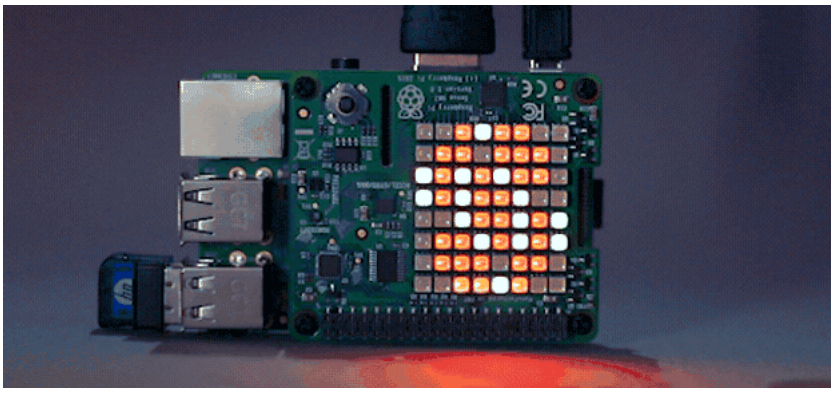
## Beschränkungen der ATmega Controller

| Parameter | Kursinformationen |
| --- | --- |
| Veranstaltung: | `Vorlesung Digitale Systeme` |
| Semester | `Sommersemester 2021` |
| Hochschule: | `Technische Universität Freiberg` |
| Inhalte: | `Überblick zur ATmega Familie` |
| Link auf den GitHub: | https://github.com/TUBAF-Ifi-LiaScript/VL_DigitaleSysteme/blob/main/lectures/03_Limitations.md |
| Autoren | Sebastian Zug, Karl Fessel & Andrè Dietrich |

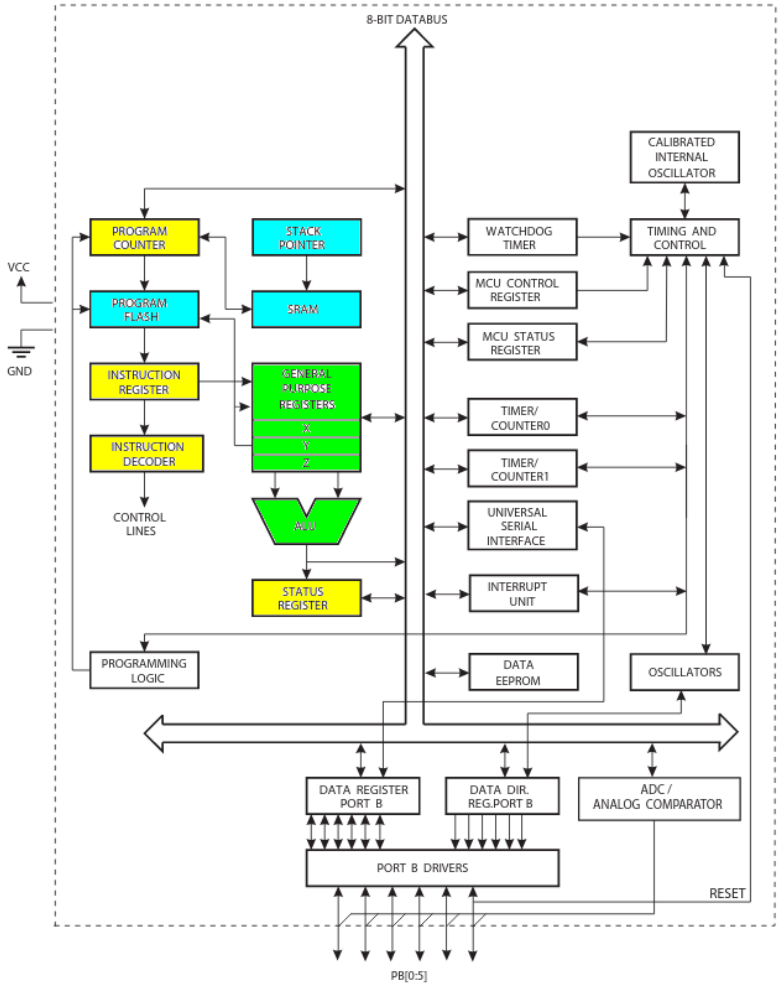


### Welche Einschränkungen ergeben sich aus der Architektur?

Warum sprechen wir im Zusammenhang mit den Controllern von fehlender Performance verglichen mit anderen Systemen?

## 8-Bit Datenbreite

**Figure 2-1.** Block Diagram




ATtiny Architektur [AtTinyArchitecture]

Die Festlegung auf 8-Bit Operanden und Ausgabe bei den arithmetisch/logischen Operationen erfordert umfangreiche Berechnungen schon bei bescheidenen Größenordnungen.

Simulation time: 00:08.593

**avrlibc.cpp**

```
1  #define F_CPU 16000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5
6  int main (void) {
7    Serial.begin(9600);
8
9    volatile int sample;
10
11   asm volatile("ldi  r16, 250" "\n\t"
12                "ldi  r17, 100" "\n\t"
13                "mul  r16, r17" "\n\t"
14                "movw %0, r0" "\n\t"
15                "eor r1, r1" "\n\t"
16              : "=a" (sample)
17              :
18              : "r16", "r17");
19
20   Serial.print("Das Ergebnis ist ");
21   Serial.println(sample);
22
23   while(1) {
24       _delay_ms(1000);
25   }
26   return 0;
27 }
```

```
Sketch uses 1434 bytes (4%) of program storage space. Maximum is 32256 bytes.
Global variables use 197 bytes (9%) of dynamic memory, leaving 1851 bytes for local variables. Maximum is 2048
bytes.
```

Wie lange dauert die Berechnung für die in Zeile 11 - 13 genannten Befehle?

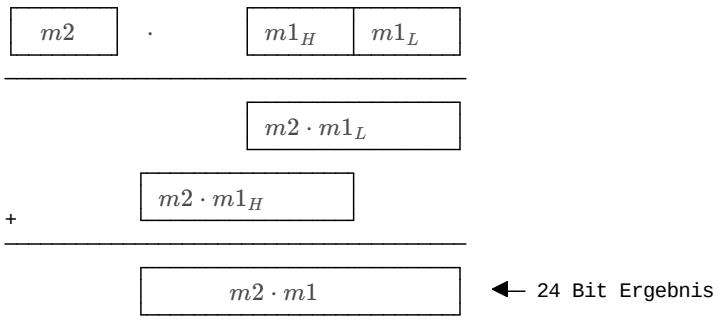
Warum scheitert das Ganze, wenn `r1` keine 0 enthält?

Mit dem 8-Bit Multiplikator decken wir aber nur Konstellationen hab, für die gilt das die Faktoren beide immer kleiner als 256 sein müssen. Um das Problem mit größeren Binärzahlen zu lösen, betrachten wir zunächst nur diese Kombination aus 16 und 8. Das Verständnis dieses Konzepts hilft, die Methode zu verstehen, so dass Sie später in der Lage sein werden, das 32-mal-64-Bit-Multiplikationsproblem zu lösen.

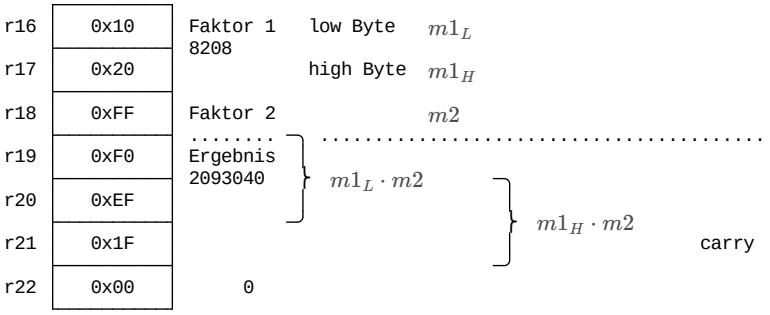
Die Mathematik dafür ist einfach, ein 16-Bit-Binär sind einfach zwei 8-Bit-Binäre, wobei der höchstwertige dieser beiden mit dezimal 256 oder hex 100 multipliziert wird. Das 16-Bit-Binär $m1$ ist also gleich $256 * m1_H$ plus $m1_L$, wobei $m1_H$ das MSB und $m1_L$ das LSB ist. Die Multiplikation von $m1$ mit dem 8-Bit-Binär $m2$ ist also, mathematisch formuliert:

$$m1 \cdot m2 = (256 \cdot m1_H + m1_L) \cdot m2 = 256 \cdot m1_H \cdot m2 + m1_L \cdot m2$$

Welche Abschnitte sind in der Brechnung notwendig?




Wir brauchen also nur zwei Multiplikationen durchzuführen und beide Ergebnisse zu addieren. Die Multiplikation mit 256 erfordert keine Hardware, da es sich um einen Sprung zum nächsthöheren Byte handelt. Lediglich der Übertrag bei der Additionsoperation muss beachtet werden.




.

**avrlibc.cpp**

```
#define F_CPU 16000000UL

#include <avr/io.h>
#include <util/delay.h>

int main (void) {
  Serial.begin(9600);

  volatile long sample;

  asm volatile("ldi  r16, 0x10" "\n\t"
               "ldi  r17, 0x20" "\n\t"
               "ldi  r18, 0xFF" "\n\t"
               "eor  %D0, %D0" "\n\t"
               "mul  r16, r18" "\n\t"
               "mov  %A0, r0" "\n\t"
               "mov  %B0, r1" "\n\t"
               "mul  r17, r18" "\n\t"
               "add  %B0, r0" "\n\t"
               "mov  %C0, r1" "\n\t"
               "brcc NoInc" "\n\t"
               "inc  %C0" "\n\t"
               "NoInc:" "\n\t"
               "eor r1, r1" "\n\t"
               ""
              : "=r" (sample)
              :
              : "r16", "r17", "r18" );


  Serial.print("Das Ergebnis ist ");
  Serial.println(sample);

  while(1) {
      _delay_ms(1000);
  }
  return 0;
}
```

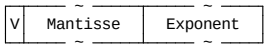
```
Sketch uses 1450 bytes (4%) of program storage space. Maximum is 32256 bytes.
Global variables use 197 bytes (9%) of dynamic memory, leaving 1851 bytes for local variables. Maximum is 2048
bytes.
```

Für die Muliplikation von größeren Werten wird die Berechnung entsprechend aufwändiger.

[AtTinyArchitecture] Firma Microchip, Handbuch AtTiny Family, https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2586-AVR-8-bit-Microcontroller-ATtiny25-ATtiny45-ATtiny85_Datasheet.pdf

## Fehlende Fließkommaeinheit

Die Gleitkommadarstellung besteht dann aus dem Vorzeichen, der Mantisse und dem Exponenten. Für binäre Zahlen ist diese Darstellung in der IEEE 754 genormt.




V=Vorzeichenbit

```
1      23          8          = 32 Bit (float)
1      52         11          = 64 Bit (double)                          .
```

**Merke:** Die Verrechnung von Gleitkommazahlen ist entsprechend aufwändig:

1. Homogenisierung der Exponenten und Mantissen
2. Berechnung des Ergebnisses
3. Normierung des Resultats

## Fehlende Festkommaeinheit

Neben den Fließkommadarstellungen lassen sich auch Festkommakonzepte für die Darstellung gebrochener Zahlen in Hardware/Software umsetzen. Dabei wird die Speicherbreite in den Anteil vor und nach einer spezifischen und unveränderlichen Kommaposition eingeteilt.

Ein Beschreibungsformat dafür ist die Q-Notation bei der die Anzahl der Nachkommastellen (und optional die Anzahl der ganzzahligen Bits) angegeben wird. Eine Q15-Zahl hat z. B. 15 Nachkommastellen; eine Q1.14-Zahl hat 1 ganzzahliges Bit und 14 Nachkommastellen.

**Achtung:** Für vorzeichenbehaftete Festkommazahlen gibt es zwei widersprüchliche Verwendungen des Q-Formats. Bei der einen Verwendung wird das Vorzeichenbit als Ganzzahlbit gezählt, in der anderen Variante jedoch nicht. Zum Beispiel könnte eine vorzeichenbehaftete 16-Bit-Ganzzahl als Q16.0 oder Q15.0 bezeichnet werden. Um diese Unklarheit zu beseitigen wird teilweise ein U für `unsigned` eingefügt.

| Konfiguration | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 |
|---|---|---|---|---|---|---|---|
| UQ0.8 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| UQ1.7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| UQ2.6 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

| Konfiguration | Auflösung | größte Zahl | kleinste Zahl |
|---|---|---|---|
| `Qm.n` | $2^{-n}$ | $2^{m-1} - 2^{-n}$ | $-2^{m-1}$ |
| `UQm.n` | $2^{-n}$ | $2^{m} - 2^{-n}$ | $0$ |

Eine 16 Bit breite, vorzeichenbehaftete Festkommazahl `Q15.1` kann also Zahlenwerte im Bereich $[-16384.0, +16383.5]$ abbilden. Die Auflösung der Darstellung ist $2^{-n} = 0.5$

**Merke:** Anders als eine Fließkommazahl ist die Auflösung der Festkommazahl konstant!

Bei der Rechnung mit Festkommazahlen werden die binären Muster prinzipiell so verarbeitet wie bei der Rechnung mit ganzen Zahlen. Festkomma-Arithmetik kann daher von jedem digitalen Prozessor durchgeführt werden, der arithmetische Operationen mit ganzen Zahlen unterstützt. Dennoch sind einige Regeln zu beachten, die sich auf die Position des Kommas vor und nach der Rechenoperation beziehen:

- Bei Addition und Subtraktion muss die Position des Kommas für alle Operanden identisch sein. Ist dies nicht der Fall, sind die Operanden durch Schiebeoperationen entsprechend anzugleichen. Die Kommaposition des Ergebnisses entspricht dann der Kommaposition der Operanden.
- Bei Multiplikation entspricht die Anzahl der Nachkommastellen des Ergebnisses der Summe der Anzahlen der Nachkommastellen aller Operanden.
- Die Wortbreite des Endergebnisses wird auf die gewünschte Breite reduziert. Dabei wird häufig Sättigungsarithmetik und Rundung verwendet.

```
int16_t q_add(int16_t a, int16_t b)
{
    return a + b;
}

int16_t q_add_sat(int16_t a, int16_t b)
{
    int16_t result;
```

Der Dynamikbereich von Festkommawerten ist zwar wesentlich geringer als der von Fließkommawerten mit gleicher Wortgröße. Warum sollte man dann einen Mikrocontroller oder Prozessor mit Festkomma-Hardwareunterstützung verwenden?

- **Größe und Stromverbrauch** - Die logischen Schaltungen der Festkomma-Hardware sind viel weniger kompliziert als die der Fließkomma-Hardware. Das bedeutet, dass die Festkomma-Chipgröße im Vergleich zur Fließkomma-Hardware kleiner ist und weniger Strom verbraucht.
- **Speicherverbrauch und Geschwindigkeit** - Im Allgemeinen benötigen Festkommaberechnungen weniger Speicher und weniger Prozessorzeit.
- **Kosten** - Festkomma-Hardware ist kostengünstiger, wenn Preis/Kosten eine wichtige Rolle spielen. Wenn digitale Hardware in einem Produkt verwendet wird, insbesondere bei Massenprodukten, kostet Festkomma-Hardware viel weniger als Fließkomma-Hardware.

Wie ist das Ganze implementiert? Seit der Version 4.8 integriert der avr-gcc eine entsprechende Bibliothek `stdfix.h`, die vordefinierte Typen integriert:

| Typname | Typ | Größe in Byte | QU | Q |
|---|---|---|---|---|
| _Fract | short | 1 | 0.8 | ±0.7 |
| | long | 4 | 0.32 | ±0.31 |
| | long long | 8 | 0.64 | ±0.63 |
| _Accum | short | 1 | 8.8 | ±8.7 |
| | long | 4 | 32.32 | ±32.31 |
| | long long | 8 | 16.48 | ±16.47 |

**Merke:** Daneben existieren verschiedene andere Festkommabibliotheken, die andere Konfigurationen unterstützen und verschiedene Implementierungen aufzeigen.

Lassen Sie uns einen genaueren Blick auf die Implementierung werfen. Im Codebeispiel, dass Sie im Projektordner XXX finden, addieren wir zwei Variablen unterschiedlichen Formates.

**FixedPoint.c**

```
#define F_CPU 16000000UL

#include <avr/io.h>
#include <stdfix.h>

int main (void) {

  unsigned short _Accum fixVarA = 1.5K;
  short _Accum fixVarB =  -1.5K;
  long _Accum fixResult = fixVarA * fixVarB;

  while(1);
  return 0;
}
```

Für die `variableA` ergibt sich dabei folgender Auszug des Programmspeichers, sofern das Beispielprogramm ohne Optimierung übersetzt wird.

```
short _Accum fixVarB =  -1.5K;
11c:  80 e4         ldi r24, 0x40 ; 64
11e:  9f ef         ldi r25, 0xFF ; 255

  +--------+
```

## Vergleich der Softwarelösungen auf dem AVR

Um eine Evaluation durchzuführen wurde der Python Wrapper `pysimavr` für die AVR Core Simulation genutzt.

https://github.com/buserror/simavr

Im Projektordern finden Sie unter `../codeExamples/avr/fixedPoint/pySimAVR` das Miniprojekt. Dabei sind zwei Beispiele vorgesehen:

- Evaluation der Laufzeit mittels UART Ausgaben
- Evaluation der Laufzeit über togglende Pins

Im Ergebnis zeigt sich folgendes Bild:

| Variable | Dauer |
|---|---|
| `_delay_ms (100);` | 100000.12500 us |
| `unsigned short _Accum` | 2771.68700 us |
| `unsigned long _Accum` | 45760.37500 us |
| `long _Accum` | 50463.25000 us |

## Aufgaben

- [ ] Integrieren Sie die Berechung im Beispiel vcdBased auf Basis von `float` und `double` Werten