UMach

eine virtuelle Maschine

Georg-Simon-Ohm-Hochschule

Überblick

1. Projektbeschreibung
2. Architektur
3. Assembler
4. Demos
5. Debugging

UMach ist eine sehr lange Geschichte...

Teil I

Projektbeschreibung

Inhalt

Zielsetzung

Was wird geliefert

Ähnliche Werke

Das Ziel

Es soll eine komplette virtuelle und programmierbare Maschine entworfen, dokumentiert und implementiert werden.
Dabei soll die Maschine auch praktisch benutzbar sein – man sollte Programme assemblieren und ausführen können.

Gewünschter Workflow

1. Assembler Programm editieren.

```
loop:   SET R1 137
        CMP R1 ZERO
        BE  finish
        DEC R1
        JMP loop
finish: EOP
```

2. Das Programm assemblieren.

```
uasm -o myprog.umx myprog.uasm
```

3. „Bytecode“ ausführen.

```
umach -v myprog.umx
```

Nützlichkeit

- Studienmaterial für GDI, Rechnerarchitektur, Betriebssysteme, virtuelle Maschinen.
- Interesse an der ISA-Ebene, Low Level Arbeit mit Bits, Adressen, Registern.
- Alternative zur MIPS ASM?

Was wird geliefert

1. Spezifikation der Maschine
2. Spezifikation des Assemblers
3. Maschine in C99
4. Assembler in C99
5. Debugger (integriert und als Qt-Anwendung)
6. Demos

Ähnliche Werke

JVM Die Java Virtual Machine. Kennen nur wenige als virtuelle Stackmaschine.

MMIX Wurde von Donald Knuth entwickelt (der Mann hinter TEX und Literate Programming).
Wird in „The Art of Computer Programming“ als hypothetischer Rechner benutzt.
64 Bit, RISC, 256 Befehle, 256 Register,
MMIXAL als ASM-Sprache.
Kennt keiner.
Wichtigste Inspirationsquelle.

Teil II

Architektur und Implementierung

Inhalt

- Architektur
  - Maschinentyp
  - Register
    - Allzweckregister
    - Spezialregister
  - Befehle
  - Befehlsformate
  - Befehlsmenge
  - Speichermodell
  - I/O
  - Interrupts
- Implementierung
  - Programmablauf
  - Sprungtabellen

Maschinentyp

- UMach ist eine registerbasierte RISC Maschine.
  Wenige Befehle (69) mit fester Länge.
- Load/Store Speicherzugriff über Registerangabe.

```
LW    R1 R2 # R1 ← mem(R2)
SW    R1 R2 # R1 → mem(R2)
```

- Port I/O

```
IN   R17 R18 R19
OUT  R1  R2  R3
```

- Keine Pipeline
- Big endian

Register

Die UMach Maschine hat 32 Allzweckregister und 13 Spezialregister.
Jedes Register ist genau 32 Bit lang.
Die Register werden intern durch Nummern identifiziert:
Register Nummer 0, Nummer 1, Nummer 2... Nummer 44.

Allzweckregister

Die Register mit Nummern 1 bis 32 können frei verwendet werden. Sie werden von der Maschine ohne explizite Anweisung nicht geändert, außer dass sie beim Start der Maschine mit dem Wert Null belegt werden.
Die Allzweckregister haben die Namen $R1$, $R2$, … $R32$.
Register mit den Namen $R0$ und $R33$ gibt es nicht.

Spezialregister

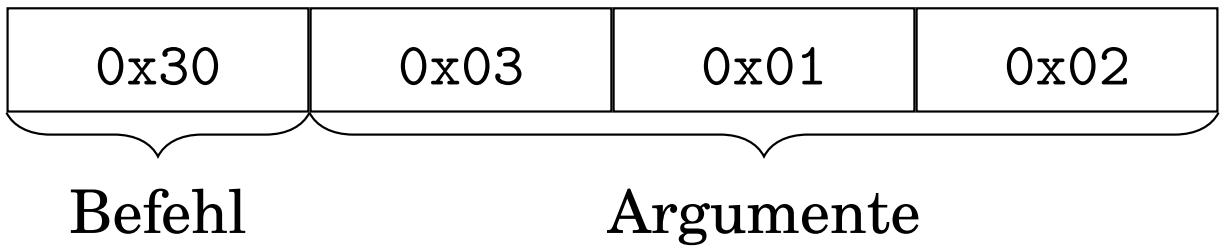
Dienen der Steuerung der Maschine und haben besondere Aufgaben. Sie werden von der Maschine verändert.
Die meisten sind schreibgeschützt.

