ALU

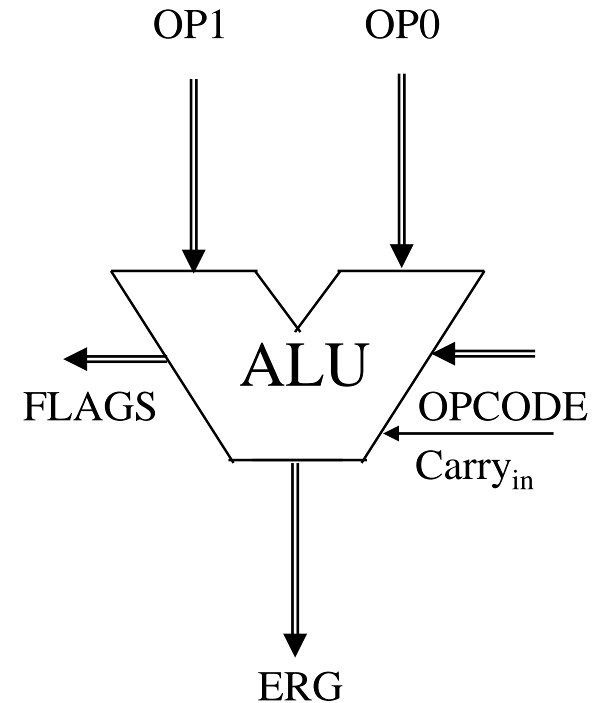
ALU-Aufbau

Eine ALU (arithmetisch-logische Einheit) besteht in der Regel aus

- Addierer
- Logischer Einheit
- Shifter

Eingänge in eine ALU: zwei Operanden, Instruktionscode

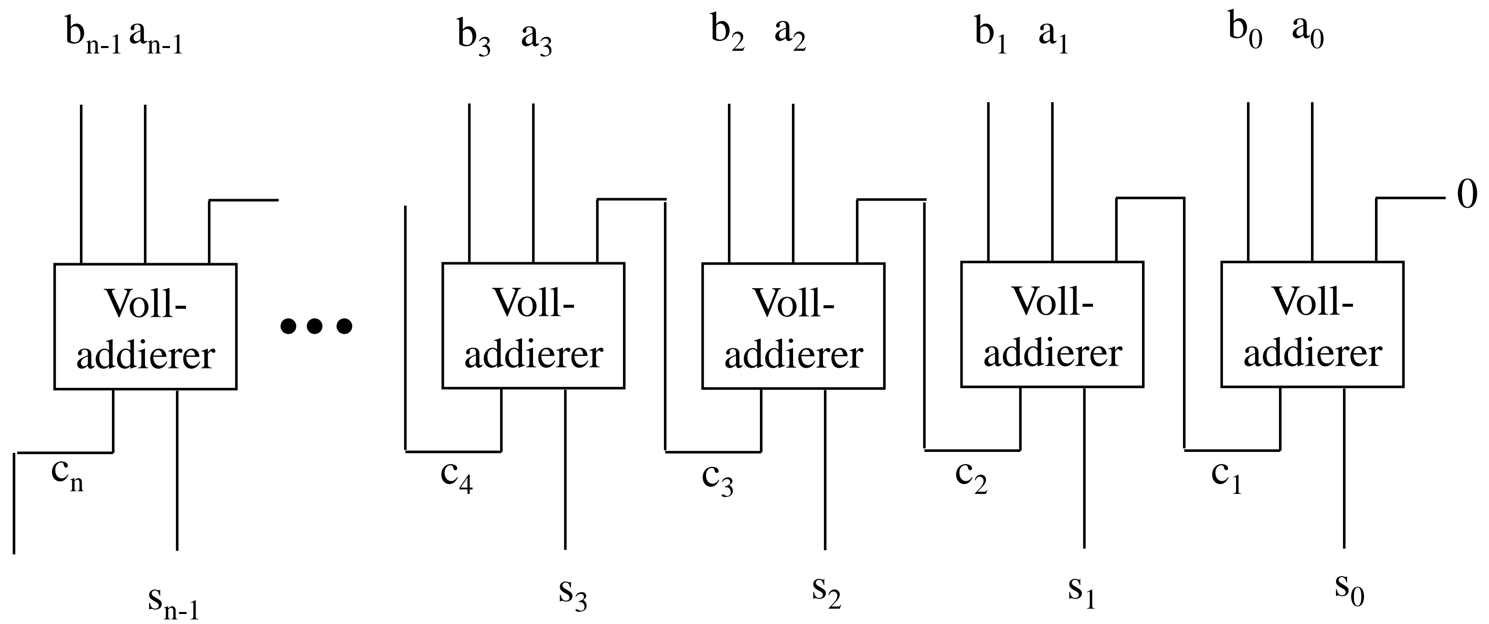
Ausgänge einer ALU: Ergebnis, Flags



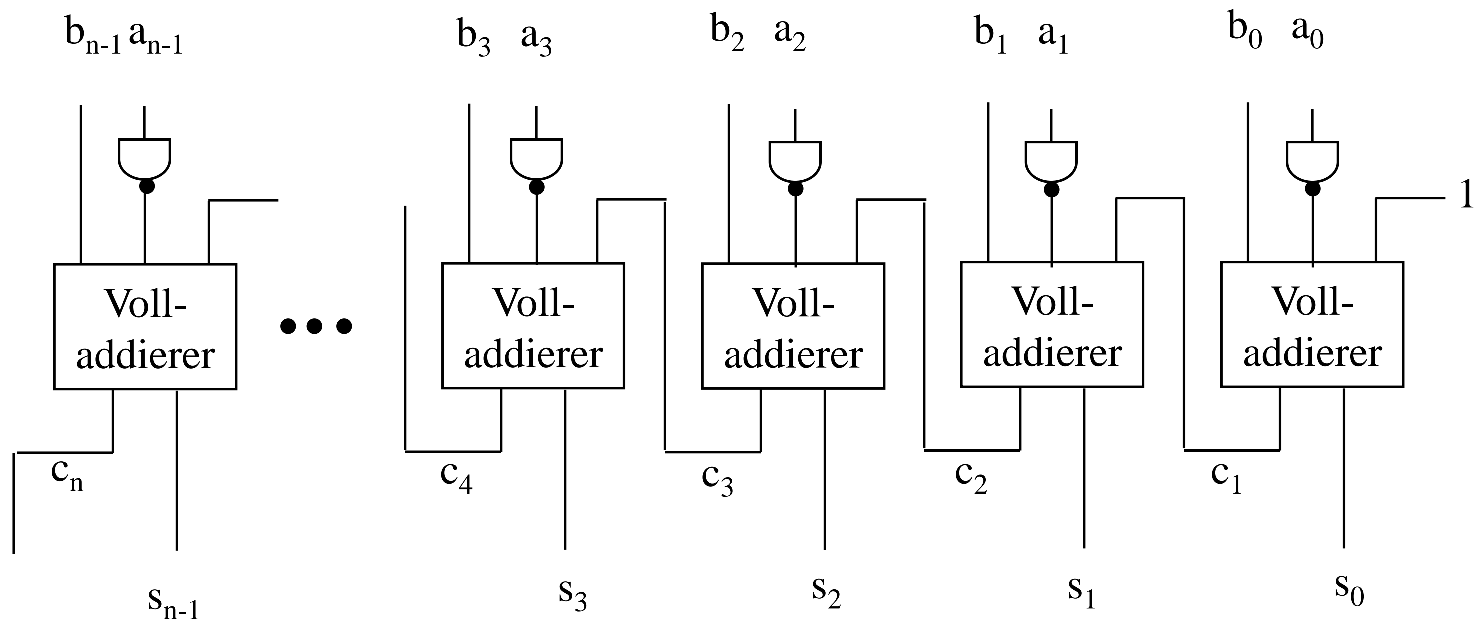
Addierer

Kernstück jeder ALU ist ein Addierer. Wir sehen einen Ripple-Carry-Addierer auf der nächsten Folie.
Wie können wir diesen aber auch zum Subtrahieren benutzen?
Indem wir das Zweierkomlement des einen Operanden bilden und an den einen Eingang des Addierers führen.
Wie bilden wir das Zweierkomplement?
Indem wir jedes Bit invertieren (Einer-Komplement) und noch 1 addieren. Um für diese Addition nicht einen weiteren Addierer zu benötigen, mißbrauchen wir einfach den Carry-Eingang des Addierers, in den wir bei der Subtraktion eine 1 anstelle der 0 (bei der Addition) eingeben. Ein entsprechendes Schaltnetz sehen wir auf der übernächsten Folie.
Später werden wir noch weitere Operationen durch den Addierer ausführen lassen. Dadurch wird die Beschaltung seiner Ein- und Ausgänge noch etwas aufwendiger.

