Arithmetic & Logical Unit (ALU) und Steuerwerk

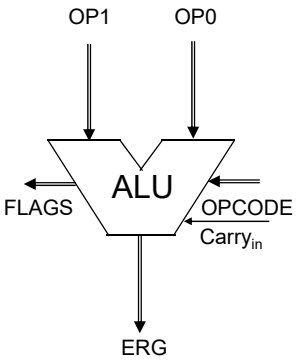
Essentieller Teil eines Prozessors ist eine ALU

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

- Addierer
- Logischer Einheit
- Shifter (Bit-Schieber)

Eingänge in eine ALU:
- Operanden OP1, OP2, $Carry_{in}$
- Instruktionscode (OPCODE)

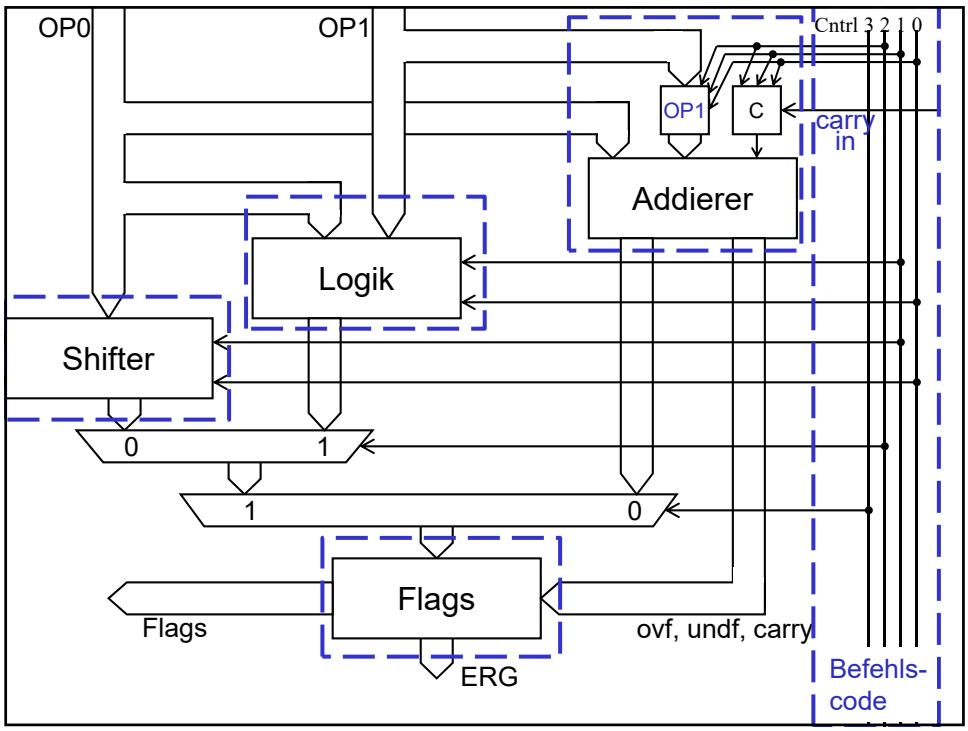
Ausgänge einer ALU:
- Ergebnis ERG, Signale (FLAGS)



Aufbau und Struktur der ALU

Die nächste Folie zeigt den prinzipiellen Aufbau der ALU. Die ALU erlaubt die Berechnung von arithmetisch-logischen Operationen auf Operatoren.
Die Steuereingänge (Control crtl) wirken einerseits auf die Einheiten (Shifter, Logik, Addierer) selbst, andererseits wählen sie durch Steuerung zweier Datenweg-Multiplexer das Ergebnis der jeweils für die aktuelle Operation zuständigen Einheit aus, um es an den ERG-Ausgang der ALU weiterzuleiten. Die Bits von Ctrl codieren damit gleichzeitig die Befehle der ALU.

Die Eingaben OP1, OP2, das Ergebnis ERG und die Signale FLAGS liegen in Datenregistern. Die ALU selbst ist ein durch Ctrl gesteuertes Schaltnetz. Die Einheiten (Addierer, Logik, Schifter) werden im Folgenden besprochen.



1. Befehlssatz und Codierung

Die vier Control-Bits $crtl_i$ steuern die Alu und stellen den Befehlssatz dar. 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 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.) Es könnten auch andere Bitkombinationen gewählt werden. 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
Addierer: ctrl3=0	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
Shifter: ctrl3=1 ctrl2=0	SETF	ERG:=0	1	0	0	0
	SLL	ERG:=2*OP0	1	0	0	1
	SLR	ERG:=OP0 div 2	1	0	1	0
	SETT	ERG:=-1	1	0	1	1
Logik: ctrl3=1 ctrl2=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

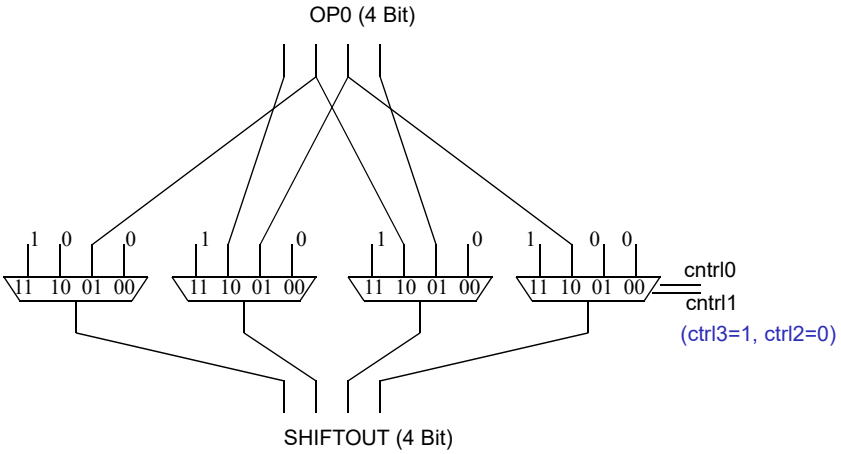
2. Der Shifter

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.

Aufbau eines 4-Bit Shifters

Befehl	Bedeutung	ctrl1	ctrl0	Operation Ergebnis
SETF	Set False = 0	0	0	(SHIFTOUT = 0000)
SETT	Set True = -1	1	1	(SHIFTOUT = 1111)
SLL	Shift logical left	0	1	(SHIFTOUT = OP0 * 2)
SLR	Shift logical right	1	0	(SHIFTOUT = OP0 div 2)



3. Die Logik-Einheit

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

AND

a_i	b_i	e_i
0	0	0
0	1	0
1	0	0
1	1	1

NAND

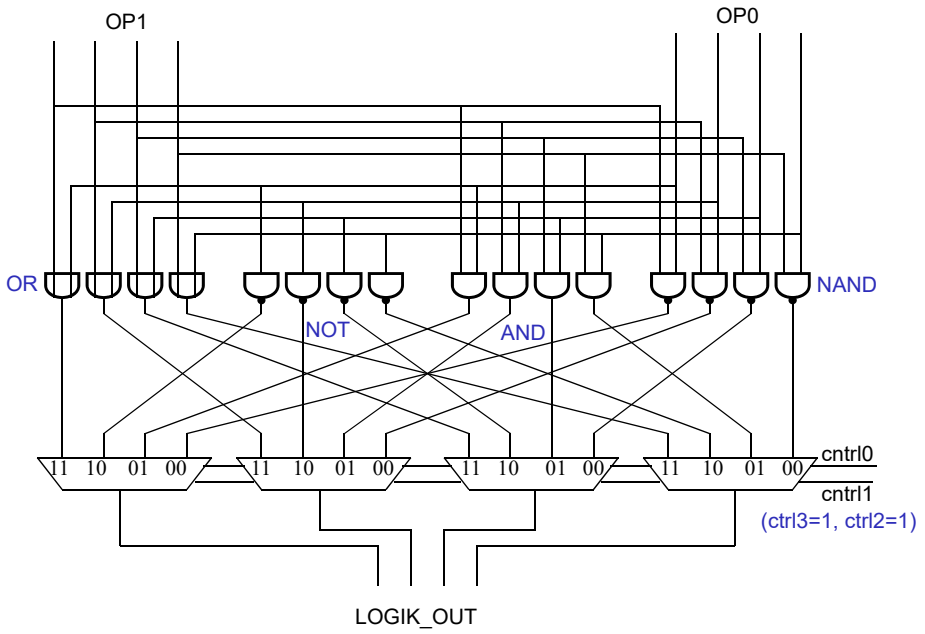
a_i	b_i	e_i
0	0	1
0	1	1
1	0	1
1	1	0

OR

a_i	b_i	e_i
0	0	0
0	1	1
1	0	1
1	1	1

Befehl	Bedeutung	ctrl1	ctrl0
NAND	ERG:=OP0 NAND OP1	0	0
AND	ERG:=OP0 AND OP1	0	1
NOT	ERG:=NOT OP0	1	0
OR	ERG:=OP0 OR OP1	1	1

