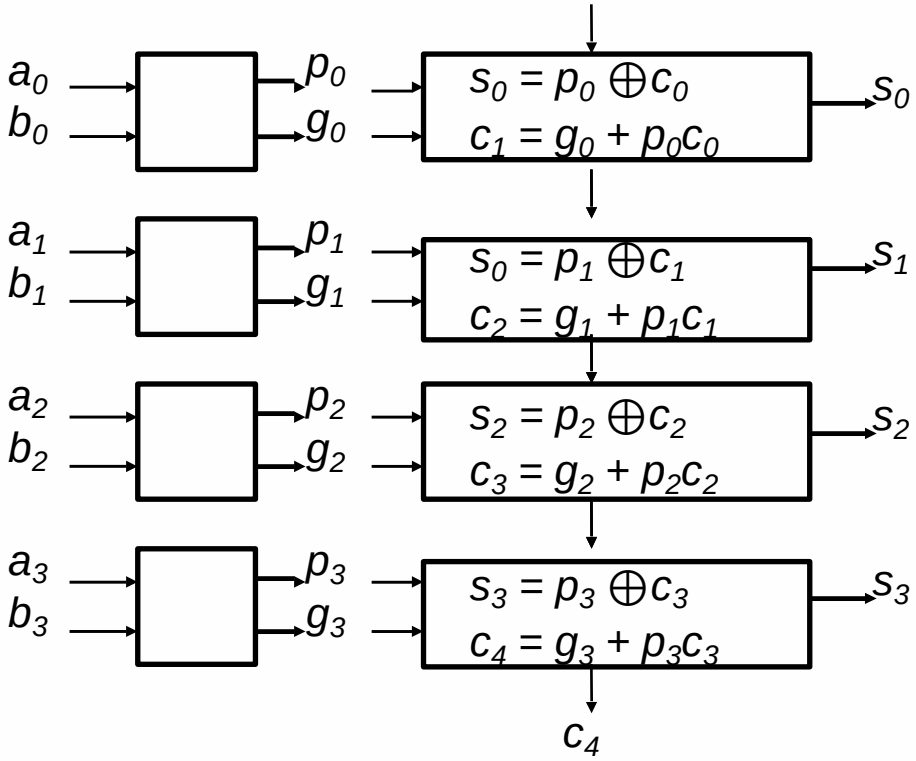
Wie können wir die Übertragserzeugung so gestalten, dass sie assoziativ ist?

Umformulierung der Basisaddierer-Stufe

a	b	c_i	c_{i+1}	s	
0	0	0	0	0	*carry "kill"*
0	0	1	0	1	$k_i = a_i' b_i'$
0	1	0	0	1	
0	1	1	1	0	*carry "propagate"*
1	0	0	0	1	$p_i = a_i \oplus b_i$
1	0	1	1	0	
1	1	0	1	0	*carry "generate"*
1	1	1	1	1	$g_i = a_i\, b_i$

$$c_{i+1} = g_i + p_i c_i$$
$$s_i = p_i \oplus c_i$$

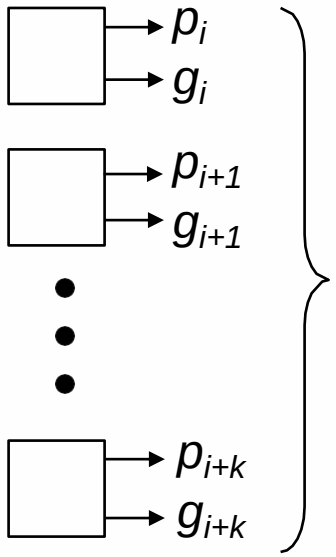
Ripple-Addierer mit P- und G-Signalen



$$p_i = a_i \oplus b_i$$
$$g_i = a_i\, b_i$$

Bislang kein Vorteil gegenüber dem Ripple-Addierer: Zeit $\propto N * T_{add}$

Gruppierung P- and G- Signalen



c_{in}

$$P = p_i\, p_{i+1} \ldots p_{i+k}$$

$$G = g_{i+k} + p_{i+k} g_{i+k-1} + \ldots + (p_{i+1} p_{i+2} \ldots p_{i+k}) g_i$$

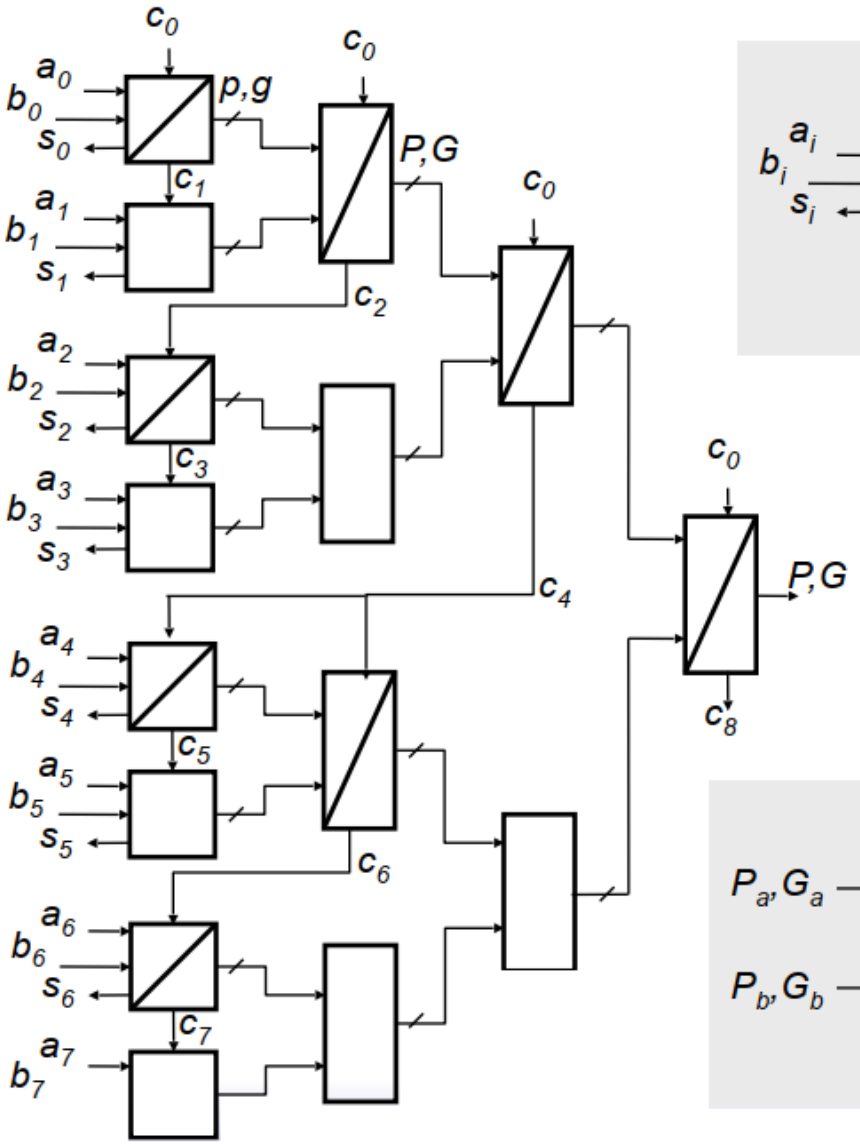
c_{out}

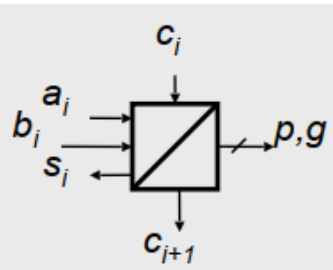
P wahr, wenn die Gruppe als Ganzes einen Übertrag nach c_{out} propagiert

G wahr, wenn die Gruppe als Ganzes einen Übertrag erzeugt $\boxed{c_{out} = G + Pc_{in}}$

Die Gruppen P und G können hierarchisch erzeugt werden.

Carry Look-Ahead Addierer

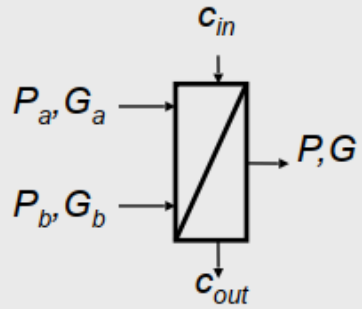




$p = a \oplus b$
$g = ab$

$s = p \oplus c_i$

$c_{i+1} = g + c_i p$



$P = P_a P_b$
$G = G_b + G_a P_b$

$C_{out} = G + c_{in} P$

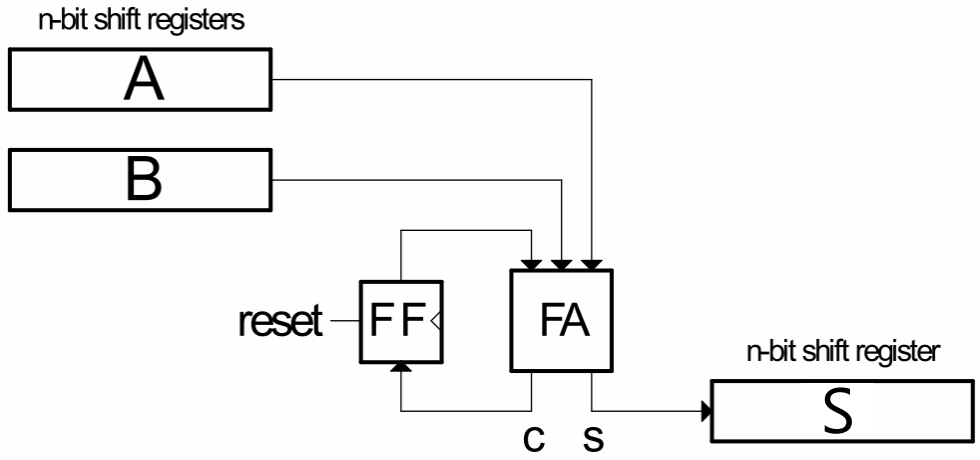
8-Bit Carry Look-Ahead Addierer

Sequentieller Bit Addierer

A, B und R werden in Schieberegistern gehalten.

Einmal pro Taktzyklus nach rechts schieben.
Reset wird vom Controller aktiviert.

