Technische Informatik II
Rechnerarchitektur

1.Einführung in MMIX

Matthias Dräger

E-Mail: mdraeger@mi.fu-berlin.de
www: www.matthias-draeger.info/lehre/sose2010ti2/
tinyurl.com/sose2010ti2

Zuletzt bearbeitet: 23.04.2010

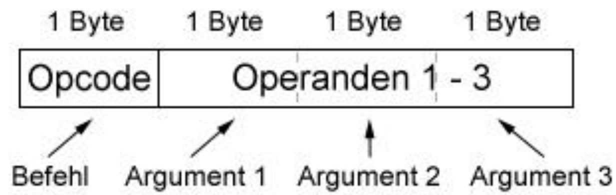
Was ist MMIX?

- virtueller 64-Bit-Modellcomputer
- Nachfolgemodell von MIX
- beschrieben von Donald E. Knuth in seinem Werk *The Art of Computer Programming*
- verwendet die Assembler-Sprache *MIXAL*
- hat Ähnlichkeit zu folgenden Prozessoren:
  - CrayI
  - IBM601
  - IBM801
  - RISCII
  - ClipperC300
  - AMD29K
  - Motorola88K
  - Inteli960
  - Alpha21164
  - POWER2
  - MIPSR4000
  - HitachiSuperH4
  - StrongARM110
  - Sparc64

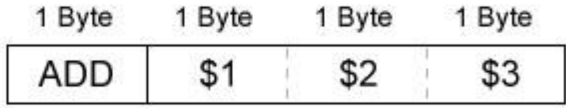
http://de.wikipedia.org/wiki/Donald_Ervin_Knuth

Architektur

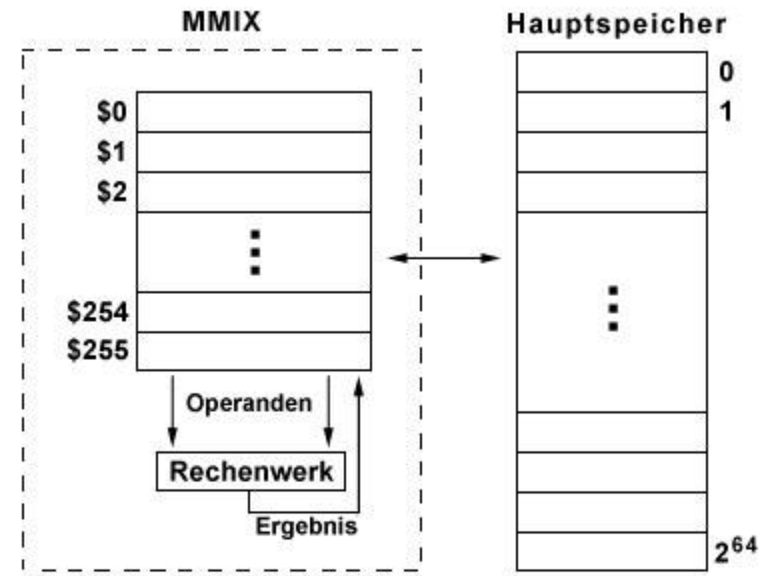
- MMIX besitzt $2^8 = 256$ allgemein verwendbare Register ($0 - $255)
  - Register = elektr. Schaltungen, die mehrere Bit (hier 64 Bit) digitale Informationen speichern können
- 32 Spezialregister: rA,…, rZ, rBB, rTT, rWW, rXX, rYY, rZZ
  - enthalten spezielle Informationen, wie z.B. Divisionsrest, Überläufe, Rücksprungadressen,…
- Rechenwerk (ALU) führt Operationen durch
- Befehlswort besteht aus 4 Byte



Bsp.: `ADD $1,$2,$3`

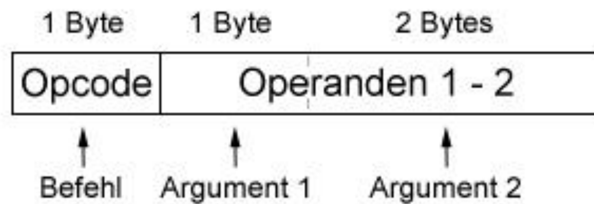


„addiere Register $2 mit $3 und schreibe das Ergebnis nach $1"