Aufbau der 4-Bit Logik-Einheit



4. Der Addierer

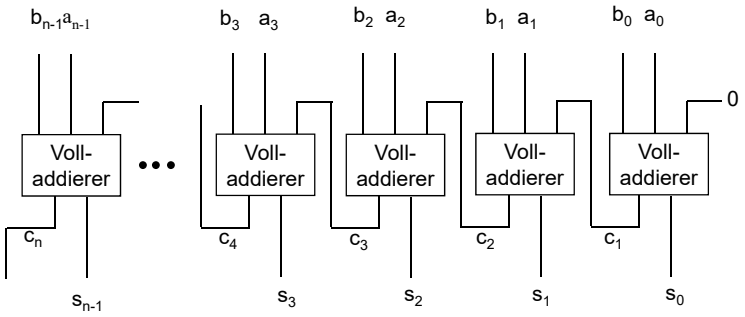
Kernstück jeder ALU ist ein Addierer. Wir sehen einen Ripple-Carry-Addierer auf der nächsten Folie (Carry-Select/Save geht natürlich auch).

Wiederholung:

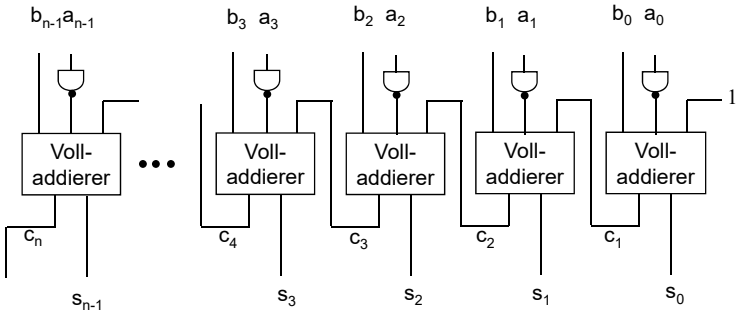
- Wie können wir Addierer zum Subtrahieren benutzen? Indem wir das Zweierkomlement des einen Operanden bilden und an den einen Eingang des Addierers führen.
- Wie bilden wir das Zweierkomplement? Indem wir jedes Bit invertieren (Einer-Komplement) und noch 1 addieren. Um für diese Addition nicht einen weiteren Addierer zu benötigen, nutzen 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 den nächsten Folien. Später werden wir noch weitere Operationen durch den Addierer ausführen lassen. Dadurch wird die Beschaltung seiner Ein- und Ausgänge noch etwas aufwendiger.