Addition von 2 n-bit Zahlen benötigt n Taktzyklen



	A	B	S	s_i	c_{i+1}
t=0	1011	0011	0000	0	1
t=1	0101	0001	1000	1	1
t=2	0010	0000	1100	1	0
t=3	0001	0000	1110	1	0

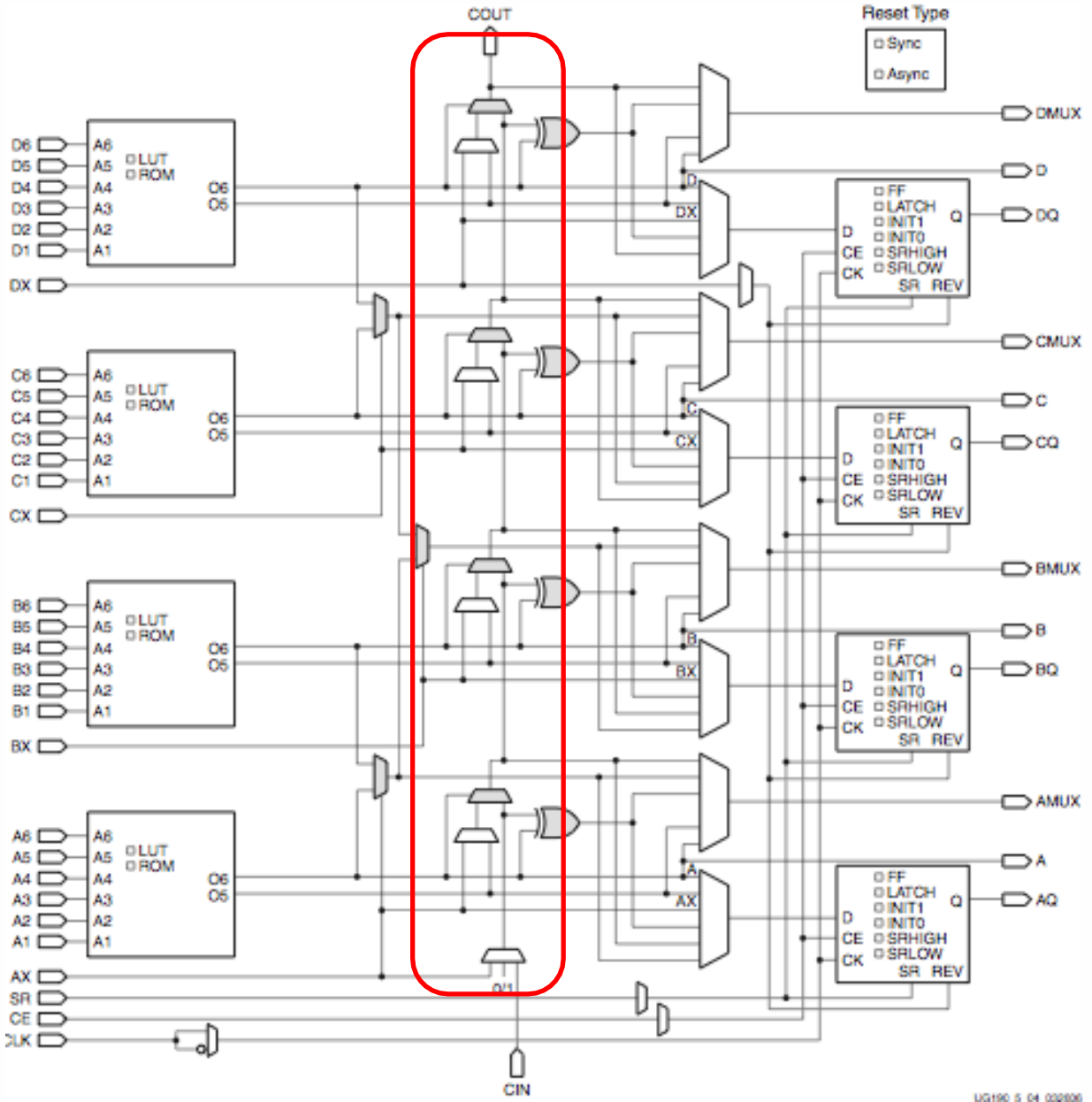
Addierer auf FPGAs

Dedizierte Carry-Logik bietet schnelle arithmetische Carry-Fähigkeit für Hochgeschwindigkeits-Arithmetikfunktionen.

Auf Virtex 5 FPGA

Verzögerung zwischen Cin und Cout (pro Bit) = 40ps, gegenüber 900ps zwischen F und X.

64-Bit-Additionsverzögerung = 2,5ns.



Addierer Zusammenfassung

Typ	Platz	Zeit
Ripple	O(N)	O(N)
Carry Select	O(N)	$O(\sqrt{N})$
Carry Look-Ahead	O(N)	O(log(N))
Bit-serial	O(1)	O(N)

Die "realen" Kosten der Carry-Select-Funktion belaufen sich auf mindestens das 2fache der "realen" Kosten der Ripple-Funktion.

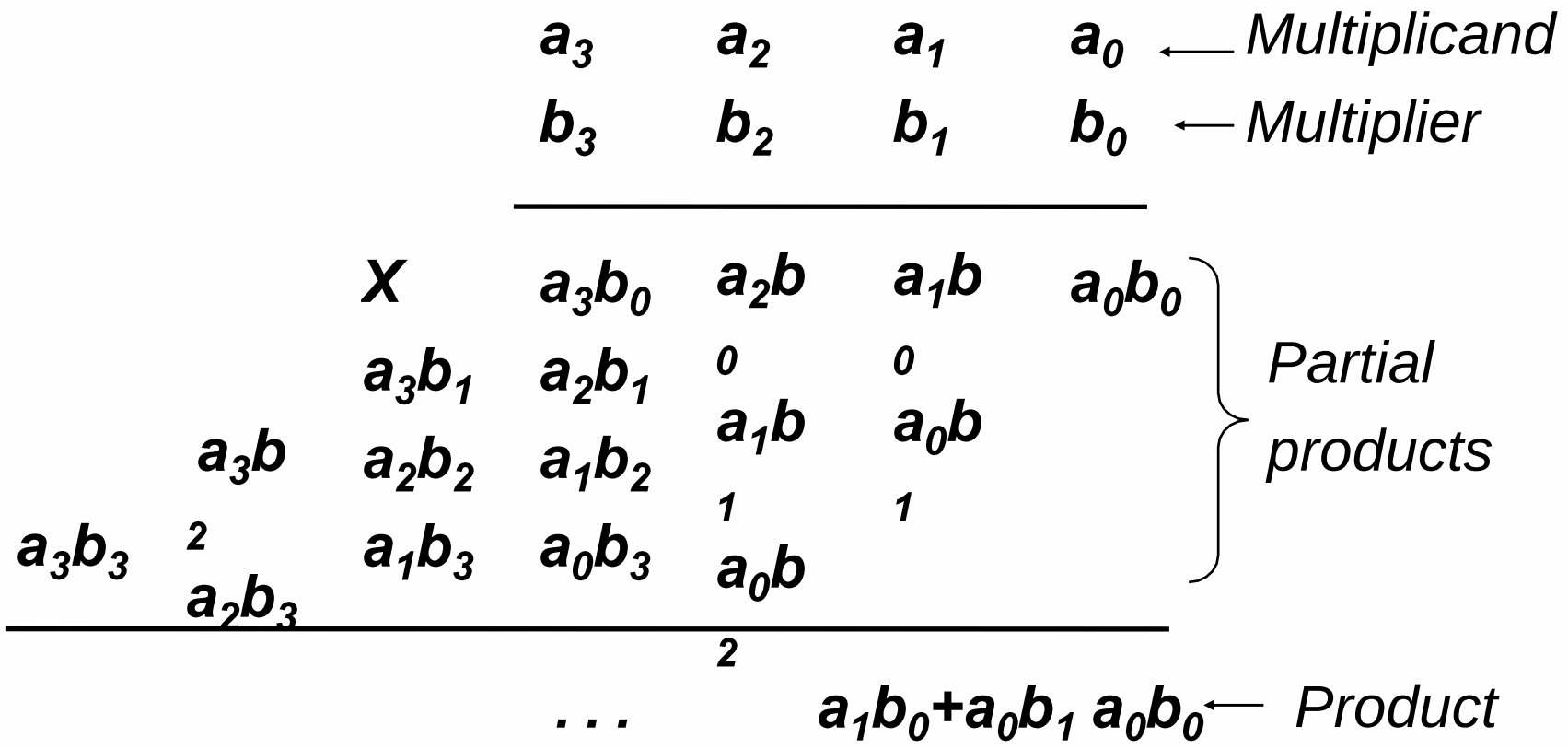
Die "realen" Kosten des CLA betragen wahrscheinlich mindestens das 2-fache der "realen" Kosten des Carry-Select.

Die tatsächlichen multiplikativen Konstanten hängen von den Implementierungsdetails und der Technologie ab. Die O-Notation versteckt diese Konstanten

FPGA- und ASIC-Synthesetools versuchen automatisch, die beste Addiererarchitektur zu wählen.

