UMach

eine virtuelle Maschine

L. Beraru W. Linne S. Beer W. Fink

Betreuer: Prof. Dr. Kern
Georg-Simon-Ohm-Hochschule

29. November 2012

Überblick

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

UMach ist eine sehr lange Geschichte...

Teil I

Projektbeschreibung

Inhalt I

Das Ziel

- Eine komplette virtuelle Maschine soll entworfen, dokumentiert und implementiert werden.
- Die Maschine soll 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
```

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 (virtuelle Stackmaschine).
MMIX	Wurde von Donald Knuth entwickelt. 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 I

Maschinentyp

- UMach ist eine registerbasierte RISC Maschine.
  Wenige Befehle (69) mit fester Länge (32 Bit).
- Load/Store Speicherzugriff über Registerangabe.

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

- Big endian
- Port I/O

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

Register

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

Allzweckregister

- Register mit Nummern 1 bis 32 können frei verwendet werden.
- Werden von der Maschine ohne explizite Anweisung nicht geändert, außer dass sie mit Null initialisiert werden.
- Namen entsprechen der Nummerierung:
  $R1, R2, \ldots R32$.

Spezialregister

- Dienen der Steuerung der Maschine.
- Haben besondere Aufgaben.
- Werden von der Maschine im Betrieb verändert.
- Meisten sind schreibgeschützt.

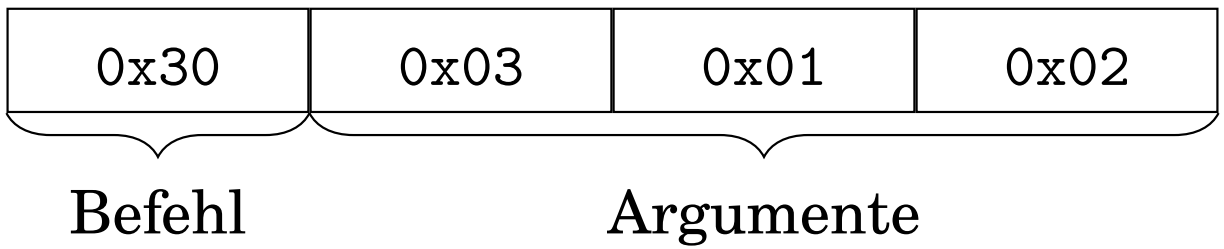
Spezialregister - Liste

Name	Nummer	Beschreibung
PC	33	Program Counter
DS	34	Data Segment
HS	35	Heap Segment
HE	36	Heap End
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

Befehlsformat

Allgemeines Format

Befehlslänge: 4 Byte. Erstes Byte ist der Befehlscode.



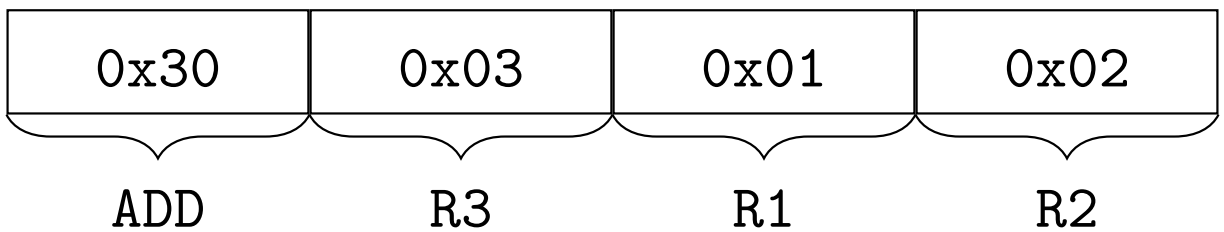
Befehlsformat

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.

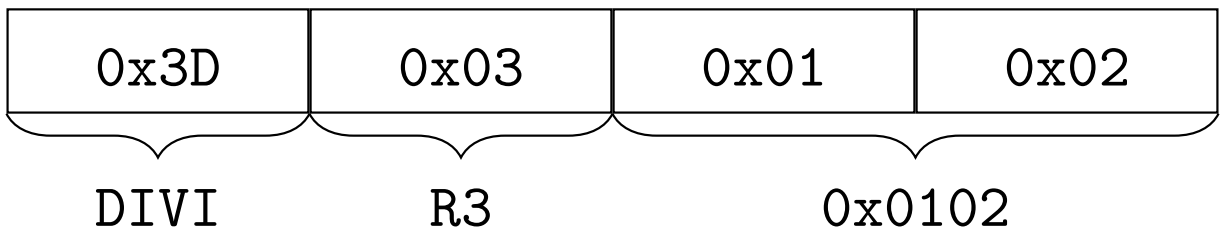
Befehlsformat - Beispiel RRR

`ADD`: drei Registernummern.



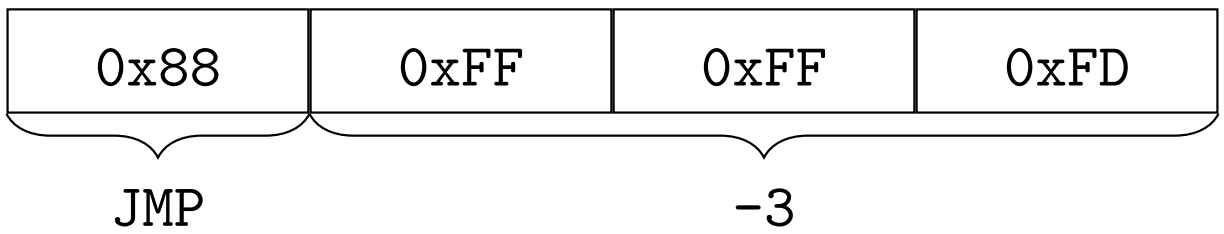