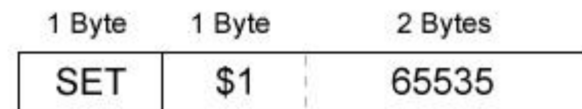


Architektur

- es gibt auch Befehle mit weniger als drei Operanden:
  - Befehl mit zwei Operanden
  - 2 Bytes = 2^{16} = 65.536
    - d.h.: $0 \le$ Operand $2 < 65.536$

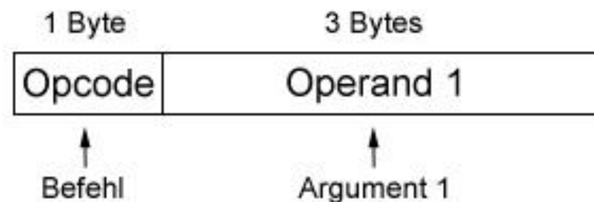


Bsp.: `SET $1,65535`

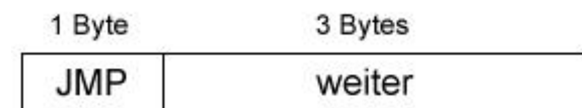


„setze in Register $1 den Wert 65535"

  - Befehl mit einem Operand



Bsp.: `JMP weiter`



„springe zur Marke *weiter*"

Freie Universität Berlin

Organisation des Hauptspeichers

Segment	Symbol	Startadresse	Zweck
Textsegment		#0 bis #FF	Interruptroutinen
Textsegment		#100	Programm und statische Daten
Datensegment	Data_Segment	#2000 0000 0000 0000	veränderliche Daten (z.B. Arrays, Texte, …)
Poolsegment	Pool_Segment	#4000 0000 0000 0000	Kommunikation zwischen Programm und Betriebssystem
Stacksegment	Stack_Segment	#6000 0000 0000 0000	Auslagerung der Inhalte von Registern bei Unterprogrammaufrufen

- der Pseudobefehl LOC legt die Adresse fest, ab der im weiteren Verlauf der Assemblierung der Speicher gefüllt wird
  - Pseudobefehl: ist kein MMIX-Befehl, sondern eine Anweisung an das Assemblerprogramm
- das Zeichen @ steht für „The place where we are at" und beinhaltet die aktuelle Adresse

Bsp. für LOC: `LOC #100` springt ins Textsegment

`LOC Data_Segment` springt ins Datensegment

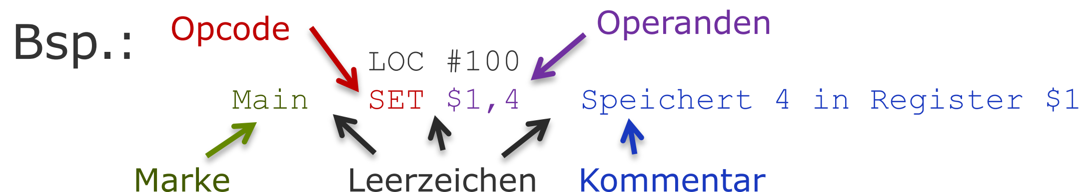
Konventionen in MMIX

1. Marken (Sprungmarken, Label) stehen an erster Position in einer Zeile. Bei der Marke `Main` beginnt das Programm.
2. Das erste Leerzeichen* in einer Zeile beendet die Marke – die nächsten Zeichen werden als (Pseudo-)Befehl (Opcode) aufgefasst.
3. Zwischen Befehl und dem ersten Argument (Operand) muss mindestens ein Leerzeichen* stehen.
4. Die Argumente (Operanden) werden lediglich durch Kommata getrennt (KEINE Leerzeichen!!!)
5. Das nächste Leerzeichen* steht für das Ende des Befehls. Alle weiteren Zeichen werden als Kommentar interpretiert.
6. Es gibt keine Kommentarzeichen.

   Möchte man in eine Zeile nur ein Kommentar schreiben, so muss das erste Zeichen ein Sonderzeichen sein (#, //, %, …), sonst wird es als Marke interpretiert.

* anstelle von Leerzeichen empfehlen sich Tabulatoren, um die Lesbarkeit zu verbessern

Bsp.:



Konstanten und Pseudobefehle

Konstanten

- Folgen von Dezimalziffern: 0,1,2,3,4,5,6,7,8,9
- Folgen von Hexadezimalziffern angeführt von einer Raute #, z.B. #1, #FF
- Zeichen in Hochkommata: 'a', 'b', 'C', '1'
- Zeichenketten: z.B. "Hallo" (entspricht: 'H', 'a', 'l', 'l', 'o')
  - **Hinweis:** Zeichenketten müssen mit 0 terminiert werden.

Pseudobefehle

- *Label* IS *Ausdruck*
  - Label ist nun ein Synonym für Ausdruck
  - Ausdruck kann eine Konstante oder ein Register sein

- *[Label]* LOC *Ausdruck*
  - bindet Label an die aktuelle Position im Speicher (Label ist optional)
  - Ausdruck muss eine Konstante sein

Pseudobefehle (2)

Pseudobefehle zum Vorbelegen von Speicher

- *[Label]* BYTE *Ausdruck*$_1$,…,*Ausdruck*$_n$
  - ein Ausdruck ist 1 Byte lang
  - in Label wird die Anfangsadresse von *Ausdruck*$_1$ gespeichert
  - erhöht die aktuelle Adressposition um n
  - Zusatzinfo: wird zum Speichern von Zeichenketten verwendet

- *[Label]* WYDE *Ausdruck*$_1$,…,*Ausdruck*$_n$
  - ein Ausdruck ist 1 Wyde (= 2 Byte) lang
  - erhöht die aktuelle Adressposition um $2 \cdot n$

- *[Label]* TETRA *Ausdruck*$_1$,…,*Ausdruck*$_n$
  - ein Ausdruck ist 1 Tetra (= 2 Wyde = 4 Byte) lang
  - erhöht die aktuelle Adressposition um $4 \cdot n$

- *[Label]* OCTA *Ausdruck*$_1$,…,*Ausdruck*$_n$
  - ein Ausdruck ist 1 Octa (= 2 Tetra = 4 Wyde = 8 Byte) lang
  - erhöht die aktuelle Adressposition um $8 \cdot n$

Unser erstes Programm

Das Programm soll folgende Formel berechnen: $z = \frac{a+b}{a \cdot c}$

Mit den Werten: a = 2   b = 8   c = 5

erstes-programm.mms

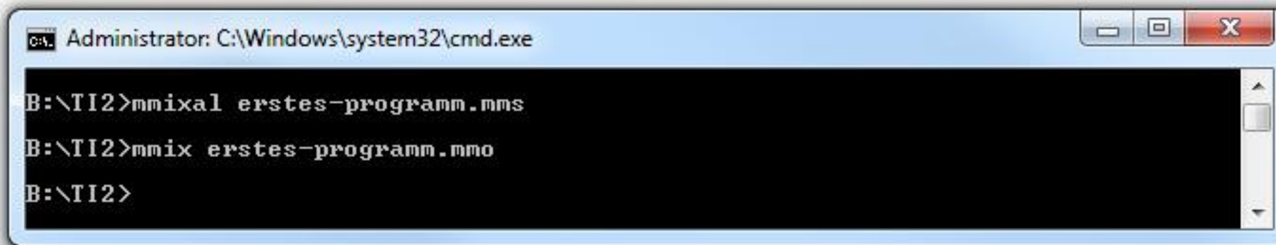
```
###############################################
# berechnet: z = (a+b)/(a*c) mit a=2, b=8, c=5
###############################################
          LOC #100
a         IS        $1        Synonym a für Register 1
b         IS        $2        Synonym b für Register 2
c         IS        $3        Synonym c für Register 3
z         IS        $4        Synonym z für Register 4

Main      SET       a,2       setze a auf 2
          SET       b,8       setze b auf 8
          SET       c,5       setze c auf 5

          ADD       b,a,b     b = a + b  Zähler berechnen
          MUL       a,a,c     a = a * c  Nenner berechnen
          DIV       z,b,a     z = b / a (ganzzahlige Division)

          TRAP      0,Halt,0  beende das Programm
```

Hinweis: Bei dem Befehl `DIV` handelt es sich um eine ganzzahlige Division. Entsteht bei der Division ein Rest, so wird dieser im Spezialregister rR abgespeichert.

Programme assemblieren

Schritt 1: Mit MMIXAL die Quelldatei `erstes-programm.mms` assemblieren

Schritt 2: Die entstandene Objektdatei `erstes-programm.mmo` mit MMIX ausführen

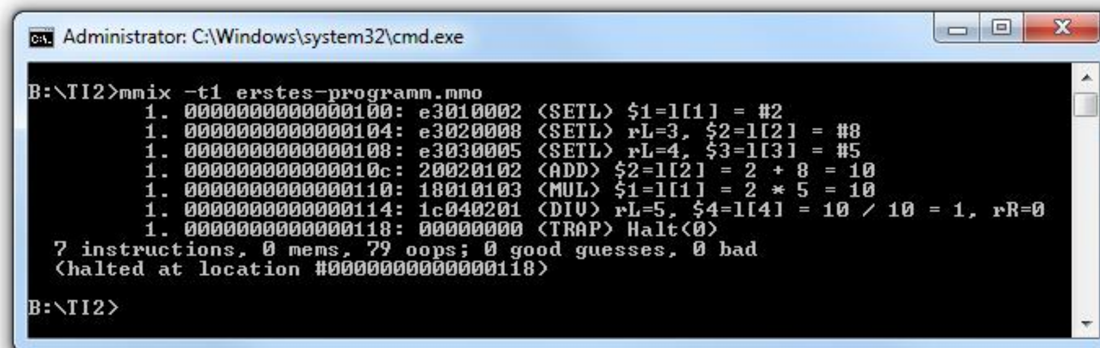


Programme debuggen

Kommandozeilen-Optionen von MMIX

```
Usage: mmix <options> progfile command-line-args...
 with these options: (<n>=decimal number, <x>=hex number)
-t<n> trace each instruction the first n times
-e<x> trace each instruction with an exception matching x
-r    trace hidden details of the register stack
-l<n> list source lines when tracing, filling gaps <= n
-s    show statistics after each traced instruction
-P    print a profile when simulation ends
-L<n> list source lines with the profile
-v    be verbose: show almost everything
-q    be quiet: show only the simulated standard output
-i    run interactively (prompt for online commands)
-I    interact, but only after the program halts
-b<n> change the buffer size for source lines
-c<n> change the cyclic local register ring size
-f<filename> use given file to simulate standard input
-D<filename> dump a file for use by other simulators
```

Trace von `erstes-programm.mmo`



Hello World

```
% helloworld.mms: Beispiel vom Übungsblatt 1, Aufg.3
string  BYTE  "Hallo Welt!",#a,0  % auszugebende Zeichenkette (#a ist ein Zeilenumbruch,
                                  % 0 schließt die Zeichenkette ab)
Main    GETA  $255,string         % Adresse der Zeichenkette in Register $255 ablegen
        TRAP  0,Fputs,StdOut      % Zeichenkette, auf die mit Register $255
                                  % verwiesen wird, nach StdOut ausgeben (Systemaufruf)
        TRAP  0,Halt,0            % Prozess beenden
```

Hinweis: Der Befehl **GETA** (get address) kann nur auf Speicheradressen zugreifen, die nicht weiter als #FFFF Speicheradressen entfernt sind.

Sonst: Fehlermeldung von MMIXAL:
"helloworld2.mms", line 7: relative address is more than #ffff tetrabytes forward!
(One error was found.)

```
% verbesserte Version helloworld2.mms: Daten gehören ins Data_Segment!
        LOC   Data_Segment        % ins Data_Segment springen
        GREG  @
string  BYTE  "Hallo Welt!",#a,0  % auszugebende Zeichenkette (#a ist ein Zeilenumbruch,
                                  % 0 schließt die Zeichenkette ab)
        LOC   #100
Main    LDA   $255,string         % GETA funktioniert hier nicht, da das Data_Segment zu
                                  % weit entfernt ist (von #100)
        TRAP  0,Fputs,StdOut      % Zeichenkette, auf die mit Register $255
                                  % verwiesen wird, nach StdOut ausgeben (Systemaufruf)
        TRAP  0,Halt,0            % Prozess beenden
```

Textausgabe

```
% textausgabe.mms - mehrzeilig Text ausgeben
           LOC Data_Segment      % ins Data_Segment springen
% Wichtig: aktuelle Adresse mit GREG in ein globales Register speichern, sonst funktioniert
% LDA nicht, da für den Befehl eine "Referenzadresse" benötigt wird.
           GREG @
% mehrzeiliger Text in einer Zeile
msg1       BYTE  "Hello World",#a,9,"MMIX macht Spass",10,9,9,";) ",1,0
% mehrzeiligen Text in mehreren Zeilen
msg2       BYTE  "Hello World",#a
           BYTE  9,"MMIX macht Spass",10
           BYTE  9,9,";) ",1,0
% 2 Leerzeilen
lf         BYTE       10,10,0    % zwei Leerzeilen
           % unser Hauptprogramm
           LOC #100
Main       LDA $255,msg1          % Speichere Nachrichtenadresse nach $255
           TRAP 0,Fputs,StdOut    % Ausgabe vom globalen Register $255
           LDA  $255,lf          % Leerzeilen ausgeben
           TRAP 0,Fputs,StdOut
           LDA  $255,msg2        % Speichere Nachrichtenadresse nach $255
           TRAP 0,Fputs,StdOut   % Ausgabe vom globalen Register $255
           TRAP 0,Halt,0         % Programm beenden
```

Sprungbefehle

- Ziel: Man möchte zu einer bestimmten Stelle im Programm springen.
  - in Java nicht vorhanden, aber in Pascal, C, C++, C#, PHP (seit 2009)
- in MMIX gibt es zwei Sprungbefehle:
  - JMP *Marke* — Der Assembler springt zur angegebenen Marke im Programm.

```
% jmp-weiter.mms
        LOC #100
Main    JMP weiter      % springe zur Marke "weiter"
        SET $0,5        % setze $0 auf 5
        JMP stopp       % springe zur Marke "stopp"
weiter  SET $0,6        % setze $0 auf 6
stopp   TRAP 0,Halt,0   % beende Programm
```

Welcher Wert steht in Register $0? → 6, denn `SET $0,5` wird nie ausgeführt.

  - GO $X,$Y,$Z|Z
    - die Adresse des nächsten Befehls (der ohne Ausführung von GO gekommen wäre) wird in $X gespeichert
    - Assembler berechnet aus $Y und Z eine absolute Sprungadresse und springt dorthin
    - mit GO kann nur rückwärts gesprungen werden!

Kontrollstrukturen

- Ziel: Wenn etwas den Wert *a* hat, dann tue *b*, sonst *c*.
  - in Java (und vielen anderen Programmiersprachen): if-then-else-Anweisung
- werden in MMIX mit Sprüngen zu Marken realisiert
- die folgenden Befehle prüfen Register $X und Springen (im Erfolgsfall) nach YZ

BZ $X,YZ	(branch if zero)	springe nach YZ, wenn **$X den Wert 0** enthält
BNZ $X,YZ	(branch if nonzero)	springe nach YZ, wenn **$X nicht den Wert 0** enthält
BN $X,YZ	(branch if negative)	springe nach YZ, wenn **$X negativ** ist
BNN $X,YZ	(branch if nonnegative)	springe nach YZ, wenn **$X nicht negativ** ist
BP $X,YZ	(branch if positive)	springe nach YZ, wenn **$X positiv** ist
BNP $X,YZ	(branch if nonpositive)	springe nach YZ, wenn **$X nicht positiv** ist
BOD $X,YZ	(branch if odd)	springe nach YZ, wenn **$X ungerade** ist
BEV $X,YZ	(branch if even)	springe nach YZ, wenn **$X gerade** ist

- um zwei Zahlen zu Vergleichen, benutzt man den Befehl CMP:

```
CMP $X,$Y,$Z (compare)  vergleicht $Y und $Z und speichert folgendes Ergebnis in $X:
```

$$\$X = \begin{cases} -1, \text{falls } \$Y < \$Z \\ 0, \text{falls } \$Y = \$Z \\ 1, \text{falls } \$Y > \$Z \end{cases}$$

Freie Universität Berlin

Beispiel: Kontrollstrukturen

- Ziel: Wenn c > 5, dann rechne c = c + 1, sonst c = c - 1.

```
% beispiel-if.mms
           LOC        #100
c          IS         $1
x          IS         $2         % Variable zum Zwischenspeichern des Ergebnisses von CMP
Main       SET        c,7        % setze Startwert  c = 7
           CMP        x,c,5      % Vergleiche c mit 5

           % x kann nur drei verschiedene Werte annehmen:
           %     -1, wenn c < 5
           %      0, wenn c = 5
           %      1, wenn c > 5
           % Wir prüfen nun x auf nicht-positiv (BNP), um in den else-Zweig zu springen,
           % falls die Bedingung c > 5 nicht erfüllt wird.
           BNP        x,else

           % hier folgt nun der then-Zweig
           ADD        c,c,1       % rechne c = c + 1
           JMP        endelse     % überspringe den else-Zweig (springt zur Marke endelse)

           % jetzt kommt der else-Zweig
else       SUB        c,c,1       % rechne c = c - 1

endelse    TRAP       0,Halt,0 % beende Programm
```

Schleifen

- Ziel: Ein bestimmter Codeabschnitt soll beliebig oft ausgeführt werden
- in der Regel enthalten Assemblersprachen keine Schleifen
- man benutzt CMP, die Branch-Befehle und optional eine Laufvariable
- Arten von Schleifen:
  - vorprüfende Schleife

```
while(Bedingung){
    Anweisung1;
    Anweisung2;
}
```

  - nachprüfende Schleife

```
do{
    Anweisung1;
    Anweisung2;
}while(Bedingung);
```

  - Zählschleife

```
for(Startwert; Bedingung; Anweisung){
    Anweisung1;
    Anweisung2;
}
```

→ wird der Einfachheit halber in eine vor- oder nachprüfende Schleife umgewandelt

Vorprüfende Schleife

- Beispiel: Was wird berechnet?

```
          LOC #100
n         IS  $0              % Laufvariable
res       IS  $1              % Ergebnis speichern
Main      SET n,6             % setze n auf Startwert 6
          SET res,1           % Ergebnis mit 1 initialisieren

% prüfe nun, ob n nicht größer 0, sonst verlasse Schleife
% oder anders ausgedrückt: wiederhole, solange n > 0
while     BNP n,ewhile
          MUL res,res,n       % rechne: res = res * n
          SUB n,n,1           % n = n - 1
          JMP while           % springe zum Schleifenanfang
ewhile    TRAP 0,Halt,0       % beende Programm
```

→ Es wird n-Fakultät, also 6! berechnet.

- die Register haben folgenden Wert:
  - *n* hat den Wert 0
  - *res* hat den Wert 720

Nachprüfende Schleife

- Beispiel: Was wird berechnet?

```
            LOC #100
n           IS  $0
i           IS  $1              % Laufvariable
res         IS  $2              % Ergebnis speichern
tmp         IS  $3              % speichert Ergebnis von CMP

Main        SET n,6             % setze n auf Endwert 6
            SET i,1             % initialisiere Zählvariable
            SET res,0           % Ergebnis mit 0 initialisieren

loop        ADD res,res,i       % rechne: res = res + i
            ADD i,i,1           % erhöhe i um 1
            CMP tmp,i,n                   % vergleiche i mit n
            % wenn i > n, dann hat tmp den Wert 1
            % prüfen, ob tmp nicht positiv ist -> zum Schleifenanfang springen
            BNP tmp,loop
            TRAP 0,Halt,0       % beende Programm
```

→ Das Ergebnis ist: $res = \sum_{i=1}^{n} i$

- die Register haben folgenden Wert:
  - *n* hat den Wert 6
  - *i* hat den Wert 7
  - *res* hat den Wert 21

Zusammenfassung

- Architektur
- Aufbau des Hauptspeichers
- Einfache Programmierung in MMIX
  - Textausgabe
  - Speichern ins Daten-Segment
  - Sprungbefehle
  - Kontrollstrukturen
  - Schleifen