Mitschrift
IT-Systeme SS2015
Prof. Dr. Martin Ruckert

M. Zell

13. Oktober 2015

Inhaltsverzeichnis

1 Zahlendarstellungen und Konventionen

Heute ging es um die Zahlendarstellung und Konventionen in MMIX. Ein Schwerpunkt war Umwandlung von Zahlen in unterschiedliche Zahlensysteme, insbesondere:

- Binär → Dezimal → Binär
- Binär → Hexadezimal → Binär

Diese Umwandlung ist hauptsächlich für Input und Output wichtig.

1.1 Binär → Hexadezimal

Eine Speicherzelle in MMIX (Octa) hat 64 Bit. Je 4 Bit (Nibble, halbes Byte) entsprechen einer hexadezimalen Ziffer. Die Umwandlung einer Speicherzelle - ein Byte pro [] - kann also direkt erfolgen: [.][.][.][10110011] → [.][.][.][B3] In MMIX kann man die Ziffern mittels SLU und AND #F extrahieren. Man kann auf verschiedene Arten ein Zeichen in ASCII-Code umwandeln:

Beispiel:
$1100_2 = C_{16}$
$1101_2 = D_{16}$

1.1.1 Mit einem Branch

Wert x	ASCII-Code
0	'0' + x
1	'0' + x
⋮	⋮
9	'0' + x
10	'A' - 10 + x
11	'B' - 10 + x
⋮	⋮
15	'F' - 10 + x

CMP mit 10
PBN → + '0'
sonst + ('A'-10)

1.1.2 Mit einer Tabelle

```
Tabelle BYTE "0123456789ABCDEF"
        LDA tmp,Tabelle
        LDBU $X,tmp,ziffer
```

1.1.3 Laufzeitvergleich

Mit der Tabelle werden ca. 4 Zyklen pro Ziffer benötigt, mit einem Branch $4 + \frac{10*1+5*3}{15}$ Zyklen.

1.2 Hexadezimal → Binär

Gegeben sind ASCII-Codes 0,...,9,A,...,F,a,...,f. Diese können ebenfalls mittels Tabelle umgewandelt werden oder mit CMP, Branch und anschließendem Zusammensetzen mittels SRU, OR (Laufzeit: 4 Zyklen pro Ziffer)

1.3 Dezimal → Binär

Jetzt soll "12345"von hinten nach vorne umgewandelt werden. Als erstes werden die Zeichen in eine Zahl umgewandelt: $Zeichen * 10^{Stelle}, Stelle \in Z$. Mit dem Hornerschema: sieht das für das Beispiel wie folgt aus: $(((1 * 10 + 2) * 10 + 3) * 10 + 4) * 10 + 5$

1.3.1 Pseudocode

```
Schleife:
ASCII Code lesen
'0' subtrahieren
Mit 10 multiplizieren
ASCII code lesen
'0' subtrahieren
Addieren ...
```

1.3.2 MMIX Code

Jetzt geht es darum in MMIX die Umwandlung von Dezimalzahlen in binäre Zahlen zu erledigen. Als erstes wandeln wir nur ein einziges Zeichen um:

```
        PREFIX :ToBinary:
str IS $10      % String mit Dezimalziffern
d IS $1
% Returnwert die entsprechende Zahl

% Version 1: nur ein Zeichen z.B. "5"
:ToBinary       LDBU d,str,0
        SUB d,d,'0'
        SET $0,d
        POP 1,0
```

Nun wandeln wir genau zwei Zeichen um:

```
% Version 2: genau zwei Zeichen z.B. "04" oder "15"
n IS $2
:ToBinary       LDBU d,str,0
        SUB d,d,'0'                     % 1. Stelle
        MUL n,d,10
        LDBU d,str,1
        SUB d,str,1
        SUB d,d'0'
        ADD n,n,d

        SET $0,n
        POP 1,0
```

Nun eines oder zwei. Der String endet mit einem Nullbyte!

```
% Version 3: Ein oder zwei Zeichen, String endet mit einem Null Byte
z IS $3
:ToBinary       LDBU d,str,0
```

```
        SUB z,d,'0'             % 1. Stelle

        LDBU d,str,1
        BZ d,single             % single digit

        MUL n,z,10
        SUB z,d,'0'
        ADD n,n,z

        SET $0,n
        POP 1,0

single:         SET $0,z
        POP 1,0
```

1. Optimierung. Eine Variable weniger. Das z brauchen wir nicht mehr.

```
% Version 3b: Ein oder zwei Zeichen, String endet mit einem Null Byte
str IS $0
d IS $1
n IS $2
:ToBinary       LDBU d,str,0
        SUB n,d,'0'             % 1. Stelle

        LDBU d,str,1
        BZ d,finish             % single digit

        MUL n,n,10
        SUB d,d,'0'
        ADD n,n,d

finish:         SET $0,z
        POP 1,0
```

Nun wandeln wir beliebig viele Zeichen eines Strings um.

```
% Version 4: mit beliebig vielen Ziffern; endet immer mit Nullbyte
str IS $0
d IS $1
n IS $2

:ToBinary       SET n,0

Loop:           LDBU d,str,0
        BZ d,finish
        ADD str,str,1
        MUL n,n,10
        SUB d,d,'0'
        ADD n,n,d
        JMP loop

finish:         SET $0,n
```

```
        POP 1,0
```

Indem wir das Programm umstellen können wir auf den JMP verzichten, was einen Zyklus einspart.

```
% Version 5: Optimierung der Schleife
% Laufzeit 15 k + 6 + 2
str IS $0
d IS $1
n IS $2

:ToBinary       SET n,0
        JMP 1F

Loop:           ADD str,str,1
        MUL n,n,10
        SUB d,d,'0'
        ADD n,n,d

1H                      LDBU d,str,0
        PBNZ d,Loop

finish:         SET $0,n
        POP 1,0
```

Nun ersetzen wir den zyklen-intensiven Befehl MUL (ca. 10 Zyklen)

```
% Version 6: Ersatz fuer MUL
% Laufzeit 7 k + 6 + 2
str IS $0
d IS $1
n IS $2

:ToBinary       SET n,0
        JMP 1F

Loop:           ADD str,str,1

        % Multiplikation mit 10
        4ADDU n,n,n
        SL n,n,1

        SUB d,d,'0'
        ADD n,n,d

1H                      LDBU d,str,0
        PBNZ d,Loop

finish:         SET $0,n
        POP 1,0
```

1.3.3 Exkurs xADDU-Befehle

Die 2ADDU/4ADDU/8ADDU/16ADDU-Befehle benutzen wir hier für eine schnelle Alternative zur Multiplikation mit MUL. Sonst kann man diese Befehle z.B. für die Adressierung: Index * 2/4/8/16 + Basis nutzen. Beispiel:

```
4ADDU n,n,n       n <-- 4n+n (Multiplikation mit 4)

2ADDU $X,$Y,$Z   $X <-- 2 * $Y + $Z
4ADDU $X,$Y,$Z   $X <-- 4 * $Y + $Z
8ADDU $X,$Y,$Z   $X <-- 8 * $Y + $Z
16ADDU $X,$Y,$Z $X <-- 16 * $Y + $Z
```

1.4 Binär → Dezimal mit Division

Nun geht es um die Umwandlung von Zahlen in Strings (z.B.12345 nach "12345"). Hier benötigen wir die Division. Diese ist in MMIX mit ca. 60 Zyklen sehr teuer. Dennoch führt sie zum Ziel.

```
DIV n,n,10 % teuer ca. 60 Zyklen
GET d,rR % rR Register ist mit dem Rest der letzten Division gefuellt
ADD d,d,'0'
```

1.5 Binär → Dezimal mit Multiplikation

Weil die Division teuer ist, kann man stattdessen multiplizieren. Die Idee ist, dass der Zahlenbereich eingeschränkt wird. In den oberen 32 Bit ist die Zahl die umgerechnet werden soll und in den unteren lauter Nullen. Die Zahl n ($n < 10^9$) wird ein einziges Mal geteilt ($n/10^9$). Dadurch wird erreicht, dass die Zahl im unteren 32-Bit-Block landet. Nun wird mit 10 Multipliziert. Dadurch gelangt die vorderste Ziffer im oberen 32-Bit-Block. Diese wird ausgegeben und anschließend mittels ADDMH #F gelöscht. Dieser Vorgang wird nun wiederholt. Die Laufzeit beträgt nun ca. 60 + 6K für k Ziffern statt 60 k.

```
[123456788].[00...0]

[0].[123456789]
*    10    =
[1].[23456789]
[0].[23456789]
*    10    =
[2].[3456789]
```

2 Synchronisierung von Threads

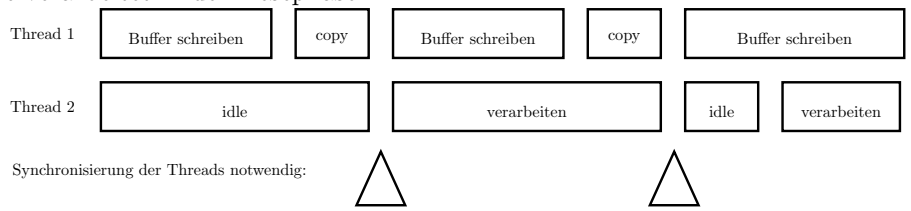
2.1 Vorteile und Eigenschaften

- gemeinsamer Speicher (Adressraum)
- getrennte Register
- getrennter Stack

2.2 Wiederholung

Der Wechsel zwischen Threads passiert mit SAVE (Alles kommt in den Stack; auf der obersten Position bleibt die Adresse der Daten) und UNSAVE (Nimmt die Adresse und holt die Daten vom Thread). Threadwechsel sind im Vergleich zu Prozesswechseln billig.

Problem: buffered IO. Der eine Thread speichert, liest und kopiert der andere verarbeitet in den Lesephasen:



2.3 Synchronisierung

2.3.1 Einfachster Fall:

Zwei Threads mit einem gemeinsamen Buffer:

Synchronisation mit einer Semaphore ("Licht", Ämpel")

***

Sequentiell inkonsistent: Zur Abhilfe gibt es den Befehl SYNC. SYNC 3 bis SYNC 7 sind **privileged** (Abb. 1).

SYNC	Bezeichnung
0	drain pipeline
1	finish all stores before starting following stores
2	finish all loads before starting following loads
3	beides; 1 und 2
4	Power Save Mode (z.B. bei Wartezyklen)
5	empty WriteBuffers and caches to memory
6	clear TLB (Translation Lookasside Buffers, d.h. Cache für page tables)
7	Clear all caches

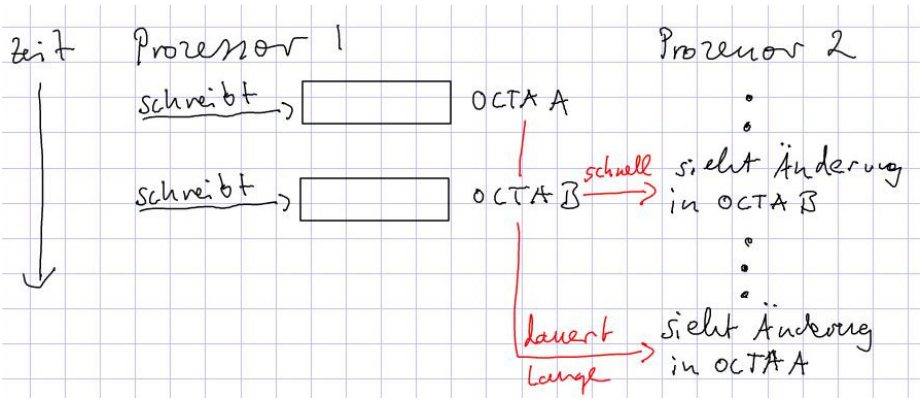


Abbildung 1: Beispiel zur sequentiellen Inkonsistenz

2.3.2 Double Buffering (Abb. 2) und Buffer Swapping (Abb. 3)

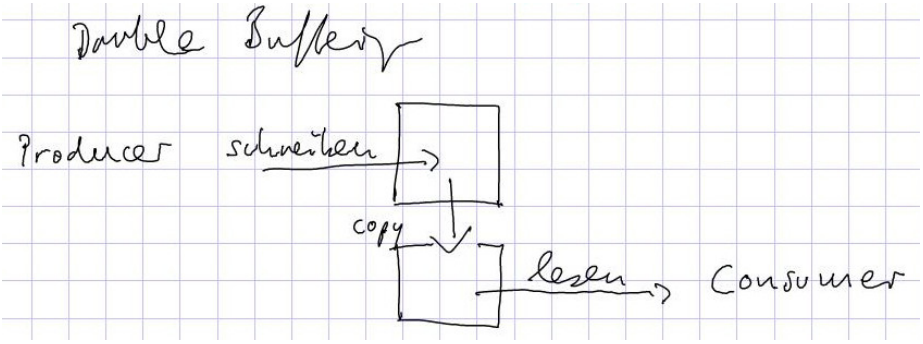


Abbildung 2: Double Buffering

```
        % Semaphore, Adresse von Buffer, Adresse von anderer Semaphore
S1      OCTA    1,Buffer1,S2
S2      OCTA    0,Buffer2,S1

Consumer:
tmp IS $0
buffer IS $1
S GREG 0

        LDA s,S1        Initialisieren

1H      LDO tmp,s,0
                BZ tmp,1B
                SYNC 2
                LDO buffer,s,8
                %.... Use Buffer/lesen
```

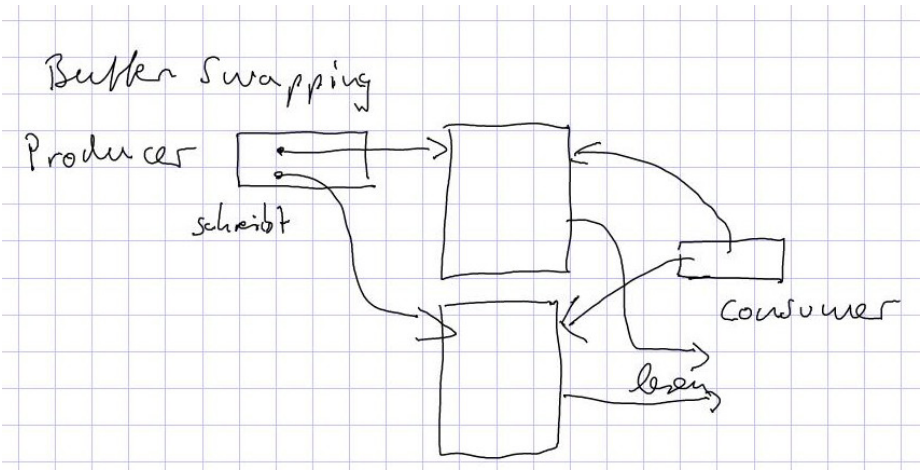


Abbildung 3: Buffer Swapping

```
        STCO  0,s,0       Release
        LDO   s,s,16      Buffer swap
```

Diese Synchronisation erfordert, dass der Thread, der den Speicher freigibt, bestimmt, wem der Speicher als nächstes gehört. Falls unklar ist welcher Thread als nächstes Zugriff braucht macht man folgendes. Dabei steht die 0 für „Ressource ist frei verfügbar“und 1 für „Ressource wird gerade verwendet“:

```
S       OCTA    0
```

Naive Lösung Eine naive Lösung könnte wie folgt aussehen, allerdings gibt es ein **Problem:** Beide wollen auf die Liste zugreifen. Es kann also passieren, dass beide gleichzeitig auf S zugreifen. Man braucht daher eine Operation die in einem Schritt (ununterbrechbar) ein OCTA liest, vergleicht und schreibt. Das ist bei MMIX die CSWAP $X,$Y,$Z Instruktion.

```
***
1H      LDO     $0,S    Warte bis S=0
        BNZ $0,1B
        SYNC 2
        STCO 1,S
        ...
        Use Release
        SYNC 1
        STCO 0, Release
```

2.4 Fortsetzung Synchronisation von Threads und Prozessen

```
CSWAP $X,$Y,$Z  % adresse = $Y+$Z
```

Vergleiche M[adresse] mit rP. Falls gleich:

rP: Prediction register

```
speichern: M[adresse] $ <-- $ $X
setze: $X <-- 1
```

sonst:

```
lade: rP <-- M[adresse]
setze: $x <-- 0
```

Die Instruktion ist atomar, d.h. **ununterbrechbar**.

Gemeinsame Daten

Abbildung 4: Gemeinsame Daten

```
Aquire: falls S == 0, setze S auf 1 (blocking)
        bearbeite buffer (protected code)

Release: setze S auf 0
```

Was nicht geht:

```
1H      LDO $0,S
        BNZ $0,1B
        STCO 1,S
```

mit CSWAP

```
1H      PUT rP,0
        SET $0,1
        CSWAP $0,S      % atomare Operation
        BZ $0,1B
```

2.5 Strategien zum Schreiben in den Cache

Write allocate Die passende cacheline wird geladen, der Wert kommt in die cacheline.

ganz viele Erklärungen von Herrn Ruckert, die v.a. parallel zum Abmalen des Tafelbilds nicht aufgeschrieben werdne können.

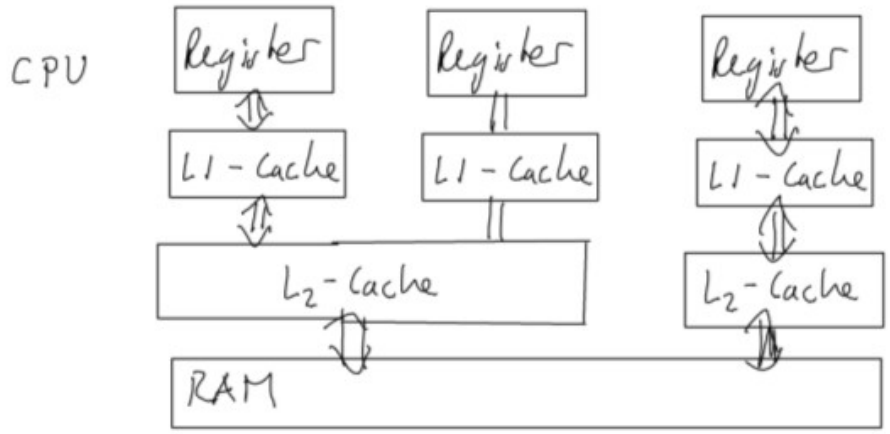


Abbildung 5: Speicherhirarchie in der CPU

Write Back Die passende cacheline wird geladen, geändert und geschrieben.

Write through Falls die passende cacheline vorhanden ist, wie write back, sonst wird der Wert am cache vorbei in die nächsttiefere Speicherebene geschrieben (vgl. Abb. 5).

2.6 Funktionsweise von Caches

Caches dienen dem schnellen Speicherzugriff. Meist sind sie direkt am Chip (L1-Caches). L2-Caches sind im gleichen Gehäuse der CPU. Es gibt sogenannte cachelines (z.B. 64 Byte). Es wird immer die komplette cacheline geladen oder gespeichert, nie einzelne Bytes (Abb. 6).

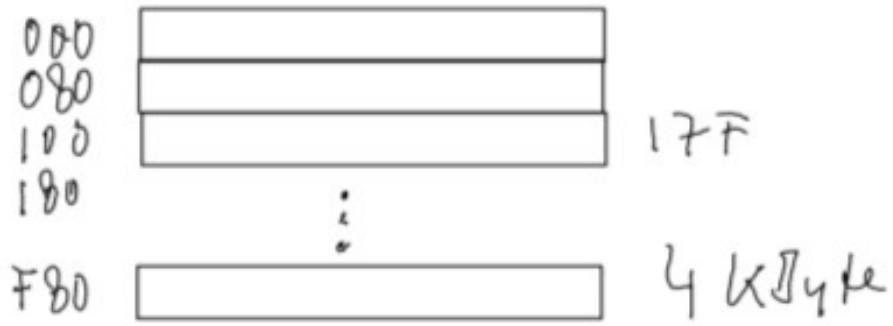


Abbildung 6: Cachelines

2.7 Non Blocking Synchronisation

wait-free: Jede Operation wird in endlich vielen Schritten fertig.

lock-free: Irgendeine Operation wird in endlich vielen Schritten fertig.

obstruction-free: Ein Thread wird in endlich vielen Schritten fertig, wenn alle anderen Threads gerade pausieren.

2.8 Cacheprotokolle MESI

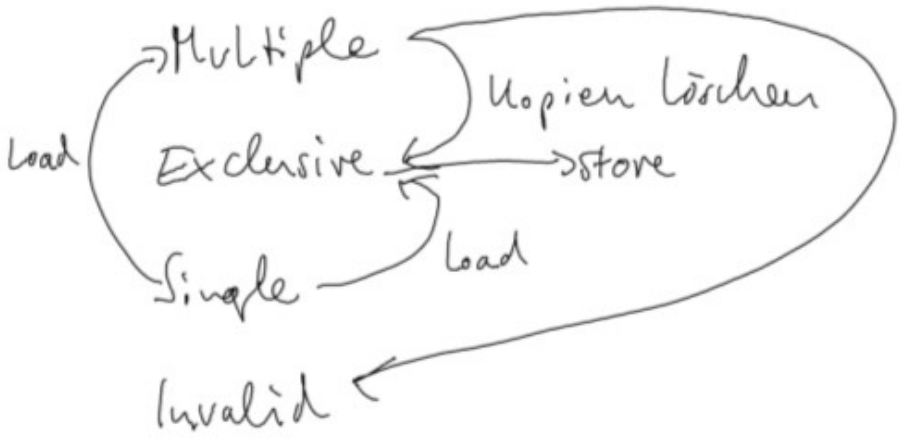


Abbildung 7: Funktionsweise von Cache MESI

3 Datenstrukturen

Bisher haben wir z.B. Arrays kennengelernt. Jetzt schauen wir uns dynamische Datenstrukturen an. Das sind Datenstrukturen, die zur Laufzeit die Größe ändern.

Dynamische Speicherverwaltung z.B. die Funktion malloc(size) in C. Diese Funktion gibt die Adresse zurück, an der size BYTE verfügbar sind. Die Funktion free(adresse) gibt den Speicher an der angegebenen Adresse wieder frei.

memmory allocate

3.1 Verkettete Listen

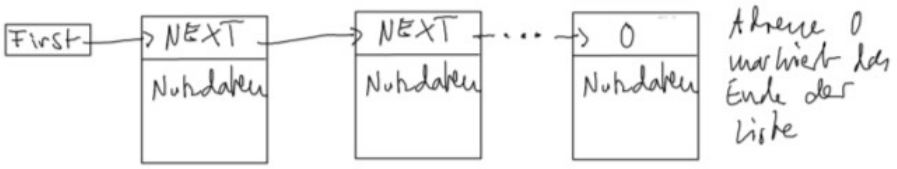


Abbildung 8: verkettete Listen

Insert: Verkettete Liste mit NEXT (Adresse des nächsten Knotens) und VALUE (Wert 1 OCTA).

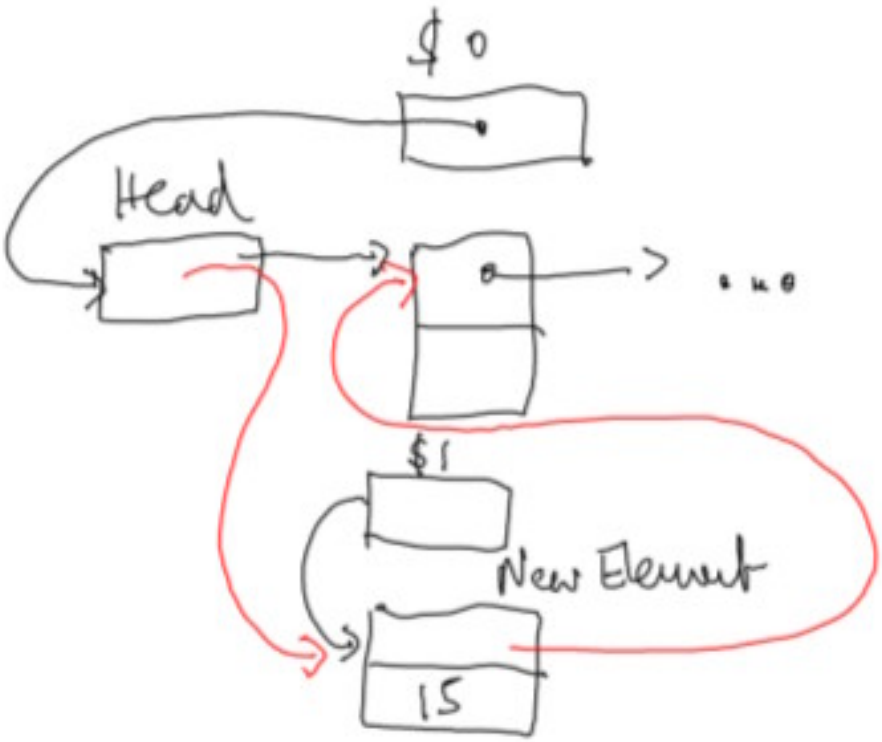


Abbildung 9: insert bei verketteten Listen

```
        PREFIX :Insert:
Head IS $0      % Adresse des list heads (param 1)
NewElement IS $1        % Adresse des neuen Elements (param 2)
tmp IS $2

:Insert
        LDOU tmp,Head,0
        STOU NewElement,Head,0
        STOU tmp,NewElement,0
        POP 0,0
```

Variante 2

```
        PREFIX :Insert:
Head IS $0      % Adresse des list heads (param 1)
NewElement IS $1        % Adresse des neuen Elements (param 2)
tmp IS $2

:Insert
        LDOU tmp,Head,0
        STOU tmp,NewElement,0
```

```
        STOU NewElement,Head,0

        POP 0,0


        PREFIX :Delete:
Head IS $0      % Adresse des list heads (@param 1)
% Rueckgabewert:
%       Null: wenn die Liste leer ist
%       sonst: Adresse des geloeschten Elements
return IS $1
tmp IS $2

:Delete LDOU return,Head,0
        BZ return,1F
        LDOU tmp,return,0
        STOU tmp,Head,0
1H      SET $0,return
        POP 1,0
```

4 Pipelineprozessoren (Superskalar)

4.1 Aufbau Pipelineprozessor

Bedingte Sprünge (Branch) stellen bei Pipelineprozessoren ein Problem dar. Dafür gibt es die sogenannte **Branch Prediction**, also das Raten ob der Branch genommen wird. In MMIX gibt es dazu Probable Branches (PBZ statt BZ) mit denen der Assembler auf einen sehr wahrscheinlichen Branch hingewiesen werden könne.

Abbildung 10: Aufbau Prozessor

4.2 Beispiel: Quicksort

Am besten in Wikipedia nachlesen.

Man hat ein Array $\{7, 3, 9, 2, 8, 6, 5, 7, 9, 1, 4\}$ von Daten, die man sortieren will. Die Idee ist, dass man partitioniert, also links die kleinen und rechts die großen Werte sortiert. Rekursiv werden dann die kleinen Werte und die großen Werte sortiert. Um zu entscheiden was groß und was klein ist verwendet man das sogenannte **Median of Three (Mittlere von drei Werten)**. Im Beispiel ist dies die Zahl 6.

Sortieren der drei Elemente $\{7, 3, 9, 2, 8, 6, 5, 7, 9, 1, 4\} \rightarrow \{4, 6, 9, 2, 8, 3, 5, 7, 9, 1, 7\}$
Diesen Schmarrn macht man solange bis das Array so aussieht: $\{1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 9\}$

```
1H      add i <-- i+1
start: load Ki  Lade Kex i
       vergleiche Ki < K
       if Ki < K goto 1H (oben)

2H      sub j <-- j-i
       load Kj
       verleiche K < Kj
       if K < Kj goto goto 2H (oben)

       if i < j tausche Ki und Kj und goto 1H

       sonst fertig
```

5 Verwaltung des dynamischen Speichers zur Laufzeit

Bei Objekten gibt es Konstruktoren und Destruktoren. In Sprachen wie C gibt es malloc, free. Im Assembler gibt es meistens ein Pool Segment (Intel: eXtra Segment). Hier wird ein sogenannter Heap eingerichtet, der Speicher reserviert und freigibt.
Wird ein Programm mit Argumenten gestartet stehen in $0 die Anzahl der Argumente und in $1 4...8 die Adresse des ersten Arguments. Die Werte der Argumente selbst stehen im Pool Segment 0x4000...0.

5.1 Im Poolsegment dynmaisch Speicher anfordern

Die Funktion **malloc** soll erstellt werden (Parameter: Anzahl BYTE, Rückgabewert: Adresse wo diese BYTE verfügbar sind.):

- Rückgabe wert = erstes OCTA im Poolsegment.
- Anzahl BYTES auf erstes OCTA aufaddieren
- Anzahl BYTES vorher auf ein Vielfaches von acht aufrunden (wegen Alignment).

```
          LOC  #100
tmp       IS   $0
Main      SET  tmp+1,22
          PUSHJ tmp,Malloc
          STCO 22,tmp
          SET  tmp+1,8
          PUSHJ tmp,Malloc
          STCO 8,tmp,0
          TRAP 0,Halt,0

          PREFIX :Malloc:          % Version 1
size      IS   $0
base      IS   $1
free      IS   $2
tmp             IS   $3
:Malloc   SETH base,#4000
          LDOU free,base,0
          ADD  size,size,7
          ANDN size,size,7
          ADDU tmp,free,size
          STOU tmp,base,0
          SET  $0,free
          POP  1,0
```

5.2 Beispielanwendung

Wir speichern Nodes die aus zwei OCTA bestehen, einem Zeiger NEXT und einem Wert VALUE. Wir wollen zwei Funktionen New und Old haben. **Old**

hat einen Parameter (Knoten) und fügt den Knoten der Liste mit alten Knoten hinzu. **New** prüft, ob alte Knoten vorhanden sind und wenn ja, ob diese genutzt werden können. Sonst nutzt man Malloc und reserviert sich so neuen Speicher für ein neuen Knoten (Node).

```
          LOC  Data_Segment
          GREG    @
OldNodes  OCTA    0       % derzeit leer

          LOC  #100
tmp       IS      $0
NEXT      IS      0
VALUE     IS      8
Main      PUSHJ   tmp,New
          STCO    22,tmp,VALUE

          PUSHJ   tmp,New
          STCO    8,tmp,VALUE

          SET     tmp+1,tmp
          PUSHJ   tmp,Old

          PUSHJ   tmp,New
          STCO    200,tmp,VALUE

          TRAP    0,Halt,0        % Ende Gelaende

          PREFIX  :Malloc:        % Version 1
size      IS      $0
base      IS      $1
free      IS      $2
tmp               IS      $3
:Malloc   SETH    base,#4000
          LDOU    free,base,0
          ADD     size,size,7
          ANDN    size,size,7
          ADDU    tmp,free,size
          STOU    tmp,base,0
          SET     $0,free
          POP     1,0

          PREFIX  :Old:
Node      IS      $0      % Parameter
First     IS      $1

:Old      LDOU    First,:OldNodes % laedt Zeiger auf alten Knoten
          STOU    First,Node,:NEXT
          STOU    Node,:OldNodes
          POP     0,0
```

```
          PREFIX  :New:
rJ        IS      $0
First     IS      $1
tmp       IS      $2
:New      LDOU    First,:OldNodes
          BZ      First,1F          % nichts drin

          LDOU    tmp,First,:NEXT % laden des Nextpointers vom 1. Knoten
          STOU    tmp,:OldNodes   % speichern
          SET     $0,First
          POP     1,0

1H        GET     rJ,:rJ
          SET     tmp+1,16          % Parameter Groesse = 10 uebergeben
          PUSHJ   tmp,:Malloc
          PUT     :rJ,rJ
          SET     $0,tmp
          POP     1,0
```

5.3 Verschiedene Funktionen auf verketteten Listen

5.3.1 Einfügen am Anfang

Haben wir bereits betrachtet.

5.3.2 Löschen am Anfang

Haben wir bereits betrachtet.

5.3.3 Einfügen am Ende

```
:insertEnde LDOU          last,Head,0
0H        LDOU    tmp,last,:NEXT
          BZ      tmp,1F

          SET     last,tmp
          JMP     0B

1H        STOU    Node,Last,:NEXT
          POP     0,0
```

```
:insertEnde SET last,Head
0H        LDOU    tmp,last,:NEXT
          BZ      tmp,1F

          SET     last,tmp
          JMP     0B
```

```
1H      STOU    Node,Last,:NEXT
        POP     0,0


:insertEnde     LDOU    tmp,Head,:NEXT
        BZ      tmp,1F

        SET     Head,tmp
        JMP     :InsertEnde

1H      STOU    Node,Head,:NEXT
        POP     0,0


Nodes   IS      $0
Head    IS      $1
tmp     IS      $2

:InsertEnd      LDOU    tmp,Head,:NEXT
        BZ      tmp,1F
        SET     Head,tmp
        JMP     :InsertEnd
1H      STOU    Node,Head,:NEXT
        POP     0,0


Nodes   IS      $0
Head    IS      $1
tmp     IS      $2
1H      SET     Head,tmp
:InsertEnd      LDOU    tmp,Head,:NEXT
        PBNZ    tmp,1B
        STOU    Node,Head,:NEXT
        POP     0,0


Nodes   IS      $0
Head    IS      $1
tmp     IS      $2
1H      SET     Head,tmp
:InsertEnd      LDOU    tmp,Head,:NEXT
        PBNZ    tmp,1B
        STOU    Node,Head,:NEXT
        POP     0,0
```

6 Fragen von Studenten. Antworten vom Professor

6.1 Wie funktioniert loop unrolling?

```
        PREFIX :InsertEnd:
new     IS $0    Adresse des neuen Knoten
head    IS $1    Adresse von der Adresse des ersten Knotens
next    IS $2
NEXT    IS 0     Offset des NEXT-Feldes

% Variante 1
1H      SET head,next
:InsertEnd      LDOU    next,head,NEXT
        BNZ     next,1B
        STOU    new,head,NEXT
        POP     0,0

% Variante 2
1H      SET head,next
:InsertEnd      LDOU    next,head,NEXT
        BZ      next,0F
        SET     head,next
        LDOU    next,head,NEXT
        BNZ     next,1B
        STOU    new,head,NEXT
        POP     0,0

0H      STOU    new,head,NEXT
```

copy propagation Statt der Kopie verwendet man das Original. Der optimierte Code lautet dann:

```
:InsertEnd LDOU next,head,NEXT
        BZ      next,0F

        LDOU    head,next,NEXT
        PBNZ    head,:InsertEnd
        STOU    new,next,NEXT
        POP     0,0

0H      STOU    new,head,NEXT
        POP     0,0
```

6.2 Wie kann man die Anzahl von Bytes auf ein Vielfaches von 8 aufrunden?

6.2.1 Wie geht das Dezimal?

- Abrunden: $1234567 \rightarrow 1234000$

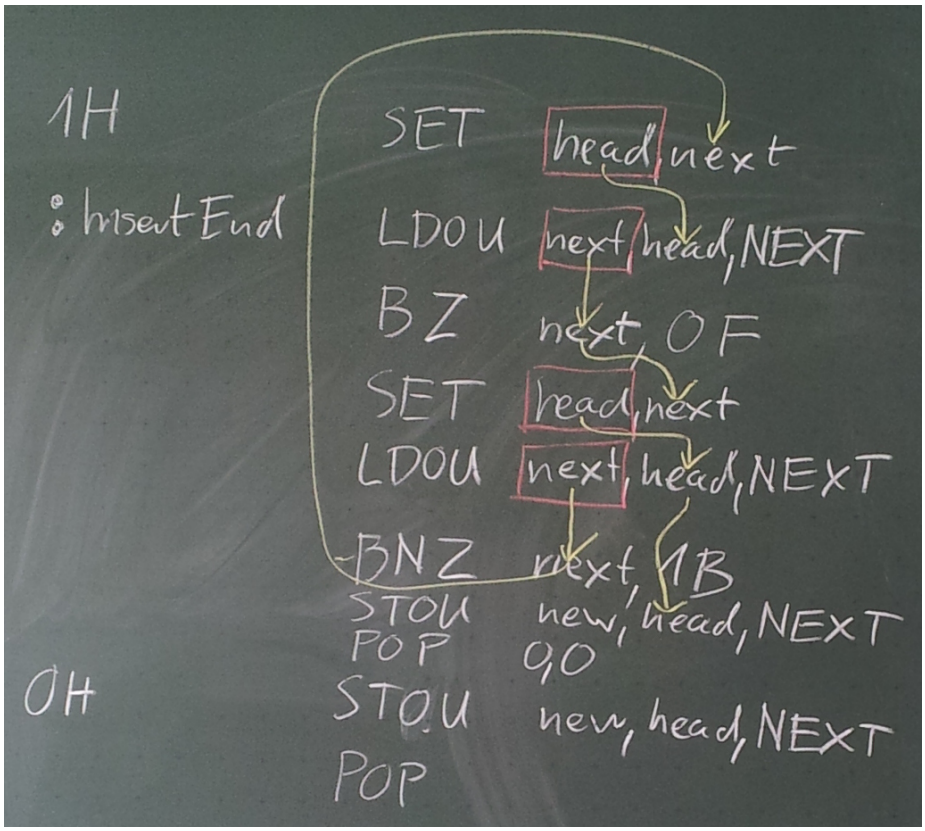


Abbildung 11: Optimierung mittels loop doubleing; Rot: neuer Wert, Gelb: Verwendung

- Aufrunden: $1234567 \rightarrow 12345000$
- Aufrunden durch Abrunden: Erst $+999$, dann abrunden: $1234557 + 999 = 12345566 \rightarrow 12345000$

6.2.2 Wie geht das binär?

Abrunden auf Vielfaches von 8 (Abb. 12)

```
AND     size,size,7
(AND    size,size,#FFF....F8 % wg. Groesse nicht moeglich)
ANDN    size,size,7 % Ersatz fuer Zeile oben drueber
```

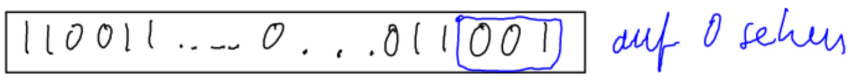


Abbildung 12: Abrunden auf Vielfaches von 8 (binär)

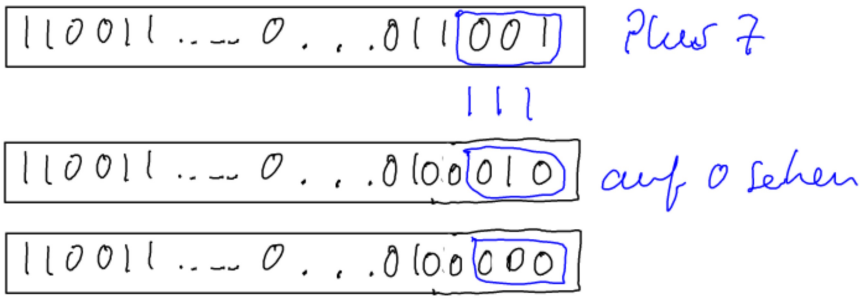


Abbildung 13: Aufrunden auf Vielfaches von 8 (binär)

Aufrunden auf Vielfaches von 8 (Abb. 13)

7 Verbesserung der Freispeicherverwaltung

7.1 ganz einfach

- malloc(n) nimmt die nächsten n Byte ($(n+7)8 \sim 7$) vom Poolsegment.
- free(p) wird nicht implementiert

7.2 einfach für eine feste Größe N

Verwalten einer Liste mit freien Blöcken.

malloc(N)

- wenn Freiliste leer N Bytes vom Poolsegment
- sonst erstes Element der Freiliste

free(p) Node p in die Freiliste einfügen.
Das ist einfach für eine Variable n. Die Knoten sehen so aus: Abb. 14

malloc(n): n wir auf ein Vielfaches von 8 aufgerundet. In der Freiliste wird nach einem **passenden** Element gesucht. Wenn nicht, wird aus dem Poolsegment n+8 Byte (8 entspricht der Größe size=)genommen. Size wird eingetragen und Adresse vom OCTA nach size zurückgegeben.

Was heißt passend? Es gibt unterschiedliche Strategien.

- First Fit: den ersten der passt
- Best Fit: den der als bester passt

Best Fit und First Fit Beide Strategien können mit dem Teilen von Knoten kombiniert werden (Abb. 15). Teilen in der Regel nur, wenn der Rest eine gewisse Mindestgröße hat. Problem: Speicherfragmentierung.

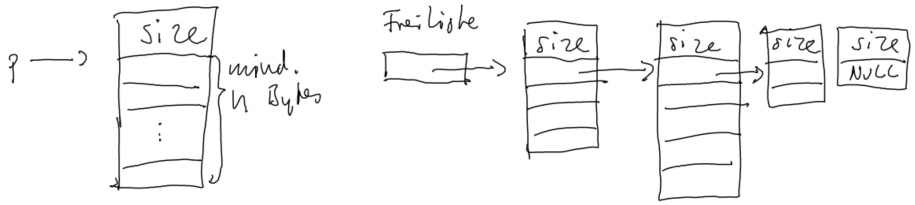


Abbildung 14: So könnte die Freiliste aussehen

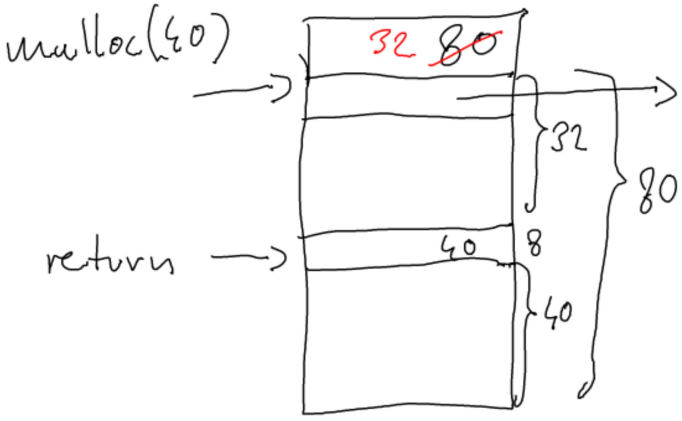


Abbildung 15: geteilte Knoten

Beispiel: Knoten zwischen 16 und 1600 Byte zufällig gemischt, First Fit, Anforderung von 64 Byte.

7.3 Verbesserungen

- Durchsuchen der Freiliste nicht jedes Mal von vorne, sondern von der letzten Fundstelle aus.
- Zusammenfügen von Knoten beim Freigeben. Nachteil: Durchsuchen der Liste dauert ggf. lange. Man kann das verkürzen, indem man die Liste nach Adressen sortiert.
- Garbage Collection

  Es erfolgt keine Freigabe

  stattdessen gibt es ein Verzeichnis aller Zeiger (auf dem Stack, z.B. in Java).

  Hier gibt es einen Mark and Sweep Algorithmus. Man beginnt mit den Zeigern auf dem Stack und markiert sie , dann markiert man alle Objekte, auf die sie zeigen und alle Zeiger in den Objekten und so weiter (Mark). Nun wird der Speicher durchlaufen und nicht-markierte OElemente freigegeben (Sweep).