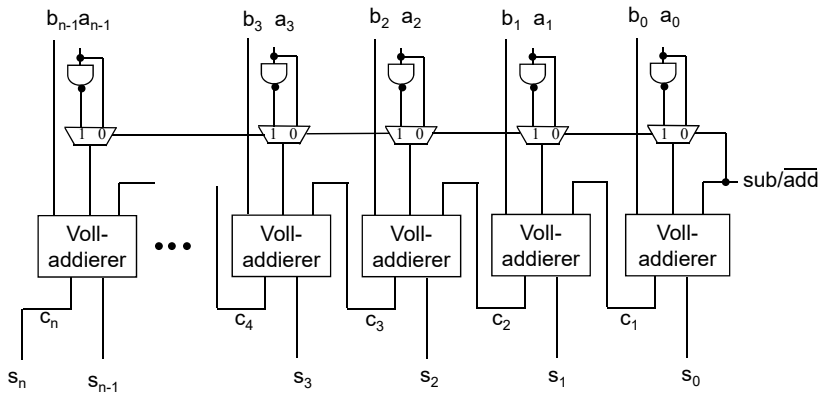
Addierer



Subtrahierer

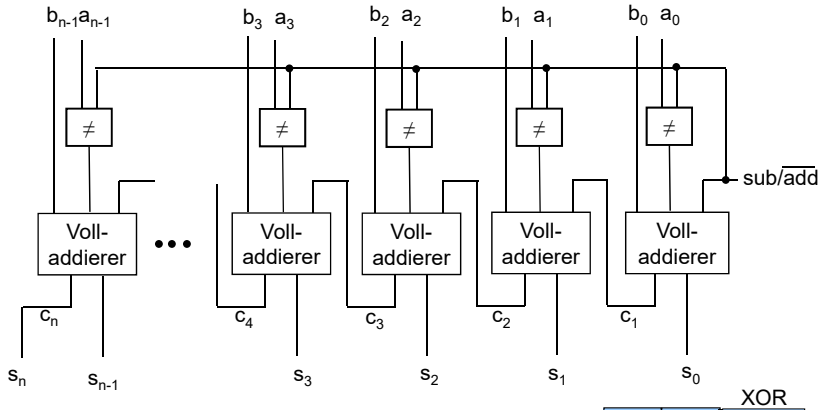


Addierer/Subtrahierer



Umschaltung von Datum und deren Invertierung durch 2-auf-1 Multiplexer

Addierer/Subtrahierer



Vereinfachung: Inverter und 2-auf-1 Multiplexer, realisiert durch XOR-Gatter

	sub	a_i	XOR $a_i \oplus$ sub
add	0	0	0
	0	1	1
sub	1	0	1
	1	1	0

Addierer und seine Eingänge

Der Addierer hat als einen Operanden immer OP0. Der zweite zu addierende Operand kann OP1, -OP1, 1, -1 oder 0 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 C-Eingang gleich 0. Bei der Addition mit Berücksichtigung des alten $Carry_{in}$ ist der Eingang gleich OP1, der C-Eingang gleich $Carry_{in}$. Bei der Subtraktion werden alle Bits von OP1 invertiert und der C-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 C-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 OP1-Eingang gleich 0 und C = 0 und beim zweiten OP1-Eingang auf -1 (alle Bits sind 1) und C = 1.

Beim Inkrementieren ist der OP1-Eingang auf 0 und der C-Eingang auf 1, beim Dekrementieren ist der OP1-Eingang auf -1 (alle Bits sind 1) und der C-Eingang ist auf 0.

Operationen und Operanden des Addierers

Die vorgenannten Operationen nutzen folgende Operanden und erzeugen folgendes Ergebnis:

Befehl	OP0	OP1	C	Ergebnis ERG	
SET	OP0	0000	0	ERG = OP0	
DEC	OP0	1111	0	ERG = OP0 + 1111 = OP0 – 1	
ADD	OP0	OP1	0	ERG = OP0 + OP1	
ADC	OP0	OP1	$Carry_{in}$	ERG = OP0 + OP1 + $Carry_{in}$	
SET	OP0	1111	1	ERG = OP0 = OP0 – 1 + 1	(OP0 + 1111 + 0001)
INC	OP0	0000	1	ERG = OP0 + 1	(OP0 + carry=1)
SUB	OP0	$\overline{OP1}$	1	ERG = OP0 + $\overline{OP1}$ + 1 = OP0 - OP1	(2-Kompl.)
SBC	OP0	$\overline{OP1}$	$Carry_{in}$	ERG = OP0 + $\overline{OP1}$ + $Carry_{in}$	(1-Kompl. + C_{in})

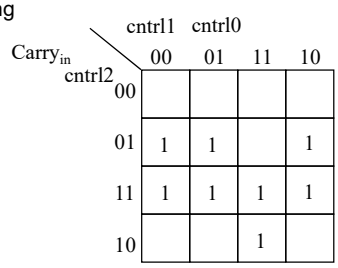
Schaltnetze für Carry und Belegung von OP1 (alle Bits)

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.

Befehl	$Carry_{in}$	cntrl2	cntrl1	cntrl0	C	OP1-Steuerung (für jedes Bit)
SET	0	0	0	0	0	0 (OP1=0000)
DEC	0	0	0	1	0	1 (OP1=1111 = -1)
ADD	0	0	1	0	0	OP1 (OP1 durchreichen)
ADC	0	0	1	1	0	OP1 (OP1 durchreichen)
SET	0	1	0	0	1	1 (Opt1=1111)
INC	0	1	0	1	1	0 (Opt1=0000)
SUB	0	1	1	0	1	$\overline{OP1}$ (OP1 invertieren)
SBC	0	1	1	1	0	$\overline{OP1}$ (OP1 invertieren)
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	$\overline{OP1}$
SBC	1	1	1	1	1	$\overline{OP1}$

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

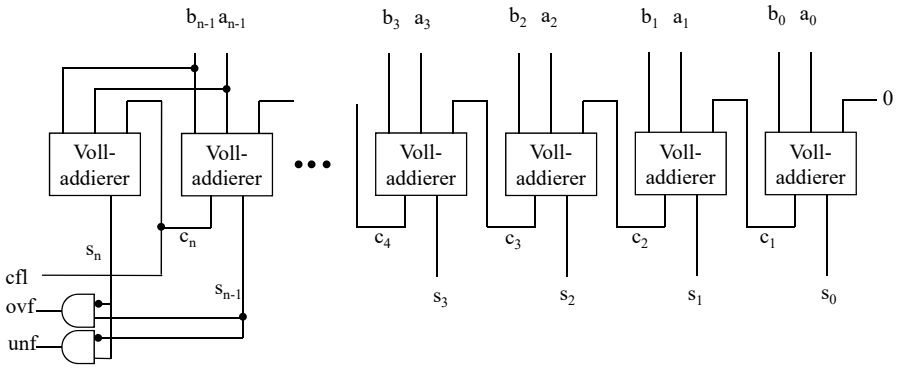


C-Eingang = $\overline{cntrl1}$ cntrl2 +
$\overline{cntrl0}$ cntrl2 +
cntrl0 cntrl1 $Carry_{in}$

5. Flags: Überlauf-Erkennung beim Addierer

Wie 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 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 1 und s_n gleich 0 ist. Wenn s_n gleich 1 und s_{n-1} gleich 0 ist, hat ein Unterlauf (Ergebnis ist kleiner als die kleinste darstellbare negative 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.

