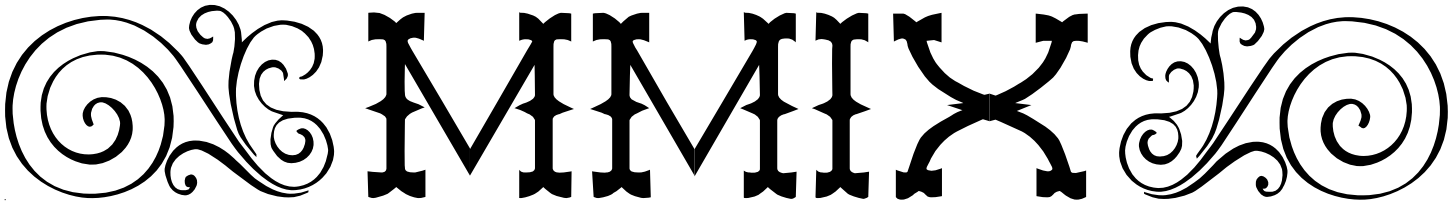
# Kapitel 11: MMIX

# RISC Maschine MMIX




- Ausgesprochen wie „emmix“
- RISC Maschine von Donald E. Knuth, Stanford University
- Wieso diese? Modern, durchdacht – und vor allem – komplett mit Assembler und Simulator für Windows, Mac und Linux

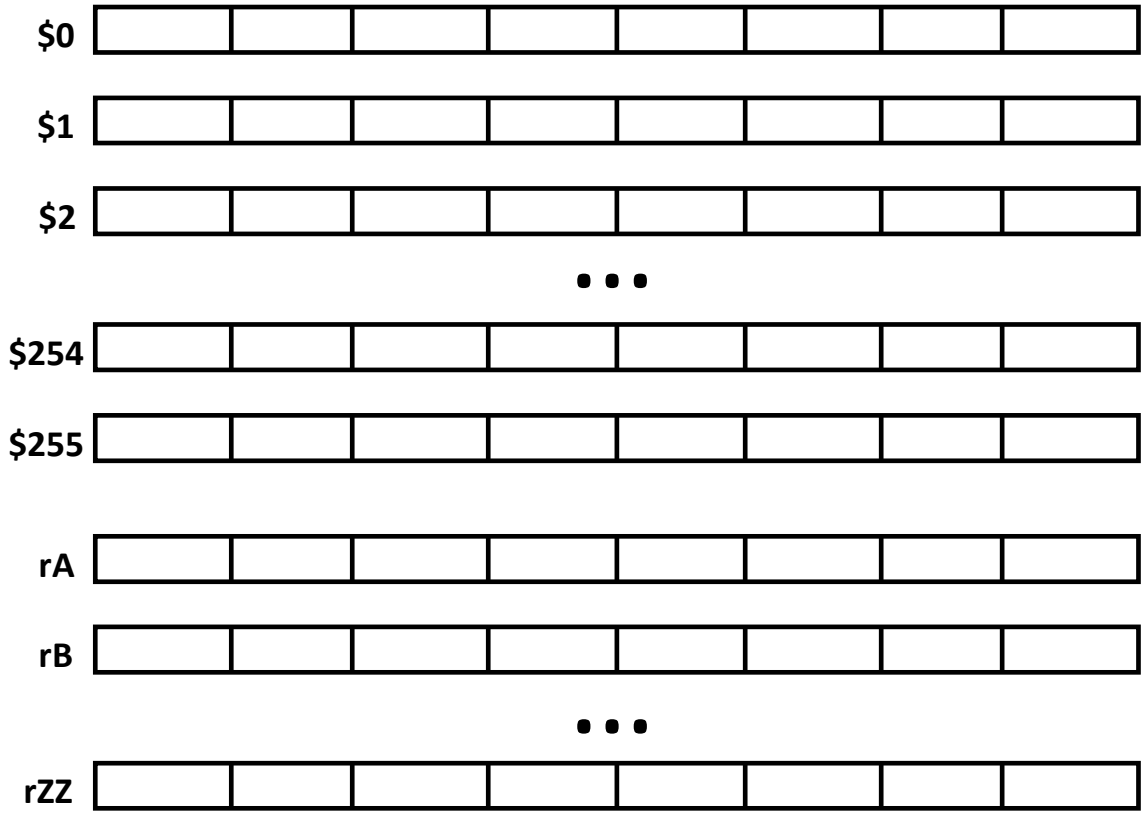
Quelle: http://mmix.cs.hm.edu

# Software/Dokumentation

- Assembler, Simulator und Handbuch:

http://mmix.cs.hm.edu/

# Register- und Speicherstruktur von MMIX




256 general purpose Register

32 special purpose Register

$2^{64}$ Speicherzellen adressierbar, 1 Byte pro Zelle

| M[0] | M[1] | M[2] | M[3] | M[4] | M[5] | M[6] | M[7] | ... |
|---|---|---|---|---|---|---|---|---|
| ... | | | | | | $M[2^{64}-3]$ | $M[2^{64}-2]$ | $M[2^{64}-1]$ |

# Allzweck-Register

- 256 Register mit jeweils 64 Bit (8 Byte), benannt mit $0, $1, …, $254, $255
  - Hier zum besseren Verständnis:  $X, $Y, $Z

- Je nach Kontext aufgefasst als
  - Bitmaske
  - vorzeichenlose Ganzzahl
  - vorzeichenbehaftete Ganzzahl
  - Gleitkommazahl

# Spezial-Register (Auswahl)

- rA (Arithmetik-Ausnahmen)
- rH (Erweitertes Resultat für Multiplikation)
- rR (Rest bei Division)
- rJ (Rücksprungadresse für Unterprogramm-Aufrufe)
- rM (Multiplex-Mask)
- rG, rL (Global/Local threshold register)

# Speicherorganisation


1 Wyde = 2 Bytes $M_2[4] = M_2[5]$

1 Tetra = 2 Wydes $M_4[4] = M_4[7]$

1 Octa = 2 Tetras $M_8[4] = M_8[0]$

- Die $2^{64}$ Bytes Speicher werden zu $2^{63}$ Wydes gruppiert und mit $M_2[n]$ bezeichnet.

  $M[0]M[1] = M_2[0] = M_2[1]$, ..., $M[2k]M[2k+1] = M_2[2k] = M_2[2k+1]$

- Die $2^{63}$ Wydes werden zu $2^{62}$ Tetras gruppiert ($M_4[n]$).

  $M[0]M[1]M[2]M[3] = M_4[0] = M_4[1] = M_4[2] = M_4[3]$

- Analog für $M_8[n]$

# Beispiel

| $a$ | 7F | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 |
|---|---|---|---|---|---|---|---|---|---|---|
| $M_1[a]$ | $00_{16}$ | $ff_{16}$ | $00_{16}$ | $80_{16}$ | $01_{16}$ | $02_{16}$ | $03_{16}$ | $04_{16}$ | $05_{16}$ | $06_{16}$ |

- $M_1[81]$ = 0x00
- $M_2[81]$ = 0xff00
- $M_4[81]$ = 0xff008001
- $M_8[81]$ = 0xff00800102030405

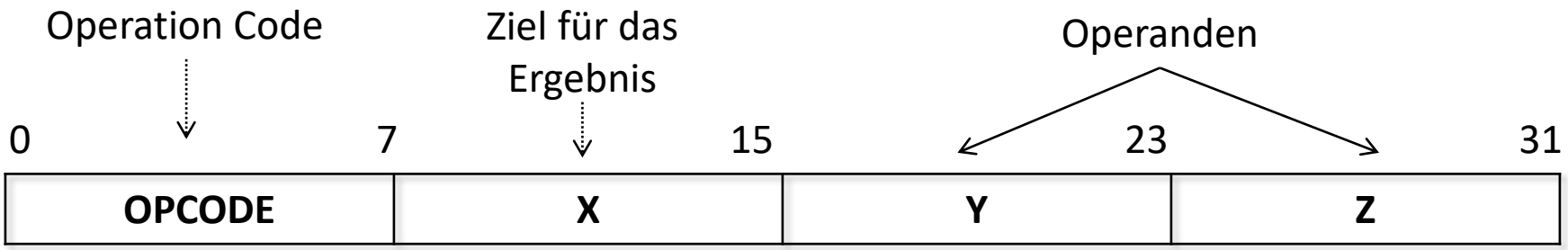
# Klassen von Befehlen

- Arithmetisch-Logische Befehle (Ganzzahl)
- Floating-Point-Befehle
- Load- / Store-Befehle
- Sprungbefehle
- Kontrollbefehle (für das Betriebssystem)

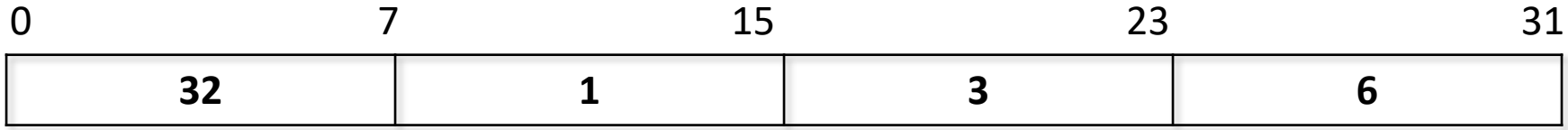
Allgemein:

- Alle Instruktionen sind 4 Bytes lang und im Speicher auf durch 4 teilbare Adressen ausgerichtet (siehe $M_4$).

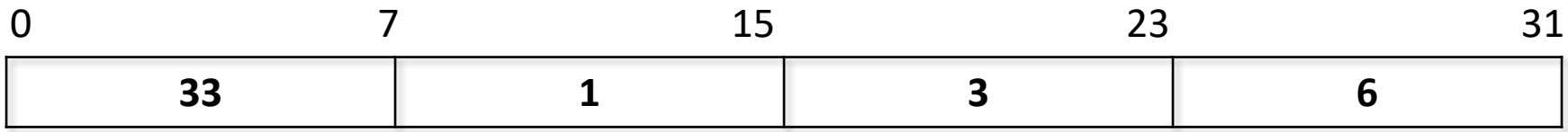
# Einheitliches Befehlsformat




- Beispiel:
- ADD $1,$3,$6 $1 := $3 + $6




- ADD $1,$3,6 $1 := $3 + 6



# Arithmetik

- ADD, SUB, MUL, DIV, NEG
- ADDU, SUBU, MULU, … (vorzeichenlose Variante)
- rA für Ausnahmen, rH für MUL, DIV

- Da bspw. “+” im 2er-Komplement wie “normale” Addition funktioniert, unterscheiden sich viele **U-Befehle nur im Setzen von Überlauf-Bits.

# Logische (Bitweise) Befehle

- AND, OR, XOR
- NAND, NOR, NXOR
- ANDN, ORN
- SL, SR (bit shift left/right)
- MUX (multiplex)
  - X := (Y and rM) or (Z and !rM)
- SADD (sideways add)
  - „zählt gesetze Bits“

# Vergleiche

- CMP $X,$Y,$Z
  - $X := 0, falls $Y==$Z
  - $X := 1, falls $Y > $Z (2er Komplement)
  - $X := -1, falls $Y < $Z (2er Komplement)

- CMPU $X,$Y,$Z
  - $X := 0, falls $Y==$Z
  - $X := 1, falls $Y > $Z (ohne Vorzeichen)
  - $X := -1, falls $Y < $Z (ohne Vorzeichen)

# Bedingte Zuweisungen

- CS<*condition*> $X,$Y,$Z
- „conditional set“
- if condition ($Y) then $X := $Z

- ZS<condition> $X,$Y,$Z
- „zero or set“
- if condition ($Y) then $X := $Z else $X := 0

- Bedingungen:
  - CSN (negative), CSNN (nonnegative)
  - CSZ (zero), CSNZ (nonzero)
  - CSP (positive), CSNP (nonpositive)
  - CSOD (odd), CSEV (even)
  - Ebenso für ZS<condition>

# Laden/Speichern

- MMIX ist RISC
- Load (LD) und Store(ST)-Anweisungen

Beispiele:

- LDB $X,$Y,$Z „load byte“ $X := $M_1$[$Y+$Z]
- LDWU $X,$Y,$Z „load wyde“ $X := $M_2$[$Y+$Z]
- STT $X,$Y,$Z „store tetra“ $M_4$[$Y+$Z] := $X

# Laden aus Speicher (1/2): unsigned

| $a$ | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 |
|---|---|---|---|---|---|---|---|---|---|---|
| $M_1[a]$ | $00_{16}$ | $ff_{16}$ | $00_{16}$ | $80_{16}$ | $01_{16}$ | $02_{16}$ | $03_{16}$ | $04_{16}$ | $05_{16}$ | $06_{16}$ |

- Sei \$Y = 80, \$Z = 3
- LDBU \$X,\$Y,\$Z — \$X := $M_1[83]$ = 00 00 00 00 00 00 00 $01_{16}$
- LDWU \$X,\$Y,\$Z — \$X := $M_2[83]$ = 00 00 00 00 00 00 $80\ 01_{16}$
- LDTU \$X,\$Y,\$Z — \$X := $M_4[83]$ = 00 00 00 00 $ff\ 00\ 80\ 01_{16}$
- LDOU \$X,\$Y,\$Z — \$X := $M_8[83]$ = $ff\ 00\ 80\ 01\ 02\ 03\ 04\ 05_{16}$

- \$X := $M_x$[\$Y+\$Z]

# Laden aus Speicher (2/2): signed

| $a$ | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 |
|---|---|---|---|---|---|---|---|---|---|---|
| $M_1[a]$ | $00_{16}$ | $ff_{16}$ | $00_{16}$ | $80_{16}$ | $01_{16}$ | $02_{16}$ | $03_{16}$ | $04_{16}$ | $05_{16}$ | $06_{16}$ |

- Sei \$Y = 80, \$Z = 3
- LDB \$X,\$Y,\$Z — \$X := $M_1[83] = 00\ 00\ 00\ 00\ 00\ 00\ 00\ 01_{16}$
- LDW \$X,\$Y,\$Z — \$X := $M_2[83] = ff\ ff\ ff\ ff\ ff\ ff\ 80\ 01_{16}$
- LDT \$X,\$Y,\$Z — \$X := $M_4[83] = ff\ ff\ ff\ ff\ ff\ 00\ 80\ 01_{16}$
- LDO \$X,\$Y,\$Z — \$X := $M_8[83] = ff\ 00\ 80\ 01\ 02\ 03\ 04\ 05_{16}$

- \$X := $M_x$[\$Y+\$Z] (vorzeichenerweitert)

# Schreiben in Speicher

| $a$ | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 |
|---|---|---|---|---|---|---|---|---|---|---|
| $M_1[a]$ | $00_{16}$ | $f0_{16}$ | $f1_{16}$ | $f2_{16}$ | $f3_{16}$ | $f4_{16}$ | $f5_{16}$ | $f6_{16}$ | $f7_{16}$ | $06_{16}$ |

- Sei $Y = 80, $Z = 3
- $X = $(f0f1f2f3f4f5f6f7)_{16}$
- STB $X,$Y,$Z — $M_1[83] = f7_{16}$
- STW $X,$Y,$Z — $M_2[83] = f6f7_{16}$
- STT $X,$Y,$Z — $M_4[83] = f4f5f6f7_{16}$
- STO $X,$Y,$Z — $M_8[83] = f0f1f2f3f4f5f6f7_{16}$

- $M_x[$Y+$Z] := $X (wird ggf. abgeschnitten)

# Exkurs: Little-Endian

| $a$ | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 |
|---|---|---|---|---|---|---|---|---|---|---|
| $M_1[a]$ | $00_{16}$ | $ff_{16}$ | $00_{16}$ | $80_{16}$ | $01_{16}$ | $02_{16}$ | $03_{16}$ | $04_{16}$ | $05_{16}$ | $06_{16}$ |

- MMIX ist *big-endian*
  - LDT $X,$Y,$Z ($Y+$Z=84)
  - $X := $(02030405)_{16}$

- Exkurs: little-endian
  - MOV r8, dword [84]
  - r8 := $(05040302)_{16}$

# Kontrollfluss

- Sprünge:
  - JMP *<relative address>*
  - GO *<absolute address>*
- Bedingter Sprünge:
  - B*<condition>* $X, *<relative address>*
- Bedingungen:
  - BN (negative), BNN (nonnegative)
  - BZ (zero), BNZ (nonzero)
  - BP (positive), BNP (nonpositive)
  - BOD (odd), BEV (even)

# Spezialanweisungen

- SWYM (sympathize with your machinery = NOP = „mache nichts“)
- GET / PUT (von/in Spezialregister lesen/schreiben)
- GETA (relative Adresse in Register laden)
- TRAP (Betriebssystemaufruf)
- GREG (Variablendefinition mit General Purpose Register)

# Assemblerprogrammierung

- Mnemonische Abkürzung für Befehle
- Abstraktionsniveau zwischen höherer Programmiersprache und reiner Maschinensprache
- Menschenlesbare Darstellung
- Von der Architektur des betreffenden Rechners abhängig
- Ein Assemblerprogramm wird durch einen *Assembler* in Maschinensprache transformiert

# Beispiel 0: ex0.mms

```
% Bilde den Mittelwert der natuerlichen Zahlen a,b,c

a     GREG   10           //Definiere globale Var. a, b, c
b     GREG   21           //$254 := 10; $253 := 21; $252 := 30
c     GREG   30

      LOC  #100          //Position des Programms im Speicher
Main  ADDU $0,a,b
      ADDU $0,$0,c
      DIV  $0,$0,3
Stop  TRAP 0,Halt,0      //Ende des Programms
```

# Trace von Beispiel 0

```
% mmixal ex0.mms
   [erzeugt Maschinenprogramm ex0.mmo]
% mmix -t1 ex0.mmo
        1. 0000000000000100: 2200fefd (ADDU) $0=1[0] = #a + #15 = #1f
        1. 0000000000000104: 220000fc (ADDU) $0=1[0] = #1f + #1e = #3d
        1. 0000000000000108: 1d000003 (DIVI) $0=1[0] = 61 / 3 = 20, rR=1
        1. 000000000000010c: 00000000 (TRAP) Halt(0)
  4 instructions, 0 mems, 67 oops; 0 good guesses, 0 bad
  (halted at location #000000000000010c)
```