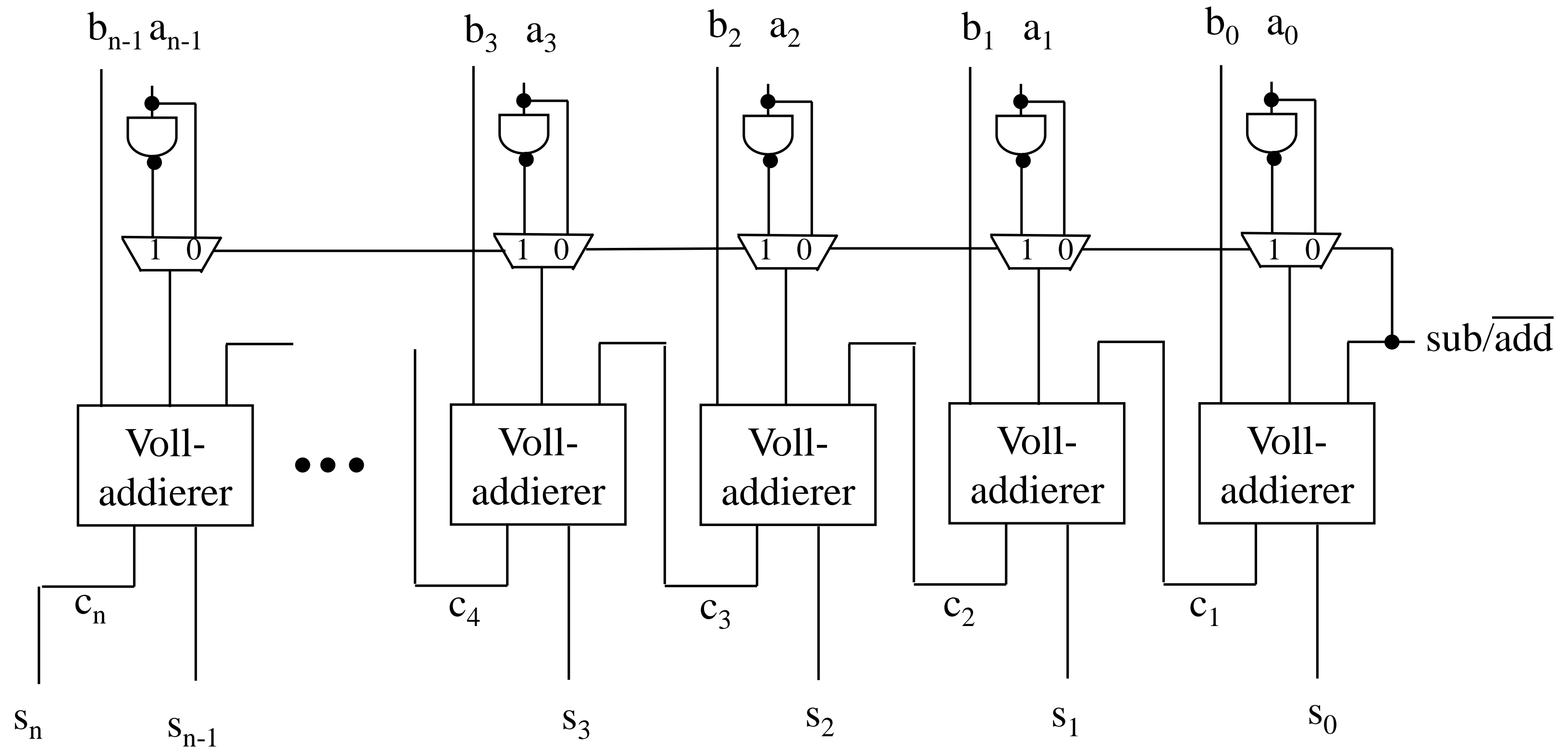
Addierer



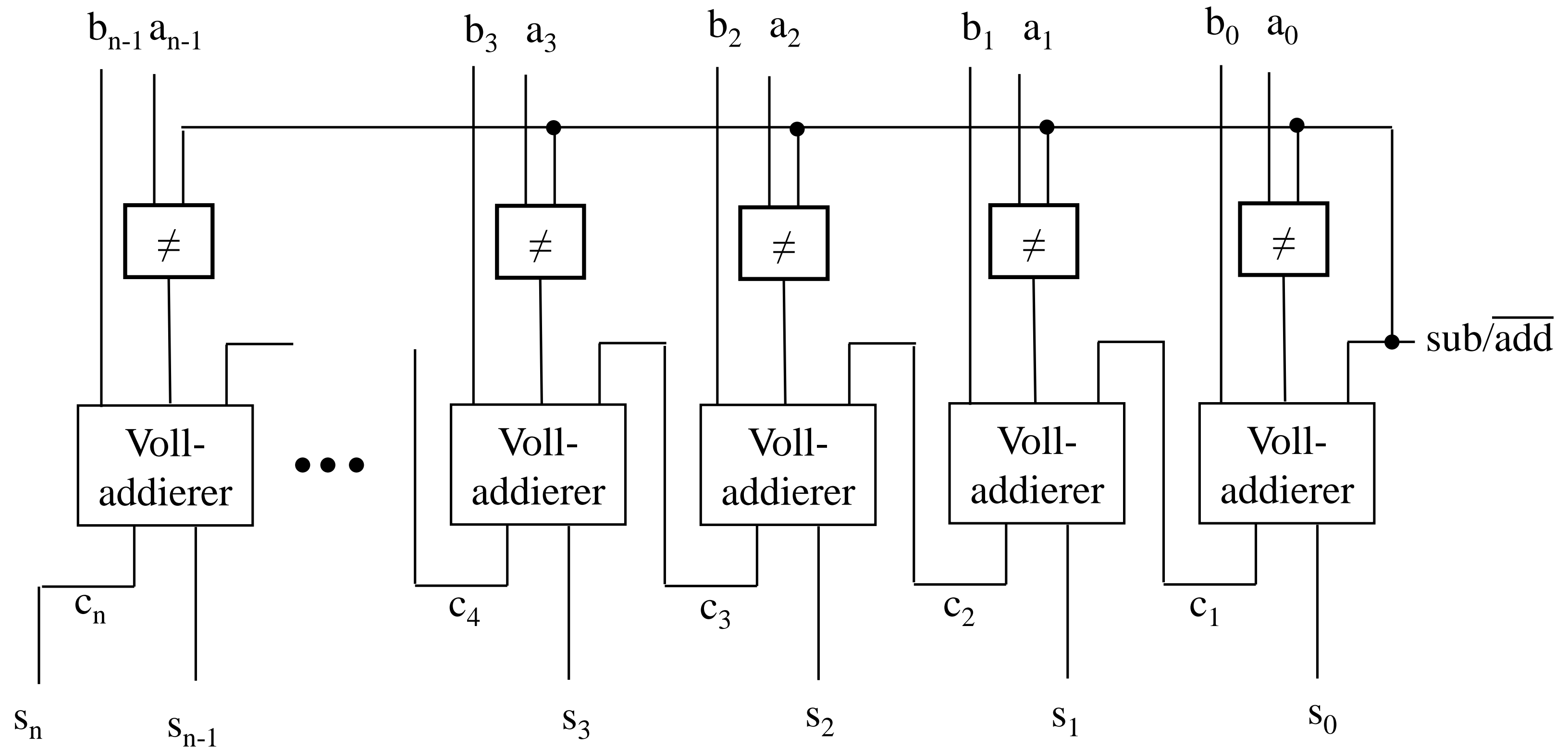
Subtrahierer



Addierer/Subtrahierer



Addierer/Subtrahierer



Befehlssatz

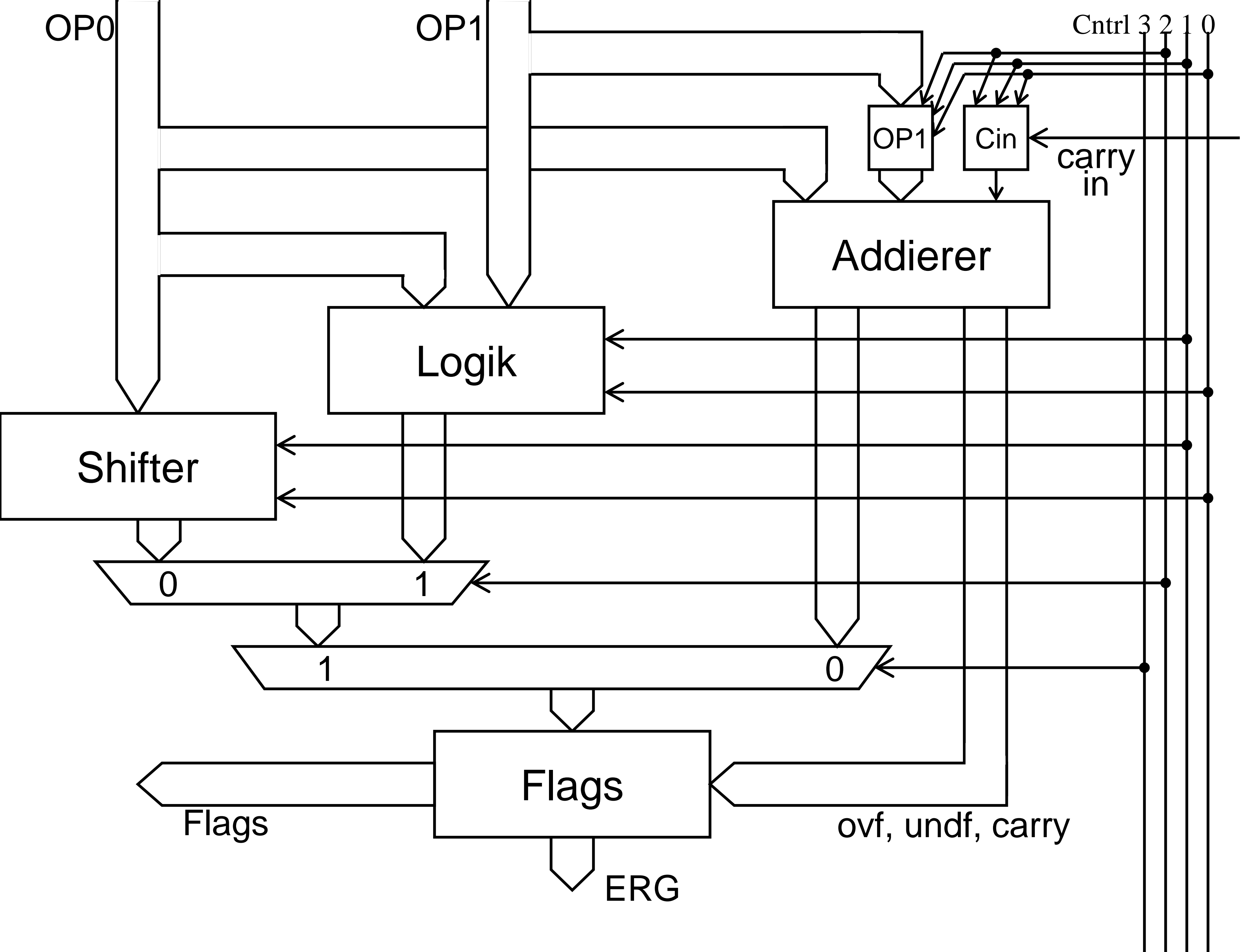
Nun müssen wir uns im Klaren darüber sein, was für einen Befehlssatz wir mit unserer ALU ausführen können wollen. Die folgende Folie zeigt eine typische Auswahl der Operationen, die auf der ALU eines modernen RISC-Prozessors ausgeführt werden können. Man beachte, daß diese Befehlsauswahl einige Redundanz 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.
Die 16 Befehle werden in vier Bits $cntrl_3,...,cntrl_0$ codiert. Dabei entscheidet $cntrl_3$, ob es eine arithmetische Operation ist oder nicht und $cntrl_2$ ob eine shift- oder logische Operation bzw. eine Addition oder Subtraktion.

