Rechnerarchitektur: 21.10.2010

Adressierungsarten (1/8)

Registeradressierung

- MMIX: `ADD $0, $1, $2`
- sonstige: `ADD R1,R2`
- wirkt auf Register, das explizit im Befehlswort angegeben ist

Adressierungsarten (2/8)

Unmittelbare Adressierung

- MMIX: `ADD $1,$1,10`
- sonstige: `ADD R1,10`
- Direktoperand im Befehlswort (hier: 10)

Adressierungsarten (3/8)

Programmzähler-relative Adressierung

- MMIX: `JMP @+12`
- sonstige: `LD R1, [PC,offset]`
- Effektive Adresse EA = PC + Offset

Adressierungsarten (4/8)

Absolute Speicheradressierung

- MMIX: –
- sonstige: `MOV Reg, Mem`
- Effektive Speicheradresse ist Teil des Befehlswortes

Adressierungsarten (5/8)

Register-indirekte Adressierung

- MMIX: – (ähnlich: `LDO $0,$255,0`)
- sonstige: `MOV Reg1, [Reg2]`
- Effektive Adresse befindet sich in einem Register

Adressierungsarten (6/8)

Indiziert speicher-relative Adresse

- MMIX: –
- Sonstige: `MOVE Reg1, [Reg2+Mem]`
- EA ist Summe aus Registerwert und Speicherwert

Adressierungsarten (7/8)

Indiziert register-relative Adressierung

- MMIX: `LDO $0,$255,$1`
- sonstige: `LOAD R1, [R2,R3]`
- Summe zweier Registerinhalte liefert die EA

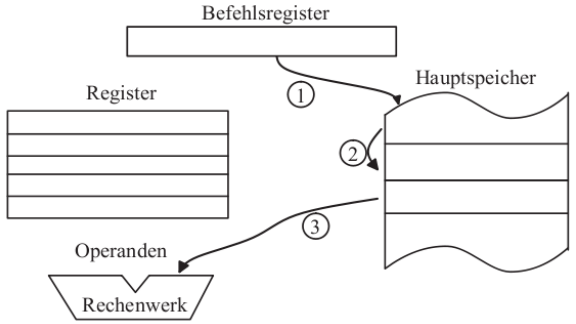
Adressierungsarten (8/8)

indiziert register-relative Adressierung mit Index

- MMIX: `LDO $0,$255,8`
- sonstige: `LOAD R1, [R2, Offset]`
- EA = Registerinhalt + Konstante

Zweistufige Speicheradressierung (1/2)

- Berechnung der effektiven Adresse erfodert bereits einen Speicherzugriff
- als Prozessor-Adressierungsart heute unüblich



Zweistufige Speicheradressierung (2/2)

- kommt in Programmierpraxis aber oft vor:

```
01          LOC   Data_Segment
02  Buffer  BYTE  0
03          LOC   Buffer+80          Puffer anlegen
04  * Argumentbereich:
05  Arg     OCTA  Buffer             Adresse des Puffers
06          OCTA  80                 Puffergröße
07
08          LOC   #100
09  Main    LDA   $255,Arg           Adresse des \
                                     Argumentbereichs
10          TRAP  0,Fgets,StdIn      Einlesen
11          TRAP  0,Halt,0
```


Befehlsarten

Themen, die in den ersten Semestern zu kurz gekommen sind

- Gleitkommabefehle
- Bedingte Befehle
- SIMD-Befehle
- Nicht-unterbrechbare Befehle
- spezielle Aspekte der Speicherzugriffsbefehle

Gleitkommazahlen (1/5)

- Einführung: „Kommazahlen“ – vom Dezimal- zum Dualsystem
- Dezimal: 2 9 , 1 4 3

Dual: 1 1 0 1 , 1 0 0 1 $_b$ (= 13,5625$_d$)

Gleitkommazahlen (2/5)

- Eine Zahl Z wird durch eine **Mantisse** f und einen **Exponenten** E dargestellt. Es gilt:

  $Z = \pm |f| \cdot 2^E$

- Für die Speicherung normalisierte Darstellung mit $1 \leq f < 2$. Die dabei stets auftretende Eins vor dem Komma muss nicht mit gespeichert werden.

  (vgl. normalisierte Darstellung im Dezimal-system: $-753{,}481 = -7{,}53481 \cdot 10^2$)

Gleitkommazahlen (3/5)

- Gleitkommazahlen mit 64 Bit Breite (IEEE 754)
  - Ein Bit v für das Vorzeichen (gesetztes Bit bedeutet negatives Vorzeichen)
  - Elf Bit für die Charakteristik e des Exponenten. Zum Exponent E addiert man den **Exzess** $q = 2^{11-1} - 1$: $e = E + q = E + 1023$ (e ist vorzeichenlos, E nicht)
  - Die restlichen 52 Bit für die Nachkommastellen des Betrags des gebrochenen Anteils f

Gleitkommazahlen (4/5)

Beispiel: 1,0 (auch binär: 1,0)

- $1{,}0 = 1{,}0 \cdot 2^0$: $f = 1$, $E = 0$
- Vorzeichen v: 0 (positiv; 1 Bit)
- Exponent E: 0; $e = E + 1023 = 1023 =$ 011 1111 1111 (11 Bit)
- Mantisse f: 000...0 (52 Bit, führende 1 nicht gespeichert – nur die Nachkommastellen)
- Ergebnis: 0011 1111 1111 0000 0... = #3FF0 0000 0000 0000

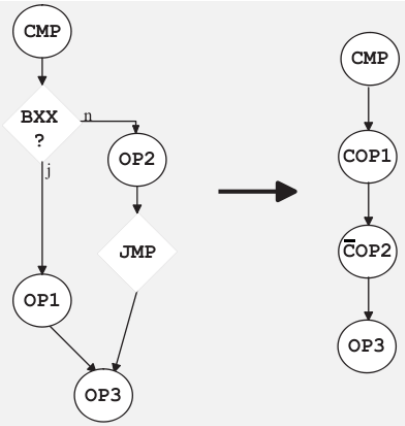
Gleitkommazahlen (5/5)

Sonderfälle

- $e = f = 0$: steht für 0; es gibt also eine „positive“ und eine „negative Null“
- $e = 2047$, $f = 0$: unendlich (∞) (∞ oder $-\infty$, je nach v)
- $e = 2047$, $f > 0$: keine Zahl („not a number“, NaN)
- vom Betrag kleinste/größte Zahlen bei ca. $\pm 10^{-308}$ und $\pm 10^{308}$

Bedingte Befehle (1/3)

- Mit **bedingten Befehlen** Sprünge vermeiden
  - gut für Pipelining
  - keine Branch Prediction nötig
- Führe einen Befehl nur aus, wenn eine bestimmte Bedingung erfüllt ist



Bedingte Befehle (2/3)

- MMIX hat dafür die Befehle CS**xx** und ZS**xx**:
  - CS**xx** `$X,$Y,$Z`:
    $X ← $Z, falls $Y die Bedingung **xx** erfüllt.
  - ZS**xx** `$X,$Y,$Z`:

$$\$X \leftarrow \begin{cases} \$Z, \text{ falls } \$Y \text{ die Bedingung } \mathbf{xx} \text{ erfüllt} \\ 0, \quad \text{sonst} \end{cases}$$

Bedingte Befehle (3/3)

Andere Prozessoren

- **ARM:** Jede Instruktion hat ein 4-Bit Condition-Field, in dem angegeben wird, von welchen Flags die Ausführung des Befehls abhängen soll, z. B.: `MOV MI`: „Move if Minus'“
- **x86:** Seit Pentium Pro: Conditional Move `CMOVxx` (Reg,Reg und Reg,Mem)

SIMD-Befehle (1/2)

- **Intel Multimedia-Extensions MMX**, 1996
  - Erste Implementierung einer Parallel-Architektur in einem Standard-Mikroprozessor:
  - Acht 64-Bit-Register MM0 ... MM7 enthalten jeweils
    - acht unabhängige Bytes (Packed Bytes)
    - vier unabhängige Wydes (Packed Words)
    - zwei unabhängige Tetras (Packed Double-Words)

    dabei **saturierte** Operationen ohne Überlauf zwischen den unabhängigen Worten möglich ($a+b > max$ => add(a, b) = max)

SIMD-Befehle (2/2)

- **AMD 3DNow!**, 1998 im K6-2
  - SIMD mit je zwei Floating-Point-Werten
- **Intel Streaming SIMD Extensions (SSE)**
  - Zusätzlich acht 128-Bit Register xmm0 ... xmm7
  - Zwei oder vier unabhängige (Float-) Operanden je Register möglich
  - Zwei-Adress-Befehle, z. B. `sqrtps xmm0,xmm1`
- **Motorola Altivec**
  - 128-Bit-Vektoren, die als vier 32-Bit-Floats mit Drei-Adress-Befehlen bearbeitet werden können.

Speicherzugriff: Big- und Little-Endian (1/3)

- *Endianness*, *Byte Order* und *Byte Sex*
  - big endian: ABCD → AB, CD („große“ Positionen zuerst)
  - little endian: ABCD → CD, AB („kleine“ Positionen zuerst)
- Frage: Was gibt dieses Programm aus?

```
#include <stdio.h>
void main () {
  long long ll = 0X123456789abcdef;
  unsigned char* pc = (unsigned char*)(&ll);
  int i;
  for (i=0; i<8; i++)
    printf("%02hx ", *(pc+i));
}
```

- Ausgaben:
  - `ef cd ab 89 67 45 23 01` (little-endian; Intel)
  - `01 23 45 67 89 ab cd ef` (big-endian; MMIX, Motorola)

Speicherzugriff: Big- und Little-Endian (2/3)

- Bei ARM-Prozessoren ist das Format einstellbar.
- Vorsicht beim Einlesen von binär gespeicherten Daten: Z. B. speichern
  - das BMP-Format little-endian und
  - das JPG-Format und Java-Class-Files big-endian.
- Beispiel: BMP-Datei

Speicherzugriff: Big- und Little-Endian (3/3)

```
$ hexdump -C testbild.bmp | head -3
00000000  42 4d 36 4b 00 00 00 00  00 00 36 00 00 00 28 00
00000010  00 00 9f 00 00 00 28 00  00 00 01 00 18 00 00 00
00000020  00 00 00 4b 00 00 11 0b  00 00 11 0b 00 00 00 00

$ ls -l testbild.bmp
-rw-r--r-- 1 esser esser 19254 2010-10-20 13:42 testbild.bmp
```

- 36 4B: Dateigröße, hier: $4B36_h$ = 19254 Bytes

Übung heute nachmittag

- Heute: Einführung in die Programmiersprache **Python**
- Wenn Sie Python schon kennen: Freistunde...

- in 3 Gruppen (13:30, 15:15, 17:00) im PC-Raum R1.009

Vorschau 28.10.2010

- Leistungsmessung und Leistungsbewertung (Benchmarks)