DHBW
Duale Hochschule
Baden-Württemberg
Stuttgart

Mikrocomputertechnik II

Vorlesung und Übungen

Ausgabe 0.05, 10.03.2014
Autor: Stephan Rupp

Inhaltsverzeichnis

1. Digitale Schaltungen

1.1. Kombinatorische Logik

Unter kombinatorischer Logik versteht man Schaltungen, mit denen sich Funktionen der Boolschen Algebra realisieren lassen, also logische Ausdrücke, die mit UND, ODER und weiteren Operatoren realisierbar sind. Die Logik bzw. auch Schaltfunktion stellt einen algebraischen Ausdruck dar. Folgende Abbildung zeigt eine solche Schaltfunktion.

Wertetabelle

Eingänge		Ausgänge			
x1	x2	y1	y2	y3	y4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Schaltfunktion

y1 = x1‘ x2‘
y2 = x1‘ x2
y3 = x1 x2‘
y4 = x1 x2

Schaltnetz

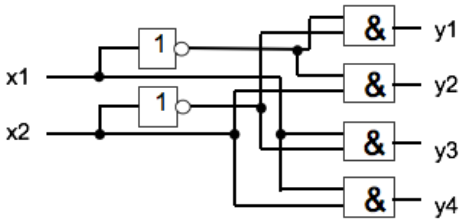


Bild 1.1 Schaltfunktion

Die Abbildung zeigt eine Schaltfunktion f(x1, x2) mit zwei Eingangswerten. Die Schaltfunktion umfasst die 4 Vektoren y1, y2, y3 und y4. Für die Beschreibung der Schaltfunktion sind folgende Methoden gezeigt: die Wertetabelle bzw. Wahrheitstabelle, algebraische Gleichungen, sowie ein Schaltnetz mit Logik-Gattern. Diese Beschreibungen sind gleichwertig.

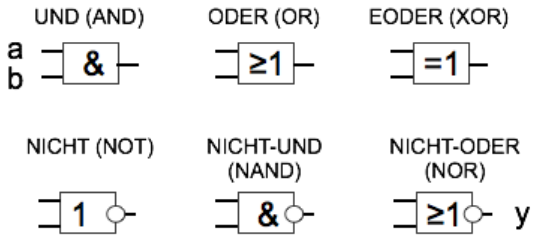
Bei der algebraischen Gleichung wurden die Operatoren UND als Multiplikation dargestellt (wobei der Multiplikations-Operator * weggelassen wurde), der Operator ODER als Addition (mit dem Zeichen +), sowie die Negation mit Apostroph (x‘ für NICHT x). Diese Schreibweise ist etwas kompakter als die sonst gebräuchlichen Operatoren ∧ (für logisches UND), ∨ (für logisches ODER), und Oberstrich für die Negation. Die Schaltfunktion liesse sich auch als Programmcode beschreiben, siehe folgender Text.

```
boolean x1, x2, y1, y2, y3, y4;
//
y1 = !x1 && !x2;
y2 = !x2 &&  x2;
y3 =  x1 && !x2;
y4 =  x1 &&  x2;
```

Hierbei kennzeichnet && den UND-Operator und !x die Negation. Alle Variablen sind boolsche Variable mit den Werten wahr (=1) oder falsch (=0).

Logik-Gatter

Für Logik-Gatter sind die in folgender Abbildung gezeigten Operatoren üblich. Außer den bereits genannten Operatoren UND, ODER sowie der Negation sind die speziellen Operatoren Exklusiv-ODER, Nicht-UND, sowie Nicht-Oder hinzugekommen. Die Abbildung zeigt auch die hierfür gebräuchlichen Schaltsymbole, sowie die Wertetabelle der Operatoren.



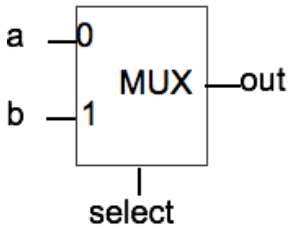
Logik-Gatter							
A	B	AND	OR	XOR	NOTB	NAND	NOR
0	0	0	0	0	1	1	1
1	0	0	1	1	1	1	0
0	1	0	1	1	0	1	0
1	1	1	1	0	0	0	0

Bild 1.2 Logik-Gatter

Zur Darstellung der Wertetabelle als algebraischer Ausdruck genügen die Operatoren UND, ODER sowie die Negation. Die Schaltzeichen dienen nur dem rascheren Verständnis der gewünschten Operation im Kontext einer kombinatorischen Logik.

Multiplexer

Als Beispiel für ein Schaltnetz soll der in folgender Abbildung gezeigter Multiplexer dienen. Ein Multiplexer dient dazu, ausgewählte Eingangssignale auf den Ausgang zu schalten. Zur Auswahl der Eingangssignale werden Steuersignale verwendet. Im Sinne der Schaltfunktion stellen die zu schaltenden Eingänge und die Steuersignale Eingangssignale dar.



select	a	b	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

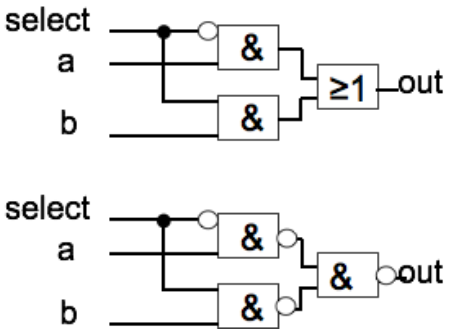


Bild 1.3 Multiplexer

Die Wertetabelle zeigt die Funktion der Schaltung: Mit Hilfe des Steuersignals select = 0 wird der Eingang a auf den Ausgang geschaltet. Mit Hilfe des Steuersignals select = 1 wird der Eingang b auf den Ausgang geschaltet. Mit Blick auf die Werte 1 in der Wertetabelle lässt sich folgende Schaltfunktion finden: out = select‘ a b‘ + select‘ a b + select a‘ b + select a b.

Übung 1.1: Zeigen Sie, dass sich diese Schaltfunktion reduzieren lässt zu der Schaltung, wie in der Abbildung oben rechts gezeigt.

Übung 1.2: Zeigen Sie, dass sich die Schaltung in der Abbildung unten rechts aus der Schaltung darüber durch Umformung ergibt.

Wollte man den Multiplexer programmieren, wäre folgender Programmcode als Beschreibung der Funktion geeignet.

```
boolean a,b, select, out;
```

```
    if (!select) out = a;
    else out = b;
```

Sowohl die Schaltnetze in der Abbildung als auch der Programmcode zeigen unmittelbar die Funktionsweise des Multiplexers: Abhängig vom Steuersignal select wird entweder Eingang a oder Eingang b auf den Ausgang geschaltet.

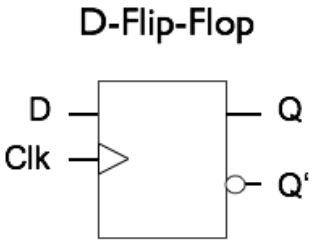
1.2. Register

Die bisher gezeigten Logik-Bausteine dienen der Kombination von Signalen. Das Ausgangssignal folgt hierbei der Schaltung, sobald sich ein Eingangssignal verändert hat. Die Laufzeiten der Signaländerung richten sich hierbei nach der Gatterlaufzeit und der Anzahl der jeweils zu durchlaufenden Gatter. Diese Art von Schaltungen wird auch als *Schaltnetz* bezeichnet.

Anders verhält sich Logik, bei der man Signale mit Hilfe eines Taktsignals übernimmt. Hier werden die Signale nur zu einem vorgegebenen Zeitpunkt abgetastet bzw. interpretiert. Beispielsweise wird ein Signal dann übernommen, wenn das Taktsignal eine steigende bzw. fallende Flanke zeigt. Solche zeitabhängigen Schaltungen werden auch als *Schaltwerk* bezeichnet.

D-Flip-Flop

Der einfachste Baustein für taktsynchrone Logik ist ein sogenanntes D-Flip-Flop, wie in folgender Abbildung gezeigt.



Eingänge		Ausgänge	
D	Clk	Q	Q'
0	^	0	1
1	^	1	0
x	0	Q_{alt}	Q'_{alt}
x	1	Q_{alt}	Q'_{alt}

Bild 1.4 D-Flip-Flop

Der Baustein verfügt über einen Signaleingang D und einen Ausgang Q (sowie den negierten Ausgang Q‘). wie die Wertetabelle zeigt, hat das Eingangssignal keinen direkten Einfluss auf den Ausgang. Nur dann, wenn am Takteingang Clk (für engl. Clock) eine steigende Taktflanke vorliegt, wird das Eingangssignal in den Baustein übernommen. In der Wertetabelle ist die steigende Taktflanke durch das Symbol ^ gekennzeichnet.

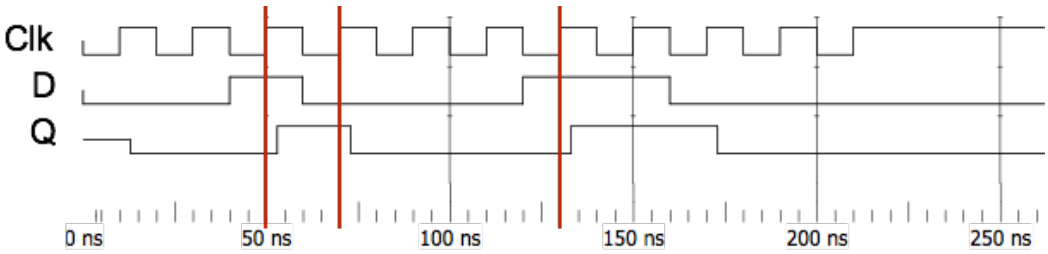


Bild 1.5 Zeitverhalten des D-Flip-Flops

Erreicht der Takt den konstanten Pegel 0 oder 1, spielt der Wert des Eingangssignals überhaupt keine Rolle (in der Wertetabelle gekennzeichnet durch das Zeichen x für „don‘t care“). In diesem Fall behält der Ausgang den zuletzt gespeicherten Wert. Wie das Beispiel zeigt, lässt sich das Flip-Flop als getaktete Logik zwar mit einigen zusätzlichen Vereinbarungen mit Hilfe einer Wertetabelle beschreiben, einfacher gelingt jedoch die Beschreibung mit Hilfe eines Zeitdiagramms.

Das Zeitdiagramm zeigt, dass der Wert des Ausgangssignals Q des Flip-Flops dem des Eingangssignals D folgt. Mit jeder steigenden Taktflanke (siehe Taktsignal in der ersten Zeile) wird mit einer geringfügigen Schaltverzögerung der Wert des Eingangssignals übernommen. Dieser Wert bleibt auf jeden Fall bis zur nächsten steigenden Taktflanke gespeichert. Zu diesem Zeitpunkt wird dann der aktuelle Eingangswert des Eingangssignals übernommen. Dieses verhalten entspricht somit einer Abtastung des Eingangssignals D mit jeder steigenden Taktflanke.

Wollte man das Verhalten des D-Flip-Flops programmieren, wäre zunächst ein Modell für die Abhängigkeit des Taktsignals zu suchen. Eine Interrupt-Service-Routine wäre hierzu geeignet, wobei als Interruptsignal der Takt zu verwenden wäre. In folgendem Beispiel wird dies mit Hilfe der Methode attachInterrupt(), gelöst, die mit Hilfe des Parameters RISING bei jeder steigender Flanke des Taktsignals auslöst. Die Programmschleife loop() wird dann jedes Mal zugunsten der Interrupt-Service Routine unterbrochen. Das Signal D wäre über einen Digitaleingang einzulesen. Der Ausgang Q folgt de, Eingangssignal D zum Zeitpunkt jeder ansteigenden Taktflanke.

```
int outPin = 13;
volatile int D, Q;

void setup() {
  pinMode(outPin, OUTPUT);
  attachInterrupt(0, flipFlop, RISING);
}

void loop() {
  digitalWrite(outPin, Q);
}

void flipFlop(){
  Q = D;
}
```

Schieberegister

Mehrere Flip-Flips lassen sich zu einem Schieberegister zusammen schalten. Folgende Abbildung zeigt diese Anordnung für eine Kette von 8 Bits. Das erste Flip-Flop hat als Eingang das Signal D. Die Ausgangssignale der folgenden Flip-Flops sind mit den jeweiligen Eingangssignalen verbunden, bis schliesslich das letzte Flip-Flop der Kette das Ausgangssignal Q schaltet.

Innerhalb der Kette wird so mit jedem Takt der aktuelle Wert jedes Flip-Flops in das folgende Flip-Flop übertragen. Bis das aktuelle Eingangssignal D am Ende der Kette als Signal Q erscheinen kann, vergehen 8 Takte.

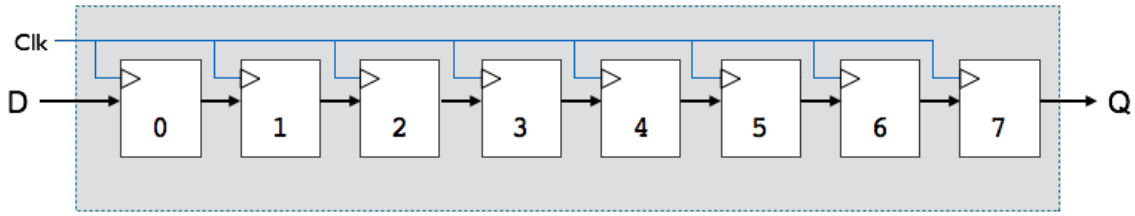


Bild 1.6 Schieberegister

Folgende Abbildung zeigt das Zeitverhalten des Schieberegisters. Wie zu erwarten, wird das Eingangssignal D mit dem ersten steigenden Takt in das Schieberegister übernommen und taucht 7 weitere Takte später am Ausgang Q auf. Das Signal Q ist gegenüber D um insgesamt 8 Takte später.

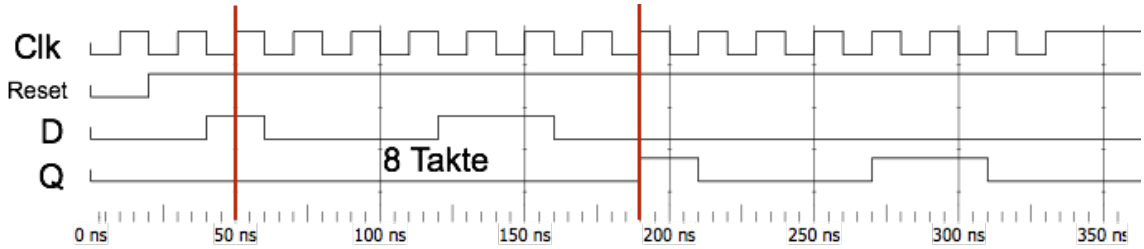


Bild 1.7 Zeitverhalten des Schieberegisters

Möchte man das Verhalten des Schieberegister durch ein Programm beschreiben, wäre folgender Programmtext geeignet. Hierbei ist angenommen, dass die Routine shiftReg() als Interrupt-Service-Routine angemeldet ist, die auf steigende Taktflanken reagiert. Mit jedem Takt wird das letzte Register in den Ausgang Q geschrieben, und anschliessend der Inhalt des Registers eine Stelle weiter geschoben. Das erste Register nimmt den Wert am Eingang auf. Im Programm wird hierfür eine Zählschleife verwendet.

```
void shiftReg(){
    Q = Reg(7);
    for (int i = 0; i<7, i++) {
       Reg(7-i) = Reg (7-i-1);
    }
    Reg(0) = D;
}
```

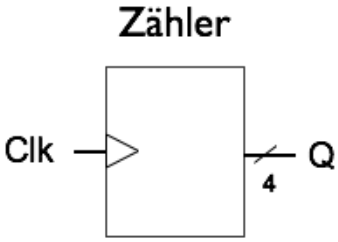
Übung 1.3: Skizzieren Sie eine Schaltung, die es erlaubt 8 Bits parallel in ein Schieberegister zu übertragen, um anschliessend das Schieberegister seriell über den Ausgang Q auszulesen. Hinweis: Verwenden Sie Multiplexer mit geeigneten Steuersignalen.

Übung 1.4: Beschreiben Sie das Zeitverhalten der Schaltung für einen Schreibvorgang (parallel) und einen Lesevorgang mit allen hierzu benötigten Signalen. Hinweis: Erstellen Sie ein Zeitdiagramm.

1.3. Zähler

Um eine definierte Anzahl Bits zu übertragen, bzw. um einen Zeitraum in Taktintervallen abzumessen bzw. einen Takt zu teilen sind Zähler nützlich. Im einfachsten Fall besitzt ein Zähler nur einen

Takteingang und geeignet kodierte Ausgänge, wie z.B. der in folgender Abbildung gezeigte 4-Bit Zähler für Zahlen von 0 bis 15 (bzw. 0 bis F in hexadezimaler Schreibweise.



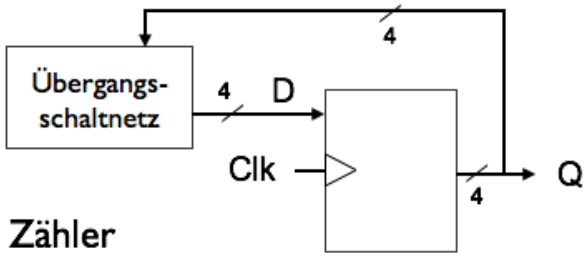
Q3	Q2	Q1	Q0	Hex
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	B
1	1	0	0	C
1	1	0	1	D
1	1	1	0	E
1	1	1	1	F

Bild 1.8 4-Bit Zähler mit Wertetabelle

Auch als Programmcode ist ein solcher Zähler einfach zu beschreiben, wie folgendes Beispiel zeigt. Hierbei ist angenommen, dass die Interrupt-Service-Routine counter() mit jeder steigenden Taktflanke aufgerufen wird. Der Zählbereich lässt sich einfach als Obergrenze einstellen.

```
volatile int cnt = 0;
//
void counter(){
        cnt = cnt +1;
        if (cnt >= 16) then cnt = 0;
    }
    Reg(0) = D;
}
```

Bei der praktischen Realisierung als Schaltkreis wäre eine Möglichkeit, den Zähler als rückgekoppeltes Schieberegister mit der benötigten Anzahl an Bits zu realisieren, beispielsweise für einen Zähler von 1 bis 8 mit 8 Registern. Als Startwert (Preset) wird beispielsweise das unterste Bit zu 1 gesetzt. Um den Zählerstand binär mit insgesamt 3 Stellen zu kodieren (von 0 bis 7), wäre ein Encoder von 8 Bits auf 3 Bits als Schaltnetz zu realisieren.



Q3	Q2	Q1	Q0	D3	D2	D1	D0	Hex
0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0	1
0	0	1	0	0	0	1	1	2
0	0	1	1	0	1	0	0	3
0	1	0	0	0	1	0	1	4
0	1	0	1	0	1	1	0	5
0	1	1	0	0	1	1	1	6
0	1	1	1	0	0	0	0	7

Bild 1.9 Realisierung des Zähler mit Register und Übergangsschaltnetz

Eine andere mögliche Realisierung ist in der Abbildung oben beschrieben. Hier übernimmt ein Register das Speichern des aktuellen Zählerstandes. Der Ausgang des Registers führ zu einem Schaltnetz, das den aktuellen Zustand des Registers in den folgenden Zustand übersetzt. Beträgt der aktuelle Zählerstand z.B. „0000“ = 0, so soll der folgende Zustand „0001“ = 1 betragen. Der Ausgang des Schaltnetzes führt an den Eingang des Registers.

Da das Schaltnetz nicht taktabhängig reagiert, wird nach Vorliegen des Ausgangs Q sofort der Folgezustand D im Schaltnetz berechnet (bzw. mit den üblichen Gatterlaufzeiten). Zum nächsten Takt wird dieser Zustand D dann in das Register übertragen und erscheint als aktueller Zustand Q am Ausgang des Registers. Mit Hilfe der Definition des Folgezustands erfolgt auch der maximale Stand des Zählers. Beim in der Abbildung gezeigten Beispiel folgt auf den Zustand 7 wieder der Zustand 0, d.h. dieser Zähler zählt von 0 bis 7.

Folgende Abbildung zeigt das Zeitverhalten des Zählers. Hierbei werden alle 4 Bits des Zählers verwendet, d.h. der Zähler zählt vom 0 bis 15 (bzw. hexadezimal 0 bis F).

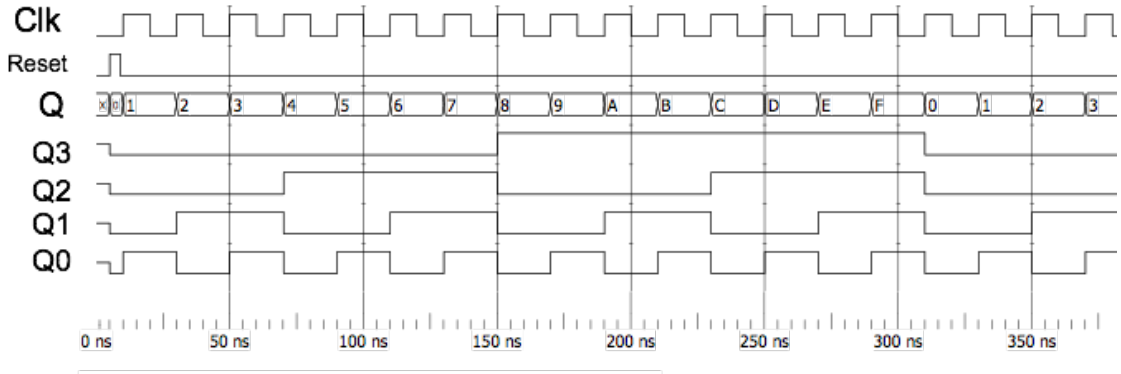


Bild 1.10 Zeitdiagramm des Zählers

Übung 1.5: Skizzieren Sie das Schaltbild eines Zählers von 1 bis 8 (bzw. 0 bis 7), der mit Hilfe eines rückgekoppelten Schieberegisters und eines Encoders realisiert ist. Beschreiben Sie die Wertetabelle für den Encoder. Skizzieren Sie ein passendes Schaltnetz.

Übung 1.6: Skizzieren Sie die Schaltfunktion für das Übergangsschaltnetz des Zählers aus Abbildung 1.9. Geben Sie ein Schaltnetz hierfür an und optimieren Sie gegebenenfalls.

1.4. Tabellen und Speicher

Tabellen oder Speicher ermöglichen beispielsweise die Unterbringung einer Wertetabelle und somit ebenfalls die Möglichkeit zur Realisierung einer Schaltfunktion. Ganz allgemein lassen sich Tabellen bzw. Speicher zum Schreiben bzw. Lesen von Informationen verwenden. Speicher sind aus einer Kombination aus Adressdekoder und Speicherzellen aufgebaut.

Für einen Speicher, aus dem nur ausgelesen werden soll, werden als Eingänge Adressleitungen benötigt. Die Adressleitungen definieren in binärer Schreibweise die auszulesende Speicherzelle. Werden beispielsweise im Speicher 8 Bits bzw. 8 Teilen gespeichert, so werden 3 Adressleitungen benötigt, um die gewünschte Speicherzelle zu adressieren. Im allgemeinen lassen sich mit Hilfe von n Adressleitungen 2^n Zeilen adressieren.

Mit Bezug zur Wertetabelle einer Schaltfunktion entspricht jede Spalte der Speicherzelle einer Spalte der Wertetabelle. Die Anzahl der Spalten lässt sich auch zu einem gespeicherten Wort zusammenfassen, die Bits aus 8 Spalten beispielsweise zu einem Byte. Folgende Abbildung zeigt als Beispiel für die Realisierung einer Wertetabelle in einem Speicher den bereits eingangs gezeigten Multiplexer. In diesem Fall wird die Wertetabelle einfach komplett zeilenweise abgespeichert. Die Auswahl einer speziellen Zeile geschieht über die Adressleitungen: Adresse „010“ liefert am Ausgang des Speichers den Eintrag für die dritte Zeile (Zeile 2 von 0 aus gezählt).

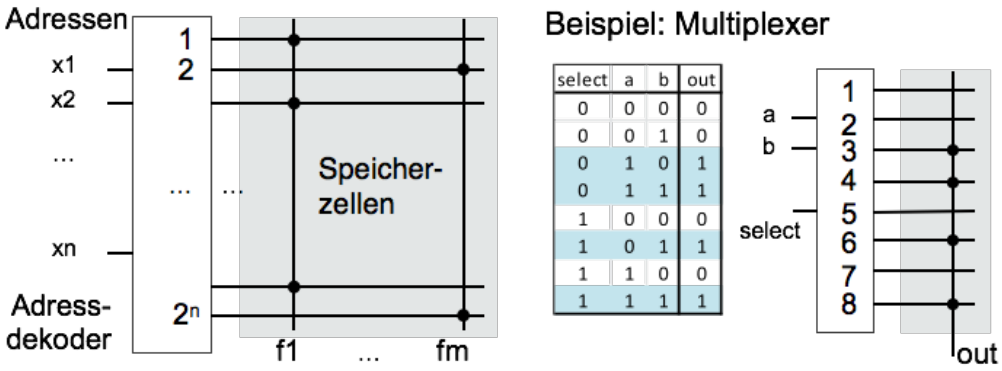


Bild 1.11 Speicher

Der so dargestellte Speicher ist als reines Schaltnetz ausgeführt. Der Tabelleneintrag wird unmittelbar im Anschluss an das Anlegen einer Adresse an den Ausgang gegeben. Im Zusammenhang mit Mikrocontrollern gebräuchlicher sind taktgesteuerte Speicher: Hier wird nur bei Vorliegen einer Taktflanke Information ausgegeben bzw. eingelesen. Solche Speicher entsprechen einer Kombination eines Adressdecoders mit Registern, wie folgende Abbildung zeigt.

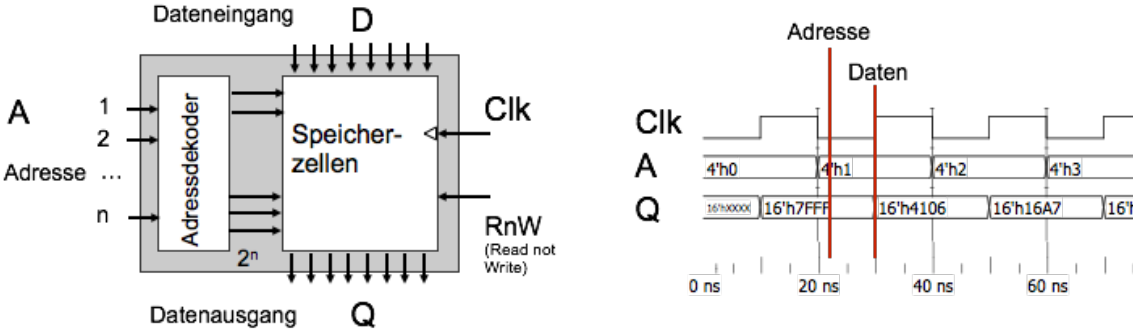


Bild 1.12 taktgesteuerter Speicher

Der Speicher hat folgende Eingänge: Adresse für die zu lesende bzw. für die zu beschreibende Speicherzeile, einen Dateneingang für Schreiboperationen, sowie einen Datenausgang für Leseoperationen. Das Steuersignal RnW (für „read, not write“) gibt vor, ob gelesen oder geschrieben werden soll. Bei Vorliegen eines Taktes erfolgt die Ausgabe des adressierten Datums an den Datenausgang, bzw. das Speichern des Signals am Dateneingang in die adressierte Speicherzeile. Das Zeitdiagramm neben dem Blockschaltbild zeigt den Ablauf für Leseoperationen beginnend von Adresse 0.

Im Beispiel wurde 4 Adressleitungen verwendet, der Speicher besitzt eine Wortbreite von 16 Bits. Man erkennt, dass vor dem Arbeitstakt zum Lesen bzw, Schreiben eine gültige Adresse vorliegen muss. Im Beispiel steht die Adresse bereits mit der vorausgegangenen fallenden Taktflanke zur Verfügung.

Wollte man das Verhalten eines Speichers mit Programmcode beschreiben, so würde man jede Zeile des Speichers als Vektor bzw. Array mit einem Datentyp passender Wortbreite definieren. Einen Speicher mit 32 Zeilen der Wortbreite 16 Bit also beispielsweise als Array mit 32 Worten vom Datentyp Short (16 Bit). Die Adresse wird als Index des Arrays verwendet. Das Auslesen (bzw. das Schreiben) wäre dann eine Wertzuweisung an die jeweilige Adresse. Folgender Programmtext zeigt ein Beispiel.

```
// global variables
short memory[32], dIn, dOut;
int adr;
boolean RnW;

// main loop
if (RnW) then dOut = memory(adr);
else memory(adr) = dIn;
```

Speicher wird üblicherweise entweder gelesen oder beschreiben wird. Den Modus bestimmt ein Steuersignal, z.B. RnW („read, not write“). Es genügt eine Adressleitung. Im speziellen Fall ist jedoch auch das gleichzeitige Lesen und Schreiben eines Speichers möglich. Solche Zweiport-Speicher (engl. dual ported RAM) benötigen dann zwei getrennte Adressleitungen zum Lesen (RA für „read address“) und zum Speichern (WA für „write address“). Damit nicht mit jedem Takt in den Speicher geschrieben wird, dient ein Signal W_EN („write enable“) der Steuerung. Folgende Abbildung zeigt den Speicher und das Zeitdiagramm.

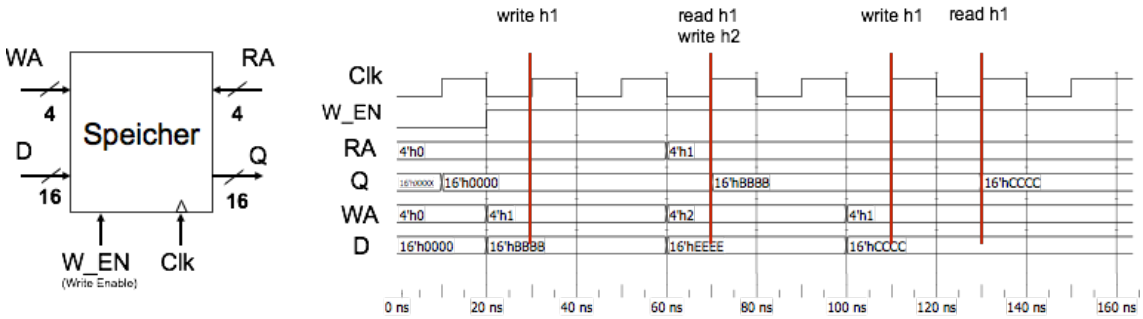


Bild 1.13 Zweiport-Speicher (Dual-ported RAM)

Solange unterschiedliche Speicheradressen gewählt werden ist gleichzeitiges Schreiben und Lesen möglich. Wie beim regulären Speicher werden die Daten nach Anliegen der Adressen mit dem nächsten Arbeitstakt bereit gestellt. Soll in die gleiche Adresse geschrieben und gelesen werden, so wird zunächst der bereits gespeicherte Wert ausgegeben, währen der Speicher mit dem gleichen Takt beschrieben wird. Mit dem folgenden Arbeitstakt steht der neue Wert dann zum Auslesen bereit. Dieses Verhalten entspricht dem eines D-Flip-Flops bei der Aufnahme eines neuen Signals.

Übung 1.7: Ein Mikrocontroller verfügt über 128 kBytes Programmspeicher der Wortlänge 16 Bit und über 4 kBytes Arbeitsspeicher der Wortlänge 8 Bit. Welche Breite haben die Adressbusse für den Programmspeicher und für den Arbeitsspeicher?

Übung 1.8: Wie viele Abtastwerte im 32 Bit-Format (bzw. im 16 Bit Format) können Sie im Arbeitsspeicher des Mikrocontrollers ablegen? Welche Möglichkeiten zur Erweiterung gibt es grundsätzlich?

2. Rechenwerk

2.1. Addierwerk

Ein einfaches Rechenwerk soll zunächst in der Lage sein, Zahlen im binären Zahlensystem zusammenzuzählen. Die Addition im Zweiersystem läuft hierbei genauso ab wie im Zehnersystem: Die Summe zweier Zahlen kann zu einem Übertrag auf die nächste Stelle im Zahlensystem führen. Eine binäre Eins „1“ addiert sich mit einer weiteren binären Eins „1“ zu einer binären Zwei, also „10“. Die Funktion eines solchen Halbaddierers lässt sich als Schaltlogik wie in der folgenden Abbildung gezeigt realisieren.

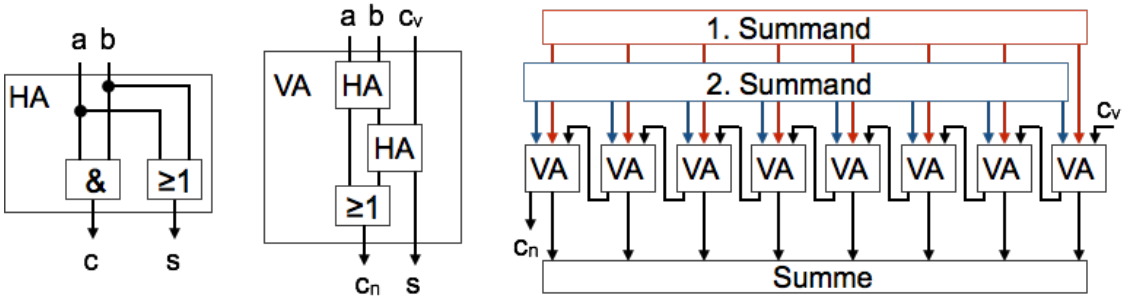


Bild 2.1 Halbaddierer, Volladdierer und Addierwerk

Aus den binären Zahlen a und b entsteht die Summe s, sowie der Übertrag c (engl. carry forward). Möchte man mehr als zwei binäre Zahlen addieren, so muss man davon ausgehen, dass es auch einen Übertrag aus der vorausgegangenen Addition gibt. Man benötigt außer den Zahlen a und b auch einen Eingang für einen bereits vorhandenen Übertrag. Eine solche Anordnung aus zwei Halbaddierern und einer ODER-Verknüpfung Zeigt der Volladdierer in der Abbildung. Aus einer Kette von Volladdierern entsteht schließlich das in der Abbildung rechts gezeigte Addierwerk für zwei 8-Bit Werte. Die Überträge werden hierbei zur nächsten höherwertigen Stelle addiert.

Addition und Subtraktion

Um außer der Addition aus Subtraktionen auszuführen, muss für die binären Zahlen ein Vorzeichen eingeführt werden. Hierfür wird vereinbart, dass das erste Bit als Vorzeichen verwendet wird. Für positive Zahlen ist das Vorzeichenbit 0, für negative Zahlen ist das Vorzeichenbit 1. Mit Hilfe des Vorzeichens und der verbleibenden 7 Bits lassen sich Zahlen im Bereich -128 bis +127 darstellen. Folgende Abbildung zeigt die hierzu übliche Darstellung im Zweierkomplement.

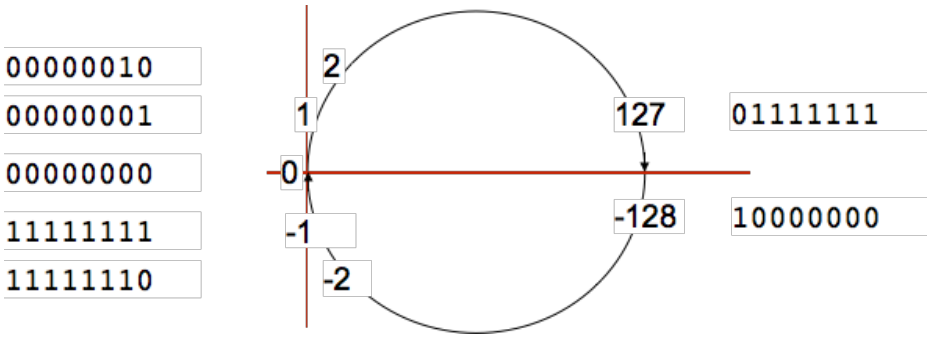


Bild 2.2 Darstellung binärer Zahlen als Zweierkomplement

Das Zweierkomplement einer positiven Zahl a erhält man durch folgende Operationen: (1) Negieren aller Bits, (2) Addieren einer 1. Diese Operationen werden für die Bildung der Subtraktion im Addierwerk realisiert. Außerdem erhält das Rechenwerk noch eine Status-Anzeige (Status-Bits), die es ermöglicht, zu prüfen, ob ein Überlauf eingetreten ist und ob ein Ergebnis gleich Null ist. Folgende Abbildung zeigt das Rechenwerk.

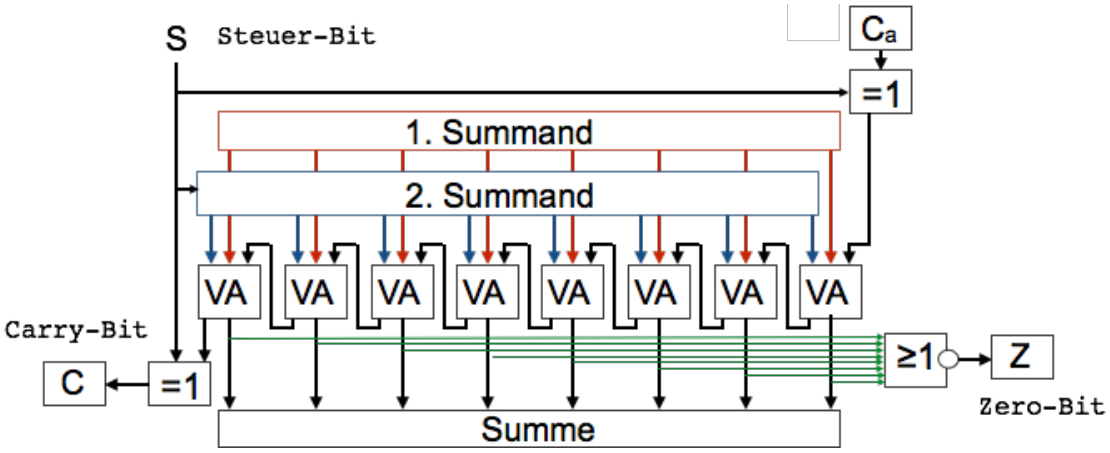


Bild 2.3 Rechenwerk für arithmetische Operationen mit Status-Bits

Vorzeichenlose Zahlen

Der Übertrag der Berechnung steht im Rechenwerk als Carry-Bit zur Verfügung. Ein eventuell vorhandener alter Übertrag C_a aus einer vorausgegangenen Operation wird ebenfalls berücksichtigt. Die Verknüpfung aller Bits des Ergebnisses mit einer NOR-Operation zeigt als Zero-Bit an, ob das Ergebnis gleich Null ist (alle Bits sind Null, und somit das Zero-Bit gleich Eins).

Außerdem enthält das Rechenwerk ein Steuersignal S mit folgender Aufgabe: Mit S=0 arbeitet das Rechenwerk als Addierwerk. In diesem Fall entspricht das Carry-Bit dem Übertrag der Operation und kann zur Anzeige eines Überlaufs verwendet werden. Mit der Vorgabe S=1 für das Steuerwerk wird eine Subtraktion ausgeführt. Hierzu wird der zweite Operand (der Subtrahend) negiert, sowie das mitgeführte alte Carry-Bit negiert. Dies führt ohne Vorgeschichte zu einem Übertrag C_a=1 für den 2. Operanden, der somit der Zweierkomplement-Darstellung seines negativen Wertes entspricht. Die Subtraktion wird somit als Addition des negativen Wertes des Subtrahenden ausgeführt. Das Steuersignal S=1 negiert außerdem das Carry-Bit des Ergebnisses.

Übung 2.1: Testen Sie die Operation des Rechenwerkes für folgende Aufgaben: (1) Addition zweier binärer Zahlen (z.B. 3 plus 2), (2) Subtraktion zweier positiver binärer Zahlen mit positivem Ergebnis (z.B. 3 minus 2), (3) Subtraktion zweier positiver binärer Zahlen mit Ergebnis Null, (4) Subtraktion zweier positiver binärer Zahlen mit negativem Ergebnis. Welche Bedeutung hat das Carry-Bit jeweils? Hinweis: Führen sie die Operationen im binärem Zahlenformat ohne Vorzeichen aus.

Übung 2.2: Kann das Rechenwerk folgende Operationen durchführen:

(1) add = Addition zweier Dualzahlen ohne alten Übertrag (d.h. C_a=0),

(2) adc Addition zweier Dualzahlen mit vorhandenem alten Übertrag (d.h. C_a=1),

(3) inc = inkrementieren des ersten Operanden (zweiter Operand = 0),

(4) dec = dekrementieren des ersten Operanden (zweiter Operand = 0),

(5) sub = Subtraktion des zweiten Operanden (Subtrahend) vom ersten Operanden (Minuend),

(6) neg = Negieren des ersten Operanden (Zweierkomplement).

Hinweis: Skizzieren Sie die Abläufe, den Wert des Steuerbits für die Operation, sowie die Werte der Statusbits nach der Operation.

Bei einer vorzeichenlosen Addition (Datentyp Unsigned) kennzeichnet das Carry-Bit als neunte Stelle einen eventuell auftretenden Überlauf (z.B. 254 +2). Bei einer vorzeichenlosen Subtraktion kennzeichnet das Carry-Bit einen eventuell auftretenden Unterlauf (z.B. 2 minus 3). Da bei den Operanden kein Vorzeichen vorhanden ist, also keine negativen Operanden existieren, kann ein Überlauf bzw. Unterlauf nur bei der Zahl Null auftreten.

Zahlen mit Vorzeichen

Bei vorzeichenbehafteten Zahlen (vom Datentyp Signed) gibt es außer dem Übergang bei der Null eine weitere kritische Stelle im Zahlenformat: nämlich den Übergang von +127 zu -128. Hier liefert das Carry-Bit alleine kein ausreichendes Indiz für einen Überlauf: Die Addition von 127 + 1 führt zum Ergebnis -128 (Überlauf), wobei das neunte Bit als Carry-Bit hiervon unbeeinflusst bleibt.

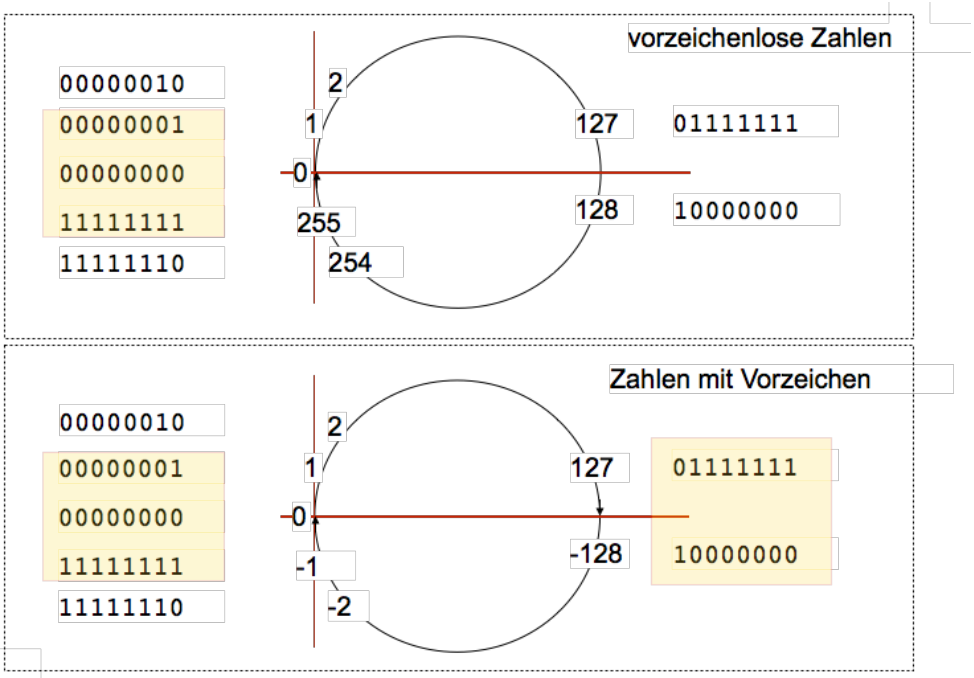


Bild 2.4 Übergänge bei Zahlenformaten mit und ohne Vorzeichen

Übung 2.3: Zu welchem Ergebnis führen die Operationen (1) 127 +1, (2) -128 -1? Liefert das Carry-Bit hierfür ein Indiz für einen Überlauf bzw. Unterlauf? Führen Sie die Operation im binären Format aus.

Übung 2.4: Durch welche Massnahmen lassen sich Überschreitungen am Übergang von 127 zu -128 erkennen? Welche Bitposition muss bei einer Zahl mit Vorzeichen hierzu beobachtet werden?

Bei Operationen mit negativen Zahlen wird das Carry-Bit zwar verändert, liefert jedoch kein zuverlässiges Indiz für einen Unterlauf bzw. Überlauf. Hierfür muss das Vorzeichen beider Operanden, sowie das Vorzeichen des Ergebnisses mit betrachtet werden. Hierbei sind mit Blick auf den Zahlenstrahl in der vorausgegangenen Abbildung folgende Fälle zu unterscheiden:

- Addition zweier positiver Zahlen: Das Ergebnis ist positiv. Im Falle eines Überlaufs an der Stelle Null liefert das Carry-Bit eine Indikation. Im Falle eines Überlaufs bei 127 wechselt das Vorzeichen des Ergebnisses in einen unerlaubten Zustand (negatives Vorzeichen).
- Addition zweier negativer Zahlen: Das Ergebnis ist negativ. Ein Überlauf an der Stelle Null kann nicht auftreten. Das Carry-Bit liefert keine Indikation für einen Überlauf. Im Falle eines Überlaufs an der Stelle -128 wechselt das Vorzeichen des Ergebnisses in einen unerlaubten Zustand (positives Vorzeichen).
- Addition einer negativen Zahl und einer positiven Zahl: in allen Kombinationen sinnvolle Ergebnisse, da die grössten darstellbaren Zahlen 127 und -128 sind. Ein Überlauf kann nicht auftreten. Das Carry-Bit liefert keine zuverlässige Indikation für einen Überlauf.

Demnach gibt es zur Erkennung von Überläufen bzw. Unterläufen am Übergang an der Stelle 127/-128 folgende Möglichkeiten: (1) Die Addition zweier positiver Zahlen zeigt ein negatives Ergebnis, (2) die Addition zweier negativer Zahlen zeigt ein positives Ergebnis.

Rechenwerk mit Statusbits

Zur Indikation von Überschreitungen des Zahlenbereichs wird daher ein weiteres Statusbit eingeführt: V (für engl. oVerflow). Ausserdem werden folgende Statusbits eingeführt: S als Vorzeichen-Bit (für engl. Sign), sowie N als Indikator für eine negatives Ergebnis (engl. Negative).

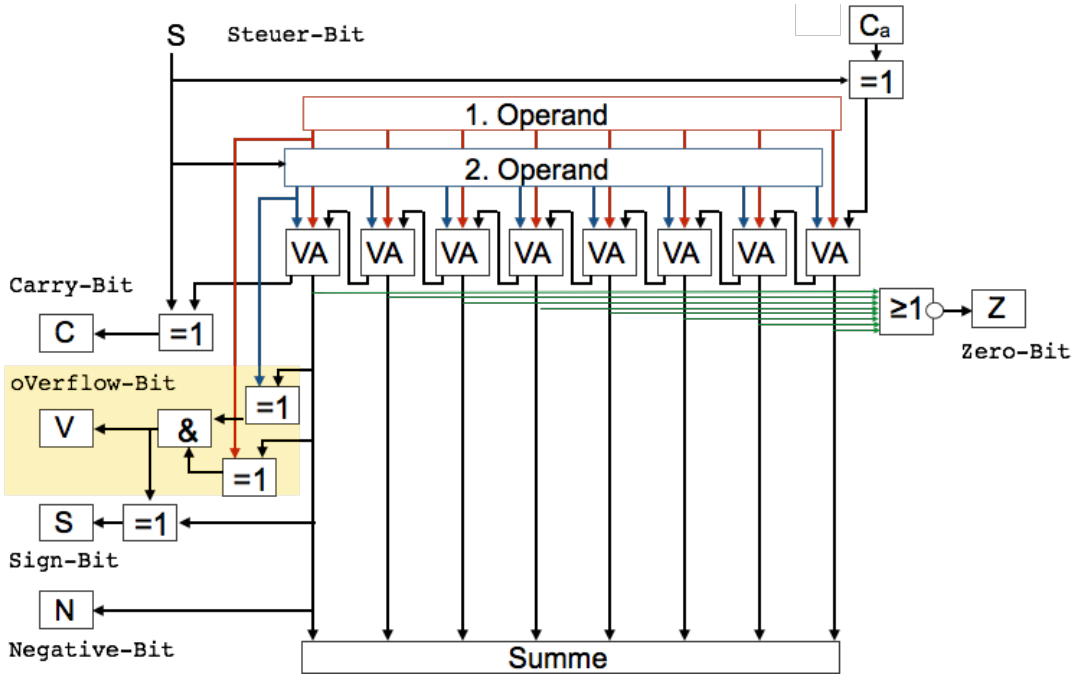


Bild 2.5 Rechenwerk mit Statusbits

Die Abbildung oben zeigt das Rechenwerk mit der Beschaltung für die Statusbits. Die Überlaufanzeige V spricht auf folgende Bedingung an: beide Operanden haben ein unterschiedliches Vorzeichen als das Ergebnis. In diesem Fall wird das Vorzeichen-Bit S negiert. Die Negativanzeige N

reflektiert lediglich das Vorzeichen des Ergebnisses, ohne Rücksicht auf eine eventuelle Überschreitung des Wertebereichs.

2.2. Arithmetisch Logische Einheit

Das im letzten Abschnitt beschriebene Rechenwerk lässt sich auch für logische Verknüpfungen zwischen den Operanden verwenden. Solche Operationen wären z.B. die Verknüpfung beider Operanden mit UND, ODER, Exklusiv-ODER, bzw. die Negation. Folgende Abbildung zeigt die Realisierung im Rechenwerk am Beispiel einer UND-Verknüpfung.

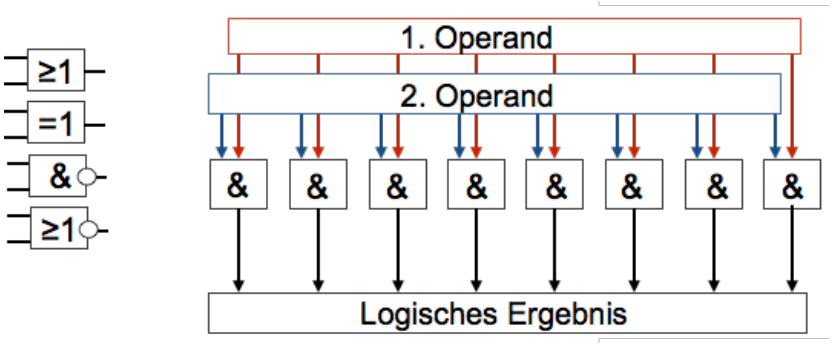


Bild 2.6 Logische Operationen im Rechenwerk

Weiterhin lassen sich auch Operationen zum Verschieben von Bits mit der Arithmetisch-Logischen Einheit realisieren. Solche Operationen sind interessant für Multiplikationen mit Vielfachen von 2 (Verschieben nach Links), bzw. Divisionen durch 2 (Verschieben nach Rechts).

Für das Rechenwerk einschließlich der logischen Operationen wird die Bezeichnung Arithmetisch-Logische Einheit verwendet (engl. ALU - Arithmetic-Logic Unit). Folgende Abbildung zeigt die Arithmetisch Logische Einheit in vereinfachter Form.

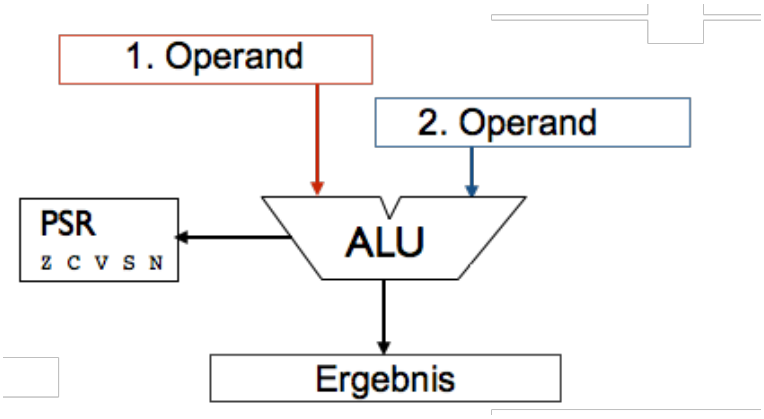


Bild 2.7 Arithmetisch Logische Einheit mit Status Register

Statt der einzelnen Bits sind nun die Operanden als Register der Wortlänge 8 Bits dargestellt, ebenso das Ergebnis der Operation. Das Steuersignal zur Subtraktion ist nicht im Detail dargestellt. Die Status-Bits zur Information über das Ergebnis Null (Zero-Bit), das Carry-Bit für einen Überlauf positiver Zahlen im Wertebereich, die Anzeige für den Überlauf einer Operation mit vorzeichenbehafteter Zahlen (V, Overflow-Bit), sowie die Anzeige für das Vorzeichen (S, Sign-Bit) und ein

negatives Ergebnis (N, Negative Bit) sind in einem Status-Register zusammengefasst (PSR für engl. Processor Status Register).

Die Arithmetisch Logische Einheit enthält das Rechenwerk für Addition und Subtraktion, das Rechenwerk für logische Operationen, sowie ggf. Schiebeoperationen. Die ALU ist als rein kombinatorische Logik ausgeführt, d.h. als Schaltnetz. Die Operanden der ALU, sowie das Ergebnis der Operation müssen in geeigneten Registern, bzw. in einem Datenspeicher bereit gestellt werden.

2.3. Rechenmaschine

Mit Hilfe der Arithmetisch Logischen Einheit soll nun eine Rechenmaschine realisiert werden. Die Rechenmaschine soll Anweisungen als Stapel verarbeiten: Nach einer Anweisung „Start“ soll sie Anweisungen für eine Operation wie z.B. Addition oder Subtraktion ausführen. Die Operanden werden ebenfalls im Stapel untergebracht. Im Fall eines mechanischen Rechenwerks wäre der Stapel beispielsweise als Stapel von Lochkarten realisierbar. Hier gehen wir davon aus, dass die Anweisungen in einem digitalen Speicher abgelegt sind, wie in Abschnitt 1 beschrieben. Es wird die in folgender Abbildung gezeigte Anordnung verwendet.

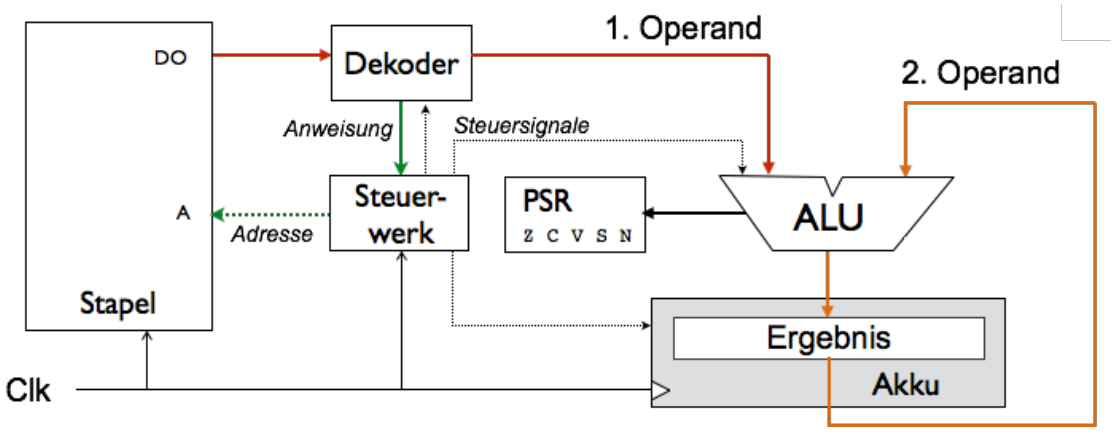


Bild 2.8 Rechenmaschine mit Stapelverarbeitung

Links in der Abbildung ist der Speicher für den Stapel der Anweisungen gezeigt. Jede Anweisung und jeder Operand ist dort als 8-Bit Wert dort gespeichert. Ein Steuerwerk adressiert den jeweils nächsten Wert. Der Stapel und das Steuerwerk sind taktgesteuert (siehe Signal Clk für engl. clock). Mit jedem Takt wird ein Wert aus dem Stapel ausgelesen. Ein als Schaltnetz ausgeführter Dekoder führt folgende Funktionen auf Anweisung des Steuerwerkes aus: (1) Im Falle eines Befehls erfolgt die Dekodierung des Befehls mit Meldung der Anweisung an das Steuerwerk, (2) im Falle eines zu verarbeitenden Wertes wird dieser als 1. Operand an das Rechenwerk weiter gegeben.

Das Rechenwerk wird ebenfalls vom Steuerwerk gesteuert. Um eine Reihe von Zahlen zu addieren bzw. zu subtrahieren, wurde die Arithmetisch Logische Einheit mit einem Register ausgestattet, das mit Akku (kurz für Akkumulator) bezeichnet ist. Dieses Register kann mit einem Takt ein neues Ergebnis der ALU aufnehmen (speichern). Am Ausgang des Registers steht das zuletzt gespeicherte Ergebnis als 2. Operand für die Verrechnung zur Verfügung. Die Arbeitsweise des Akkus entspricht den in Abschnitt 1 beschriebenen Registern.

Es soll nun folgender Stapel von Anweisungen verarbeitet werden:

```
start       ; Akku löschen, Verarbeitung beginnen
```

```
        add         ; folgenden Wert zum Akku addieren
        Wert 1      ; Akku = Akku + Wert 1
        add         ; folgenden Wert zum Akku addieren
        Wert 2      ; Akku = Akku + Wert 2
        sub         ; folgenden Wert vom Akku subtrahieren
        Wert 3      ; Akku = Akku - Wert 3
        ...         ; weitere Anweisungen und Werte
        ...
        end         ; Ende der Verarbeitung: Anhalten
```

Dieser Anweisungstext beschreibt zwar die gewünschten Operationen aus Sicht des Operateurs der Rechenmaschine, ist aber in dieser Weise nicht lesbar für die Maschine. Es muss eine Kodierung der Anweisungen vereinbart werden, z.B. start = „0001 0000“, end = „0000 0000“, add = „0010 0000“, sub = „0011 0000“. Außerdem muss das Steuerwerk wissen, auf welche Anweisungen ein Wert folgt (Anweisungen add und sub), und auf welche Anweisungen nicht (Anweisungen start und end). Diese Vereinbarungen werden implizit getroffen.

Folgender Text zeigt nun das in den Stapel der Anweisungen in kodierter, maschinenlesbarer Form: „0001 0000“, „0010 0000“, „0000 0011“, „0010 0000“, „0000 0111“, „0011 0000“, „0000 0010“, „0000 0000“. Diese Folge würde die Werte 3 und 7 addieren, sowie den Wert 2 hiervon subtrahieren. Als Ergebnis sollte die Zahl 8 im Akku verbleiben.

Programmsteuerung

Zur Steuerung des Programms dienen folgende Komponenten: (1) der Dekoder, (2) das Steuerwerk. Aufgabe des Dekoder ist es, bei einer Anweisung diese zu dekodieren, d.h. bei den im Beispiel verwendetet Anweisungen start, add, sub und end beispielsweise die ersten 4 Bits, in denen die Anweisungen kodiert sind, an das Steuerwerk zu geben. Außerdem schaltet der Dekoder im Falle eines Wertes die 8 Bits aus dem Stapel als 1. Operanden an die ALU weiter.

Das Steuerwerk ist für die Ablaufsteuerung verantwortlich. Es hat folgende Aufgaben:

- Aus dem Stapel mit der Adresse A die nächste Anweisung bzw. den nächsten Wert auszulesen. Hierzu verwendet das Steuerwerk einen Zähler (Programmzähler).
- Den Dekoder umzuschalten für (1) die Dekodierung einer Anweisung, (2) die Weitergabe eines Wertes als 1. Operanden an die ALU.
- Die ALU zu steuern: (1) Für den Befehl add ist das Steuerbit S=0, (2) für den Befehl sub ist das Steuerbit S=1. Außerdem gibt das Steuerwerk mit Hilfe eines Enable-Signals den Takt für das Akku-Register frei, so dass das Ergebnis der Operation dort gespeichert wird und als 2. Operand für den folgenden Befehl zur Verfügung steht.

Der Dekoder kann als Schaltnetz realisiert werden, d.h. als rein kombinatorische Logik, enthält jedoch ein Register für den 1. Operanden. Der Dekoder wird durch das Steuerwerk im Takt geschaltet. Das Steuerwerk arbeitet als Schaltwerk, reagiert also auf den Systemtakt. Insgesamt ergibt sich für das im Beispiel genannte Programm der in folgender Ablauf:

- Takt 1: Adresse 0 an den Stapel geben
- Takt 2: Adresse 1 an den Stapel geben; Stapel: Befehl start ausgeben; Akku: löschen
- Takt 3: Adresse 2 an den Stapel geben; Stapel: Befehl add ausgeben;
- Takt 4: Adresse 3 an Stapel geben; Stapel: Wert 1 ausgeben; Steuerwerk: add Signal an Akku geben
- Takt 5: Adresse 4 an Stapel geben; Stapel: Befehl add ausgeben; Akku: Ergebnis speichern (Akku: Wert 1)

- Takt 6: Adresse 5 an Stapel geben; Stapel: Wert 2 ausgeben, Steuerwerk: add Signal an Akku geben
- Takt 7: Adresse 6 an Stapel geben; Stapel Befehl sub ausgeben; Akku: Ergebnis speichern (Akku: Wert 1 + Wert 2);
- Takt 8: Adresse 7 an Stapel geben; Stapel: Wert 3 ausgeben; Steuerwerk: sub Signal an Akku geben;
- Takt 9: Adresse 8 an Stapel geben; Stapel: Befehl end ausgeben; Akku: Ergebnis speichern (Akku: Wert 1 + Wert 2 - Wert 3);
- Takt 10: Adresse anhalten; Stapel: anhalten; Steuerwerk: anhalten.

Durch die taktabhängigen Vorgänge ergibt sich eine Fließbandverarbeitung (engl. pipeline) an den Stationen (1) Stapel, (2) Steuerwerk, (3) Akku-Register. Alle Stationen sind parallel in Betrieb und arbeiten in der angegebenen Reihenfolge aufeinander zu. Folgende Abbildung illustriert den Prozess der Verarbeitung.

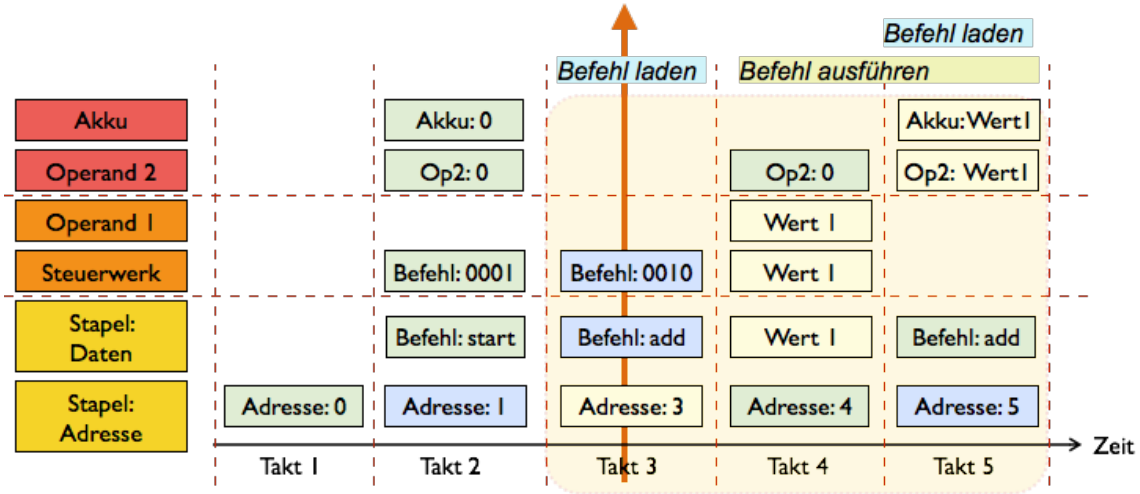


Bild 2.9 Ablauf des Programms

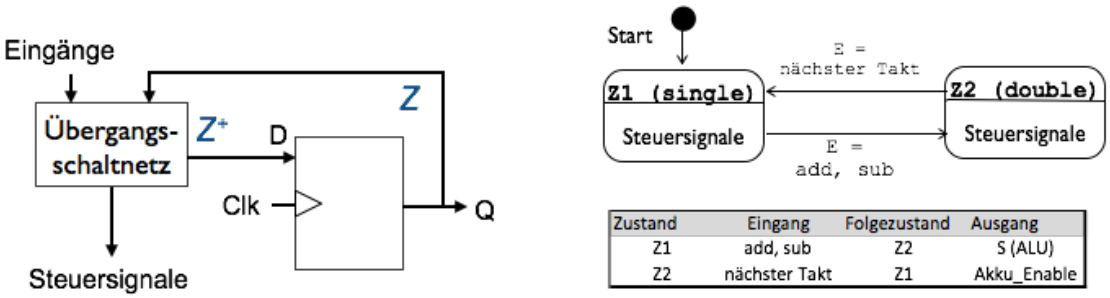
Wegen der Fließbandverarbeitung ist in Takt 1 zunächst nur der Stapel in Aktion. In Takt 2 steht der erste Befehl aus dem Stapel bereit und kann dekodiert und verarbeitet werden. In Takt 3 steht der folgende Befehl zur Verfügung, kann jedoch noch nicht ausgeführt werden, da hierzu ein Wert aus dem Stapel benötigt wird. Dieser Wert steht dann in Takt 4 zur Verfügung. In Takt 5 kann der Akku schliesslich das Ergebnis in seinem Register speichern. Wegen der Fliessbandverarbeitung eilt die Stapeladresse (bzw. der Programmzähler) der aktuellen Verarbeitung immer um 2 Takte voraus.

Ebenfalls in der Abbildung erkennbar sind (1) die Überlappung des Ladens des folgenden Befehls mit der Ausführung des aktuellen Befehls am Fliessband, (2) die Tatsache, dass die Ausführung der Befehle add und sub 2 Takte benötigt, da hierzu noch ein Wert aus dem Stapel geladen werden muss. Die Befehle start und end sind mit einem Takt ausführbar. Das Steuerwerk muss also eine Unterscheidung treffen zwischen Befehlen, die mit einem Takt ausführbar sind, und Befehlen, die zwei Takte benötigen.

Zustandsautomat

Zur unterschiedlichen Behandlung eintaktiger und zweitaktiger Befehle wird das Steuerwerk als Zustandsautomat ausgeführt. Hierunter ist zu verstehen, dass das Steuerwerk unterschiedlich

reagiert, wenn es sich im Zustand 1 (ein Takt pro Befehl) und im Zustand 2 (zwei Takte pro Befehl) befinden. Die Zustände des Automaten mit ihren Übergängen lassen sich in einem Diagramm darstellen, wie in folgender Abbildung gezeigt.



Zustand	Eingang	Folgezustand	Ausgang
Z1	add, sub	Z2	S (ALU)
Z2	nächster Takt	Z1	Akku_Enable

Bild 2.10 Steuerwerk als Zustandsautomat

Auf der linken Seite ist die Realisierung des Automaten dargestellt. Ähnlich dem Zähler aus Abschnitt 1 dient ein Register zur Speicherung des aktuellen Zustands Z. Aus diesem aktuellen Zustand Z leitet sich der Folgezustand Z^+ ab, und zwar in Abhängigkeit der Eingangssignale des Automaten. Da hier nur zwei Zustände unterschieden werden sollen, genügt als Zustandsspeicher ein Register mit einem Bit. Die Berechnung des Folgezustands erfolgt durch das Übergangsschaltnetz in der Abbildung links, also durch eine kombinatorische Logik.

Diese Logik muss nun noch definiert werden. Hierzu dient das Diagramm auf der rechten Seite oben in der Abbildung. Der Automat soll bei Bedarf zwischen zwei Zuständen wechseln können: Der Zustand Z1 kennzeichnet hierbei einen eintaktigen Befehl, der Zustand Z2 einen zweitaktigen Befehl. Initialzustand ist Z1. Beim Start des Automaten befindet sich dieser in Z1. Beim Dekodieren des Befehls start verbleibt der Automat ebenfalls in diesem Zustand. Nur beim Dekodieren der zweitaktigen Befehle add bzw. sub wechselt der Automat beim nächsten Takt in den Zustand Z2. Dort soll er genau einen Takt bleiben und mit dem folgenden Takt wieder in den Zustand Z1 zurück kehren.

Die Aktionen, die der Automat im jeweiligen Zustand ausführt, d.h. die Steuersignale, die er jeweils schaltet, sind abhängig vom jeweiligen Zustand, sowie vom jeweiligen Befehl. In beiden Zuständen wird der Zähler für die folgende Adresse im Stapel erhöht. Beim Befehl start erfolgt das Löschen des Akkus. Beim Befehl add wird das Steuerbit des Akkus gesetzt, der Automat wechselt mit dem Takt in den Folgezustand Z2.

Mit diesem Takt wird der zu addierende Wert bereit gestellt, der Automat setzt das Steuersignal Akku_Enable, damit der Akku mit dem folgenden Takt das Ergebnis aufnehmen kann. Mit dem gleichen Takt wechselt der Automat wieder in des Zustand Z1. Für den Entwurf der Logik des Übergangsschaltnetzes wurde das Zustandsdiagramm rechts oben in der Abbildung in eine Tabelle übersetzt, die darunter dargestellt ist (die sogenannte Zustandsübergangstabelle).

Übung 2.5: Erstellen Sie ein Programm (einen Stapel von Anweisungen, vergleichbar Assembler-Sprache) zur Addition bzw. Subtraktion einer Reihe von Zahlen. Kodieren Sie das Programm in Maschinensprache. Führen Sie das Programm manuell in einer Tabelle aus. Verfolgen Sie die Fliessbandverarbeitung und die Zustandswechsel des Automaten.

Übung 2.6: Verändert Sie die Rechenmaschine so, dass sie die Anzahl der Werte in einem Programm zählt (zwischen der Start-Anweisung und der Stopp-Anweisung). Hinweis: Verwenden Sie die Operation der ALU zum Inkrementieren.

Übung 2.7: Erstellen Sie ein Programm, das die Verarbeitung bei Überschreitung des Wertebereichs stoppt. Hinweis: Verwenden Sie das Status-Register und erweitern Sie die Anweisungen in geeigneter Weise.

Übung 2.8: Erstellen Sie ein Programm, das von zwei vorgegebenen Werten A und B das Maximum ermittelt. Welche Erweiterungen der Rechenmaschine sind hierfür erforderlich? Führen Sie die benötigten Befehle als Anweisungen (Assembler-Anweisungen) ein.

3. Mikrocontroller

Die im letzten Abschnitt beschriebene Rechenmaschine enthält bereits wesentliche Bestandteile eines Mikrocontrollers, ist jedoch in ihrer Funktion noch sehr eingeschränkt: Für einen Programmablauf in Abhängigkeit der Ergebnisse fehlen Möglichkeiten für Verzweigungen, also die Möglichkeit der Abfrage von Bedingungen in Kombination mit Sprungbefehlen. Außerdem fehlt die Möglichkeit zur Speicherung von Ergebnissen außerhalb des Akkus.

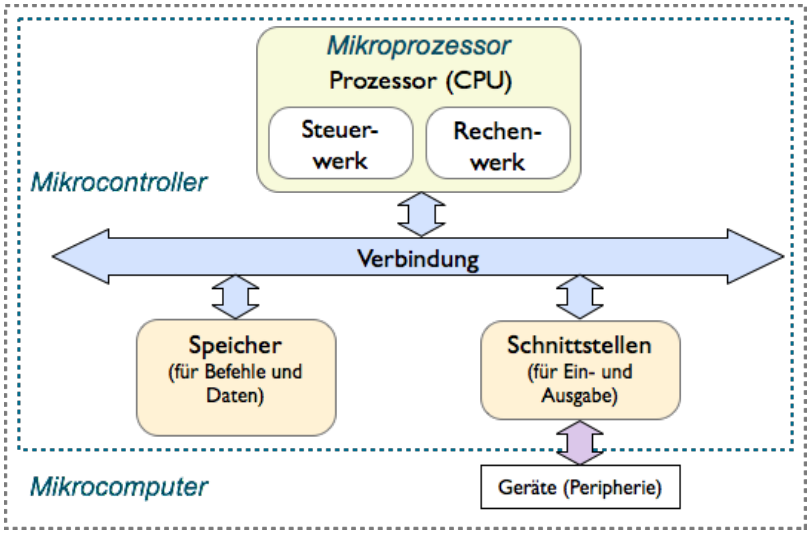


Bild 3.1 Begriffe: Mikroprozessor, Mikrocontroller und Mikrocomputer

Für einen Mikrocontroller charakteristisch ist die Programmierbarkeit, der Speicher für Programm und Daten, sowie Schnittstellen zum Anschluss externer Geräte. Die Abbildung oben zeigt die grundsätzliche Architektur. Wesentlicher Bestandteil des Mikrocontrollers ist der Mikroprozessor, bestehend aus Rechenwerk und Steuerwerk. Zu einem Computer bzw. Mikrocomputer gehören dann noch Geräte zur Eingabe (Tasten, Tatstur, Maus, berührungsempfindliche Anzeigte, ...) und Ausgabe (LEDs, Anzeige, Bildschirm, ...), sowie Schnittstellen zur Vernetzung mit anderen Systemen.

3.1. Prozessorarchitektur

Unter der Architektur eines Prozessors oder Mikrocontrollers versteht man alle Komponenten und Abstraktionen, die zu seiner Programmierung benötigt werden. Hierzu gehört speziell der Befehlssatz und die Möglichkeiten zur Übergabe und Speicherung der Operanden. Ein einfacher Mikropro-

zessor ist in Akkumulator-Architektur aufgebaut. In dieser Architektur funktioniert der Akkumulator (kurz Akku) als einziges Register, wie bei der Rechenmaschine aus dem letzten Abschnitt. Operanden werden direkt aus dem Datenspeicher entnommen und mit dem Inhalt des Akkus verrechnet. Ergebnisse finden sich im Akku wieder und können von dort aus weiter verarbeitet werden, bzw. in den Datenspeicher geschrieben werden.

Folgende Abbildung zeigt den grundsätzlichen Aufbau des Mikrocontrollers bestehen aus dem Prozessor, dem Programmspeicher und dem Datenspeicher. Bei Mikrocontrollern ist es üblich, dien letztgenannten Komponenten direkt auf dem Chip unterzubringen. In diesem Fall verfügt der Controller über einen vom Datenbus getrennten Programmbus. Mikroprozessoren für den Desktop-Bereich bzw. für Server verfügen über externe Speicher. Um Anschlussleitungen zu sparen, ist hier in der Regel der Programmspeicher mit dem Arbeitsspeicher kombiniert und verfügt über einen gemeinsamen Adressbus.

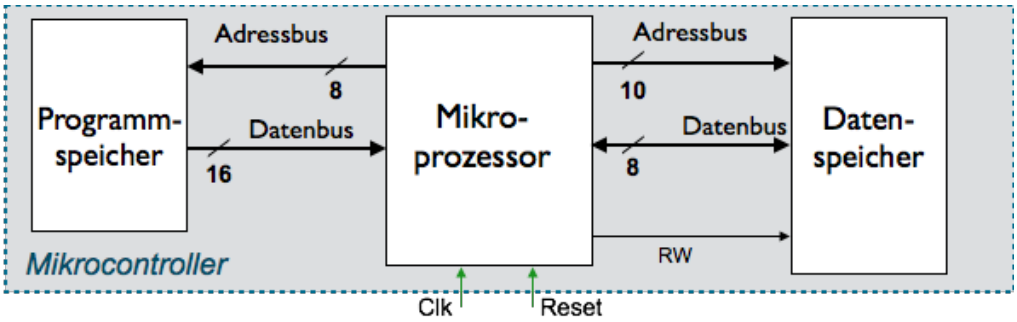


Bild 3.2 Grundsätzlicher Aufbau des Mikrocontrollers

Zur Prozessorarchitektur gehören die Wortbreiten für Programm und Daten, sowie die hierfür gewählten Breiten der Adressbusse. Im Beispiel verwendet der Prozessor 16-Bit breite Befehle, und 8 Bit breite Daten. Der Programmspeicher wird mit einem 8-Bit breiten Adressbus adressiert, ausreichend für kleine Programme mit bis zu 256 Worten. Der Datenspeicher wird mit 10 Bit adressiert, ausreichend für 1024 Bytes an Daten.

Befehlssatz

Die folgende Abbildung zeigt den Befehlssatz des Prozessors.

Befehlssatz			Beschreibung	Prozessor Status Register (PSR) Flags				
Assembler	Kürzel	Opcode		Zero (Z)	Carry (C)	oVerflow (V)	Sign (S)	Neg. (N)
nop	NOP	0	No Operation	-	-	-	-	-
lsl	LSL	1	Akku: C <- Akku <- 0, logic. shift left	x	x	x	x	x
lsr	LSR	2	Akku: 0 -> Akku -> C, logic. shift right	x	x	x	x	x
ldi K	LDI	3	Akku<=K, load immediate 8bit constant	-	-	-	-	-
ld Addr	LDA	4	Akku<=(Addr), load Akku from memory	-	-	-	-	-
st Addr	STR	5	(Addr)<=Akku, store Akku to memory	-	-	-	-	-
add Addr	ADD	6	Akku<=Akku+(Addr), add	x	x	x	x	x
sub Addr	SUB	7	Akku<=Akku-(Addr), subtract	x	x	x	x	x
and Addr	ANDA	8	Akku<=Akku AND (Addr), logical AND	x	-	0	x	x
eor Addr	EOR	9	Akku<=Akku XOR (Addr), logical XOR	x	-	0	x	x
or Addr	ORA	A	Akku<=Akku OR (Addr), logical OR	x	-	0	x	x
jmp K	JMP	B	jump to address constant (K)	-	-	-	-	-
brbc bit, offset	BRBC	C	branch if bit clear (C, Z)	-	-	-	-	-
brbs bit, offset	BRBS	D	branch if bit set (C, Z)	-	-	-	-	-

Bild 3.3 Befehlssatz des Mikroprozessors

Die Tabelle in der Abbildung ist wie folgt aufgebaut: Neben dem Befehl in Assembler-Sprache (z.B. ldi K) findet sich ein Kürzel, das z.B. in Zeitdiagrammen verwendet wird. Daneben ist in hexadezimaler Schreibweise der Befehlscode aufgeführt (Opcode kurz für engl. Operation Code). Die Kodierung wird nur für die Implementierung in die Maschinensprache verwendet und spielt für die Programmierung in Assembler keine Rolle.

Assembler Sprache

Daneben findet sich eine kurze Beschreibung der Befehle (z.B. für ldi K: Akku<= K, d.h. der Akku übernimmt den Wert der Konstanten K). Die folgenden Spalten zeigen die Bits des Prozessor-Status-Registers. Ein Kreuz an einer der Bits bedeutet, dass der Befehl einen Einfluss auf das besagte Bit hat, d.h. dieses Bit je nach Ergebnis der Operation verändert. Der Befehl ldi K hat als reine Ladeoperation keinen Einfluss auf die Statusbits, ebenso wie die anderen Befehle, die nur Daten verschieben, bzw. die Sprungbefehle. Einfluss auf die Statusbits haben hingegen die arithmetischen und logischen Operationen.

Die durch den Befehlssatz und durch den grundsätzlichen Aufbau gegebene Prozessorarchitektur ist bereits ausreichend, um Assembler-Programme zu schreiben. Eine genauere Kenntnis der Implementierung ist hierfür nicht erforderlich. Mit Hilfe der Befehle soll ein Programm erstellt werden, das zwei Zahlen addiert und das Ergebnis im Datenspeicher hinterlegt.

```
; ASL (Assembler Language) für den DHBW MCT-Mikroprozessor
; Einfaches Testprogramm: c = a + b
; mit den Werten a = 5 und b = 3

start:      ldi  5          ; Akku <= A (mit Wert A = 5)
            st   $0         ; DS(0) <= Akku (Wert A=5 abspeichern)
            ldi  3          ; Akku <= B (mit Wert B = 3)
            add  $0         ; Akku <= Akku + DS(0)  (A + B)
end:        st   $1         ; DS(1)<= Akku
```

Der eigentliche Programmtext ist hier in der mittleren Spalte zu sehen. Die Markierungen (Labels) „start“ und „end“ dienen nur der Übersicht. Der mit einem Strichpunkt abgesetzte Text in der Spalte rechts ist als Kommentar zu lesen und soll die Programmschritte erläutern. Im folgenden wird nun untersucht, wie das Programm innerhalb des Prozessors abläuft. Zu diesem Blick auf das Innenleben muss man sich allerdings auf eine mögliche Implementierung der Prozessorarchitektur festlegen.

Ein wesentliches Merkmal der gewählten Prozessorarchitektur ist die Verwendung eines Akkumulators (kurz Akku) als einzigen Register. Das Rechenwerk funktioniert also genauso, wie in der im letzten Abschnitt gezeigten Rechenmaschine (siehe Abbildung 2.8). Der Akku dient zur Aufbewahrung eines Operanden als Quelle und als Ergebnis der Rechenoperation. In der durch den Befehlssatz hier beschriebenen Prozessorarchitektur wird der zweite Operand jedoch aus einem Datenspeicher geladen: der Befehl add $0 addiert beispielsweise den Inhalt des Akkus zum Inhalt der Speicherzelle DS(0). Mit dem Kürzel $0 wird also die Speicherzelle mit der Adresse 0 bezeichnet.

Der Akku selbst kann mit einer Konstanten aus dem Programmspeicher geladen werden (siehe z.B. ldi 5), bzw. mit einem Ladebefehl aus dem Datenspeicher (im Befehlssatz siehe lda $0). Ebenso kann der Inhalt des Akkus in den Datenspeicher geschrieben werden (siehe z.B. str $1). Die

Verbindung zwischen dem Datenspeicher und dem Rechenwerk wird somit über den Akku hergestellt. Die Adressierung des Datenspeichers erfolgt hierbei direkt aus dem Assembler-Kode heraus.

Übung 3.1: Erstellen Sie mit dem gegebenen Befehlssatz ein Assembler-Programm, das eine vorgegebene Zahl A (kleiner als 64) in die Speicherzelle 0 schreibt. Anschliessend soll folgende Berechnung ausgeführt werden: B = 16 * A + A. Das Ergebnis soll in Speicherzelle 2 geschrieben werden. Hinweis: Verwenden Sie eine Schiebeoperation für die Multiplikation.

3.2. Implementierung der Prozessorarchitektur

Die Assembler-Sprache dient zwar der maschinennahen Programmierung, kann aber von der Maschine nur in bitkodierter Form ausgeführt werden. Hierzu ist zu definieren, wie die einzelnen Befehle zu kodieren sind. Hierbei gelten die bereits getroffenen Vorgaben über

- die Wortlänge der Befehle (16 Bits)
- die Wortlänge der Daten (8 Bits)
- die Adressbreiten des Befehlsspeichers (8 Bit) und Datenspeichers (10 Bit).

Kodierung der Befehle

Darüber hinaus muss definiert werden, wie viele unterschiedliche Befehle der Prozessor ausführen soll. Hierfür wird eine Zahl von 32 unterschiedlichen Befehlen vorgegeben. Somit werden für die Kodierung jeden Befehls 5 Bits benötigt. Von den 16 Bits eines Befehls verbleiben somit 11 Bits, die für Operanden bzw. Adressen verwendet werden können. Folgende Abbildung zeigt eine Vorgabe für die Befehlsformate.

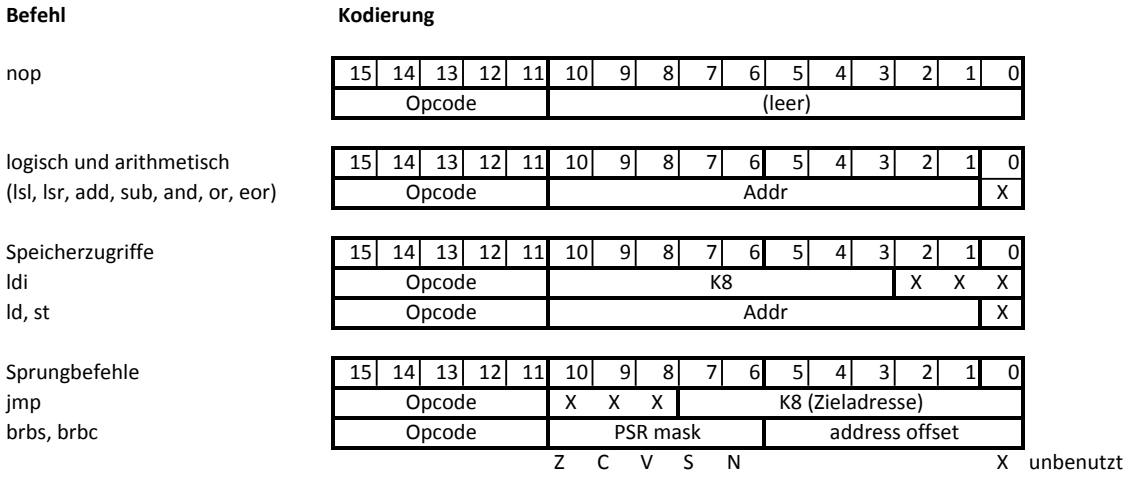


Bild 3.4 Befehlsformate des Prozessors

Der einfachste Befehl führt überhaupt keine Funktion aus (NOP für engl. No Operation) und hat nur zur Folge, dass der nächste Befehl geladen wird. Die übrigen Befehle wurden nach ihrer Funktion bzw. nach ihrem Befehlsformat gruppiert.

Die größte Gruppe ist hierbei die der logischen und arithmetischen Operationen. Bei allen diesen Befehlen befindet sich ein Operand bereits im Akku, der zweite Operand im Datenspeicher. Im Assembler-Befehl wird die Adresse dieses Operanden im Speicher angegeben. Diese Adresse Addr

folgt nun im Befehlsformat dem Befehlscode (Opcode). Da der Datenbus in der gewählten Architektur 10 Bits breit ist, genügen als Adresse die auf den Opcode folgenden 10 Bits (vom 10 bis 1).

Der Befehl zum Laden des Akkus aus dem Datenspeicher, sowie der Befehl zum Speichern des Inhalts des Akkus in den Datenspeicher sind ebenso kodiert: Auf den Opcode folgt die Adresse der gewünschten Speicherzelle. Beim Befehl zum direkten Laden des Akkus LDI folgt auf den Befehlscode die 8-Bit Konstante K. Die übrigen Bits werden nicht benötigt.

Bei den Sprungbefehlen ist zu unterscheiden zwischen dem unbedingten Sprung JMP und den Sprüngen mit Bedingung. Beim unbedingten Sprung folgt auf den Befehlscode die Zieladresse. Da der Adressbus des Programmspeichers mit 8 Bit vorgegeben wurde, genügt für eine absolute Sprungadresse eine 8-Bit Konstante. Dass die Sprungadresse absolut vorgegeben wurde, ist hierbei eine willkürliche Festlegung. Eine Alternative wäre eine Adresse relativ zur Adresse des aktuellen Befehls im Programmspeicher.

Die bedingten Sprungbefehle BRBS (für engl. branch if bit set) und BRBC (für engl. branch if bit clear) führen nur zu einem Sprung (bzw. zu einer Verzweigung), wenn die genannte Bedingung erfüllt ist. Andernfalls erfolgt keine weitere Aktion und es wird der nächste Befehl in der Reihe ausgeführt. Als Bedingung gilt, ob ein bestimmtes Bit im Prozessor-Status-Register gesetzt ist (BRBS), bzw. nicht gesetzt ist (BRBC). Welches Bit hierfür betrachtet werden soll, ist im Feld PSR des Befehls angegeben. Die Auswahl erfolgt bei der hier gewählten Kodierung einfach durch eine Maske über die 5 Bits des Prozessor-Status-Registers. Mit dem Kürzel PSR Mask im Befehlsformat ist diese Maske gemeint (d.h. durch eine 1 an einer der Stellen wird das betreffende Bit ausgewählt).

Die Vorgaben für die Befehlsformate folgen zwar der Prozessorarchitektur, allerdings haben diese Festlegungen im Detail auf die Prozessorarchitektur überhaupt keinen Einfluss. Auf diese Art kann ein in Assembler geschriebenes Programm auch auf einer unterschiedlichen Implementierung des Prozessors funktionieren, der als Variante ein Mitglied der Prozessorfamilie dar stellt. Das Assembler-Programm muss dann auf die unterschiedliche Kodierung des Prozessors übersetzt werden.

Maschinensprache

Folgender Text zeigt die Kodierung des Assembler-Programms mit den in der Abbildung oben gezeigten Befehlsformaten. Im Unterschied zur Assembler-Sprache ist dieses Programm nun direkt von der Maschine ausführbar (es ist in Maschinensprache kodiert).

```
-- Maschinensprache für den DHBW MCT-Mikroprozessor
-- Einfaches Testprogramm: c = a + b
-- mit den Werten a = 5 und b = 3
-- aus dem Assembler-Programm manuell übersetzt

0 => "0001100000101000",        -- LDI 5
1 => "0010100000000000",        -- STR $0
2 => "0001100000011000",        -- LDI 3
3 => "0011000000000000",        -- ADD $0
4 => "0010100000000010",        -- STR $1
```

Auch dieser Programmtext enthält noch Textkommentare (hier mit „--“ eingeleitet), sowie die Zeilennummern für die Ablage der Befehle im Programmspeicher am Anfang jeder Zeile (z.B. „0 =>“)

gefolgt von einem 16Bit-Wert in doppelten Hochkommas). In den Programmspeicher werden nur die 16-Bit Werte geladen. In den ersten 5 Bits erkennt man die Befehlscodes (Opcodes) der Befehle aus Abbildung 3.2 (z.B. 00011 für hexadezimal 3 bzw. den Befehl LDI). In den folgenden Bits sind die Konstanten bzw. die Adressen enthalten, wie in Abbildung 3.4 vereinbart.

Übung 3.2: Übersetzen Sie Ihr Programm aus Übung 3.1 in Maschinensprache. Hinweis: Verwenden Sie ggf. ein Tabellenkalkulationsprogramm der Übersicht halber.

Prozessor

Die folgende Abbildung zeigt das Innenleben des Mikrocontrollers als Blockschaltbild. Der Prozessor besteht aus einem Steuerwerk und einem Rechenwerk. Die übrigen Komponenten sind der Programmspeicher und der Datenspeicher. Wie bereits im Befehlssatz festgelegt, verfügt das Rechenwerk über eine Akku-Architektur. Der Akku als einziges Register ist die Brücke zum Datenspeicher. Das Rechenwerk entspricht somit ganz der im letzten Abschnitt bereits diskutierten Rechenmaschine. Allerdings wird das Rechenwerk nun mit Daten aus dem Datenspeicher versorgt.

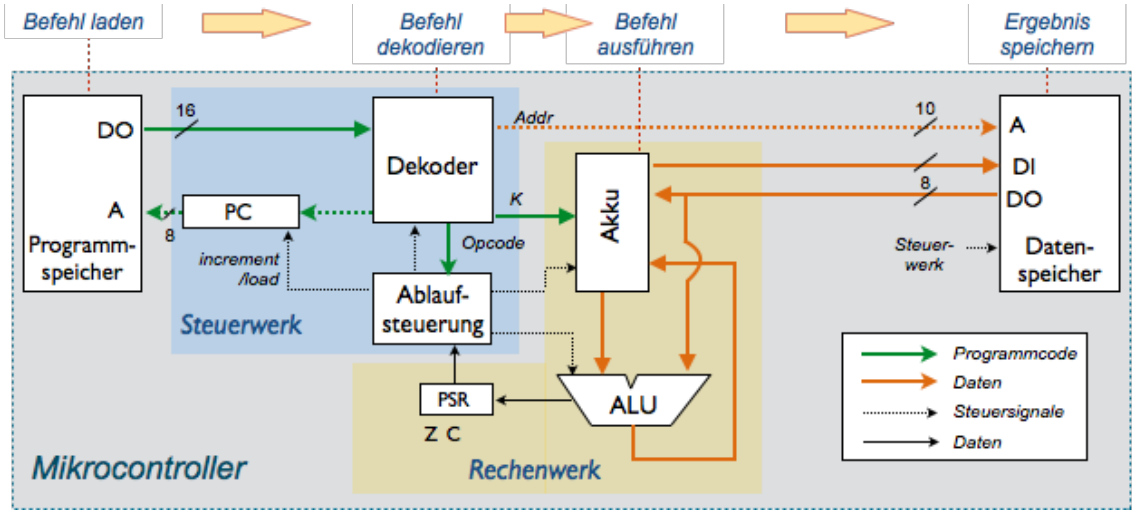


Bild 3.5 Prozessor mit Akkumulator-Architektur

Bei der im Befehlssatz verwendeten direkten Adressierung wird die Adresse des jeweiligen Operanden aus dem Datenspeicher direkt im Befehl codiert. Aus diesem Grund entspringt im Blockschaltbild die Adresse des Datenspeichers dem Dekoder, der die Adresse dem Befehlscode entnimmt. Ausgang und Eingang des Datenspeichers sind direkt mit dem Akku verbunden. Die Auswahl übernimmt der Akku in Abhängigkeit des jeweils vorliegenden Befehls. Das direkte Laden des Akkus mit einer Konstanten im Befehl ist in der Abbildung als Verbindung zwischen Dekoder und Akku dargestellt.

Zum Steuerwerk gehören neben der bereits bekannten Ablaufsteuerung und dem Befehls-Dekoder der Programmzähler (PC für engl. Program Counter). Wie der Name bereits ausdrückt, ist dieser Zeiger in den Programmspeicher als Zähler ausgeführt, der den jeweils folgenden Befehl adressiert. Im Falle eines Sprungbefehls wird der Programmzähler von der Ablaufsteuerung neu geladen.

Die Ablaufsteuerung verfügt nun über Steuerleitungen zu allen anderen Komponenten. Der grundsätzliche Ablauf ist hierbei wie folgt: (1) der Dekoder übergibt den Steuerwerk den Befehlscode

des aktuellen Befehls. Ja nach Befehl stellt der Dekoder auch die Adresse in den Datenspeicher bzw. die in den Akku zu ladende Konstante zur Verfügung. (2) das Steuerwerk bedient in Abhängigkeit des Befehlscodes die Steuerleitungen zu den Komponenten. Rechenwerk und Steuerwerk arbeiten taktgesteuert. Wie die Abbildung zeigt, laufen bzgl. des Kontrollflusses alle Fäden in der Ablaufsteuerung zusammen. Mit dem Datenpfad hat die Ablaufsteuerung hingegen nichts zu tun.

Ablauf des Programms

Folgende Abbildung zeigt den Programmablauf. Dargestellt sind neben dem Takt (Clk) und einem Reset-Signal der Programmzähler (PC) als Adresse am Programmspeicher, sowie der Datenausgang des Programmspeichers (PS-DO). Man erkennt dass mit steigenden Taktflanken die Adresse des Programmspeichers hochgezählt wird. Mit jeder folgenden Taktflanke gibt der Programmspeicher den zugehörigen Befehl aus (unter PS-DO als 16-Bit Wort in hexadezimaler Schreibweise notiert).

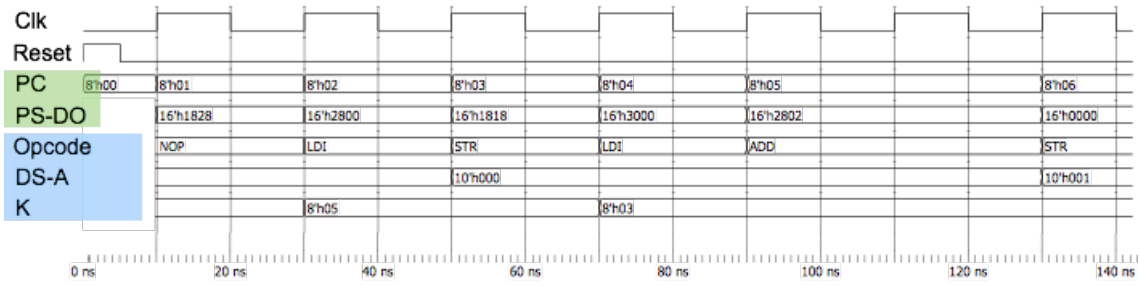


Bild 3.6 Programmablauf am Steuerwerk

Darunter dargestellt sind drei Ausgänge des Dekoders: (1) der Befehlscode (Opcode), den der Dekoder aus dem 16-Bit Programmwort entnimmt, (2) ggf. eine Adresse für den Datenspeicher (DS-A), sowie (3) ggf. eine Konstante K zum Laden des Akkus. In der Zeile Opcode erkennt man den Ablauf des vorher beschriebenen Testprogramms (wobei durch den Reset als initialer Wert am Ausgang des Dekoders der Befehl NOP vorliegt):

- LDI K=5
- STR $0
- LDI K=3
- ADD $0
- STR $1.

Man kann dem Verlauf des Programms unmittelbar folgen. Etwas irritierend ist möglicherweise der Vorlauf des Programmzählers gegenüber dem aktuell dekodierten Opcode: Der Zähler ist jeweils 2 Takte voraus. Der erste Takt Verzögerung kommt durch den Programmspeicher zustande, wie die Zeile unmittelbar unter dem Programmzähler zeigt. Einen weiteren Takt benötigt der Dekoder zum Dekodieren des Befehls. Zum Zeitpunkt n eilt der Befehlszähler also bereits 2 Schritte vor.

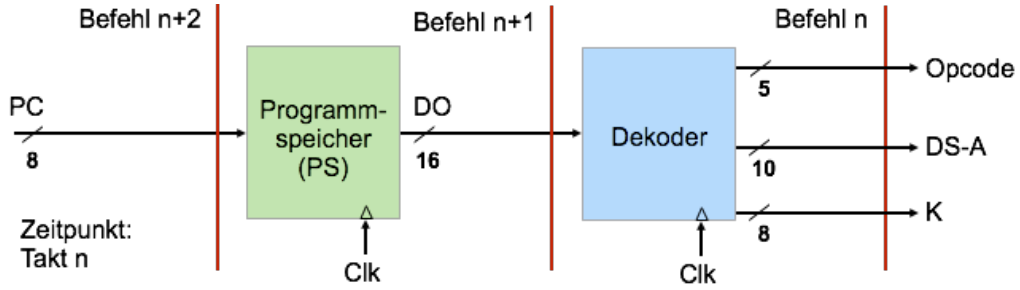


Bild 3.7 Programmspeicher und Dekoder als Schieberegister

Im Grunde genommen entspricht die Anordnung einem zweistufigen Schieberegister, wie in der Abbildung oben gezeigt. Wenn am Ausgang des Dekoders der erste Befehl LDI K=5 von Adresse 0 dekodiert ist, zeigt der Befehls-zähler (PC) bereits auf die Adresse 2. Der Befehl am Adresse 1 wurde bereits ausgelesen und wird noch am Ausgang des Programmspeichers vorgehalten, bis er mit der nächsten Taktflanke in den Dekoder übernommen wird. Die Reihenfolge der Befehle wird hierdurch nicht geändert. Sobald die Kette einmal angelaufen ist, folgt auch mit jedem Takt ein neuer Befehl.

Folgende Abbildung zeigt nun den Ablauf des Programms zusammen mit dem Rechenwerk und dem Datenspeicher. Außer dem bereits bekannten Ablauf um den Programmspeicher und Dekoder sind die Ausgänge der ALU und des Akkus dargestellt, sowie die Adressleitung des Datenspeichers (DS-A), der Eingang des Datenspeichers (DS-DI), der Ausgang des Datenspeichers (DS-DO), sowie die ersten beiden Speicherzellen DS(0) und DS(1).

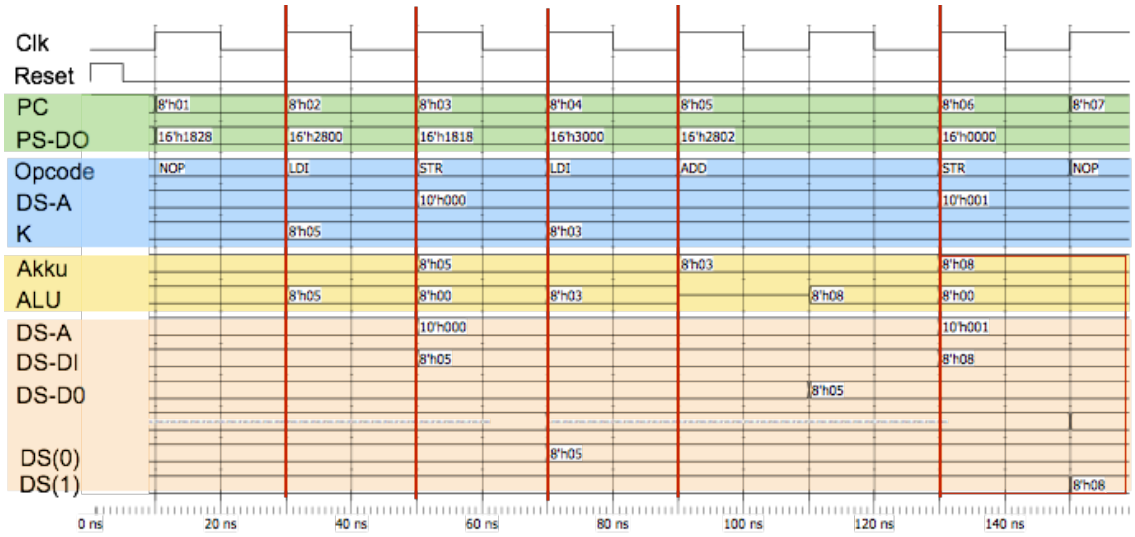


Bild 3.8 Programmablauf mit Rechenwerk und Datenspeicher

Der prinzipieller Ablauf des Programms ist wie folgt:

- LDI K=5: Die Konstante J liegt mit dem Opcode am Dekoder an, somit nach der Schaltung im Blockschaltbild (siehe Abbildung 3.5) auch am Eingang des Rechenwerks (ALU in der Abbildung oben). Im folgenden Takt wird der Wert dann in den Akku übernommen. In diesem folgenden Takt wurde bereits der nächste Befehl dekodiert.

- STR $0: Mit dem Opcode liegt am Ausgang des Dekoders auch die Zieladresse DS-A im Datenspeicher zur Aufnahme des Akku-Inhalts (10‘h000 in der Abbildung). Mit dem folgenden Takt wird der Inhalt des Akkus übernommen, wie der Inhalt der Speicherzelle DS(0) zeigt. Mit diesem Takt wurde bereits der nächste Befehl dekodiert.

- LDI K=3: Der Akku wird mit der Konstante K=3 geladen. Ablauf wie im ersten Befehl.

- ADD $0: Ein Operand (K=3) befindet sich bereits im Akku, der zweite muss erst aus dem Datenspeicher geladen werden. Mit dem Opcode liegt am Ausgang des Decoders bereits die Adresse der gewünschten Speicherzelle (DS-A wiederum 10‘h000). Es benötigt jedoch einen weiteren Takt, um des zugehörigen Wert aus dem Datenspeicher zu laden. Der Wert findet sich dann auf der Leitung DS-DO, die zum Eingang für den zweiten Operanden der ALU führt (siehe Signal ALU am Eingang des Rechenwerkes). Die ALU als reines Schaltnetz stellt das Ergebnis unmittelbar zur Verfügung. Das Laden des Ergebnisses in den Akku benötigt jedoch einen weiteren Takt. Insgesamt benötigt der Befehl ADD (gemessen von der Dekodierung bis zur Fertigstellung) also 3 Takte, bzw. einen Takt länger als die übrigen Befehle. Mit dem 3. Takt kann aber bereits der folgende Befehl dekodiert werden.

- STR $1: Mit dem Opcode liegt die Zieladresse DS-A am Datenspeicher (mit Wert 10‘h 001). Der Inhalt des Akkus wird mit dem folgenden Takt in die Speicherzelle DS(1) geschrieben.

Die Ausführung des Programms zeigt folgende Eigenschaften des Prozessors: (1) Auch bei der Ausführung reicht ein Befehl in den folgenden hinein: Während des Abspeichern eines Ergebnisses in den Akku bzw. in den Datenspeicher kann bereits der folgende Befehl dekodiert werden. (2) Akku und Datenspeicher sind hierbei ebenfalls als Schieberegister hintereinander geschaltet. Folgende Abbildung zeigt diese Anordnung.

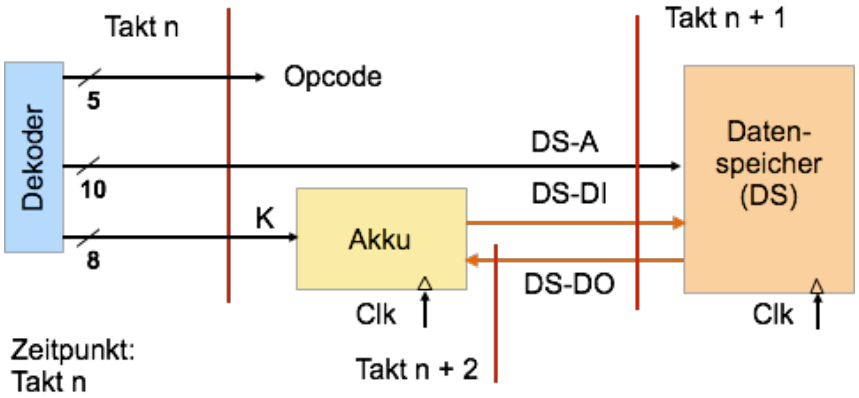


Bild 3.9 Akku und Datenspeicher als Schieberegister

Zum Zeitpunkt n sei der aktuelle Befehl dekodiert und die Zieladresse DS-A bzw. Konstante K liege vor. Es benötigt einen weiteren Takt, um die Konstante K im Akku zu speichern, bzw. um einen Wert aus der adressierten Speicheradresse zu lesen. Ist letztere Operation Teil eines arithmetischen oder logischen Befehls wie ADD $0, so wird ein weiterer Takt benötigt, um das Ergebnis im Akku zu speichern.

Übung 3.3: Skizzieren Sie den zeitlichen Ablauf Ihres Programms aus Übung 3.1 in einen Zeitdiagramm. Hinweis: Übernehmen Sie die Struktur aus Abbildung 3.8 und verwenden Sie der Übersichtlichkeit halber ggf. ein Programm zu Tabellenkalkulation.

Fliessbandverarbeitung

Wie die Ausführung des Programmbeispiels zeigt, sind die Schritte zur Ausführung der Befehle so aneinander gekettet, dass möglichst in keiner der beteiligten Stationen ein Leerlauf entsteht. Diese Art der Ausführung ist auch als Fliessbandverarbeitung bekannt (engl. pipeline processing). Wie in der Serienproduktion entsprechen Programmspeicher, Dekoder, Rechenwerk und Datenspeicher einzelnen Stationen im Fertigungsprozess. Mit dem Systemtakt als Arbeitstakt wird wie an einem Förderband das Werkstück im Prozessor der Befehl an der nächsten Station weiter verarbeitet.

Damit diese Methode funktioniert, müssen alle Stationen im Arbeitsprozess ihren Arbeitsschritt im Takt beenden. Folgend Abbildung zeigt das Prinzip. Als Arbeitsstationen sind Programmspeicher, Dekoder, Rechenwerk und Datenspeicher eingezeichnet. Es wird angenommen, dass der Arbeitsprozess von links nach rechts verläuft. Mit jedem Arbeitstakt wird das Werkstück von einer Arbeitsstation zur nächsten übergeben. Das Fliessband läuft also nicht kontinuierlich, sondern befördert alle Werkstücke mit dem nächsten Arbeitstakt eine Station weiter.

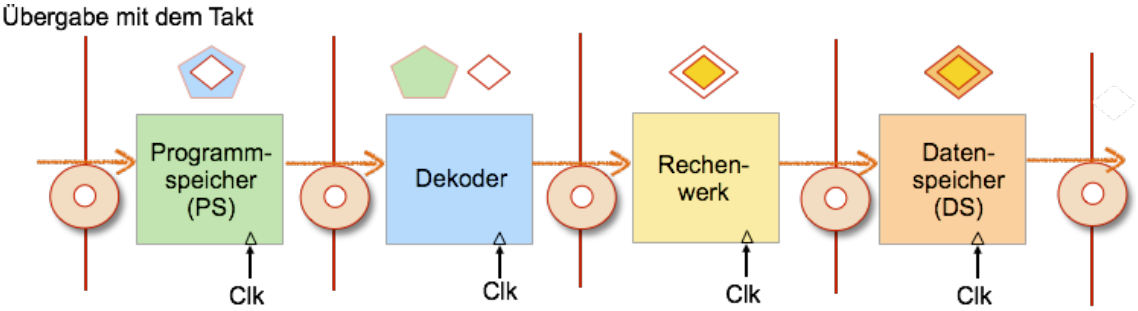


Bild 3.10 Fliessbandverarbeitung

Auf diese Methode lassen sich alle Arbeitsstationen gleichzeitig auslasten. Leerlauf gibt es nur mit dem Anlaufen der Strecke mit dem ersten Werkstück. Was geschieht aber, wenn eine der beteiligten Stationen für bestimmte Operationen mehr als einen Arbeitstakt benötigt? Im Programmbeispiel war das beim Befehl ADD der Fall. Hier benötigt das Rechenwerk einen zusätzlichen Takt.

Während dieses zusätzlich benötigten Taktes müssen alle Stationen vor dem Rechenwerk einen Takt aussetzen. Das Fliessband vor dem Rechenwerk wird hierzu einen Takt lang angehalten. Wie man im Zeitdiagramm in Abbildung 3.8 erkennt, zählt während dieses zusätzlich benötigten Taktes der Befehlszähler nicht weiter, und der Dekoder dekodiert auch nicht seinen bereits geladenen nächsten Befehl.

Ohne diese Massnahme (Fliessband vor der Station anhalten) käme der komplette Ablauf durcheinander. An einem realen Fliessband bekäme das Rechenwerk das nächste Werkstück zugeschoben, bevor es mit dem aktuellen Werkstück fertig ist. Hier fällt also eins der Werkstücke vom Band und der Prozess gerät völlig durcheinander.

Welche Einheit ist nun dafür zuständig, zu erkennen, dass ein aktueller Arbeitsschritt in einer der beteiligten Stationen einen zusätzlichen Arbeitstakt benötigt? In der Praxis übernimmt diese Rolle am besten die Komponente, die auch in den Prozess eingreifen kann und das Fliessband für alle

vorderen Stationen für die benötigte Zeit anhält. Im Blockschaltbild ist diese Einheit die Ablaufsteuerung des Steuerwerks. Diese Komponente besitzt Steuerleitungen zu allen übrigen Komponenten. Sie erkennt am Opcodes des Befehls den Bedarf nach einem zusätzlichen Arbeitstakt und hält für diese Zeit die voraus produzierenden Einheiten an.

3.3. Ablauf der einzelnen Befehle

Wie werden die einzelnen Befehle nun abgearbeitet? Im folgenden wird der Ablauf der Befehle der Gruppen NOP, Schiebeoperationen, logische und arithmetische Operationen, Speicherzugriffe und Sprungbefehle Schritt für Schritt erläutert.

No Operation

Dieser Befehl bewirkt nur, dass der Programmzähler inkrementiert wird, so dass der nächste Befehl geladen wird.

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

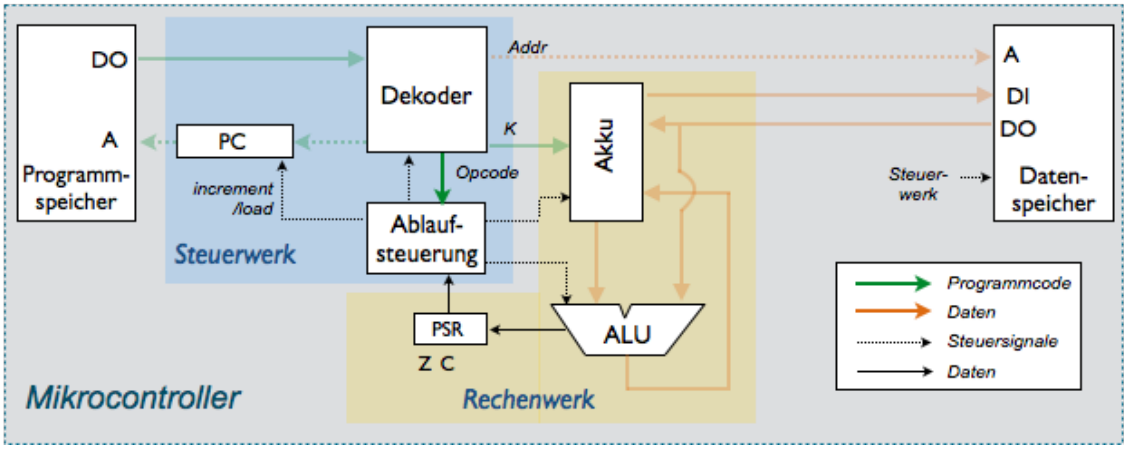


Bild 3.11 Ausführen des Befehls NOP

Aktion: Das Steuerwerk veranlasst keine Aktion. Das Inkrement-Signal für den Programmzähler (PC) bleibt gesetzt, so dass der PC nach dem Laden des folgenden Befehls weiter inkrementiert wird.

Benötigte Takte: ein Takt.

Schiebeoperationen

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

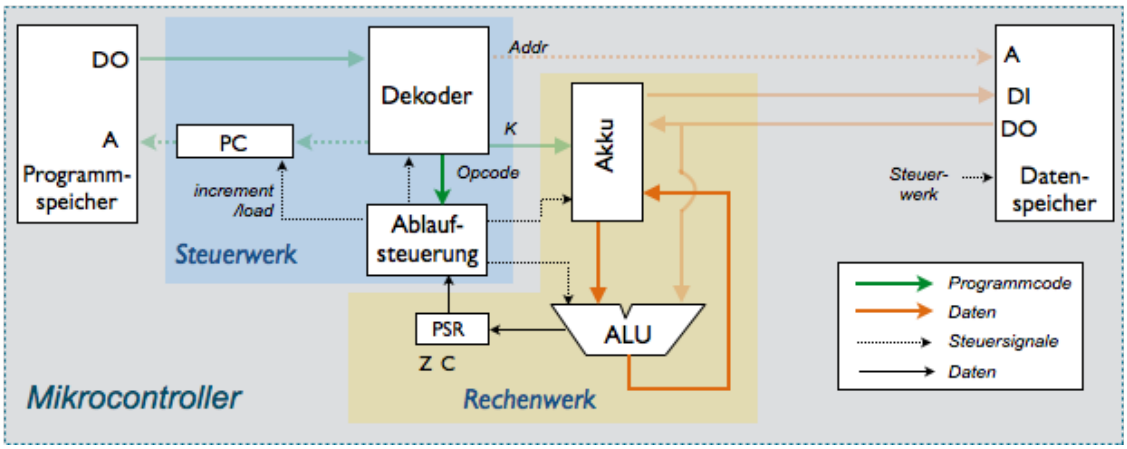


Bild 3.12 Ausführung der Schiebeoperation

Aktion: Die Schiebeoperationen lsl und lsr wirken auf ALU und Akku und benötigen keine weiteren Operanden (der Operand befindet sich bereits im Akku und steht am Eingang der ALU). Das Steuerwerk signalisiert der ALU die Verschiebeoperation.

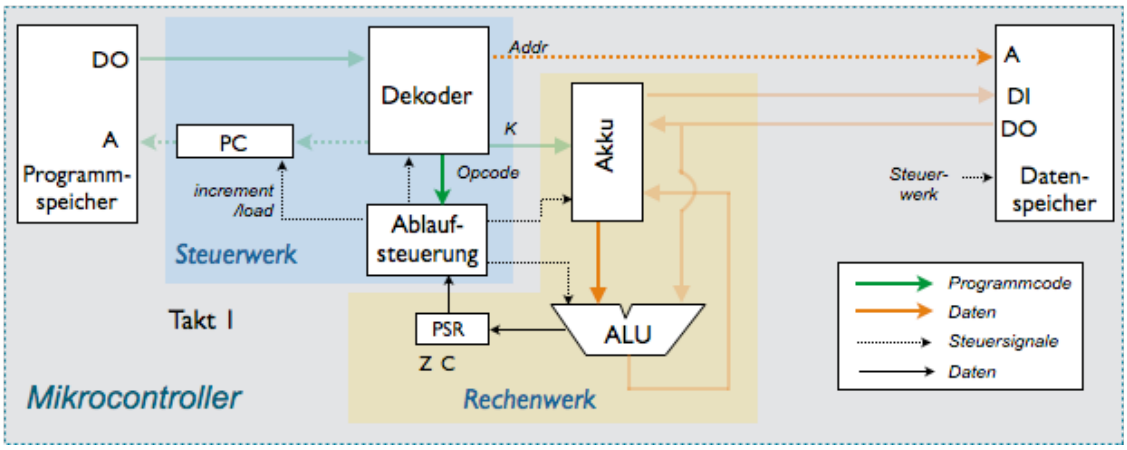
Die ALU arbeitet als Schaltnetz, führt die Verschiebung unmittelbar durch und setzt das PSR entsprechend. Das Ergebnis steht am Ausgang der ALU zur Verfügung und wird mit dem nächsten Takt in den Akku übernommen.

Benötigte Takte: ein Takt.

Arithmetische und Logische Operationen

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

Logische und arithmetische Operationen enthalten neben dem Opcode die Adresse des zweiten Operanden, der mit dem Inhalt des Akkus verarbeitet wird. Diese Adresse legt der Dekoder an den Adressbus des Datenspeichers.



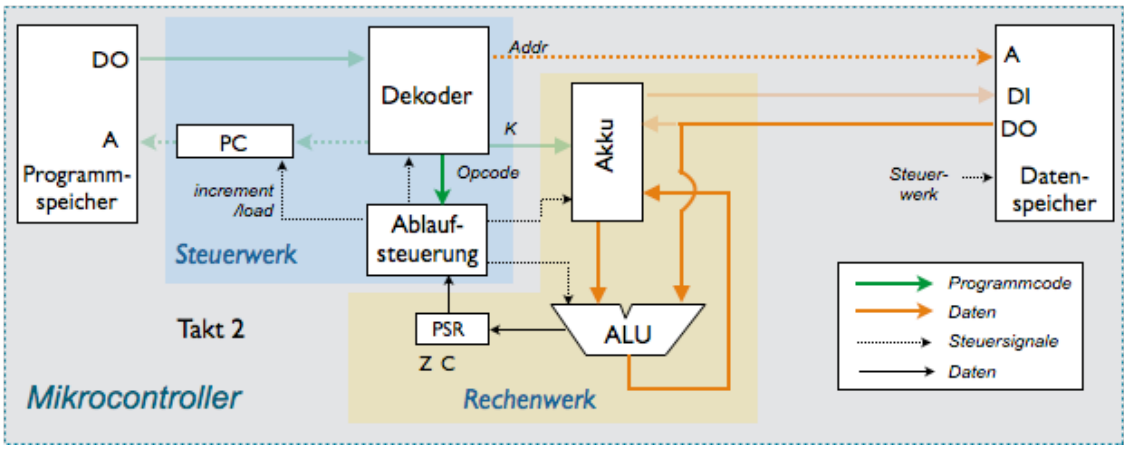


Bild 3.13 Ausführung logischer und arithmetischer Operationen

Aktion: Ein Takt wird benötigt, um den Operanden aus dem Datenspeicher auszulesen. Das Steuerwerk legt im ersten Takt das Lesesignal an den Datenspeicher. Der Akku wird während dieses Taktes nicht benötigt. Sein Inhalt enthält bereits den anderen Operanden der Operation. Ebenso führt die ALU im ersten Takt keine Operationen durch. Damit kein weiterer Befehl aus dem Programmspeicher geladen wird, wird das Inkrementieren des PC deaktiviert (der PC zeigt ja bereits auf den nächsten Befehl), sowie der Programmspeicher und der Dekoder angehalten. Das Fliessband der vorgelagerten Arbeitsabschnitte steht somit in diesem Takt still.

Im zweiten Takt steht der zweite Operand aus dem Datenspeicher zur Verfügung. In der ALU erfolgt unmittelbar die logische bzw. arithmetische Operation mit den nunmehr vorliegenden beiden Operanden. PSR und Ergebnis stellt die ALU als Schaltnetz unmittelbar zur Verfügung, so dass mit dem folgenden Takt der Akku das Ergebnis aufnehmen kann.

Damit das Steuerwerk den zweitaktigen Modus korrekt bearbeitet, schaltet es beim ersten Takt in einen Zweitaktmodus (als Zustand eines Zustandsautomaten). In diesem zweiten Zustand verhält es sich dann bei gegebenen Opcode anders als im ersten Takt und führt die benötigten Operationen aus.

Benötigte Takte: Zwei Takte.

Speicherzugriffe

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

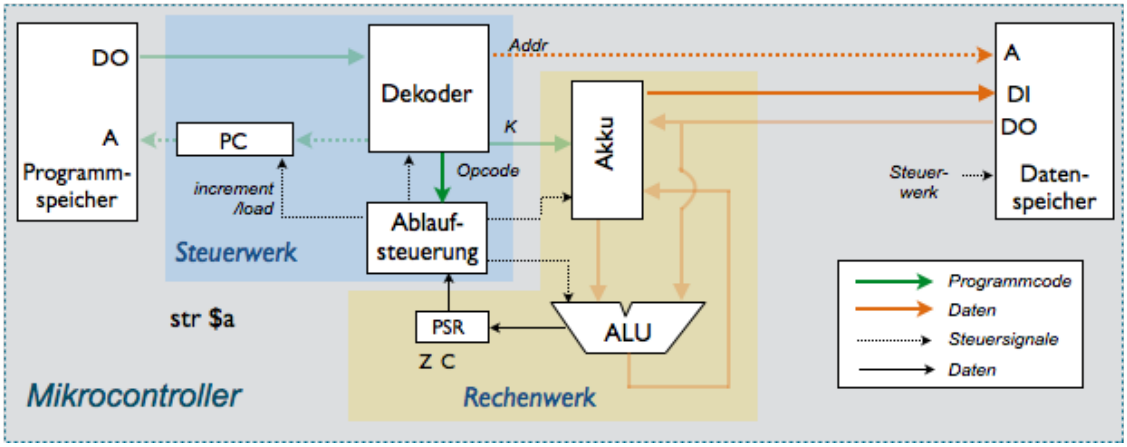


Bild 3.14 Schreiboperation ausführen

Aktion: (A) Schreiboperation: Die Zieladresse liegt zusammen mit dem dekodierten Opcode bereits vor. Der Akku enthält bereits den zu speichernden Inhalt. Der Datenspeicher erhält ein Lesesignal und kann den Inhalt des Akkus mit dem folgenden Takt übernehmen.

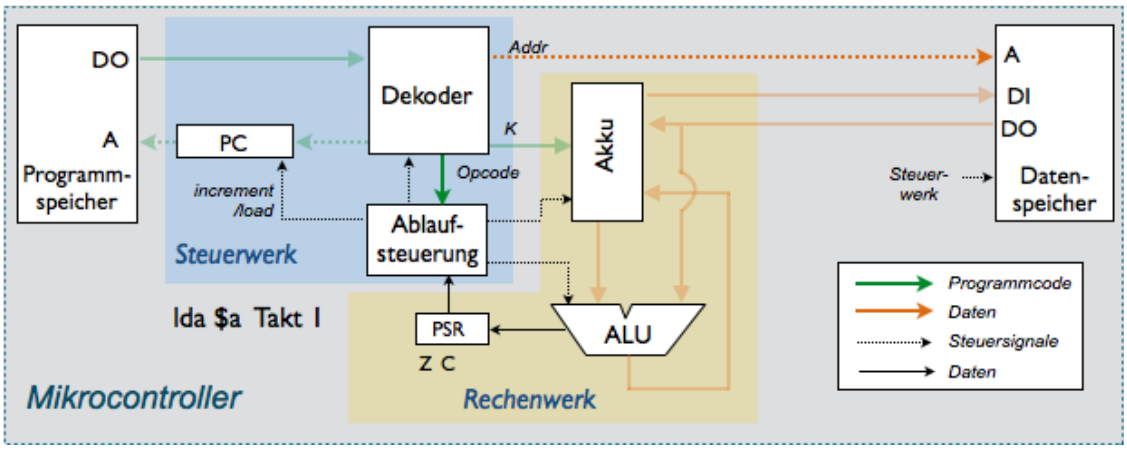


Bild 3.15 Leseoperation ausführen

Aktion (B) Leseoperationen: Die Quelladresse liegt zusammen mit dem dekodierten Opcode bereits vor. Der folgende Takt wird benötigt, um die Quelle aus dem Datenspeicher auszulesen. Der Akku kann den Wert dann mit dem nächsten folgenden Takt übernehmen.

Benötigte Takte:

- Schreiboperation: Ein Takt
- Leseoperation: Zwei Takte

Übung 3.4: Funktionen der Ablaufsteuerung: Gehen Sie den Ablauf des Beispielprogramms aus Abbildung 3.8 Befehl für Befehl durch. Welche Aktionen muss die Ablaufsteuerung jeweils durchführen? Vermerken Sie die Aktionen im Zeitdiagramm. Definieren Sie passende Steuersignale für die einzelnen Komponenten des Mikrocontrollers.

Übung 3.5: Funktionen der Ablaufsteuerung: Gehen Sie den Ablauf Ihres Programms aus Übung 3.1 Befehl für Befehl durch. Welche Aktionen muss die Ablaufsteuerung jeweils durchführen? Vermerken

Sie die Aktionen im Zeitdiagramm. Hinweis: Verwende Sie der Übersichtlichkeit halber ggf. ein Programm zur Tabellenkalkulation.

Sprungbefehle

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

Aktion: Das Steuerwerk liest das PSR aus und entscheidet, ob die Sprungbedingung ausgeführt wird. Ergebnisse: (1) kein Sprung: keine Aktion, mit dem folgenden Takt könnte der nächste Befehl geladen werden. (2) Sprung: der Offset für den Programmzähler (bzw. der absolute Wert) liegt bereits am Ausgang des Dekoders an. Das Steuerwerk gibt dem PC Anweisung, seinen Inhalt mit dem nächsten Takt mit dem Offset zu erhöhen (bzw. den absoluten Wert zu laden). Programmspeicher und Dekoder werden für diesen Takt deaktiviert, damit während der Aktualisierung des PCs nicht der bisher adressierte Befehl aus dem Programmspeicher gelesen wird.

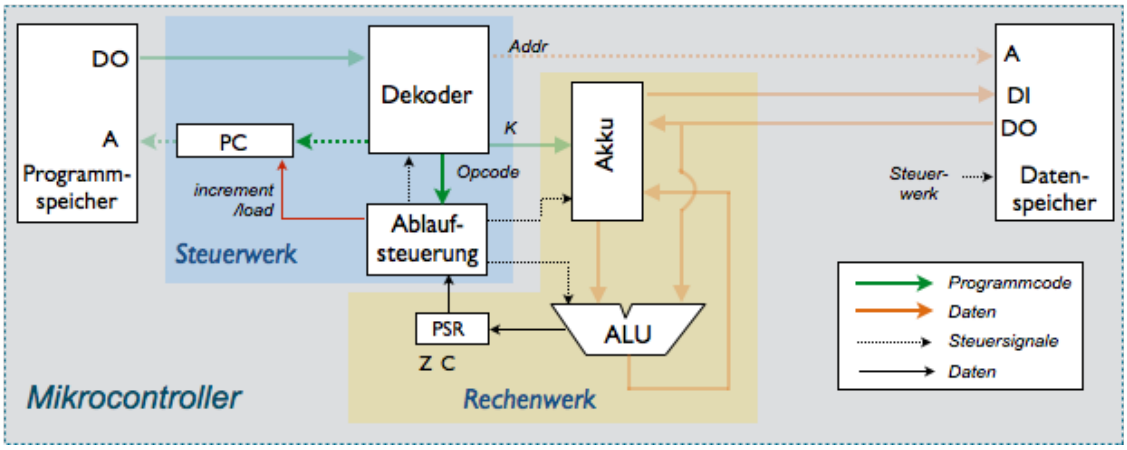


Bild 3.11 Sprungbefehl ausführen

Benötigte Takte: Zwei Takte.

Da ja vor Prüfung der Bedingung nicht feststeht, ob ein Sprung stattfindet. Werden Programmzähler, Programmspeicher und Dekoder auf jeden Fall angehalten. Im folgenden Takt wird dann für den Fall, dass kein Sprung erfolgt, die Bearbeitung fortgesetzt. Da die Pipeline nur angehalten wurde, ist die Fortsetzung ohne Störungen möglich.

Im Fall, dass gesprungen wird, hat der Programmzähler im zweiten Takt den neuen Stand. Mit dem folgenden Takt liest der den Befehl an der Sprungadresse aus dem Programmspeicher. Allerdings befinden sich auf dem Weg vom Programmspeicher zum Dekoders noch vor dem Anhalten der Pipeline bereits geladenen, nun nicht mehr korrekte Befehl.

Diese Situation lässt sich z.B. dadurch entschärfen, dass man im Programm im Anschluss an einen Sprungbefehl grundsätzlich einen Leerlaufbefehl (NOP) anbringt, der nur der Fließbandverarbeitung geschuldet ist. In einer Entwicklungsumgebung übernimmt solche Massnahmen der Compiler für das Assembler-Programm bzw. der Compiler für die Programmierung in Hochsprachen.

3.4. Programme mit Sprungbefehlen

Als Beispiel für ein programm mit Sprungbefehlen soll das Maximum zweier Zahlen A und B bestimmt werden. Die beiden Zahlen seien im Datenspeicher in den Speicherzellen DS(1) und DS(2) abgelegt. Das Ergebnis soll in Speicher DS(0) abgelegt werden. Den Ablauf beschreibt folgende Programmtext.

```
; ASL (Assembler Language) für den DHBW MCT-Mikroprozessor
; Maximum zweier Zahlen A und B ermitteln
; Bedingung: A und B sind im Datenspeicher bei $01 und $02 abgelegt
;
             ld  $01           ; Akku <= A
checkAB:     sub $02           ; Akku <= A - B
             brbc N, maxB      ; if (A ≥ B) then branch to maxA
maxB:        ld  $02           ; Akku <= B
             st  $00           ; DS(0) <= B
             ldi 0             ; Akku <= 0
             st  $03           ; (DS(3) <= 0
             anda $03          ; ALU <= 0 AND 0 = Zero
             brbs Z, end       ; branch to end
maxA:        ld  $01           ; max(A,B) = A
             st  $00           ; DS(0) <= A
end:         ld  $00           ; Akku <= max (A,B)
```

Der erste Sprungbefehl verwendet das Vorzeichenbit des Differenz von A-B. War A größer oder gleich B, so bleibt das Ergebnis positiv, also das Vorzeichenbit N = 0. Diese Bedingung wird mit der Anweisung BRBC (engl. für branch if bit clear) mit Verweis auf das Vorzeichenbit geprüft. In diesem Fall ist A das Maximum, und das Programm verzweigt zur Markierung maxA. An der Markierung wird das Ergebnis max(A,B) = A in der Speicherzelle DS(0) abgelegt.

Im anderen Fall (B > A) findet kein Sprung statt, der nächste Befehl wird geladen. Der besseren Lesbarkeit wegen ist an dieser Stelle die Markierung maxB eingefügt. In diesem Zweig wird das Ergebnis max (A,B) = B an der Speicherzelle DS(0) abgelegt. Die Markierung maxA muss dann aber übersprungen werden. Statt eines unbedingten Sprungbefehls wird hierfür ein bedingter Sprung mit erzwungener Bedingung verwendet: (1) ldi 0 lädt den Wert Null in den Akku, (2) die folgende Prüfung auf das Zero-Bit führt dann auf jeden Fall zu einem Sprung ans Ende des Programms.

Übung 3.6: Übersetzen Sie das Programm in die Maschinensprache. Wie erfolgt die Kodierung der Sprungbefehle? Erläutern Sie Ihr Vorgehen.

Die Abbildung auf der folgenden Seite zeigt den Ablauf des Programms. Hierbei wurden vorab folgende Werte im Datenspeicher hinterlegt A= 7 in DS(1), B= 5 in DS(2). Man erkennt im Ablauf das Laden des Wertes A in den Akku. Hierauf folgt die Subtraktion von B. Da der Inhalt des Akkus mit den vorgegebenen Werten positiv bleibt, ist das Maximum beider Werte A. Der folgenden Sprungbefehl prüft das Vorzeichen des Ergebnisses und wird somit ausgeführt.

Im Ablauf erkennt man, das im zweiten Takt des Sprungbefehls (BRBC) der Programmzähler (PC) auf die Zieladresse erhöht wird. Das Programm springt zur Markierung maxA und lädt den Befehl ld $01, d.h. den Wert A in den Akku. Dieser wird dann in die Speicherzelle DS(0) geschrieben.

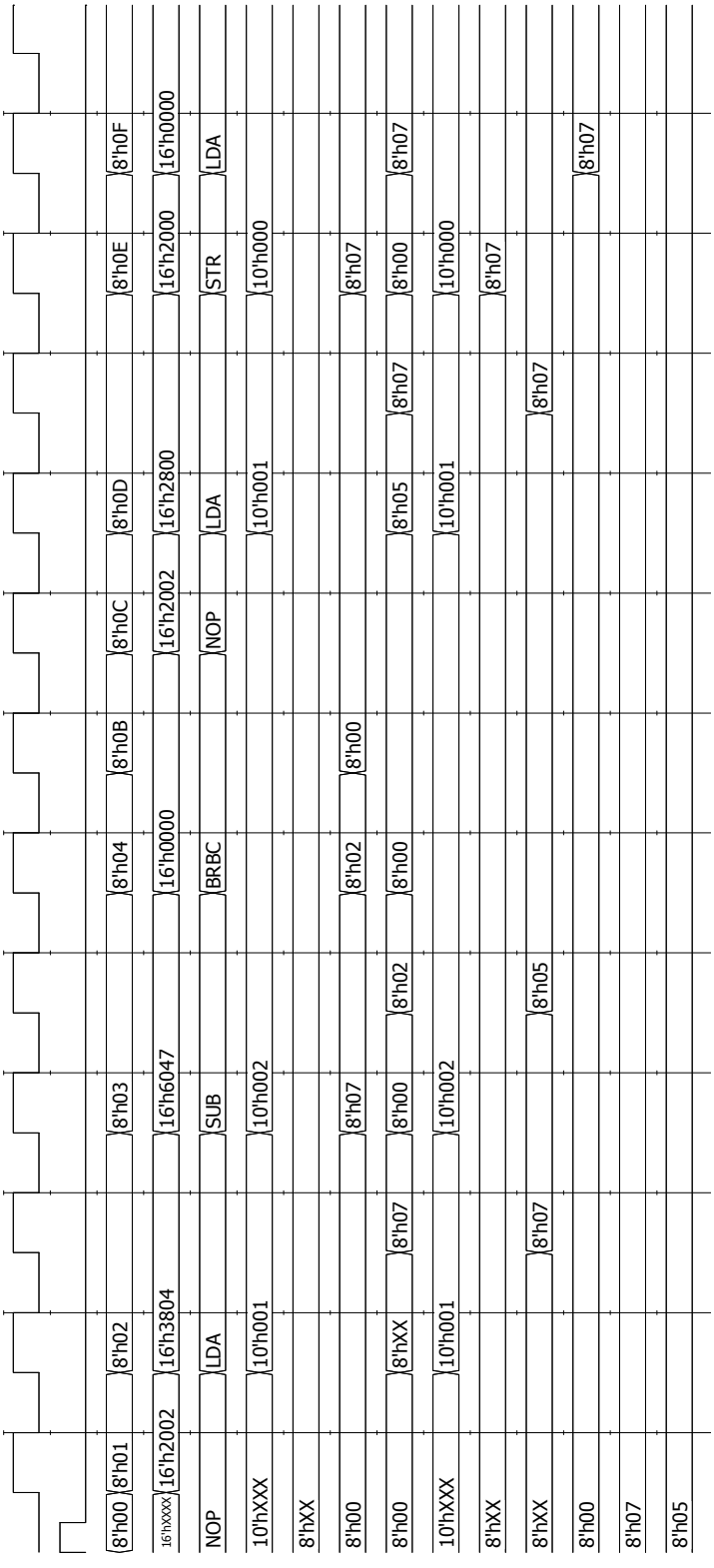


Bild 3.12 Ablauf des Programms mit Sprungbefehlen

Im Akku und im Speicher DS(0) verbleibt das Maximum der beiden Werte. Im Zeitdiagramm erkennt man ebenfalls, dass in das Programm in Maschinensprache zwei Leerlaufbefehle (NOP) vor den Sprungbefehl eingefügt wurden. Diese dienen dazu, zu vermeiden, dass auf dem Weg vom Programmzähler zur Ablaufsteuerung bei einem Sprung nicht mehr passende Befehle in die Verarbeitung geraten.

Übung 3.7: Vertauschen Sie die Werte A und B im Programmbeispiel (d.h. DS(1) = A = 5 und DS(1) = B = 7. Wie sollte das Programm jetzt ablaufen? Erstellen Sie ein Zeitdiagramm. Hinweis: Verwenden Sie der Übersichtlichkeit halber ggf. ein Programm zur Tabellenkalkulation.

Übung 3.8: Erstellen Sie in Assembler-Sprache unter Verwendung des gegebenen Befehlssatzes ein Programm zur Berechnung des Maximums dreier Zahlen max(A, B, und C). Hierbei sei angenommen, dass sich die Zahlen im Datenspeicher unter den Adressen 1, 2 und 3 befinden. Das Ergebnis wird im Speicher unter der Adresse 0 abgelegt. Hinweis: Erstellen Sie zunächst ein Aktivitätsdiagramm bzw. einen Ablaufplan für das Vorgehen. Verwenden Sie Sprungbefehle und geeignete Prüfbedingungen hierfür.

Übung 3.9: Skizzieren Sie den zeitlichen Ablauf des Programms aus Übung 3.8 für ein ausgewähltes Szenario. Erstellen Sie ein Zeitdiagramm. Hinweis: Verwenden Sie der Übersichtlichkeit halber ggf. ein Programm zur Tabellenkalkulation.

4. Erweiterungen des Mikrocontrollers

4.1. Unterprogramme

Durch strukturierte Programmierung lassen sich Programme übersichtlicher gestalten. Für häufig verwendete Aktivitäten lassen sich beispielsweise in Makros bzw. Unterprogrammen unterbringen. Bei Makros werden wiederkehrende Aktivitäten für den Assembler mit speziellen Anweisungen geklammert. Im Programmtext wird dann stellvertretend für die Aktivität der Name des Makros aufgerufen. Diese Methode hat auf den Prozessor keinerlei Einfluss. Der Assembler fügt jedes Mal, wenn das Makro aufgerufen wie, den Programmcode des Makros in den Maschinencode ein.

Eine andere Methode zur strukturierten Programmierung ist die Verwendung von Unterprogrammen. Auch hier wird eine wiederkehrende Aktivität als Einheit verpackt, statt als Makro allerdings als Unterprogramm. Im Assemblerprogramm wird dort, wo die besagte Aktivität benötigt wird, der Name des Unterprogramms aufgerufen. Es ist hierzu also eine Erweiterung des Befehlssatzes des Prozessors erforderlich.

Die Realisierung erfolgt so, dass der Assembler den Programmcode des Unterprogramms an einer geeigneten Stelle im Programmspeicher ablegt. Bei Aufruf des Unterprogramms erfolgt ein Sprung an diese Adresse. Zur Unterstützung von Unterprogrammen sind im Prozessor folgende Erweiterungen erforderlich:

- Befehl zum Aufrufen eines Unterprogramm (aus dem Hauptprogramm)
- Befehl zur Rückkehr ins Hauptprogramm (aus dem Unterprogramm)
- Rettung (Speicherung) des Programmzählers beim Sprung ins Unterprogramm, damit das Hauptprogramm bei der Rückkehr an dieser Stelle weiter arbeiten kann.
- Rettung von Zwischenergebnissen (Inhalt des Akkus) des Hauptprogramms, die bei der Rückkehr ins Hauptprogramm wieder benötigt werden.

Abstrakt ausgedrückt muss also der Kontext (Befehlszähler und ggf. Inhalt des Akkus) bei Start des Unterprogramms gesichert werden. Bei der Rückkehr aus dem Unterprogramm wird dieser Kontext wieder in den Prozessor geladen. Diese Auslagerung und Wiederherstellung des Kontexts wird in englischer Sprache als Context Switching bezeichnet. Je nach Art des Prozessors gehören zum Kontext neben der Rücksprungadresse (Programmzähler) und dem Akku als einzigem Register auch sonstige benötigte Registerinhalte. Folgende Abbildung zeigt den Kontrollfluss bei Makros bzw .Unterprogrammen.

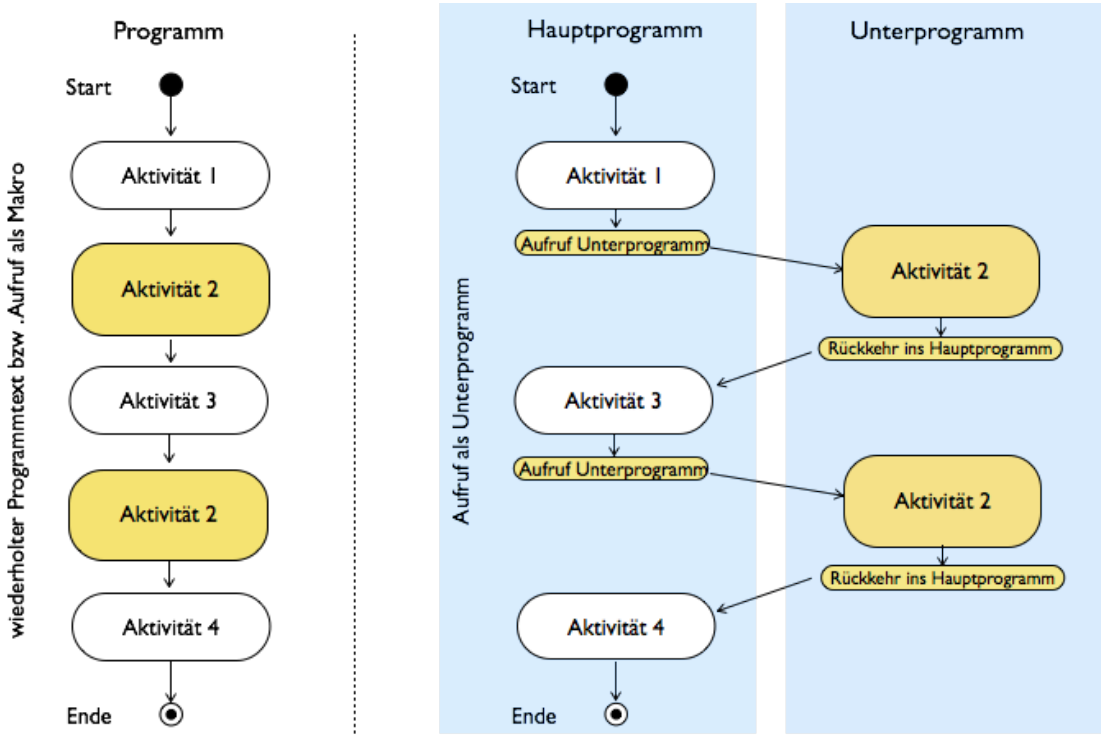


Bild 4.1 Kontrollfluss bei Makros und Unterprogrammen

In Falle der Verwendung von Makros bleibt der Programmablauf im gezeigten Beispiel linear und folgt dem Kontrollfluss. Bei der Verwendung von Unterprogrammen bleibt der Kontrollfluss zwar linear, der Programmablauf verzweigt jedoch in ein Unterprogramm. Das Unterprogramm löst die Rückkehr ins Hauptprogramm aus. Aus dem Unterprogramm liesse sich ein weiteres Unterprogramm aufrufen. Auf diese Weise entsteht ein ineinander verschachtelter Ablauf.

Wohin wird der Kontext des aufrufenden Programms (Programmzähler, ggf. Akku-Inhalt) ausgelagert? Die nahe liegende Möglichkeit ist der Datenspeicher. Da bei der Rückkehr die dort abgelegten Informationen in genau der umgekehrten Reihenfolge benötigt werden, wie sie abgelegt werden, lässt sich dieser Bereich als Stapel (engl. Stack) organisieren: neue Informationen werden übereinander gestapelt und bei Bedarf von oben nach unten wieder abgetragen. Wie wird der ausgelagerte Bereich (Stapel) adressiert? Hierzu wird ein spezielles Register definiert, das als Zeiger auf den nächsten freien Platz auf dem Stapel dient: der Stapelzeiger (engl. Stack Pointer)

Stapel und Stapelzeiger

Folgende Abbildung zeigt die Funktionsweise des Stapels. Vor Aufruf des Unterprogramms zeigt der Stapelzeiger SP auf das nächste freie Feld im Stapel. Bei Aufruf des Unterprogramms wird dort zunächst der aktuelle Stand des Befehlszeigers abgelegt (genauer: die Adresse des auf den den

Aufruf folgenden Befehls). Der Stapelzeiger wird anschliessend inkrementiert. Es folgt die Ablage des Akkus aus dem Hauptprogramm. Der Stapelzeiger wird erneut inkrementiert. Diese Situation ist auf der rechten Seite der Abbildung zu sehen.

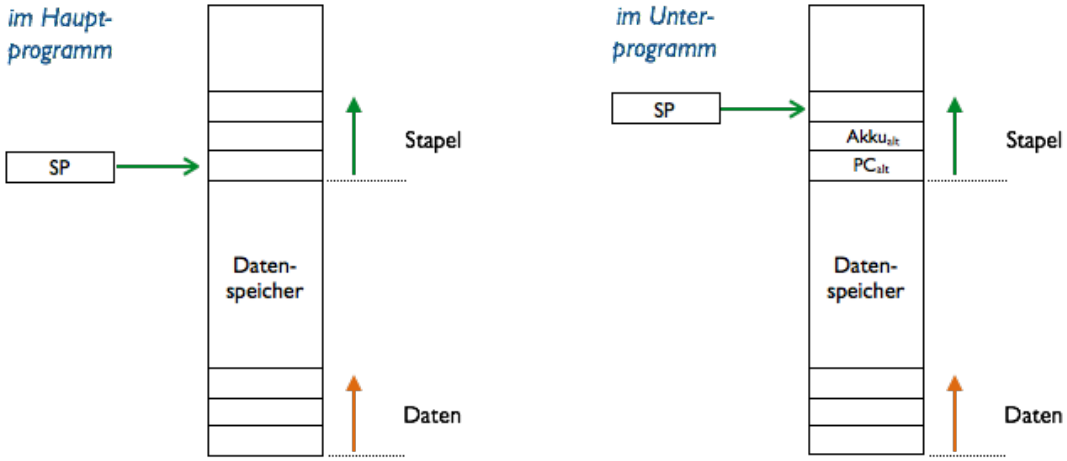


Bild 4.2 Funktionsweise des Stapels

Bei der Rückkehr in das Hauptprogramm wird zunächst der Akku-Inhalt wieder hergestellt (vom Stapel zurück in den Akku transferiert). Anschliessend übernimmt der Befehlszähler den Stand aus dem Stapel. Somit erfolgt ein Rücksprung ins Hauptprogramm an die Stelle unmittelbar nach dem Aufruf des Unterprogramms.

Stapel und Daten teilen sich den Datenspeicher, haben jedoch unterschiedliche Einsprungadressen. Beim hier betrachteten Musterprozessor wurde die Startadresse der Daten willkürlich an der Adresse Null (d.h. DS(0)) festgelegt. Der Stapel beginnt mit der Startadresse 32 (d.h. DS(32). In der Praxis sind je nach Prozessor ebenfalls solche Festlegungen erforderlich.

Neue Befehle

Es wird folgende Arbeitsaufteilung zwischen Hauptprogramm und Unterprogramm gewählt:

- Hauptprogramm: ruft Unterprogramm auf. Hierzu wird ein neuer Befehl definiert: call Ziel. Der Ablauf ist wie bei einem Sprungbefehl an die Adresse Ziel, jedoch wird zusätzlich der Stand des PC (Programmzählers) auf dem Stapel gesichert.
- Unterprogramm: sichert Inhalt des Akkus. Hierzu wird ein neuer Befehl definiert: push. Als einziges Register dient der Akku als impliziter Operand. Sein Inhalt wird auf dem Stapel gespeichert, der Stapelzeiger inkrementiert.
- Unterprogramm: erledigt seine Aktivität.
- Unterprogramm: Bereitet die Rückkehr ins Hauptprogramm vor. Hierzu wird der Inhalt des Akkus aus dem Stapel zurück in den Akku transferiert. Als Gegenspieler des Befehls push wird hierzu der Befehl pop definiert. Der Stapelzeiger wird dekrementiert und zeigt auf den alten Stand des Akkus im Stapel. Implizites Ziel der Operation ist der Akku als einziges Register.
- Unterprogramm: löst die Rückkehr ins Hautprogramm aus. Hierzu wird ein neuer Befehl definiert: ret. Auch dieser Befehl funktioniert wie ein Sprungbefehl, wobei die Zieladresse der auf dem Stapel abgelegte Stand des PC ist. Der Stapelzeiger wird dekrementiert. Der Befehlszähler wird auf den alten Stand zurück gesetzt.

- Hauptprogramm: setzt seine Tätigkeit fort.

Folgendes Beispiel zeigt den Aufruf eines Unterprogramms.

```
; ASL (Assembler Language) für den DHBW MCT-Mikroprozessor
; Muster zum Aufruf eines Unterprogramms
;
main:         ldi  0            ; Akku <= 0
              st  $01           ; DS(1) <= 0
              call count        ;
              call count        ;
              call count        ;
              ...
; count.asm Muster für ein Unterprogramm
count:        push              ; (SP)<= Akku, SP <= SP +1
              ldi 1             ; Akku <= 1
              add $01           ; Akku <= Akku + DS(1)
              st  $01           ; DS(1) <= Akku
              pop               ; SP <= SP -1; Akku <= (SP)
              ret               ; return
```

Übung 4.1: Erläutern Sie die Funktion des Unterprogramms. Was geschieht bei jedem Aufruf?

Übung 4.2: Erläutern Sie den Ablauf im Hauptprogramm. Skizzieren Sie die Inhalte des Stapels mit jedem Aufruf. Skizzieren Sie den Inhalt von DS(1). Skizzieren Sie den Inhalt des Akkus im Hauptprogramm (vor und nach jedem Aufruf des Unterprogramms).

Das Musterprogramm verwendet bereits die neuen Befehle. Folgende Abbildung zeigt die für die Implementierung gewählte Kodierung.

Erweiterungen

Befehlssatz			Beschreibung	Prozessor Status Register (PSR) Flags				
Assembler	Kürzel	Opcode		Zero (Z)	Carry (C)	oVerflow (V)	Sign (S)	Neg. (N)
push	PUSH	E	(SP)<= Akku; SP <= SP+1	-	-	-	-	-
pop	POP	F	SP<SP-1; Akku<= (SP)	-	-	-	-	-
call K	CALL	10	(SP)<= PC; SP<=SP +1; jump to address K	-	-	-	-	-
ret	RET	11	SP<= SP-1; PC<=(SP); jump to PC	-	-	-	-	-

Erweiterungen

nop, push, pop, ret

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					(leer)										

Sprungbefehle
jmp, call

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					X	X	X	K8 (Zieladresse)							

Bild 4.3 Neue Befehle und Befehlsformate

Mit den neuen Befehlen wird die Liste der bisher vorhandenen Befehle verlängert. Der Befehlssatz umfasst nun also 18 Befehle (hexadezimal x11 plus 1). Die Befehle Push und Pop verhalten sich wie Speicherbefehle, wobei im Vergleich zu Ld und St der Operand wiederum implizit der Akku ist,

jedoch die direkte Adresse entfällt, da diese implizit durch den Stapelzeiger gegeben ist. Die Befehle Call K und Ret verhalten sich wie Sprungbefehle, wobei Call K den Sprung vom Haupt-programm bzw. dem übergeordneten Programm aus durchführt, und Ret vom Unterprogramm aus startet.

Für die neuen Befehle sind keine neuen Befehlsformate erforderlich. Push, Pop und Ret haben keine expliziten Operanden und sind wie die Leerlaufoperation Nop kodiert. Der Befehl Call K folgt der Kodierung des absoluten Sprungbefehls Jmp K.

Erweiterungen des Prozessors

Neben den neuen Funktionen durch den erweiterten Befehlssatz ist der Prozessor nun mit einem Stapelzeiger zu erweitern, wie folgende Abbildung zeigt.

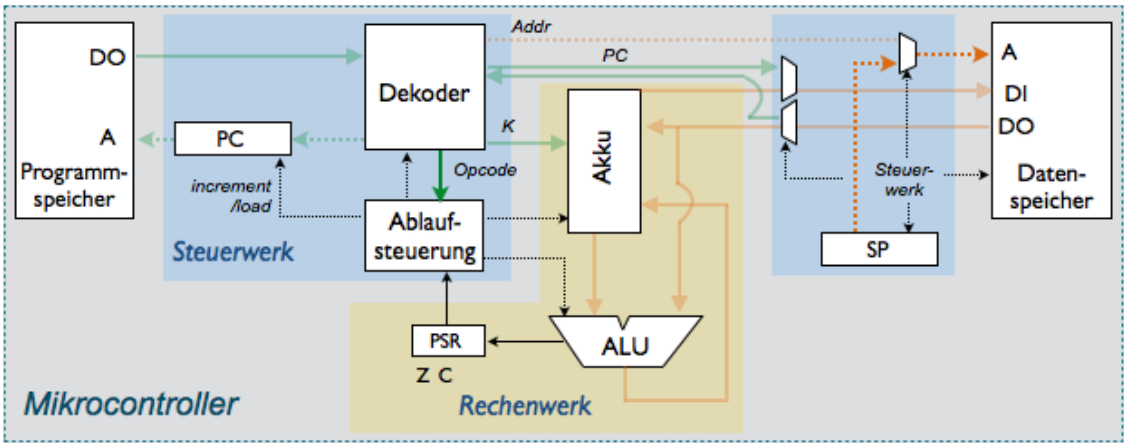


Bild 4.4 Implementierung des Stapels und Stapelzeigers

Der Stapelzeiger (SP) ist als Register ausgeführt, ebenso wie der Befehlszeiger (PC). Ziel des Stapelzeigers ist jedoch der Datenspeicher, nicht der Programmspeicher. Das Inkrementieren zw. Dekrementieren des Stapelzeigers übernimmt die Ablaufsteuerung im Steuerwerk. Die Ablaufsteuerung stellt auch die Weichen für die korrekte Adresse am Datenspeicher durch den hierfür vorgesehenen Multiplexer. Push- und Pop-Operationen erfolgen nun wie Ladezugriffe oder Speicherzugriffe vom Akku aus, jedoch mit der Adresse aus dem Stapelzeiger (SP).

Das Laden bzw. Speichern des Programmzählers für die Befehle Call K bzw. Ret übernimmt ebenfalls die Ablaufsteuerung im Steuerwerk. Die diesbezüglichen Datenpfade sind im Blockdiagramm nicht dargestellt. Auch diese Datentransfers verhalten sich wie Ladeoperationen bzw. Speicheroperationen des Akkus, wobei die Adressen vom Stapelzeiger bereit gestellt werden.

Aus dem Blockdiagramm lassen sich für die einzelnen Befehle folgende Taktzyklen ableiten:

- Call K: 2 Takte: (1) Sperren der Pipeline, (2) Speichern und neu Laden des PC, Aktivieren der Pipeline
- Ret: 2 Takte: (1) Sperren der Pipeline und Auslesen des Stacks, (2) neu Laden des PC, Aktivieren der Pipeline
- Push: 1 Takt (wie Speichern des Akkus mit St Addr)
- Pop: : 2 Takte (wie Laden des Akkus mit Ld Addr)

Programm mit Stapeloperationen

Um den Ablauf des Musterprogramms mit den Operationen zum Sichern und Wiederherstellen des Akkus auf dem Stack, sowie für den Aufruf eines Unterprogramms genauer zu verfolgen, wird der Programmtext in die Maschinensprache mit den genauen Adressen der einzelnen Instruktionen übersetzt. Folgender Abschnitt zeigt den Programmtext.

```
-- Maschinensprache für den DHBW MCT-Mikroprozessor
-- Aufruf des Unterprogramms „count": ab Zeile 12

 0 => "0001100000000000", -- LDI 0 : Hauptprogramm
 1 => "0010100000000010", -- STR $1
 2 => "1000000000001100", -- CALL Count (Zeile 12)
 3 => "0000000000000000",  -- NOP_3
 4 => "1000000000001100", -- CALL Count (Zeile 12)
 5 => "0000000000000000",  -- NOP
 6 => "1000000000001100", -- CALL Count (Zeile 12)
 7 => "0000000000000000",  -- NOP
 8 => "0000000000000000",  -- NOP
 9 => "0000000000000000",  -- NOP
10 => "0000000000000000",  -- NOP
11 => "0000000000000000",  -- NOP
12 => "0111000000000000", -- PUSH: Unterprogramm Count
13 => "0001100000001000", -- LDI 1
14 => "0011000000000010",  -- ADD $1
15 => "0010100000000010", -- STR $1
16 => "0111100000000000", -- POP
17 => "1000100000000000", -- RET
18 => "0000000000000000",  -- NOP_1  (ab hier sind nur noch NOPs im PS)
19 => "0000000000000000",  -- NOP_2
```

Mit Zeile Null startet das Hauptprogramm. Die einzige Aktion besteht darin, den Wert Null in die Speicherzelle mit der Adresse 1 zu speichern. Dann erfolgt der Aufruf des Unterprogramms Count, das im Beispiel bei Zeile 12 beginnt. Da der Aufruf des Unterprogramms einen Sprungbefehl ausführt, wird nach dem Aufruf von Call eine Leerlaufanweisung NOP eingefügt. Diese Anweisung ist zum Zeitpunkt der Dekodierung des Befehls Call bereits aus dem Programmspeicher ausgelesen worden und wird daher als nächste Anweisung ausgeführt, unabhängig von der neuen Adresse im Programmzähler.

Der Programmzähler sollte nun den Befehl in Zeile 12 adressieren, d.h. im Anschluss an den NOP in Zeile 5 sollte der Befehl Push in Zeile 12 dekodiert werden. Die Instruktionen im Unterprogramm count dienen dazu, jeden Aufruf zu zählen und in Speicherzelle 1 zu hinterlegen. Hierzu wird zunächst der Akku mit dem im Hauptprogramm ggf. benötigten Inhalt auf dem Stapel gespeichert (Befehl Push). Anschliessend wird der alte Stand aus Speicherzelle 1 ausgelesen, im Akku inkrementiert und in Speicherzelle zurückgeschrieben. Vor dem Rücksprung ins Hauptprogramm mit Hilfe der Anweisung Ret erfolgt die Wiederherstellung des Akkus vom Stapel mit Hilfe der Anweisung Pop.

Der Rücksprung sollte nun an die Adresse 3 erfolgen, d.h. den nächsten Befehl, der im Hauptprogramm auf den Aufruf Call folgt. Dann erfolgt in Zeile 4 der erneute Aufruf des Unterprogramms Count. Der Rücksprung sollte dieses Mal an die Adresse 5 erfolgen (als nächster Befehl nach dem Programmaufruf). In Zeile 6 erfolgt schliesslich der dritte Aufruf des Unterprogramms. Der

Rücksprung sollte nun an die Adresse 7 erfolgen. Das Zeitdiagramm auf der folgenden Seite zeigt den Ablauf des Musterprogramms.

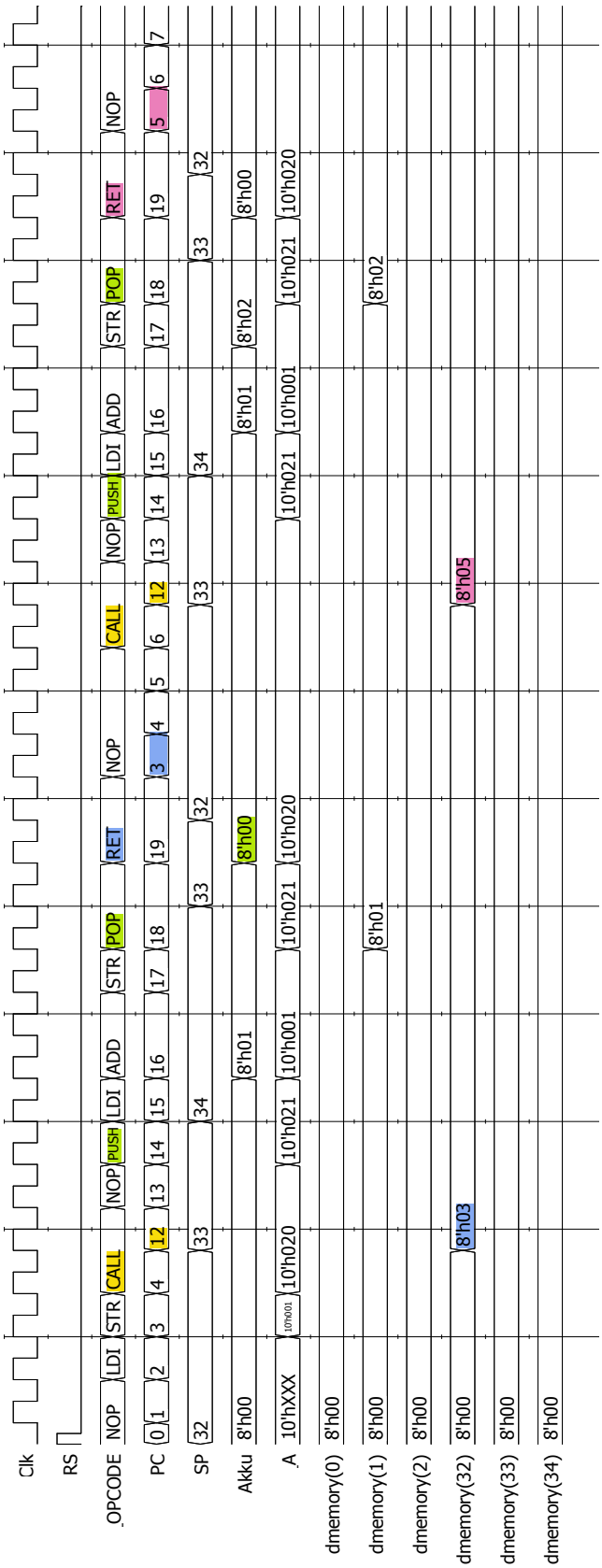


Bild 4.5 Ablauf des Musterprogramms mit Stapeloperationen

Im Zeitablauf dargestellt sind der Opcode, die Stände des Programmzählers (PC) und Stapelzeigers (SP), sowie der Inhalt des Akkus. Darunter findet sich die Adresse A des Datenspeichers, sowie die Inhalte der Speicherzellen 0 bis 2, sowie 32 bis 33. Der Stapelzeiger ist hier so eingestellt, dass er im initialen Zustand auf die Speicherzelle 32 zeigt. Der Stapel wächst also von Adresse 32 aufwärts. Der Ablauf des Programms gestaltet sich wie folgt:

- Aufruf des Unterprogramms mit Opcode Call (gelbe Markierung): Der Programmzeiger zeigt bereits auf die Adresse 4, wenn Call aktuell dekodierte wird (der Befehl Call findet sich an Adresse 2). Das Vorauseilen des PC um 2 ist ein Effekt der bereits erläuterten Fließbandverarbeitung. Die Adresse 4 im Programmzähler wird nun überschrieben mit der Adresse 12 des Unterprogramms. Zur gleichen Zeit wird der alte Stand des PC in die oberste Zeile des Stapels nach DS(32) geschrieben (blaue Markierung). Der Stapelzeiger (SP) wird inkrementiert und zeigt nun auf DS(33).
- Dekodieren des Befehls Push (grüne Markierung) an Adresse 12: Der Programmzähler zeigt hier bereits auf Adresse 14, eilt also wiederum 2 Zeilen vor. Die Leerlaufanweisung NOP vor dem Befehl Push stammt aus Zeile 3 und ist ein Effekt der Fließbandverarbeitung. Mit dem Befehl Push wir der Inhalt des Akkus (Null) im Stapel in die Speicherstelle DS(33) geschrieben (die nach der Initialisierung bereits den Wert 0 hatte). Der Stapelzeiger wird inkrementiert und zeigt nun auf DS(34)
- Ausführung des Unterprogramms: Das Unterprogramm zählt seinen ersten Aufruf. Als Ergebnis wird der Wert 1 nach DS(1) geschrieben. Vor dem Rücksprung ins Hauptprogramm wird mit der Anweisung Pop der alte Inhalt des Akkus vom Stapel gelesen (aus DS(34)) und somit der Akku wieder hergestellt (siehe grüne Markierung). Der Stapelzeiger wird hierbei dekrementiert und zeugt nun auf DS(33) als nächsten freier Platz im Stapel.
- Rücksprung ins Hauptprogramm mit Opcode Ret (blaue Markierung): Aus dem Stapel (Speicherzelle DS(32)) wird der alte Stand des Programmzählers ausgelesen und wieder hergestellt. Der PC zeigt nun auf Zeile 3, d.h. den nächsten Befehl nach dem Aufruf des Unterprogramms in Zeile 2.
- Nächster Aufruf des Unterprogramms mit Opcode Call (gelbe Markierung): Der Programmzähler steht bei 6, wenn Zeile 4 (Call) dekodiert wird, und wird nun wieder auf den Programmtext des Unterprogramms in Zeile 12 gesetzt.
- Der weitere Ablauf folgt den bereits beschriebenen Mechanismen. Jeder Aufruf des Unterprogramms wird durch das Unterprogramm gezählt und in DS(1) gespeichert. Der Rücksprung erfolgt jeweils auf den Befehl, der dem letzten Aufruf folgt. Beim zweiten Aufruf des Programms also nach Zeile 5 (siehe violette Markierung).

Übung 4.3: Skizzieren Sie den Ablauf folgender Befehle im Blockdiagramm: (1) Call K, (2) Ret, (3) Push, (4) Pop. Erläutern Sie den jeweiligen Bedarf an Taktzyklen. Hinweise: Startpunkt ist der dekodierte Befehl. Verwenden Sie die Vorlagen in Anhang A.

Übung 4.4: Erläutern Sie den zeitlichen Ablauf der Befehle aus Übung 4.3 mit Hilfe eines Zeitdiagramms nach dem Muster in Bild 4.5. Hinweis: Verwenden Sie eine handschriftliche Skizze bzw. ein Programm zur Tabellenkalkulation.

Verwendung des Stapels ausserhalb von Unterprogrammen

... Beispiele

... rekursive Programmierung

Bild 4.6 Stapel bei rekursiver Programmierung

...

...

Übung 4.5: ...

Übung 4.6: ...

4.2. Unterbrechungssystem

... Interrupts

...

4.3. Ports für Geräte

...

...

4.4. Timer

...

...

4.5. Serielle Schnittstellen

...

...

Übung 3.1: ...

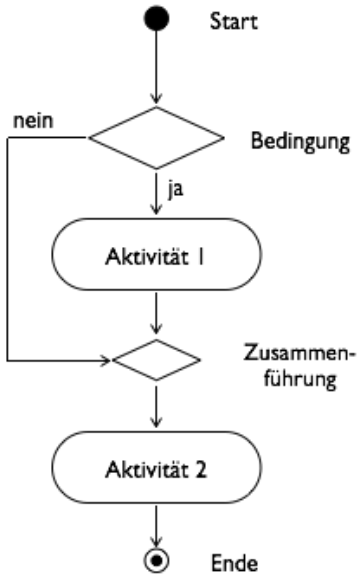
```
. . .
```

...

5. Übungen

5.1. Kontrollfluss mit Verzweigungen (Sprungbefehle)

Folgende Abbildung zeigt einen Kontrollfluss, d.h. den Ablauf eines Programms, als Aktivitätsdiagramm. Das Diagramm beschreibt den Startpunkt des Kontrollflusses, den Endpunkt, sowie die Aktivitäten (in Blöcken mit abgerundeten Ecken).



Kontrollfluss mit Verzweigung

Mit Rauten ist eine Verzweigung bzw. Zusammenführung des Kontrollflusses bezeichnet. In der Verzweigung wird eine Bedingung geprüft. Je nach Ergebnis wird entweder der eine oder der andere Pfad durchlaufen. Die Zusammenführung kennzeichnet den Punkt, in dem die beiden alternativen Pfade wieder zusammen kommen.

Frage 1: In der gezeichneten Darstellung wird je nach Bedingung der Block „Aktivität 1“ übersprungen. Zeichnen Sie Diagramme für folgende Fälle: (1) Alternative Ausführung: Je nach Bedingung wird entweder Aktivität 1 ausgeführt, oder Aktivität 2. Die alternativen Wege laufen dann wieder bei Aktivität 3 zusammen. (2) Fallunterscheidung: Es werden sukzessive folgende Fälle geprüft: Fall 1 => Aktivität 1, Fall 2 => Aktivität 2, Fall 3 => Aktivität 3.

Frage 2: Wie können Sie die in Frage 1 beschriebenen Verzweigungen in einer Hochsprache realisieren (z.B. C, C++, Java)? Nennen Sie Beispiele.

Frage 3: Wie können Sie die in Frage 1 beschriebenen Verzweigungen in Assembler mit den in diesem Manuskript verfügbaren Befehlen programmieren? Nennen Sie Beispiele.

Frage 4: Zählschleifen: (1) Beschreiben Sie in einer Hochsprache eine Zählschleife, die z.B. die Zahlen von 1 bis 10 addiert. (2) Skizzieren Sie ein Aktivitätsdiagramm hierzu. (3) Beschreiben Sie eine Realisierung in Assembler mit den in diesem Manuskript verfügbaren Befehlen.

Frage 5: Schleifen mit Bedingungen: Welche Kontrollflüsse werden beschrieben durch „while Bedingung“ (Abweisschleife) bzw. „do Aktivität while Bedingung“ (Durchlaufschleife)? (1) Zeichnen Sie Aktivitätsdiagramme. (2) Beschreiben Sie eine Realisierung in Assembler mit den in diesem Manuskript verfügbaren Befehlen.

Frage 6: Abbruchbedingungen: Unter Umständen muss der vorgesehene Kontrollfluss unterbrochen werden, z.B. bei Überschreitung des Zahlenbereichs (Überlauf, Unterlauf) bzw. eines Grenzwertes. Wie können Sie solche Abbruchbedingungen realisieren? (1) Skizzieren Sie den Kontrollfluss unter Verwendung eines der bereits vorliegenden Diagramme. (2) Wie ist die Abbruchbedingung in Assembler zu realisieren? Nennen Sie ein Beispiel.

5.2. Unterprogramme

Zur strukturierten Programmierung lassen sich Aktivitäten in Unterprogramme verlagern. Die Unterprogramme lassen sich vom Hauptprogramm wie Befehle aufrufen.

Frage 1: Erstellen Sie ein Unterprogramm zur Multiplikation zweier ganzer Zahlen kleiner als 64. Die Zahlen seien im Datenspeicher unter den Adressen $0 und $1 abgelegt. Das Ergebnis soll unter der Adresse $3 abgelegt werden. Hinweis: Die Multiplikationen lassen sich im binären System aus einer Kombination Additionen und Verschiebungen nach links darstellen. Beispiel: 7 * 3 = 7 * 2 + 7 * 1.

Frage 2: Verwenden Sie das Unterprogramm in einem Hauptprogramm. Wie werden die Parameter übergeben? Wie wird der Rückgabewert übergeben?

Frage 3: Skizzieren Sie den Inhalt des Stapels und des Stapelzeigers beim Ablauf im Programmablauf (mit Bezug zu Frage 2).

Frage 4: ...

5.3. Interrupt-Serviceroutinen

...

...

5.4. Prozessor mit Akku-Architektur

Folgender Ausschnitt zeigt die Schleife eines Hochsprachenprogramms.

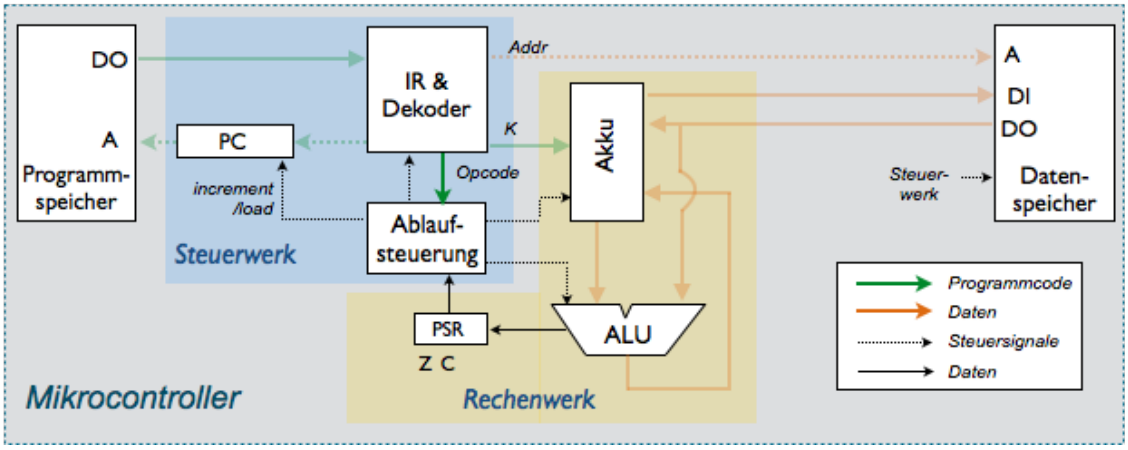
```
// Lauflicht

// global variables
byte lights;

// program loop
void loop() {

   if (lights == 0) {lights = 128; }  // start from 0b10000000

   PORTB = lights;                    // lighting up

   lights = lights >> 1;              // shift right

   delay(100);                        // wait 100 ms
}
```

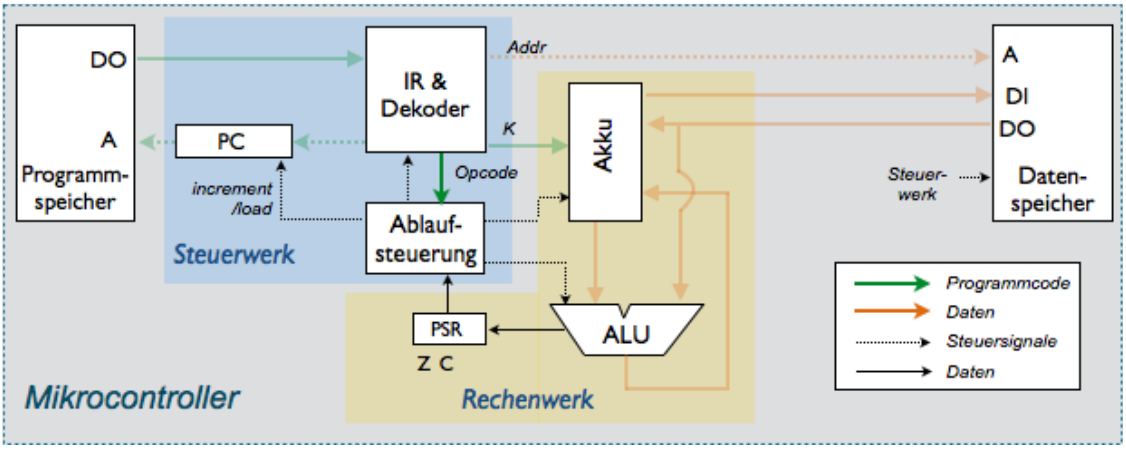
Frage 1.1 (8 Punkte): Übersetzten Sie das Programm in Assemblersprache mit dem Befehls-satz des MCT-Mikrocontrollers. Bitte ergänzen Sie Kommentare zum Ablauf.

Frage 1.2 (4 Punkte): Zum Ablauf auf dem Mikrocontroller muss das Programm in Maschinen-sprache übersetzt werden. Beschreiben Sie den grundsätzlichen Ablauf dieser Übersetzung. Wie werden Befehle und Operanden kodiert? Hinweis: Sie brauchen das Programm hierzu nicht in Maschinensprache zu übersetzen.

Frage 1.3 (8 Punkte): Beschreiben Sie den Ablauf Ihres Programms in einer Tabelle über 10 Takte. Die Tabelle soll folgende Signale zeigen: Clock, PC, PS_DO, Opcode (Dekoder-Ausgang), Addr, K, ALU-Out, Carry-Bit, Akku, PortB. Die Bezeichnungen entsprechen dem Blockschaltbild des Prozessors in der folgenden Abbildung.

Frage 1.4 (4 Punkte): Folgende Abbildung zeigt den MCT-Mikrocontroller mit Akku-Architektur. Beschreiben Sie den Ablauf der Schiebeoperation lsr über zwei Takte.

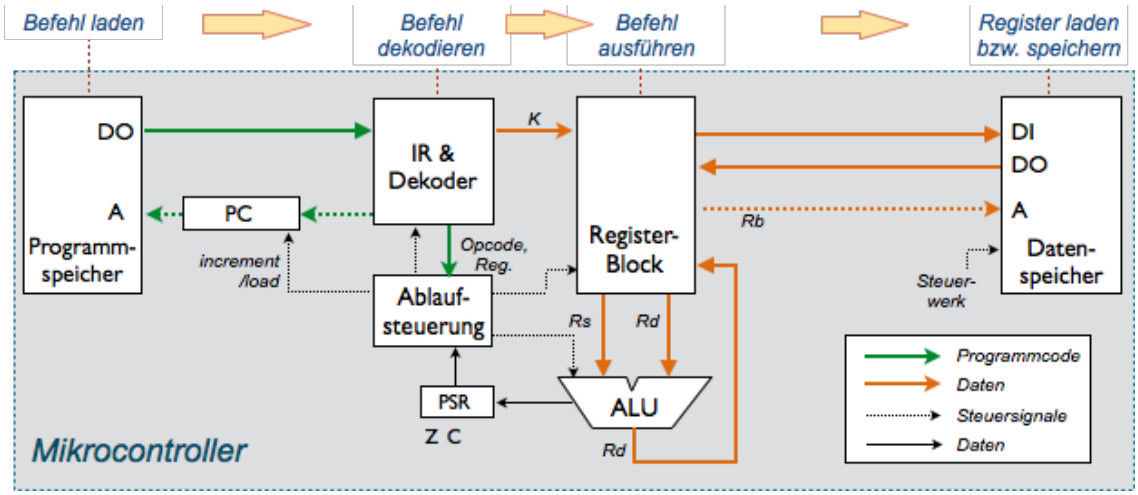




Hinweis: Startpunkt ist der dekodierte Befehl am Ausgang des Dekoders. Zeichnen Sie die jeweils aktiven Signale (Leitungen) im Diagramm nach.

5.5. Prozessor mit Register-Architektur

Anstelle des Akkus wird eine Registerbank mit 16 Registern eingeführt. Jedes Register kann als Akku verwendet werden. Folgendes Blockschaltbild beschreibt die Realisierung.



Frage 2.1 (4 Punkte): Beschreiben Sie die wesentlichen Unterschiede zur Akku-Architektur aus Aufgabe 1 (Laden von Registern, ALU-Operationen, Laden und Speichern von Operan-den im Datenspeicher).

Frage 2.2 (4 Punkte): Nennen Sie Unterschiede im Befehlssatz und in der Befehlskodierung (Beispiele: Sprungbefehle, Laden und Speichern von Ergebnissen, ALU-Operationen).

Frage 2.3 (4 Punkte): Gibt es Vorteile, die die Registerarchitektur gegenüber der Akku-Architektur bringt (Geschwindigkeit, Programmierbarkeit)? Lassen sich die Register als Teil des Datenspeichers realisieren?

Frage 2.4 (6 Punkte): Schreiben Sie Ihr Programm aus Aufgabe 1.1 auf die Registerarchitektur um. Welche Unterschiede gibt es?

5.6. Erweiterungen des Prozessors (Akku-Architektur)

Zur Unterstützung der strukturierten Programmierung durch Unterprogramme wird der Befehlssatz des Mikrocontrollers durch folgende Befehle erweitert:

- CALL Addr: Ruft Unterprogramm auf.
- RET: Rückkehr aus dem Unterprogramm.
- PUSH: Sichert den Inhalt des Akkus im Datenspeicher.
- POP: Stellt den Inhalt des Akkus aus dem Datenspeicher wieder her.

Die Sicherung des Programmzählers und Akkus im Datenspeicher geschieht in einem gesonderten Bereich (Stapel, engl. Stack), der mit Hilfe eines Stapelzeigers (SP für engl. Stack Pointer) adressiert wird. Folgende Übersicht zeigt die Befehlserweiterung.

Erweiterungen

Befehlssatz			Beschreibung	Prozessor Status Register (PSR) Flags				
Assembler	Kürzel	Opcode		Zero (Z)	Carry (C)	oVerflow (V)	Sign (S)	Neg. (N)
push	PUSH	E	(SP)<= Akku; SP <= SP+1	-	-	-	-	-
pop	POP	F	SP<SP-1; Akku<= (SP)	-	-	-	-	-
call K	CALL	10	(SP)<= PC; SP<=SP +1; jump to address K	-	-	-	-	-
ret	RET	11	SP<= SP-1; PC<=(SP); jump to PC	-	-	-	-	-

Erweiterungen

nop, push, pop, ret

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					(leer)										

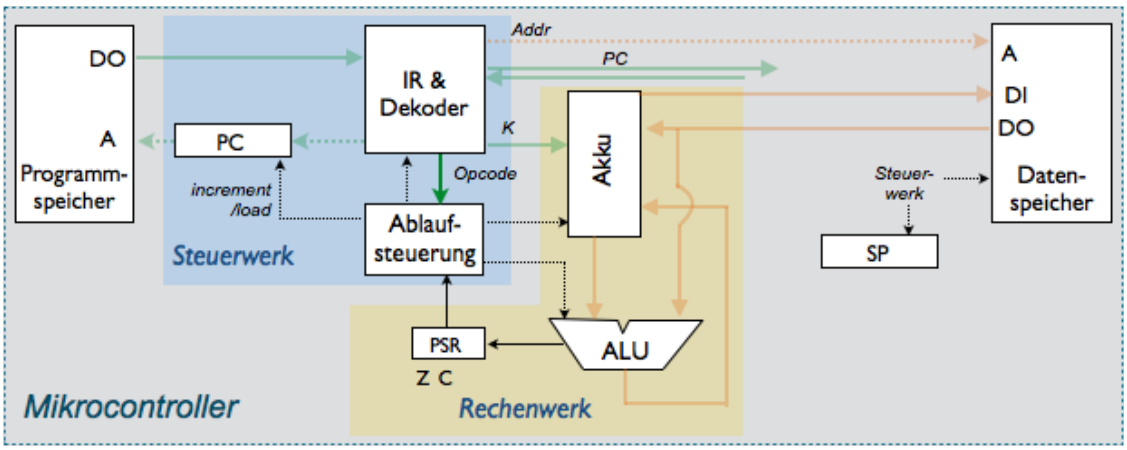
Sprungbefehle
jmp, call

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					X	X	X	K8 (Zieladresse)							

Frage 3.1 (8 Punkte): Erläutern Sie die Funktion der Befehle im Detail (Akku, Stapel, PC).

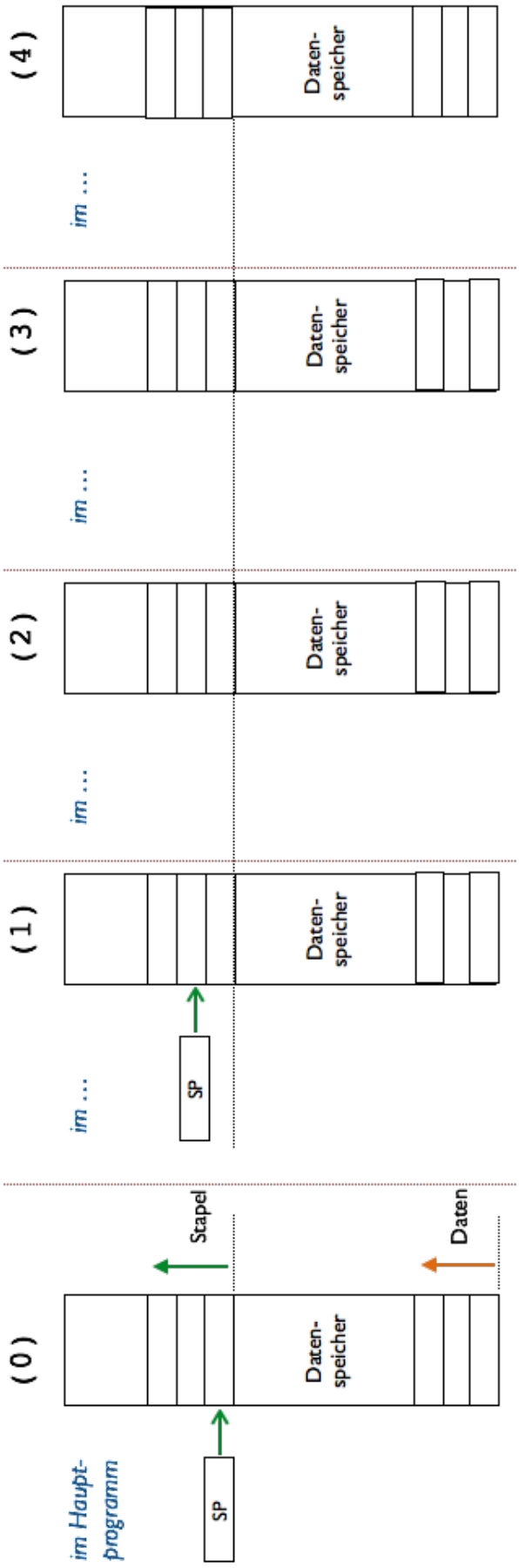
Frage 3.2 (4 Punkte): Welche Befehle werden vom Hauptprogramm aufgerufen? Welche Befehle werden vom Unterprogramm aufgerufen? Skizzieren Sie den Ablauf des Aufrufs eines Unterprogramms in einem Aktivitätsdiagramm.

Frage 3.3 (4 Punkte): Zur Realisierung o.g. Befehle muss der Prozessor erweitert werden. Ergänzen Sie die erforderlichen Erweiterungen im Blockschaltbild.



Frage 3.4 (4 Punkte): Wie viele Takte benötigen die Befehl jeweils: CALL Addr, RET, PUSH, POP. Begründen Sie Ihre Antwort.

Frage 3.5 (4 Punkte): Folgendes Diagramm beschreibt den Stapel im Datenspeicher mit dem SP als Zeiger nach Ablauf eines Befehls. Ergänzen Sie das Diagramm für folgende Schritte: (0) Startpunkt, (1) CALL Addr, (2) PUSH, (3) POP, (4) RET.


(4)
Daten-
speicher
im ...
(3)
Daten-
speicher
im ...
(2)
Daten-
speicher
im ...
(1)
Daten-
speicher
im ...
SP
(0)
Stapel
Daten
Daten-
speicher
im Haupt-
programm
SP

Frage 3.6 (10 Punkte): Das Zeitdiagramm in der folgenden Abbildung (Seite 7) beschreibt den Ablauf eines Programms.

(a) Rekonstruieren Sie den Programmtext aus dem Ablauf (bitte mit Zeilennummern, mit Adressen und Operanden, sowie mit Kommentaren).

(b) Erläutern Sie den Ablauf der Befehle PUSH und POP im Detail (Inhalt Akku und Stapel)

(c) Erläutern Sie den Ablauf der Befehle CALL und RET im Detail (Programmzähler, folgender Befehl, Speicherung und Restaurierung des Programmzählers, Rücksprung an welche Adresse, Ursprung und Zweck des Befehls NOP im Anschluss an CALL).

(d) Erstellen Sie ein Zustandsdiagramm des Prozessors mit den verwendeten Befehlen.

Hinweis: Sie können Zusammenhänge gerne auch im Diagramm mit Pfeilen bzw. Kommentaren markieren.

5.7. Erweiterungen des Prozessors (Akku-Architektur)

...

Englisch - Deutsch

Activity Diagramm	Aktivitätsdiagramm
Arithmetic-Logic Unit	Arithmetisch Logische Einheit
Carry forward	Übertrag
Library	Bibliothek
Look up Table	Funktionstabelle
Network	Netz
Program Counter	Programmzähler
Shift Register	Schieberegister
Stack Pointer	Stapelzeiger
State Diagram	Zustandsdiagramm
State Event Table	Zustandsübergangstabelle

...

Abkürzungen

ALU	Arithmetic-Logic Unit
CPU	Central Processing Unit
IDE	Integrated Development Environment
IP	Internet Protokoll
LUT	Look up Table

...

Literatur

(1) S. Rupp, Mikrocomputertechnik I, Vorlesungsmanuskript, DHBW, 2013

(2) Günter Schmitt, Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie: Programmierung in Assembler und C - Schaltungen und Anwendungen, Oldenbourg Wissenschaftsverlag, 2010, ISBN-13: 978-3486589887

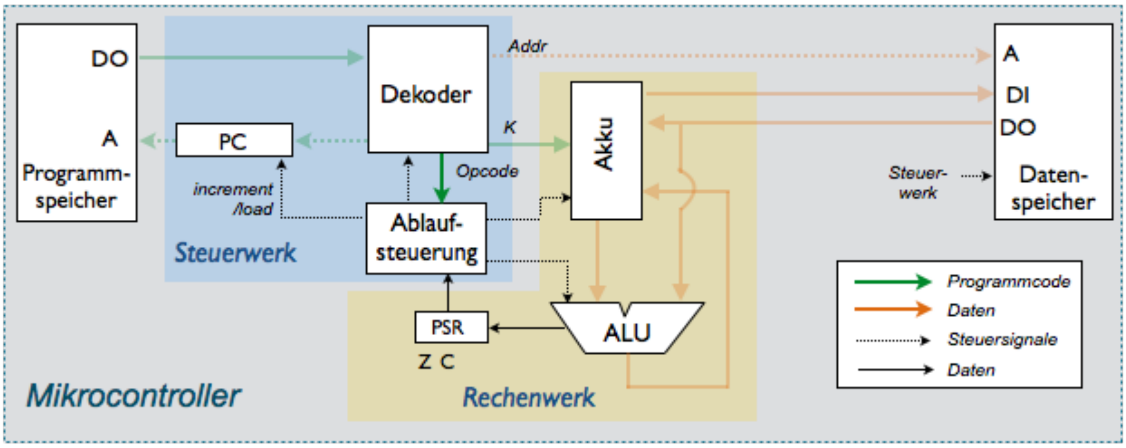
(3) Uwe Brinkschulte, Theo Ungerer, Mikrocontroller und Mikroprozessoren, Springer, 3. Auflage, 2010, ISBN-13: 978-3642053979

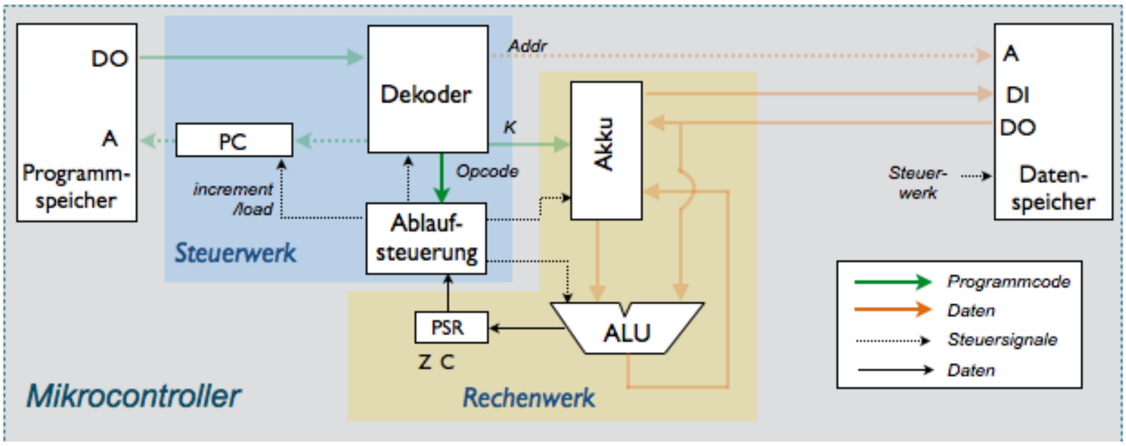
(4) S. Rupp, Mikrocomputertechnik II, Laborübungen, DHBW, 2014

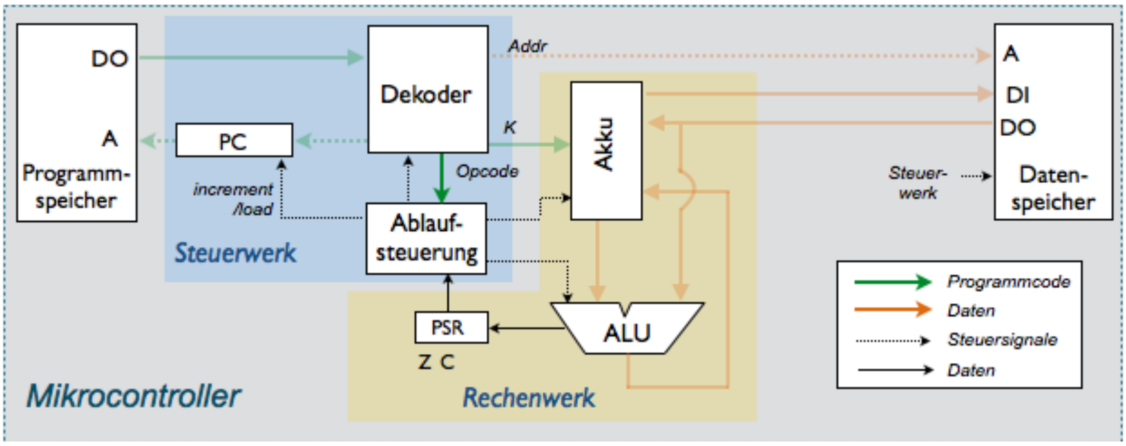
(5) Digilent Analog Discovery Kit (Taschenlabor mit Messeingängen und Signalausgängen, lässt sich als Messplatz und zur Erzeugung von Testsignalen verwenden), siehe Trenz Electronic

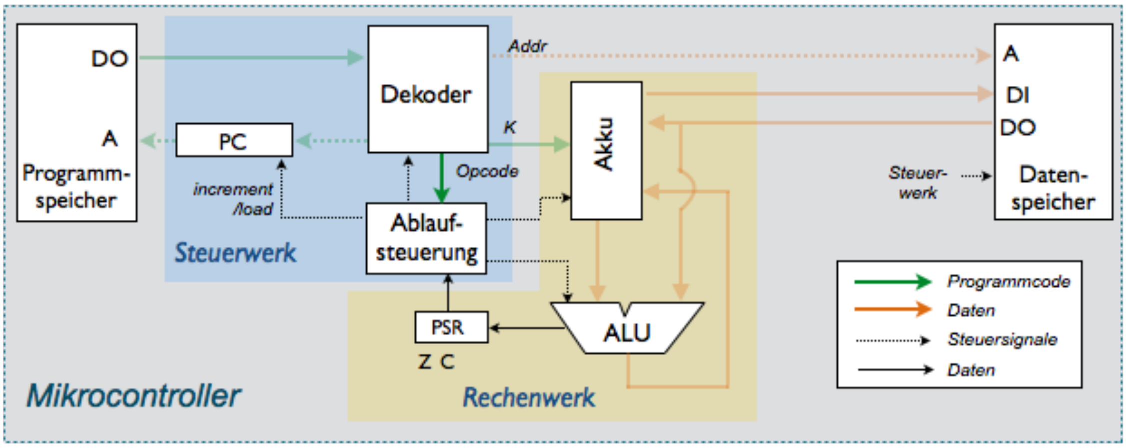
Anhang A - Vorlage für Abläufe im Prozessor

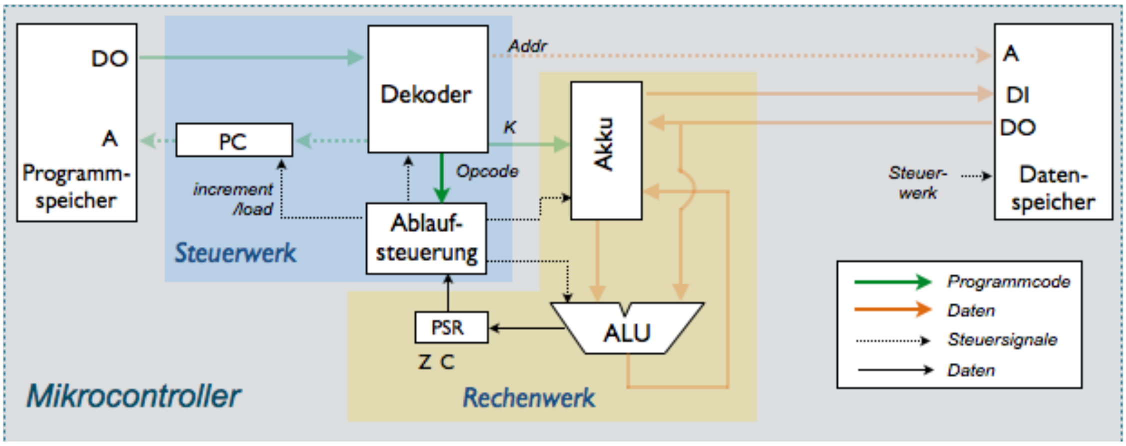
Folgende Vorlagen sind für Skizzen zum Ablauf von Befehlen verwendbar.

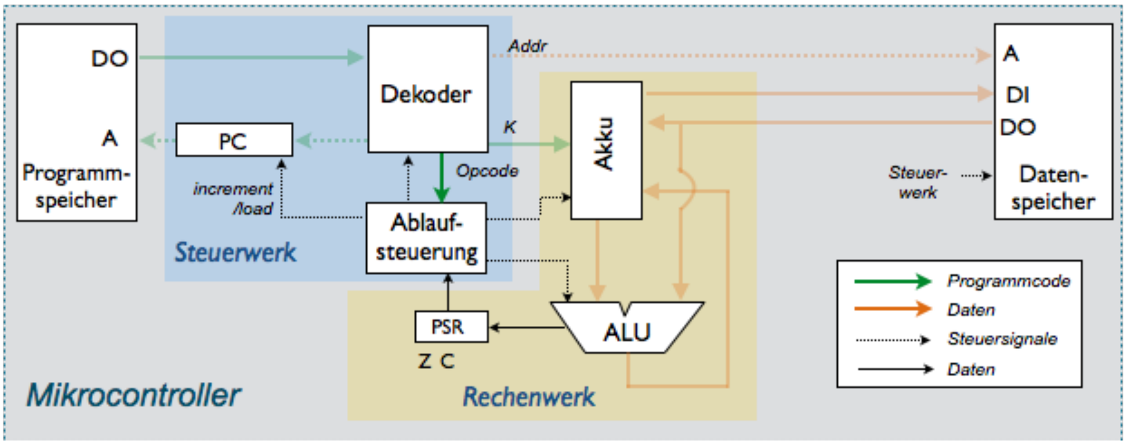


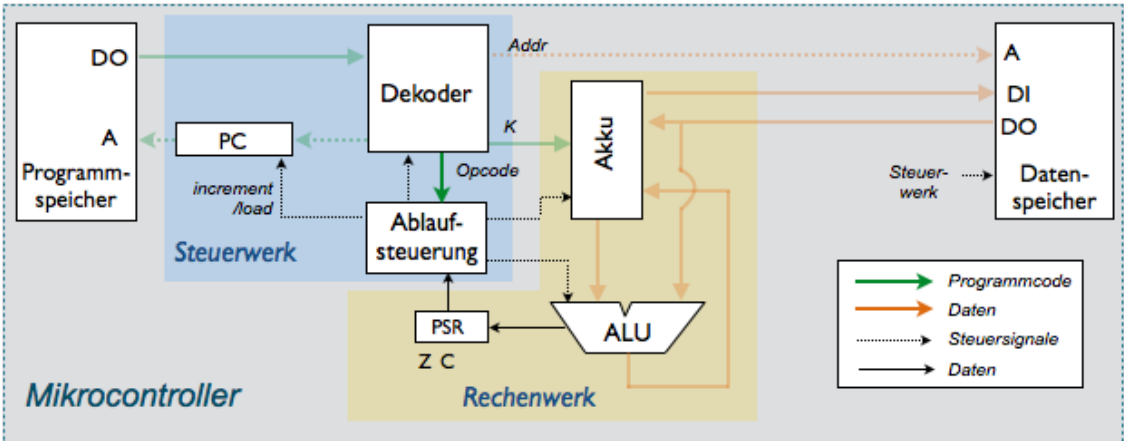


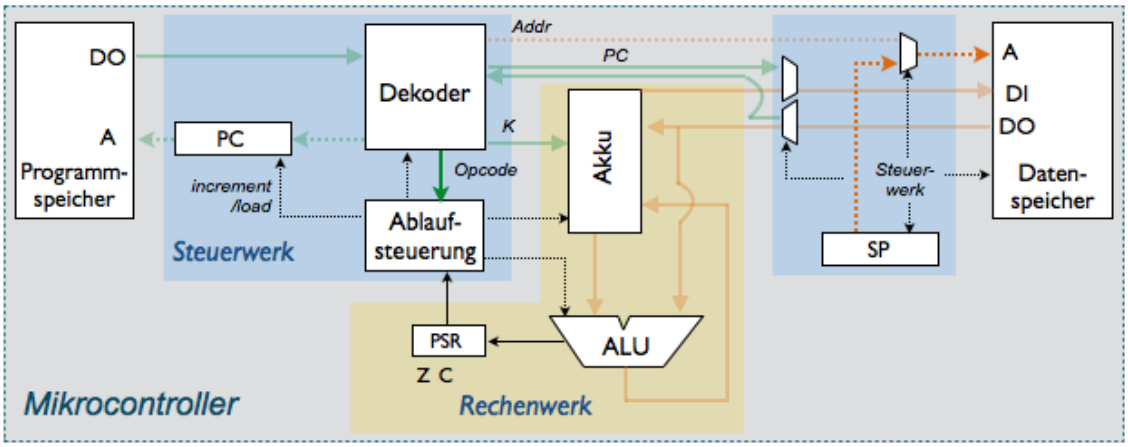


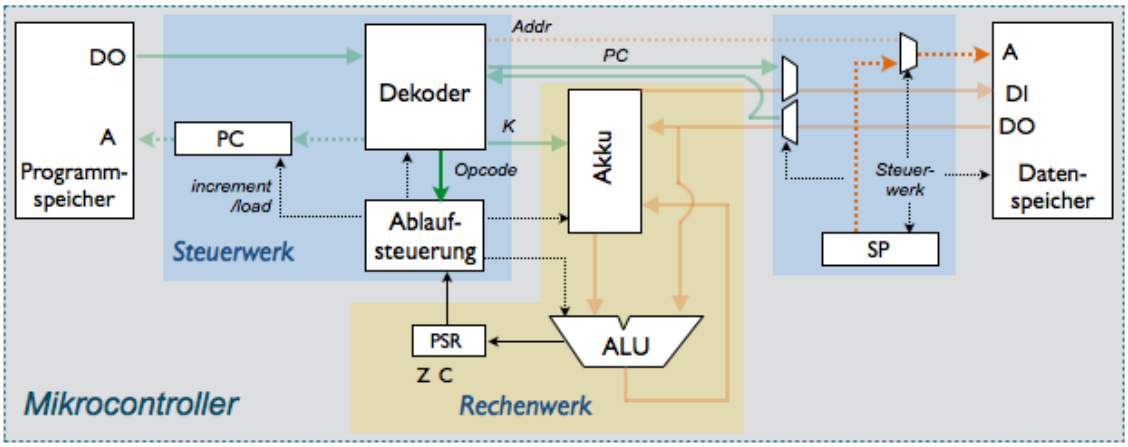

DO
A
Programm-speicher
PC
increment /load
Steuerwerk
Dekoder
Addr
K
Opcode
Ablauf-steuerung
Akku
PSR
Z C
ALU
Rechenwerk
A
DI
DO
Steuer-werk
Daten-speicher
Programmcode
Daten
Steuersignale
Daten
Mikrocontroller

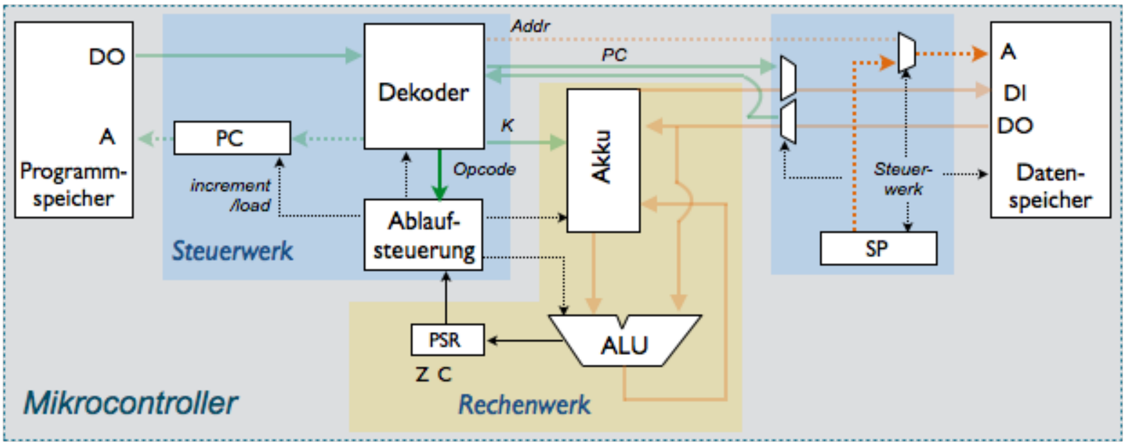

DO
A
Programm-speicher
PC
increment /load
Steuerwerk
Dekoder
Addr
K
Opcode
Ablauf-steuerung
Akku
PSR
Z C
ALU
Rechenwerk
A
DI
DO
Steuer-werk
Daten-speicher
Programmcode
Daten
Steuersignale
Daten
Mikrocontroller

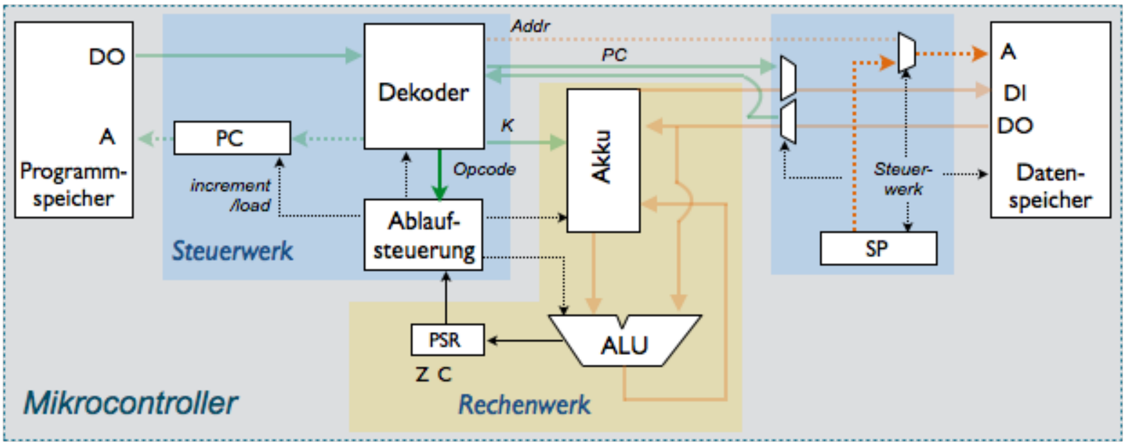

DO
A
Programm-speicher
PC
increment /load
Steuerwerk
Dekoder
Addr
K
Opcode
Ablauf-steuerung
Akku
PSR
Z C
ALU
Rechenwerk
A
DI
DO
Steuer-werk
Daten-speicher
Programmcode
Daten
Steuersignale
Daten
Mikrocontroller

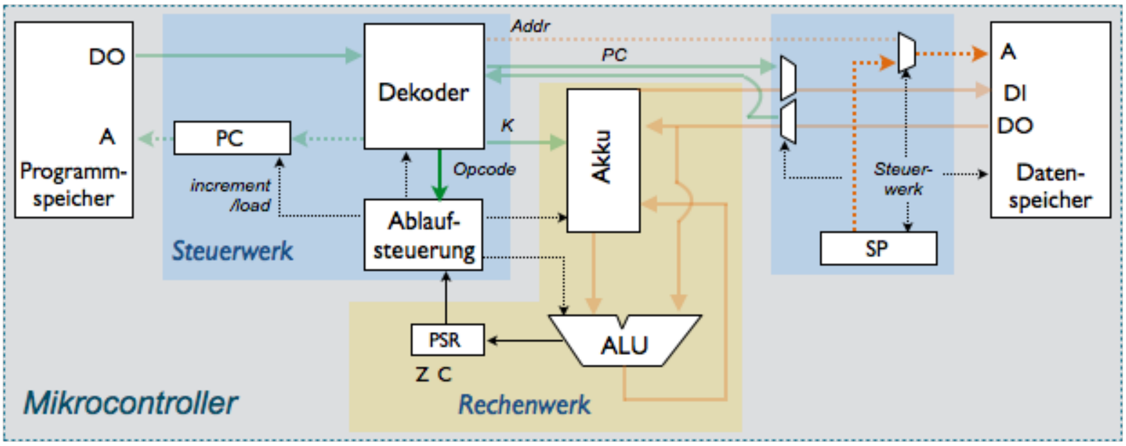

DO
A
Programm-
speicher
PC
increment
/load
Dekoder
Addr
K
Opcode
Akku
Ablauf-
steuerung
Steuerwerk
PSR
Z C
ALU
Rechenwerk
Mikrocontroller
A
DI
DO
Steuer-
werk
Daten-
speicher
Programmcode
Daten
Steuersignale
Daten


DO
A
Programm-
speicher
PC
increment
/load
Dekoder
Addr
PC
K
Opcode
Akku
Ablauf-
steuerung
Steuerwerk
PSR
Z C
ALU
Rechenwerk
Mikrocontroller
A
DI
DO
Steuer-
werk
Daten-
speicher
SP


DO
A
Programm-
speicher
PC
increment
/load
Dekoder
Addr
PC
K
Opcode
Akku
Ablauf-
steuerung
Steuerwerk
PSR
Z C
ALU
Rechenwerk
Mikrocontroller
A
DI
DO
Steuer-
werk
Daten-
speicher
SP


DO
A
Programm-
speicher
PC
increment
/load
Steuerwerk
Dekoder
Addr
PC
K
Opcode
Ablauf-
steuerung
Akku
PSR
Z C
ALU
Rechenwerk
Steuer-
werk
SP
A
DI
DO
Daten-
speicher
Mikrocontroller


DO
A
Programm-
speicher
PC
increment
/load
Steuerwerk
Dekoder
Addr
PC
K
Opcode
Ablauf-
steuerung
Akku
PSR
Z C
ALU
Rechenwerk
Steuer-
werk
SP
A
DI
DO
Daten-
speicher
Mikrocontroller


DO
A
Programm-
speicher
PC
increment
/load
Steuerwerk
Dekoder
Addr
PC
K
Opcode
Ablauf-
steuerung
Akku
PSR
Z C
ALU
Rechenwerk
Steuer-
werk
SP
A
DI
DO
Daten-
speicher
Mikrocontroller

Anhang B - Implementierung des Mikrocontrollers in VHDL

Folgende HDL-Implementierung kann vom Dozenten und ggf. von den Studenten für Demonstrationen der internen Abläufe im Mikrocontroller verwendet werden. Der HDL-Text kann hierzu von der PDF-Vorlage in den HDL-Editor eines Simulationsprogramms kopiert werden (z.B. Modelsim). Die Implementierung enthält folgende Komponenten: Package mit Vereinbarungen für alle Komponenten, Programmspeicher mit Testprogramm für den Mikrocontroller, Datenspeicher, Instruktionsregister mit Programmzähler, Rechenwerk (ALU mit PSR und Akku), Steuerwerk mit Zustandsautomat, Testprogramm.

Package mit Vereinbarungen für alle Komponenten

```
--- Package for the DHBW MCT controller (VHDL)

library ieee;
use ieee.std_logic_1164.all;

package MCT_Pack_uP is
-- types representing opcodes
type OPTYPE is (NOP, LSL, LSR, LDI, LDA, STR, ADD,
  SUB, ANDA, EOR, ORA, JMP, BRBC, BRBS, BCLR, ERR);

-- data type and address type
subtype P_Type is std_logic_vector(15 downto 0);
-- 16-bit of programm code
subtype D_Type is std_logic_vector (7 downto 0);
--  8-bit of data
subtype DA_Type is std_logic_vector(9 downto 0);
-- 10-bit addresses of data memory
subtype PA_Type is std_logic_vector(7 downto 0);
--  8-bit addresses of program memory

-- constants
constant P_Width  : integer := 16;
constant D_Width  : integer := 8;
constant DA_Width : integer := 10;
constant PA_Width : integer := 8;

end MCT_Pack_uP;
```

Programmspeicher mit Testprogramm

```
--- Program_Memory for the DHBW MCT controller (VHDL)
```

```
use work.MCT_Pack_uP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity P_ROM is
       port ( Clk, Pipe_EN : in std_logic; -- Clock
              PADDR : in  PA_TYPE; PDOUT : out P_TYPE);
end P_ROM;

architecture RTL of P_ROM is

 -- array of 2**8 samples, each 16 bits wide (reduced for tests)
 type ROM_array is array(0 to 32) of P_TYPE;

       -- instantiate memory object with sample program
       signal pmemory : ROM_array   := (
         0 => "0001100000101000", -- LDI 5
         1 => "0010100000000000", -- STR $0
         2 => "0010100000010100", -- STR $10
         3 => "0001100000011000", -- LDI 3
         4 => "0011000000000000", -- ADD $0
         5 => "0010100000000010", -- STR $1
         others => "0000000000000000");

 begin
       -- read process
       process (Clk, Pipe_EN) begin
         if (Pipe_EN = '1') then
           if (rising_edge(Clk)) then
       PDOUT <= pmemory(to_integer(unsigned(PADDR)));
      end if;
    end if;
  end process;

end RTL;
```

Datenspeicher

```
--- Data Memory for the DHBW MCT controller (VHDL)

use work.MCT_Pack_uP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
```

```
entity D_RAM is
  port ( Clk, DS_EN, RnW : in std_logic;
         -- Clock, D_RAM Enable, ReadNotWrite
        RWADDR : in  DA_TYPE;
        DATIN  : in  D_TYPE; DTOUT  : out D_TYPE);
end D_RAM;

architecture RTL of D_RAM is

 -- array of 32 samples, each 8 bits wide (reduced for tests)
 type RAM_array is array(0 to 32) of D_TYPE;

       -- instantiate memory object of X_RAM
       signal dmemory : RAM_array  := (others => x"00");

 begin
  -- read & write process
  process (Clk) begin
        if (DS_EN = '1') then
            if (rising_edge(Clk)) then
              if (RnW = '0') then      -- write access
                    dmemory(to_integer(unsigned(RWADDR))) <= DATIN;
              elsif (RnW = '1') then   -- read access
                    DTOUT <= dmemory(to_integer(unsigned(RWADDR)));
              end if;
            end if;
        end if;
  end process;

end RTL;
```

Befehlsdekoder mit Programmzähler

```
-- Instruction Decoder for DHBW MCT controller (VHDL)
-- also includes Program Counter (PC)

use work.MCT_Pack_uP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity IR_DEC is
  port ( Clk: in std_logic; RS : in std_logic; DIN : in P_Type;
         PC_EN, Pipe_EN, PC_LOADA, PC_LOADR : in std_logic;
```

```
            A: out DA_Type; -- addresses of operands
            PSR_Bit : out std_logic_vector(4 downto 0);
            -- bits for branch operation
            PC : out PA_Type; OPCODE : out OPTYPE; K8 : out D_Type);
end IR_DEC;

architecture RTL of IR_DEC is
  signal PCINT : signed(8 downto 0); -- internal program counter;
  signal Offset: unsigned(10 downto 0);

  begin
  decoder : process (Clk, RS) begin
    if (RS = '1') then OPCODE <= NOP; PSR_Bit <= "00000";
    elsif (Pipe_EN = '1') then
        if rising_edge(Clk) then
        case DIN(15 downto 11) is
          when "00000" => OPCODE <= NOP;
          when "00001" => OPCODE <= LSL;
          when "00010" => OPCODE <= LSR;
          when "00011" => OPCODE <= LDI;  K8 <= DIN(10 downto 3);
          when "00100" => OPCODE <= LDA;  A  <= DIN(10 downto 1);
          when "00101" => OPCODE <= STR;  A  <= DIN(10 downto 1);
          when "00110" => OPCODE <= ADD;  A  <= DIN(10 downto 1);
          when "00111" => OPCODE <= SUB;  A  <= DIN(10 downto 1);
          when "01000" => OPCODE <= ANDA; A  <= DIN(10 downto 1);
          when "01001" => OPCODE <= EOR;  A  <= DIN(10 downto 1);
          when "01010" => OPCODE <= ORA;  A  <= DIN(10 downto 1);
          when "01011" => OPCODE <= JMP;
                          Offset<=unsigned(DIN(10 downto 0));
          when "01100" => OPCODE <= BRBC; PSR_Bit <= DIN(10 downto 6);
                          Offset<=unsigned(DIN(10 downto 0));
          when "01101" => OPCODE <= BRBS; PSR_Bit <= DIN(10 downto 6);
                          Offset<=unsigned(DIN(10 downto 0));
          when others  => null;
        end case;
      end if;
      end if;
  end process decoder;

  set_internal_PC : process (Clk, RS)
    begin
    if (RS = '1') then PCINT <= (others => '0');
    elsif rising_edge(Clk) then
      if (PC_EN = '1') then
        if (Pipe_EN = '1') then
            PCINT <= PCINT + 1;
```

```
            elsif (PC_LOADA = '1') then
                PCINT <= signed('0' & Offset(7 downto 0));
            elsif (PC_LOADR = '1') then
                if (Offset(5) = '1') then
                  PCINT <= PCINT - signed("0000" & Offset(4 downto 0));
                elsif (Offset(5) = '0') then
                  PCINT <= PCINT + signed("0000" & Offset(4 downto 0));
                end if;
            end if;
          end if;
        end if;
      end process set_internal_PC;

      PC <= std_logic_vector(PCINT(7 downto 0));

    end RTL;
```

Rechenwerk (ALU mit PSR und Akku)

```
    --- ALU, PSR and Akku for the DHBW MCT controller (VHDL)

    use work.MCT_Pack_uP.all;
    library ieee;
    use ieee.std_logic_1164.all;
    use IEEE.numeric_std.all;

    entity ALU_PSR_Akku is
      port ( OPCODE : in OPTYPE; K, DS_DO : in D_Type;
             Clk, ACC_EN : in std_logic;
             DS_DI: out D_Type; CFlag, ZFlag, NFlag: out std_logic :='0');
    end ALU_PSR_Akku;

    architecture RTL of ALU_PSR_Akku is
    signal Zero : D_Type := (others => '0');
    signal Akku, RdOut : D_Type := (others => '0');

    begin

      process (OPCODE, K, DS_DO, Akku)
        variable RsVar, RdVar, RdoutVar : D_Type;
        variable RVar : signed(D_Width downto 0); -- 9 bits wide
        begin
          RsVar := DS_DO; -- copy Rs from data memory
          RdVar := Akku;  -- copy Rd from Akku
```

```
          RdoutVar := Zero; -- avoids latches in synthesis

          case OPCODE is
             when LSL => CFlag <= RdVar(D_Width-1); -- msb
                RdOutVar(D_Width-1 downto 1) := RdVar(D_Width-2 downto 0);
                         RdOutVar(0):='0';          -- lsb
             when LSR => CFlag <= Akku(0);          -- lsb
                RdOutVar(D_Width-2 downto 0) := Akku(D_Width-1 downto 1);
                         RdOutVar(D_Width-1):='0';   -- msb
           when ADD => RVar := signed('0' & RsVar) +  signed('0' & RdVar);
                  RdOutVar := std_logic_vector(RVar(D_Width-1 downto 0));
                  NFlag <= RVar(D_Width);
           when SUB => RVar := signed('0' & RsVar) -  signed('0' & RdVar);
                  RdOutVar := std_logic_vector(RVar(D_Width-1 downto 0));
                  NFlag <= RVar(D_Width);
           when ANDA=> RdOutVar := RsVar AND RdVar;
           when EOR => RdOutVar := RsVar XOR RdVar;
           when ORA => RdOutVar := RsVar OR RdVar;
           when STR => DS_DI <= Akku;
           when LDA => RdOutVar := DS_DO;
           when LDI => RdOutVar := K;
           when  others => null; -- no action
        end case;
        if (RdOutVar = Zero) then ZFlag <='1'; end if;
        RdOut <= RdOutVar;
      end process;

      load_akku : process (Clk) begin
       if (ACC_EN = '1') then
          if rising_edge(Clk) then
              Akku <= RdOut;
          end if;
        end if;
      end process load_akku;

    end RTL;
```

Steuerwerk mit Zustandsautomat

```
    --- FSM and Top Module for the DHBW MCT controller (VHDL)

    use work.MCT_Pack_uP.all;
    library ieee;
    use ieee.std_logic_1164.all;
    use IEEE.numeric_std.all;
```

```
entity FSM_Top_Module is
        port ( Clk, RS: in std_logic);
end FSM_Top_Module;

architecture RTL of FSM_Top_Module  is

component P_ROM is
       port ( Clk, Pipe_EN  : in std_logic; -- Clock, Pipeline enable
              PADDR : in  PA_Type;
              PDOUT : out P_Type);
end component;

component D_RAM is
       port ( Clk, DS_EN, RnW  : in std_logic;
              -- Clock, D_RAM enable, Read not Write
              RWADDR : in  DA_Type;
              DATIN : in D_Type;
              DTOUT : out D_Type);
end component;

component IR_DEC is
       port ( Clk: in std_logic; RS : in std_logic; DIN : in P_Type;
              PC_EN, Pipe_EN, PC_LOADA, PC_LOADR : in std_logic;
              A : out DA_Type; -- addresses of operands
              PSR_Bit : out std_logic_vector(4 downto 0);
              PC : out PA_Type; OPCODE : out OPTYPE; K8 : out D_Type);
end component;

component ALU_PSR_Akku is
       port (OPCODE : in OPTYPE; K, DS_DO : in D_Type;
             Clk, ACC_EN : in std_logic;
             DS_DI  : out D_Type; CFlag, ZFlag, NFlag: out std_logic);
end component;


-- Top_Module internal signals

type state_type is (Z1, Z2);
-- Z1: operation executes in one clock cycle,
-- Z2: operation executes in two clock cycles

signal next_state, current_state: state_type :=Z1;

-- to interconnect modules incl. controls
signal T_OPCODE : OPTYPE;
```

```
signal T_PC : PA_TYPE;
signal T_A : DA_TYPE;
signal T_DIN : P_TYPE;
signal T_DI, T_DO, T_K8 : D_Type;
signal T_Zero, T_Carry, T_Neg : std_logic;
signal T_PSR : std_logic_vector (4 downto 0);
signal T_Pipe_EN, T_PC_EN, T_DS_RnW : std_logic :='1';
signal T_DS_EN, T_PC_LOADA, T_PC_LOADR, T_ACC_EN : std_logic :='0';

begin

-- instantiate and connect modules to Top Module
PMem: P_ROM port map (Clk => Clk, Pipe_EN=> T_Pipe_EN,
                      PADDR => T_PC, PDOUT => T_DIN);

DMem: D_RAM port map (Clk => Clk, DS_EN=> T_DS_EN, RnW => T_DS_RnW,
                      RWADDR => T_A, DATIN => T_DI, DTOUT => T_DO);

IR_Decoder: IR_DEC port map (Clk => Clk, RS => RS, DIN => T_DIN,
            Pipe_EN => T_Pipe_EN, PC_EN => T_PC_EN,
            PC_LOADA => T_PC_LOADA, PC_LOADR => T_PC_LOADR,
            A => T_A, PSR_Bit => T_PSR, PC => T_PC,
            OPCODE => T_OPCODE, K8 => T_K8);

Alu : ALU_PSR_Akku port map (OPCODE => T_OPCODE, K => T_K8,
            DS_DO => T_DO, Clk => Clk, ACC_EN => T_ACC_EN,
            DS_DI => T_DI, ZFlag => T_Zero, CFlag => T_Carry,
            NFlag => T_Neg);

-- run processes
update_state_register: process(Clk, RS)
  begin
       if (RS='1') then
         current_state <= Z1;
       elsif rising_edge(Clk) then
         current_state <= next_state;
       end if;

end process update_state_register;

-- state transitions & actions (single logic)
logic_next_state_and_actions: process(current_state, T_OPCODE)

  begin

    case current_state is
```

```
        when Z1 => -- one clock cycle per operation

            case T_OPCODE is
               when NOP => T_PC_EN<='1'; T_Pipe_EN<='1'; T_DS_EN<='0';
                           T_PC_LOADA<='0'; T_PC_LOADR<='0';
                           T_ACC_EN<='0';

               when LSL | LSR | LDI => T_Pipe_EN<='1'; T_PC_EN<='1';
                                       T_ACC_EN<='1'; T_DS_EN<='0';

               when STR => T_Pipe_EN<='1'; T_PC_EN<='1';
                           T_DS_EN<='1'; T_DS_RnW<='0'; T_ACC_EN<='0';

               when ADD | SUB | ANDA | EOR | ORA | LDA  =>
                            T_Pipe_EN<='0'; T_ACC_EN<='0';
                            T_DS_EN<='1'; T_DS_RnW<='1';
                            next_state<=Z2;  -- 2nd cycle

               when JMP =>  T_Pipe_EN<='0'; T_PC_EN<='1'; T_PC_LOADA<='1';
                            next_state<=Z2;  -- 2nd cycle

               when BRBC =>
                    case T_PSR is
                    when "10000" =>
                         if (T_Zero = '0')  then
                             T_Pipe_EN<='0'; T_PC_LOADR<='1';
                              next_state<=Z2;
                         elsif (T_Zero ='1') then
                             T_Pipe_EN<='1'; T_PC_LOADR<='0';
                         end if;

                    when "01000" =>
                         if (T_Carry = '0') then
                             T_Pipe_EN<='0'; T_PC_LOADR<='1';
                             next_state<=Z2;
                         elsif (T_Zero ='1') then
                             T_Pipe_EN<='1'; T_PC_LOADR<='0';
                         end if;

                    when "00001" =>
                         if (T_Neg = '0') then
                             T_Pipe_EN<='0'; T_PC_LOADR<='1';
                             next_state<=Z2;
                         elsif (T_Neg ='1') then
                             T_Pipe_EN<='1'; T_PC_LOADR<='0';
```

```
                    end if;
                when others => null;
                end case;

            when BRBS =>
                case T_PSR is
                when "10000" =>
                    if (T_Zero = '1')  then
                        T_Pipe_EN<='0'; T_PC_LOADR<='1';
                        next_state<=Z2;
                    elsif (T_Zero ='0') then
                        T_Pipe_EN<='1'; T_PC_LOADR<='0';
                    end if;

                when "01000" =>
                    if (T_Carry = '1') then
                        T_Pipe_EN<='0'; T_PC_LOADR<='1';
                        next_state<=Z2;
                    elsif (T_Zero ='0') then
                        T_Pipe_EN<='1'; T_PC_LOADR<='0';
                    end if;

                when "00001" =>
                    if (T_Neg = '1') then
                        T_Pipe_EN<='0'; T_PC_LOADR<='1';
                        next_state<=Z2;
                    elsif (T_Neg ='0') then
                        T_Pipe_EN<='1'; T_PC_LOADR<='0';
                    end if;

                when others => null;
                end case;

           when others  => null;
       end case; -- opcode one cycle

    when Z2 => -- second clock cycle per operation

      case T_OPCODE is

        when BRBS | BRBC| JMP => -- re-animate pipeline
            T_PC_EN<='1'; T_Pipe_EN<='1';
            T_PC_LOADR<='0'; T_PC_LOADA<='0';
            next_state<=Z1;

        when ADD | SUB | ANDA | EOR | ORA | LDA  =>
```

```
                    T_ACC_EN<='1'; T_DS_EN<='0'; -- enable akku for result
                    T_Pipe_EN<='1';     -- re-animate pipeline
                    next_state<=Z1;

              when others  => null;
            end case; -- opcode two cycles

       when others => null;
     end case; -- states

end process logic_next_state_and_actions;

end RTL;
```

Test Program

```
--- Testbench for FSM and Top Module of DHBW MCT controller (VHDL)

use work.MCT_Pack_uP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity test_SP is
end test_SP ;

architecture Behavioural of test_SP  is

component FSM_Top_Module is
   port(Clk, RS : in std_logic);
end component;

-- Test bench internal signals
signal T_Clk, T_RS : std_logic :='0';

begin
-- connect FSM_Top_Module to testbench
PCT1: FSM_Top_Module port map (Clk=>T_Clk, RS=>T_RS);

-- run tests
reset : process begin
   T_RS <= '1'; wait for 5 ns; T_RS <= '0'; wait;
end process reset;

clock : process begin
```

```
        T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
    end process clock;

    end Behavioural;
```