Multiplikation

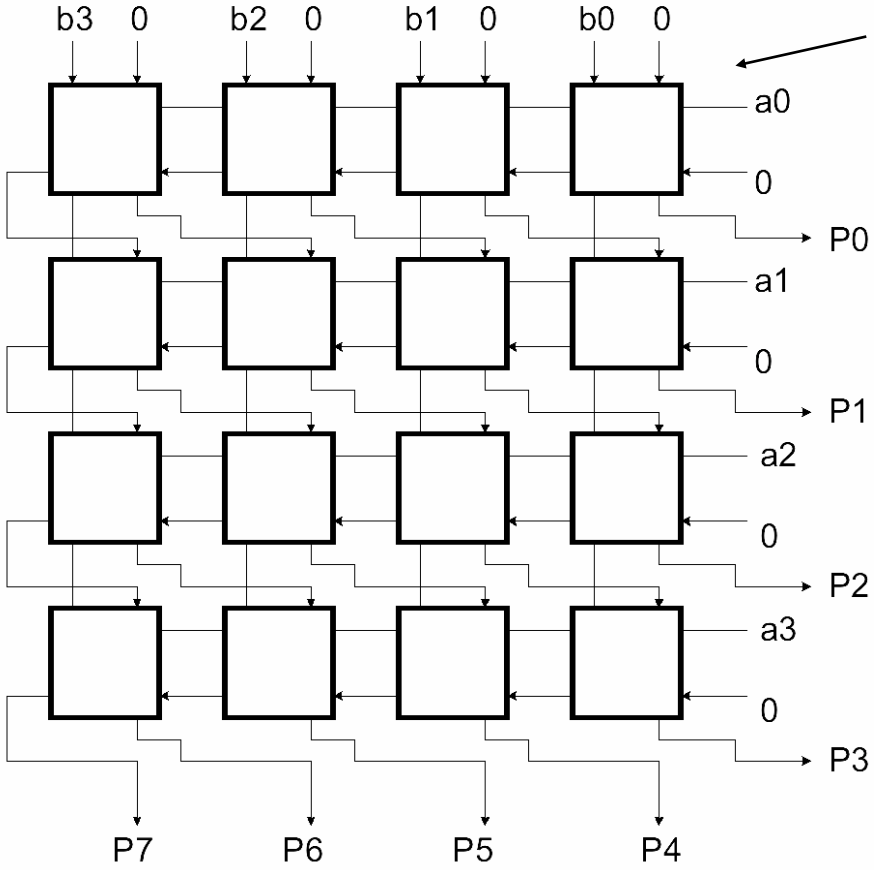
Manuelles Verfahren



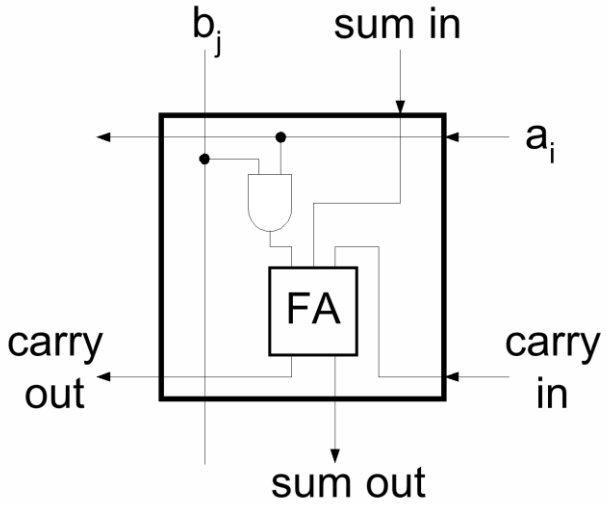
Für die Multiplikation gibt es auch viele verschiedene Schaltungen mit Kosten Trade-offs (Zeit/Platz)

Kombinierter Multiplikator ohne Vorzeichen

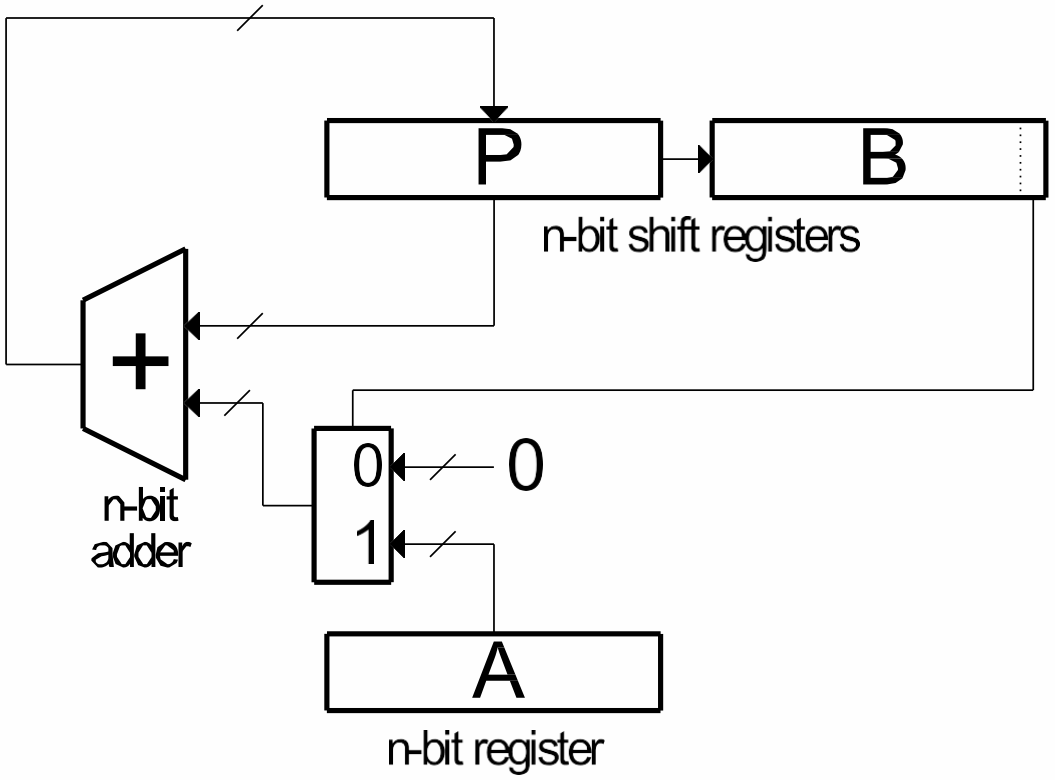
Einzyklische Multiplikation: Erzeugt alle n Teilprodukte gleichzeitig.



Jede Zeile: n-Bit-Addierer mit UND-Gattern



“Shift and Add” Multiplikator



Summiert jedes Teilprodukt, eins nach dem anderen.

Im Binärformat ist jedes Teilprodukt eine verschobene Version von A oder 0.

Steuerungsalgorithmus:

1. P ← 0, A ← multiplicand, B ← multiplier
2. If LSB of B==1 then add A to P
   else add 0
3. Shift [P][B] right 1
4. Repeat steps 2 and 3, n-1 more times.
5. [P][B] has product.

Kosten:

Zeit ∝ n Zyklen
Platz ∝ n

Division

Einige Prozessoren verwenden eigene Divisionseinheiten mit einer internen Darstellung der Zwischenergebnisse durch zwei Worte.

Die Division ist eine seltene Operation, weshalb viele Prozessoren auf dedizierte Hardware verzichten und sie stattdessen in Software implementieren.

Eine gängige Methode dafür ist die Newton-Raphson-Methode, die die Nullstelle einer Funktion durch eine konvergierende Folge von Werten approximiert.

Die Division a/b kann als a · 1/b berechnet werden, wobei der Kehrwert von b bestimmt wird. Bei Bedarf wird b normalisiert, und die Iteration reduziert den Aufwand von 2^n auf 3log n Operationen.

ALU

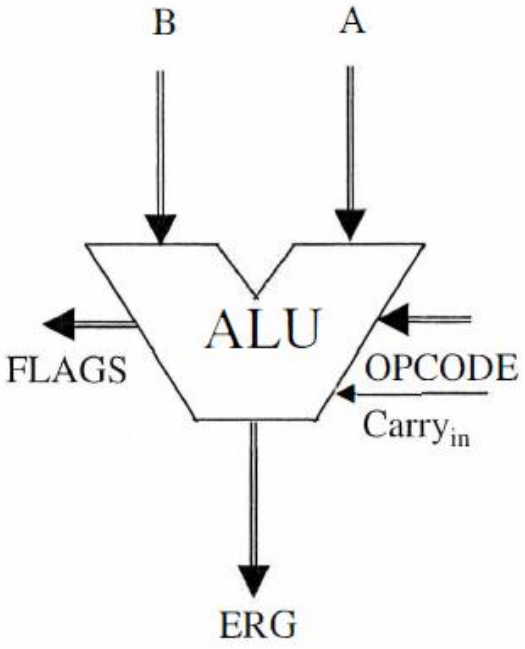
Arithmetisch-Logische Operationseinheit (ALU)

Die Folie zeigt den prinzipiellen Aufbau einer Arithmetisch-Logische Operationseinheit (ALU). Sie besteht typischerweise aus Funktionen zu binärer Arithmetik, logische Verknüpfungen und Schiebe-Operationen.

Die Steuereingänge wirken einerseits auf die Einheiten (Shifter, Logik, Addierer) selbst, andererseits wählen sie durch Steuerung zweier Datenweg-Multiplexer das Ergebnis aus der jeweils für den aktuell zuständigen Einheit, um es an den ERG-Ausgang der ALU weiterzuleiten.

Eingänge in eine ALU: zwei Operanden, Instruktionscode

Ausgänge einer ALU: Ergebnis, Flags

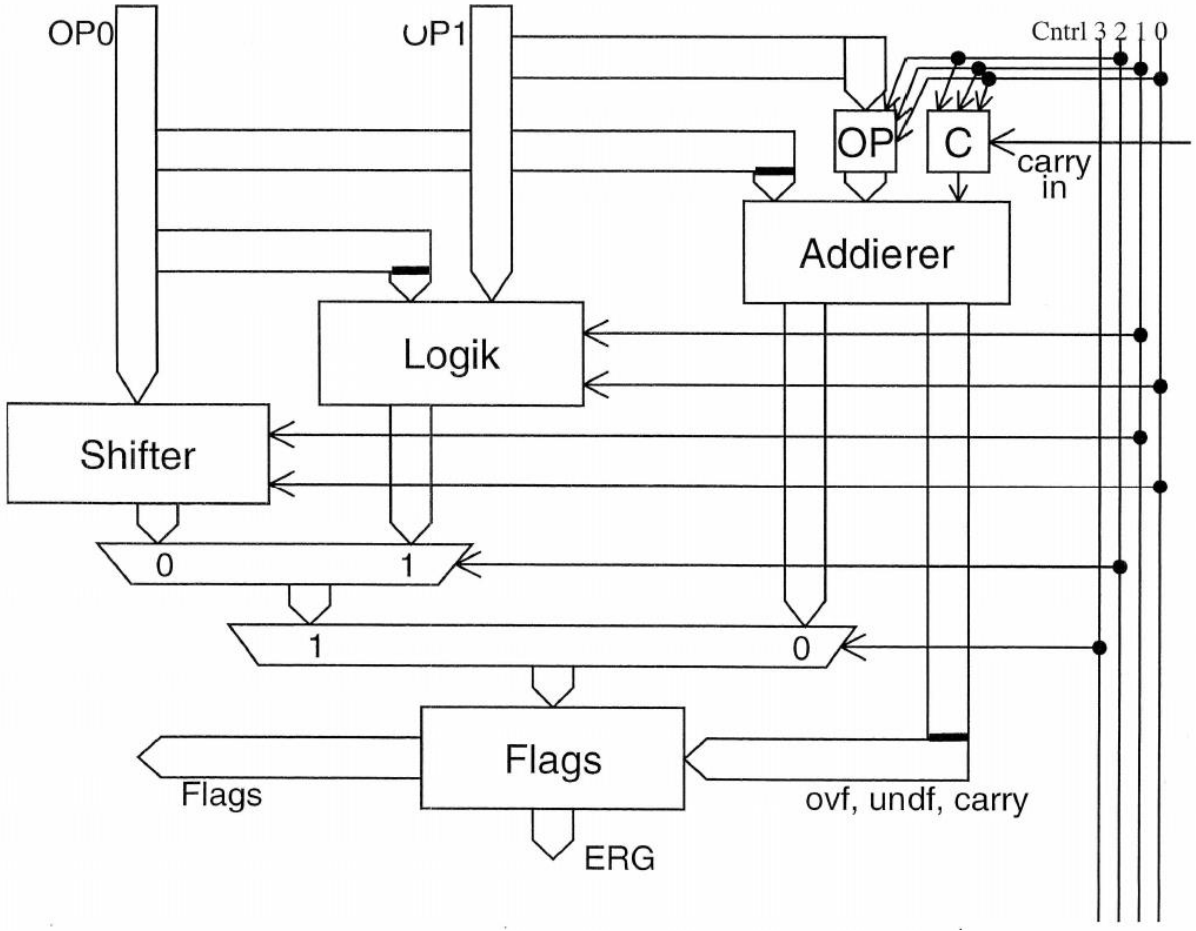


ALU-Komponenten

Die ALU besteht aus rein kombinatorischer Logik und, in dieser Beispiel, hat vier Steuereingänge mit den 16 verschiedene Mikrobefehle codiert werden können

Die 16 Befehle werden in vier Bits Cntrl3, ... , Cntrl0 codiert.