Spezialregister - Liste

Name	Nummer	Beschreibung
PC	33	Program Counter
DS	34	Data Section (begin)
HS	35	Heap Section (first byte)
HE	36	Heap Section (last byte)
SP	37	Stack Pointer
FP	38	Frame Pointer
IR	39	Instruction Register
STAT	40	Status Register
ERR	41	Error Register
HI	42	Higher 32 Bits (Division, Multiplik.)
LO	43	Lower 32 Bits (Division, Multiplik.)
CMPR	44	Ergebniss von CMP
ZERO	0	Immer konstant Null

Befehle

Ein Befehl ist immer 32 Bits lang (4 Byte), auch wenn er keine Argumente nimmt (RISC Architektur). Erstes Byte ist der Befehl, die anderen 3 Bytes sind eventuelle Argumente.



Befehlsformate

Wie MMIX von Knuth, hat UMach mehrere Befehlsformate, d.h. mehrere Arten, wie man die Argumente einteilen und interpretieren kann.
Manche Befehle erwarten Registernummern, andere direkte nummerische Angaben verschiedener Längen, andere eine Mischung davon, andere keine Argumente.

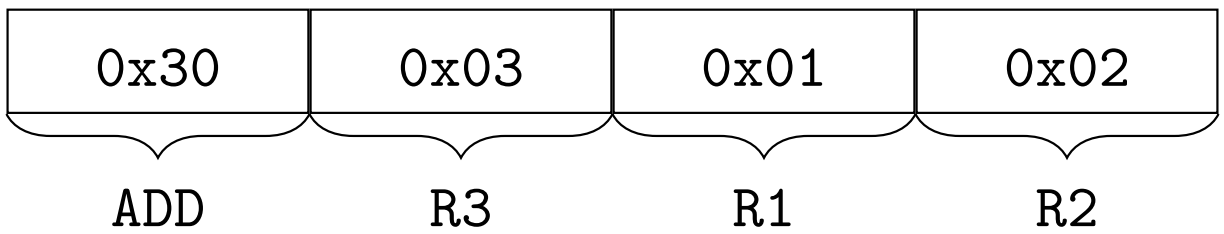
Befehlsformate

Format	zweites Byte	drittes Byte	viertes Byte
000	nicht verwendet		
NNN	3 Bytes Zahl		
R00	R_1	nicht verwendet	
RNN	R_1	2 Bytes Zahl	
RR0	R_1	R_2	nicht verwendet
RRN	R_1	R_2	1 Byte Zahl
RRR	R_1	R_2	R_3

(R_1, R_2, R_3: erste, zweite, dritte Registernummer)
Alle Zahlenangaben: big endian.

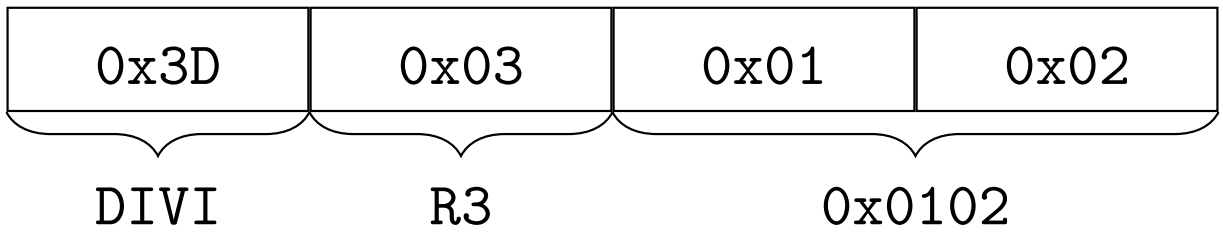
Befehlsformate - Beispiel RRR

`ADD`: drei Registernummern.



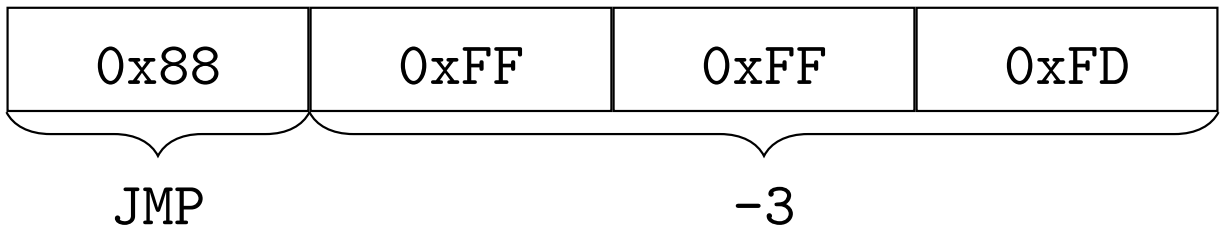
Befehlsformate - Beispiel RNN

`DIVI`: eine Registernummer und eine 2-Byte Zahl.



Befehlsformate - Beispiel NNN

`JMP`: ein 3-Byte Offset (vorzeichenbehaftet).



Befehlsmenge

1. Kontrollinstruktionen: `NOP`, `EOP`
2. Lade- und Speicherbefehle: `SET`, `LW`, `SB`, `PUSH`
3. Arithmetische Instruktionen: `ADD`, `SUB`, `INC`
4. Logische Instruktionen: `AND`, `XOR`, `SHL`, `ROTL`
5. Vergleichsinstruktionen: `CMP`, `CMPI`
6. Sprunginstruktionen: `JMP`, `BE`, `BL`
7. Unterprogramminstruktionen: `CALL`, `RET`, `GO`
8. Systeminstruktionen: `INT`
9. I/O Instruktionen: `IN`, `OUT`

Insgesamt 69 Befehle.

Speichermodell

Der Speicher der UMach Maschine enthält hauptsächlich

- Programmcode
- Programmdaten
- Freier Speicher

Kein Memory Mapped I/O.
Gesamte Speichergröße ist nach dem Start der Maschine fest.

Segmente

Der Speicher wird in Segmenten eingeteilt.

1. Interrupttabelle
2. Programmcode (Code-Segment)
3. Programmdaten (Daten-Segment)
4. Heap (Freispeicher)
5. Stack (Lokaler Speicher)

Der Code- und Daten-Segment werden aus der Programmdatei geladen. Der Rest ist dynamisch und kann durch Register (`HE`, `SP`) oder Befehle (`PUSH`, `POP`) manipuliert werden.

Heap und Stack Manipulieren

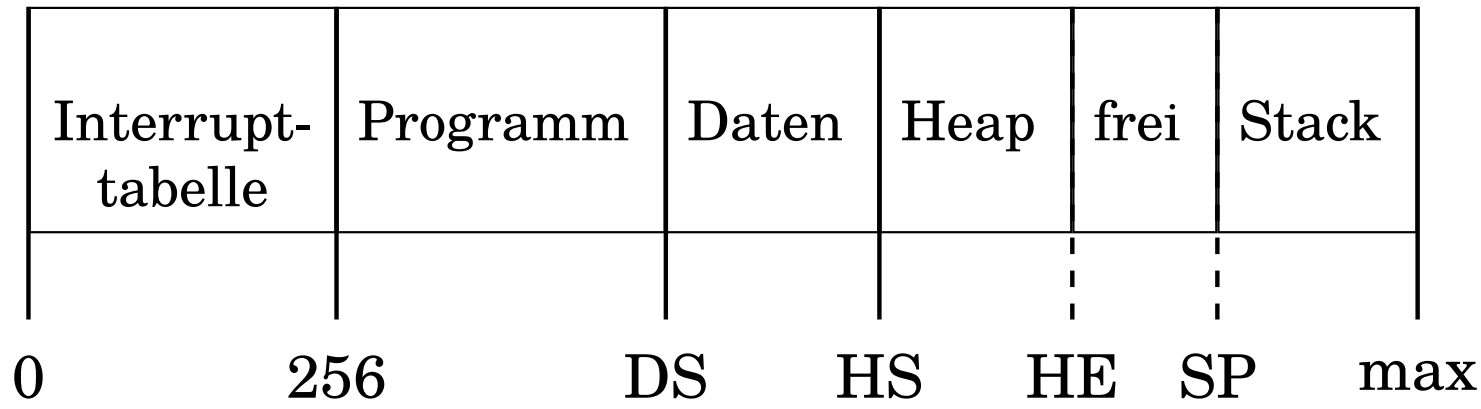
```
ADDI HE HE 128
# ...
SUBI HE HE 128
```

Speicher auf dem Heap reservieren erfolgt dadurch, dass der Inhalt des Registers `HE` (Heap End) hochgezählt wird.
Speicher freigeben durch runterzählen.

```
SUBI SP SP 32
# ...
ADDI SP SP 32
```

Lokaler Speicher wird durch Veränderung des Registers `SP` (Stack Pointer) erreicht.

Speicher-Layout



Segmentation Fault: schreiben in Code-Segment.
Stack Overflow: Befehl `PUSH` führt zum Überlappen der Register `SP` und `HE`.

Port I/O

Die UMach Maschine verwendet Port I/O, d.h. sie hat Befehle zum Ausgeben und Einlesen von Daten. Es werden Ports verwendet: durchnummerierte Ausgänge und Eingänge.

Ports

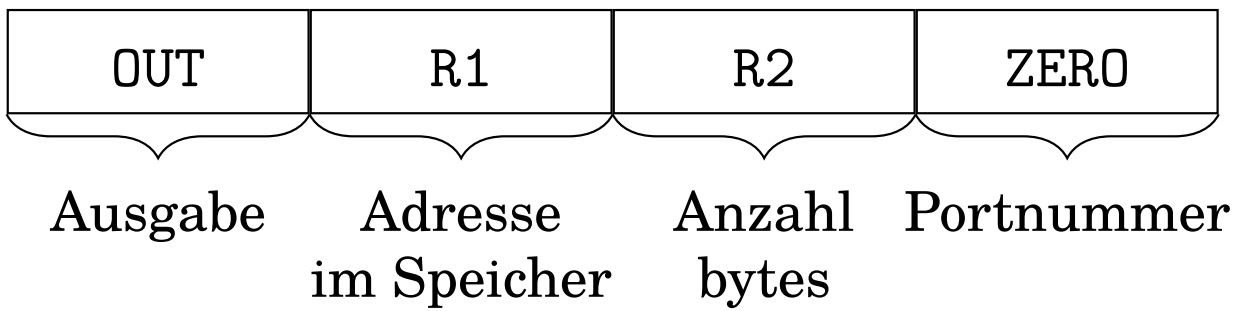
Es gibt 8 Ausgabeports und 8 Eingabeports, die jeweils von 0 bis 7 durchnummeriert sind.
Ausgabeport 0: `stdout`.
Eingabeport 0: `stdin`.

Transfer

Der Datentransfer findet direkt zwischen Speicher und I/O-Ports statt. Der Transfer blockiert die Maschine solange der Transfer noch nicht fertig ist.
Die I/O-Befehle haben das Format RRR (drei Registernummern).

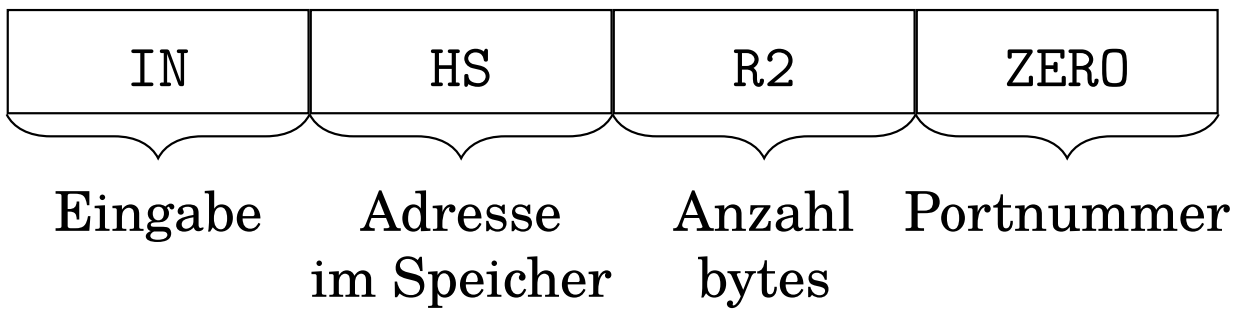
Ausgabe

Die Ausgabe erfolgt durch verwendung des Befehls `OUT`



Eingabe

Die Eingabe erfolgt durch verwendung des Befehls `IN`



Interrupts

Unterbrechungen im normalen Programmfluss, die mit einer Interruptnummer versehen sind und die abgefangen werden können. Analog zu „exceptions“ in Java/C++.
Abfangen heißt, dass eine Subroutine mit der Interruptnummer verbunden wird.
Ist ein Interrupt nicht mit einer Subroutine verbunden, so stoppt die Maschine wenn der Interrupt passiert.

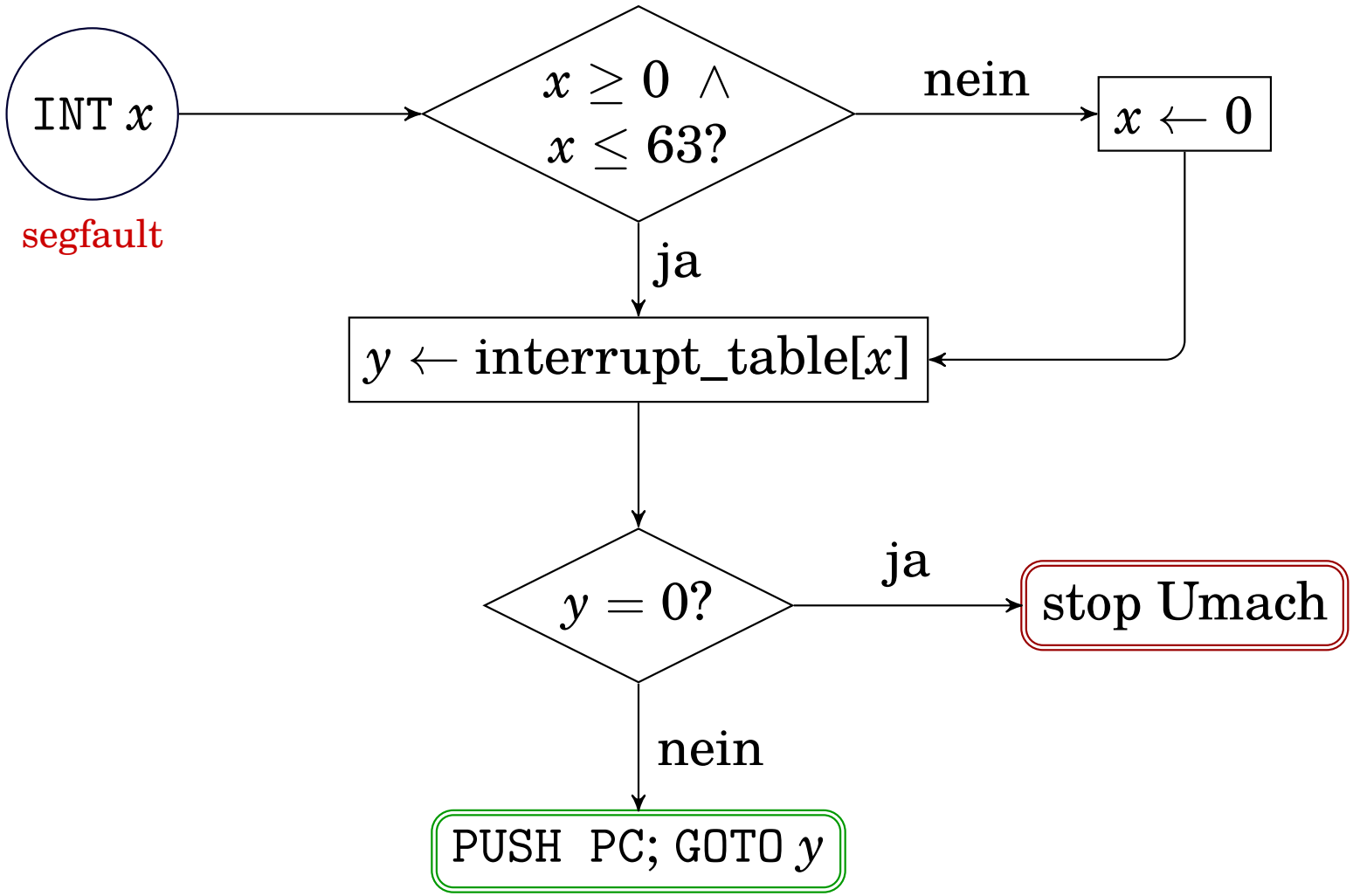
Arten von Interrupts

1. Hardware-Interrupts: wenn etwas schief mit einem Befehl geht: Division durch Null, Stack Overflow, falsche Befehlsnummer, ungültige Speicheraddresse, schreiben in das Codesegment, etc.
2. Software-Interrupts: werden vom Programmierer durch den Befehl `INT` angestoßen.

Interrupttabelle

Die Interrupttabelle startet an der Addresse Null und besteht aus 64 Einträgen, jeweils 32 Bit groß (=256 Bytes).
Jeder der 64 Einträge entspricht einer Interruptnummer.
Interrupt 26 $\rightarrow$ Adresse $26 \cdot 4 = 104$.
An jedem Index steht entweder Null oder die Addresse einer Subroutine (Interrupt Handler). Diese wird ausgeführt, wenn der entsprechende Interrupt generiert wird.
Adresse des Interrupt Handlers ist Null $\mapsto$ Maschine ausschalten.

Wie läuft ein Interrupt ab



Programmablauf

Die Maschine hat grundsätzlich zwei Schritte, die sie immer wieder wiederholt: fetch und execute.

```
void core_run_program(void)
{
    while (running) {
        core_fetch();
        core_execute();
    }
}
```

Fetch

Fetch: die nächste Instruktion aus dem Speicher holen.

```
void core_fetch(void)
{
    if (! running) { return; }
    mem_read                    // read from mem
    ( instruction,              // whereto
      registers[PC].value,      // wherefrom
      4                         // how much
    );
}
```

Lese 4 Bytes aus dem Speicher ab der Adresse `PC` in den Puffer `instruction`.
Nach `fetch` steht der neue Befehl im globalen Array `instruction[4]`. (big endian)

Execute

Befehl ausführen und PC inkrementieren.

```
struct command *cmd =
       command_by_opcode(instruction[0]);
if    (cmd != NULL)
      { cmd->opfunc();                    }
else { interrupt(INT_INVALID_CMD); }
registers[PC].value += 4;
```

Es wird nach einem Funktionszeiger gesucht, der dem Befehlscode entspricht (die Funktion implementiert den Befehl). Fall vorhanden, ausführen. Falls nicht, Interrupt generieren.
(Der Funktionszeiger ist in einer Struktur `command` gepackt.)

Sprungtabellen

Wie wird schnell nach einem Funktionszeiger gesucht?
Mit Sprungtabellen: ein Array von Funktionszeigern in Strukturen gepackt, wo jede Struktur genau an dem Index steht, der gleich dem entsprechenden Befehlscode ist.
Suchaufwand $O(1)$. Schneller geht's nicht.

Sprungtabellen – Auszug

```
struct command opmap[OPMAX] = {
    [0x00] = {0x00, "NOP", core_nop ,NUL},
    [0x04] = {0x04, "EOP", core_eop ,NUL},
    [0x10] = {0x10, "SET", core_set ,RNN},
    ....
    [0x90] = {0x90, "GO" , core_go  ,R00},
    [0x91] = {0x91, "CALL", core_call,NNN},
    [0x92] = {0x92, "RET", core_ret ,NUL},
    [0xA0] = {0xA0, "INT", core_int ,NNN},
    [0xB0] = {0xB0, "IN" , core_in  ,RRR},
    [0xB8] = {0xB8, "OUT", core_out ,RRR}
};
```

(C99 Magie.)

Suchen in der Sprungtabelle

Wie findet mal die Funktion, die einem Befehlscode entspricht?

```
struct command* command_by_opcode
(int opcode)
{
    if (opmap[opcode].opname) {
        return & opmap[opcode];
    } else {
        return NULL;
    }
}
```

Ein Beispiel: `ADD`-Befehl

Eintrag in der Sprungtabelle:

```
[0x30] = {0x30, "ADD", core_add, RRR}
```

Der Befehl `ADD` hat die Befehlsnummer `0x30` und die Funktion `core_add` steht am Index `0x30` in der Sprungtabelle.

ADD – Implementierung

```
int core_add(void) {
    int32_t a = 0;
    int32_t b = 0;

    read_register  (instruction[2], &a    );
    read_register  (instruction[3], &b    );
    write_register (instruction[1], a + b);
    return 0;
}
```

(Veränderte Version, Error Checks gelöscht).

Teil III

Demos

Inhalt

Meist verwendete Befehle

Hilfsfunktionen

Hello World

Fibonacci Zahlen

Zahl raten

Tic Tac Toe

SET

`SET R1 5` oder `SET R1 label`
Tut das und jenes. Label wird durch Adresse ersetzt.
Beispiel:

Sprungbefehle

- `JMP label`
- `BL label` springt zum angegebenen Label, falls R1 kleiner R2 ist Weitere Möglichkeiten: `BLE`, `BG`, `BGE`, `BE`

Vergleiche

`CMP R1 R2`

Immer or einer bedingten Verzweigung.

IO-Befehle

- `IN R1 R2 ZERO`
- `OUT R1 R2 ZERO`

Arithmetische Befehle

`ADD` Addiert $a + b = 0$

`SUB`

`INC`

`DEC` Berechnet $\pi^2 = \frac{a^i}{e}$

`MUL`

`DIV` Muss man auf die Null aufpassen, denn die Null wurde erst im 9ten Jahrhundert erfunden und alte Programme nicht mehr kompilieren.

und die entsprechenden Immediate-Varianten

Funktionen

```
CALL funktion
```

Hilfsfunktionen

- inputint
- printint
- putchar
- newline

Hello World

```
SET R1 hello
SET R2 13
OUT R1 R2 ZERO
.data
.string hello "Hello World!"
```

Fibonacci Zahlen

$$\text{Folge } X_n = X_{n-1} + X_{n-2} \text{ mit } X_1 = 1, \text{ und } X_2 = 2$$

Zahl raten

Erzeugt eine Pseudozufallszahl aus dem eingegebenen Seed.
Gibt Rückmeldung ob die geratene Zahl kleiner oder größer als die gesuchte ist.
Zählt die Anzahl der Versuche.

Tic Tac Toe

Belegung der Register:

- R1-R9: Spielfelder
- R10: Aktueller Spieler
- R20: Anzahl der Spielzüge

Spielzyklus:

1. Eingabe
2. Ausgabe
3. Auswertung

Am ende des Spiels:

- neue Runde oder Beenden
- bei neuer Runde aufräumen