Hardwarenahe Programmierung (= Assembler)

Klaus Kusche

Inhalt

- Grundlagen und Überblick
- Programmierung x86-64 unter Windows:
  - x86-Architektur
  - Assembler-Syntax
  - Win64 ABI
- Programmierung Atmel ATmega 8515:
  - ATmega-Architektur
  - Beispiele für I/O
  - Interrupts

Ziel

- **Verständnis HW:**
  Prozessor, Mikrokontroller, Interrupts, I/O, ...

  ==> Kennenlernen von *2 Prozessor-Architekturen*
- **Verständnis SW:**
  "Was macht der Compiler aus meinem C-Code?"

  ==> Kennenlernen des Win64-ABI als Beispiel
- **Grundkenntnisse & Denkweise der Assembler-Programmierung**
  - im *Zusammenspiel mit C-Programmen*
  - auf "*nackter*" Hardware (ohne OS)

Nicht Ziel

- Tiefere Assembler-Kenntnisse
  (z.B. komplexe Makros)
- Praktische Assembler-Programmier-Fertigkeiten
  auf sofort beruflich einsetzbarem Level
- Vollständige Abdeckung der Plattformen
  - x86: *Ohne* Gleitkomma, SSE/AVX, …
    Nur Anwendungs-Ebene
    (ohne Interrupts, Betriebssystem, …)
  - ATmega: I/O usw. nicht flächendeckend

Voraussetzungen

- ***Rechner-Architektur*:**
  Aufbau und Funktionsweise eines Rechners
  (Prozessor, Speicher, ...)
- **Programmierung:**
  C
  CodeBlocks (Compiler, Linker, ...)

Motivation

- Manche Dinge *muss* man in Assembler codieren...
- Kenntnis

  "Was macht der Compiler aus meinem C-Code?"

  - hilft, *effizienteren C-Code* zu schreiben

  - hilft beim *Debuggen*

- Prozessor-Verständnis und
  Verständnis für die Implementierung
  von Programmiersprachen gehört zur

  "informatischen Allgemeinbildung"!

Ein “Programm” aus Prozessor-Sicht

… ist eine *Folge vieler 0 und 1* im Speicher

Genauer:

Ist eine Folge von ***“Maschinenbefehlen”***,
die der Prozessor *der Reihe nach* abarbeitet.

Jeder Maschinenbefehl

… ist **1-32 Bytes lang**

… und durch ***irgendwelche Bitmuster*** codiert
(*Art* des Befehls, *Operanden*, …)

Was ist Assembler? (1)

Die ***Sprache*** "Assembler":

"Lesbare" Notation *für Maschinenbefehle,*
d.h. ***textuelle Darstellung von Binärcode***
(statt Hex oder Binär,
weil Hex ist unlesbar/unschreibar)

Das ***Programm*** "Assembler":

Erzeugt aus einem *Assembler-Quelltext*
einen binären *Object- oder Exe-File*
(wie ein Compiler)

Was ist Assembler? (2)

Die Sprache “Assembler”:

Darstellung des *Binärcodes* *“Eins zu eins”*:

1 Maschinenbefehl = 1 Zeile Assembler-Programm

Aufeinanderfolgende Zeilen im Programm
==> *Aufeinanderfolgend gespeicherte*
Befehle oder Daten im Binärcode

Listing-Ausgabe beim Assembler einschalten
==> Enthält *für jede Quellcode-Zeile*
- den resultierenden *Binärcode (in Hex)*
- dessen *Adresse* im Speicher

Was ist Assembler? (3)

Allgemeine Syntax einer Zeile:

[Label:] *Befehl* *[Operanden]* *[Kommentar]*

- **Befehl:** ***"Mnemonic"*** = *Merkwort* (Abkürzung)
  z.B. **JMP**, **MOV**, **DIV**, **RET**
- **Label:**
  Name als Symbol für eine Adress-Konstante,
  nämlich für die *Speicheradresse derjenigen Zeile*,
  vor der das Label steht (Code oder Daten)
- **Operanden:** *Zahlen, Register-Namen, Labels, ...*

Was ist Assembler? (2)

Beispiel für x86:

Assembler: **`ADD eax, edx`**

Binärcode: 0000 0001 1101 0000

Bedeutung:

“Addiere den Inhalt des 32-Bit-Registers DX zum 32-Bit-Register AX”

Assembler Programmierstil (1)

Assembler kennt

- ... ***keine "Strukturierung"***
  mit **`{ }`** oder Einrückung
- ... ***keine `if`'s oder Schleifenbefehle***
  (weil die Prozessoren nur Sprungbefehle haben und keine Schleifen-Befehle!)

==> Assembler tendiert von Natur aus zu

"Spaghetti-Code"!

Trotzdem (gerade deshalb) ...

Assembler Programmierstil (2)

- ***In Schleifen denken***, nicht nur in Sprüngen!

  ==> *Nicht wild herumspringen*,
  vor allem nicht in Schleifen hineinspringen!

  ==> Zuerst *strukturierten Algorithmus überlegen*!!!

- *Keinen ewig langen Spaghetti-Code* produzieren!

  Auch Assembler kennt *Funktionen*
  ==> zu lange Codestücke in Funktionen *aufteilen*!

- Idente Codestücke *nicht immer wieder kopieren*!

  ==> *Makros* oder *Funktionen* verwenden!

Assembler Programmierstil (3)

Makros:

Für *immer wieder benötigte Befehls-Folgen*:
Machen den Code *übersichtlicher / lesbarer*!

Wie **`#define`** in C:

Werden vom Assembler (zumindest gedanklich)
vor der Übersetzung im Quellcode ***textuell ersetzt***

Unterschied zu Funktionen:

==> Kosten zu Laufzeit *keinen Overhead*

==> Aber machen den *Code größer*

Wie schreibt man Assembler-Code?

Als *normalen Programmtext* mit einem Editor

- Entweder in *eigenen Source-Files*
  (Endung unter DOS/Windows meist **`.asm`**,
  unter Unix / Linux meist **`.s`** oder **`.S`**)
- oder bei *ganz kurzen* Code-Stücken
  "inline" in einem C-File:
  `__asm__ (" ... ")`
  (Compiler-abhängig, machen wir *hier nicht*!)
- bzw. als *"Intrinsic-Funktionsaufruf"*
  (vordef. Funktionen, übersetzt in *genau 1 Befehl*)

Warum Assembler? (1)

Früher:

Assembler war mit Abstand
schnellste Programmiersprache!

Heute:

Gilt allgemein noch für "*kleine*" Prozessoren.

Der *Geschwindigkeits-Vorsprung*
gegenüber *optimierenden Compilern*
bei *"normalen"* Anwendungen
auf *"großen"* Prozessoren ***ist praktisch Null!***

Warum Assembler? (2)

Aber:

Assembler ist immer noch
deutlich speicher-effizienter!
(primär bei Daten, auch bei Code)

Vor allem auf *Kleinst-Systemen*:

- Mikrokontroller haben ev. *zu wenig RAM & ROM* für eine Programmierung in Hochsprache!
- Jedes Kilobyte mehr erhöht die *HW-Kosten*, bei Mikrokontrollern in Millionen-Stückzahlen fallen wenige Cent ins Gewicht!

