Assembler-Befehle

David Vajda

12. Februar 2026

Der Computer setzt sich zusammen aus

1. CPU (Central Processing Unit):
2. Arbeitsspeicher (RAM)
3. Input: Eingabe: Peripherie
4. Output: Ausgabe: Peripherie

Das ist eine extrem verkürzte zusammenfassung. so etwa sieht es bei John von Neumman aus. Die Namen der Entwicklung der Compute r

1. John von Neumann
2. Charles Babagge
3. John V. Atanasov
4. Harward ... (Architektur)
5. Konrad Zuse
6. Touring

Verschiedene Architekturen, beispiele moderne Prozessoren:

1. Z80, Zilog
2. x86, x85, 80286, 80386, 80486, Pentium I, II, III, IV, ...
3. so wie weitere Intel 8039
4. atmega8
5. MIPS32
6. PIC
7. ARM Cortex etc.
8. ...

Verschiedene Architekturen:

1. von Neumann Architektur: RAM enthaelt Programm und Daten

2. Harward geht anders: z.B. Atmega8 ist wie Harward, möglicherweise zu 100 prozent, es gibt ueberschneidungen:
   (a) Arbeitsspeicher (da sind Daten drin) das ist fluechtiger Speicher, hier kommen Tabellen rein (SRAM)
   (b) einen Programmspeicher, der programmspeicher ist Flash-EPROM
   (c) Festen langfristigen fuer Daten: EEPROM

Dann muss man unterscheiden

1. Code Segment - hier gehoert das Programm rein, da sind aber auch Daten drin, also Zeichenketten usw.
2. Daten Segment, hier kommen die programmierten Tabellen rein
3. Extra Segment: Heap, den Heap, vorsicht: an der stelle herrscht bei mir gerade unsicherheit:
   - Heap: heisst: atmega8: feste daten: EEPROM, das sind feste daten - das ist harward architektur, die zwischen Daten und Code trennt
   - Intel x86 ist eine mischung aus harward und von neumann, erstens: und zweitens: weil sie haben einen RAM, ein Code Segment, ein Datensegment
   - x86 ist mischung, beides im RAM, aber Code und Daten in einem, aber getrennt. Segmentregister: Heap sind normalerweise: feste daten.
4. Stack
5. I/O: Ein und Ausgabe: z.B. Memory Mapped I/O Ports

Jetzt darauf kommt es an, die Befehle und die Register, folgendes:

1. es ist stark davon abhängig wie die Architektur ist (ISA - Instruction Set Architecture) nicht die Microarchitektur, halbweil: von der Speicherorginasation hängen die Befehle hab

man hat generell, folgende klassen:

1. Arithmetisch Logische Befehle
2. Transportbefehle
3. Sprungbefehle und bedingte Verzweigungen

Was es daneben gibt ist architektur abhängig und lassen wir jetzt aussen vor und zwar folgendes:

1. Schiebe und Rotationsbefehle könnte in den Bereich Arithmetik passen wird aber getrennt gesehen
2. Programmsteuerbefehle
3. Systemsteuerbefehle.
4. viele weitere.

Jetzt nur zu den drei genannten: wir brauchen daneben:

1. Register
2. Flags - Statusflags ...
3. Die Breite (Byte, Wort, Doppelwort)
4. so wie die grundrechenarten, ...

Die Register (General Purpose Register) allzweckregister - beim x86 sind

1. `ax, bx, dx, cx`:
   (a) `ax`: akkumulator
   (b) `bx`: basisregister
   (c) `dx`: datenregister
   (d) `cx`: zaehlregister
   (e) `bp`: basepointer (basis pointer)
   (f) `sp`: stackpointer
   (g) `si`: source index (quellindex)
   (h) `di`: destination index (zielindex

bei anderen prozessoren, haben sie generell meist:

`r31, r30, ..., r1, r0`

und folgendes: bei dem Atmega8 sind die letzten 6 register, von jeweils zwei zu drei zusammengesetzt:

1. `r31:r30`: `Z`
2. `r29:r28`: `Y`
3. `r27:r26`: `X`

als wichtigsten schritt inzwischen sie haben es erraten, brauchen wir die breite von registern:

1. Das Byte (kleinstes) 8 Bit
2. Das Wort (meistens) oder das halbwort (16 Bit)
3. Das Doppelwort 32 Bit (manchmal das Wort)
4. Das Quadword (64 Bit) manchmal das Doppelwort

das müssen wir uns einfach merken. neben den registern im Prozessor gibt es den Arbeitsspeicher.

1. Prozessor (CPU mit Registern)
2. der Arbeitsspeicher

der Arbeitsspeicher, besteht ebenso aus zellen, die werden angesprochen wie im prozessor die register, über binäre addressen. im prozessor ist uns das als programmierer, ISA - Instruction Set Architecture nicht bewusst. Wir benuetzen die namen der Register `ax, bx, dx, cx, ...` oder `r31 ... r0`, so, daneben: müssen wir wissen, der arbeitsspeicher besteht aus zellen, die ebenso wie die register. allerdings sind es viel mehr. z.B. `64k 64k` heisst `2^16` oder sie haben `4M` oder `4G`.

im Arbeitsspeicher speichern wir Variablen, das ist das wichtigste!!! Also wichtig

1. Arbeitsspeicher
2. Variable

Bei einer Variable müssen wir uns merken:

1. Sie hat einen wert (eine speicherstelle im RAM hat immer einen Wert, und sei es 0)
2. Eine Addresse
3. Vom Computer selber her nicht unbedingt einen Typ (float, Integer, Character,...)
4. Sie hat einen Namen. auch im Computer, selber, entspricht die Addresse dem namen, weil der Name wird durch addresse ersetzt.

das wichtigste bei einer variablen ist:

1. name
2. wert
3. typ
4. addresse

was den Typ betrifft: entspricht das der Breite im Computer, der unterliegt im Computer oft keiner deutung.
der Name wird zur addrese Das hier ist allerdings das wichtigste: unbedingt merken fuer alle hochsprachen

1. name
2. wert
3. typ
4. addresse

das letzte wichtige ist: wir fangen immer bei 0 zu addressieren. also gehen immer von 0 an los
nebenbei koennen nicht nur zellen im RAM beanspruchen, sondern bereiche. Die Register wiederum sind letzten endes auch nur namen

Jetzt noch mal hingucken:

1. name
2. wert
3. typ
4. addresse

jetzt muessen wir variablen im arbeitsspeicher im maschinenprogramm deklarieren, dies ueber namen, label: label gibt es:

1. im programm selber (Sprungziel)
2. Variablen

die variablen sind selber:

1. ein Byte oder ein vielfaches eines Bytes: Deklaration:

   ```
   label1: .db 0x00, 0x00, 0x00
   ```

2. ein Word:

   ```
   label2: .db 0x0000, 0x0000
   ```

jetzt haben die einen namen. und wir haben einen bereich im arbeitsspeicher ueber addressen. dessen gibt es addressierungsarten, hier folgende einfach auswendig lernen, intel:

1. Direktwert
2. Register-Register
3. Direkte
4. Indirekte
5. Indizierte

Folgendes:

1. direktwert ist eigentlich keine addressierung, so wie register-register auch nicht, das bedeutet: der zugriff ist innerhalb mehr oder weniger des prozessors
2. nur daten im RAM gelten als Addressierungsart.
3. Was direkt wert heisst, ist normalerweise konstante, die steht zwar im RAM, im Code, trotzdem ist das so eher keine addressierung
   direktwert entspricht dem word immidiate und konstante
4. wichtigstes kriterium für intel: wir haben einen MOV-Befehl

sonst haben wir (Transportbefehle)

1. `mov` ist Register-Register (normalerweise)
2. Laden, Load - von RAM in Prozessor

3. Speichern, Store (von Prozessor nach RAM)

und bei intel ist alles MOV, aber die addressierungsarten bleiben:

1. Direktwert

```
mov ax, 0x0f
```

2. Register-Register

```
mov ax, bx
mov cx, dx
mov bx, ax
```

3. Direkte

```
mov di, label1
mov si, label2
mov dx, label3
```

4. Indirekte

```
mov si, label1
;; ==>>>
mov ax, [si]

mov bx, [si]
mov dx, [di]

; das register in eckigen klammern ist die addresse steht im register

;; indirekt gibt es auch anders herum:

mov [bx], dx

;; nur das ist bei intel mov, und sonst ist das store und das erste load.
```

5. Indizierte

```
mov cx, [bs+bp+4]
;; kann sein das geht an der stelle nicht.
```

Bei Atmega8/MIPS 32, gibt es wiederum

1. immidiate, konstante
2. register-register (mov)
3. direkt: lds
4. indirekt: das ist normal bei atmega8, das ist ld

5. indirekt mit autoinkrement / dekrement, und das inkrement ist ein PostInkrement und das Dekrement ist ein Predekrement
6. Indiziert: mit verschiewert, wie bei Intel, da gehen aber glaube ich zwei Register

muessen wir unterscheiden:

1. mov: register register
2. `ldi` das heisst: lade direktwert - bei intel ist immidiate

   ```
   ldi r16, 0x00
   ```

3. laden kann ich in einen direktwert nichts,
4. lds - ohne Autoinkrement und Autodekrement, das auto ist bei Predekrement und Postinkrement automatisch enthalten
5. ldd - ist mit verschiebung, indiziert heisst das
6. lds: das ist das direkte: das entspricht, der konstanten addresse. das lds
7. daneben: haben sie beim Atmega8 lpm und spm, das entspricht jetzt wiederum dem programmspeicher

und beim mips32, haben sie auch:

1. Register-Register, MOV
2. Direktwert (bei intel) immidiate, konstante, ldi
3. Direkt
4. Indirekt, autoinkrement und dekrement
5. indiziert mit verschiebewert, bias, vor und zurueck

so das sind die addressierungarten. daneben haben sie arithmetischen befehle

```
Summe = Summand1 + Summand2
Produkt = Multiplikand * Multiplikator
Quotient = Dividend / Divisor
Subtrahend = Minuend - Subtrahend
Quotient = Dividend / Divisor + Rest

Grundrechenarten:

Addition
Multiplikation
Division
Subtraktion

Dann die Logikoperationen

AND
```

```
OR
XOR
NOT

wichtige rolle spielt immer

Carry
Borrow
```

damit ergeben sich die befehle:

- ADD
- SUB
- MUL
- DIV
- AND
- OR
- XOR
- NOT

- ADC - Add with Carry

das drueckt sich so aus

```
add ax, bx
add dx, cx
...

add ax, bx
sub ax, bx

bei multiplikation

kommt der eine teil in AX rein und das Ergebnis steht in

DX:AX

MSB - Most Significant Bit (Byte)
LSB - Least Significant Bit

100010

die letzte 0 ist das LSB und die obere das MSB

bei der DIVISON
```

```
DIV BX

das bedeutet:

DX:AX / BX

und
der Rest in DX und das Quotient in AX

also

ADD AX, BX
SUB AX, BX
DIV BX
```

die status und steuerflags sind:

1. Carry Flag
2. Auxiiliare Carry Flag
3. Sign Flag
4. Overflow Flag
5. Parity Flag

gut, jetzt die sprungbefehle aus sicht von intel

```
Erst CMP - vergleichen, dann springen, nicht 100 pro richtig, verzweigungen, bedingung hei

JMP - das heisst: bedingungslos, ohne CMP, genauso aehnlich wie CALL und RET, benutzen abe

JE - Jump if Equal / JNE - Jump if Not Equal
JG - Jump if Greater / JNG - Jump if Not Greater
JA - Jump if Above / JNA - Jump if Not Above
JGE - Jump if Greater / JNGE - Jump if not Greater
JAE - Jumpf if Above / JNAE Jump if Not Above
JL - Jump if Less / JNL Jump if Not less
JB - Jump if Below / JNB - Jump if not below
JLE - Jump if Less or Equal / JNLE ...
JBE ... / JNBE
JZ - Jump if Zero (JNZ)
JC - Jump if Carry (JNC)
```