Dieses Dokument ist eine kurze und oberflächliche Einführung in die Programmiersprache des MMIX-Simulators, der im Rahmen des Kurses 'Einführung in die Technische Informatik' an der RWTH Aachen behandelt wird. Gemacht für die, die programmieren können und sich praktisch mit MMIX auseinandersetzen wollen, aber verständlicherweise keine Lust dazu haben, sich durch die offizielle Dokumentation zu kämpfen.

# 0 Software-Setup

Für das Programmieren mit MMIX werden grundsätzlich zwei getrennte Programme benötigt:

1. MMIX-Assembler (`mmixal`): Nimmt das vom Programmierer geschriebene Programm als `.mms`-Datei und produziert eine kompilierte `.mmo`-Datei.
2. MMIX-Simulator (`mmix`): Nimmt die `.mmo`-Datei und simuliert die Ausführung des Programms.

Beide können unter mmix.cs.hm.edu/exe (Windows) bzw. mmix.cs.hm.edu/bin (Linux) heruntergeladen werden. Für die Ausführung von Beispielen in diesem Dokument sollte davon ausgegangen werden, dass sich `mmix`, `mmixal` und die `.mms`/`.mmo`-Dateien in einem gemeinsamen Ordner befinden.

# 1 Register

Im virtuellen MMIX-Chip stehen dem Benutzer 256 Register, also Speicherzellen, zur Verfügung. Die Namen der Register folgen dem Format `$n` und beginnen bei 0: `$0`, `$1`, `$2`, ..., `$253`, `$254`, `$255`. Der Name ist dabei immer als ein fester Text zu verstehen: Man kann weder im Programmverlauf einen Namen durch einen anderen ersetzen, um auf einem anderen Register zu arbeiten, noch lässt sich ein Name auf eine andere Zelle umlenken.

Der Inhalt eines Registers besteht aus genau 64 Bits, also 8 Bytes. Im Kontext der MMIX-Programmierung heißen 2 Bytes ein *Wyde*, 4 Bytes bzw. 2 Wydes heißen ein *Tetra*, und 8 Bytes / 4 Wydes / 2 Tetras heißen ein *Octa*. Ein Register ist dementsprechend ein Octa groß. Wird einer Anweisung ein Register gegeben, so ist die Interpretation des Inhaltes der Anweisung überlassen. Eine bestimmte Anweisung versteht ein Register entweder als:

1. Einen Ort, an den das Ergebnis zu schreiben ist,
2. oder als den enthaltenen Wert in Form eines vorzeichenlosen Integers,
3. oder als den enthaltenen Wert in Form eines Integers im Zweier-Komplement,
4. oder als den enthaltenen Wert in Form eines IEEE-Floats.

Neben den numerierten Registern gibt es noch zusätzliche Register, auf die Operatoren ggf. Nebenergebnisse schreiben können. Diese haben dann Namen des Formats `$r..`, z.B. `$rR` für den Rest einer Division.

# 2 Programmstruktur und Anweisungen

Ein MMIX-Programm besteht grundsätzlich aus zwei Teilen: Anweisungen für den MMIX-Assembler (genannt Assembler Directives, auch Pseudo-Operationen) und Anweisungen für MMIX selber (genannt Instructions.)

Gültige Pseudo-Operationen sind:

- `IS`: Die Anweisung `name IS expr` gibt vor, dass `name` restlichen Programm durch `expr` ersetzt werden soll.
- `GREG`: Die Anweisung `name GREG value` sucht sich ein freies Register aus, setzt dessen Wert auf `value` und gibt vor, dass `name` auf das Register verweist. Der normale Name für das belegte Register kann dann nicht mehr benutzt werden. Der Inhalt des Registers kann nicht verändert werden, wenn er nicht anfangs auf 0 gesetzt wurde!

- `LOC`: Die Anweisung `LOC #N` gibt dem Assembler vor, dass er die nachfolgenden Instructions an die `N`-te Adresse der Zieldatei zu schreiben hat. In der Praxis markiert `LOC` daher den Beginn des wirklichen Programmes, und damit MMIX das Programm richtig lesen kann, wird an die 100. Adresse geschrieben: `LOC #100`

Eine Instruction hat die Struktur `Label OP a,b,c`. Das Label kann leer sein, und je nach Typ des Instructions entfällt das letzte `,c`.

Der Teil einer Anweisung, der vollständig groß geschrieben wird, heißt Opcode (für Operation Code). Vor dem Opcode hat immer mindestens ein Leerzeichen zu stehen, unabhängig davon, ob das Label leer ist; ansonsten wird das Programm nicht kompiliert. Außerdem wird für die Lesbarkeit und Ästhetik wärmstens empfohlen, den Code so einzurücken, dass eine Art Tabelle entsteht:

| Ungültiger Code | Gültiger Code | Guter Code | Besserer Code |
|---|---|---|---|
| `Main OPA $0,$1,$2`<br>`OPB $0,$1,$2`<br>`OPCD $0,$1,$2` | `Main OPA $0,$1,$2`<br>`OPB $0,$1,$2`<br>`OPCD $0,$1,$2` | `Main OPA $0,$1,$2`<br>`OPB $0,$1,$2`<br>`OPCD $0,$1,$2` | `Main OPA $0,$1,$2`<br>`OPB $0,$1,$2`<br>`OPCD $0,$1,$2` |

In den allermeisten Fällen errechnet ein Instruction aus den Eingaben `b` und `c` einen Wert, der in das Register `a` geschrieben wird. Zum Beispiel weist die Instruction `AND $0,$1,$2` dem Register `$0` das Ergebnis der Addition der Inhalte von `$1` und `$2` zu.

Hier ein Beispiel für ein einfaches arithmetisches Programm:

```
1 a     GREG 15        // Wir weisen a ein anonymes Register von Wert 15 zu.
2 b     GREG 93        // Wir weisen b ein anonymes Register von Wert 93 zu.
3       LOC  #100
4 Main  ADD  $1,a,b    // a und b werden in $1 addiert.
5       DIV  $0,$1,2   // $1 wird durch 2 geteilt und in $0 geschrieben.
6 Stop  TRAP 0,Halt,0  // Das Programm wird erfolgreich beendet.
```

Die letzte Zeile ist Boilerplate, aber muss am Ende von jedem Programm stehen. Eine Liste der wichtigen Operationen kann im Internet nachgeschaut werden und wird in der Klausur beiliegen.

# 3 Kontrollstrukturen

Häufig möchte ein Programmierer einen Abschnitt seines Programms nur unter bestimmten Bedingungen ausführen lassen oder mehrmals wiederholen lassen. Dazu werden in MMIX Sprünge benutzt. Der einfachste Sprung lässt sich mit dem Opcode `JMP` ausführen: sobald das Programm die Anweisung `JMP Label` erreicht, setzt es die Ausführung an der Zeile mit dem entsprechenden Label fort. Zum Beispiel würde die Anweisung `Loop JMP Loop` den MMIX-Simulator in eine Endlosschleife senden.

Um einen Sprung nur in bestimmten Fällen auszuführen, wird ein sogenannter Branch benutzt. Die Anweisung `BN $X Label` springt zum entsprechenden Label, wenn der Inhalt von `$X` negativ ist; ansonsten wird die Anweisung wie ein NOOP behandelt und das Programm fährt mit der nächsten Zeile fort. Es gibt viele verschiedene Branch-Opcodes: Zum Beispiel aktiviert `BZ` dann, wenn das Register 0 ist, und `BP` aktiviert wenn das Register positiv ist. Damit lassen sich beliebige Kontrollstrukturen umsetzen.

| C-Ähnlich | MMIX |
|---|---|
| `if (x > 0) {`<br>`  // do thing`<br>`}` | `BNP  $X,Cont`<br>`     // do thing`<br>`Cont ...` |
| `do {`<br>`  // do thing`<br>`} while(x > 0)` | `Loop ... // do thing`<br>`     BP $X,Loop` |
| `while(x > 0){`<br>`  // do thing`<br>`}` | `Loop BNP $X,Cont`<br>`     // do thing`<br>`     JMP Loop`<br>`Cont ...` |
| `for(int i = 0; i < 5;`<br>`  i++){`<br>`  // do thing`<br>`}` | `     SET $I,0`<br>`Loop CMP $C,$I,5`<br>`     BNN $C,Cont`<br>`     // do thing`<br>`     ADD $I,$I,1`<br>`     JMP Loop`<br>`Cont ...` |

Hier ein Beispiel für ein Programm mit Kontrollstrukturen:

```
tmp   IS    $0
a     IS    $1
b     IS    $2
i     IS    $3
n     IS    $4
      LOC   #100
Main  SET   n,17
      SET   i,1
      SET   a,1
FChk  CMP   tmp,i,n
      BN    tmp,FItr     // Wenn i<n ist, fuehre die Schleife aus
      JMP   End          // Ansonsten breche ab
FItr  ADD   tmp,a,b
      SET   b,a          // b' = a
      SET   a,tmp        // a' = a+b
      ADD   i,i,1        // i++
      JMP   FChk         // Wiederhole die Schleife
End   TRAP  0,Halt,0
```

# 4 Speicher

Bei MMIX handelt es sich um eine sogenannte RISC-Architecture. Das heißt, man interagiert mit dem Speicher ausschließlich, indem Daten zwischen dem Speicher und den Registern transferiert: Es gibt keine anderen Befehle, die auf den Speicher zugreifen.

Der Speicher ist eine Folge von Bytes; der virtuelle MMIX-Chip besitzt $2^{64}$ Bytes. Jedem dieser Bytes wird eine Adresse von 0 bis einschließlich $2^{64}-1$ zugewiesen, also ein Octa. Möchte man also nun ein Byte an der Stelle `$A` aus dem Speicher in ein Register `$X` laden, so würde man dazu den Opcode `LDB $X,$A,0` verwenden. Das letzte Argument stellt eine Verschiebung da, `LDB $X,$A,1` würde also das Byte laden, das auf `$A` folgt. Betrachten wir nun als Beispiel diesen Speicherblock:

| Adresse | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Inhalt | 01101010 | 11000100 | 00000011 | 10011001 | 11110110 | 01111000 | 00110101 | 11001110 |

Ein Aufruf von `LDB $X,$A,0` mit `$A`=6 führt demnach dazu, dass `$X`=0...0110101 ist. Allerdings tritt beim Aufruf mit `$A`=4 ein vorerst unerwartetes Verhalten auf: Das geladene Ergebnis ist 1...10110 statt 0...011110110! Der Grund dafür ist, dass der Opcode `LDB` das Byte als vorzeichenbehaftet betrachtet, also den Dezimalwert −10 ausliest, und diesen dann als vorzeichenbehaftetes Octa speichert (da ein Register ja 64 Bits hat und nicht 8 wie ein Byte). Wenn stattdessen der vorzeichenlose Wert, also hier 0...011110110, geladen werden soll, wird `LDBU` verwendet, mit einem *U* für *Unsigned*.

Möchte man nun eine größere Zahl, also ein Wyde, Tetra oder Octa laden, so gibt es auch für diese Fälle Opcodes, nämlich `LDW(U)`, `LDT(U)` und `LDO(U)`. Dabei sind die zu ladenden Wydes, Tetras und Octas quasi in festen Blöcken angeordnet: Die Adressen 0 bis 7 laden alle dasselbe Octa, die Adressen 0 bis 3 und 4 bis 7 jeweils dasselbe Tetra, usw. Zum Beispiel hat die Instruction `LDWU $X,$A,0` mit `$A`=3 das Ergebnis `$X`=0...01110011001, genauso wie mit `$A`=2. `$A`=1 hingegen produziert `$X`=0...0110101011000100

Die Gegenoperation, also das Opcode, das Daten aus einem Register in den Speicher lädt, fängt mit `ST` an, also `STB`, `STTU`... Dabei ist zu beachten, dass der Wert im Register innerhalb des vorgegeben Formats liegen muss. Versucht man 1...10110 mit `STBU` abzuspeichern, liegt ein Fehler vor, 18446744073709551606 ist schließlich viel größer als der maximale Wert eines Bytes. Verwendet man stattdessen `STB`, wird wie erwartet das Byte 11110110 geladen. Möchte man überflüssige Bits ignorieren, so wird das zu ladende Register zuerst mit der richtigen Maske 0...011111111 verundet (`AND $X,$X,255`).

Hier ein Beispiel für ein Programm, das mit dem Speicher arbeitet:

```
1  tmp  IS   $0
2  cur  IS   $1 // current state
3  ptr  IS   $2 // stack pointer
4  vstt IS   $3 // visit table
5       LOC  #100
6  Main LDBU cur,ptr,0
7       CMPU tmp,cur,15
8       BZ   tmp,End // if state is the solution
9       SRU  tmp,vstt,cur
10      AND  tmp,tmp,1
11      BP   tmp,RmPt // if visit table is true at the current index
12      SET  tmp,1
13      SLU  tmp,tmp,cur
14      OR   vstt,vstt,tmp // write to visit table
15      SRU  tmp,cur,3
16      MULU tmp,tmp,15
17      XOR  tmp,cur,tmp
18      SUBU tmp,tmp,3
19      BZ   tmp,RmPt
20      SUBU tmp,tmp,3
21      BZ   tmp,RmPt
22      SUB  tmp,tmp,1
23      BZ   tmp,RmPt
24 StPt XOR  tmp,cur,9
25      STBU tmp,ptr,0
26      XOR  tmp,cur,10
```

```
27        STBU  tmp,ptr,1
28        XOR   tmp,cur,12
29        STBU  tmp,ptr,2
30        XOR   tmp,cur,8
31        STBU  tmp,ptr,3
32        ADD   ptr,ptr,3
33        JMP   Main
34  RmPt  BZ    ptr,End
35        SUB   ptr,ptr,1
36        JMP   Main
37  End   TRAP  0,Halt,0
```

# 5 Anmerkung zum Debugging

Bei der Ausführung eines Programms `program.mms` ruft man zuerst `mmixal program.mms` bzw. `mmixal.exe program.mms` auf und dann `mmix program.mmo` bzw. `mmix.exe program.mmo`. Dabei wird jedoch kein Ergebnis sichtbar, man kann also nicht verifizieren, ob das Programm richtig läuft. Dazu verwendet man ein sogenanntes Trace mit `mmix -t1 program.mmo`: dieser Aufruf gibt die Details jeder Operation bis zu einer Tiefe von 1 an. Möchte man sich zusätzlich Zeilen anschauen, die wiederholt werden, verwendet man `-t2` (bis zu zwei Wiederholungen), `-t3` usw.