Flags als Signale: Carry, Underflow und Overflow



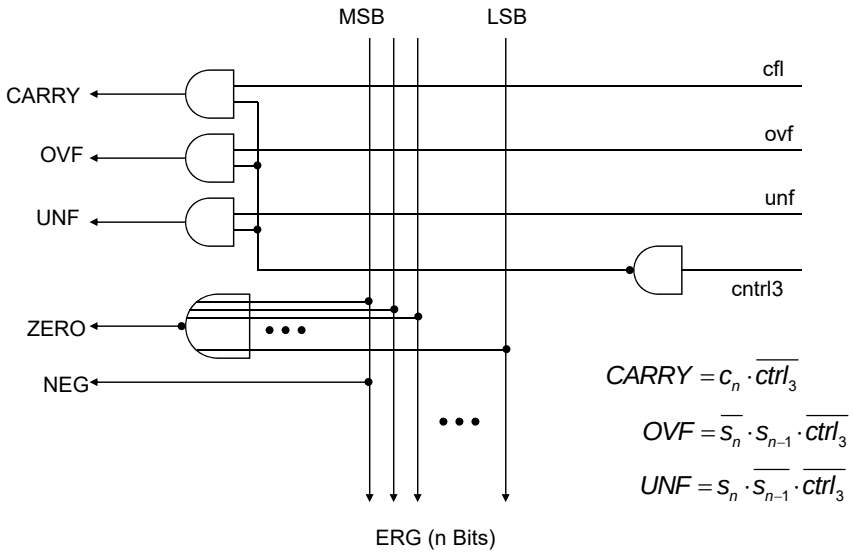
$$ovf = \overline{s_n} \cdot s_{n-1} \qquad unf = s_n \cdot \overline{s_{n-1}} \qquad cfl = c_n$$

Flag-Testschaltung

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-Testschaltung



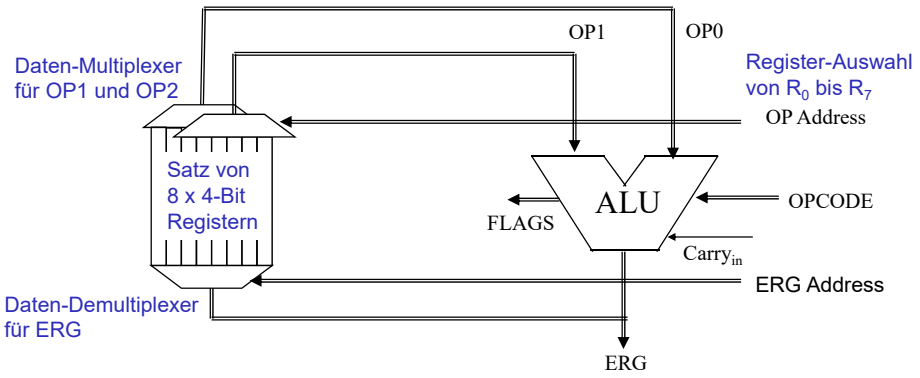
$$CARRY = c_n \cdot \overline{ctrl_3}$$

$$OVF = \overline{s_n} \cdot s_{n-1} \cdot \overline{ctrl_3}$$

$$UNF = s_n \cdot \overline{s_{n-1}} \cdot \overline{ctrl_3}$$

Hinzufügen von Daten-Registern

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



Adress: Auswahl eines der 7 Register (3 Bit Adresse)

Befehlsformat

Wie müssen die Befehle für einen solchen Prozessor aufgebaut sein?

Bits:	0 1 2 3	4 5 6	7 8 9	10 11 12
	OP-Code	ERG	OP0	OP1

AND	R7	R4	R2
1 1 0 1	1 1 1	1 0 0	0 1 0

Ein Befehl besteht aus dem OP-Code (cntrl, hier 4 Bit, 16 Befehle), gefolgt von den Registeradressen Ri (hier 3 Bit, i= {0, 1, … ,7}).

- Es gibt Befehle mit 1 Register (z.B. SETF Ri), 2 Register (z.B. INC Ri, Rj), und 3 Registern (z.B. ADD Ri, Rj, Rk).
- Das 1. Register ist immer Ergebnisregister ERG (auch Destination DEST oder Zielregister genannt) und wird vom Befehl beschrieben.
- Die anderen Register sind Operandenregister OP1, OP2, auch Quellregister genannt, und werden vom Befehl gelesen.

Beispiel: Zwei positive Zahlen multiplizieren

Diesen „Prozessor“ wollen wir jetzt mit einem Programm versehen, um etwas zu berechnen.

Wir wollen das Produkt aus zwei positiven 4-Bit-Zahlen (MSB=0) 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.

Beispiel: Zwei positive Zahlen multiplizieren

Wir benötigen einige Hilfsregister:

R2: Enthält die Zahl 1. Sie 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-Operation 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.

```
Ziel: R6 R7 = R1 * R0, werte Bits von R0 aus, multipliziere bitweise mit R1
Initialisierung
1       SETF    R6              Initialisieren von R6 mit 0000
2       INC     R2,R6           R2 ist jetzt 0001
Teste Bit 0 von R0: wenn 1, dann R4=1111, sonst R4=0000
3       AND     R3,R0,R2        Ist das letzte Bit von R0 = 1 ?
4       SUB     R4,R6,R3        R4 ist jetzt 1111 oder 0000, abhängig von R3
5       AND     R7,R1,R4        Multiplikation von R1 mit Bit 0 von R0 nach R7
Verschiebe Bits von R0 nach rechts für Multiplikation mit Bit 1
6       SLR     R0,R0
7       SETF    R4              R4 zurück auf 0 setzen
```

```
Ziel: R6 R7 = R1 * R0, werte Bits von R0 aus, multipliziere bitweise mit R1
Initialisierung
1       SETF    R6              Initialisieren von R6 mit 0000
2       INC     R2,R6           R2 ist jetzt 0001
Teste Bit 0 von R0: wenn 1, dann R4=1111, sonst R4=0000
3       AND     R3,R0,R2        Ist das letzte Bit von R0 = 1 ?
4       SUB     R4,R6,R3        R4 ist jetzt 1111 oder 0000, abhängig von R3
5       AND     R7,R1,R4        Multiplikation von R1 mit Bit 0 von R0 nach R7
Verschiebe Bits von R0 für Multiplikation mit Bit 1
6       SLR     R0,R0
7       SETF    R4              R4 zurück auf 0000 setzen
Teste Bit 1 von R0: wenn 1, dann R4=1111, sonst R4=0000
8       AND     R3,R0,R2        Ist das letzte Bit von R0 = 1 ?
9       SUB     R4,R4,R3        R4 ist jetzt 1111 oder 0000, abhängig von R3
10      AND     R5,R1,R4        Multiplizieren von R1 mit Bit1 von R0 nach R5
Addiere 2*R5 mit Carry-Übertrag in R4 für Zwischensumme
11      SETF    R4              R4 zurück auf 0000 setzen
12      ADD     R5,R5,R5        Verschieben von R5 um ein Bit links (mit Carry)
13      ADC     R4,R4,R4        Auffangen eines eventuell entstehenden Carrys
Akkumuliere die erste Teilsumme aus (R7 + 2*R5) mit Carry-Übertrag aus R4
14      ADD     R7,R5,R7        Hinzufügen zur Summe (niederwertiger Teil)
15      ADC     R6,R4,R6        Hinzufügen zur Summe (höherwertiger Teil)
```

```
Verschiebe Bits von R0 für Multiplikation mit Bit 2
16      SLR     R0,R0
17      SETF    R4              R4 zurück auf 0000 setzen
Teste Bit 2 von R0: wenn 1, dann R4=1111, sonst R4=0000
18      AND     R3,R0,R2        Ist das letzte Bit von R0 = 1 ?
19      SUB     R4,R4,R3        R4 ist jetzt 1111 oder 0000, abhängig von R3
20      AND     R5,R1,R4        Multiplizieren von R1 mit Bit1 von R0 nach R5
```

```
Verschiebe Bits von R0 für Multiplikation mit Bit 2
16      SLR     R0,R0
17      SETF    R4              R4 zurück auf 0000 setzen
Teste Bit 2 von R0: wenn 1, dann R4=1111, sonst R4=0000
18      AND     R3,R0,R2        Ist das letzte Bit von R0 = 1 ?
19      SUB     R4,R4,R3        R4 ist jetzt 1111 oder 0000, abhängig von R3
20      AND     R5,R1,R4        Multiplizieren von R1 mit Bit1 von R0 nach R5
Addiere 4*R5 mit Carry-Übertrag in R4 für Zwischensumme
21      SETF    R4              R4 zurück auf 0000 setzen
22      ADD     R5,R5,R5        Verschieben von R5 um ein Bit links (mit Carry)
23      ADC     R4,R4,R4        Auffangen eines eventuell entstehenden Carrys
24      ADD     R5,R5,R5        Verschieben um noch ein Bit links (mit Carry)
25      ADC     R4,R4,R4        Auffangen eines eventuell entstehenden Carrys
Akkumuliere die zweite Teilsumme aus (R7 + 4*R5) mit Carry-Übertrag aus R4
26      ADD     R7,R5,R7        Hinzufügen zur Summe (niederwertiger Teil)
27      ADC     R6,R4,R6        Hinzufügen zur Summe (höherwertiger Teil)
```

Der Code gilt nur für positive Zahlen (MSB von R0 und R1 beide 0), d.h. es werden nur die Bits 0 – 2 von R0 berücksichtigt.

Auf der nächsten Folie sehen Sie die Registerbelegungen nach jedem Programmschritt.

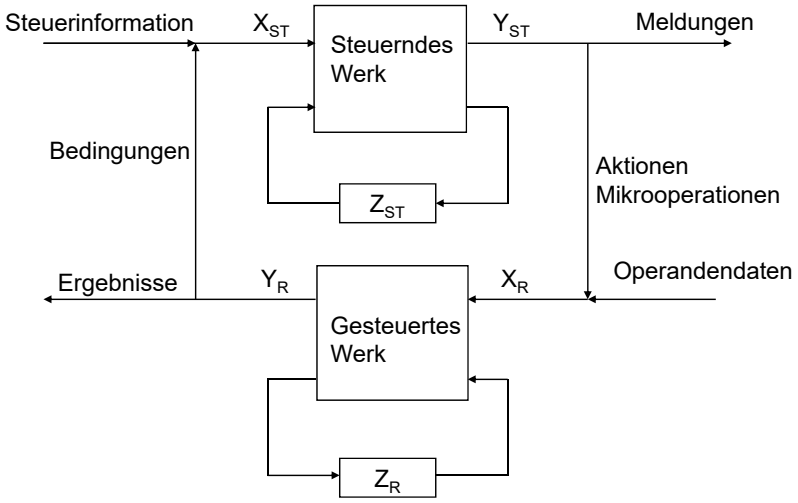
Register:	R0	R1	R2	R3	R4	R5	R6	R7
Programmschritt:	0	0	2	3	4	10	1	5
Registerinhalt:	7 = 0111	3 = 0011	0001	0001	1111	0011	0000	0011
	6			8	7	12	15	14
	0011			0001	0000	0110	0000	1001
	16			18	9	20	27	26
	0001			0001	1111	0011	21 = 0001	0101
					13	22		
					0000	0110		
					17	24		
					0000	1100		
					19			
					1111			
					21			
					0000			
					23			
					0000			
					25			
					0000			

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.

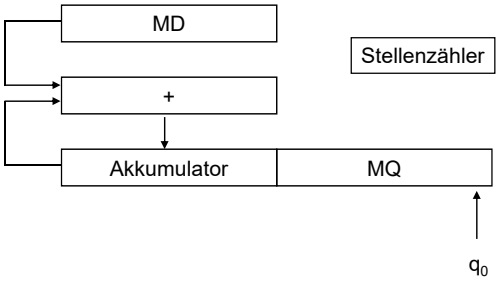
Die folgende Folie zeigt das grundsätzliche Prinzip eines Steuerwerks: Es gibt ein steuerndes (Schalt-)werk und ein gesteuertes Werk. Das gesteuerte Werk ist häufig ein Rechenwerk. Das Steuerwerk hat - wie jedes Schaltwerk - Eingaben, Ausgaben und Zustände. Die Eingaben sind einerseits die Befehle der 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 AKKU, 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 entsprechenden Register geladen, der Akkumulator wird mit 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. Wenn q_0 =0 ist, wird die Mikrooperation „0“ generiert, die kein Register verändert.

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.
Die folgende Seite zeigt ein Beispiel für die Multiplikation der Zahlen 5 in MD (0101) und 11 in MQ (1011), Ergebnis 5 * 11 = 55 (0 0 1 1 0 1 1 1).