Dabei entscheidet Cntrl3, ob es eine arithmetische Operation ist oder nicht und Cntrl2 ob eine shift- oder logische Operation bzw. eine Addition oder Subtraktion ist.



Befehlssatz

Die Folie zeigt eine typische Auswahl der Operationen, die auf der ALU eines modernen RISC-Prozessors ausgeführt werden können.

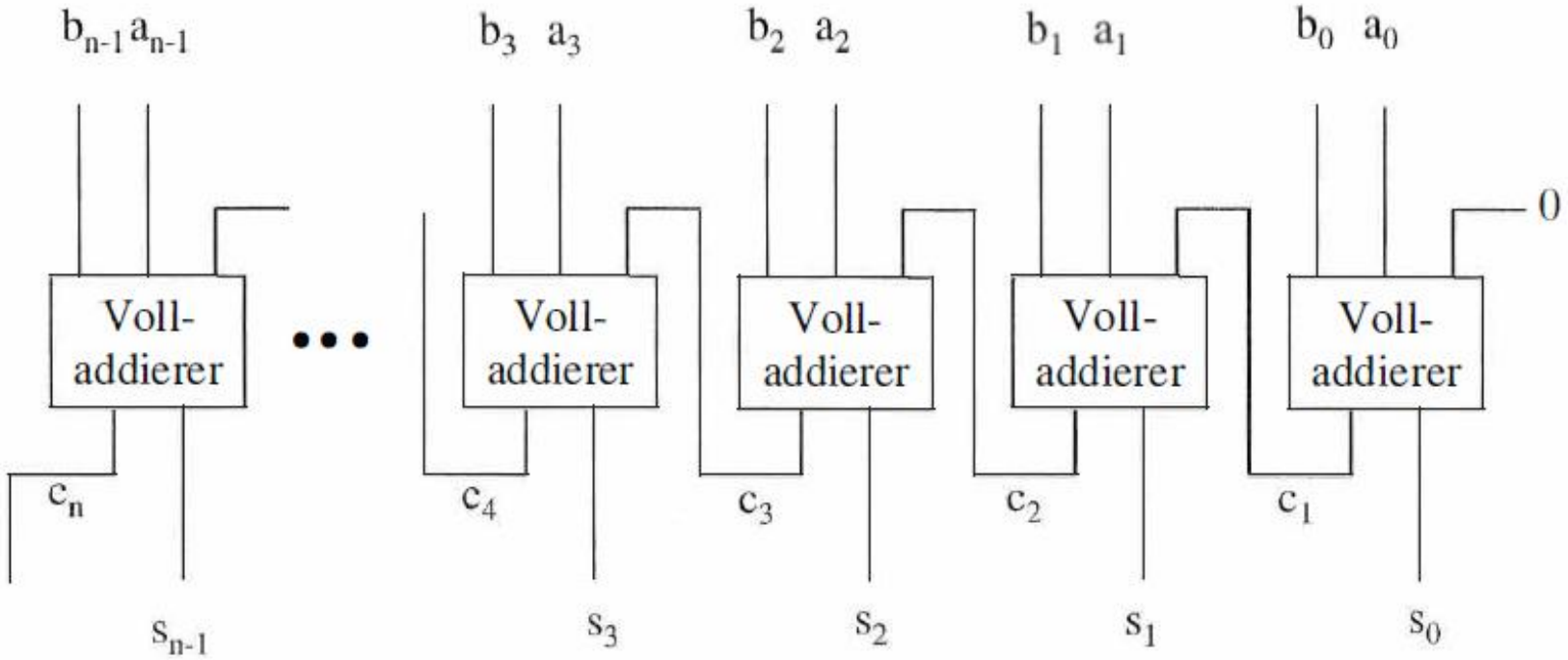
Man beachte, dass diese Befehlsauswahl einige Redundanz beinhaltet (der SET-Befehl ist zweimal vorhanden, die logischen Befehle könnten anders codiert werden usw.)

Wir entscheiden uns für diese einfache Version, um die Implementierung möglichst übersichtlich zu halten.

Befehl	**Bedeutung**	**Codierung**			
		cntrl3	cntrl2	cntrl1	cntrl0
SET	ERG:=OP0	0	0	0	0
DEC	ERG:=OP0-1	0	0	0	1
ADD	ERG:=OP0+OP1	0	0	1	0
ADC	ERG:=OP0+OP1+$Carry_{in}$	0	0	1	1
SET	ERG:=OP0	0	1	0	0
INC	ERG:=OP0+1	0	1	0	1
SUB	ERG:=OP0-OPI	0	1	1	0
SBC	ERG:=OP0-OP1-$Carry_{in}$	0	1	1	1
SETF	ERG:=0	1	0	0	0
SLL	ERG:=2*OP0	1	0	0	1
SRL	ERG:=OP0 div 2	1	0	1	0
SETT	ERG:= 1	1	0	1	1
NAND	ERG:=OP0 NAND OP1	1	1	0	0
AND	ERG:=OP0 AND OP1	1	1	0	1
NOT	ERG:=NOT OP0	1	1	1	0
OR	ERG:=OP0 OR OP1	1	1	1	1

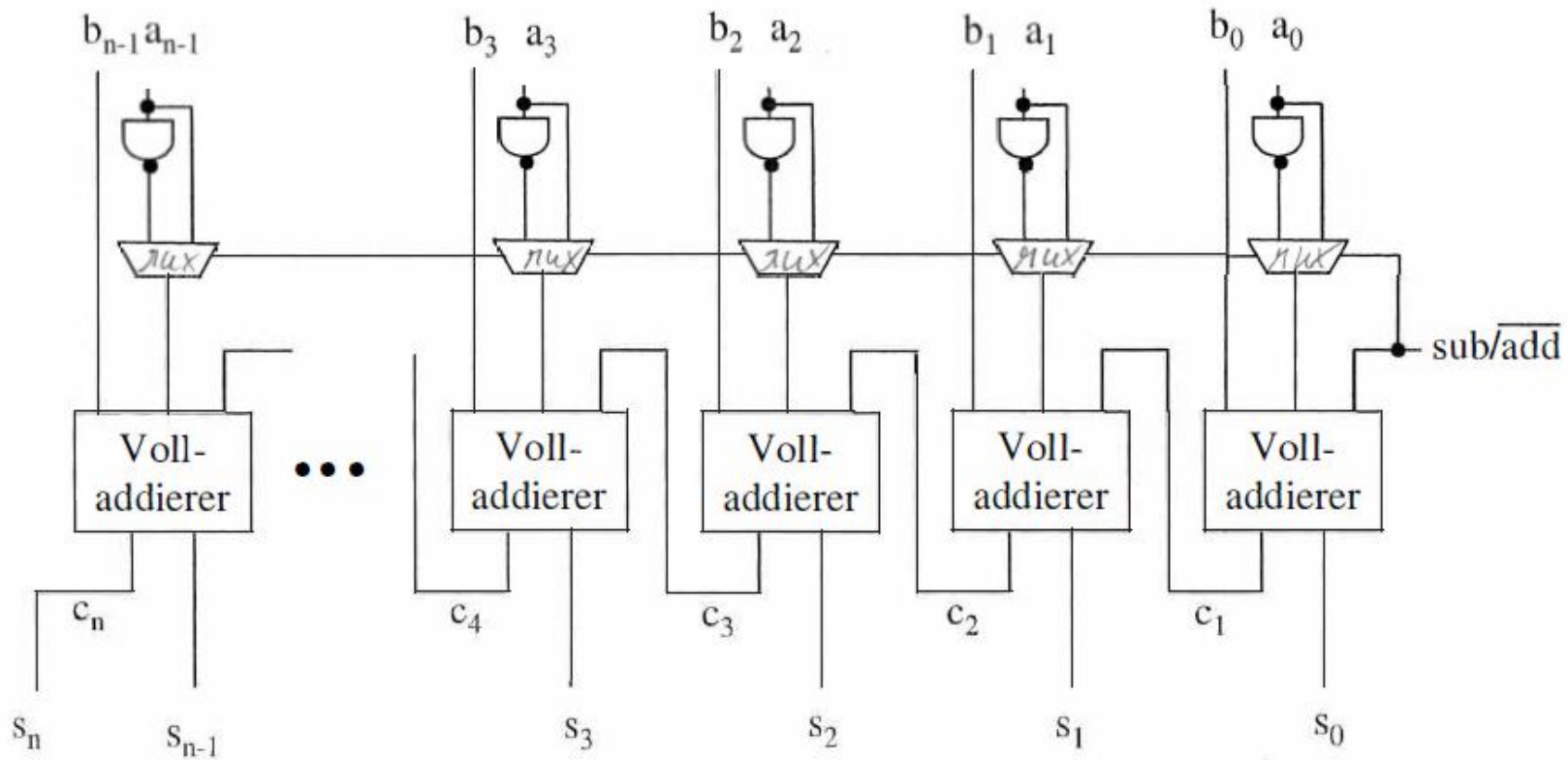
Kernstück jeder ALU ist ein Addierer

Ripple-Carry-Addierer



Subtraktion

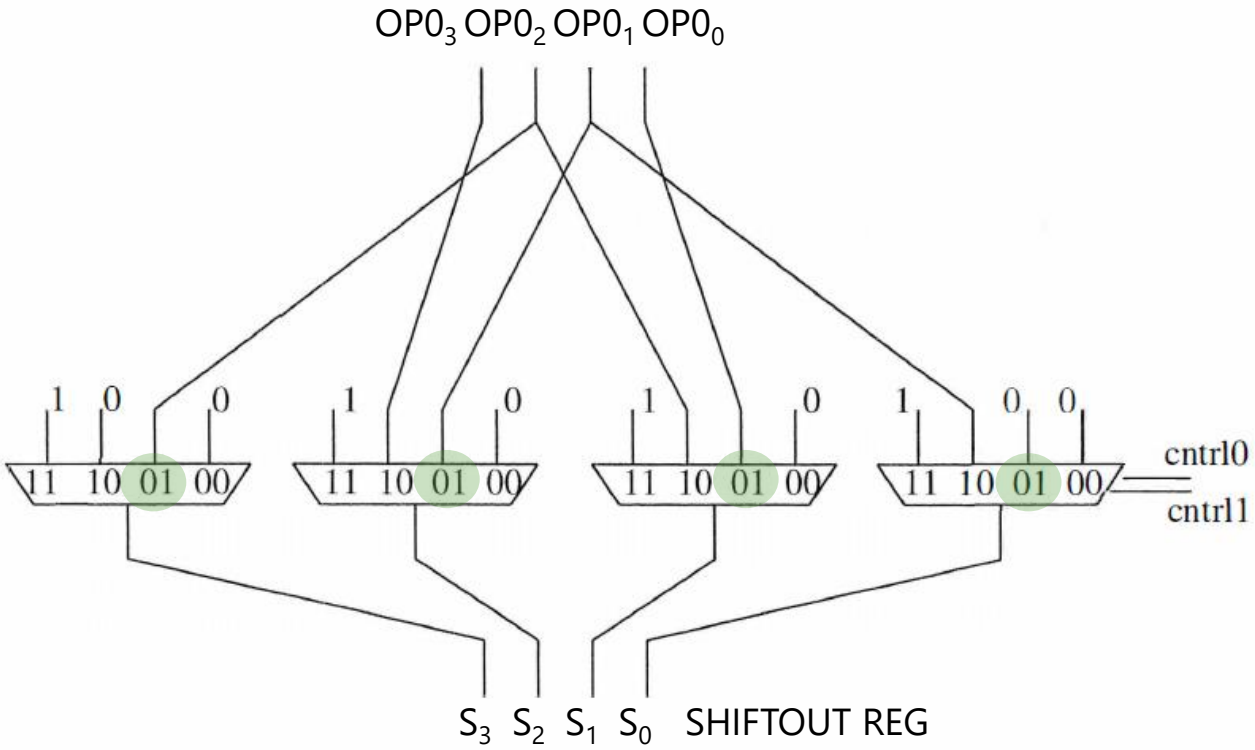
Das Zweierkomplement wird gebildet, indem wir jedes Bit invertieren (Einer-Komplement) und noch 1 addieren. Um für diese Addition nicht einen weiteren Addierer zu benötigen, nehmen wir einfach den Carry-Eingang des Addierers, in den wir bei der Subtraktion eine 1 anstelle der 0 (bei der Addition) eingeben.



ALU Shifter

Aufgabe des **Shifters** ist, die Befehle SETF (setze auf 0), SLL (shift logical left), SLR (shift logical right) und SETT (setze alle bits auf 1).

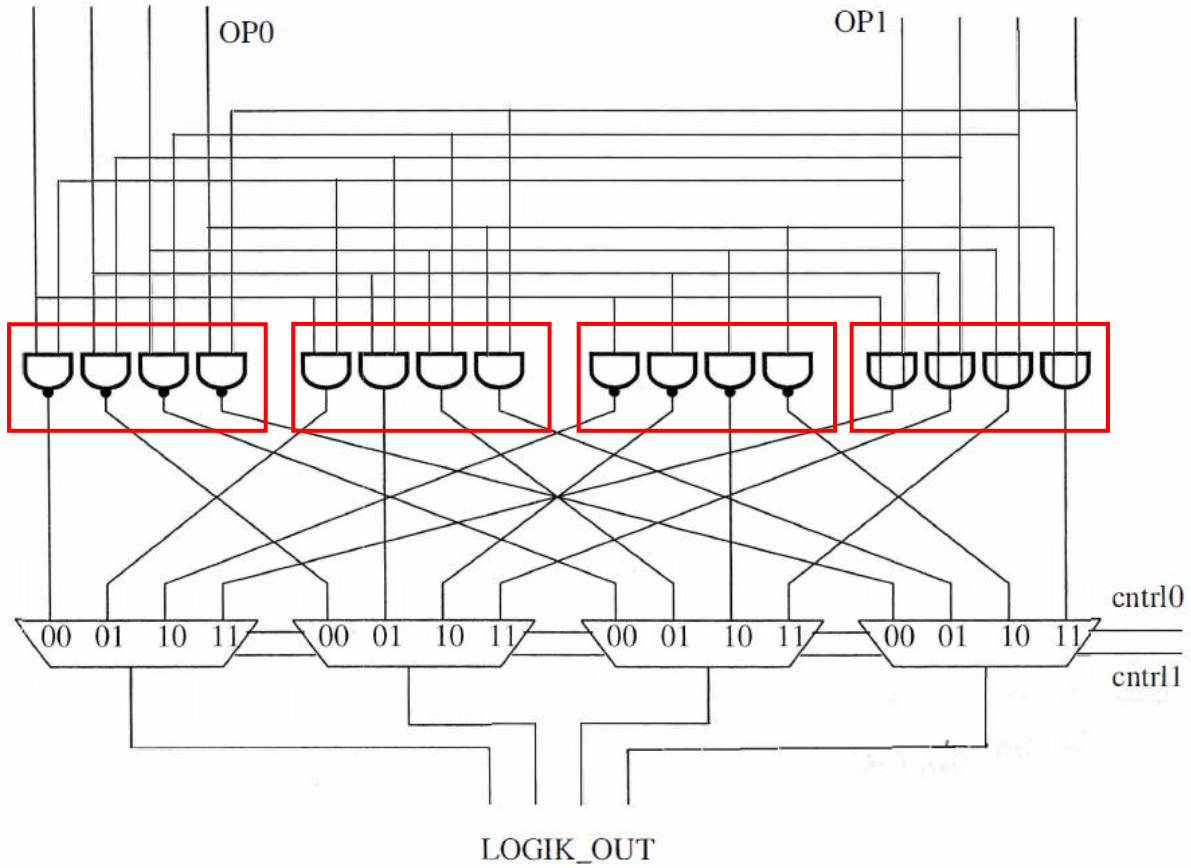
Diese Funktionen können mit einfachen 4-auf-1 Multiplexern wahrgenommen werden, einen für jedes Bit des Ergebnisses. Bei den beiden Shift-Befehlen wird das jeweils neu eingeschobene Bit auf 0 gesetzt und das herausgeschobene Bit wird verworfen.



Beispiel:
SLL (Befehlcode 1001)

Logischen Befehle

NAND, AND, NOT, oder OR werden gesondert bit-weise berechnet und über Multiplexer wird das durch den aktuellen Befehl geforderte Ergebnis ausgewählt.



ALU Operanden

Der Addierer/Subtrahierer hat als einen Operanden immer OP0. Der zweite zu addierende Operand kann OP1, 0, -1, 1, oder –OP1 sein, je nachdem, ob addiert, subtrahiert, inkrementiert, dekrementiert oder unverändert durchgereicht werden soll.

Bei der **einfachen Addition** ist der zweite Eingang gleich OP1, der Carry-Eingang gleich 0. Bei der Addition mit Berücksichtigung des alten Carryin ist der Eingang gleich OP 1, der Carry-Eingang gleich Carryin·

Bei der **Subtraktion** werden alle Bits von OP1 invertiert und der Carry-Eingang ist 1. Auf diese Weise wird das Zweierkomplement von OP1 zu OP0 addiert. Bei der Subtraktion mit Carry werden die Bits von OP1 invertiert und Carryin wird an den Carry-Eingang des Addierers gelegt.

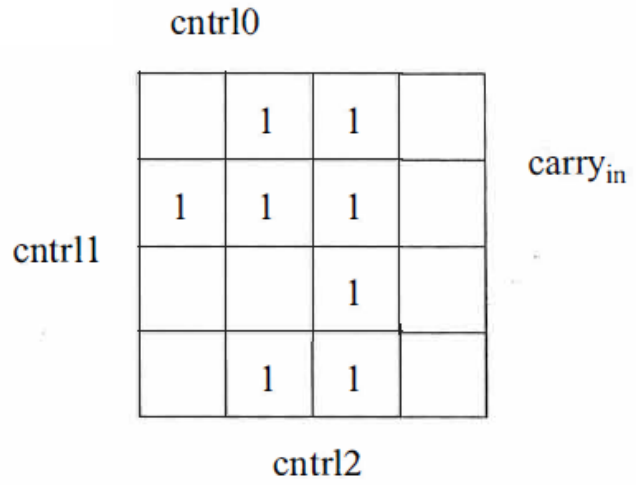
Bei **SET-Befehlen** wird OP0 + 0 berechnet. Es gibt zwei SET-Befehle. Beim ersten wird +0 addiert, beim zweiten -0 subtrahiert. Also ist beim ersten der zweite Addierer-Eingang gleich 0 und das Carry ist 0 und beim zweiten ist der zweite Addierereingang auf -1 (alle Bits sind 1) und das Carry auf 1.

Beim **Inkrementieren** ist der zweite Addierer-Eingang auf 0 und der Carry-Eingang auf 1, beim **Dekrementieren** ist der zweite Addierer-Eingang auf -1 (alle Bits sind 1) und der Carry-Eingang ist auf 0.

Arithmetisch-Logische Operationseinheit (ALU)

Das **Schaltnetz für den Carry-Eingang wird** wie folgt abgeleitet.
Dabei ist das Schaltnetz nur einmal für den Carry-Eingang des ersten Volladdieres vorzusehen.