Warum Assembler? (3)

Auf *ganz kleinen Plattformen*

- gibt es teilweise gar *keine* (guten) *Compiler*
- gibt es teilweise *keine Standard-Libraries*

Weil:

- Aus Speicherplatz-Gründen *sinnlos*
- Portierung eines optimierenden C-Compilers incl. Standard-C-Libraries *kostet* (ev. Millionen!), Entwicklung eines Assemblers ist viel billiger

Warum Assembler? (4)

Heute vor allem:

Assembler wird verwendet,
*wo **Maschinenbefehle** vorkommen,*
*die sich **in Hochsprachen-Programmen***
gar nicht ausdrücken bzw. ansprechen lassen!

- *Zugriff* auf I/O, Interrupts, CPU-Spezial-Register
- *Privilegierte Befehle* für's Betriebssystem (MMU, Prozesswechsel, Prozess-Synchronisierung)
- *Multimedia-Befehle* (SSE, AVX), Krypto-Befehle, ...

… und warum nicht? (1)

Assembler-Entwicklung

- ist viel *langsamer* / *teurer* / *aufwändiger*
- ist *fehleranfällig*, *schwer zu lesen* & zu warten
- ist *Prozessor-abhängig* und nicht portabel (bei Wechsel auf andere Prozessor-Architektur: *Komplette Neuentwicklung* nötig!!!),
- ist ev. *Betriebssystem-abhängig* (*umschreiben* bei Betriebssystem-Wechsel!)
- kann meist nicht auf vordefinierte Libraries zurückgreifen

… und warum nicht? (2)

Historisch:

- Zu Beginn der EDV (< 1960) gab es *nur* Assembler
- Die Einführung von *Hochsprachen & Compilern* war *Voraussetzung für die moderne EDV*!
  (wäre nur mit Assembler nicht möglich gewesen!)

Daher:

- *So viel wie möglich in C* schreiben!!!
- Nur die *Teile*, wo es sinnvoll / notwendig ist, in *Assembler* schreiben!

Anwendung (1)

"Ganz kleine" Mikrokontroller
(8/16 bit, wenige KB RAM & ROM):

- Oft *komplett in Assembler* programmiert
  (weil ohnehin nur sehr wenig & einfacher Code)
- Meist *auf nackter Hardware* programmiert
  (kein Betriebssystem, keine Libraries, ...)

Anwendung (2a)

Betriebssystem, BIOS &
Code größerer Steuerungen:

C mit einzelnen, *kurzen Assembler-Funktionen*

- … für den *Zugriff auf Hardware und I/O-Bausteine*
  (*"Device Driver"*)
- … rund um *privilegierte Befehle*
  für Betriebssystem-Funktionen
  (MMU- und Cache-Verwaltung,
  Synchronisierungs-Befehle, …)

Anwendung (2b)

Beispiel **Linux-Kernel**:

- Knapp *1500 Assembler-Files*
  (zum Vergleich: › 25000 C-Files),
  rund *50 pro Prozessor-Plattform*
- Meist
  - *Boot*-Code
  - *Crypto*-, String- und anderen Lib-Funktionen
  - ev. *Emulation der Gleitkomma-Befehle*
  - *Syscall & VM-Umschaltung*, Sync-Befehle
- Plus ~ *3000 Stellen* mit kurzem *Inline-Asm-Code*

Anwendung (3)

In "normalen" Programmen (C, ...):

Einzelne Assembler-Funktionen z.B. für

- *Multimedia-Codecs*, jpeg, Bild-Filter, ...
- *Verschlüsselung* & Hashfunktionen, Kompression, Mining, ...
- *Wissensch. Berechnungen* (Matrizenrechnung, ...), Rechnen mit beliebig langen Int's

Verwenden z.B. *AVX- und SSE-Befehle*,
die sich aus reinem C-Code nicht erzeugen lassen.

Wozu braucht man Assembler noch?

- Beim *Compiler-Bau*,
  zum Schreiben von JIT's, ...
- Zum *Reverse Engineering*
  (z.B. von Schadcode)

Was muss man wissen? (1)

"Architektur" des Prozessors:

- *Register*, Flags, ...
- *"Befehlssatz"*:
  - Welche Befehle gibt es?
  - Welche Wirkung haben sie?
  - Wie viele / welche Operanden sind möglich?
- *Operanden* und Adressierungsarten

Was muss man wissen? (2)

- ***Syntax des Assemblers*** *(nicht standardisiert!)*

  - *Schreibweise für Operanden*, Labels, ...
    (u.a. bei 2 Operanden:
    "rechts nach links" oder *"links nach rechts"*?)

  - Syntax von *Kommentaren*

- ***Assembler-Pseudo-Befehle*** *(nicht standardisiert!)*

  sind *Anweisungen an den Assembler* oder Linker,
  resultieren *nicht* in Maschinenbefehlen:

  z.B. *"Reserviere n Bytes Speicherplatz"*

Was muss man wissen? (3)

Bei "nackter" Hardware bzw. für I/O-Zugriff:

- ***Speicherorganisation, I/O-Adressen,***
  reservierte Speicherstellen, ...:

  "Was steht an welcher Adresse im Speicher?"

- Aufbau / Inhalt von
  Spezial-Registern, I/O-Baustein-Registern, ...
- Handling von ***Interrupts***, von Boot bzw. Reset

Was muss man wissen? (4)

Bei *Kombination mit C*:

Das "ABI" = "Applikation Binary Interface"

legt die *Regeln* fest,
an die sich *Compiler* verschiedener Sprachen
sowie *Assembler-Programme* halten müssen,

damit Daten und Funktionen
auf Binärcode-Ebene zueinander kompatibel sind.

Ev. auch:

Wie ruft man das Betriebssystem direkt auf?
(ganz anders als Funktions-Aufruf!)

ABI versus API

API = "Application Programming Interface"

=

Welche Typen, Funktionen, Klassen, ...
bietet mir eine Library an,
d.h. was kann ich in meinem Code nutzen?

... auf ***Source-Code-Ebene*** (C/C++),
unabhängig von Plattform, Binärcode, ...!

... in der Praxis: API entspricht *Header-File*

ABI (1)

Festlegung der ***Daten***:

- *Größe* von Pointern und anderen Datentypen
- *Endiness* der Daten im Speicher
- *Alignment* (==> Füllbytes)
- *Was liegt wo* im Speicher?

  - *Adressierung* von Konstanten, glob. Variablen, ...

  - *"Sections"* im Object-Files bzw. Exe-File:
    ... für *Code*, Konstanten
    ... für *globale Daten* mit / ohne Init-Wert

ABI (2)

Festlegung der ***Funktionsaufrufe***:

- Aufbau des *Stacks*, lokale Variablen, **`alloca`**
- *Parameter-Übergabe und Returnwert*, *Aufräumen* nach einem Funktionsaufruf, **`varargs`**
- *Register-Verwendung*:
  - Register mit *fixer* Bedeutung
  - *Frei verwendbare* Register (*"Caller saves"*)
  - Zu erhaltende bzw. vor Benutzung *zu sichernde* Register (*"Callee saves"*)

ABI (3)

- *Programmstart* und -ende,
  Multithreading, Thread-lokale Variablen
- *Exception-Handling* und Signale,
  `setjmp` / `longjmp`
- *Dynamisches Linken, "lazy Binding" (GOT)*
- Andere Anforderungen
  (z.B. PIC = *"Position Independen Code"*)
- C++: *Objekte & vtable, Methoden-Pointer, RTTI, ...*

“Bitbreite” von Prozessoren (1)

Z.B. ***“32-bit-Prozessor”***

==> Die *allgemeinen Register* sind 32 bit breit

==> Die arithmetischen und logischen
Operationen auf Int’s
können *32 bit auf einmal verarbeiten*

(FPU & Multimedia-Einheiten können *breiter* sein,
Adress-Rechnungen können *breiter* sein,
Bus zu RAM & I/O kann *breiter oder schmäler* sein)

“Bitbreite” von Prozessoren (2)

“1 Wort” =

Ein Int mit der *“natürlichen” Größe des Prozessors*

(d.h. bei 32 bit Prozessor: 4 Bytes)

Bei **x86**, *historisch bedingt* aus 8086-Zeiten:

- **1 Word = 2 Bytes (16 Bit)**
- **1 Double-Word = 4 Bytes (32 Bits)**
- **1 Quad-Word = 8 Bytes (64 Bits)**

Register (1)

- *Speicherzelle* (meist für 1 Wort) *in der CPU*
- Dient als *Operand für Maschinenbefehle*

Unterscheide:

- Register mit *hardwaremäßiger Funktion*
  Beispiele:
  - **IP (Instruction Pointer) = PC (Program Counter)**
    zeigt immer *auf nächsten auszuführenden Befehl*
  - **SP (Stack Pointer)**
    zeigt immer *auf unterstes belegtes Byte am Stack*
  - **Flags (Status-Bits)**

Register (2)

- *Allgemeine*, frei verwendbare *Register* (Char/Int)
- *Gleitkomma-Register*,
  Multimedia- bzw. *Vektor-Register*
- Register von *On-Chip-I/O-Devices*
- Register für die *MMU*, den *Interrupt-Controller*, ...

Bei manchen Architekturen (nicht x86):
2 oder mehr Register-Sätze, *wechselweise* sichtbar:
1 Satz für normale Prog., 1 Satz für OS & Interrupts

==> *Macht Syscalls & Interrupts schneller!*

Flags (1)

Flags = Einzelne Status-Bits,
zusammengefasst zu einem *Spezial-Register*

Die üblichen *arithmetischen Flags*:
Z (Zero), S (Sign),
C (Carry = Übertrag, *unsigned*), **O (Overflow,** *signed*),
ev. P (Parity), "Half Carry" (Übertrag 4. --› 5. bit)

- *Gesetzt* von allen *arithmetisch / logischen Befehlen*
- *Geprüft* u.a. von *bedingten Sprüngen*
- Verwendet z.B. auch bei *"Add with Carry"*

Flags (2)

Enthält auch Bits,
die der *Prozessor-Steuerung* dienen.

Wichtigstes bei x86:

IF = "Interrupt Enable Flag"

- Mit *expliziten Befehlen setz- und löschbar*
- *Sperrt* oder erlaubt alle *Interrupts*

Befehle (1)

- **Laden und Speichern**
  - auch *Push & Pop*
  - ev. Conditional Move
- **Rechen-Befehle:**
  - Arithmetische und logische Operationen
  - Bit *Shift & Rotate*
  - *Vergleiche*
  - Increment und Decrement
  - x86: **LEA** ("Load effective Address"
    = Adresse berechnen)

Befehle (2)

- Programmablauf:
  - *Unbedingte* **Sprünge**
  - *Bedingte* **Sprünge**
  - **Call & Return**

  Sprungziel:
  - *Relative* oder *absolute* Adresse
  - Beliebiger *Register- oder Speicher-Operand*
    ==> *indirekter* Sprung
- ... und viele weitere

Operanden

Für Rechenbefehle und Lade-/Speicher-Befehle:

- **Register**
- **Konstante** (*"Immediate"*)
- **Speicher**-Operand
  (durch Angabe der *Adress-Berechnung*)

Für Sprünge und Calls:

- **Label**
- Bei indirektem Sprung:
  Register oder Speicher-Operand

Adressierungsarten (1)

Bei allen Befehlen,
die Daten aus dem Speicher *lesen*
oder in den Speicher *schreiben*:

Wie wird die Speicher-Adresse angegeben?

- **Register**

  Der Register-*Inhalt* wird *als Adresse* verwendet
  (das *Register enthält einen Pointer* auf die Daten)

  Auf jeder Plattform vorhanden
  Auf manchen Plattformen:
  Mit *Increment/Decrement* des Registers

Adressierungsarten (2)

- **Register + Konstante** (*"Offset"*, *"Displacement"*):

  Für:
  - *Lokale Variablen* (*Register ist SP* oder BP)
  - Daten in einer *Struktur* (Register enth. *Pointer*)

  Auf fast jeder Plattform vorhanden

- **IP-relative Adresse**, d.h. IP +- Konstante
  (z.B. für *Konstanten*, *relativ* zum Code)

  Auf vielen Plattformen vorhanden
  (x86 erst seit Erweiterung auf 64 bit)

Adressierungsarten (3)

- **Fixe absolute Adresse**
  (oft als *Label* angegeben, nicht als Zahl)

  Für *Konstanten, globale Daten, I/O-Speicherzellen*

  Auf CISC-Plattformen fast immer vorhanden,
  auf RISC meist *Umweg mittels Laden in ein Register* nötig

- **Andere Adressierungart + weiteres Register**
  (ev. mal 1/2/4/8)
  (für *Arrays*, Register ist *Index*)

  Z.B. auf x86 vorhanden,
  auf RISC Adresse mit mehreren Befehlen ausrechnen

I/O-Zugriff

2 Varianten der *Anbindung von I/O-Bausteinen*:

- Früher oft: ***Eigene Befehle*** nur für I/O-Zugriffe:

  I/O-Bausteine hängen (gedanklich, nicht real!) an eigenem Bus
  mit *eigenem Adressenraum (I/O-Port-Nummern)*,
  sind mit normalen Speicherzugriffsbefehlen nicht erreichbar

  Z.B. "altes" x86-I/O *(Original-IBM-PC-Schnittstellen)*

- Heute meist ***"Memory Mapped I/O"***:

  I/O-Bausteine hängen am normalen Speicher-Bus
  Teile der Speicher-Adressen sind für I/O-Bausteine reserviert
  I/O-Zugriff mit ganz "normalen" Speicherzugriffsbefehlen

Interrupts (1)

Interrupt =

Unterbrechung des gerade laufenden Programms

Quellen:

- *I/O-Bausteine*, Timer, Parity Error, ...
- *MMU* (*Page Fault*)
- *CPU* (illegaler Befehl, /0, alignment error, ...), *FPU*
- *Befehl* im Programm:
  - *Breakpoint* (INT 3)
  - *BIOS Call* (10, 13), *System Call* (DOS: 21, Linux: 80)

Interrupts (2)

Was passiert in etwa (je nach Plattform)?

- Weitere *Interrupts* werden *gesperrt*
- Prozessor-*Status* wird *gesichert*
- Prozessor wechselt in den *privilegierten Modus*
  (und in den Betriebssystem-Adressraum)
- ***"Interrupt Handler"*** (auch *"ISR"*) wird *ausgeführt*

Ende des Interrupt Handlers (eigener Befehl **IRET**):

- Alles wieder *zurück*
- Meist: *Unterbrochenes Programm macht weiter*

Interrupts (3)

Wie findet der Prozessor zum Interrupt Handler?

- Jede *Interrupt-Quelle*
  hat eine *fix zugeordnete Nummer*
- An *fester Stelle im Speicher* (z.B. ab *Adresse 0*)
  liegt eine Tabelle mit den ***"Interrupt-Vektoren"***
  = den *Code-Adressen der Interrupt-Handler*
- Die Interrupt-Nummer ist *Index* in diese Tabelle

==> *Kommt Interrupt Nummer i,*
springt der Prozessor zum Interrupt-Handler dessen Adresse im i-ten Eintrag der Interrupt-Tabelle steht.

Zähler

Wichtigster I/O-Baustein & wichtigste Interrupt-Quelle:

- *Zählen* externer Impulse, Frequenz *messen*,
  Messung der Dauer externer Signale
- Generierung *periodischer Interrupts* (*"System Tick"*)
- Interne *Zeitmessung*, Systemuhr
- Realisierung von *Programm-Verzögerungen*
  (*"wecke mich in 100 ms wieder auf"*)
- Generierung *externer Signale*
  bestimmter Dauer / Frequenz, PWM

JTAG

Standardisierte *4-Pin-Schnittstelle*,
auf vielen "*kleinen*" Prozessoren vorhanden

*Liest und schreib*t Register / RAM / Flash *autonom*
(d.h. *ohne* Zutun von Prozessor oder Software!)

Ursprünglicher Zweck:
Debuggen bei hängendem System

Zweitnutzen:
Zum *Not-Laden des Flash*
(wenn "zer-flasht" bzw. wenn sonst nichts mehr geht)

Aber: *HW-Hacker-Eingang*, Sicherheitslücke!

RISC & CISC (1)

RISC = "Reduced Instruction Set Computer"

CISC = "Complex Instruction Set Computer"

CISC: Durchgehend seit ~1955, *heute*: **x86, zSeries**

Um 1980: Alternativer Ansatz RISC
Heute: **ARM, Power/PowerPC, Sparc, MIPS, ...**
Freie Cores: **OpenRISC, RISC V, OpenSPARC, LEON, ...**

Implementierung von Cores heute:
Meist *gemischt aus RISC- und CISC-Ideen*,
aus Hardware-Sicht *keine Unterscheidung* mehr!

"*Kleine*" Prozessoren: Heute *eher RISC* (z.B. ATmega)

RISC & CISC (2)

Damalige **Ziele** von RISC:

- Aus möglichst *wenig Transistoren* (*teuer & knapp!*) möglichst *viel Rechenleistung* herausholen
- Möglichst *hoher Takt*
  (früher *2-3 mal höher* als x86 gleicher Generation)

Randbedingung *(gilt heute nur mehr teilweise!)*:

- *Caches* im Vergleich zum Core *schnell*:
  Cache-*Zugriffszeit 1 Takt*,
  Cache-*Bandbreite* 1 Wort I + 1 Wort D pro Takt

RISC & CISC (3)

Umsetzung aus HW-Sicht:

- ***Direkte Dekodierung und Ausführung*** der Befehle durch Logik-Schaltungen, ***keine Mikroprogramm-gesteuerte*** Befehlsverarbeitung mehr
- ***1 Takt pro Befehl, 1 Befehl pro Takt***

RISC & CISC (4)

Auswirkung aus **Assembler-Sicht**:

- ***Wenig Befehle, einfache Befehle***
  Wenig Befehls-Formate, ***alle fix 4 Bytes lang***
  (z.B. 32 bit Konstante laden: Aufteilung auf 2 Befehle)

  (CISC: Mehr Befehle, variabel lang, komplex)

- Nur sehr ***einfache Adressierungsarten***
  ==> Mehrere Befehle zum Berechnen komplexer Adressen

  (CISC: Bis zu vier Adress-Bestandteile, Incr/Decr, ...)

- ***Viele allgemeine Register*** (meist 32)

  (CISC: Damals meist nur 8, max. 16)

RISC & CISC (5)

Auswirkung aus **Assembler-Sicht:**

- ***Load-Store-Architektur:***
  - *Rechenbefehle* arbeiten *nur auf Registern*,
    meist mit *3 Operanden*
    (reg3 = reg1 op reg2)
  - *Separate Lade- und Speicherbefehle* transferieren
    Daten zwischen Speicher und Register

  (bei **CISC:** *Rechenbefehle* können
  Operanden im Speicher direkt ansprechen)

- Bei bedingten Sprüngen: ***Delay Slot***

RISC & CISC (6)

- *Ziel* der höheren Geschwindigkeit wurde *erreicht*
- *Compilerbau* war *einfacher*
- *Entwurf* der Cores war *einfacher*

Nachteil:

Binärcode für dasselbe Programm
ist mehr als doppelt so groß
wie bei CISC-Prozessoren!

Grobe Prozessor-Einteilung (1a)

"Kleine" Mikrokontroller (µC, MCU)

8 oder 16 bit CPU (1 ... 50 MHz, einige mW),
keine FPU, keine MMU

==> *Nicht* für "große" *Betriebssysteme*
==> Ev. nur eingeschränkt Hochsprachen-tauglich

Alles auf einem Chip:

- *RAM & Flash* (oft nur wenige KB) on Chip
- "Einfache" *I/O-Devices* on Chip
- Ev. gar kein externer Bus

Kosten « 5 Euro! ("für Haushaltsgeräte, ...")

Grobe Prozessor-Einteilung (1b)

Was sind "einfache" I/O-Devices?

- *Timer / Counter*, PWM
- Watchdog
- *Digitale I/O-Pins*, Schieberegister
- *A/D- & D/A-Wandler*
- I^2C, SMBus, SPI, CAN, LIN, RS232, ...
- Tasten-Matrix-Controller

Grobe Prozessor-Einteilung (1c)

Beispiele:

- Intel: **8051** (8 bit, seit 1980!)
- AVR: **ATmega** (8 bit)
- Microchip: **PIC**, PIC24 (16 bit)
- Siemens/Infineon: **XC16x** (16 bit)
- Hitachi/Renesas: **H8** (8/16/32 bit)
- TI: **MSP** (16 bit)
- Motorola/Freescale: **68HCxx** (8 bit)

Grobe Prozessor-Einteilung (2a)

"Große" Mikrokontroller

32 bit CPU (10 ... 300 MHz, < 1 W),
ev. FPU, *keine* "echte" MMU

==> *Nicht* für "*große*" Betriebssysteme (RT-OS schon)
==> Voll *Hochsprachen-tauglich*

Alles (ev. außer Speicher) auf einem Chip:

- RAM & Flash on Chip und/oder extern,
  ev. on-Chip-Flash-Controller
- "Einfache" und "mittlere" I/O-Devices on Chip
- Oft externer Bus

Grobe Prozessor-Einteilung (2b)

Was sind "mittlere" I/O-Devices?

- *USB*-Controller (USB 2, nicht USB 3)
- *SDIO*
- "Dummer" *Display*- bzw. LCD-Controller, Touch-Screen-Interface, ...
- Ev. *100 Mbit Ethernet*, ...

Grobe Prozessor-Einteilung (2c)

Beispiele:

- ARM: Familien **Cortex M** und **Cortex R**
- Renesas: **SuperH**
- Freescale: **68xxx / Coldfire**

Grobe Prozessor-Einteilung (3a)

???: ("System-on-a-chip" ?)

32/64 bit CPU (100 ... 3000 MHz, 1 - 30 W),
FPU, oft *Vektor-Einheit*, *volle MMU*, oft *Multi-Core*

==> Voll *Betriebssystem-tauglich* (Linux, ...)
==> Voll Hochsprachen-tauglich

Alles (außer Speicher) auf einem Chip:

- RAM & Flash nur extern, aber große Caches, on-Chip-Flash-Controller
- Auch "große" I/O-Devices on Chip

Grobe Prozessor-Einteilung (3b)

Was sind "große" I/O-Devices?

- *Grafik* mit HW-Beschleunigung, *Video*-Decoder
- *Kamera*, Video-Encoder
- *Audio*
- (Gigabit) *Ethernet*, Echtzeit-Ethernet, *USB 3*
- *WiFi*, Bluetooth, Handy-Modem
- *SATA*
- *PCIe*

Grobe Prozessor-Einteilung (3c)

Beispiele:

- *Beinahe-Monopolist:*
  Von vielen Herstellern: **ARM**
- Intel: **x84 Atom**, **x86 Quark**

  (Quark: In etwa 80486@400 MHz + viel I/O,
  << 1W Chip, < 0,1 W Core
  z.B. für "Wearable Computing")
- Immer mehr verdrängt:
  Von mehreren Herstellern: **MIPS** und **PowerPC**
- Renesas: Neuere **SuperH**

Grobe Prozessor-Einteilung (3d)

Sonderfall ARM:

ARM Cortex A (*"Application"*): *Groß & schnell*:
Handy usw.

ARM Cortex R (*"Realtime"*): *Mittel*:
Mittlere Echtzeit-***Steuerungen***, ...

ARM Cortex M (*"Micro"*): *Klein*:
Mikrokontroller, ...

Aus Softwaresicht *ähnliche Architektur*,
aus Hardwaresicht größtmögliche Variationen

Grobe Prozessor-Einteilung (4a)

(Mikro-) Prozessor

64 bit CPU (300 ... 5000 MHz, 5 - 200 W),
FPU, Vektor-Einheit, volle MMU, Multi-Core

==> Voll Betriebssystem- & Hochsprachen-tauglich

Reiner Prozessor-Chip:

- RAM nur extern, aber sehr große Caches
- Kein Flash, kein Flash-Controller
- Keine / wenig I/O-Devices on Chip,
  aber viele PCIe-Schnittstellen
- Ev. Prozessor-Prozessor-Kommunikation

Grobe Prozessor-Einteilung (4b)

Beispiele:

Intel & AMD: **x86**

IBM: **zSeries**, **Power**

Sun/Oracle/Fujitsu: **Sparc**

Grobe Prozessor-Einteilung (5)

"Spezial-Prozessoren"

... werden hier *nicht* betrachtet:

- *Grafik*-Prozessoren
- *Supercomputing*-Beschleuniger
- *DSP's* (Digitale Signal-Prozessoren)
- *"Softcores"* = Prozessor-Designs
  in Hardware-Beschreibungs-Sprachen
  für FPGA's

x86: Geschichte (1)

8086/80186 (1978):

8 * *16 bit Register*
20 Bit segmentierte Adressen ohne Schutz

"16 bit Real Mode" (DOS) bzw. später
"Virtual 8086 Mode" (emuliert im 32-bit-Mode)

80286 (1982):

8 * *16 bit Register*
24 Bit segmentierte Adressen mit Schutz

"16 bit Protected Mode"

x86: Geschichte (2)

80386, 80486, Pentium, ... (1985):

8 * *32 bit Register*
32 Bit Adressen, vollwertige *paged MMU*

"32 bit Protected Mode"

AMD Athlon / Opteron (2003):

(Intel hatte eine ganz andere 64-bit-Architektur "IA-64", zog erst 2005 unter dem Druck des Marktes nach)

16 * *64 bit Register*
64 bit Adressen (aus SW-Sicht)

"x86-64" (?)

x86: Geschichte (3)

Gleitkomma-Berechnungen:

Vor Pentium 3 (1999): **x87-Gleitkommaeinheit**

*8 * 80 Bit Gleitkomma-Register*,
als *Stack* organisiert!
(bis 80486 auf *separatem Gleitkomma-Koprozessor-Chip* 8087, ...)

Ab Pentium 3: **SSE, später AVX**

Völlig getrennt von x87:
Neue Register (kein Stack), ganz *andere Befehle*
Zugleich *Vektor- bzw. Multimedia-Einheit*
(für ganze Zahlen & Gleitkomma-Zahlen)
Anfangs *8 * 128 Bit*, heute *32 * 512 Bit* Register

x86: Geschichte (4)

Multimedia- bzw. Vektor- (Supercomputing-) Erweiterungen:

Ab Pentium 2 (1996): **MMX**: Nur int
(*überlagert x87-Einheit*: Nur abwechselnd nutzbar!)

AMD (1997): MMX-Erweiterung mit float: **3DNow!**
(wurde von Intel nie übernommen)

Ab Pentium 3 (1999): **SSE, später AVX** (2008)
Ganz neue Einheit, von MMX und x87 getrennt
int & float

x86: Architektur

CISC, *sehr komplexer* Befehlssatz:

- An die 1000 Befehle
- *Variable* Länge 1 … ~25 Bytes, *viele Befehlsformate*
- Historisch gewachsen,
  "unschön" und oft "unsymmetrisch"
- ***Meist 2 Operanden*** (a += b, *nicht* c = a + b !):
  - Erster Operand: *Register oder Speicher*
  - Zweiter Operand: *Register oder Konstante*
- *Komplexe* Adressierungsarten (u.a. mit Index)

x86: “Verschärfungen”

Viele völlig inkompatible ABI's:
https://en.wikipedia.org/wiki/X86_calling_conventions

- 16 / 32 / 64 bit
- Windows, Linux oder andere OS

Zwei komplett verschiedene Assembler-Syntaxen:
https://en.wikipedia.org/wiki/X86_assembly_language

- **Intel** bzw. **Nasm**
  (ähnlich, aber nicht gleich: Microsoft, Borland, ...)
- **AT&T** bzw. **Unix / Linux**
  (gcc/gas kann *beides*, default AT&T-Syntax)

x86: Assembler allgemein

Wir verwenden Intel- bzw. Nasm-Syntax,
nicht Linux-Syntax!

==> Bei zwei Operanden:
"Ziel links, Quelle rechts", "von rechts nach links"!

- **Syntax, Einrückung, Kommentare, Labels, ...:**
  Einfach von Beispiel-Programmen abschauen!
- **Befehle:**
  ... aus Beispielen, Intel-Doku & Hausverstand
- **Flags:** Siehe Folie im allgem. Teil
  Die Flags verwenden wir vorläufig *nicht direkt*,
  nur indirekt über *bedingte Sprünge*

x86: Assembler Konstanten

- ***Normale Zahl***: Dezimal
- **`0`**... oktal, **`0x`**... hex, **`0b`**... binär
- ***Zeichen***: z.B. **`'A`** oder **`'\n`** (kein schließendes **`'`** !):
  Wert = ASCII-Wert des Zeichens
- Ein ***Label*** ist auch eine Konstante:
  Wert = ***Adresse, wo das Label steht***
- ***Rechnungen*** *mit Konstanten* ähnlich wie in C
  (Achtung: **`/`** ist je nach Assembler
  das Kommentar-Zeichen, nicht die Division!)

x86: Assembler Speicher-Operanden

Notation für *Speicher-Zugriff*: ***Adresse*** in **[ ]**

Adresse = ***Register*** (enthält hoffentlich *Pointer*!)
+ K***onstante oder Label***
+ ***zweites Register***, ev. mal 2, 4 oder 8

Jeden der drei Teile kann man *weglassen*!
Beispiele: **`[rsi] [rsp+16] [myvar] [rsi+4*rax]`**

Wenn die *Größe des Speicher-Operanden*
nicht *durch den anderen Operanden* festgelegt wird:
Vor die **[ ]** schreiben, z.B. **`byte ptr [esi]`**
(oder **`word ptr`**, **`dword ptr`**, **`qword ptr`**)

x86: Assembler-Befehle

Immer:

- **`.intel_syntax noprefix`**
  Verwende *Intel-Assembler-Syntax*
  ohne **%** vor den Register-Namen
- **`.text`**
  Das Folgende gehört in den *Code-Bereich*
- **`.globl`** ***label***
  Macht ***label*** im Linker *für andere Files sichtbar*

Weitere später bei Bedarf...

x86: Register alter 32-Bit-Modus

- **ax**, **bx**, **cx**, **dx**, **si**, **di**, **bp**, **sp**
  Die Original-8086 ***16-bit-Register***
- **eax**, **ebx**, ...
  Dieselben Register mit ***32 bit***
- **al**, **bl**, **cl**, **dl** *"Low Byte"*
  Das *hinterste Byte* (8 bit) von **ax**, **bx**, **cx**, **dx**
- **ah**, **bh**, **ch**, **dh** *"High Byte"*
  Das *zweithinterste* Byte von **ax**, **bx**, **cx**, **dx**
- Es gibt *keine* 8-bit-Variante von **si**, **di**, **bp**, **sp** !

x86: Register 64-Bit-Modus

- **`rax`**, **`rbx`**, ...
  Die acht *alten* Register mit ***64 bit***
- **`r8`**, **`r9`**, **`r10`**, ..., **`r15`**
  Acht *zusätzliche* ***64-Bit-Register***
- **`r8b`**, **`r9b`**, ... *"byte"*
  `r8w`, **`r9w`**, ... *"word"*
  `r8d`, **`r9d`**, ... *"doubleword"*
  Dieselben Register mit ***8, 16 und 32 Bit***
- **`sil`**, **`dil`**, **`bpl`**, **`spl`** (statt **`ah`**, **`bh`**, **`ch`**, **`dh`**)
  "Low Byte" von **`si`**, **`di`**, **`bp`**, **`sp`** (kaum verwendet!)

x86_64 Win ABI: Daten

Alle Daten liegen auf ihrem

"natürlichen Alignment"

- 1-Byte-Daten (**`char`**) auf *jeder* Adresse
- 2-Byte-Daten (**`short`**) auf *geraden* Adressen
- 4-Byte-Daten (**`int`**, **`float`**)
  auf *durch 4 teilbaren* Adressen
- 8-Byte-Daten (**`long`**, **`double`**, ***Pointer***)
  auf *durch 8 teilbaren* Adressen

(ev. Ausnahme: Daten in "**`packed`**" **`struct`**)

x86_64 Win ABI: Register (1)

- **`rsp`**: *Fixe* Verwendung als *Stack Pointer*

Außer für Kommazahlen (und **`struct`**-Parameter / -Return):

- **`rax`**: ***Return-Wert***,
  sonst *"frei verwendbar"*
- **`rcx`**, **`rdx`**, **`r8`**, **`r9`**:
  Beim Aufruf ***erster bis vierter Parameter***,
  sonst *"frei verwendbar"*
- **`r10`**, **`r11`**: Auch *"frei verwendbar"*
- **`rbx`**, **`rsi`**, **`rdi`**, **`rbp`**, **`r12`** … **`r15`**: *"Callee saves"*

x86_64 Win ABI: Register (2)

"frei verwendbar"

- Die ***aufgerufene Funktion*** darf diese Register ***nach Belieben ändern / verwenden***,
  ohne sie vorher zu sichern
- Die ***aufrufende Funktion*** darf sich
  nicht darauf verlassen, dass diese Register
  nach einem Aufruf noch *denselben Wert enthalten*

  Wenn sie das will,
  muss sie die Register selbst *vor dem Aufruf sichern*
  und *nach dem Aufruf wiederherstellen*

x86_64 Win ABI: Register (3)

"Callee saves"

- Die ***aufrufende Funktion*** darf sich darauf verlassen, dass diese Register *nach* einem Aufruf *noch denselben Wert enthalten*
- Die ***aufgerufene Funktion*** darf diese Register *nicht verwenden bzw. verändern*

  Wenn sie sie *doch verwenden* will:

  - Am *Beginn der Funktion* ***sichern*** (mit **push**)
  - Alten Wert *vor Return* wieder ***laden*** (mit **pop**)

x86_64 Win ABI: Stack (1)

Auf dem Stack wird *vom Aufrufer* ***vor*** *jedem Aufruf*

Platz für 4 Parameter zu je 8 Byte reserviert
(*auch* wenn die Funktion *weniger* als 4 Parameter hat!)

Dieser Platz liegt *direkt über der Return-Adresse*
(*unterster* Platz entspricht dem *ersten* Parameter)

- **Früher:** Nur als *Platz zum Sichern der Parameter* 1 bis 4 aus den Registern gedacht
- **Heute:** Von der *aufgerufenen Funktion* ***beliebig verwendbar!*** (lokale Variablen usw.)

x86_64 Win ABI: Stack (2)

- Bei *mehr* als 4 Parametern:

  Übergabe *ab dem 5. Parameter am Stack*,
  über dem reservierten Platz für Parameter 1 bis 4 .

- Wenn die Funktion
  selbst *keine weitere Funktion aufruft*:
  Alignment des Stack-Pointers ist *egal*
  (aber *mindestens 4*, d.h. kein push/pop von 1 und 2 Byte Werten!)

- Bei Funktionen, die Calls enthalten:
  Stack-Alignment muss immer 16 sein
  (d.h. *bei Aufruf* einer Funktion, unmittelbar vor dem **`call`**:
  `rsp` muss immer eine *durch 16 teilbare Adresse* enthalten)

x86_64 Win ABI: Stack (3)

- Zugriff auf Speicher *unterhalb* des aktuellen **`rsp`** ist ***strikt verboten***
  (keine *"Red Zone"* wie in Linux)
- Ein ***"Frame Pointer"*** oder ***"Base Pointer"***
  auf das *obere Ende* des Stackbereiches
  der eigenen Funktion ist *freiwillig*,
  nur für bestimmte Konstrukte Pflicht
- Weitere Vorgaben für *Debugging*, *Exceptions*, ...

ATmega: Was ist das? (1)

Große, sehr verbreitete *Familie* ganz kleiner

8-bit-Mikrocontroller

In unserem Fall **ATmega 8515**:

- **2,7 … 5,5 V**, max. **rund 60 mW**, Standby < 10 µW
- **0 … 8 bzw. 16 MHz** (hält seinen Zustand *taktlos*), interner Taktgenerator (braucht keinen Quarz)
- Erweiterter Temperaturbereich **-40 … +85 (+125) C**
- Rund **1 €**

("Doku" bezieht sich im Folgenden auf Atmel 2512K–AVR–01/10)

ATmega: Was ist das? (2)

Von außen:
Zahlreiche Gehäuse, bei uns *40 Pin dual-in-line*

- **GND**, **VCC**
- **XTAL1** und **XTAL2**
- **RESET**
- 35 frei programmierbare I/O-Pins
  (**A0** ... **A7**, **B0** ... **B7**, **C0** ... **C7**, **D0** ... **D7**, **E0** ... **E2**)
  wahlweise mit spezieller *Doppel-Funktion*
  (z.B. *externer Datenbus*, UART, analog Input, ext. Interrupt, ...)
  (siehe Doku Figure 1 sowie ab Seite 66)

ATmega: Was ist das? (3)

Von innen:

- Ein ***8-bit-Core***
- 512 Bytes ***SRAM***
- 512 Bytes ***EEPROM*** (für *Daten*)
- 8192 Bytes ***Flash*** (für *Programmcode*, Konstanten)
- Max. 64 KB ***ext. RAM bzw. I/O-Chips*** anschließbar
- Ein ***8 bit Zähler***, ein ***16 bit Zähler***, ***Watchdog***
- ***Analog-Vergleicher***, ***SPI***, ***USART***

ATmega: Core

- ***RISC-Core:***
  - Fast alle *Befehle 16 bit lang*, manche 32 bit
  - Rechenbefehle nur *reg,reg* oder *reg,imm* , nicht *reg,mem*
- ***Harvard-Architektur:***
  Code-Speicher (Flash) + Code-Bus *strikt getrennt* von ***Daten***-Speicher (RAM) + Daten-Bus
- 2-stufige ***Befehls-Pipeline***

==> *Die meisten Befehle laufen mit*
1 Befehl pro Takt

ATmega: Code & Flash

Der **PC** (*"Program Counter"*) ist *12 Bits*,
der Code-Speicher (Flash) ist
wort-adressiert, *nicht byte-adressiert*
(d.h. 1 Adresse = 1 Befehl = 2 Bytes)

==> max. 2^{12} = 4096 Befehle, mal 2 Bytes = 8 KB Code

Zweigeteilt:

- **Unten**: *Anwendungs-Bereich*, frei programmierbar
  (kann neben Befehlen auch Konstanten enthalten)
- **Oben**: *Boot-Bereich* (fix belegt mit Code
  zum neu *Flashen* des Anwendungs-Bereiches)

ATmega: Daten

Daten-***Adressen*** sind ***16 bit*** breit:

- **`0x0`** … **`0x1F`**: 32 Bytes für ***CPU-Register*** **`r0`** … **`r31`**
- **`0x20`** … **`0x5F`**: 64 Bytes für 64 ***I/O-Register***
- **`0x60`** … **`0x25F`**: 512 Bytes ***internes SRAM***
- **`0x260`** … **`0xffff`**: Falls vorhanden:
  max. 64 KB ***ext. RAM*** und ***ext. I/O***

ATmega: EEPROM

Das *EEPROM* liegt

weder im Code-
noch im Daten-Adressraum,

sondern wird durch Lesen und Schreiben
von ***I/O-Registern*** (**EEARL**, **EEARH**, **EEDR**, **EECR**)
byteweise angesprochen.

ATmega: Register (1)

Die CPU hat

32 allgemeine 8-Bit-Register **`r0`** … **`r31`**

(zur Programm-Lesbarkeit: Mit **`.def`** Namen zuordnen!)

- Der ***12 bit*** **PC** (*Program Counter*) ist separat und nicht direkt ansprechbar
- Der ***16 bit*** **SP** (*Stack Pointer*) ist separat, aber als *I/O-Register* **`0x5D`**+**`0x5E`** ansprechbar
- Die ***8 bit*** *Flags* **SREG** (*Status-Register*) sind separat, aber als *I/O-Register* **`0x5F`** ansprechbar

ATmega: Register (2)

- Die ***Register-Paare*** **r26**+**r27**, **r28**+**r29**, **r30**+**r31** heißen **X**, **Y** und **Z**
  und können für *16-bit-Daten-Adressen* (***Pointer***) verwendet werden (*low Byte im ersten Register*)
- **r24**+**r25** ist *nicht als Adress-Register* verwendbar, aber kann auch *16 bit Addition/Subtraktion*
  (z.B. für *"lange" Zähler*)

Achtung: Einige Befehle
(z.B. alle Befehle mit *8-Bit-Konstanten*)
können nur **r16** ... **r31** ansprechen, nicht **r0** ... **r15**

ATmega: Flags

- **Z** (*"Zero"*), **C** (*"Carry"*), **H** (*"Half Carry"*) wie bei x86
- x86 **S** (*"Sign"*) heißt auf ATmega **N** (*"Negative"*),
  x86 **O** (*"signed Overflow"*) heißt auf ATmega **V**
- Neues Bit **S** (*"Signed compare"*) ist **N** *xor* **V**
- x86 **P** (*"Parity"*) gibt es auf ATmega *nicht*
- **T** (*"Transfer"*): Von manchen Befehlen (**BLD**, **BST**)
  explizit gelesenes / geschriebenes ***"1-bit-Register"***
- **I**: Globales *Interrupt-Sperrbit*
  (1 … freigegeben, 0 … gesperrt)

ATmega: I/O-Register

Siehe Übersicht Doku Seite 239

Enthalten auch **SPL** & **SPH** (**0x5D**, **0x5E**)
sowie die *Flags* **SREG** (**0x5F**)

Achtung:
Bei Zugriff über *Speicher-Befehle*: Adr. **0x20** … **0x5F**
Bei Zugriff über *I/O-Befehle* (**IN**, **OUT**, **CBI**, …):
Port-Nr. **0x0** … **0x3F**

Achtung:
Die I/O-Bit-Befehle **CBI**, **SBI**, **SBIC**, **SBIS** kennen
nur die Ports **0x0** … **0x1F**, *nicht* **0x20** … **0x3F**

ATmega: Daten-Darstellung

Alle Bereiche
(Flash, RAM, 16-bit-Register, 16-bit-I/O-Ports)
sind

little endian!

==> das *niederwertige Byte*
kommt *zuerst* im Speicher

(d.h. hat die *kleinere Adresse bzw. Register-Nummer*)

Der Stack wächst nach unten!

ATmega: Befehle (1)

Siehe Doku Seite 241 oder Web-Tutorial...

ATmega ist eine *RISC-Architektur*

==> *Rechen-Befehle* arbeiten
nur auf Register-Operanden (und ev. Konstanten),
nicht auf Operanden im Speicher

ATmega-Rechenbefehle
haben *max. zwei Opeanden* (d.h. **a += b**)
nicht drei
(d.h. für **a = b + c** braucht man *zwei Befehle*)

ATmega: Befehle (2)

Arithmetisch-logische Befehle: Die üblichen ...

- Es gibt ein *16 bit plus und minus* mit Konstanten (**ADIW** und **SBIW**), sonst nur 8 bit Rechnungen
- Es gibt ein **MUL**, aber *keine Division* (wenn Division nötig: Ausprogrammieren...)

  MUL hat ein fixes, *doppelt langes* Ziel: **r0**+**r1**
- Es gibt Befehle zum *Setzen, Löschen und Prüfen einzelner Bits* in Registern und I/O-Registern: **SBR**, **CBR**, **SBI**, **CBI**, **BLD**, **BST**, **SBRC**, **SBRS**, **SBIC**, **SBIS**

ATmega: Befehle (3)

Speicher-Befehle:

- **MOV** zwischen 2 Registern
  (auch **MOVW**: *16 bit*, d.h. 2 Register-Paare)
- **LD** aus dem Speicher, **ST** in den Speicher
- **PUSH** und **POP**
- **IN** und **OUT** (Register von/nach *I/O-Register*)
- **LPM** (*"Load Program Memory"*):
  Holt 1 Byte aus dem *Flash-Speicher*
  (**Achtung:** Label * 2 !)

ATmega: Befehle (4)

Sprung-Befehle:

- Unbedingter Sprung / Call, bedingter Sprung
- ***"Überspring"-Befehle***
  SBIC, **SBIS**, **SBRC**, **SBRS**, **CPSE**:

  - *ein Bit* in einem *I/O-Port* prüfen,
  - *ein Bit* in einem *Register* prüfen,
  - oder *zwei Register vergleichen*

  und abhängig davon den ***nächsten Befehl***
  (meist ein **RJMP** oder ein **RCALL**)
  überspringen oder ausführen

ATmega: Adressierungsarten

5 Möglichkeiten für Speicher-Zugriffe:

- ***Absolute*** Speicher-Adresse
- ***Pointer-Register*** **X**, **Y** oder **Z** als Adresse
- dasselbe mit ***post-Inkrement***
- dasselbe mit ***pre-Dekrement***
- Register **Y** oder **Z** ***plus 0...63*** als Adresse

ATmega: Interrupts (1)

Es gibt ***13 interne Interrupts*** (Timer, UART, ...)
und ***3 externe Interrupt-Leitungen***
(einstellbar: Low-aktiv oder
steigende / fallende / beide Flanken)

Kontrolle durch das ***I-Bit*** in den Flags (**CLI** / **STI**)
und durch ***Bits pro Interrupt*** in den I/O-Reg's

Es gibt Interrupts *"mit Gedächtnis"*
(*bleiben gespeichert* bis sie ausgeführt oder gelöscht werden,
d.h. werden *nachgeholt* wenn sie blockiert sind)
und *"ohne Gedächtnis"*
(*gehen verloren* wenn sie beim Eintreffen blockiert sind)

ATmega: Interrupts (2)

Bei Eintreffen eines Interrupts:

- Unterbrechung sobald aktueller Befehl fertig ist
- Sichert **PC** auf Stack, **I**-Bit wird **0** (Interrupts sperren)
- Fortsetzung gemäß entsprechendem Eintrag
  der Interrupt-Tabelle

==> Sprung zum Interrupt-Handler

Am Ende des Interrupt-Handlers: Befehl **RETI**
Holt **PC** vom Stack, setzt **I**-Bit wieder auf **1**

==> Unterbrochenes Programm macht weiter

ATmega: Interrupts (3)

Achtung:

*Außer dem **PC** wird **nichts***
automatisch gesichert,
auch nicht die Flags!

==> Man muss die Flags (& ev. weitere Register)
im Interrupt-Handler sichern & zurück-kopieren!

Typischerweise werden die Flags nach **`r15`** gesichert
(**`IN R15,SREG`** und **`OUT SREG,R15`**)

==> **`r15`** *freihalten*, im normalen Code *nicht nutzen!*

ATmega: Interrupts (4)

Die ***Interrupt-Tabelle***

- … liegt *am Beginn des Flash* (ab Adresse **`0`**)
- … Eintrag 0: ***Reset-Adresse***
  = Sprungbefehl (**`rjmp`**) zum *Programmstart*
- … Eintrag 1 … 16: ***Adresse des Interrupt-Handlers***
  für jede der 16 möglichen Interrupt-Quellen
  (auch als **`rjmp`**-Befehl)
  (siehe Table 22 auf Seite 54)

Eintrags-Nummer entspricht *Priorität, niedere zuerst*
(wenn mehrere Interrupts *zugleich* anstehen)

ATmega: Interrupts (5)

Beim Programmstart:

- Zuerst **SP** initialisieren
  (typischerweise auf **RAMEND**, kommt gleich...)
- Dann erst Interrupts freigeben (**SEI**)

Beispiel zum ***Einrichten der Interrupt-Tabelle***
im Assembler-Programm:

```
.CSEG                ; gehört ins Flash ...
.ORG 0000            ; ... ab Adresse 0
   RJMP Start        ; Reset-Vektor
   RJMP IHext0       ; Adresse des Handlers für externen INT0
   ...
```

ATmega: Assembler Tipps (1)

Das ***Kommentar-Zeichen*** ist der **`;`** ,
nicht das **#** (*C-Kommentare* funktionieren auch)

Folgende ***Pseudo-Befehle*** sind häufig:

- **`.equ`** setzt einen *Namen gleich einer Zahl*:
  `.equ SPEED = 9600`
- **`.def`** definiert einen *Namen für ein Register*:
  `.def zaehler = r24`
- **`.include`** inkludiert *Files*
  (vor allem *vordefinierte Files*
  mit Definitionen von Konstanten für die I/O-Register)

ATmega: Assembler Tipps (2)

- **`.cseg`** und **`.dseg`**: *Code-* und *Datensegment*
- **`.org`**: Legt die aktuelle Adresse fest
  `.org URXCaddr` ; UART interrupt table entry
- **`.db`** und **`.dw`**: *"define bytes"* und *"define words"*:
  Reserviert & initialisiert *Speicher im Code-Bereich*
  ```
  .db 0, '\n', 0xff
  ziffern: .db "0123456789"
  ```
- **`.byte`**: Reserviert die angegebene *Anzahl uninitialisierter Bytes* im *Daten-Bereich*:
  ```
  array: .byte 64
  ```

ATmega: Assembler Tipps (3)

RAMEND: Adresse des RAM-Endes (also **0x25f**),
aus einem Prozessor-abhängigen Include-File.

LOW(*x*) und **HIGH(*x*)**:
Unteres und oberes Byte eines 16-bit-Wertes

==> Am Programm-Anfang zur *Initialisierung* des **SP**:

```
LDI r16, HIGH(RAMEND)
OUT SPH, r16
LDI r16, LOW(RAMEND)
OUT SPL, r16
```

ATmega: ABI

Es gibt zwar ein ABI (weil es einen C-Compiler gibt), aber da wir

- weder kombiniert mit C-Programmen arbeiten
- noch Libraries verwenden
- noch ein Betriebssystem haben

ist es für uns

nicht relevant!

==> Parameter, Register usw. nach Belieben!