Freie Universität Berlin

Technische Informatik II
Rechnerarchitektur

MMIX-Crashkurs

Matthias Dräger, Markus Rudolph

E-Mail: mdraeger@mi.fu-berlin.de
rudolph@mi.fu-berlin.de

www: tinyurl.com/mmix2010
www.matthias-draeger.info/lehre/sose2010ti2/mmix.php

Zuletzt bearbeitet: 02.07.2010

Installation

Freie Universität Berlin

Installation

1. Binaries herunterladen von tinyurl.com/mmix2010
2. Entpacken in folgende Verzeichnisse:

XP: C:\Programme\mmix
Vista/7: C:\Program Files\mmix
64-Bit: C:\Program Files (x86)\mmix

(Genaue Instruktionen sind in der beigelegten PDF zu finden.)

oder falls Admin-Rechte notwendig sind (Poolrechner), dann im User-Ordner entpacken:

C:\Users\<Username>\mmix

/usr/bin

oder falls Root-Rechte notwendig sind, dann auf dem Desktop entpacken und:

$sudo cp Desktop/mmix/* /usr/bin

3. MMIX testen

Konsole/Terminal öffnen und „mmix" eintippen

Einführung

- Hauptspeicher besteht aus fünf Segmenten
  - Interrupt-Vektortabelle
  - Code-Segment (MMIX-Befehle)
  - Data-Segment (Daten, z.B. Strings)
  - Pool-Segment
  - Stack-Segment

- an eine beliebige Speicheradresse (als Hexadezimalzahl) springt man mit dem Pseudobefehl:

  LOC <Adresse> (Location)

Bsp: Um ins Code-Segment zu springen:

LOC #100

Hauptspeicher

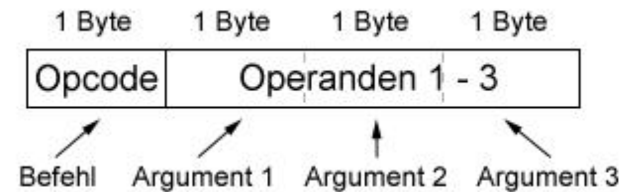
	#0
Interrupt-Vektortabelle	⋮
	#FF
	#100
Code-Segment	⋮
	#1FFF FFFF FFFF FFFF
	#2000 0000 0000 0000
Data-Segment	⋮
	#3FFF FFFF FFFF FFFF
	#4000 0000 0000 0000
Pool-Segment	⋮
	#5FFF FFFF FFFF FFFF
	#6000 0000 0000 0000
Stack-Segment	⋮
	#7FFF FFFF FFFF FFFF
	#8000 0000 0000 0000
Reserviert für das Betriebssystem	⋮
	#FFFF FFFF FFFF FFFF

(Die Zahlen mit der Raute (#) sind Hexadezimalzahlen und stellen eine Speicheradresse dar.)

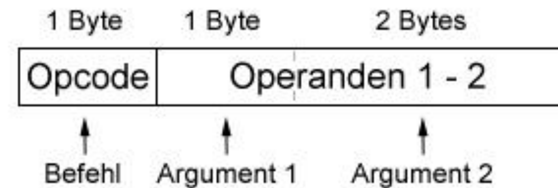
- in MMIX existieren 256 Register, die man mit $0 - $255 anspricht
- in Registern können Konstanten, Zahlen oder Speicheradressen gespeichert werden
- 32 Spezialregister mit spez. Informationen, wichtig sind:
  - rR (Divisionsrest)
  - rJ (Rücksprungadresse)

  → Werte können mit dem Befehl GET ausgelesen werden

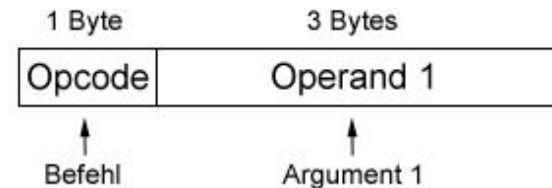
- jeder Befehl besteht aus vier Byte
- 1 Byte für den Befehl und 3 Byte für die Operanden
  - es gibt Befehle mit…
    - drei Operanden



    - zwei Operanden

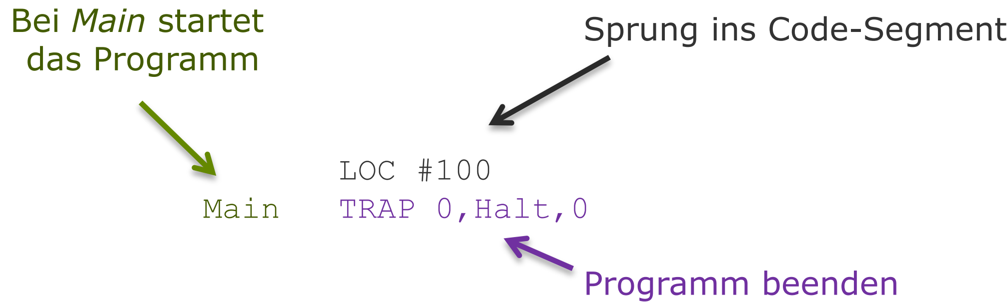


    - oder einem Operand

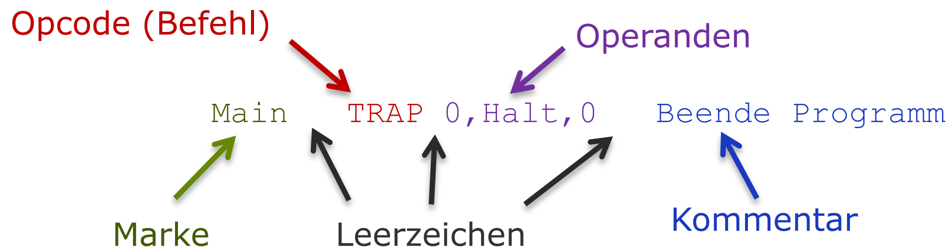


Konventionen in MMIX

Was jedes Programm beinhalten muss:



Genaue Betrachtung der Code-Zeile:



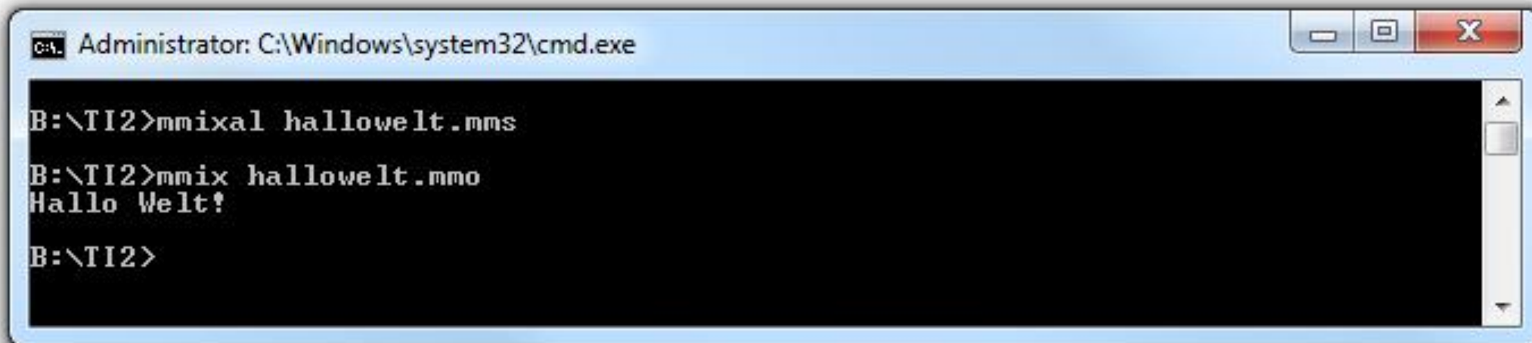
Unser erstes MMIX-Programm

Hallo Welt!

```
% hallowelt.mms:

        LOC   #100
string  BYTE  "Hallo Welt!",#a,0   % Zeichenkette anlegen
Main    GETA  $255,string          % Adresse der Zeichenkette holen
        TRAP  0,Fputs,StdOut       % gibt das aus was als Adresse in $255 steht
        TRAP  0,Halt,0             % Programm beenden
```

Das Programm assemblieren und ausführen:



Hallo Welt im Data-Segment

Nun wollen wir das Data-Segment benutzen:

```
        LOC Data_Segment
        GREG @
string  BYTE  "Hallo Welt!",#a,0

        LOC   #100
Main    LDA   $255,string
        TRAP  0,Fputs,StdOut
        TRAP  0,Halt,0
```

Kontrollstrukturen

Gerade oder ungerade?

Aufgabe 1: Gesucht wird ein Programm, das prüft, ob eine Zahl gerade oder ungerade ist und dementsprechend eine Meldung ausgibt.

```
% evenodd.mms
            LOC Data_Segment
            GREG @
gerade      BYTE "Zahl ist gerade",#a,0
ungerade    BYTE "Zahl ist ungerade",#a,0

            LOC #100
Zahl        IS  $0
Rest        IS  $1
tmp         IS  $2

Main        SET Zahl,91

            ???

            TRAP 0,Halt,0
```

Nützliche Befehle:
- DIV (dividieren)
- GET (Wert von Spez.-Register holen)
- BZ (Springe, wenn Null)
- BNZ (Springe, wenn nicht Null)
- JMP (Springe zu einer Marke)

In einer Hochsprache ausgedrückt:

```
char* gerade   = "Zahl ist gerade\n";
char* ungerade = "Zahl ist ungerade\n";

Zahl = 91
if(Zahl % 2 == 0) {
  output(gerade);
} else {
  output(ungerade);
}
```

Schleifen

Dezimal in Binär

Aufgabe 2: Schreibe `evenodd.mms` derart um, dass die Dezimalzahl als Binärzahl dargestellt wird.

```
% dec2bin.mms
            LOC Data_Segment
            GREG @
gerade      BYTE "0",0
ungerade    BYTE "1",0
newline     BYTE #a,0

            LOC #100
Zahl        IS  $0
Rest        IS  $1

Main        SET Zahl,9
loop        DIV Zahl,Zahl,2
            GET Rest,rR
            ???
            TRAP 0,Halt,0
```

In einer Hochsprache ausgedrückt:

```
char* gerade   = "0";
char* ungerade = "1";
char* newline  = "\n";

Zahl = 91
while ( Zahl != 0 ) {
  if(Zahl % 2 == 0) {
    output(gerade);
  } else {
    output(ungerade);
  }
  Zahl = Zahl / 2;
}
output(newline);
```

Ausgabe von Zahlen

Dezimal in Binär mit Bufferausgabe

Aufgabe 3: Erweitere `dec2bin.mms`, sodass die Binärziffern in einen Buffer geschrieben werden und dieser ausgegeben wird.

```
% dec2bin_store.mms
            LOC Data_Segment
            GREG @
bufsize     IS 8
buffer      BYTE "        ",#a,0
newline     BYTE #a,0

            LOC #100
Zahl        IS  $0
Rest        IS  $1
i           IS  $2
adr         IS  $3

Main        SET Zahl,9
            SET i,bufsize
            ???
            TRAP 0,Halt,0
```

In einer Hochsprache ausgedrückt:

```
int bufsize    = 8;
char* buffer   = "        \n";
char* newline  = "\n";

int Zahl   = 9;
int i      = bufsize;
do {
  i = i - 1;
  buffer[i] = (Zahl % 2) + '0';
  Zahl = Zahl / 2;
} while ( Zahl != 0 );

output(buffer);
```

Unterprogramme

Schematische Darstellung (Unterprogramme)

$255	
$254	
⋮	
$7	
$6	
$5	
$4	
$3	40
$2	30
$1	20
$0	10

Übergebe dem Unterprogramm alle Register > $2 und springe zur Marke ***bsp***

PUSHJ $2,bsp

$255	
$254	
⋮	
$4	
$3	
$2	
$1	
$0	40

Unterprogramm wird ausgeführt.

$255	
$254	
⋮	
$4	
$3	
$2	
$1	
$0	100

POP 1,0

$255	
$254	
⋮	
$7	
$6	
$5	
$4	
$3	
$2	100
$1	20
$0	10

Es gibt **einen** Rückgabewert ($0).

```
        LOC #100

% Unterprogramm anlegen
bsp     ADD   $0,$0,60          % erhöht Register $0 um 60
        POP   1,0               % eine Rückgabe -> $0

% Hauptprogramm
Main    SET    $0,10            % setze Register $0 mit Wert 10
        SET    $1,20
        SET    $2,30
        SET    $3,40            % dieser Wert soll später übergeben werden

        PUSHJ $2,bsp            % rufe Unterprogramm bsp auf
        TRAP 0,Halt,0
```

Dezimal in Binär

Aufgabe 4: Lagere die Hauptfunktionalität aus `dec2bin_store.mms` in ein Unterprogramm namens `dec2bin` aus.

```
% dec2bin_push.mms
            LOC Data_Segment
            GREG @

bufsize     IS 8
buffer      BYTE "        ",#a,0
newline     BYTE #a,0

            LOC #100
Zahl        IS  $1
Main        SET Zahl,5
            PUSHJ $0,dec2bin
            TRAP 0,Halt,0
```

In einer Hochsprache ausgedrückt:

```
int bufsize    = 8;
char* buffer   = "        \n";
char* newline  = "\n";

void dec2bin(int num) {
  int i = bufsize;
  do {
    i = i - 1;
    buffer[i] = (num % 2) + '0';
    num = num / 2;
  } while ( num != 0 );

  output(buffer);
}

int zahl = 9;
dec2bin(zahl);
zahl = 11;
dec2bin(zahl);
```

Rekursion

Bestandteile einer Rekursion

- eine Rekursion benötigt einen Rekursionsanker
- der Wert von rJ (Rücksprungadresse) muss in einem Register gesichert werden, damit er nicht überschrieben wird
  - GET $0,rJ (holt die Rücksprungadresse)
  - PUT rJ,$0 (speichert eine Rücksprungadresse)

```
fak_n   IS  $0
adr_rJ  IS  $1
res     IS  $2
tmp     IS  $3

fak     SUB tmp,fak_n,1

        BNP tmp,fak_end
        GET adr_rJ,rJ

        PUSHJ res,fak
        MUL fak_n,fak_n,res
        PUT rJ,adr_rJ

fak_end POP 1,0
```

Aufruf mit:

```
  SET $1,6
  PUSHJ $0,fak
```

Rekursive Ausgabe

Aufgabe 5: Ändere das Programm `dec2bin_push.mms` derart ab, dass das Unterprogramm `dec2bin` sich rekursiv aufruft und die Zahl Stellenweise ausgibt.

```
% dec2bin_rek.mms
            LOC Data_Segment
            GREG @

buffer      BYTE 0,0
newline     BYTE #a,0

            LOC #100
Zahl        IS  $1
Main        SET Zahl,5
            PUSHJ $0,dec2bin

            LDA $255,newline
            TRAP 0,Fputs,StdOut
            TRAP 0,Halt,0
```

In einer Hochsprache ausgedrückt:

```
char* buffer   = " ";
char* newline  = "\n";

void dec2bin(int num) {
  int rest = num % 2;
  num = num / 2;
  dec2bin(num);
  rest = rest + '0';
  buffer[0] = rest;
  output(buffer);
}

dec2bin(5);
```

Datenstruktur Array

Speicherzugriffe

Aufgabe 6:
Berechne den Mittelwert aus dem gegebenen Array und gebe ihn rekursiv aus.

```
% array.mms
            LOC Data_Segment
            GREG @
adr         IS   @
array       OCTA 10,20,30,40,50
nr          IS  (@-adr)/8

buffer      BYTE 0,0
newline     BYTE #a,0
```

Nützliche Befehle:
- PUSHJ (Unterprogramm aufrufen)
- POP (Unterprogramm beenden)
- GET (Wert Spez.-Register laden)
- LDO (Lade OCTA)
- ADD (addiere)

In einer Hochsprache ausgedrückt:

```
int[] array = {10,20,30,40,50}
int nr = 5;

int summe = 0;
int i     = nr;

do {
  i = i - 1;
  int no = array[i];
  summe = summe + no;
} while(i!=0);

int mittel = summe / nr;
print(mittel);

void print(int num) {
  int rest = num % 10;
  num = num / 10;
  print(num);
  rest = rest + '0';
  buffer[0] = rest;
  output(buffer);
}
```

Danke!