Befehl	carry	cntrl2	cntrl1	cntrl0	C-Eingang
set	0	0	0	0	0
dec	0	0	0	1	0
add	0	0	1	0	0
adc	0	0	1	1	0
set	0	1	0	0	1
inc	0	1	0	1	1
sub	0	1	1	0	1
sbc	0	1	1	1	0
set	1	0	0	0	0
dec	1	0	0	1	0
add	1	0	1	0	0
adc	1	0	1	1	1
set	1	1	0	0	1
inc	1	1	0	1	1
sub	1	1	1	0	1
sbc	1	1	1	1	1

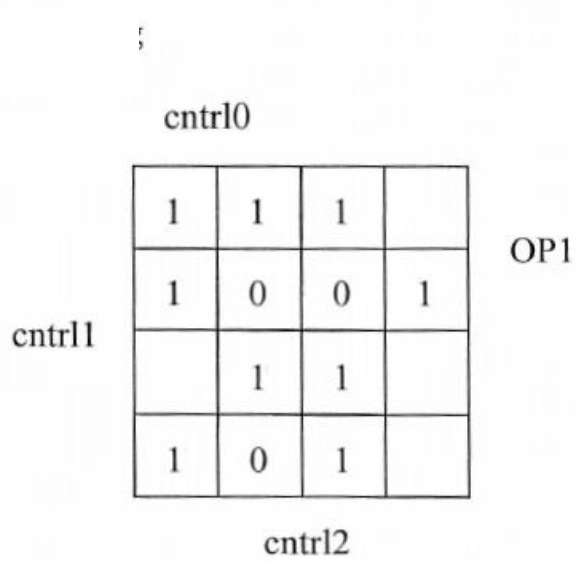


$$\text{C-Eingang} = \text{cntrl2}\,\overline{\text{cntrl1}} + \overline{\text{cntrl0}}\,\text{cntrl2} + \text{cntrl0}\,\text{cntrl1}\,\text{carry}_{in}$$

Arithmetisch-Logische Operationseinheit (ALU)

Die **Schaltnetz für zweiten Eingang des Addierers** wird wie folgt abgeleitet. Dabei ist das Schaltnetz für den zweiten (OP1) Eingang des Addierers **für jedes Bit einzeln** vorzusehen.

Befehl	OP1	cntrl2	cntrl1	cntrl0	OP-Eingang
set	0	0	0	0	0
dec	0	0	0	1	1
add	0	0	1	0	0
adc	0	0	1	1	0
set	0	1	0	0	1
inc	0	1	0	1	0
sub	0	1	1	0	1
sbc	0	1	1	1	1
set	1	0	0	0	0
dec	1	0	0	1	1
add	1	0	1	0	1
adc	1	0	1	1	1
set	1	1	0	0	1
inc	1	1	0	1	0
sub	1	1	1	0	0
sbc	1	1	1	1	0



$$\text{OP-Eingang} = \text{cntrl0}\,\overline{\text{cntrl1}}\,\text{OP1} + \text{cntrl0}\,\overline{\text{cntrl2}}\,\text{OP1} + \overline{\text{cntrl1}}\,\text{cntrl2}\,\text{OP1} + \text{cntrl1}\,\text{cntrl2}\,\overline{\text{OP1}} + \overline{\text{cntrl0}}\,\overline{\text{cntrl1}}\,\text{cntrl2}$$

Über- und Unterläufe

Wie bereits aus Kapitel 1 bekannt, können Über- und Unterläufe bei der Addition erkannt werden anhand eines zusätzlichen Sicherungsbits, das eine Kopie des höchstsignifikanten Bits darstellt. Man benutzt nun für eine m-Bit-Addition einen m+1-Bit Addierer, wobei die Sicherungsstelle ganz normal mitaddiert wird.

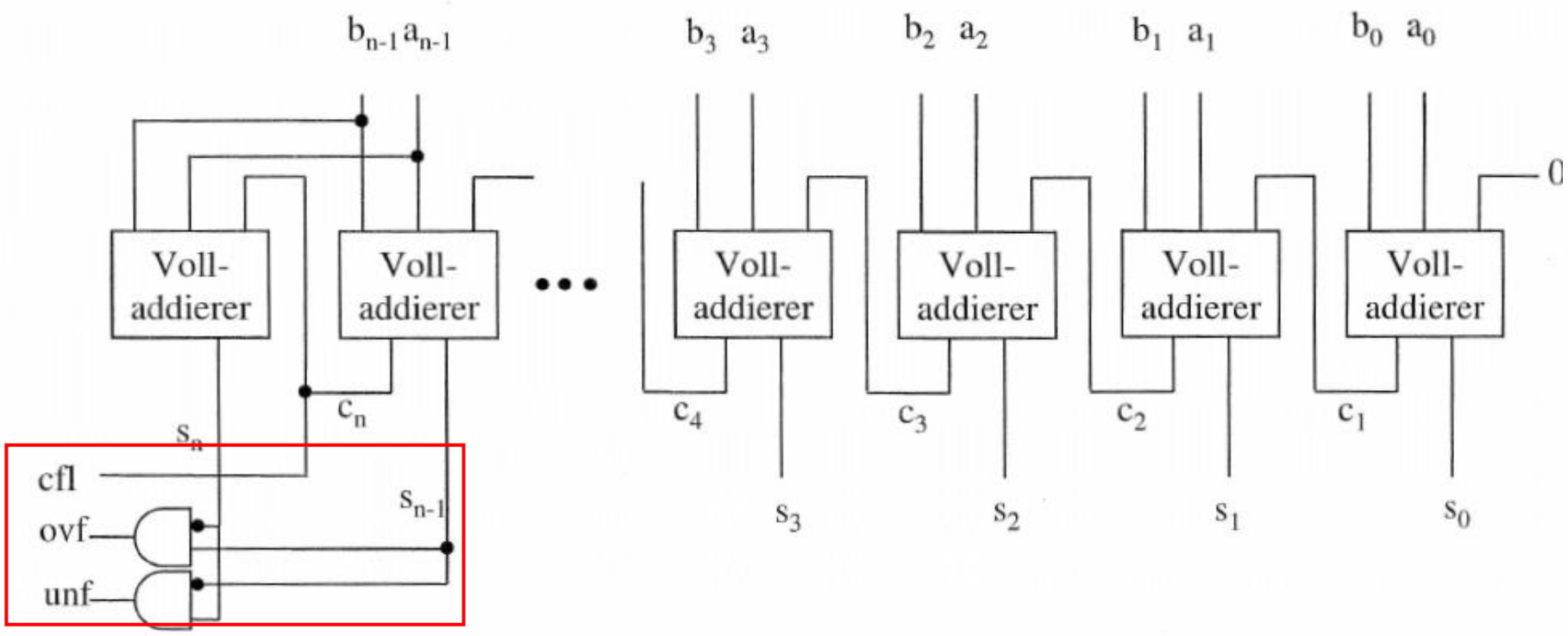
Das Ergebnis der Addition ist also die Summe s_{m-1}, s_{m-2}, ... , s_1 , s_0, das ausgehende Carry-Bit (= Carry-Flag) cm, sowie ein künstliches Summenbit s_m.

Das künstliche Carry-Bit c_{m+1} hat keine arithmetische Bedeutung. Ein ***Überlauf*** (Ergebnis ist größer als die größte darstellbare Zahl) ist nun aufgetreten, wenn $s_{m-1}=0$ und $s_m=1$.

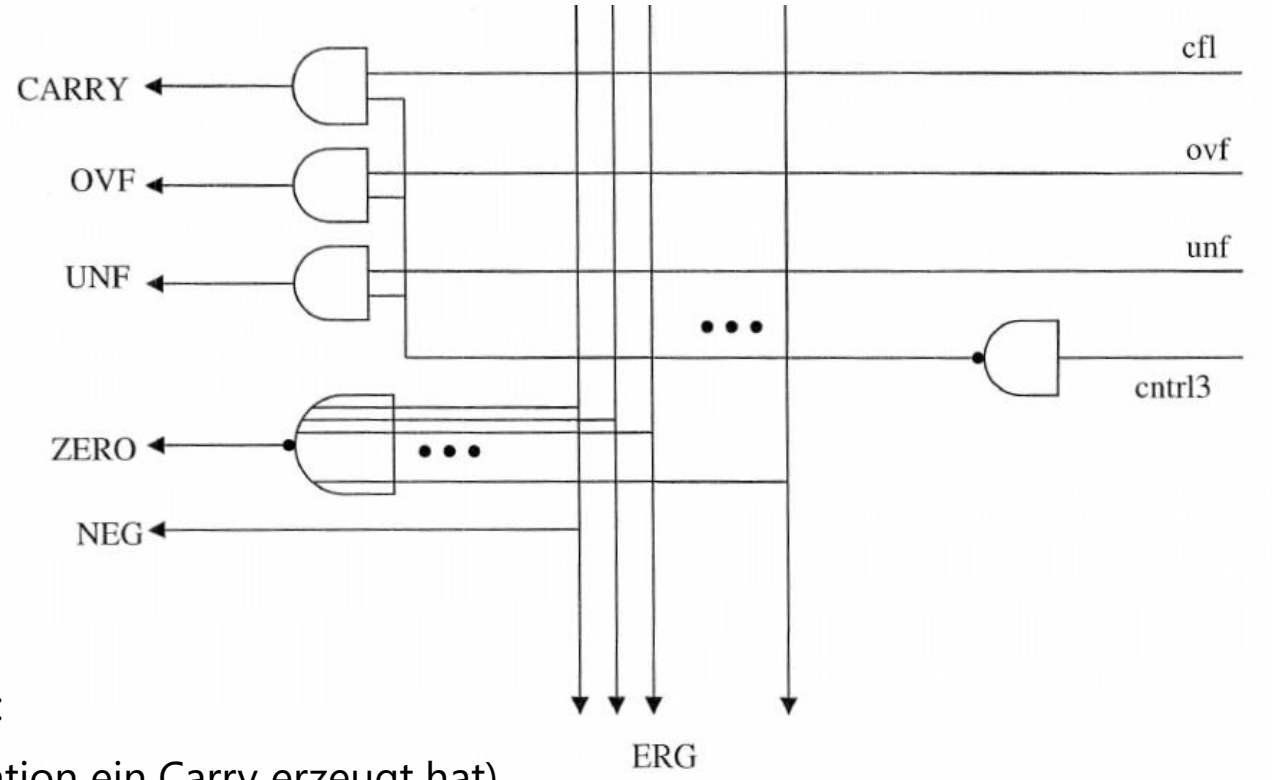
Wenn $s_m=0$ und $s_{m-1}=1$ ist, hat ein ***Unterlauf*** (Ergebnis ist kleiner als die kleinste darstellbare Zahl) stattgefunden. Wenn $s_{m-1}=s_m$, ist die Addition ist fehlerfrei verlaufen.

Arithmetisch-Logische Operationseinheit (ALU)

Das Schaltnetz zeigt die Ergänzung unseres Addierers, mit der wir **Über- und Unterläufe erkennen** und als Flags (ovf (overflow) und unf (underflow)) an eine übergeordnete Einheit übermitteln können.



ALU Flags



Fünf **Flags** sollen insgesamt generiert werden:

Carry-Flag (=1, falls eine arithmetische Operation ein Carry erzeugt hat)

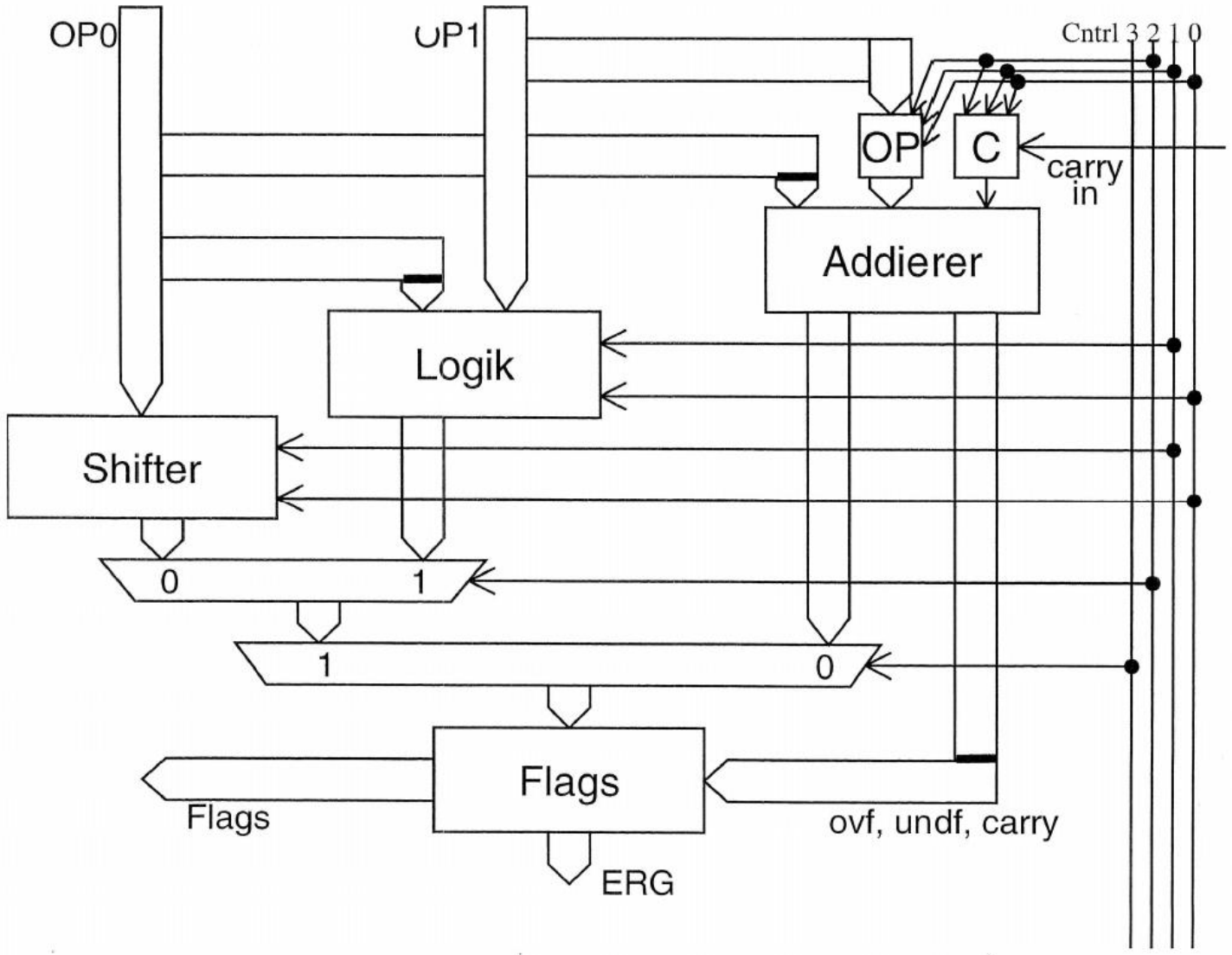
Neg-Flag (=1, wenn das Ergebnis eine negative Zahl darstellt)

Zero-Flag (= 1, wenn das Ergebnis gleich 0 ist)

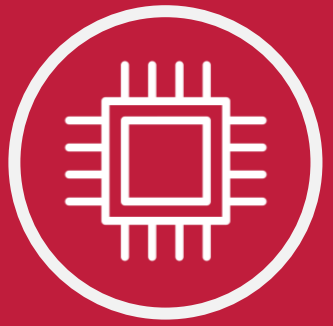
ovf-Flag (=1, wenn eine arithmetische Operation einen Überlauf erzeugt hat)

unf-Flag (=1, wenn eine arithmetische Operation einen Unterlauf erzeugt hat)

Arithmetisch-Logische Operationseinheit (ALU)



5. Mikrorechner



Speicher

Was gibt es für Speichermedien?

Einteilung nach der Bauart des Speichermediums:

Magnetbänder (Speicherkapazität: >100 TBytes pro Band, Zugriffszeit: bis 3 Minuten): wichtig für die Speicherung von‚Cloud-Daten' oder als Backup Speicher

Plattenlaufwerke (magnetisch oder optisch, Speicher- kapazität im TByte Bereich, Zugriffszeit von einigen ms): Daten werden auf die Oberfläche von Scheiben geschrieben

Halbleiter (Silizium integrierte Schaltung, **DRAM**, Kapazität im GBytes Bereich, Zugriff innerhalb 50 ns): flüchtiger elektronischer Speicherbaustein mit wahlfreiem Zugriff

Halbleiter (Silizium integrierte Schaltung, **Flash**, Speicher- kapazität von >2 TBytes, Zugriff in dem μs Bereich): nichtflüchtige Speicherung mit niedrigem Energieverbrauch

Was gibt es für Speicherfunktionen?

Einteilung nach der Funktion:

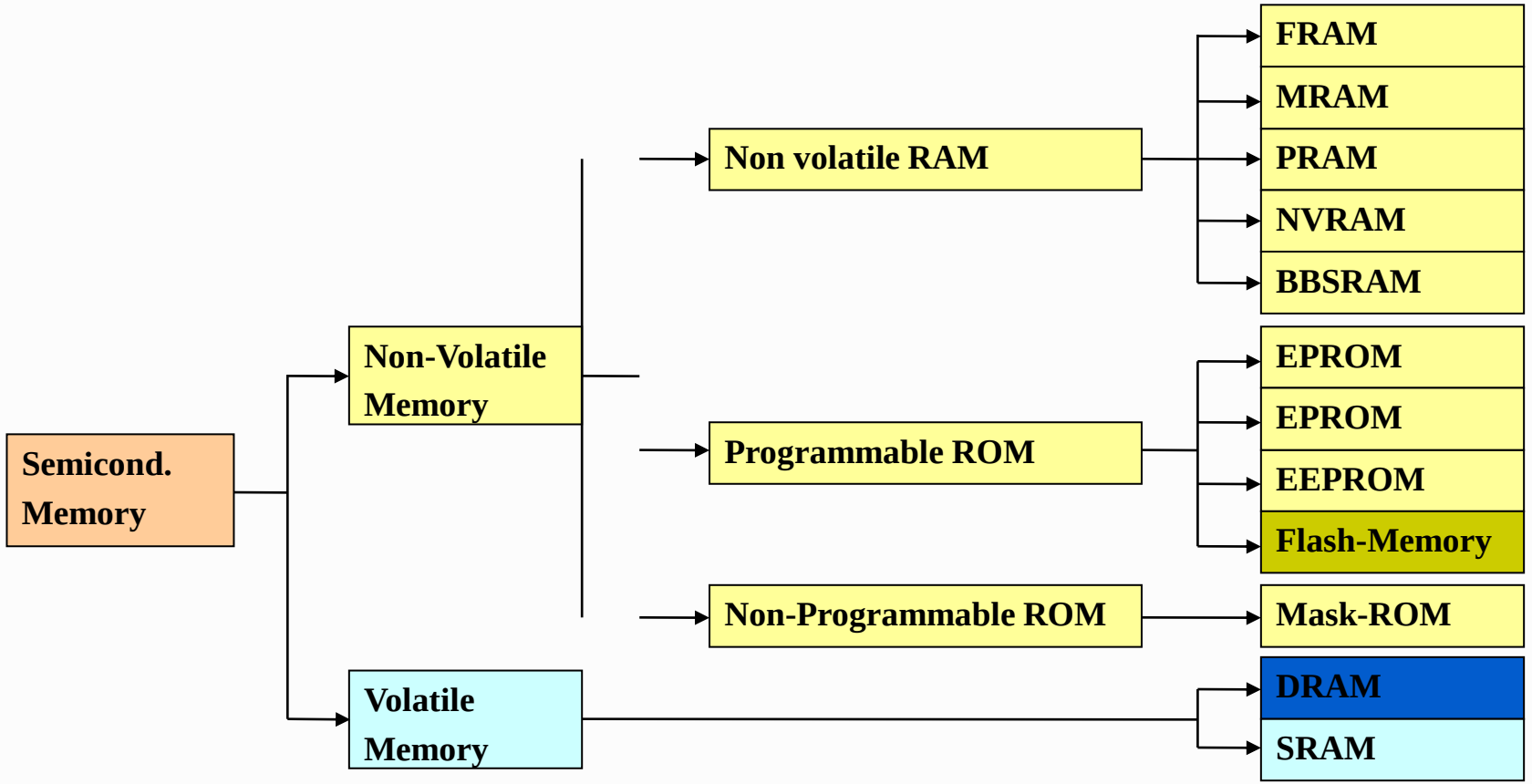
ROM: ‚read only memory' (nur lesen)

RAM: ‚random access memory' (gleichwertiges Lesen und Schreiben, wahlfreier Zugriff, d.h. Lese- und Schreibezeit unabhängig von der Position der Zelle)

Statischer oder **dynamischer** Speicher: optisch / elektronisch

Permanente (nichtflüchtige) oder nur **temporäre** (flüchtige) Speicherung (in Abhängigkeit von der Betriebsspannung)

Arten von Halbleiterspeichern



Eigenschaften der HL-Speicherarten

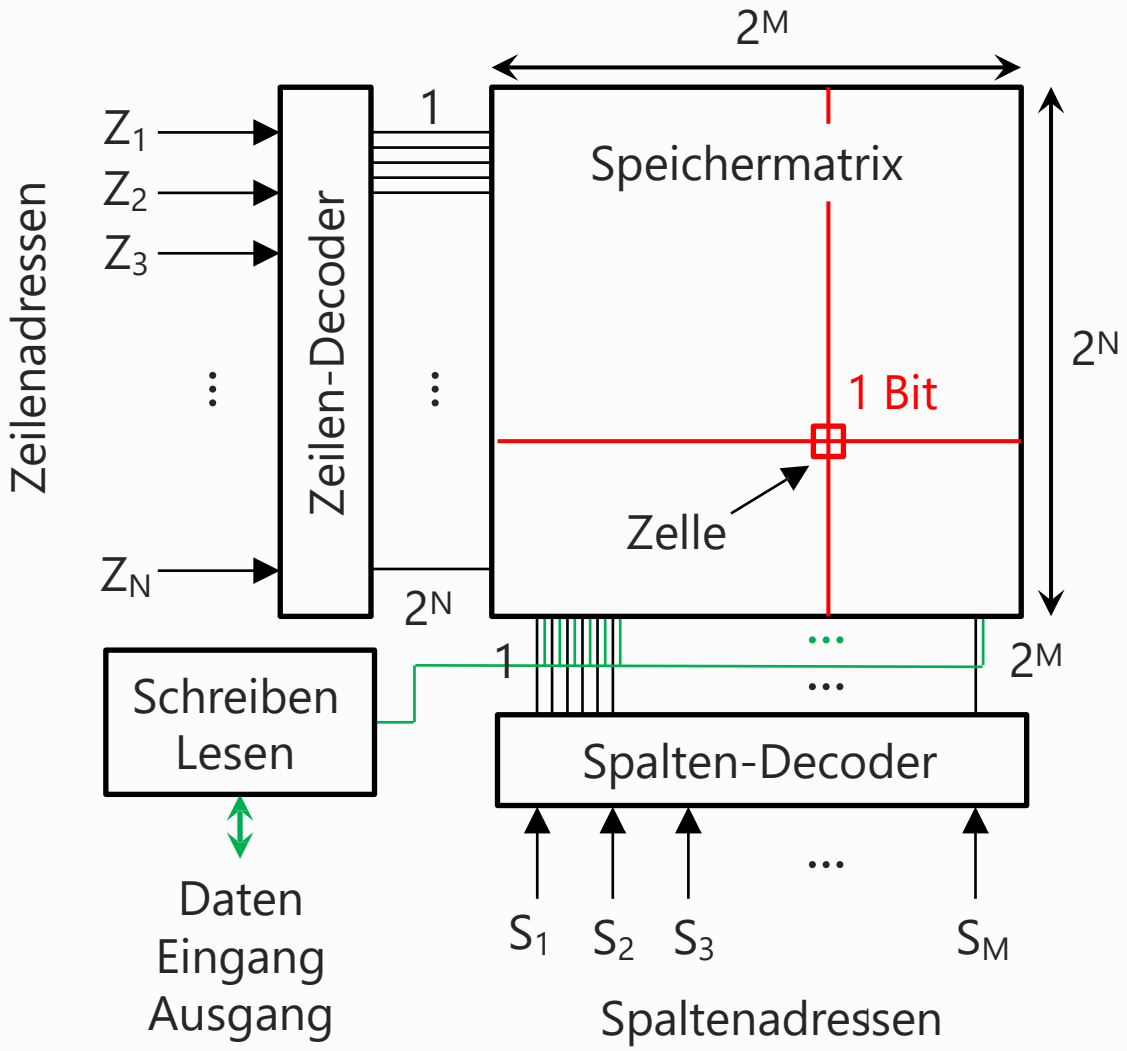
Typ	Daten-erhalt beim Ab-schalten	Notwen-digkeit des Refresh	Wieder-beschreib-barkeit	erreichbare Dichte	erreichbare Schreib-geschwin-digkeit	erreichbare Lese-geschwin-digkeit	technolo-gische Komplexität	Aufwand bei der monolithtischen System-integration	gegen-wärtige Produkt-reife
ROM	ja	nein	nein	sehr hoch	-	mäßig	gering	kein	sehr hoch
PROM	ja	nein	nein	mäßig	gering	mäßig	gering	hoch	sehr hoch
EPROM	ja	nein	ja, aber UV löschen	gering	gering	mäßig	mäßig	sehr hoch	sehr hoch
EEPROM	ja	nein	ja	gering	gering	mäßig	hoch	sehr hoch	sehr hoch
Flash-EEPROM	ja	nein	ja	hoch	gering	hoch	hoch	sehr hoch	hoch
SRAM	nein	nein	ja	mäßig	sehr hoch	sehr hoch	gering	kein	sehr hoch
DRAM	nein	ja	ja	sehr hoch	hoch	hoch	mäßig	hoch	sehr hoch
FeRAM	ja	nein	ja	sehr hoch	hoch	hoch	mäßig	mäßig	mäßig
MRAM	ja	nein	ja	sehr hoch	hoch	hoch	mäßig	mäßig	gering

Typische Speicherorganisation

Halbleiter Speicherelemente werden oft als Zellenarray oder Matrizen organisiert, die von einem Zeilen-Decoder und einem Spalten-Decoder angesteuert werden

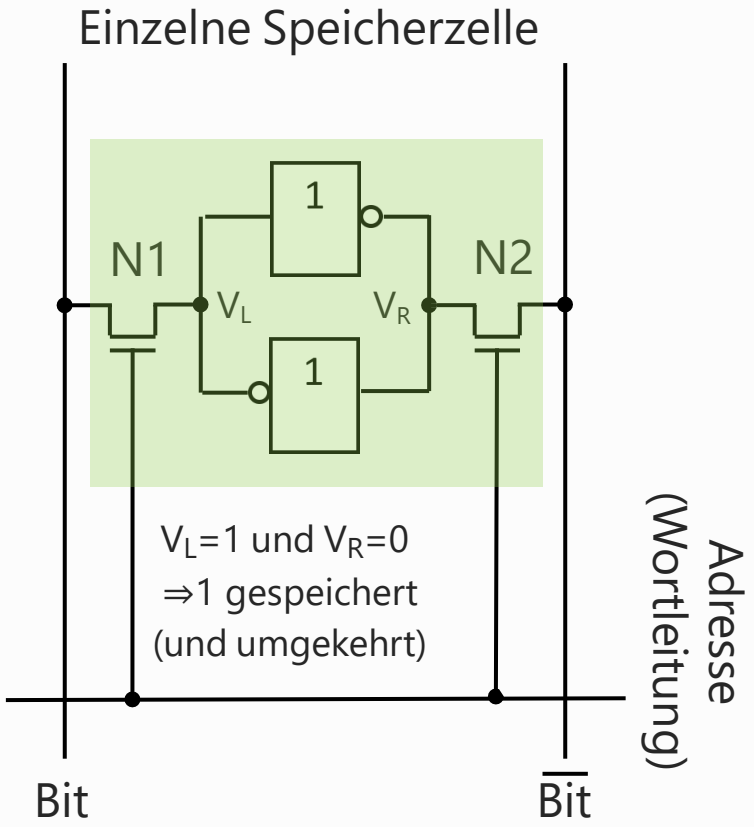
Eigenschaften:

- 2^N Zeilen
- 2^M Spalten
- $2^N \times 2^M$ Bits werden gespeichert
- Lesen und schreiben sind möglich (**RAM**)
- Decoder nötig, um die richtige Zelle (Bit) zu selektieren



SRAMs

Es gibt zwei Gruppen von flüchtigen Speichern: SRAM und DRAM. Statische ‘random access memories’ (**SRAMs**) können Information speichern, solange die Versorgungsspannung angelegt ist. SRAMs finden überall Anwendung, wo Daten **schnell** im Zugriff sein müssen, wie in Prozessoren als Cache



SRAM Eigenschaften:

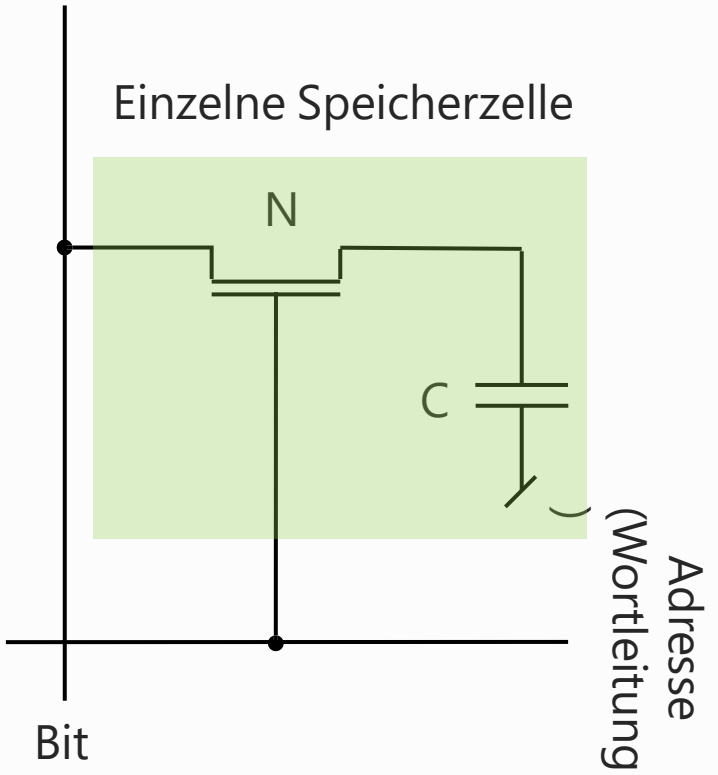
Eine Zelle besteht aus 6 Transistoren (2 pro Inverter, 2 Schalttransistoren N1 und N2)

Mit der **Wortleitung** kann eine Speicherzelle angewählt werden

Über die **Bitleitungen** kann der Speicherinhalt gelesen oder gesetzt werden

DRAMs

Dynamische ‘random access memories’ (**DRAMs**) können Daten speichern, die während des Betriebs periodisch (20 ms) wieder aufgefrischt werden müssen. DRAMs werden oft als Arbeitsspeicher in Prozessoren eingesetzt. Eine DRAM Zelle besetzt 4 Mal weniger Platz als eine SRAM Zelle (mehr Dichte) aber die Zugriffszeiten sind länger (langsamer)



DRAM Eigenschaften:

Eine Zelle besteht aus einem Schalttransistor (N) und einem Kondensator (C). Wenn C geladen (leer) ist, ist eine 1 (0) gespeichert

Über die Bitleitungen kann der Speicherinhalt gelesen oder gesetzt werden, über die Wortleitung wird eine Zelle selektiert