Befehlssatz:

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 mit $Carry_{in}$	0	0	1	1
SET	ERG:=OP0	0	1	0	0
INC	ERG:=OP0+1	0	1	0	1
SUB	ERG:=OP0-OP1	0	1	1	0
SBC	ERG:=OP0-OP1 mit $Carry_{in}$	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

Struktur der ALU

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

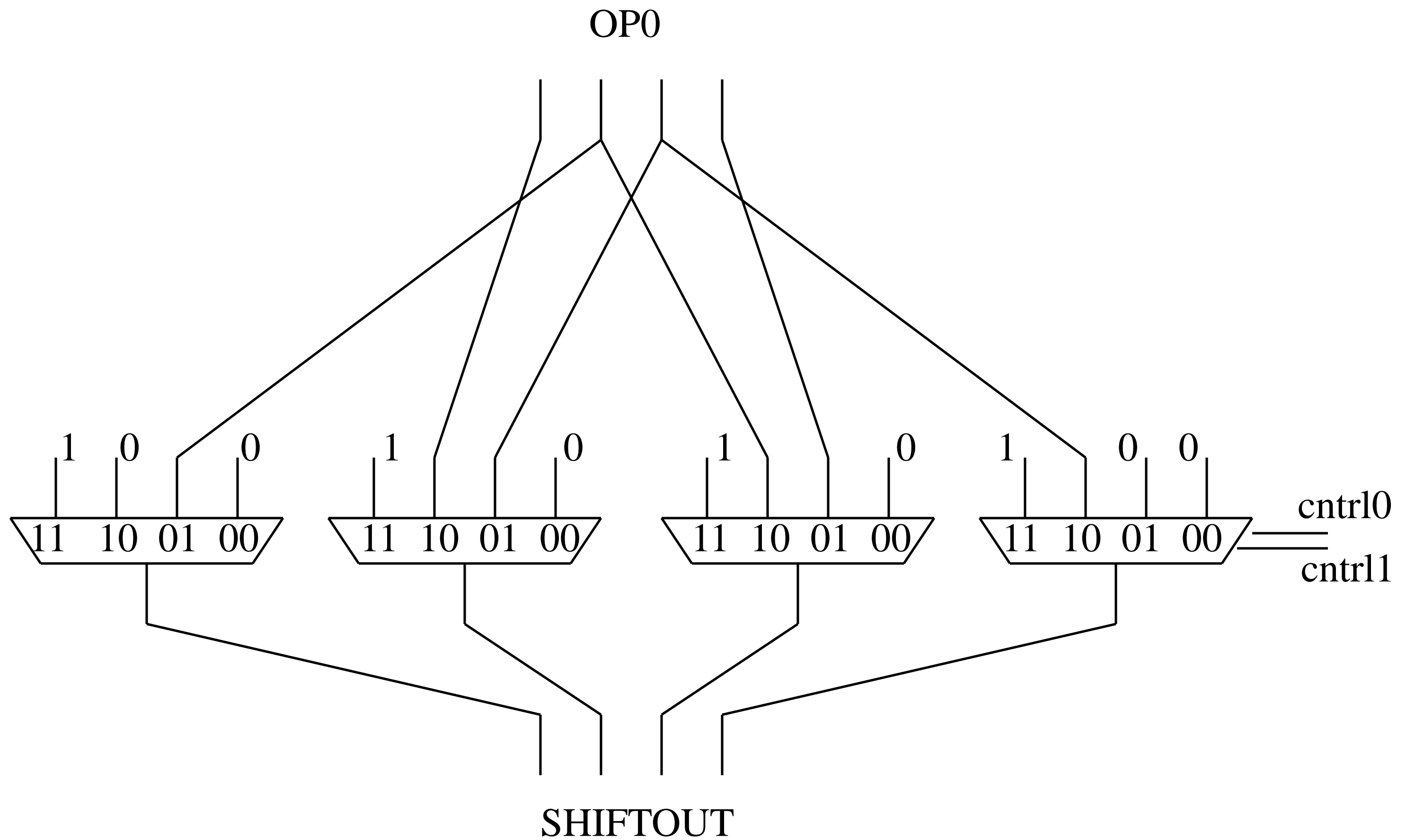


OP0
OP1
Cntrl 3 2 1 0
OP1
Cin
carry in
Addierer
Logik
Shifter
0
1
1
0
Flags
Flags
ovf, undf, carry
ERG

Der Shifter

Wir wissen bereits, was eine Leitung ist, und was ein Datenweg-Multiplexer ist. Es bleibt die Aufgabe, die Einheiten mit Leben zu füllen, von denen wir durch den Befehlssatz zunächst nur eine funktionale Beschreibung haben.
Wir fangen mit der einfachsten Einheit an, dem Shifter. Seine Aufgabe ist, die Befehle SETF (setze auf FALSE, setze auf 0), SLL (shift logical left), SLR (shift logical right) und SETT (setze auf TRUE, setze 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.
Wenn ein Ringschieben realisiert werden soll, kann man in der Einheit eine entsprechende Modifikation realisieren.
Die folgende Folie zeigt den Shifter für eine Datenbreite von 4 Bit.