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	

Abfolge im Multiplizierwerk

Beispiel: Berechne B * A, Ergebnis E in (Akku-MQ)
B= 5 = (0101) in MD
A = 11 = (1101) in MQ
AKKU = (0000)
E= Zwischenergebnisse

	MD = B = 0 1 0 1								
	Akku = 0 0 0 0				MQ = A = 1 0 1 1				
Addiere $0+B*a_0$, $a_0=1$, E= 5	0	1	0	1	a_3	a_2	a_1	a_0	
Schiebe 1 Bit rechts	0	0	1	0	1	a_3	a_2	a_1	a_0
Addiere $(2a_1+a_0)B$, $a_1=1$, E=15	0	1	1	1	1	a_3	a_2	a_1	
Schiebe 1 Bit rechts	0	0	1	1	1	1	a_3	a_2	a_1
Addiere $(4a_2+2a_1+a_0)B, a_2=0$, E=15	0	0	1	1	1	1	a_3	a_2	
Schiebe 1 Bit rechts	0	0	0	1	1	1	1	a_3	a_2
Add. $(8a_3+4a_2+2a_1+a_0)B, a_3=1$, E=55	0	1	1	0	1	1	1	a_3	
Schiebe 1 Bit rechts	0	0	1	1	0	1	1	1	a_3

q0

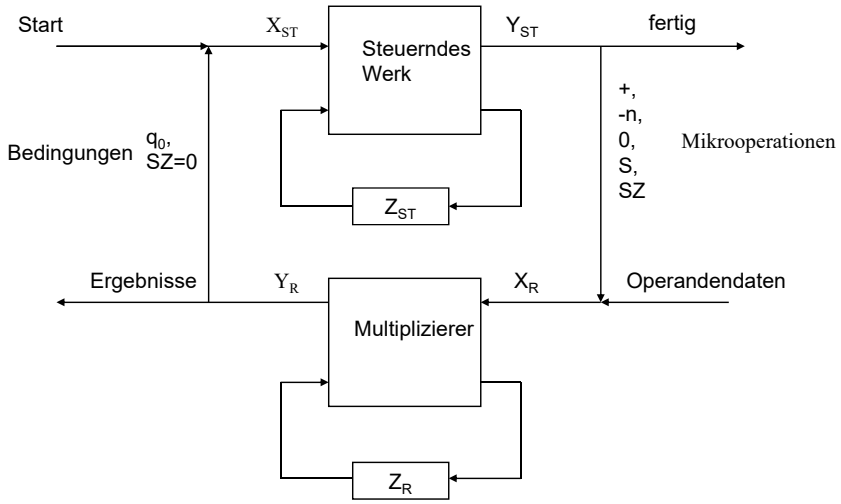
Befehle des Steuerwerks

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:

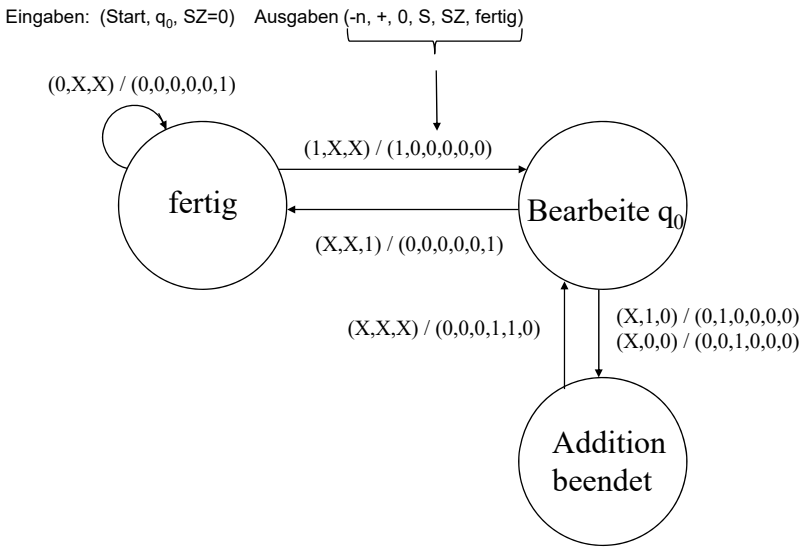
- Wenn ein „Start“-Signal kommt, muss der Stellenzähler mit „-n“ initialisiert werden.
- Wenn q_0 interpretiert wird, muß MD genau dann auf den Akkumulator addiert werden, wenn $q_0 = 1$ ist.
- Nach jedem solchen Additionsschritt muss der Akkumulator und das MQ um ein Bit geschoben und der Stellenzähler 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 mit Mikrooperationen



Automatengraph



Automaten-Zustände

Wir codieren die Zustände mit

00: fertig
01: Bearbeite q_0
10: Addition beendet
11: kommt nicht vor: don‘t 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: