Mikrocomputer

INF 5T

Wintersemester 2004/2005

Inhaltsverzeichnis

1 Einführung

1.1 Prüfung und Literatur

1.2 Architektur eines Mikrocomputers

Derivat: Halbleiter-Hersteller bieten Controller-Fam. an, welche auf einer CPU basieren (= ein Maschinenbefehlsatz), jedoch mit unterschiedlichen Peripherieaussattung (on-chip).

Ausgewählte Herstellerliste:

Hersteller	*Produkt*
Infineon	8051-Fam. 8-Bit C167-Fam. 16-Bit (Einsatz im Praktikum) TriCore 32-Bit
Motorola	68 ... 05 / 08 / 12 PowerPC 32-Bit
Philips	8051 XA
MicrochipTech	PIC
Atmel	8051 AVR ARM 32-Bit (sehr beliebt bei Handys)
Intel	ARM (-> XSCALE) 8051
Penesas	H8S, M16 bis 32-Bit
Zilog	Z80

1.3 Algorithmus

1.3.1 Algorithmusentwicklung

Nicht Thema dieser Vorlesung

1.3.2 Möglichkeiten der Algorithmusimplementierung

PROZESSOR mit Programmspeicher	Fest verdrahtete Logik	Programmierbare Logik
Programm schreiben in C, C++, Java, Delphi, ...		VHDL (Sprache)
Kompilieren / Übersetzen		Synthese
Maschinenprogramm, relative object-files	Schaltplan / Schematic	
Linken / Binden benötigt Speicheradressen des mC-Systems	Platzieren und Verdrahteten	

PROZESSOR mit Programmspeicher	Fest verdrahtete Logik	Programmierbare Logik
Maschinenprogramm mit physikalischen Adressbereichen	Layout, Lageplan	
Laden (serielle Schnittstelle)	Film, Platinenhersteller, Bestücken (z.B. löten)	
Baugruppe mit programmierten Programmspeicher	Fertige Baugruppe	
• Standard-Baustein • billig • flexibel programmierbar	• statische lösung • zeit- & kostenintensiv	• sehr kostenintensiv • komplex • nicht jedes VHDL-Programm ist realisierbar

1.4 Informationsabbildung und -darstellung

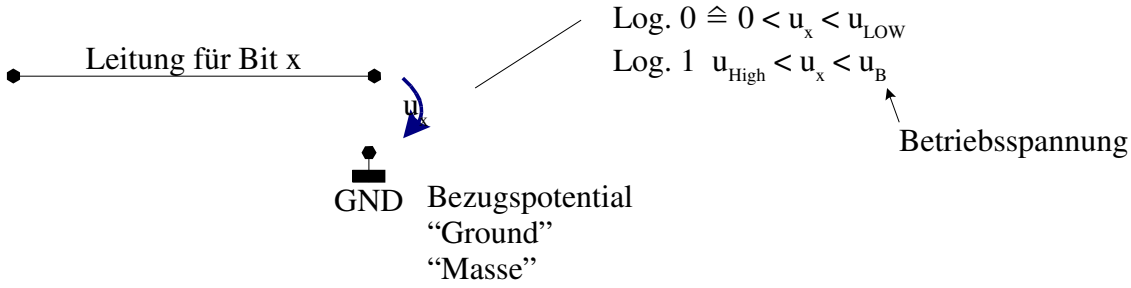
1. Abbildung
2. Darstellung
   analoge: zeitkontinuierlich
   wertkontinuierlich
   digitale: zeitdiskret
   wertdiskret, wertbegrenzt

   Vorteile:
   - einfache Darstellung durch binäre Zustände
   - einfach speicherbar
     z.B. Kondensator geladen / ungeladen
     Flipflop
   - störsicher

   Gruppierung von Bits:

4 Bit	Nibble	
8 Bit	Byte	
16 Bit	Word	int
32 Bit	long word	

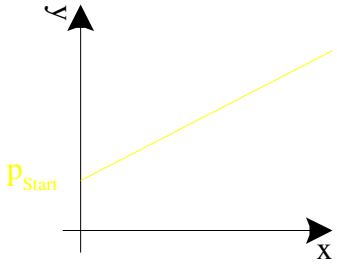
3. Binäre Zustände in der Digitaltechnik
   a) Low-High Potential



   b) Zeitverzögerung

c) Leistung
Leistungsverbrauch logischer Gatter

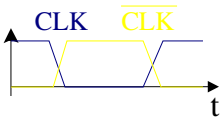
$$p = \underbrace{f_{CLU}}_{\text{Taktfrequenz}} \cdot \underbrace{c}_{\text{getaktete Kapazität bzw. Chipfläche}} \cdot \underbrace{u_B^2}_{\text{Betriebsspannung}} + p_{Start}$$



1.5 Logikfamilien

Unterscheidungsmerkmale

- Halbleitertechnologie (CMOS, Bipclon, BICMOS, ...)
- Betriebsspannung
- Low-High Pegel
- Leistungsaufnahme
- Geschwindigkeit
- Treiberleistung
- single-ended differential



- Terminierung

1.6 Logikbausteine

1.6.1 Bezeichnungssystematik

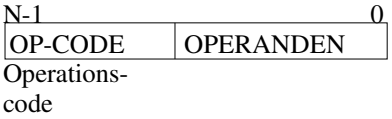
1.6.2 Beispiele der 74er Serie

2 CPU

2.1 Grundlagen Befehlsabarbeitung und Maschinensprache

2.1.1 Prinzip Maschinenprogramm

Aufbau eines Maschinenbefehls



2.1.2 Befehlsanordnung im Speicher

Jede Speicherzelle hat eine Adresse: z.B. von 0x0000 bis 0xFFFF; jede Stelle stellt dabei 4 Bit zu, also ist in dem Beispiel der Gesamtadressraum 16 Bit

Mit den Adressen werden BYTE adressiert, nicht WORDs. D.h. bei 16bit und 32bit Prozessoren werden immer Bytes adressiert, auch wenn 2 bzw. 4 Byte auf einmal ausgelesen werden (können).
=> Adressen werden (meist) byteweise gezählt

Bem.	***Adresse***	***Inhalt***	***Assembler***
CPU holt ersten Befehl nach Reset	0x0000	Befehl 1	NOP
	0x0001	Befehl 2	JUMP
Zieladresse des Sprunges (low Byte)	0x0002	ALB	
Zieladresse des Sprunges (hight Byte)	0x0003	AHB	
	...		
Zieladresse	0x00..	Befehl 3	

2.1.3 Maschinenprogramm

C-Beispielprogramm:

```
unsinged int a, b[5], c, i;  // 16-bit
a = 0;
c = 2;

for( i = 5; i > 0; i-- ) {
  a = a + b[i];
  a = a * c;
}
```

2.1.4 Prinzip Maschinenprogramm

Folgerungen:

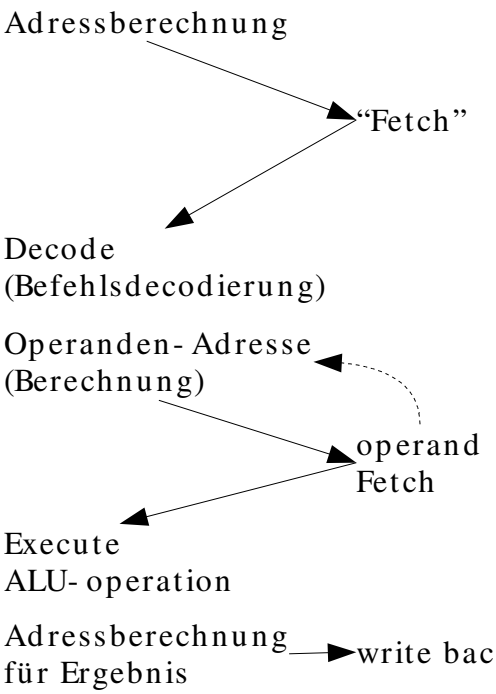
- Maschinenbefehle sind aufgebaut aus:
  - Operations-Code „OpCode“
  - Operanden
- interne Register ermöglichen es dem Compiler, das Maschinenprogramm effizienter zu gestalten:
  - effiziente Adressieren (da Adressen wesentlich kürzer)
    => kürzerer Code, schnellere Ausführung
  - schnellerer Zugriff als auf (externen) Speicher

- wird ein Befehl falsch interpretiert (durch Störung), kann der nachfolgende Operand als Befehl interpretiert werden oder umgekehrt
  => die CPU schmiert ab
- Befehlsabarbeitung besteht aus mehreren Phasen

2.1.5 Befehlszyklus

effiziente Abarbeitung durch „Pipelining“

Unit \ Cycle	*i*	*i+1*	*i+2*	*i+3*	*i+4*	*i+5*
Fetch	N	N+1	N+2	N+3	N+4	N+5
Decode		N	N+1	N+2	N+3	N+4
Operand Fetch			N	N+1	N+2	N+3
Execute				N	N+1	N+2
Write Back					N	N+1



2.2 Programmiermodelle

2.2.1 Blockdiagramm CPU

2.2.2 Registerstrukturen

1. Daten- und Adressenregister
2. Kontrollregister
   a) Steuerbits für ALV
   b) Interupt-Control-Register SER

2. Steuerregister
   a) PSW
      Statusbits der ALV
      aktuelle Priorität
      Registersatz

Warum Register für Adressierung und Operanden?

- effiziente Adressierung
  mehrere Registeradressen (Registerindex)
  können in einem Maschinenbefehl codiert werden
- schneller Zugriff auf interne Register
- Optimierungsmöglichkeit für Compiler

2.2.3 Adressierung Operanden

Prüfungsfrage: Nennen Sie drei Adressierungsarten und erläutern Sie diese kurz.

Beschreibungen siehe Folien

1) Immediate
2) Memory direct
3) Register direct
4) Register indirect
5) Base Offset
6) Zirkuläre Adressierung

2.2.4 Adressierung Code

Unterschieden wird

1) absolute Adressierung
2) relative Adressierung
3) direkte Adressierung
4) indirekte Adressierung

Die (physikalische) Adressierung kann erfolgen über

1) einen Instruction Pointer
2) mehrere Instruction Pointer → Segmentierung
   → Banking

2.2.5 Grundstrukturen C-Assembler

2.3 Betriebsarten der CPU

2.3.1 Normaler Programmablauf

```
int main( void )
{
  Initialisieren();                                  CALL Init
  while( 1 ) {          /* Polling-Schleife */
    EingängeLesen();    /* !! no busy waiting !! */  L1: CALL Input
    AusgangszuständeBerechnen();                     CALL Comp
    AusgängeSchreiben();                             CALL Output
  }                                                  JUMP L1
}
```

Adresse der Routine "Initialisieren"

Main-Loop
Polling-Schleife oder Hauptschleife

Enthält keine „busy-waitings“ (aktives Warten auf Eingaben).

Typischerweise werden in der Pollingschleife Benutzereingaben und Displayausgaben bearbeitet oder Ereignisse bearbeitet, welche Bearbeitungszeiten bis ≈ 50ms tolerieren.

Im Allgemeinen läßt sich eine maximale Durchlaufzeit der Pollingschleife ermitteln → falls der Controller nur die Pollingschleife bearbeitet, wäre diese „echtzeitfähig“ bis zu dieser Durchlaufzeit.

Echtzeitfähig = maximale Rechenzeit wird garantiert.

Ereignisse, welche eine schnellere Bearbeitung erfordern als die Pollingschleife Laufzeit hat, werden in Interruptfunktionen behandelt.

2.3.2 Interrupt-Modus

2.3.2.1 Begriffe

Unterscheidung „Exceptions“.

- Asynchron zum Programmlauf (externe Ereignisse)
- Synchron zum Programmlauf (Stackover- / -underflow, ...)
- maskierbar (d.h. kann die Exception deaktiviert werden)
- Priorität

Einteilung nach Filk/Liebig:

- Traps: synchron zum Programmlauf
- Interrupts: asynchron

Einteilung Infineon C166-Serie:

- Traps: nicht maskierbar
  – Hardware-Traps (Reset, NMI (Non Maskable Interrupt), Bufferoverflow, ...)
  – Software-Traps
- Interrupts: maskierbar, Priorität einstellbar (im wesentlichen die asynchronen HW-Ereignisse)

2.3.2.2 Abarbeitung von Interrupts

– Prioritäten

– Aufruf der Interrupt-Servicefunktion

– Interrupt-Latenzzeit

Prüfungsfrage: Was passiert bei einem Interrupt? (siehe Folie „Aufruf der Interrupt-Service...“

2.3.2.3 Interrupt Control Register

<Folie „Aufbau Interrupt Control Register“ und „Programm Status Word ...“>

z.B. T3IC = 88; //dezimal
0x48; //hexadezimal

⇒ enable flag und Priorität 6 wurden gesetzt

IEN = 1 // IEN bitadressierbar
// im Headerfile hinterlegt

T3IE = 0; // → T3-Interrupt disabled

2.3.2.4 Prinzipielle Programmstruktur mit Interrupt-Servicefunktion

```
#include <reg161.h>

int main( void )
{
  T3IC=0x48;
  IEN=1;
  ...      //restliche Initialisierungen
  while( 1 ) {
```

```
    EingängeLesen();
    Zustände();
    Ausgänge();
  }
}


/* Interrupt-Servicefunktion
void Timer3( void ) interrupt 0x23 {  /* 0x23 ist die Interrupt-Nr. */
  ...
}
```

2.3.3 Power-Down Modi

Stromspar-Modi können durch verschiedene Maßnahmen eingeleitet werden.

- Abschalten von Peripheriemodulen, welche (momentan) nicht benötigt werden
- Verringerung der Taktfrequenz (durch Konfigurationsbits)
- CPU abschalten (Aufwecken z.B. durch Interrupt)

spezielle Befehle, z.B. SLEEP, IDLE, PWRDOWN

2.3.4 DMA-Modus

„Direct Memory Access“
CPU übergibt die Kontrolle über den Speicher an ein Peripheriegerät.

3 Peripheriemodule

3.1 Digitale Ports

Ports sind Anschlüsse des Controller-Chips für Steueraufgaben. Es gibt zwei Typen:

- Digitale Eingangsports
  Eingangsspannung am Port-Pin wird in eine logische 0 oder 1 verwandelt (→ Electrical Characteristics);
- Digitaler Ausgangsport
  Logische 0 oder 1 wird in ein Low- oder High-Potential von einem Pin umgesetzt.

Typischerweise werden die Portpins Portregistern zugeordnet.

```
P2          16-Bit      0xFFC0      physikalische Adresse
...
VAR = *((unsigned int volatile *) 0xFFC0);
```

Im (Keil)-C erfolgt der Zugriff über sogenanntes SFR (eng. Special Funktion Register). Diese Funktionen und andere Werte sind in der spezifischen Header-Datei zum jeweiligen Controller hinterlegt (z.B. #include <reg161.h>)

Beispiel:

```
SFR         P2          0xFFC0
```

- Umschaltung zwischen digitalem Eingangsport und Ausgangsport erfolgt typischerweise über Konfigurationsregister
  z.B. beim Port P2 mittels Registers DP2

Beispiel:

z.B.

...	4	3	2	1	0	
	0	1	1	0	0	DP2
...	...	...	Output	Input	Input	P2

in C. Hinweis: DP2 heißt Direction Port 2

```
DP2 = 0x0000;  /* alles Eingänge */
Variable = P2;
DP2 = 0xFFFF;  /* alles Ausgänge */
P2 = Variable;
```

3.2 Timer und Counter

3.2.1 Grundaufbau

Wichtigste Verwendungszwecke:

- Zeitbasis erzeugen (unabhängig von der CPU-Clock, denn wir wollen vielleicht nur Milli-Sekunden oder Sekunden für die Messtechnik verwenden)
- Ereignisse zählen

Beschreibung zu Folie <mc_kap_3-2_timer>, Seite 1:

1. Wir haben eine CPU. Diese CPU ist MHz-getaktet, typischerweise mit 16 und 20 MHz. Wir könnten also die CPU zählen lassen. Das Problem ist nur, dass der Prozessor 16 Bit breit ist und somit max. bis $2^{16} = 65535$ zählen kann.

2. Ersatz: ein Vorteiler (engl. Prescaler) teilt die Frequenzen in 2er Pozenzen (also 2^0, 2^1, ..., 2^{15}). Den Wert des Teilers (also 0 – 15) liest der Vorteiler aus dem Vorteiler-Register (engl. Prescale-Register). Dieses Register kann über den Bus beschrieben werden.
3. Der n-Bit-Zähler (engl. n-Bit-Counter) zählt die Impulse, die der Vorteiler abgegeben hat. Aber nur wenn der Counter überläuft, wird ein Interrupt-Request-Flag gesetzt. Wenn wir also früher einen Interrupt auslösen wollen, dann müssen wir den Counter bei einem Wert != 0 starten lassen, und schon läuft er eher über!
4. Zeitbasiserzeugung nur mit Zählerregister. Laden in Interrupt-Servicefunktion

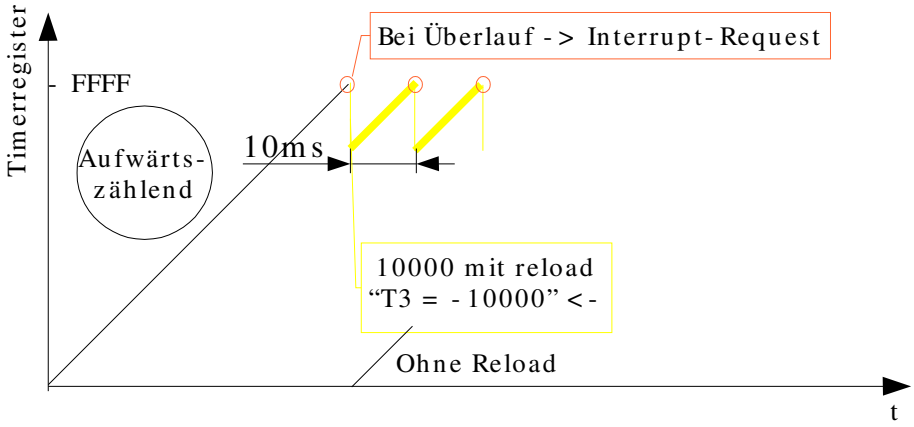
```
void Timer interrupt 0x...
{
 TIM = Startwert;    // -10000
 ...
}
```

   nicht exakt, da Aufruf Zeit benötigt

   genauer:

```
TIM = TIM + Startwert;  // bei uns: TIM - 10000
```

5. separates Reload-Regster; kein Code in der Service-Routine nötig. Das Reload-Register speichert also den Initialwert des Counters, für die nächste Zählung, nachdem ein Interrupt ausgelöst wurde
6. Wenn ein Interrupt angefordert wird, ist TFLag = 1, ansonsten 0.
7. Der rechte Schalter dient dient zum An- und Ausschalten der Zeitzählung
8. Der linke Schalter bestimmt den Modus, also ob der Zähler raufzählen soll (i++) oder runter zählen soll (i--).
9. Gate-Modus: zählen, nur wenn high ist

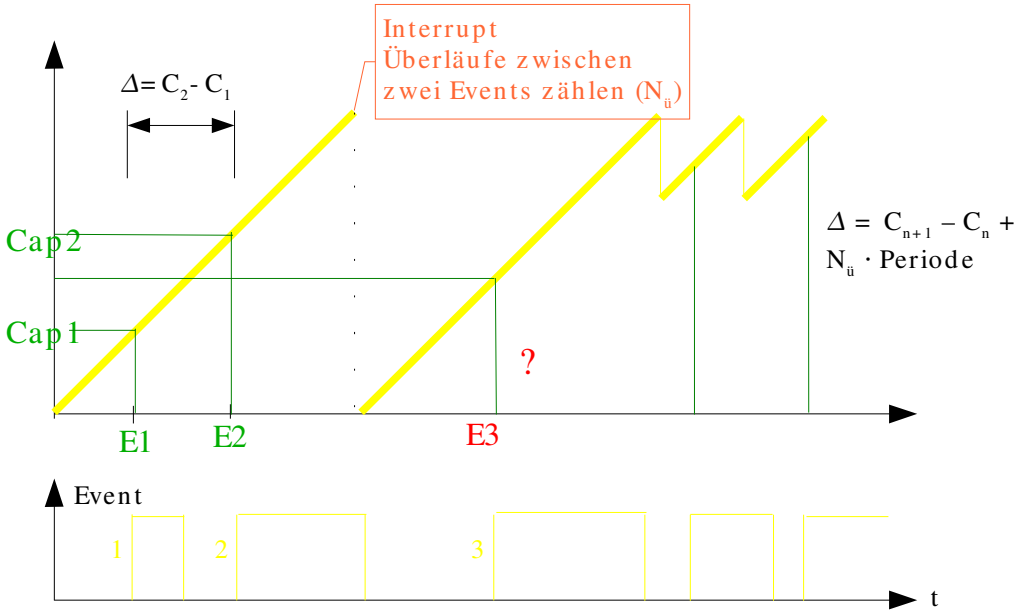


Prescaler 1/16 → Eingangstakt Timer = 1MHz
gewünscht: 10ms Überlaufperiode 10000 * zählen

3.2.2 Capture (Beschreibung zu Folie <mc_kap_3-2_timer>, Seite 2):

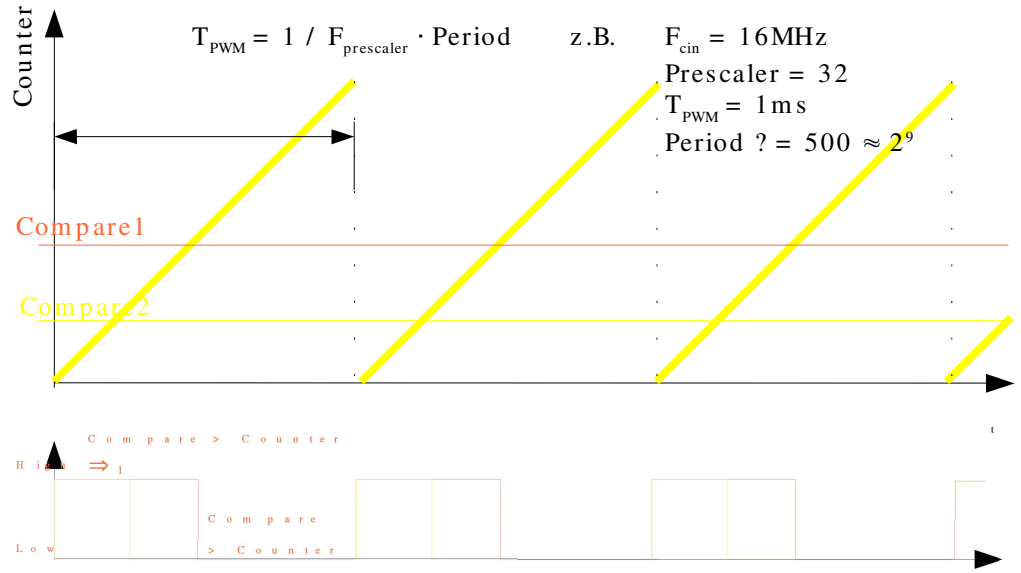
1. Bei einem Ereignis wird der Counter-Wert in das Capture-Register kopiert (damit man also feststellen kann, wie viel Zeit bis zu diesem Ereignis vergangen ist).
   Beispiel: Ereignis E1 erzeugt Cap1, Ereignis E2 erzeugt Cap2. Wie groß ist die Zeitdifferenz?

   Lösung: $\Delta t = (cap2 - cap1) \cdot \frac{Vorteiler}{F_{CPU}}$
2. Wann wird reagiert? Z.B. nur bei positiven Flanken



3.2.3 Pulsweitenmodulation (PWM) (Beschreibung zu Folie <mc_kap_3-2_timer>, Seite 3):

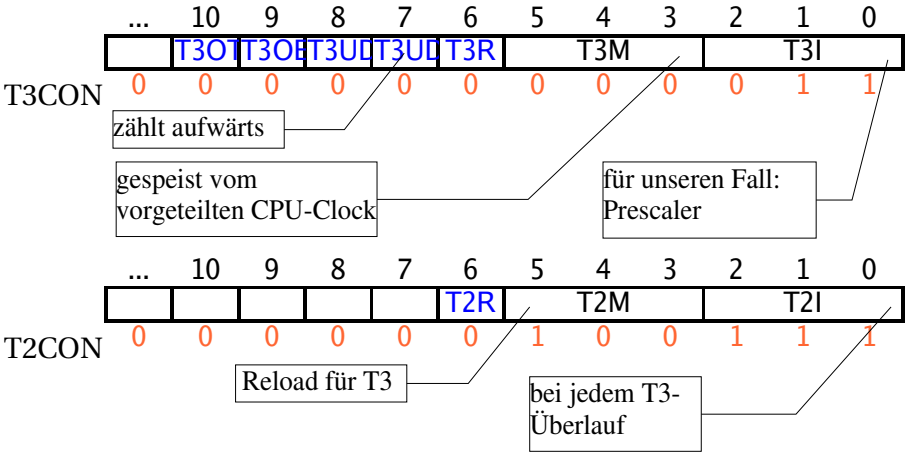
1. Wir führen jetzt einen Compare-Wert ein
2. Unterhalb des Compare-Wertes ist COUT high (1), oberhalb low (0).
3. COUT ist also ein Pulsweitensignal, das angibt, wann ein Ereignis eingetreten ist (wo also der Compare-Wert lag).
4. $Periodendauer = \frac{Vorteiler}{F_{CPU}} \cdot Rücksetzwert$



3.2.4 Vorlesungsübung 2

Zeitbasis = 10 ms

$F_{CPU} = 16\,MHz$ -> soll 4 Micro-Sek. Periode haben => 250 kHz



T3-Takt 4 Mic-Sec / 250 kHz

Überlauf-Periode: 10.000 Mic-Sec

=> Reload-Wert = 2500 (weil Timer 3 hochzählt)

Der C-Code könnte irgendwie so aufgebaut sein:

```
#define TC_TxM(y)    ( (y << 3) & 0x0038)  // y: {0;...;7}

T3CON = 0x0043;
//besser:
T3CON = TC_TxM( 0 ) | TC_TxI( 3 );
T3UD  = 0;  //bewusst initialisieren, so dass er aufwärts zählt
T3UDE = 0;
  ...
T2CON = 0x0027;
//besser:
T2CON = TC_TxM( 4 ) | TC_TxI( 7 );
T2R = 0;
T2 = -2500;  //Reload-Register Laden
T3 = -2500;

T3IC = IC_ILVL( 3 )  //Interrupt-Register konfigurieren
                     //hier sieht man auch die Priorität
T3IE = 1;  // Interupt aktivieren (enablen)
IEN  = 1;  // globale Enable/Disalbe-Flag

T3R  = 1;  // so, jetzt geht's los!


void main( void )
{
  unsigned int wtimer1000ms = 100;
  unsigned int f1000ms = 0;  //flag, ob eine sec. vergangen ist
                             //wird in der Polling-Schleife geprüft
  ...
  while( 1 ) {
    ...  // read inputs
    if( f1000ms ) {  // dann schrei!
      if( ENGINEON_1 ) ..++;
  }
}


//interrupt-service-Routine
```

```
void TIMER3( void ) interrupt 0x23
{
  wtimer1000ms--;
  if( wtimer1000ms == 0 ) {
    wtimer1000ms = 100;
    f1000ms = 1;
  }
}
```

3.3 Serielle Datenübertragung

3.3.1 Allgemeines

3.3.1.1 Unterscheidung parallel – seriell

parallel

zu einem Zeitpunkt werden mehrere Bits auf mehreren Leitungen übertragen

seriell

ein Bit je Zeit auf einer Leitung

Vorteil:

- definierte geometrische Struktur möglich
- keine Laufzeitdifferenzen
- preisgünstigeres Kabel

3.3.1.2 Kommunikationsmodus

- Simplex: Übertragung nur in eine Richtung (z.B. Master → Slave)
- Halb-Duplex: eine Übertragungsstrecke; beide Richtungen sind möglich, aber immer nur nacheinander (Master → Slave, Master ← Slave)
- Duplex: zwei Übertragungsstrecken, beide Richtungen gleichzeitig möglich (Master ↔ Slave)

3.3.1.3 Übertragungsmodi synchron – asynchron

Empfänger erhält „nur“ eine Folge von 0/1. Problematik:

- Anfang und Ende eines Bits
- Anfang und Ende eines Zeichens
- Anfang und Ende einer Nachricht

Synchrone Übertragung

USRT (engl. universal synchronious receive transmit device); einfache Implementierung in Microcontrollern

Takt des Senders ist beim Empfänger verfügbar

- durch separate Taktleitung (typ. auf Baugruppen)
- durch Taktregeneratoren aus dem Bitstrom

Asynchrone Übertragung

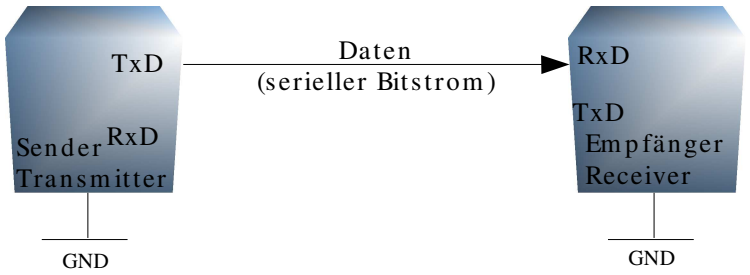
UART (engl. universal ansynchronious receive transmit device); einfache Implementierung in Microcontrollern

Takt des Senders und des Empfängers sind unabhängig. Synchronisation erfolgt typ. bei jedem Zeichen neu, bzw. die Anzahl der übertragbaren Bits ist begrenzt

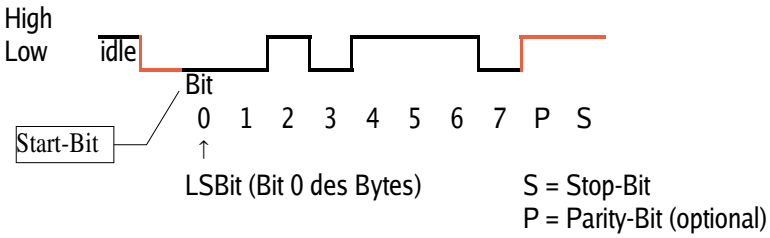
Die Kombination aus USRT und UART nennt man USART.

3.3.2 Asynchrone serielle Datenübertragung (UART)

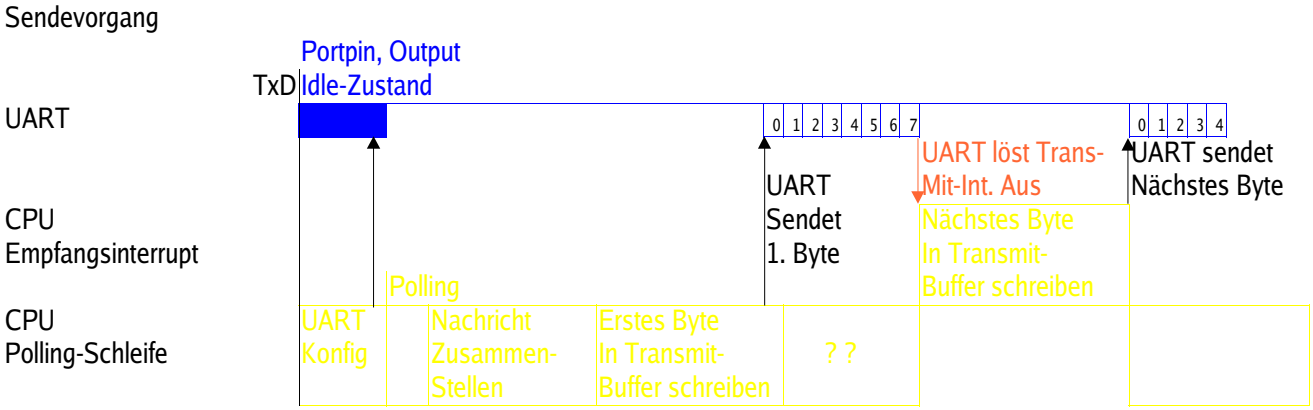
- Byte-orientiertes Übertragungsverfahren
- Synchronisation erfolgt bei jedem neuen Zeichen
- Dauer eines Bits ist Bekannt (ca.)



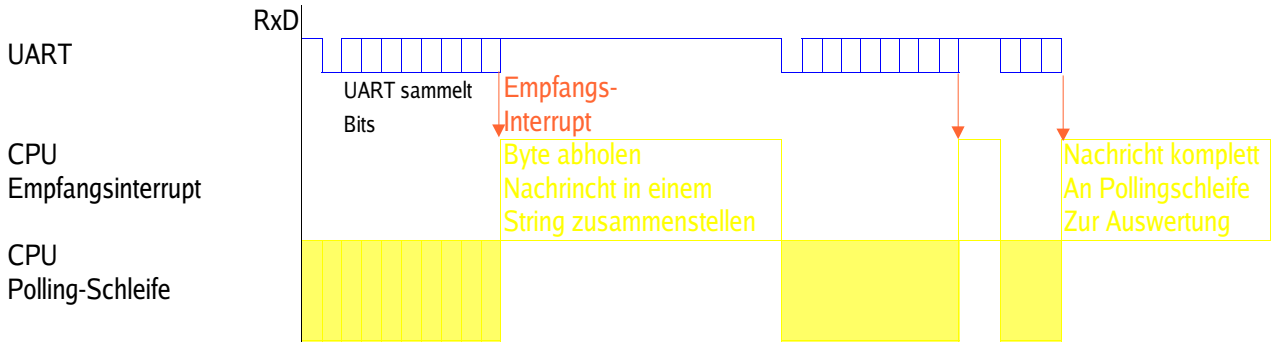
Typischer Ablauf Sendevorgang:



Flanke dient zur Synchronisation, Abtastung in der Bit-Mitte (typischerweise bis zu drei Abtastungen).

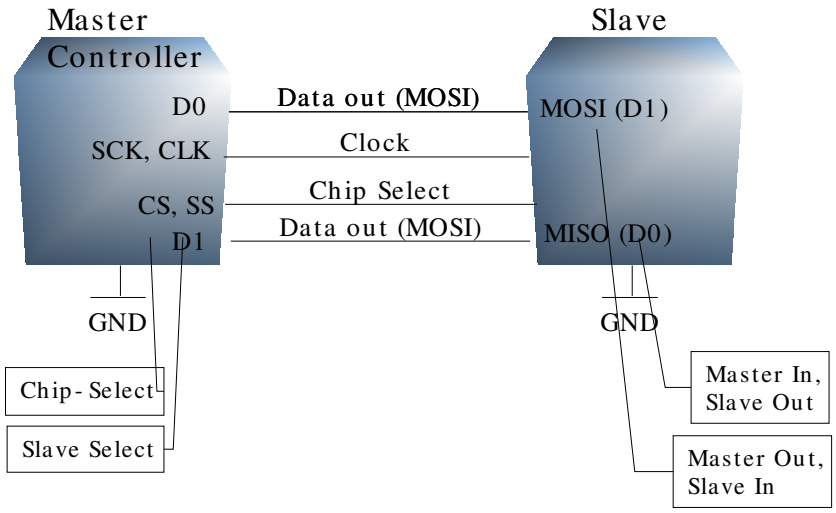


Empfangsvorgang

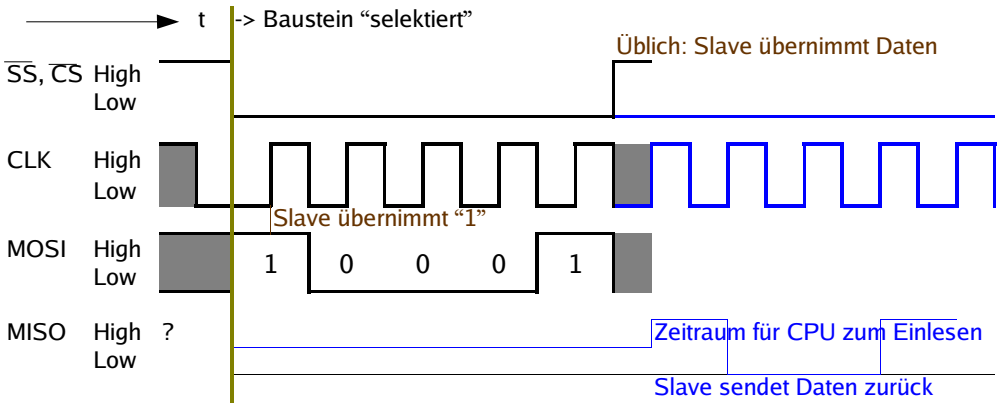


3.3.3 Synchrone Datenübertragung

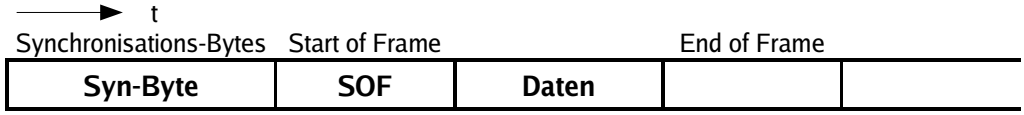
3.3.3.1 Separate Übertragung des Taktes – Serial Microwire (SPI)



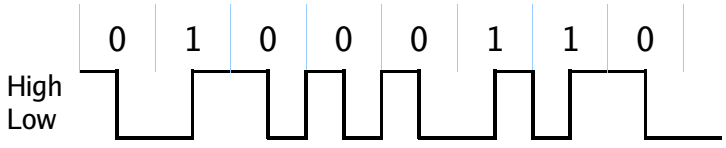
Eingesetzt zur Kommunikation auf der Baugruppe



Prinzipieller Kommunikationsablauf



um eine Taktregeneration zu ermöglichen, müssen die Bits (0, 1) kodiert werden, z.B. Manchester-Codierung



3.3.3.2 Übung: Serial Microwire

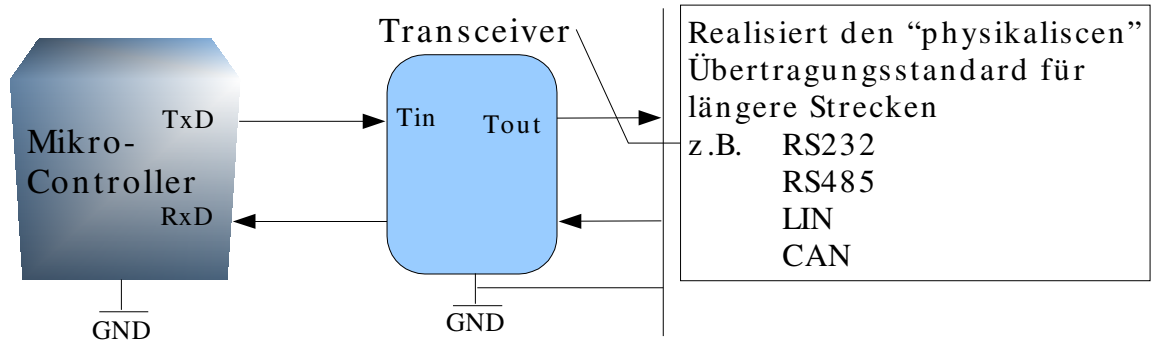
Init:

```
DP1H = 0x70;
```

Deklaration:

```
sbit LD = P1H^4;
sbit CLK = P1H^6
sbit DIN = P1H^5
load_da (int da_a, int da_b) {
           int mask;
           CLK = 0;
           LD = 0;
           mask = 0x0800;        // Bit 11
           while () {
                      if ((da_a & mask) != 0) {
                                 DIN = 1;
                      } else {
                                 DIN = 0;
                      }
                      CLK = 1;   // Flanke ⌐
                      mask = mask >> 1;
                      CLU = 0;
           }
           LD = 1;
}
```

3.3.4 Physical Layer



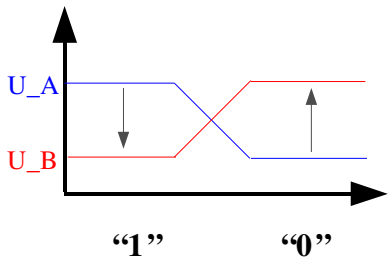
3.3.4.1 RS 232-Standard

<siehe Folie „mc_kap3_3_rs232_rs485.pdf“>

3.3.4.2 RS485/488

Merkmale von RS485:

- differentielle Datenübertragung
- verdrillte Datenleitungen, um störunempfinglich zu sein
- zumeist nur drei Leitungen: zwei ineinander verdrillte Leitungen für differentielle Datenübertragung plus eine Masse-Leitung
  => Betrieb nur in eine Richtung möglich
  => Halb-Duplex-Betrieb
- es ist auch möglich, eine Bus-Struktur aufzubauen



Der RS485-Standard wird auch vom SIEMENS-Profibus verwendet

Merkmale von RS488: genauso wie bei RS485, nur dass es zwei verdrillte Adern gibt, so dass Full-Duplex-Betrieb möglich ist.

<siehe Folie „mc_kap3_3_rs232_rs485.pdf“>

3.3.5 LIN-Bus

<siehe Folie „mc_kap3_3_lin.pdf“>

3.3.6 CAN-Bus

<siehe Folie „mc_kap3_3_can.pdf“>

4 CPU – 2. Teil

4.1 Ganzzahlarithmetik Grundlagen

Bearbeitungsreihenfolge

z.B.

$k = l \cdot g / h;$
$l = 20 \; g = 20, h = 15$
a) $k = l \cdot (g/h)$
$20 \cdot 1 = 20$
b) $k = (l \cdot g)/h = 26$

4.1.1 2-er Komplement

Idee: ALU hat nur Addierwerk für ganze Zahlen. Trotzdem sollen Substraktionen möglich sein!

$a - b \qquad a, b > 0$
$a + \underbrace{(-b)}_{\text{2-er Komplement zu } |b|}$
$a + (2^N - b)$ N Bit Breite für Darstellung der Zahlen
z.B. $b = 1, N = 4$ 10000
00001
01111

$a = 4$ 0100
1111
|0011

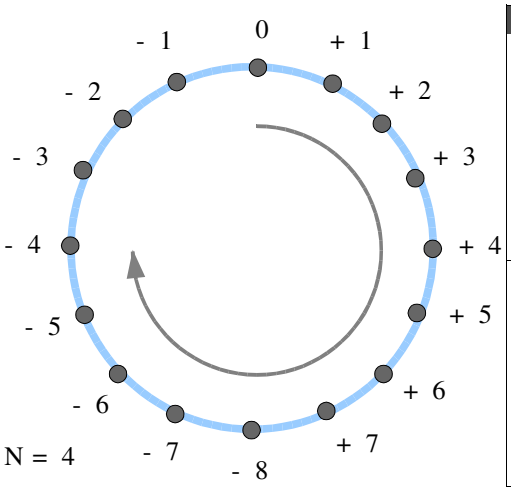
einfach HW zur Ermittlung des 2-er Komplements

$a - b \mathrel{\hat{=}} a + \underbrace{(2^N - b - 1)}_{\text{2-er Kompl. (bitweise Invertierung)}} + 1$

$a = 4$ 0100
$b = 1$ 1110
1 – Carry für Addierer des 0 – ten Bit
0011

Wertebereich 2-er komplementäre Zahlen (N Bits)

$-2^{N-1} \ldots 0 \ldots (2^{N-1} - 1)$



Zuordnungstabelle	
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

Multiplikation

N=4
a=5
b=8 unsigned signed $b=-8$
1000 2er Kompl. für 2·N Bits!
0101·1000
0010 1000 11111000
1

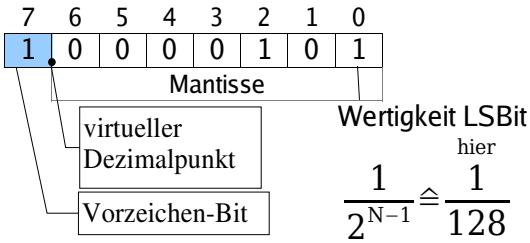
Ergebnis 2 · N Bits nötig

4.1.2 2-er Komplement als Festkommazahl

Idee: Bei einer Multiplikation soll der darstellbare Zahlenbereich nicht verlassen werden

⇒ Zahlenbereich [-1 ... 0 ... 1[

z.B. N=8



Skalierung $\cdot\frac{1}{2^{N-1}}$

Fixed Point Multiplikation Beispiel:

N = 4

+2	0.010	$\frac{2}{8}$
*-6	1.010	$-\frac{6}{8}$
-12	1.110100	$-\frac{12}{64} \triangleq -\frac{24}{128}$

für richtiges Ergebnis um 1 nach links schieben

$\triangleq -\frac{12}{128}$

Ergebnis der Ganzzahlmultipliziereinhei

ein nach links geschoben

1.110 1000

Reduktion
auf 4 Bits

$\triangleq -\frac{2}{8} \triangleq -\frac{1}{4} \Leftrightarrow \frac{-12}{64} \triangleq -\frac{3}{16}$

4.1.3 Grundprinzip der Signalverarbeitung

4.2 Programmiermodelle

4.2.1 (Registerstrukturen)

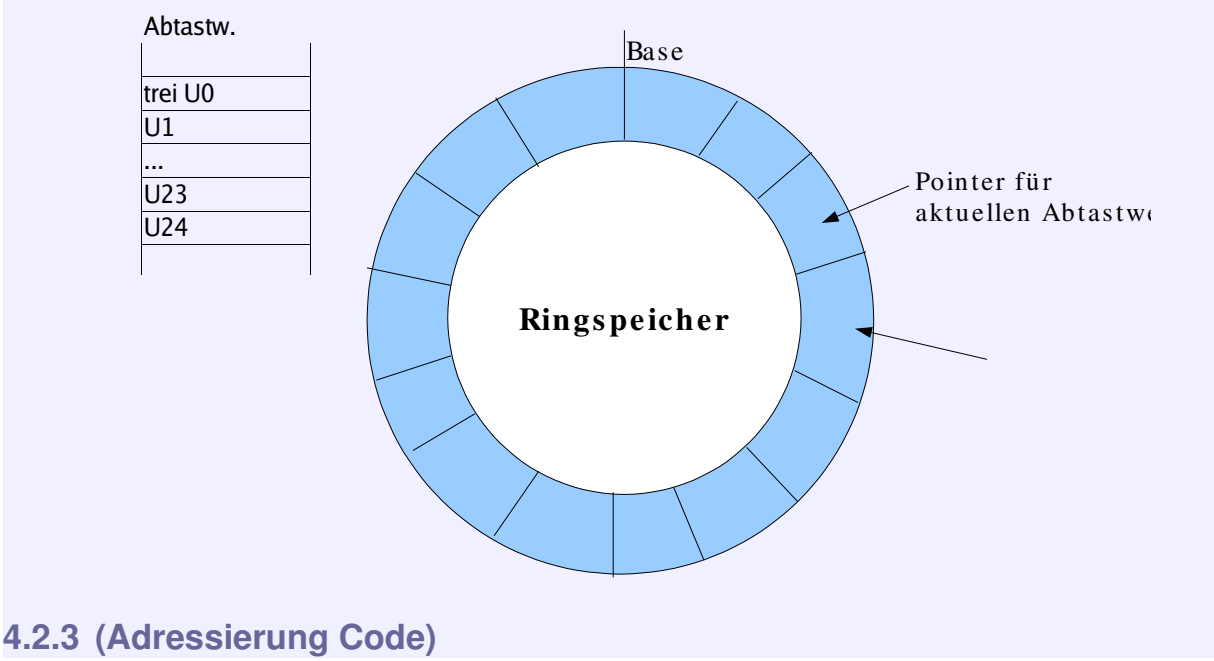
Einteilung der Register in

- Kontrollregister
  - Interrupt-Kontrollregister
  - Register zur Konfiguration der Peripheriemodule (SFR)
  - Steuerbits für die ALU u.v.m
- Statusregister; sie spiegeln den momentanen „Zustand“ der CPU wieder und ändern sich durch Operationen
  - Priorität des momentan bearbeiteten Codes
  - Status Flags der ALU (Zero-Flag, Negativ, ...)
  - Registersatz
- Adressen- und Operanden-Register

4.2.2 (Operanden-Adressierung)

Arten der Adressierung:

- Immediate: hier handelt es sich um Konstanten, die fest in den Befehl codiert sind
  z.B. ADD R3, # data3; damit reicht ein Wort als Gesamtbefehl aus
  z.B. ADD R3, # data16; damit stehen 16 Bit für die Konstante zur Verfügung; es werden zwei Wörter verwendet
- Memory Direct: der Adress-Operand ist bei 16-Bit-Prozessoren mit 16-Bit-Adressraum 24 Bit lang; somit benötigt ein Befehl zwei Wörter
- Register Direct: direkte Register-Adressierung
  - single Index
    z.B. ADD A, @Rx
  - double Index
    z.B. ADD A, Rx, Ry
- Offset
  - Base Offset: verwendet für Adressierung eines Stucture-Elements innerhalb eines Arrays
  - Implicit Base Offset: fusionieren der Bits; Vorsicht: nicht Addieren!
    z.B. A = A + @SMEM
  - Register Indexed Base Offset
  - Circular Base-Offset



4.2.3 (Adressierung Code)

Adressierung (im einfachsten Fall) durch einen „Instruction Pointer“ bzw. „Program Counter“ wird bei der Abarbeitung des Programms automatisch inkrementiert.

Ausnahmen:

- Sprünge
  - absolute Sprünge: IP wird mit neuem Wert geladen
  - relative Sprünge. zum IP wird eine Distanzadresse addiert
- Unterprogramm/Funktionsaufruf
  - direkt: Adresse der Funktion ist im Code hinterlegt
  - indirekt: Adresse der Funktion ist im Register hinterlegt (dieses Verhalten entspricht in C den Funktionszeigern)

4.2.3.1 (Ein Instruction Pointer)

Gegeben: 8-Bit-Architektur, 16-Bit-Adressraum
⇒ JMPA (absoluter Sprung): 3-Byte-Befehl.

Zieladresse:

JMPA
ZA LB
ZA HB

Gegeben: 16-Bit-Architektur
⇒ JMPA (relativer Sprung): 2-Byte-Befehl.

Zieladresse:

15	0
JMPA	ZA MS-Byte
ZA Lower Word	

; Hinweise zu „JMPA“:

- mit Bedingungen reichen u.U. 2 words nicht
- indirekte Aufrufe/Sprünge; nur 16-Bit-Register (R0 ... R15)

4.2.3.2 (Banking)

Hinweise zum PDF „...“

Banking verwendet mehrere Speicherblätter, die über spezielle Maschinenbefehle ausgewählt werden

4.2.3.3 (Segmentierung)

Hinweise zum PDF ....

Wenn der Linker intelligent genug war, die Funktion nicht an die Grenze zu legen sondern in ein Segment, dann ist alles chillig.

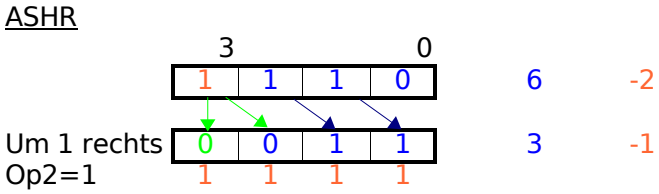
Zerteilen der physikalischen Adresse auf zwei Register.

4.3 Maschinenbefehle - Assemblersprache

Einteilung:

- Transportbefehle
- Datenbearbeitungsbefehle
- Programmsteuerbefehle
- CPU-Steuerbefehle

Hinweise zu Folie „K-4.3_AssemblerBefehle.pdf“



4.3.1 (Transportbefehle)

A = ACCU = Akkumulator

DPTR = Data-Pointer

MOV A, @R2R2: 8-Bit-Register

MOVX A, @DPTR DPTR: 16-Bit-Register

MOVC A, @DPTR

8051: Harvard-Architektur:

- internes RAM (8-Bit-Adresse MOV)
- externes RAM (16-Bit-Adresse MOVX (external))
- Code-Speicher (Konstanten + Code; 16-Bit-Adresse MOVC)

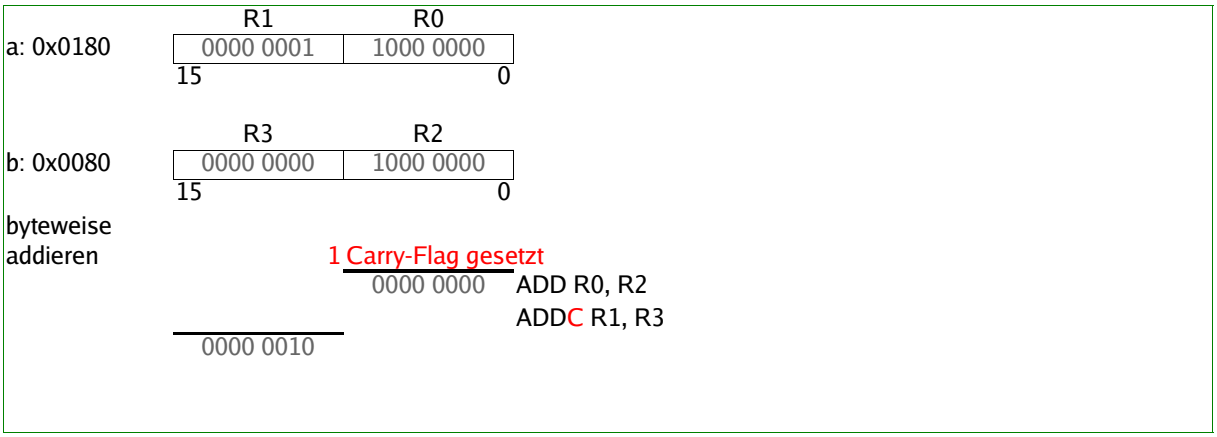
PUSH R3

4.3.2 Datenbearbeitungsbefehle

- ADC

Beispiel: 8-Bit-CPU

Es sollen zwei 16-Bit-Zahlen addiert werden: c = a + b



CMPD1 $\underbrace{op1}_{i}, \underbrace{op2}_{\#0}$; for(i = 100, i >= 0; i++) {...}

Compare op1 ⇔ op2 ⇒ Flags werden gesetzt, z.B:: ZERO_Flag falls op1 == op2

INZ

4.3.3 (Rotier- und Schiebebefehle)

ROR R0, #1 (Rotiere Register R0 um eine Stelle nach rechts)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
vorher:	1	0	0	1	0	0	1	1	0	1	0	1	1	0	1	0
nachher:	0	1	0	0	1	0	0	1	1	0	1	0	1	1	0	1

SHR R0, #2 (Rechts-Shift um zwei Stellen; es werden 0en aufgefüllt)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
vorher:	1	0	0	1	0	0	1	1	0	1	0	1	1	0	1	0
nachher:	0	0	1	0	0	1	0	0	1	1	0	1	0	1	1	0

Beispiel: Rechtsshift

```
singed int R0;
R1 = R0 / 2
```

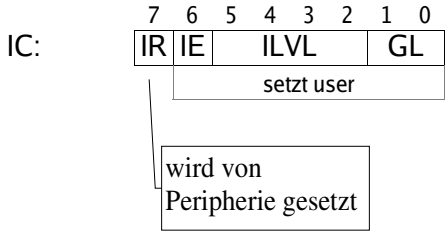
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
vorher:	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0x4000
nachher:	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	

In dem Beispiel ist R0 eine positive Zahl. Probleme treten auf, wenn negative Zahlen behandelt werden. Dann muss das Vorzeichen-Bit wieder auf die erste Stelle kopiert werden.

ASHR R0, #1

4.3.4 Bitmanipulation

z.B. IC:



IC = 0x0044 | (IC & 0x0080

Intern:

```
MOV R4, IC        <- nun setzt Peripherie IR
AND R4, 0x0080
OR  R4, 0x0044
MOV IC, R4        <- IR wird gelöscht !!!
```

Es hilft alles nichts: die CPU muss immer das gesamte Register lesen, das einzelne Bit setzten und dann das komplette Register zurückschreiben. Wenn in der Zeit von der Hardware ein Flag gesetzt wird, wird es von der CPU wieder gelöscht.

Hinweis zur Folie „Maschinenbefehle – Assemblersprache“, Seite 4

Der untere Absatz ist falsch und kann ignoriert werden.

Lösung: Protected Bits

- Register msg-ctl (bei CAN-Versuch). Zwei Bits werden gebraucht.
  Es gibt folgende Kombinationsmöglichkeiten:
  – 01: Reset durch CPU
  – 11: Setzen
  – 11: unverändert lassen

Beispiel: LEDs zum Leuchten bringen

```
// niederwertige 8 Bit von a auf Port P2 ausgeben
IEN = 0
P2 = ( P2 & 0xFF00 ) | ( a & 0x00FF );
IEN = 1
```

jetzt komme einer auf die Idee, in einer Interrupt-Funktion die anderen 8 Bits zu beschreiben. Das ist schlecht.

```
// höherwertige 8 Bits von b auf P2 ausgeben
P2 = ( P2 & 0x00FF ) | ( a & 0x00FF );
```

Das Problem das hier auftritt: wenn jemand den Port schreibt und genau in dieser Phase der Interrupt ausgelöst und behandelt wird, dann wird das Ergebnis des Interrupts wieder überschrieben, da das reguläre Schreiben von a auf Port P2 mit einer Kopie von Port 2 arbeitet und diese Kopie dann zurückschreibt. Die Lösung wurde blau dargestellt.

4.3.5 (Programm-Steuer-Befehle)

Es existieren folgende Arten von Programm-Steuer-Befehlen:

- Sprünge
- Unterprogrammaufrufe

Sprünge

JMPA cc, op2 (Sprung)

für OP2 gilt: Zieladresse im gleichen Segment „Intra-Seg“

Möglichkeiten: für cc:

- cc_uc (Condition-Code, unconditional; er springt immer) [if-else-Konstrukte]
- cc_z (Condition-Code, Zero-Flag) [bei SUB Zero-Flag gesetzt ist]
- cc_N (Condition Code, Null)
- cc_NN (Condition Code, Not Null)
- ...

„einfache“ haben oft nur wenige Bedingungen

DJNZ op1, Zieladresse (Decrement Jump Not Zero)
→ Einsatz in Schleifen, wo die Laufvariable nach unten gezählt wird.

Unterprogrammaufrufe

Intrasegment:

CALLA cc, ZielRET

```
( SP )   ← (SP ) - 2
(( SP )) ← ( IP )
( IP )   ← Ziel
```

RET

```
( IP )   ← (( SP ))
( SP )   ← ( SP ) + 2
```

Intersegment

CALLS

```
( SP )   ← ( SP ) - 2
(( SP )) ← ( CSP )
( SP )   ← ( SP ) - 2
(( SP )) ← ( IP )
( CSP )
( ISP )
```

RETS

```
( IP )   ← (( SP ))
           ( SP ) + 2
( CSP )  ← (( SP ))
           ( SP ) + 2
```

RETI (Return für Interrupt-Aufruf)

```
( IP )   ← (( SP ))
           ( SP ) + 2
( CSP )  ← ( SP )
( PSW )  ← (( SP ))
```

auslösender Befehl für Interrupt-Service-Routine:

```
TRAP
(( SP )) ← ( PSW )
(( SP )) ← ( CSP )
(( SP )) ← ( IP )
```

wird i.a. vom Interrupt-Controller bei Request in die Pipeline eingeschleust

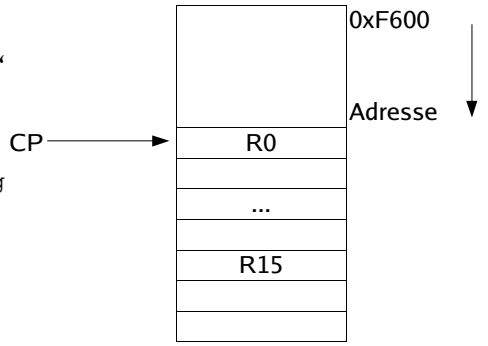
4.3.6 (CPU-Steuerbefehle)

Wechseln von Registersätzen

Context-Pointer (CP) bestimmt den „angewählten“ Registersatz

In C (Keil-IDE) sieht das so aus:

```
void NAME ( void ) interrupt NR using
REGNAME {
...
}
```



Parameterübergabe

Wenn man von einem C-Programm aus ein Assembler-Programm oder eine Assembler-Funktion aufrufen will, muss man beachten, wie der Controller die Parameter verarbeitet.

Beispiel: C-Aufruf

```
long function( int b, int c, long d, int e, int f ) { ... }
```

b → R8
c → R9
d → R10, R11
e → R12
f → Stack (weil der Controller nur 12 Register hat und alles weitere auf den Stack schieben muss)

Hinweis: die Rückgabe des Datentyps long erfolgt auf Register R4 und R7.

4.3.7 (Unterschied Controller – DSP – Prozessor)

Controller:

- integrierte Peripheriemodule und effiziente Verbindung zur CPU
- „mächtige“ Interruptverarbeitung
- Bitmanipulation

DSP:

- MAC (Multiplikations-Additions-Einheit) in einem Takt
- Zero Overhead Loops: Schleifenzähler ist in Hardware realisiert, der parallel zur MAC-Einheit integriert ist
- zirkuläre Adressierung
- bit reversed Adressierung für Spektral-Analyse via FFT: die interne Berechnung und Darstellung erfolgt von links nach rechts von LSB zu MSB; die Ausgabe erfolgt genau anders herum und somit richtig lesbar für normale Programme
- Festkomma-Arithmetik
- Saturation (Sättigung); was passiert bei Überlauf? Bei Überlauf wird der Maximal-Wert gehalten und nicht im Zahlenring wieder von vorne angefangen ⇒ entspricht bei einem Audio-Verstärker der maximalen Leistungs-Begrenzung

Prozessor:

- hohe Taktfrequenzen
- riesiger Adressraum

- Memory Management Unit mit Memory Protection

4.4 Grundregeln für effizientes C

4.4.1 Speicherwerte

Speichertypen:

–	NVM	Flash-ROM	Code, Konstanten	schreiben langsam + unflexibel, lesen mit f_{CPU}
–	VM	RAM	Variablen	schreiben + lesen flexibel mit f_{CPU}
–	NVM	EEPROM	Parameter	schreiben langsam aber flexibel

NVM: non volatile memory
VM: volatile memory

Alle 3 Typen können

–	on-chip	sehr effizienter Zugriff durch mehrere Busse, kurzer Zyklus
–	off-chip	typischerweise ein Bus für Flash + RAM, zum Teil Daten- und Adressbus gemultiplext (→ langsam)

realisiert werden.

⇒ Zeitkritischer Code in's interne Flash
häufig benötigte Variablen in's interne RAM (on-chip RAM)

4.4.2 Operandenverwendung

4.4.2.1 Konstanten

```
b = a + 3;
```

z.B. 8051 (8-Bit-Architektur)	8-Bit-Konstante	2-Byte-Befehl
	16-Bit-Konstante	3-Byte-Befehl
C166 (16-Bit-Architektur)	3-Bit-Konstante	2-Byte-Befehl
	16-Bit-Konstante	4-Byte-Befehl
ARM-7 (32-Bit-Architektur)	8-Bit-Konstante	4-Byte-Befehl

4.4.2.2 Variablen

```
B = C;
```

B, C im Registerformat

Register:

```
8051:  MOV A, RC
       MOR RB, A       /2 Byte, 2 Befehlszyklen
C166:  MOV RB, RC      /2 Byte, 1 Befehlszyklus
```

internes RAM:

```
8051:  MOV A, MEMC
       MOV MEMB, A     /4 Byte, 2 Befehlszyklen
C166:  MOV Ri, MEMC
       MOV MEMB, Ri    /8 Byte, 2 Befehlszyklen
```

externes RAM:

```
8051:  MOV DPTR, MEMC
       MOV A, @DPTR    /8 Byte, 8 Befehlszyklen
```

```
C166: Code intern    RAM extern 10 CPU-Takte (5 Befehlszyklen)
      Code extern    RAM extern 18 CPU-Takte (9 Befehlszyklen)
          Bus gemultiplext
```

4.4.2.3 Arithmetik

Multiplikation C166 V1

Code intern, Variablen in Registern

```
char       8 Bit      10 Takte
int        16 Bit     10 Takte
long       32 Bit     46 Takte
float      32 Bit     170 Takte
double     64 Bit     372 Takte
```

Sinus C166 V1

```
32 Bit     2500 ... 3100 Takte
```

Multiplikation ARM7 (ohne FPU)

```
char       8 Bit      2 Takte
int        16 Bit     2 Takte
long       32 Bit     4 Takte (?)
float                 414 Takte            GNU-C ✡
double                721 Takte            GNU-C ✡
```

Multiplikation MPC 555 mit FPU

```
float      4 Takte
double     5 Takte
```

5 Bussysteme

5.1 Grundlagen Busarchitektur

5.1.1 Überblick Busse

Ebenen im Überblick:

- Chipebene: parallele Verbindungen, CPU-ALU-Register
- Baugruppenebene:
  - parallele Verbindungen
    Prozessor/Controller – Programm- und Variablenspeicher
  - serielle Verbindungen zur Peripheriebausteinen
- Geräteebene:
  - ISA-Bus
  - PCI-Bus auf Motherboard (parallel und seriell)
  - VME (parallel und seriell)
- Anlagenebene
- Feldbusse (Profibus, ASI, Interbus, CAN, Ethernet, ...)

5.1.2 Einteilung Busarchitekturen

Überblick:

- von-Neumann-Architektur
- Havard-Architektur

von-Neumann-Architektur

Die Architektur zeichnet sich durch einen linearen Adressraum für Code und Daten aus.

Vorteil:

- effektive Speicheraufteilung möglich
- „platzsparend“, nur ein Adressbus und ein Datenbus

Beispiele: von-Neumann-Architektur

C167, Motorola 68HC12, Intel 80x86, ...

Havard-Architektur

Hier sind Adressräume für Code und Daten getrennt. Bussysteme sind mehrfach ausgeführt. Auf Baugruppen-Ebene verbraucht die Architektur viele Leitungen, aber auf Chip-Ebene ist das unkritisch.

Vorteil:

- gleichzeitiger Zugriff auf Code- und Datenspeicher möglich (DSPs - „extended“ modified Havard Architecture mit Mehrfachauslegung der Bussysteme)
- unterschiedliche (Bus-)Breiten für Code und Daten möglich ⇒ der Chip wird billiger, weil Chipfläche kostet Geld. (PIC-Serie der Firma MICROCHIP).

Nachteil:

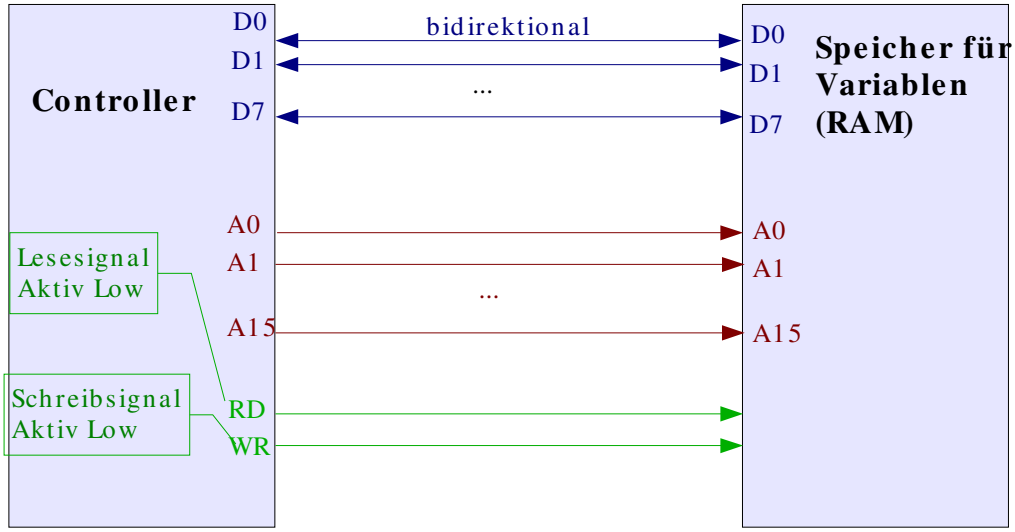
- falls die Busse aus dem Controller herausgeführt werden sollen: viele Pins nötig ⇒ weniger Pins für Steuerfunktionen
  → Busse werden zusammen gelegt

Beispiel: Havard-Architektur

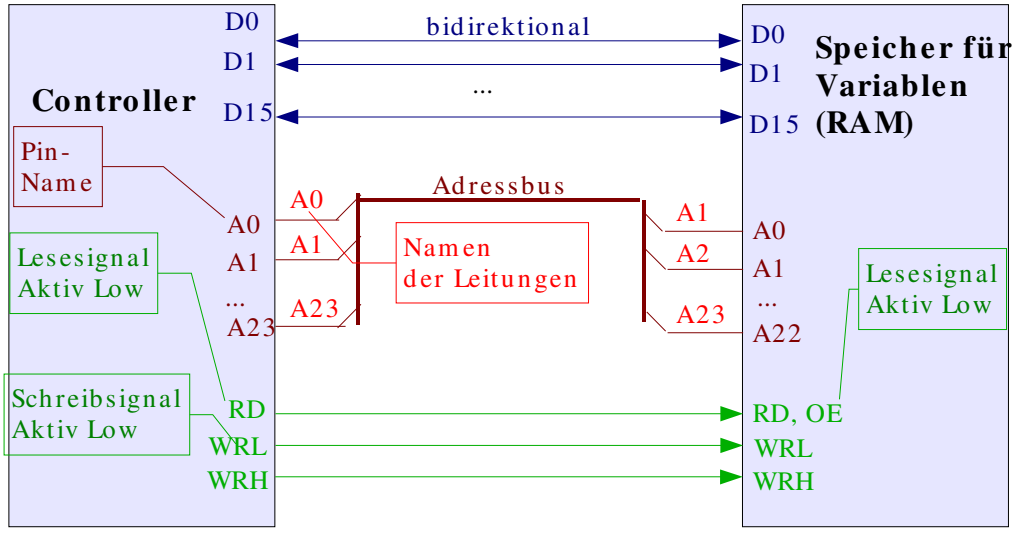
Intel 8051, AVR, PIC, DSP, ...

5.1.3 (Prinzipieller Bus-Aufbau)

8-Bit-Datenbus mit 8-Bit-Speicher:



16-Bit-Datenbus mit 16-Bit-Speicher:



Dabei muss folgendes Beachtet werden:

- Der Controller adressiert den Speicher Byte-weise, aber der RAM interpretiert den Adressbus und damit die Adress-Pins als Word-Adresse!
- WRL (Write Lower Byte): Anweisung, nur das Lower-Byte zu schreiben
- WRH (Write Higher Byte): Anweisung, nur das Higher Byte zu schreiben

Controller-Adresse = Byte-Adresse				Speicher-Adresse						
	A2	A1	A0							
...										
...	0	0	0		Lower B.	...	0	0	0
...	0	0	1	Higher B.		...	0	0	0
...	0	1	0		Lower B.	...	0	0	1
...	0	1	1	Higher B.		...	0	0	1

Zugriff	Controller-Pins A0	WRL	WRH
Word schreiben	0 immer gerade Byte-Adresse	0	0
Byte schreiben Lower Byte (D0 – D7)	0	0	1
Byte schreiben Higher Byte (D8 – D15)	1	1	0

5.2 Buszyklus

- Datenbus (8 Bit, 16, 32, 64, ...)
- Adressbus (16, 24, 32, ...)
- Steuersignale (Lesen, Schreiben)

Unterscheiung:

- synchroner Buszyklus (+ Takt)
- anynchroner Buszyklus

5.2.1 Synchroner Buszyklus

Hinweise zu Folie „K-5.2_Buszyklen.pdf“, Seite 1

Das Timing ist genauer realisiert.

Lese-Zyklus:

- Startsignal ist low, Flanke kommt, ich muss die Adresse intern speichern.
- Man benötigt immer zwei Takt-Zyklen: einen für die Adress-Übermittlung, einen für Lesen/Schreiben

Hinweise zu Folie „K-6_Speicherbausteine.pdf“, Seite 3

$2^{17} \cdot 2^5 = 2^{22} = 2^{10} \cdot 2^{10} \cdot 2^2 = 4\,Mbit$

- GW (*Global Write*): 32-bit-Zugriff
- 6 Leitungen, die sich mit Schreiben beschäftigen
- OE-Leitung: Lese-Leitung
- ZZ: hochohmiger Zustand in Stand-By-Betrieb versetzen
- CLC (*Clock*): Synchroner Speicher
- CS_x (*Chip Select*): Auswahl der Baugruppe

Hinweise zu Folie „K-6_Speicherbausteine.pdf“, Seite 4

nicht verstanden ... !

Ein Burse-Zyklus hat nur 4 32-Bit-Wörter. Das ist so festgelegt.

5.2.2 Asynchroner Buszyklus

Hinweise zur Folie „K-5.2_Buszyklen.pdf“, Seite 1

Komponenten:

- Adress-Bus: A0, ..., A23
- Master: CPU, Micro-Controller

Vorgehen beim Lese-Zyklus:

- Die CPU will Speicher Ax lesen
- beim Read wird ein Aktiv low gelegt
- der Slave ist hier bei uns der Speicher; er lässt sich ein bisschen Zeit, darauf zu antworten. Δt zwischen aktiv low und Antwort des Speichers.

Vorgehen beim Schreib-Zyklus:

- Die CPU will Daten schreiben
- mit der steigenden Flanke von aktive Low auf high ist es das Übernahmesignal für den Speicher, die Daten vom Bus zu nehmen.

Hinweise zu Folie „K-6_Speicherbausteine.pdf“, Seite 1

SRAM: Statisches RAM: es behält den Inhalt bei, wenn die Spannung anliegt. Das ist beim dynamischen RAM nicht so.

Komponenten:

- 128 kB Speicher ⇒ 17 Adressleitungen notwendig: A0 – A16
- Pin N.C. (not connected): nicht gebrauchter Pin als Reserve für die Zukunft, falls eine Adressleitung Nr. 18 für 256K gebraucht wird.
- I/O-Ports: die Japaner fangen bei 1 das Zählen an.
- CSx (*Chip-Select*): falls mehrere Bausteine zu einem großen Speicher zusammen gekoppelt wird, übernimmt jeder einen Teil-Adressraum. Mit dem Chip-Select wird dem Bauteil mitgeteilt, in welchem Adressraum er zu liegen hat.

Hinweise zu Folie „K-6_Speicherbausteine.pdf“, Seite 2

Lese-Zyklus:

- Read-Cycle = Reaktionszeit; hier 70 ns.
- wenn das Lesesignal weggenommen wird, kann es 25 ns dauern, bis der Baustein seine Treiber abgeschalten hat. Es kommt dann eh viel später der Schreib-Zugriff vom Master

Schreib-Zyklus:

wie beim Lese-Zylus.

5.2.3 Multiplex von Adress- und Datenbus

Beispiel: 8051-Derivat 80C32

Hinweis zu Folie „K-5.2_Buszyklen.pdf“, Seite 3

Havard-Architektur:

- Auf Port 0: AD0 – AD7
- Auf Port 2: A8 – A15
- Das blaue liefert der Code-Speicher
- ALE fallende Flanke: Latch

Hinweis zu Folie „K-5.2_Buszyklen.pdf“, Seite 4

Write-Zyklus:

- Der Prozessor geht ins RAM rein
- Er liefert selber das Signal
- PSEN (*Program Store Enable*): Entweder auf Busspeicher oder auf Code-Speicher zugreifen

Hinweis zu Folie „K-5.2_Buszyklen.pdf“, Seite 2

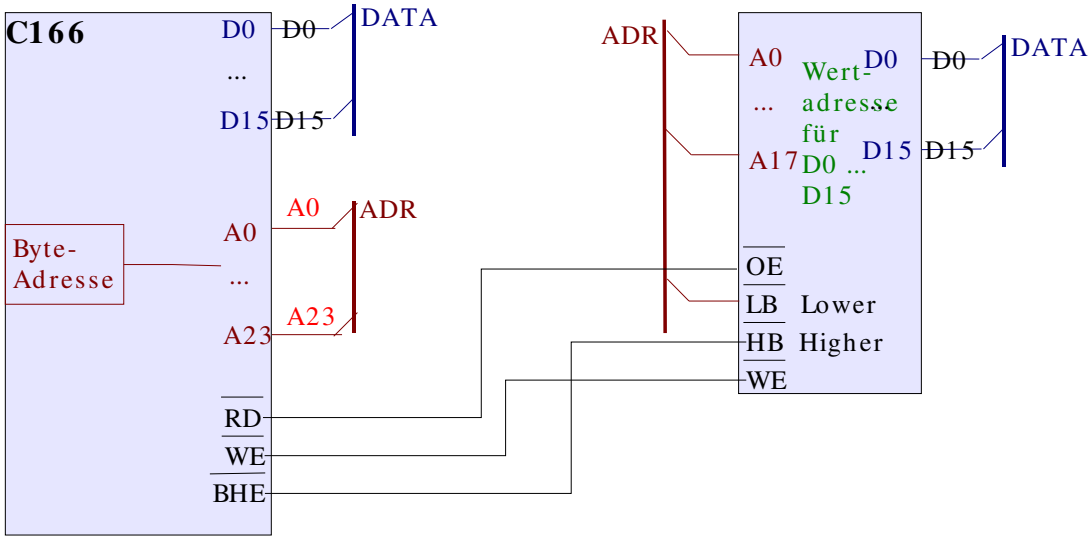
Elemente:

- BUS (P0): A0 – A15
- Segment (P4): A16 - ...
- RD: Q0 – Q15
- ALE ist verlängerbar um einen Takt (für langsame Speicher)
- BUS (P0) kann er verlängern

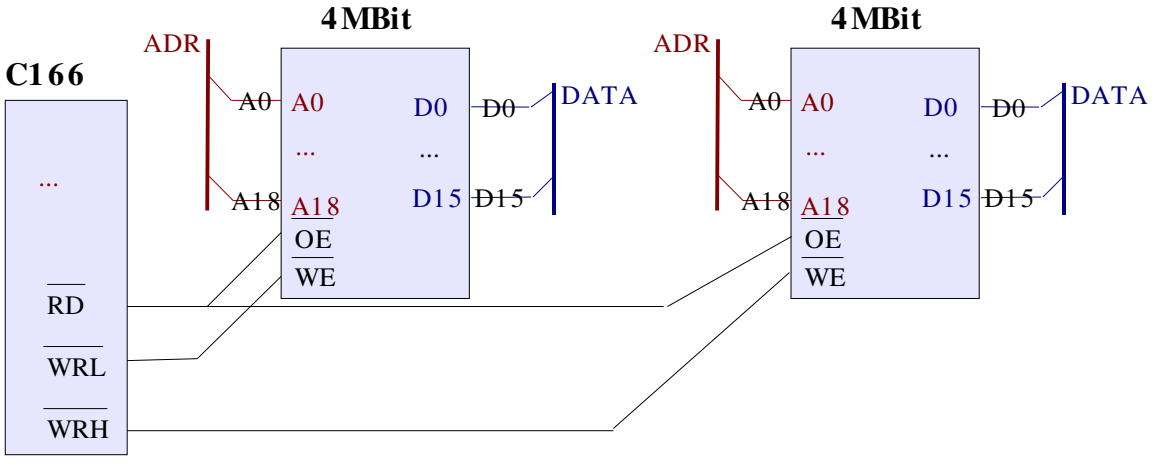
Vergleich:

- nicht-gemultiplext: 2 Takte
- gemultiplext: 3 Takte

5.2.4 Wort- und Bytezugriffe



	Byte Adresse A1	A1	Zyklus mit WE	BHE	Zyklus WRL	WRH	Signifikante Leitunge Des Datenbusses	
Wort-Zugriff (immer gerade Byte-Adresse)	0	X	0	0	0	0	D1	D15
Byte-Zugriff Gerade Adresse	0	X	0	1	0	1	D0	D7
Byte-Zugriff Ungerade Adr.	1	X	0	0	1	0	D8	D15

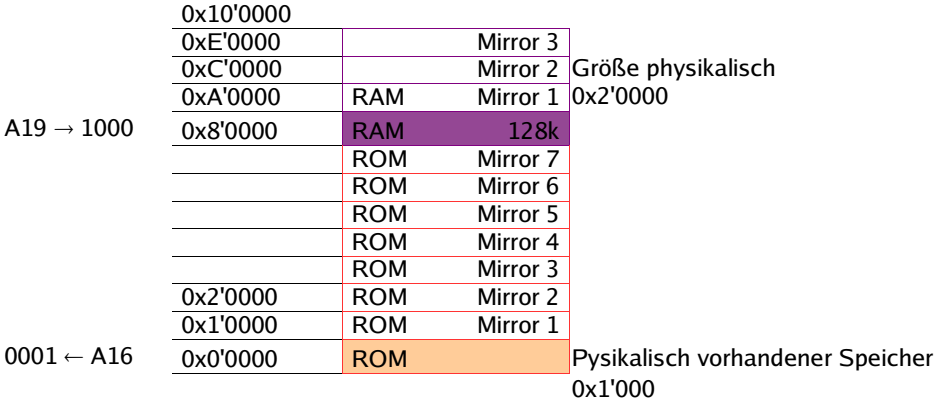


5.3 Adressraum-Dekodierung

5.3.1 Minimale Dekodierung

<Folie „K-5.3_AdressDekodierung.pdf“, Seite 1>

Adresse Adress-Bits	A19	A18	A17	A16	A16	...	A0	←Signale des Adressbusses
ROM	0	X	X	X	A15	...	A0	← Bausteine Beinchen
RAM	1	X	X	A16	A15	...	A0	



5.3.2 Maximale Dekodierung

<Folie „K-5.3_AdressDekodierung.pdf“, Seite 2>

ODER-Gatter: Nur wenn alle Eingänge auf 0 wird CS aktiviert

$\overline{LE}=1$ Di wird auf Qi durchgeschaltet
$\overline{LE}=0$ letzter Zustand der Qi bleibt gespeichert
$\overline{OE}=1$ Qi im Tri-State-Zustand
$\overline{OE}=0$ interne Qi werden auf die Pins Qi getrieben

A2	A1	A0	$\overline{y0}$	$\overline{y1}$	$\overline{y2}$	$\overline{y3}$	...
0	0	0	0	1	1	1	
0	1	0	1	1	0	1	
0	1	1	1	1	1	0	

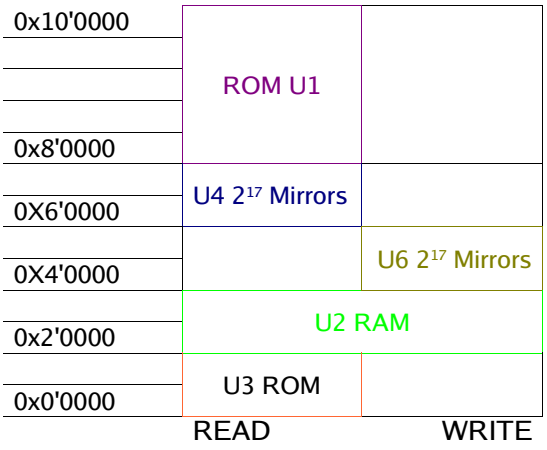
Man könnte in C den digitalen Ausgang U6 so ansprechen:

yx	Adressbus →	A19	A18	A17	A16	...	A0
y4 – y7	ROM U1	1	A18	A17	A16	...	A0
y1	RAM U2	0	0	1	A16	...	A0
y0	ROM U3	0	0	0	A16	...	A0
y2	Dig-In U4	0	1	0	X	X	X
y3	Dig-In U5	0	1	1	X	X	X
y4	Dig-Out U6	1	0	0	0	X	X

muss bei 0x0'0000 beginnen

Adresse 0x8'0000

theoretisch gibt das einen Konflikt mit ROM U1;
aber: auf U1 wird gelesen, auf U6 geschrieben;
sie kommen sich nie in die Quere!

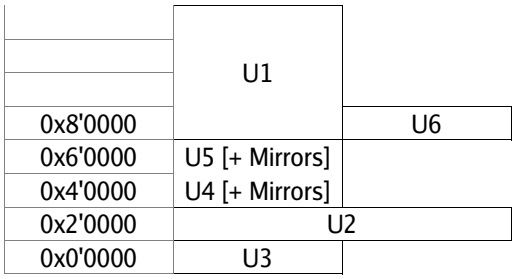



```
*(( unsigned char volatile xhuge * ) 0x8'0000 )) = 0xaF;
```

Erklärung:

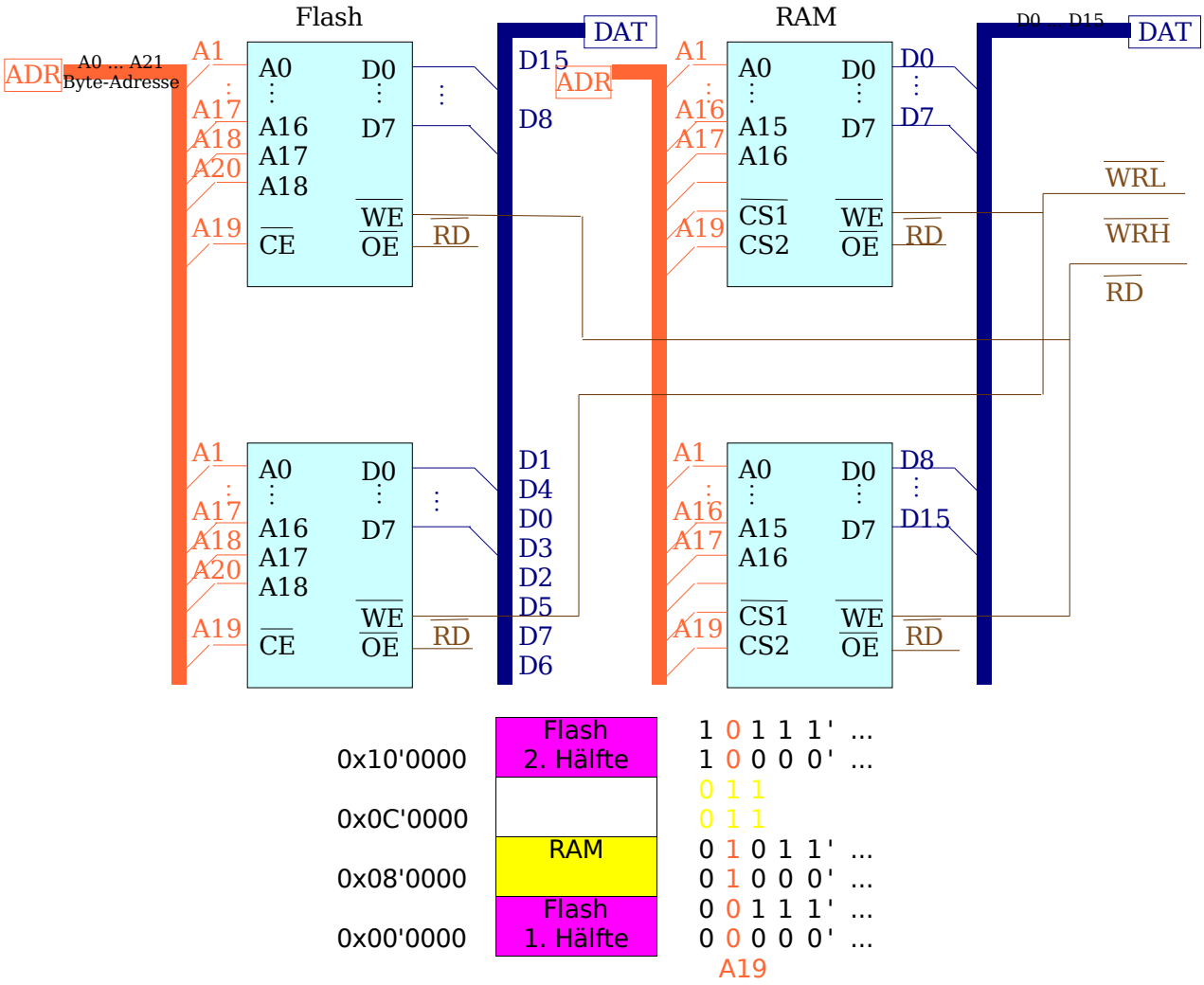
- volatile: teilt dem Compiler mit, dass er nichts optimieren oder wegschmeißen soll.
- xhuge: C167-spezifisches Konstrukt zum Ansprechen des Zeugs
- *((...) 0x8'0000): der Inhalt soll geschrieben werden.

Adressraum:



5.4 Übung 4

- Flash ab 0x00'0000 - 0x0F'FFFF
  2 * 512 kByte = 1 MByte
- RAM ab 0x08'0000
  2 * 128 kByte = 256 kByte
- Wort- und Bytezugriffe möglich



6 Speichertechnologien