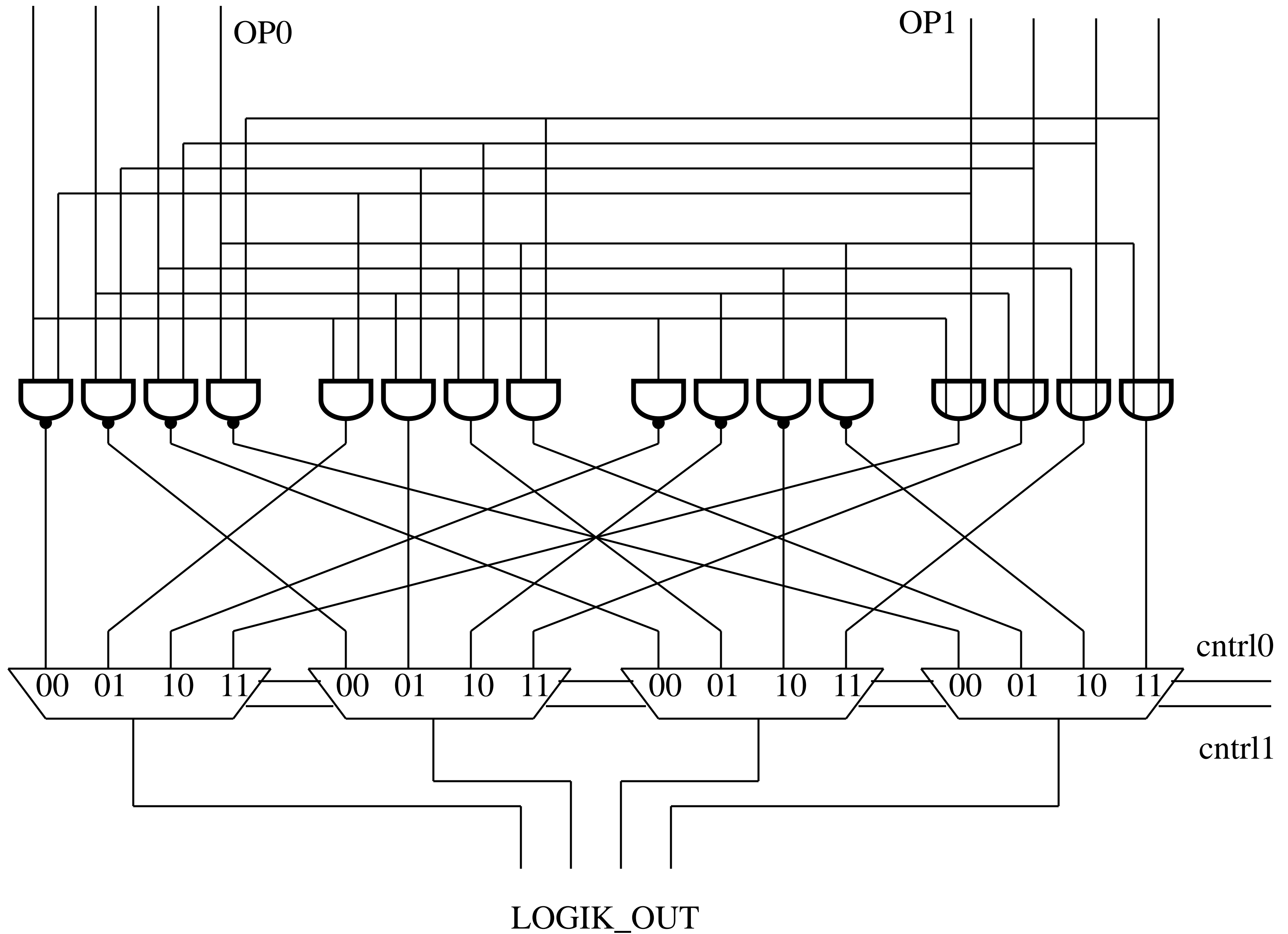
Shifter



Die Logik-Einheit

Die folgende Folie stellt eine (von vielen möglichen gleichwertige) Realisierungen der logischen Befehle dar. NAND, AND, NOT, oder OR werden gesondert bit-weise berechnet und über Multiplexer wird das durch den aktuellen Befehl geforderte Ergebnis ausgewählt.

Die Logik-Einheit



Eingänge des Addierers

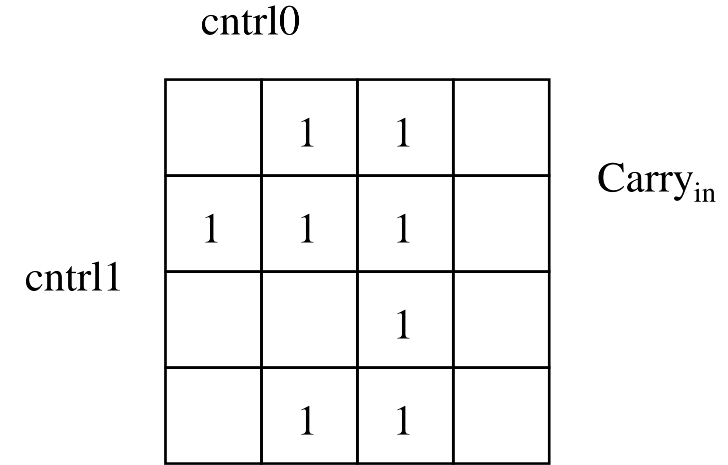
Der Addierer hat als einen Operanden immer OP0. Der zweite zu addierende Operand kann OP1, 0, -1, 1, oder -OP1 sein, je nachdem, ob addiert, subtrahiert, inkrementiert, dekrementiert oder unverändert durchgereicht werden soll. Diese Fälle werden folgendermaßen behandelt:
Bei der einfachen Addition ist der zweite Eingang gleich OP1, der Carry-Eingang gleich 0. Bei der Addition mit Berücksichtigung des alten $Carry_{in}$ ist der Eingang gleich OP1, der Carry-Eingang gleich $Carry_{in}$. 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 $Carry_{in}$ 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 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.
Die Schaltnetze für den Carry-Eingang und den zweite Eingang des Addierers werden auf den folgenden Folien abgeleitet. Dabei ist das Schaltnetz für den zweiten Eingang des Addierers für jedes Bit einzeln vorzusehen.

Eingänge des Addierers

Befehl	$Carry_{in}$	cntrl2	cntrl1	cntrl0	C	OP1-Eingang
set	0	0	0	0	0	0
dec	0	0	0	1	0	1
add	0	0	1	0	0	OP1
adc	0	0	1	1	0	OP1
set	0	1	0	0	1	1
inc	0	1	0	1	1	0
sub	0	1	1	0	1	~OP1
sbc	0	1	1	1	0	~OP1
set	1	0	0	0	0	0
dec	1	0	0	1	0	1
add	1	0	1	0	0	OP1
adc	1	0	1	1	1	OP1
set	1	1	0	0	1	1
inc	1	1	0	1	1	0
sub	1	1	1	0	1	~OP1
sbc	1	1	1	1	1	~OP1

C-Eingang des Addierers

Befehl	$Carry_{in}$	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



$$\text{C-Eingang} = \overline{\text{cntrl1}}\ \text{cntrl2} + \overline{\text{cntrl0}}\ \text{cntrl2} + \text{cntrl0}\ \text{cntrl1}\ Carry_{in}$$

Implementierung als Komplexgatter

Betrachten wir zuerst die p-Seite: Die disjunktive Minimalform liefert unmittelbar eine schnelle und einfache Realisierung der Werte, bei denen die Funktion 1 ist, indem wir die invertierten Eingänge zum Steuern von p-Transistoren verwenden, und diese in Reihe schalten für eine „und“-Verknüpfung und parallel für eine „oder“-Verknüpfung.

Für die Werte, bei der die Funktion den Wert 0 hat, ist die n-Seite verantwortlich. In diesem Fall ist die konjunktive Minimalform besser geeignet.

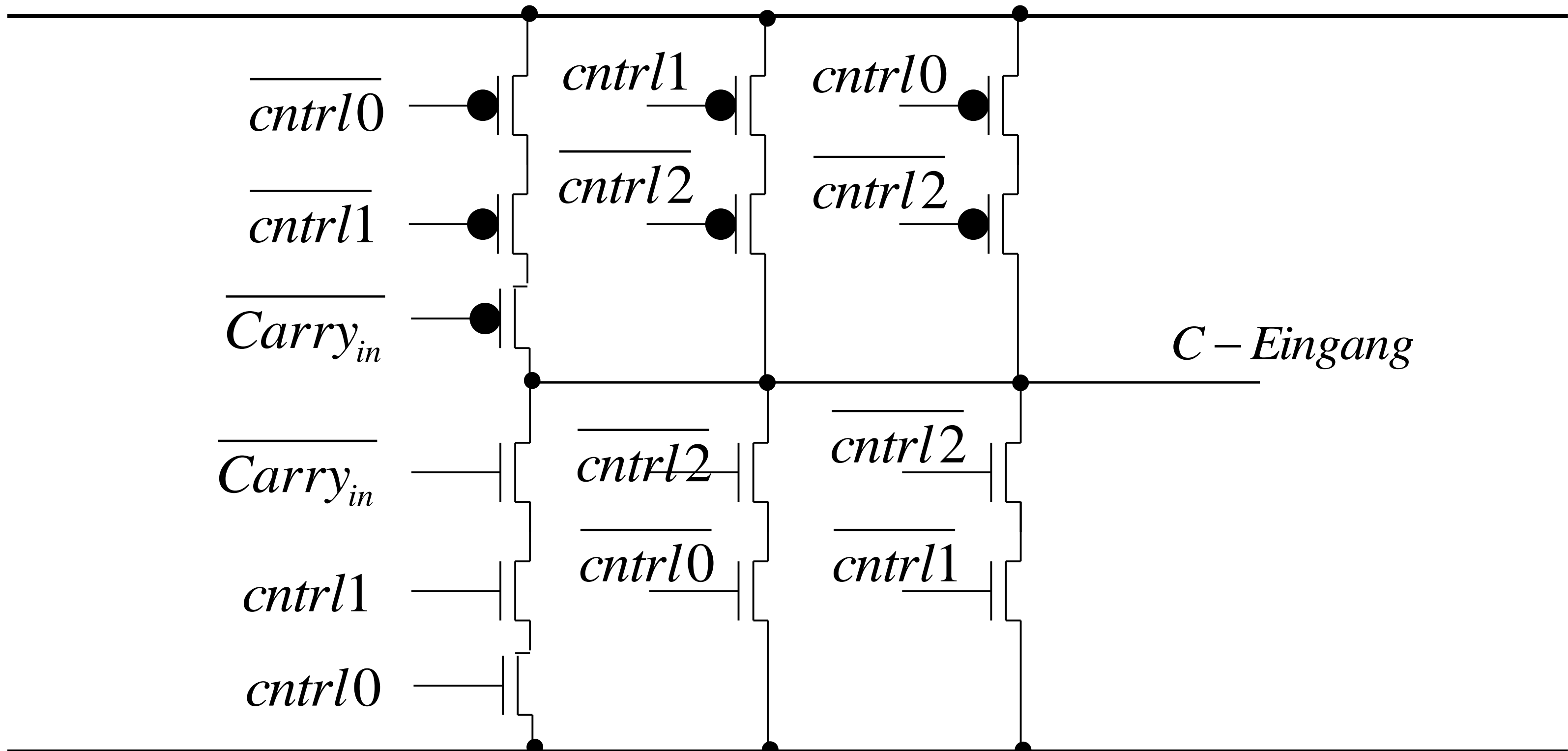
$$\overline{C - Eingang} = \overline{(cntrl0 + cntrl2) \cdot (cntrl1 + cntrl2) \cdot \left(\overline{cntrl0} + \overline{cntrl1} + Carry_{in}\right)} =$$

$$= \overline{(cntrl0 + cntrl2)} + \overline{(cntrl1 + cntrl2)} + \overline{\left(\overline{cntrl0} + \overline{cntrl1} + Carry_{in}\right)} =$$

$$= \overline{cntrl0} \cdot \overline{cntrl2} + \overline{cntrl1} \cdot \overline{cntrl2} + cntrl0 \cdot cntrl1 \cdot \overline{Carry_{in}}$$

Implementierung als Komplexgatter

Damit kommen wir zu folgendem Komplexgatter:



OP1-Eingang des Addierers

Befehl	OP1	cntrl2	cntrl1	cntrl0	OP1-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

cntrl0

	1	1	1	
	1			1
cntrl1		1	1	
	1		1	

OP1

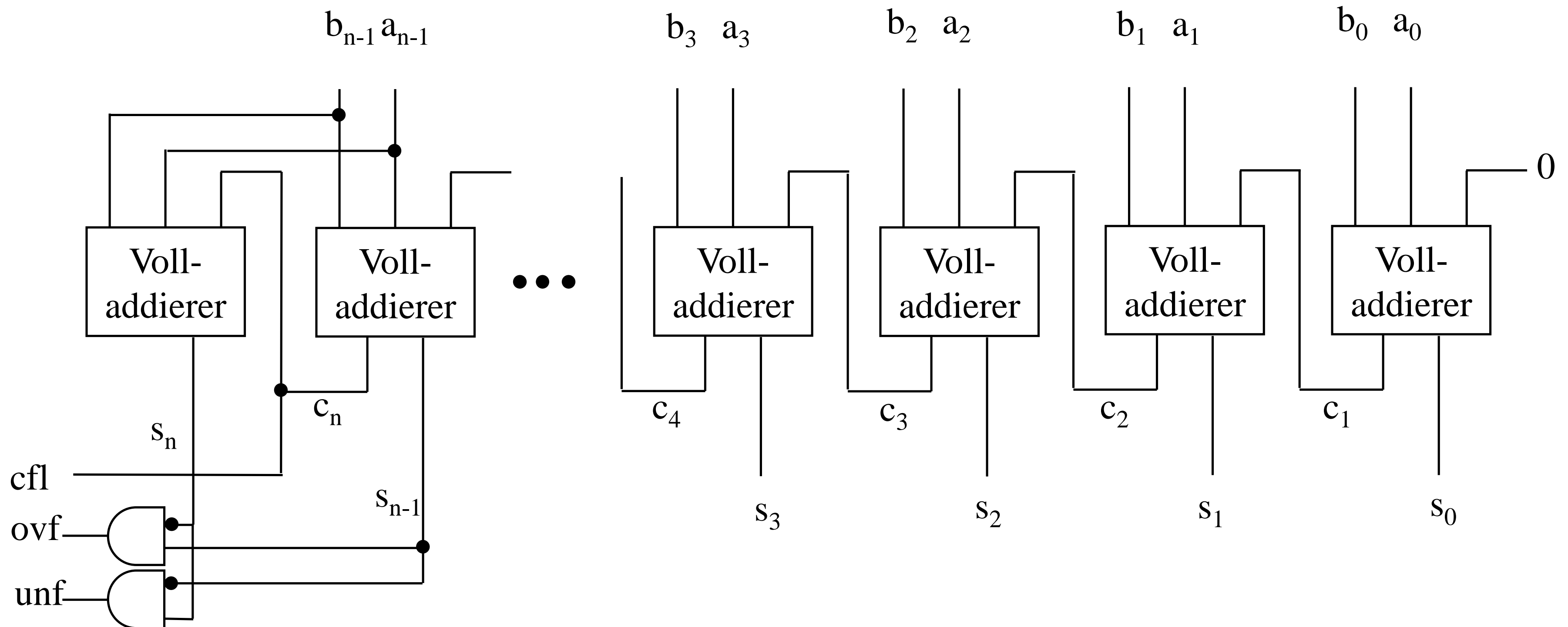
cntrl2

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

Überlauf-Erkennung beim Addierer

Wie bereits aus der ersten Vorlesung bekannt, können Über- und Unterläufe bei der Addition erkannt werden anhand eines zusätzlichen Sicherungsbits, das eine Kopie des höchst-signifikanten Bits darstellt. Man benutzt nun für eine n-Bit-Addition einen n+1-Bit Addierer, wobei die Sicherungsstelle ganz normal mitaddiert wird. Das Ergebnis der Addition ist also die Summe s_{n-1}, s_{n-2}, ..., s_1, s_0, das ausgehende Carry-Bit (= Carry-Flag) c_n, sowie ein künstliches Summenbit s_n. Das künstliche Carry-Bit c_{n+1} hat keine Bedeutung und wird daher nicht verwendet. Ein Überlauf (Ergebnis ist größer als die größte darstellbare Zahl) ist nun aufgetreten, wenn s_{n-1} gleich 0 und s_n gleich 1 ist. Wenn s_n gleich 0 und s_{n-1} gleich 1 ist, hat ein Unterlauf (Ergebnis ist kleiner als die kleinste darstellbare Zahl) stattgefunden. Wenn s_{n-1} gleich s_n ist, ist weder Überlauf noch Unterlauf aufgetreten, also die Addition ist fehlerfrei verlaufen.
Das Schaltnetz auf der folgenden Folie zeigt die Ergänzung unseres Addierers, mit der wir Über- und Unterläufe erkennen und als Flags (ovf (overflow) und unf (underflow)) an die Flag-Einheit übermitteln können.

Signale für Flags



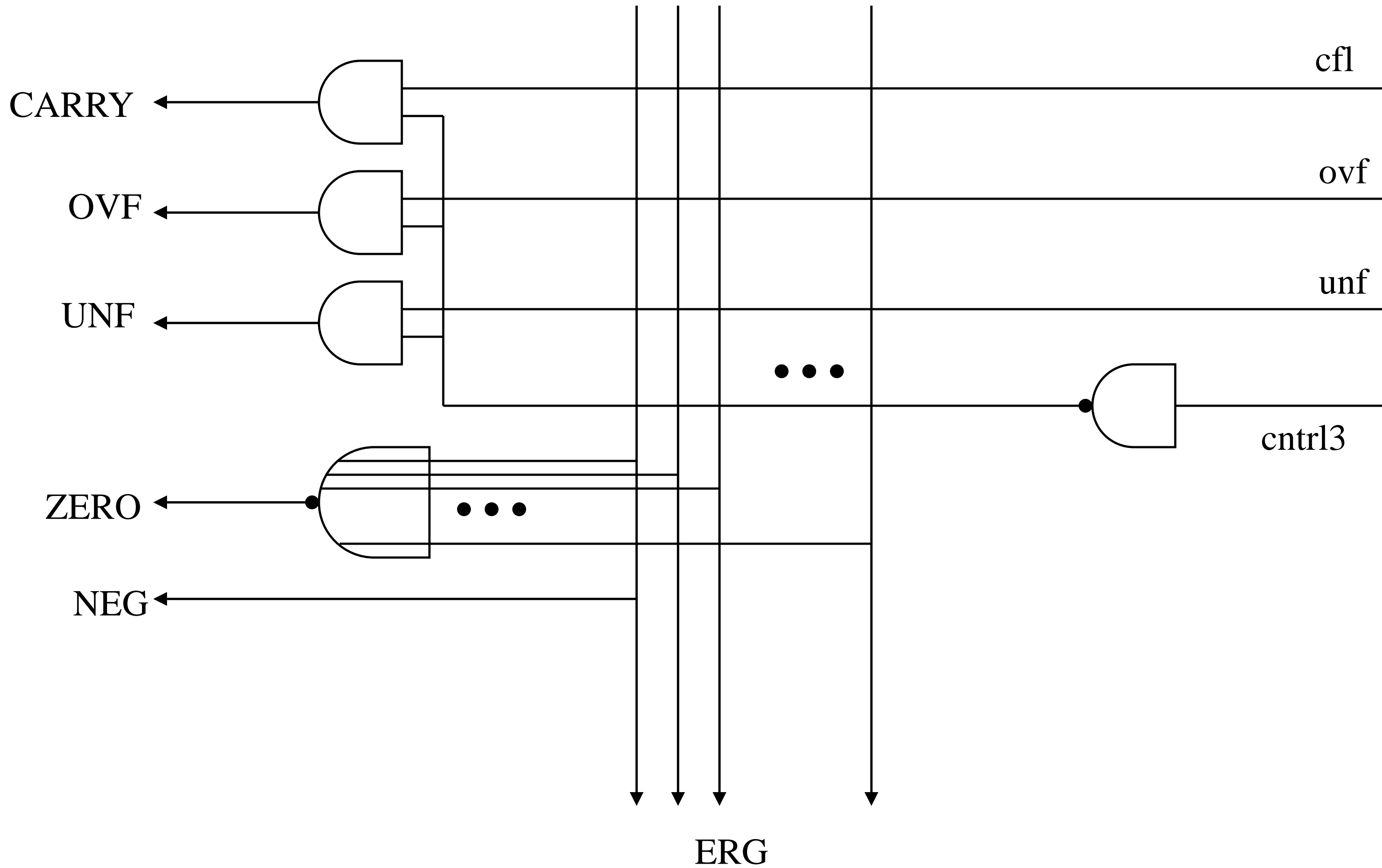
Flag-Test

Fünf Flags sollen 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)

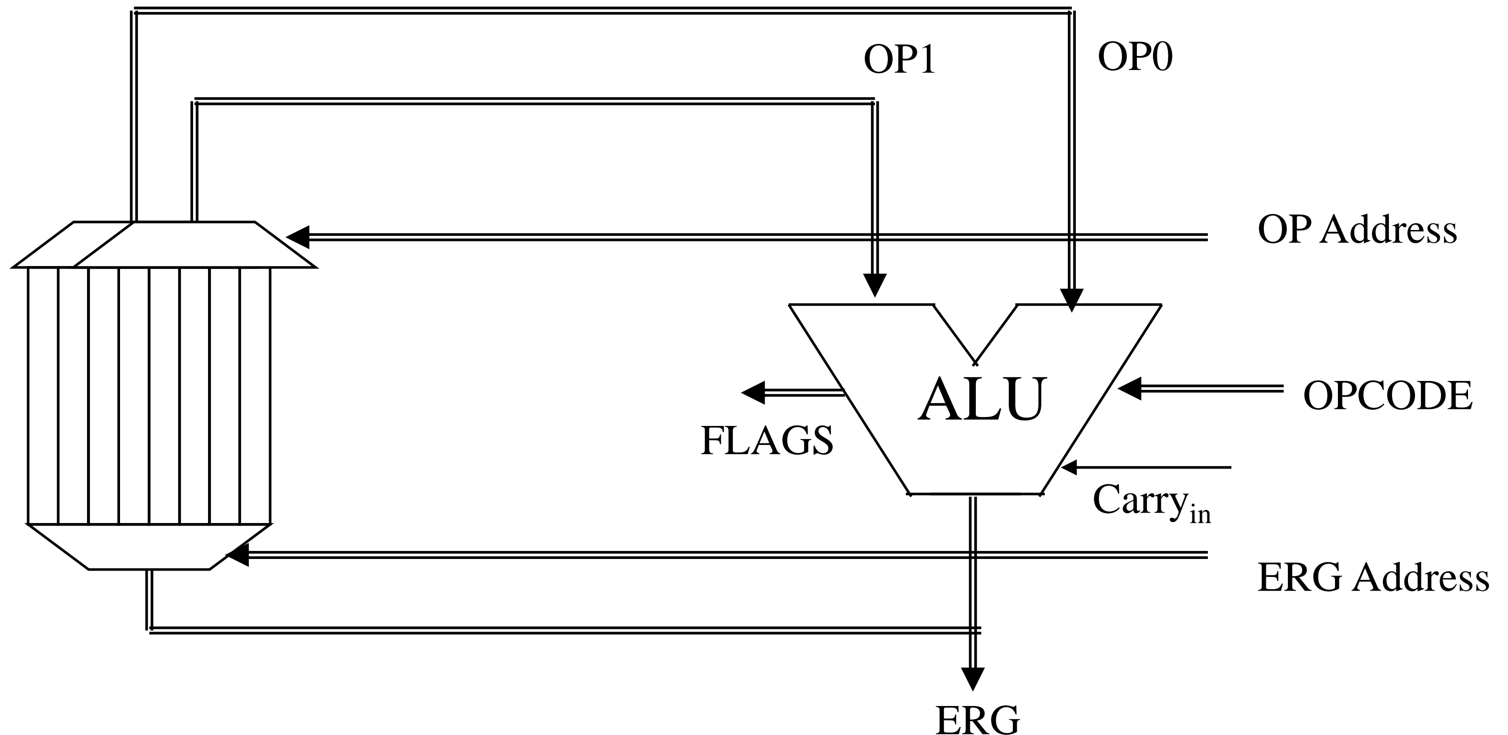
Im Einzelfall kann die Anforderungen an Flag-Einheiten sehr viel komplizierter sein, insbesondere, wenn es sich um einen Prozessor mit impliziter Condition-Behandlung handelt. Für unsere Zwecke genügt aber eine so einfache Einheit, wie sie auf der folgenden Folie dargestellt ist.

Flag-Test



Hinzufügen von Registern

Wenn wir unserer ALU jetzt noch einen Registersatz hinzufügen, so haben wir schon das Kernstück eines (sehr einfachen) Prozessors gebaut:



Beispiel: Zwei positve Zahlen multiplizieren

Diesen „Prozessor“ wollen wir jetzt mit einem Programm versehen, um etwas zu berechnen. Wir wollen das Produkt aus zwei positiven Zahlen berechnen, die am Anfang in den Registern R0 und R1 stehen sollen. Das Ergebnis soll am Ende in den Registern R6 und R7 sein. Positive 4-Bit-Zahlen können Werte zwischen 1 und 7 annehmen. Daher ist das Ergebnis zwischen 1 und 49. Somit genügt nicht ein Register zu seiner Darstellung. R6 soll am Ende den höherwertigen Teil und R7 den niederwertigen Teil des Ergebnisses enthalten.

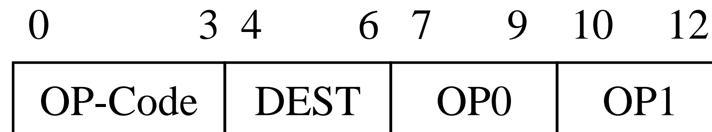
Wir benötigen einige Hilfsregister:
R2: Enthält die 1. Wird benötigt, um zu prüfen, ob das LSB von R0 gleich 1 ist.
R3: Enhält die 0 oder genau eine 1 an unterschiedlichen Bitpositionen. Wird benötigt, um die Bitfolge 0000 oder 1111 in R4 zu erzeugen.
R4: Wird als Maskenregister benötigt, um die 4-Bit-Multiplikation mit R1 als logische AND-Opreration durchzuführen. Außerdem fungiert es als Hilfsregister bei der doppeltlangen Addition.
R5: Enthält das partielle Produkt für das jeweilig zu bearbeitende Bit von R0. Wird genutzt, um dieses partielle Produkt zu der bisherigen Summe in R6 und R7 zu addieren.
R6: Höherwertiges Ergebnisregister
R7: Niederwertiges Ergebnisregister.

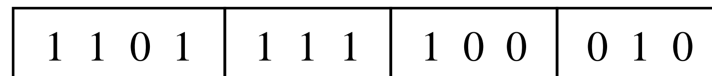
```
SETF    R6            Initialisieren von R6 mit 0
INC     R2,R6         R2 ist jetzt 1

AND     R3,R0,R2      Ist das letzte Bit von R0 = 1 ?
SUB     R4,R6,R3      R4 ist jetzt 1111 oder 0000, abhängig von R3

AND     R7,R1,R4      Multiplikation von R1 mit dem letzten Bit von R0
SRL     R0,R0

SETF    R4            R4 zurück auf 0 setzen
AND     R3,R0,R2      Ist das letzte Bit von R0 = 1 ?
SUB     R4,R4,R3      R4 ist jetzt 1111 oder 0000, abhängig von R3
```

```
AND     R5,R1,R4     Multiplizieren des zweiten Bits von R0 mit R1
SETF    R4           Initialisieren von R4 mit 0
ADD     R5,R5,R5     Verschieben von R5 um ein Bit nach links (mit Carry)
ADC     R4,R4,R4     Auffangen eines eventuell entstehenden Carrys
ADD     R7,R5,R7     Hinzufügen zur bisherigen Summe (niederwertiger Teil)
ADC     R6,R4,R6     Hinzufügen zur bisherigen Summe (höherwertiger Teil)
SRL     R0

SETF    R4           Initialisieren von R4 mit 0000
AND     R3,R0,R2     Ist das letzte Bit von R0 = 1 ?
SUB     R4,R4,R3     R4 ist jetzt 1111 oder 0000, abhängig von R3

AND     R5,R1,R4     Multiplizieren des dritten Bits von R0 mit R1
SETF    R4           Laden von R4 mit 0
ADD     R5,R5,R5     Verschieben von R5 um ein Bit nach links (mit Carry)
ADC     R4,R4,R4     Auffangen eines eventuell entstehenden Carrys
ADD     R5,R5,R5     Verschieben von R5 um noch ein Bit nach links (mit Carry)
ADC     R4,R4,R4     Auffangen eines eventuell entstehenden Carrys
ADD     R7,R5,R7     Hinzufügen zur bisherigen Summe (niederwertiger Teil)
ADC     R6,R4,R6     Hinzufügen zur bisherigen Summe (höherwertiger Teil)
```

Befehlsformat

Wie müssen die Befehle für einen solchen Prozessor aufgebaut sein?

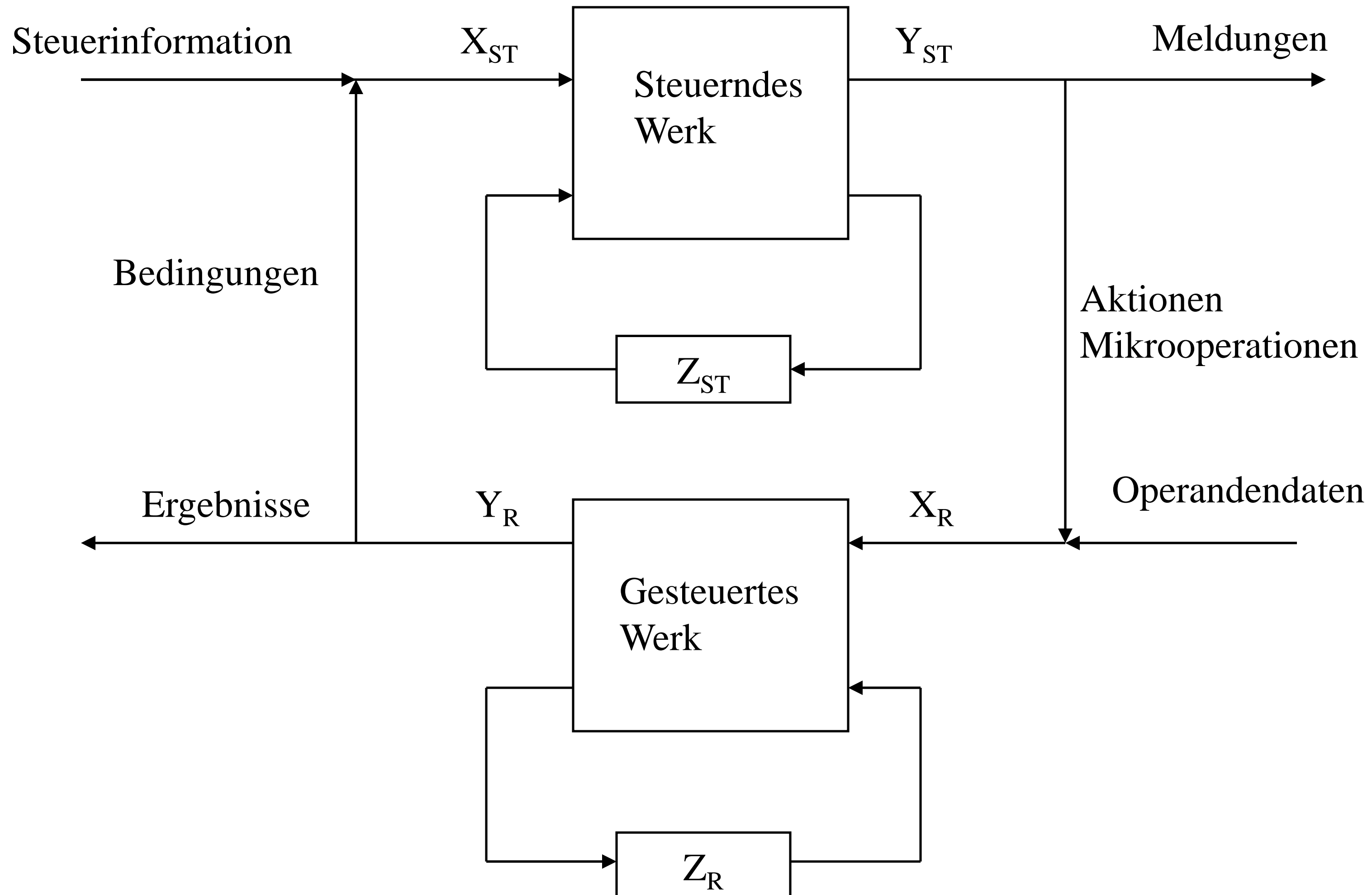




Steuerwerk

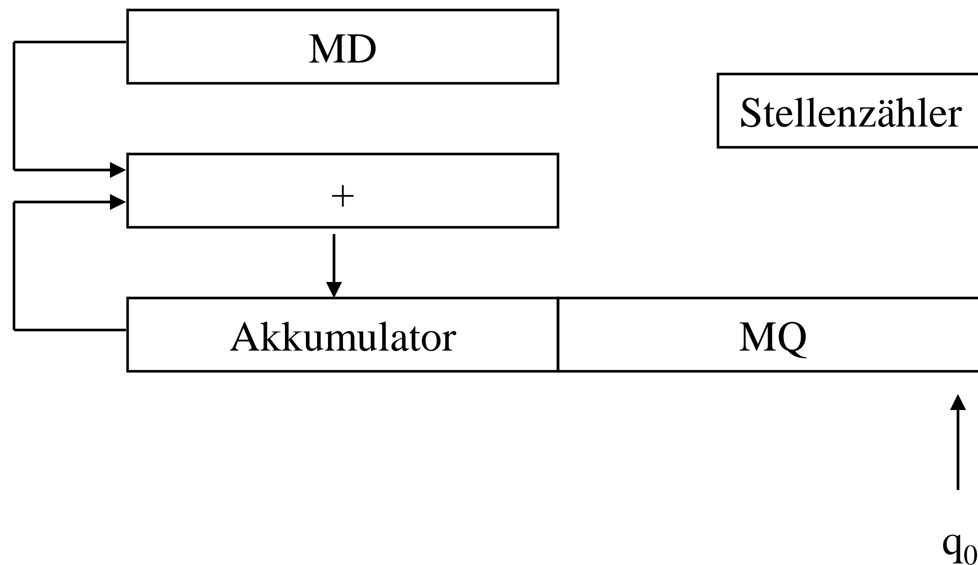
Wir kennen jetzt den Aufbau von Rechenwerken, z.B. einem Addierer oder einer logischen Einheit. In einem Computer müssen diese Einheiten aber zur richtigen Zeit aktiviert werden, Daten müssen über Multiplexer an die richtigen Einheiten geführt werden, mit anderen Worten: Die Abläufe müssen gesteuert werden. Diese Funktionen übernehmen ebenfalls Schaltwerke, sogenannte **Steuerwerke**. Wir wollen in diesem Abschnitt an einem einfachen Beispiel die Funktion eines Steuerwerks studieren.
Die folgende Folie zeigt das grundsätzliche Prinzip eines Steuerwerks: Es gibt ein steuerndes (Schalt-)werk und ein gesteuertes Werk. Das gesteuerte Werk ist häufig ein Rechenwerk. Das Steuerwerk hat - wie jedes Schaltwerk - Eingaben, Ausgaben und Zustände. Die Eingaben sind einerseits die Befehle der übergeordneten Instanz (z.B. des Benutzers oder eines übergeordneten Steuerwerks), andererseits die **Bedingungen.** Dies sind Ausgaben des Rechenwerks, mit dem es dem Steuerwerk Informationen über den Ablauf der gesteuerten Aufgabe gibt. Die Ausgaben des steuernden Werks sind einerseits die Meldungen zur übergeordneten Instanz und andererseits die **Mikrobefehle (Aktionen)**, die als Eingaben in das gesteuerte Werk gehen, und dadurch die Abläufe dort kontrollieren.

Steuerwerksprinzip



Multiplizierwerk

Das folgende Beispiel soll das Steuerwerksprinzip illustrieren:
Ein Multiplizierwerk ist zu steuern, das zwei Zahlen (z.B. mit jeweils vier Bit) multiplizieren kann. Das Multiplizierwerk besteht aus einem Multiplikanden-Register MD, einem Akkumulatorregister AKK, das als paralleles Register und als Schieberegister genutzt werden kann, einem Multiplikatorregister MQ mit derselben Eigenschaft, einem Zähler, genannt Stellenzähler (SZ) und einem Addierer. Mit q_0 wird die letzte Stelle von MQ bezeichnet.



Multiplizierwerk

Für eine Multiplikation werden Multiplikand und Multiplikator in die ensprechenden Register geladen, der Akkumulator wird mir 0 vorbesetzt. Sodann sendet die Steuerung einen Mikrobefehl „-n“, der den Stellenzähler auf -n setzt. n soll dabei die Anzahl der Stellen der zu multiplizierenden Operanden sein. Wenn q_0 1 ist, wird jetzt die Mikrooperation „+“ generiert, die das Addieren des gegenwärtigen Akkumulators mir dem Register MD bewirkt, das Ergebnis wird im Akkumulator gespeichert. Danach werden die Mikrooperationen „S“ und „SZ“ generiert. S bewirkt ein Verschieben um ein Bit nach rechts im Schieberegister bestehend aus Akkumulator und MQ. SZ bewirkt ein Inkrementieren des Stellenzählers. Wenn der Stellenzähler den Wert 0 erreicht hat, terminiert der Prozeß, das Ergebnis steht im Schieberegister. Wenn der Stellenzähler nicht 0 ist, wird wiederum q_0 interpretiert. Wenn q_0 0 ist, wird die Mikrooperation „0“ generiert, die kein Register verändert.

Die folgende Seite zeigt ein Beispiel für die Multiplikation der Zahlen 0101 und 1011 (5 * 11 = 55).

Abfolge im Multiplizierwerk

MD = 0101

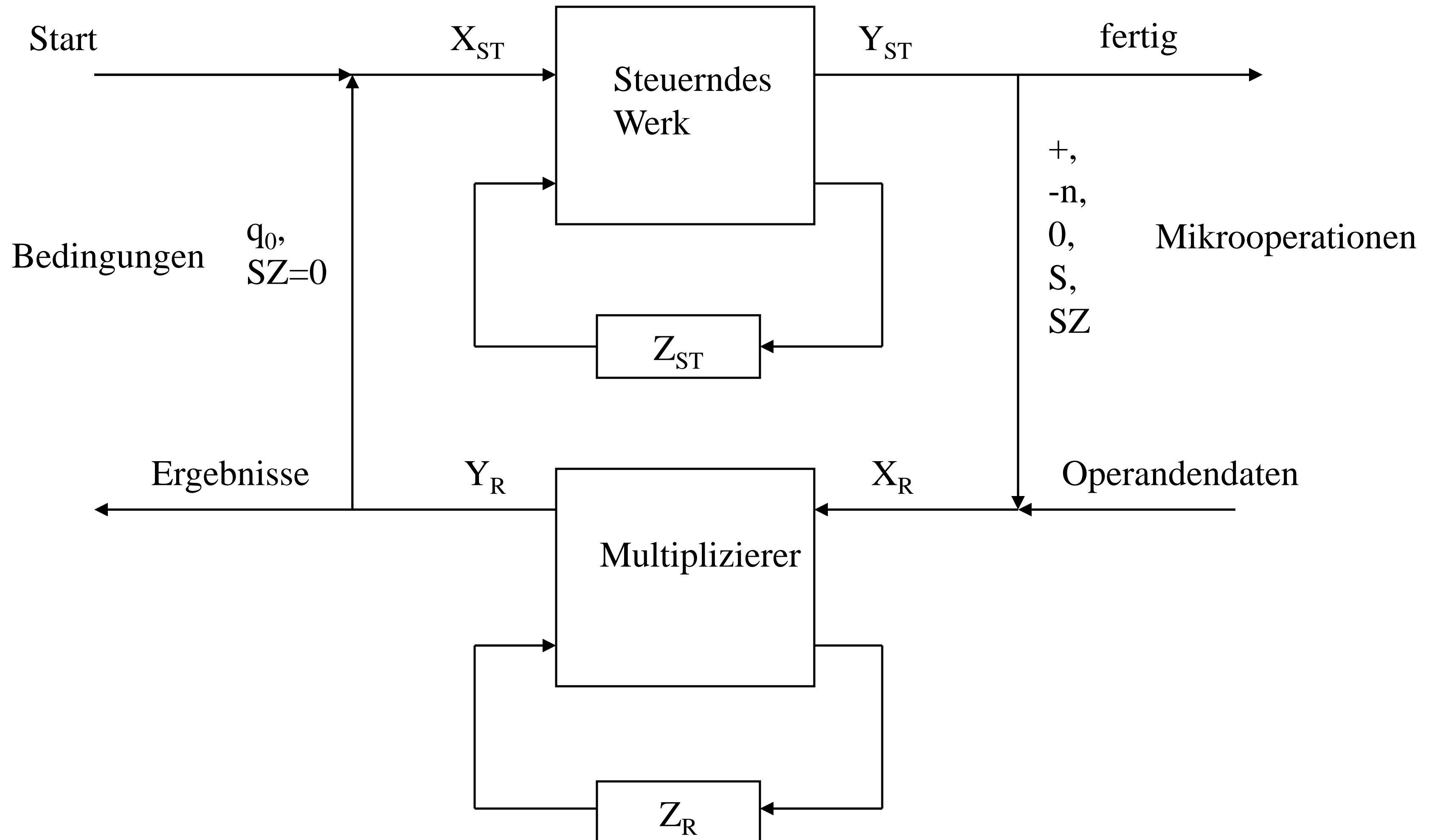
AKKU	MQ	SZ	q0	SZ=0	Start	Mikrooperation
0 0 0 0	1 0 1 1	000	1	1	1	-n
0 0 0 0	1 0 1 1	100	1	0	0	+
0 1 0 1	1 0 1 1	100	1	0	0	S, SZ
0 0 1 0	1 1 0 1	101	1	0	0	+
0 1 1 1	1 1 0 1	101	1	0	0	S, SZ
0 0 1 1	1 1 1 0	110	0	0	0	0
0 0 1 1	1 1 1 0	110	0	0	0	S, SZ
0 0 0 1	1 1 1 1	111	1	0	0	+
0 1 1 0	1 1 1 1	111	1	0	0	S, SZ
0 0 1 1	0 1 1 1	000	1	1	0	

Wir wissen bereits, wie wir ein solches Rechenwerk bauen müßten, denn es besteht nur aus uns bekannten Komponenten (Register, Schieberegister, Zähler, Addierer). An dieser Stelle interessiert uns nun aber, wie wir die Mikrooperationen zum richtigen Zeitpunkt generieren können, also wie wir dieses Rechenwerk „steuern“ können. Dazu machen wir uns klar:

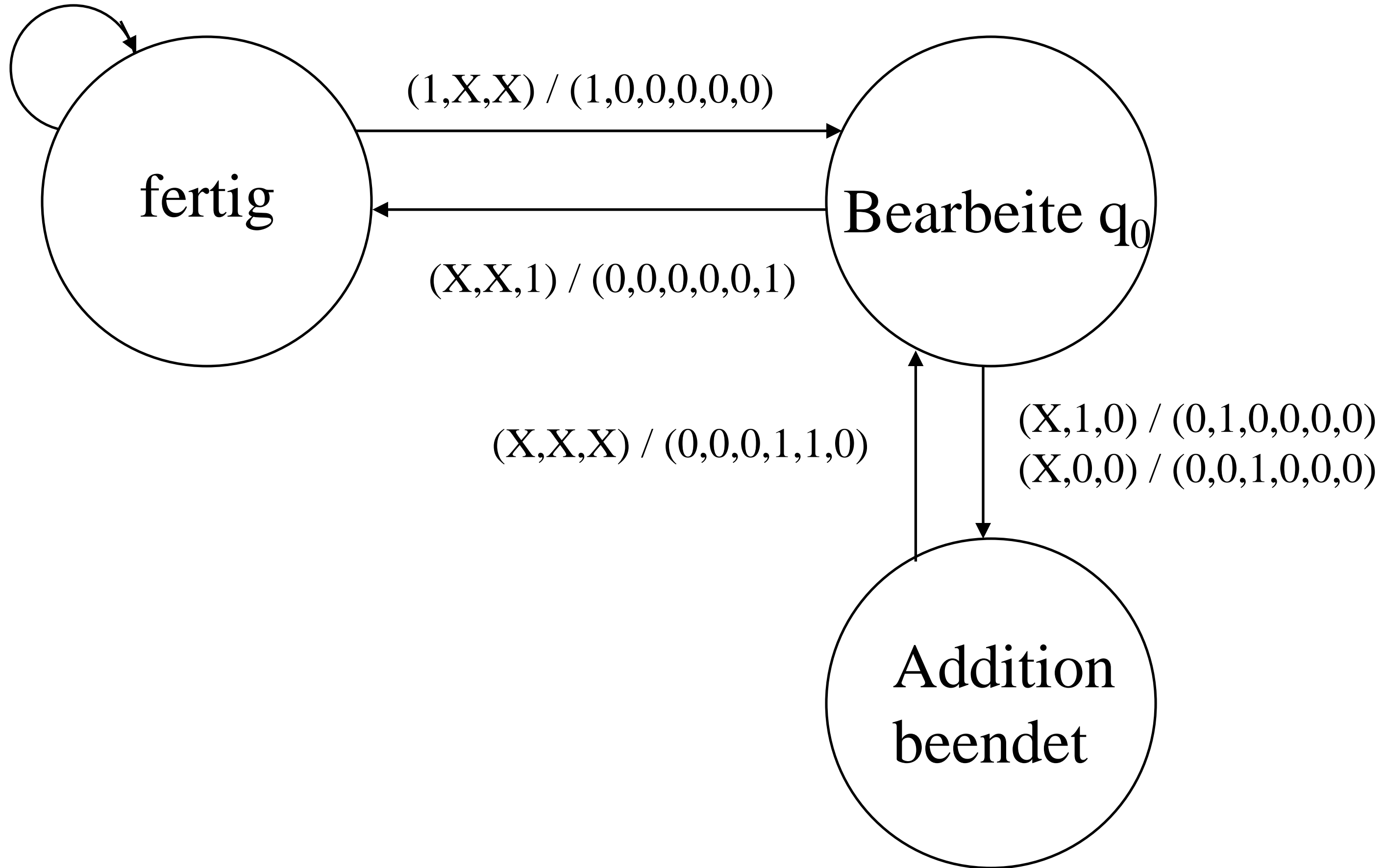
- Wenn ein „Start“-Signal kommt, muß der Stellenzähler mit „-n“ initialisiert werden.
- Wenn q_0 interpretiert wird, muß MD genau dann auf den Akkumulator addiert werden, wenn $q_0 = 1$ ist.
- Nach jedem solchen Additionsschritt muß der Akkumulator und das MQ um ein Bit geschoben und der Stellenzähler um eins erhöht (inkrementiert) werden.
- Wenn irgendwann der Stellenzähler den Wert 0 erreicht, ist die Multiplikation beendet, das Ergebnis steht im Schieberegister aus Akkumulator und MQ.

Die folgende Folie zeigt, welche dieser Signale Ein- und Ausgaben welcher Werke sind. Aus diesen Informationen können wir danach einen Automatengraphen entwickeln.

Steuerwerk



Eingaben: (Start, q_0, SZ=0) Ausgaben (-n, +, 0, S, SZ, fertig)



Wir codieren die Zustände mit

00: fertig
01: Bearbeite q_0
10: Addition beendet
11: kommt nicht vor: dont care, wobei sicherzustellen ist, dass man von dort in einen definierten Zustand gerät.

Dann entspricht dieser Automat der folgenden Wertetabelle

Start	q0	SZ=0	z1	z0	z'1	z'0	-n	+	0	S	SZ	fertig
0	X	X	0	0	0	0	0	0	0	0	0	1
1	X	X	0	0	0	1	1	0	0	0	0	0
X	1	0	0	1	1	0	0	1	0	0	0	0
X	0	0	0	1	1	0	0	0	1	0	0	0
X	X	X	1	X	0	1	0	0	0	1	1	0
X	X	1	0	1	0	0	0	0	0	0	0	1

Das ergibt nach Minimierung die Schaltwerksrealisierung auf der nächsten Folie:

Realisierung als FPLA: