



MMIX: Einführung

Einführung

- Aussprache: „em-micks“
- Autor: Donald Knuth

↳ T A O C P \rightsquigarrow MIX ———
↳ T E X \rightsquigarrow L A T_EX



—————→ MMIX

- MMIX ist ein virtueller Prozessor, mit eigener Programmiersprache
- MMIX-Programme sind „maschinennah“, d.h. eng an die Prozessorhardware angelehnt

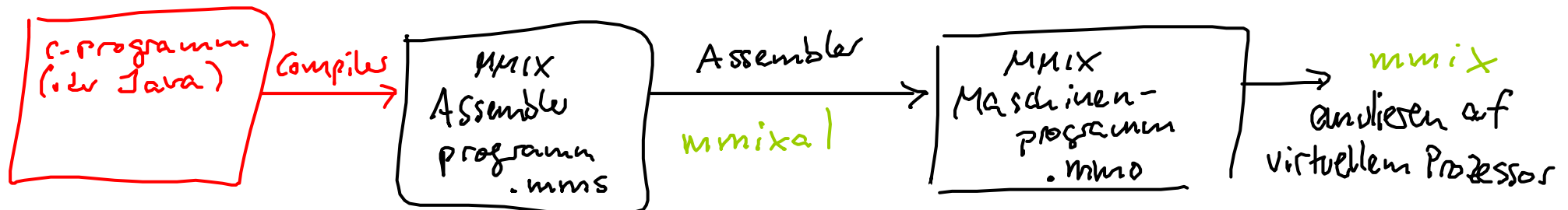
Ressourcen

- MMIX-Homepage von Knuth
<http://www-cs-faculty.stanford.edu/~knuth/mmix.html>
- MMIX-Seite zur Vorlesung 2003 von Daniel Keyzers
<http://www-i6.informatik.rwth-aachen.de/~keyzers/MMIX.html>



MMIX: Assembler und andere Tools

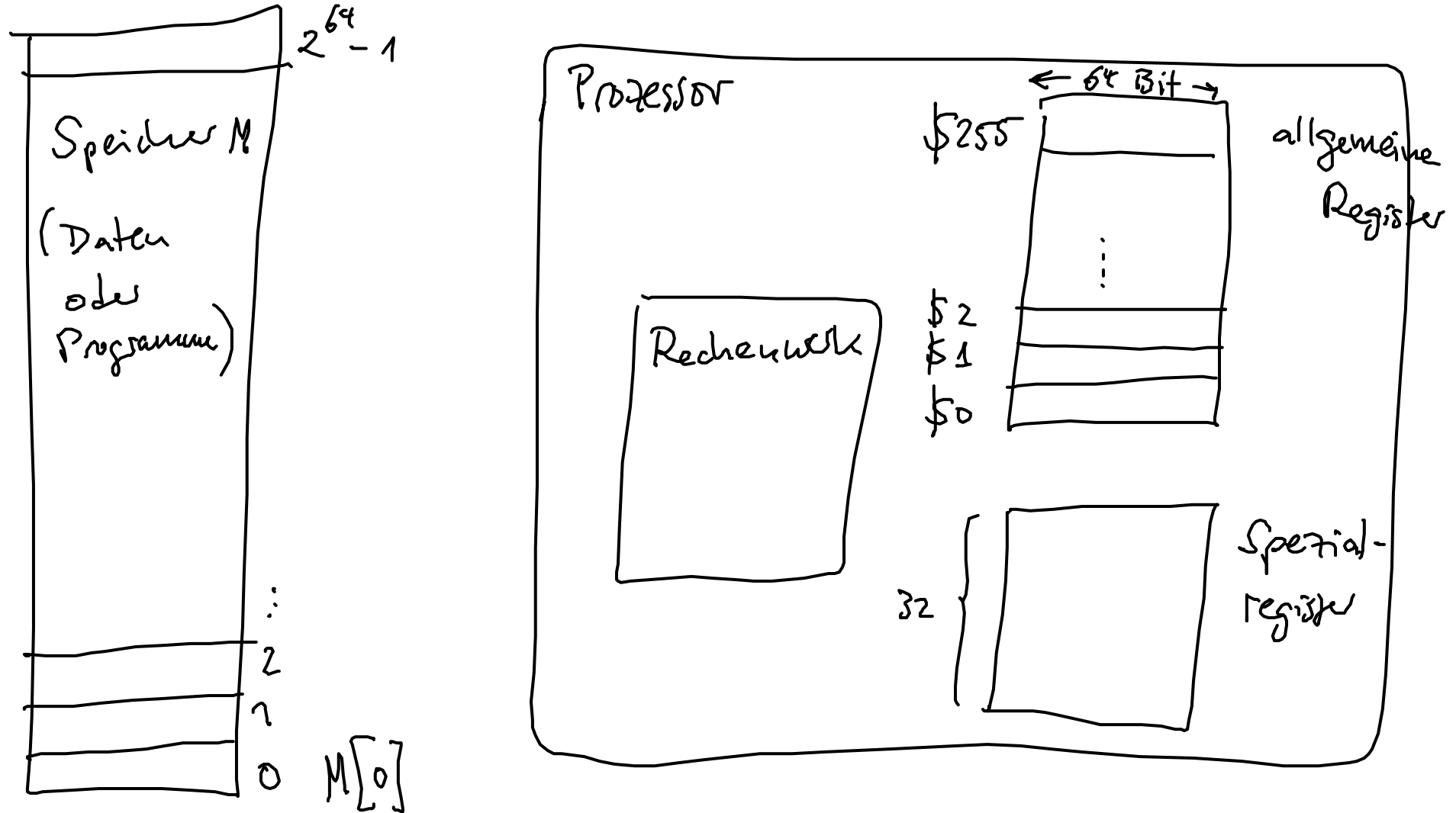
- MMIX-Programme sind nichts anderes als folgen von Bits
- Um MMIX-Programme zu schreiben, hat man eine (etwas) komfortablere Programmierumgebung/Programmiersprache (meist auch MMIX genannt)
- Programme in MMIX müssen in Bitfolgen übersetzt werden, damit sie auf dem Prozessor ausgeführt werden können („assembliert“)
 - Aufgabe des „Assemblers“
 - Maschinennahe Sprachen werden darum oft auch als Assemblersprache bezeichnet



- Um MMIX-Programme assemblieren und ausführen zu können, benötigt man:
 - Den Assembler **mmixal**, der eine Folge von Assemblerbefehlen in eine Folge von Maschinenbefehlen konvertiert (.mms-Datei -> .mmo-Datei)
 - Den Emulator **mmix**, der die Ausführung eines Maschinenprogramms (.mmo-Datei) auf MMIX emuliert, d.h. sich so verhält als ob das Programm auf MMIX ausgeführt würde.
- **mmixal** und **mmix** gibt es für alle gängigen Betriebssysteme



Programmiermodell von MMIX



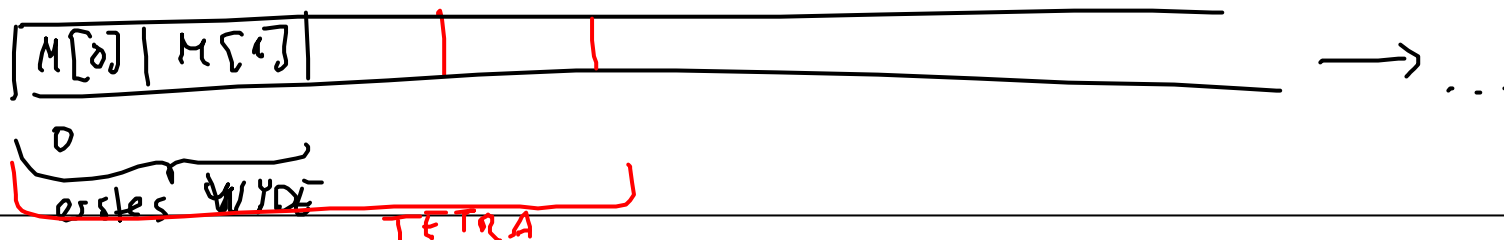


Register

- 256 Register à 64 Bit für allgemeine Aufgaben: $\$0, \dots, \255
- 32 reservierte Register für spezielle Aufgaben: $rA, \dots, rZ, rBB, rTT, rWW, rXX, rYY, rZZ$ (Divisionsrest, Überläufe, Operanden, Steuerungsbits, ...)

Virtueller Speicher

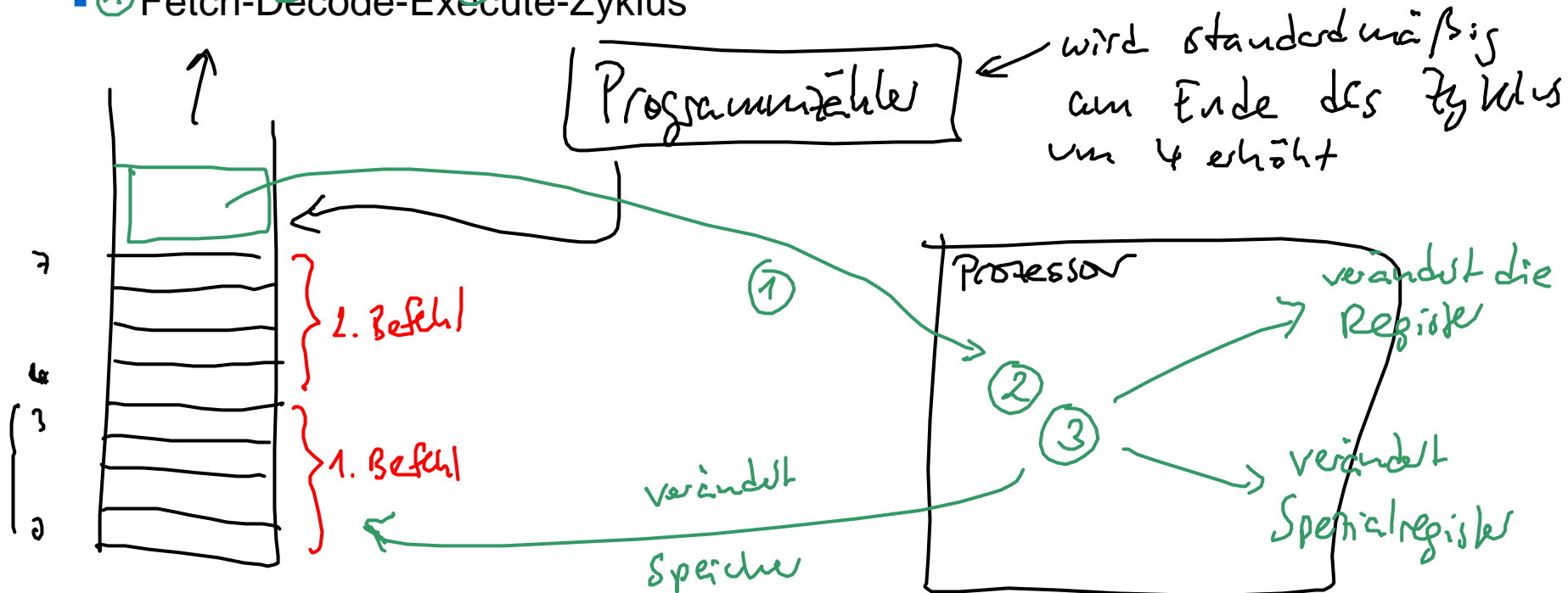
- 2^{64} Byte adressierbar durch $M[0], \dots, M[2^{64}-1]$
- Zugriff mit größerer Wortlänge:
 - Wydes (16 Bit): $M_2[0]=M_2[1], M_2[2]=M_2[3], \dots, M_2[2^{64}-2]=M_2[2^{64}-1]$
 - Tetras (32 Bit): $M_4[0]=M_4[1]=M_4[2]=M_4[3], \dots, M_4[2^{64}-4]=M_4[2^{64}-3]=M_4[2^{64}-2]=M_4[2^{64}-1]$
 - Octas (64 Bit): $M_8[0]=\dots=M_8[7], \dots, M_8[2^{64}-8]=\dots=M_8[2^{64}-1]$
- Kann Daten und Befehle enthalten





Befehlszyklus, Opcode, Argumente

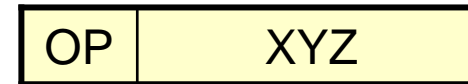
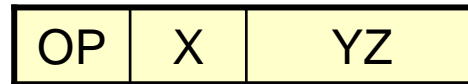
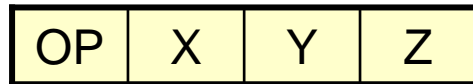
- Ein Maschinenprogramm ist eine Folge von Bits im Speicher
- Jeweils vier Byte ergeben einen Maschinenbefehl: erstes Byte ist der Befehlscode (Opcode)
- Prozessor enthält ein spezielles Programmzähler-Register
 - enthält die Adresse des nächsten auszuführenden Befehls
- ① Fetch-② Decode-③ Execute-Zyklus





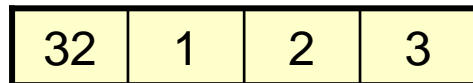
MMIX - Maschinenbefehle

Jeder Maschinenbefehl umfasst 4 Bytes

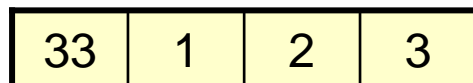


- OP: Operator
- X, Y, Z, YZ, XYZ: Operanden

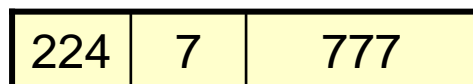
Beispiele



Addiert die Inhalte von Register 2 und 3 und legt das Ergebnis im Register 1 ab (ADD \$1,\$2,\$3)



Addiert 3 zum Inhalt von Register 2 und legt das Ergebnis in Register 1 ab (ADD \$1,\$2,3)



Setzt die höchstwertigen 16 Bit des Registers 7 auf 777 (SETH \$7,777)



Aufgabe des Assemblers mmixal:

Konvertierung eines Assemblerprogramms in ein Maschinenprogramm

Assemblerprogramm

- besteht aus einer Folge von Zeilen, die jeweils eine Anweisung enthalten
- Eine **Anweisung** besteht aus
 [Bezeichner] Operator [Operanden] (Bezeichner und evtl. auch Operanden sind optional)
getrennt durch ein oder mehrere Leerzeichen.
- Bezeichner: alle Zeichen bis zum ersten Leerzeichen
- Operator: erstes Nicht-Leerzeichen nach Bezeichner bis zum nächsten Leerzeichen
- Operanden: erstes Nicht-Leerzeichen nach Operator bis zum nächsten Leerzeichen
- alles Weitere einer Zeile wird ignoriert (Platz für Kommentare)



Beispiel eines einfachen Assemblerprogramms

	LOC	#100	Das Folgende ab Position #100 im Speicher ablegen
X1	BYTE	4	Ein Byte im Speicher mit 4 beschreiben
X2	WYDE	2	Nächstes Wyde mit 2 beschreiben
X3	OCTA	40	Nächstes Octa mit 40
List	BYTE	1,2,3	Die 3 folgenden Bytes mit den Werten 1,2,3 füllen
Y	IS	\$2	Y als Synonym für \$2 definieren
Main	LDB	\$1,X1	Wert von X1 aus Speicher in Register \$1
	LDW	Y,X2	usw.
	LDO	\$3,X3	
	MUL	\$3,Y,\$3	\$3 := \$2*\$3
	ADD	\$3,\$3,\$1	\$3 := \$3+\$1
	SUB	Y,\$1,Y	\$2 := \$1-\$2
	DIV	\$3,\$3,Y	\$3 := \$3/\$2
	TRAP	0,Halt,0	Programm beenden

Das Programm enthält

- Pseudo-Operatoren
- Bezeichner (Label)
- Assemblerbefehle
- Operanden und Kommentare



Konstanten

- Folgen von Dezimalziffern: 0,1,...,9
- Folgen von Hexadezimalziffern angeführt von einem #, z.B. #1, #C, #FF, #ABCDEF
- Zeichen in einfachen Anführungszeichen: 'a', 'b', 'C', '1', '"', ...
- Zeichenketten: z.B. "Hello" entspricht der Folge 'H','e', 'l', 'l', 'o'

Pseudo-Operatoren

entsprechen keinen Maschinenbefehlen, beeinflussen aber das Assemblieren

- *Label* **IS** *Ausdruck*
 - bindet den Bezeichner *Label* an *Ausdruck*, d.h. *Label* wird ein Synonym für *Ausdruck*
 - *Ausdruck* kann entweder eine Konstante oder eine Registerbezeichnung sein
- [*Label*] **LOC** *Ausdruck*
 - bindet den Bezeichner *Label* an die aktuelle Position im Speicher
 - die aktuelle Speicherposition wird auf *Ausdruck* verschoben
 - *Ausdruck* muss eine Konstante sein
 - dadurch Positionierung von Daten im Speicher möglich

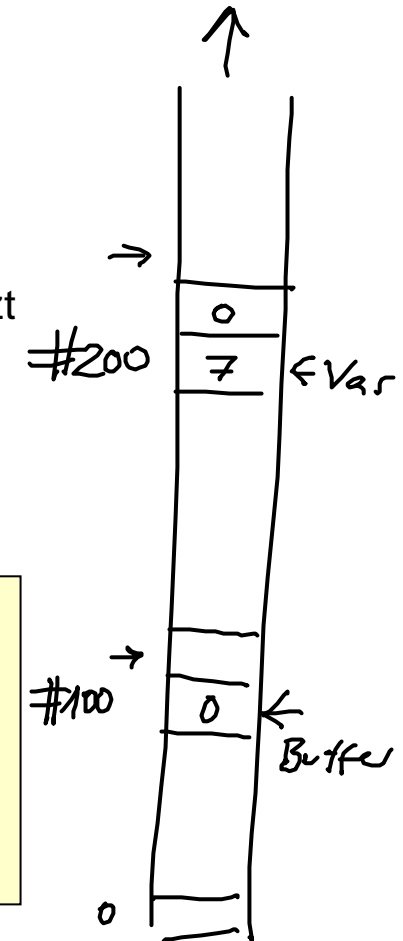


MMIX - Assembler

- **[Label] BYTE** *Ausdruck₁, Ausdruck₂, ..., Ausdruck_n*
 - bindet den Bezeichner *Label* an die aktuelle Speicherposition
 - füllt die folgenden Bytes mit den Werten von *Ausdruck₁* bis *Ausdruck_n*
 - erhöht die aktuelle Position um *n*
- **WYDE, TETRA, OCTA:**
 - aktuelle Speicherposition wird auf ein Vielfaches von 2, 4 bzw. 8 gesetzt
 - analoges Vorgehen zu **BYTE** mit größeren Werten
 - Präparierung des Speichers mit vordefinierten Werten

Beispiel: Anlegen eines 255 Byte großen Puffers

	LOC	#100	Das Folgende ab Position #100 im Speicher ablegen
BufSize	IS	255	Puffergröße auf 255 setzen
Buffer	BYTE	0	Anfang des Puffers
	LOC	Buffer+BufSize	255 Bytes überspringen
Var	WYDE	7	Weitere Variable anlegen
	...		
Main	LDA	\$1, Buffer	Programmstart
	...		



Die Programmausführung beginnt beim Bezeichner **Main**.



MMIX-Befehlsgruppen

- Operationen auf Registern
- Lade- und Speicherbefehle
- Ganzzahlarithmetik
- Bitweise logische Operationen
- Shift-Operationen und Setzen von Werten
- Vergleichsoperationen
- Bedingte Anweisungen
- Programmverzweigung
- Gleitkommaarithmetik
- Systemaufrufe (hier nur für Programmhalt und Ein-/Ausgabe)
- Hilfe für die Abarbeitung von Unterprogrammen



Operationen auf Registern

- **GET \$X,rx:**
Der Inhalt des Spezialregisters rx wird in Register \$X kopiert
- **GETA \$X,marke** (Get Address):
Die Speicheradresse von „marke“ wird in \$X abgelegt
- **PUT rx,\$X:**
Der Inhalt des Registers \$X wird in das Spezialregister rx kopiert
- **SET \$X,\$Y:**
Der Inhalt von \$Y wird nach \$X kopiert
- **SET \$X,Y:**
Der Inhalt des Registers \$X wird auf Y gesetzt

$X, Y, Z \in \{0, 1, \dots, 255\}$

$x \in \{A, B, \dots, Z, BB, TT, WW, XX, YY, ZZ\}$



Adressierung mit Registern

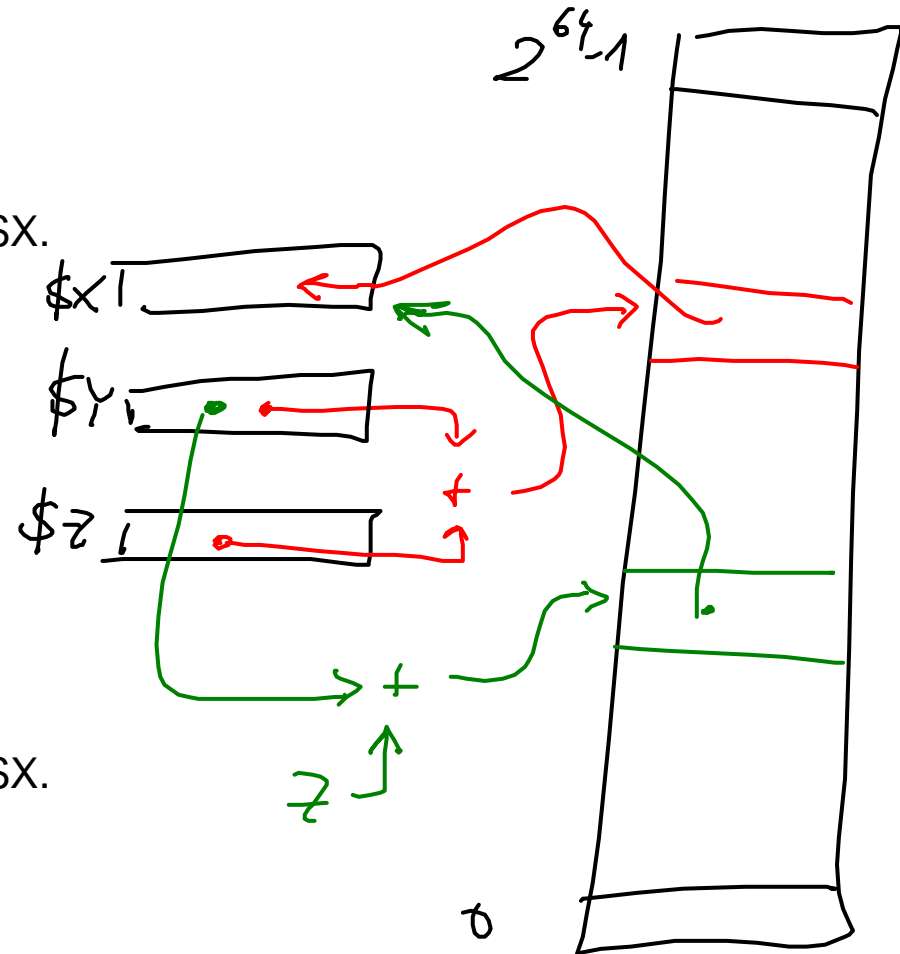
Beispielbefehl Load Byte (LDB):

- **LDB** \$X,\$Y,\$Z

Lädt den Inhalt von $M[\$Y+\$Z]$
vorzeichenbehaftete 8-Bit-Zahl nach \$X.

- **LDB** \$X,\$Y,Z

Lädt den Inhalt von $M[\$Y+Z]$ als
vorzeichenbehaftete 8-Bit-Zahl nach \$X.





MMIX - Befehlssatz

Laden: Speicher -> Register

- **LDB \$X,\$Y,\$Z** bzw. **LDB \$X,\$Y,Z** (Load Byte):
Lädt den Inhalt von $M[\$Y+\$Z]$ bzw. $M[\$Y+Z]$ als vorzeichenbehaftete 8-Bit-Zahl nach \$X.
- **LDBU** (Load Byte Unsigned):
analog, jedoch vorzeichenlos, d.h. als Zahl zwischen 0 und 255
- **LDW[U],LDT[U],LDO[U]** (Load Wyde, Load Tetra, Load Octa):
analog für $M_2[\$Y+\$Z]$, $M_4[\$Y+\$Z]$ und $M_8[\$Y+\$Z]$

Speichern: Register -> Speicher

- **STB \$X,\$Y,\$Z|Z** (Store Byte):
Das niederwertigste Byte des Registers \$X wird in $M[\$Y+\$Z]$ bzw. $M[\$Y+Z]$ gespeichert.
Ein Integer-Overflow wird signalisiert, falls $\$X \notin [-128,127]$
- **STBU** (Store Byte Unsigned):
analog, ohne Test auf Overflow
- **STW[U],STT[U],STO[U]** (Store Wyde, Store Tetra, Store Octa):
analog für Wyde, Tetra und Octa
- **STCO X,\$Y,\$Z|Z** (Store Constant Octabyte):
Das Byte X wird in $M_8[\$Y+\$Z]$ bzw. $M_8[\$Y+Z]$ gespeichert.



Addieren und Subtrahieren

- **ADD \$X,\$Y,\$Z|Z:**
berechnet $\$Y+\Z ($\$Y+Z$) unter Verwendung der 2-er Komplement-Arithmetik und legt das Ergebnis in $\$X$ ab (erzeugt einen Integer-Overflow, falls die Summe $\notin [-2^{63}, 2^{63}-1]$)
- **ADDU:**
berechnet $(\$Y+\$Z) \bmod 2^{64}$ ($(\$Y+Z) \bmod 2^{64}$) und legt das Ergebnis in $\$X$ ab
- **SUB \$X,\$Y,\$Z|Z:**
berechnet $\$Y-\Z ($\$Y-Z$) unter Verwendung vorzeichenbehafteter 2-er Komplement-Arithmetik und legt das Ergebnis in $\$X$ ab (erzeugt einen Integer-Overflow, falls die Differenz $\notin [-2^{63}, 2^{63}-1]$)
- **SUBU \$X,\$Y,\$Z|Z:**
berechnet $(\$Y-\$Z) \bmod 2^{64}$ ($(\$Y-Z) \bmod 2^{64}$) und legt das Ergebnis in $\$X$ ab
- **NEG \$X,Y,\$Z|Z:**
analog zu SUB wird $Y-\$Z$ ($Y-Z$) berechnet und in $\$X$ abgelegt
- **NEGU \$X,Y,\$Z|Z:**
analog zu SUBU wird $(Y-\$Z) \bmod 2^{64}$ ($(Y-Z) \bmod 2^{64}$) berechnet und in $\$X$ abgelegt



Multiplizieren und Dividieren

- **MUL \$X,\$Y,\$Z|Z:**
Vorzeichenbehaftetes Produkt von \$Y und \$Z (Z) wird in \$X abgelegt (erzeugt einen Integer-Overflow, falls die Summe $\notin [-2^{63}, 2^{63}-1]$)
- **MULU \$X,\$Y,\$Z|Z:**
Die niederwertigen 64 Bits des vorzeichenlosen Produktes von \$Y und \$Z (Z) werden in \$X und die höherwertigen 64 Bits im Spezialregister rH abgelegt.
- **DIV \$X,\$Y,\$Z|Z:**
Der vorzeichenbehaftete Quotient von \$Y und \$Z (Z) wird in \$X und der vorzeichenbehaftete Rest im Spezialregister rR abgelegt.
Behandlung von Spezialfällen (z.B. \$Z=0): siehe Manual
- **DIVU \$X,\$Y,\$Z|Z:**
Die vorzeichenlose 128-Bit-Zahl bestehend aus dem Spezialregister rD und \$Y wird durch \$Z (Z) dividiert und der Quotient in \$X und der Divisionsrest in rR abgelegt.



Bitweise Operationen: Logik

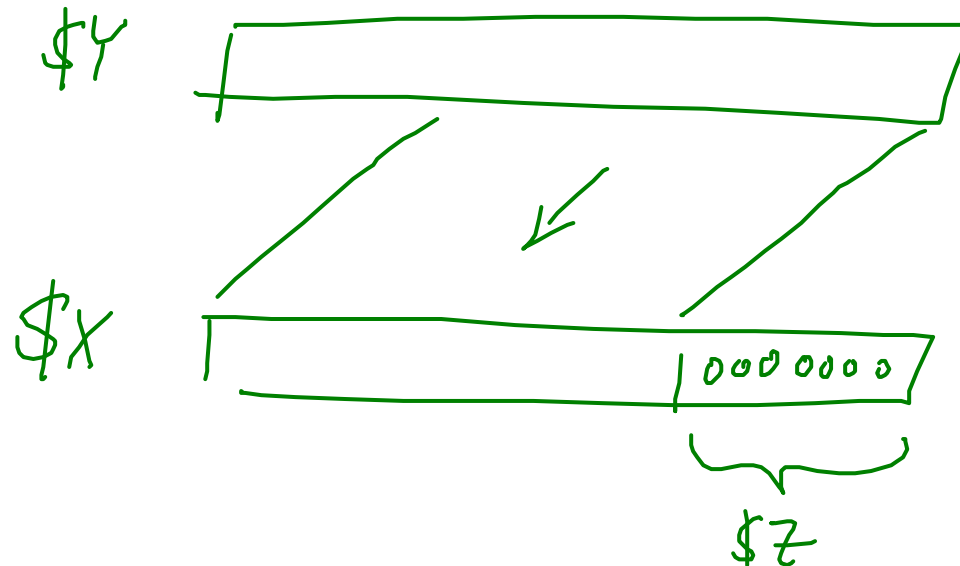
- **AND \$X,\$Y,\$Z|Z:** (analog **OR**, **XOR**)
\$X ergibt sich durch bitweise Und-Verknüpfung der Bits von \$Y und \$Z (Z):
 $\$x_i = \$y_i \wedge \$z_i$ ($\$x_i = \$y_i \wedge z_i$) für Bits $i = 0, \dots, 63$ der Operanden
- **ANDN \$X,\$Y,\$Z|Z:** (analog **ORN**)
analog mit $\$x_i = \$y_i \wedge \neg \$z_i$
- **NAND \$X,\$Y,\$Z|Z:**
 $\$x_i = \neg (\$y_i \wedge \$z_i)$
- **MUX \$X,\$Y,\$Z|Z:**
In Abhängigkeit des Spezialregisters rM werden die Bits von \$X auf \$Y oder \$Z (Z) gesetzt:
 $\$x_i = (\$y_i \wedge rM_i) \vee (\$z_i \wedge \neg rM_i)$
- Weitere, exotische Befehle (siehe Manual):
 - BDIF, WDIF, TDIF, ODIF
 - SADD, MOR, MXOR



MMIX - Befehlssatz

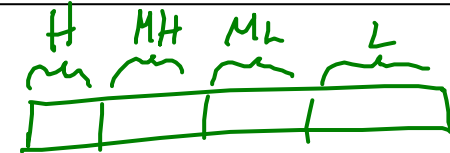
Shift-Operationen

- **SL \$X,\$Y,\$Z|Z** (Shift Left):
Links-Shift: $\$X := \$Y \cdot 2^{\$Z}$ ($\$X := \$Y \cdot 2^Z$) [$\$Y$ ist vorzeichenbehaftet, $\$Z$ (Z) vorzeichenlos]
Integer-Overflow, falls das Ergebnis $\notin [-2^{63}, 2^{63}-1]$
- **SLU** (Shift Left Unsigned): analog, wobei auch $\$Y$ als vorzeichenlos interpretiert wird
- **SR, SRU** (Shift Right): analog für Rechts-Shift





MMIX - Befehlssatz



Setzen von 16-Bit-Werten

- **SETH \$X, YZ** (Set to High Wyde):
Setzt die höchstwertigen 16 Bit von \$X auf YZ: $\$X := 2^{48} \cdot YZ$
- **SETMH, SETML, SETL** (Set to Medium High, Medium Low, Low):
analog mit $\$X := 2^{32} \cdot YZ$, $\$X := 2^{16} \cdot YZ$ und $\$X := 2^0 \cdot YZ$
- **INCH \$X, YZ** (**INCMH, INCML, INCL**) (Increase by High Wyde, Medium High, ...):
Die 16-Bit-Zahl YZ wird nach links geschiftet und zu \$X addiert (Überläufe werden ignoriert):
 $\$X := \$X + 2^{48} \cdot YZ$ ($\$X := \$X + 2^{32} \cdot YZ$, $\$X := \$X + 2^{16} \cdot YZ$, $\$X := \$X + 2^0 \cdot YZ$)
- **ORH \$X, YZ**: (**ORMH, ORML, ORCL**, **ANDNH**, **ANDNMH, ANDNML, ANDNL**)
Bitweise Oder-/NANDN-Verknüpfung mit der um 48,32,16,0 Bit nach links geschifteten Zahl YZ



Vergleiche

- **CMP** $\$X, \$Y, \$Z|Z$ (Compare):

$$\$X = \begin{cases} -1 & \text{falls } \$Y < \$Z \\ 0 & \text{falls } \$Y = \$Z \\ 1 & \text{falls } \$Y > \$Z \end{cases} \quad \text{unter Verwendung vorzeichenbehafteter Arithmetik}$$

- **CMPU**: analog mit vorzeichenloser Arithmetik

Bedingte Anweisungen

- **CSN** $\$X, \$Y, \$Z|Z$: (conditionally set if negative)
Wenn $\$Y$ negativ ist, wird $\$X := \Z ($\$X := Z$) ausgeführt, andernfalls nichts
- **CSZ, CSP, CSOD, CSNN, CSNZ, CSNP, CSEV**:
analog für „if zero“, „if positive“, „if odd“, „if nonnegative“, „if nonzero“, „if nonpositive“, „if even“
- **ZSN** $\$X, \$Y, \$Z|Z$: (zero or set if negative)
Wenn $\$Y$ negativ ist, wird $\$X := \Z ($\$X := Z$) ausgeführt, andernfalls $\$X := 0$
- **ZSZ, ZSP, ZSOD, ZSNN, ZSNZ, ZSNP, ZSEV**:
analog



Programm-Verzweigung

Normalerweise wird nach dem Befehl in $M_4[\lambda]$ der Befehl in $M_4[\lambda+4]$ ausgeführt.

- **BN \$X,marke**: (branch if negative)
Wenn \$X negativ ist, springe zu „marke“
- **BZ, BP, BOD, BNN, BNZ, BNP, BEV**:
analog (vgl. CS...)

Beispiel:

- **BP \$7,@+8**: wenn $\$7 > 0$, wird der nächste Befehl übersprungen
(@ beschreibt die Position des aktuellen Befehls)
- **JMP marke**:
Springe zu „marke“
- **GO \$X,\$Y,\$Z|Z**:
Springe zu $\$Y+\Z ($\$Y+Z$) und setze $\$X := \lambda+4$

Unterprogramm-Aufrufe (siehe Manual)

- **PUSHJ, PUSHGO**
- **POP**



Operationen mit Gleitkommazahlen (IEEE-754):

- **FADD \$X,\$Y,\$Z** (Floating Add):
Ergebnis der Addition der Gleitkommazahlen \$Y und \$Z wird in \$X abgelegt.
Diverse Spezialfälle möglich: siehe Manual
- **FSUB, FMUL, FDIV**:
analog für Subtraktion, Multiplikation und Division
- **FREM \$X,\$Y,\$Z** (Floating Remainder):
berechnet den Rest der Division $\$Y/\Z
($=\$Y - n*\Z , wobei n die ganze Zahl ist, die am nächsten an $\$Y/\Z liegt)
- **FSQRT \$X,\$Z** (Floating Square Root):
berechnet die Quadratwurzel von \$Z und legt das Ergebnis in \$X ab
(optional kann durch Y angegeben werden, wie gerundet werden soll)
- **FINT \$X,\$Z** (Floating Integer):
\$Z wird auf eine ganze Zahl gerundet und das Ergebnis in \$X abgelegt
- Vergleichsoperatoren **FCMP, FEQL, FUN, FCMPE, FEQLE, FUNE**:
siehe Manual



Konvertierung Gleitkommazahl <-> Ganzzahl

- **FIX \$X,\$Z:** (Convert Floating to Fixed)
Analog zu FINT wird eine Gleitkommazahl in eine Ganzzahl konvertiert. Mit FIX können jedoch Ausnahmesituation abgefangen werden.
- **FIXU \$X,\$Z:**
Analog zu FIX ohne dass Ausnahmesituationen auftreten.
- **FLOT \$X,\$Z|Z:** (Convert Fixed to Floating)
In \$X wird diejenige Gleitkommazahl abgelegt, die \$Z (Z) am nächsten ist. Falls Rundungen (gemäß Y) auftreten, wird eine „floating inexact exception“ ausgelöst.
- **FLOTU \$X,\$Z|Z:**
Analog zu FLOT, jedoch wird \$Z als vorzeichenlose Ganzzahl interpretiert

Rundungsparameter Y

- Mögliche Werte:
 - ROUND_OFF: Nachkommastellen abschneiden
 - ROUND_UP: In Richtung $+\infty$ runden
 - ROUND_DOWN: In Richtung $-\infty$ runden
 - ROUND_NEAR: Zur nächsten ganzen Zahl runden



Vordefinierte Systemaufrufe

- TRAP 0,Halt,0:
Programmende
- TRAP 0,Fputs,StdOut:
Ausgabe einer Zeichenkette auf der Standardausgabe
\$255 muss die Adresse der Zeichenkette enthalten
- TRAP 0,Fgets,StdIn:
Liest eine Zeile von der Standardeingabe
\$255 enthält Adresse und Größe des Eingabepuffers
- Weitere Systemaufrufe:
 - Fopen: Öffnet eine Datei
 - Fclose: Schließt eine Datei
 - Fread: Liest aus einer Datei
 - Fwrite: Schreibt in eine Datei

Greeting	BYTE	"Hello World",0
	LDA	\$255,Greeting
	TRAP	0,Fputs,StdOut

InSize	IS	100
InBuffer	BYTE	0
	LOC	InBuffer+InSize
InArgs	OCTA	InBuffer,InSize
	LDA	\$255,InArgs
	TRAP	0,Fgets,StdIn



Weiterführende Befehle

Befehle für den nächsten Exkurs:

- Unterprogrammaufrufe, Stack etc.
- Interrupts

Abschluss: Hello World!