Befehlsformat - Beispiel RNN

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



Befehlsformat - Beispiel NNN

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



Befehlsmenge

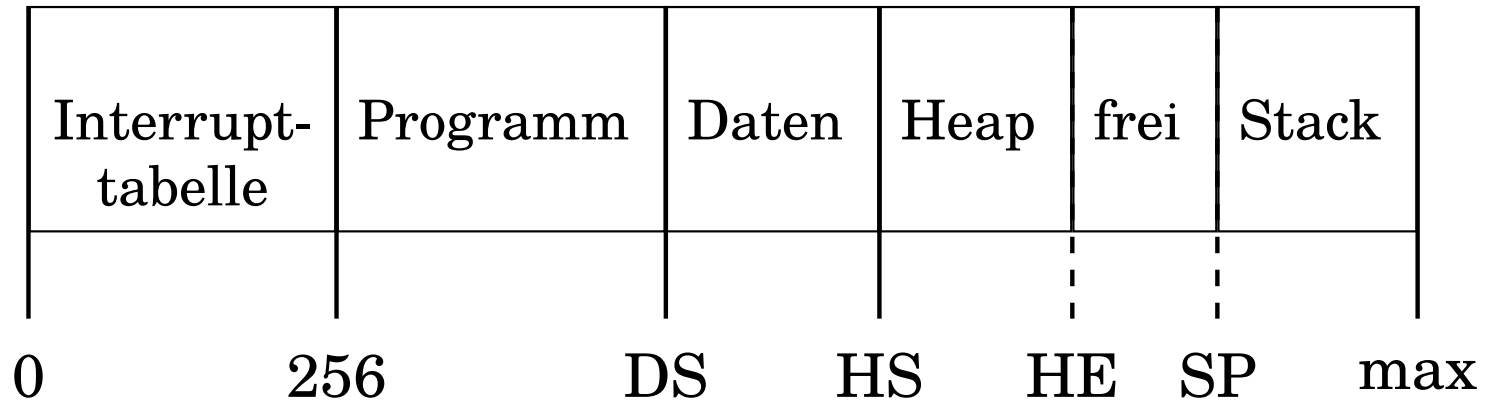
Kategorien

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

Speichersegmente



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

Heap und Stack Manipulieren

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

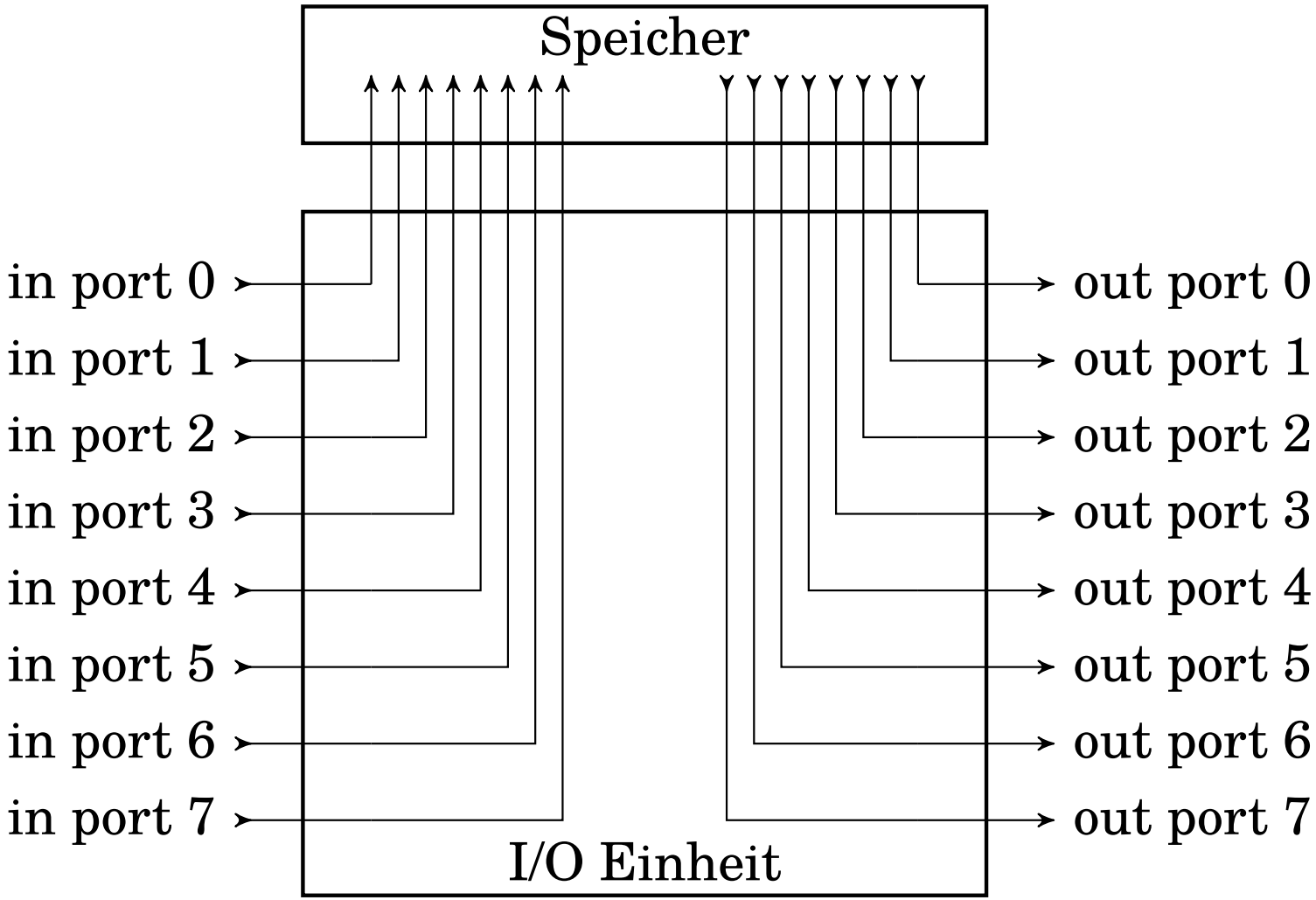
Speicher auf dem Heap reservieren erfolgt dadurch, dass der Inhalt des Registers `HE` (Heap End) hochgezählt wird.
Speicher freigeben durch runterzählen.

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

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

Port I/O

≠ Memory Mapped I/O

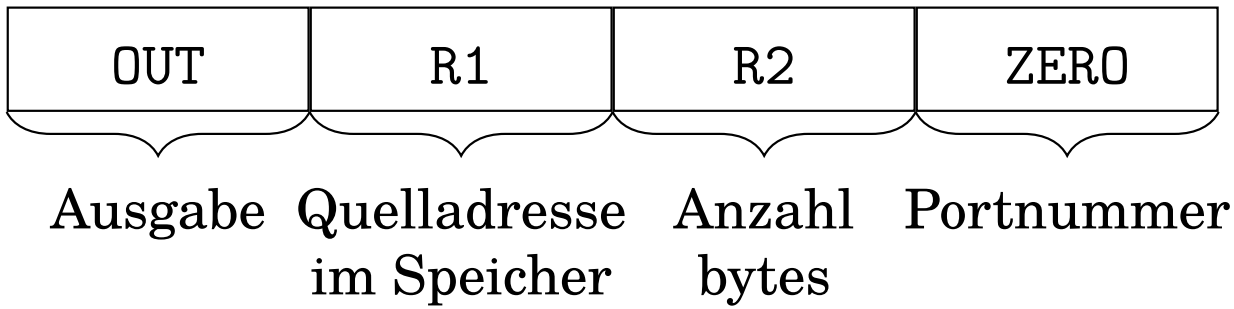


Datentransfer

- Direkt zwischen Speicher und I/O-Ports.
- Blockiert die Maschine solange der Transfer noch nicht fertig ist.

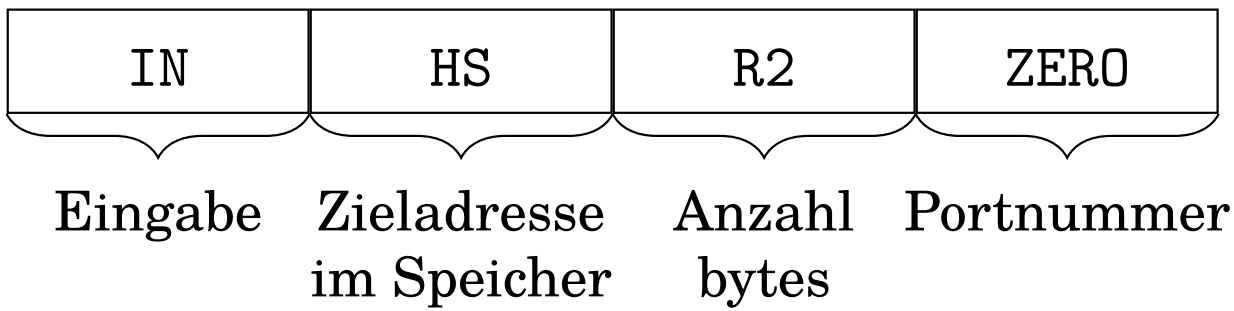
Ausgabe

Die Ausgabe erfolgt durch verwendung des Befehls `OUT`



Eingabe

Die Eingabe erfolgt durch verwendung des Befehls `IN`



Interrupts

Unterbrechungen im normalen Programmfluss (analog zu „exceptions“ in Java/C++).

- Mit einer Interruptnummer versehen (`INT 26`).
- Können abgefangen werden (interrupt handlers).

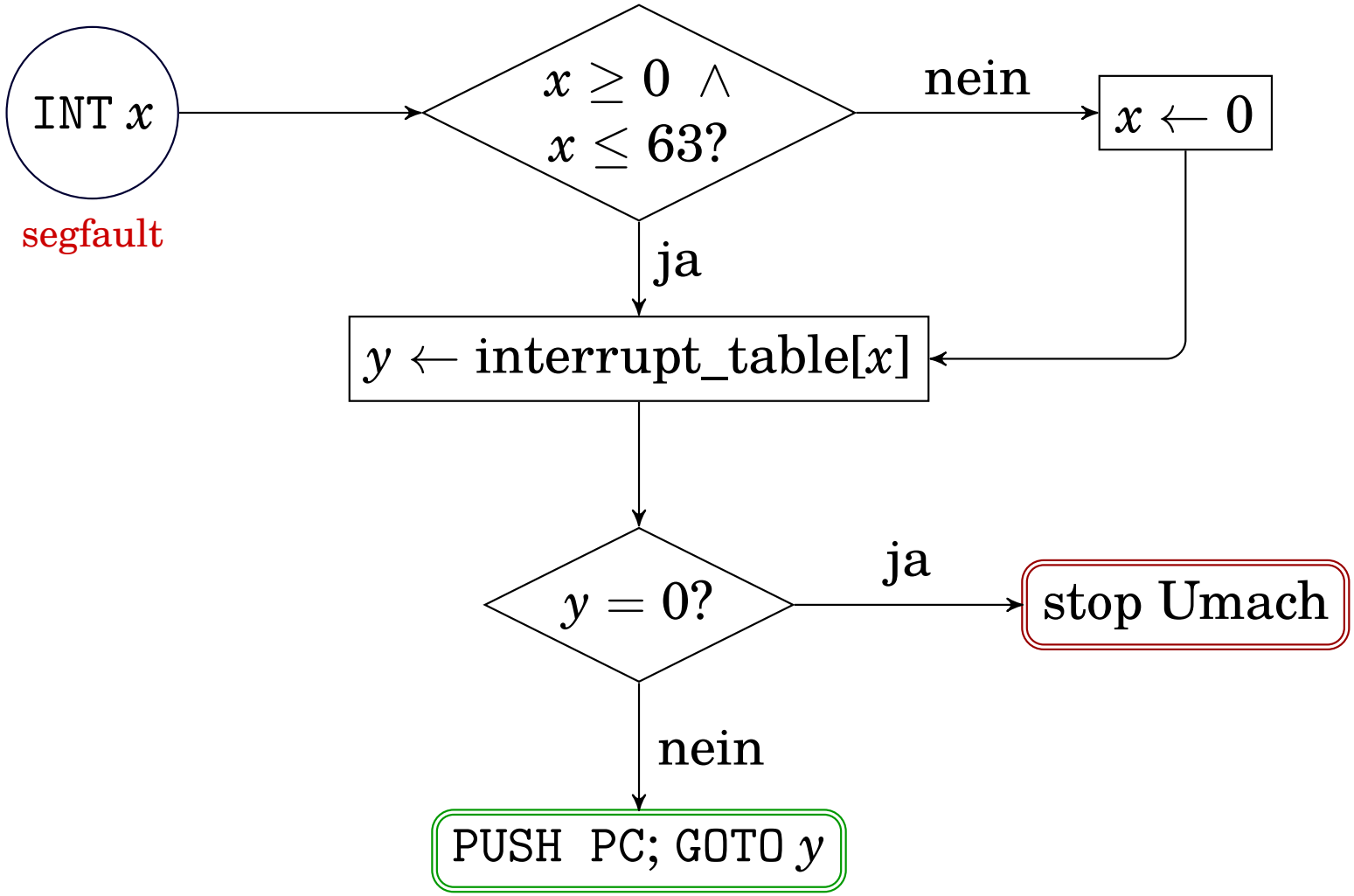
Arten von Interrupts

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

- Startet an der Addresse Null.
- 64 Einträge, jeweils 32 Bit groß.
- Jeder Eintrag entspricht einer Interruptnummer.
  Interrupt 26 $\rightarrow$ Adresse $26 \cdot 4 = 104$.
- Eintrag enthält entweder Null oder die Adresse eines „interrupt handlers“.

Wie läuft ein Interrupt ab



Programmablauf

Die Maschine führt grundsätzlich zwei Schritte aus, die sich immer wieder wiederholen: 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 globalen Puffer `uint8_t instruction[4]`.

Execute

Befehl ausführen und PC inkrementieren.

```
int opcode = instruction[0];

int (*cmd) (void) =
      command_by_opcode(opcode);

if   (cmd != NULL) { cmd();          }
else { interrupt(INT_INVALID_CMD); }
registers[PC].value += 4;
```

Es wird nach einem Funktionszeiger cmd gesucht, der dem Befehlscode entspricht. Fall vorhanden, ausführen. Falls nicht, Interrupt generieren.

Frage

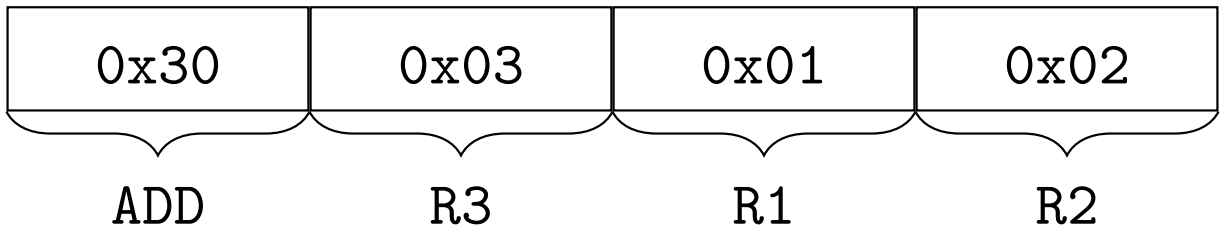
Wie wird schnell nach einem Funktionszeiger gesucht?

Sprungtabellen – Auszug

```
int (*opmap[]) (void) = {
//   opcode ->  function
//   ---------------------
     [0x00]   =  core_nop,
     [0x04]   =  core_eop,
     [0x10]   =  core_set,
     [0x30]   =  core_add,
     ....
     [0x91]   =  core_cal,
     [0x92]   =  core_ret,
     [0xA0]   =  core_int,
     [0xB0]   =  core_in ,
     [0xB8]   =  core_out,
};
```

Suchaufwand $O(1)$. Schneller geht's nicht.

Ein Beispiel: ADD-Befehl



Eintrag in der Sprungtabelle:

```
[0x30] = core_add,
```

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

Assembler

Inhalt I

Inhalt II

Aufgabe des Assemblers

Der Assembler übersetzt Quelltext in UMach Bytecode und erstellt Debuginformationen.

Erwünschte Eigenschaften

- “Angenehme” Syntax
- Performance
- Aussagekräftige Fehlermeldungen
- Nützliche Debuginformationen

Bedienung

Der Assembler “`uasm`” wird über eine Shell aufgerufen.

Aufrufsyntax: `uasm [-o outfile] [-g] [-w] file(s)`

Assembler Syntax Beispiel

```
SET R1 hello
myloop:  #useful comment
    CALL println
    DEC R2
    CMP R2 ZERO
BNE myloop
#lines containing only a comment or nil are ignored
SET R1 9001
EOP
.data  #begin of data definitions
.string hello  "Hello World!"
.int    answer 42
.int    drink  0xCAFE
```

Assembler Syntax Regeln

- Beliebig viele ‘\t’ und ‘␣’ vor und nach Tokens
- Nur Symbolbezeichner sind case-sensitiv
- Symbolbezeichner dürfen keine Zahlenwerte sein
- Symbolbezeichner bestehen aus genau einem Wort

Assembler Syntax Regeln

- Definition einer Sprungmarke endet mit ‘:’
- Sprungmarken stehen in einer eigenen Zeile
- Für alle Dateien gilt der gleiche Namensraum
- Ab ‘#’ beginnt ein Kommentar bis einschl. ‘\n’

Implementierung

- 2-pass Assembler, geschrieben in C99
- Abhängigkeiten: `glibc` und `glib`
- Funktioniert mit 32- und 64-Bit Compiler

Implementierung

Toolchain

Ausschließlich FOSS Komponenten:

- GNU/Linux
- GCC und clang
- GNU make und gdb
- glib
- Git
- LaTeX
- Valgrind

Implementierung

Datenstrukturen

- Drei Hashtabellen für
  1. Befehle *(statisch)*
  2. Register *(statisch)*
  3. Symbole *(dynamisch)*
- Eine verkettete Liste für den Inhalt von Variablen

Commands

```
typedef enum {
    CMDFMT_NUL, CMDFMT_NNN, CMDFMT_R00,
    CMDFMT_RNN, CMDFMT_RR0, CMDFMT_RRN,
    CMDFMT_RRR
} cmdformat_t;

typedef struct {
    uint8_t      opcode;
    char        *opname;
    cmdformat_t  format;
    char         has_label;
} command_t;
```

Register

```
typedef struct {
    uint8_t regcode;
    char    *regname;
} register_t;
```

Symbole

```
typedef enum {
    SYMTYPE_JUMP,
    SYMTYPE_INTDAT,
    SYMTYPE_STRDAT
} symbol_type_t;

typedef struct {
    char          *sym_name;
    symbol_type_t sym_type;
    uint32_t      sym_addr;
} symbol_t;
```

Variablen

```
typedef struct {
    char *label;
    char *value;
} string_data_t;

typedef struct {
    char   *label;
    int32_t value;
} int_data_t;
```

Variablen

```
typedef enum {
    DATATYPE_INT, DATATYPE_STRING
} data_type_t;

typedef struct {
    data_type_t type;
    union {
        string_data_t string_data;
        int_data_t    int_data;
    };
} data_t;
```

Implementierung

Assembler Pass 1

“Predict”

- Findet Sprungmarken und berechnet deren Adresse
- Berechnet die gesamte Codegröße
- Speichert init. Werte von Variablen
- Berechnet Adressen von Variablen

Implementierung

Assembler Pass 2

"Execute"

- Generiert UMach Bytecode
- Generiert Debuginformationen
- Speichert init. Werte von Variablen in das Outputfile

Implementierung

Alternativen

Der Parser muss nicht unbedingt selbst geschrieben werden, Tools wie z.B. *GNU Bison* können aus einer Grammatik einen Parser generieren.

Vorteile:

- Verständlich
- Weniger Aufwand bei komplexer Syntax

Nachteile:

- Hoher Lernaufwand
- Geringere Performance

Performance

$Durchsatz \approx 1.4 \times 10^6 \frac{Zeilen}{Sekunde}$ (AMD Athlon II X2 250, 3 GHz)

Speicherbedarf wächst linear mit der Anzahl der Symbolen

Auflösung von Symbolen meist in $\mathcal{O}(1)$

Keine linearen Suchen; Programmlaufzeit in $\mathcal{O}(n)$

Fehlerdiagnose

Der `uasm` Assembler informiert den Benutzer u.a. über folgende Fehler im Quelltext:

- Unbekannte Befehle
- Ungültige Argumente zu einem Befehl
- Unbekannte Symbole (Sprungmarken und Variablen)
- Unbekannte Register
- Ungültige Deklaration einer Variable
- Re-Definition einer Sprungmarke
- etc. . .

Fehlerdiagnose

Zusätzlich zur Art des Fehlers wird Name & Zeilennummer der Quelltextdatei in welcher der Fehler gefunden wurde ausgegeben.

Beispiele:

```
echo.uasm, line 1: No such command: <SQRT>
echo.uasm, line 2: Command <CMP> expects RR0: REG,REG
echo.uasm, line 3: Unset label <getinput>
echo.uasm, line 4: Not a register: <R77>
echo.uasm, line 6: Label <get_input> already exists
echo.uasm, line 8: No content for <myint> provided
```

Debuginformationen

Wird das generieren von Debuginformationen per "`$uasm -g ...`" aktiviert, werden folgende Dateien erstellt:

- `u.out.fmap`
- `u.out.sym`
- `u.out.debug`

File Map

Die Textdatei `u.out.fmap` enthält n 1:1 Relationen (File-ID, File-Name).

Beispiel:

```
0 tictactoe.uasm
1 func/inputint.uasm
2 func/newline.uasm
3 func/printint.uasm
4 func/putchar.uasm
```

Debug File

Die Binärdatei `u.out.debug` enthält n 32Bit-Datentripel (File-ID, Line-No, Address).

Beispiel:

```
00000000:  00 00 00 00  00 00 00 05  00 00 01 00
0000000c:  00 00 00 00  00 00 00 08  00 00 01 04
00000018:  00 00 00 00  00 00 00 09  00 00 01 08
00000024:  00 00 00 00  00 00 00 0d  00 00 01 0c
00000030:  00 00 00 00  00 00 00 0e  00 00 01 10
0000003c:  00 00 00 00  00 00 00 0f  00 00 01 14
00000048:  00 00 00 00  00 00 00 11  00 00 01 18
00000054:  00 00 00 00  00 00 00 15  00 00 01 1c
00000060:  00 00 00 00  00 00 00 16  00 00 01 20
0000006c:  00 00 00 00  00 00 00 17  00 00 01 24
```

Symbol File

Die Textdatei `u.out.sym` enthält n Datentripel (Address, Symbol-Type, Symbol-Name).

Beispiel:

```
000005e0 jmp start_inputint
000006b4 jmp printint_convert
0000050c jmp p1Won
000004d0 jmp draw
0000070c jmp putchar
00000784 str promptdraw
00000794 int newln
00000540 jmp p2Won
000005ec jmp inputint_nextnbr
```

Teil IV

Debugger

Inhalt I

Inhalt II

GUI

Grafisches Benutzerinterface für die UmachVM zur Entwicklung und für das Debuggen

Anforderungen

- Gui und Core eigene Prozesse
- Debugging
  - Setzen von Haltepunkten
  - Einzelschritt
  - Anzeigen der Codestelle
  - Auslesen und Manipulieren der Register und Daten
- Plattformunabhängig

Aufgaben eines Debuggers

- Steuerung des Programmablaufs (Haltepunkt, Einzelschritt)
- Inspizieren (Daten, Zustand)
- Modifizieren (Daten, Zustand)

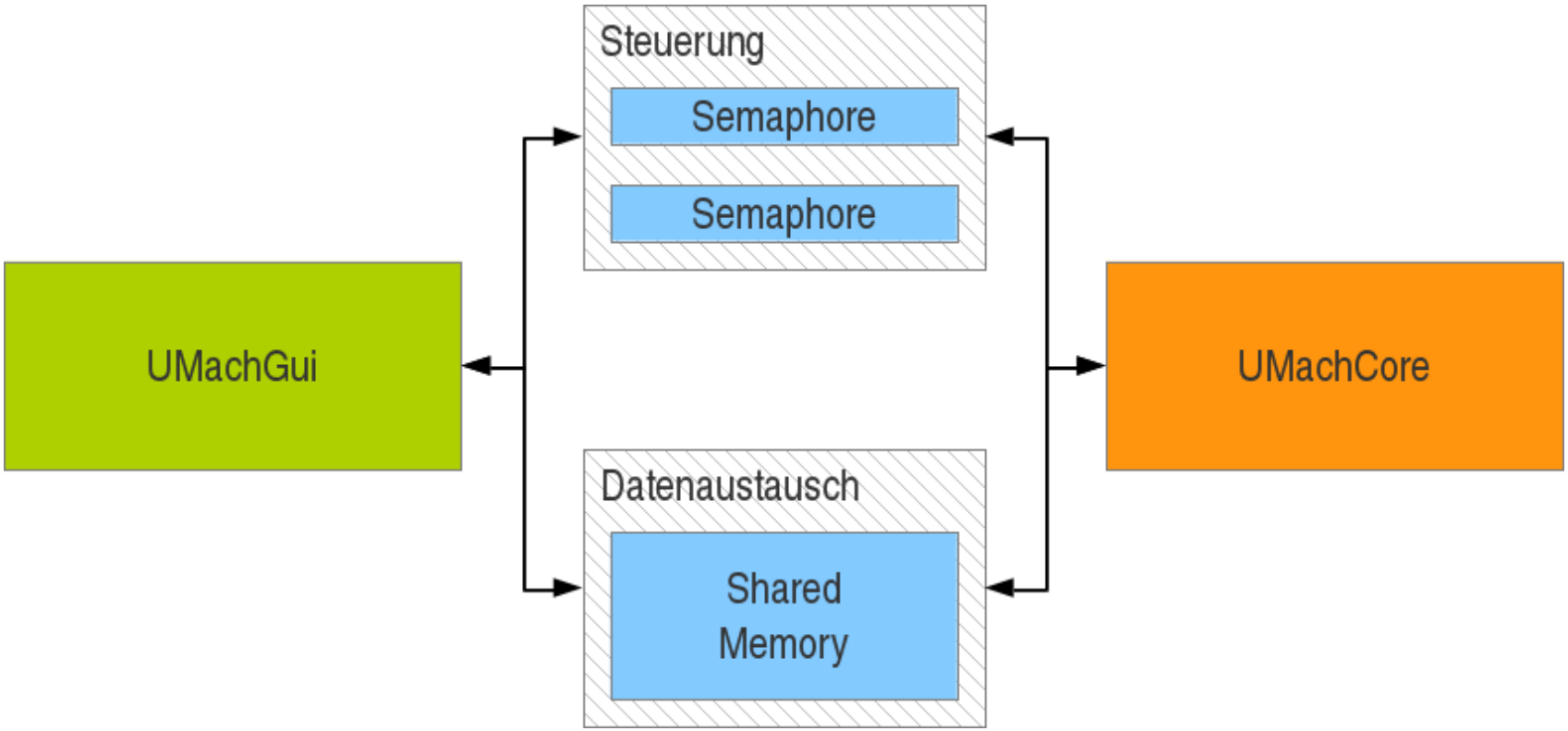
Realisierung der Steuerungsfunktionalität

- Temporäres ersetzen von Instruktionen durch spez. Interrupt
- Abgleich der Instruktionsadresse (Hardware, Software)
- Umach
  - Abgleich der Adresse
  - Assembler liefert Tabelle

Übersicht

- VM läuft als eigener Prozess
- Kommunikation zur Steuerung und Datenaustausch
- Debugging
  - Steuerung der VM
  - Auslesen der VM

Schaubild


Steuerung
Semaphore
Semaphore
UMachGui
UMachCore
Datenaustausch
Shared
Memory

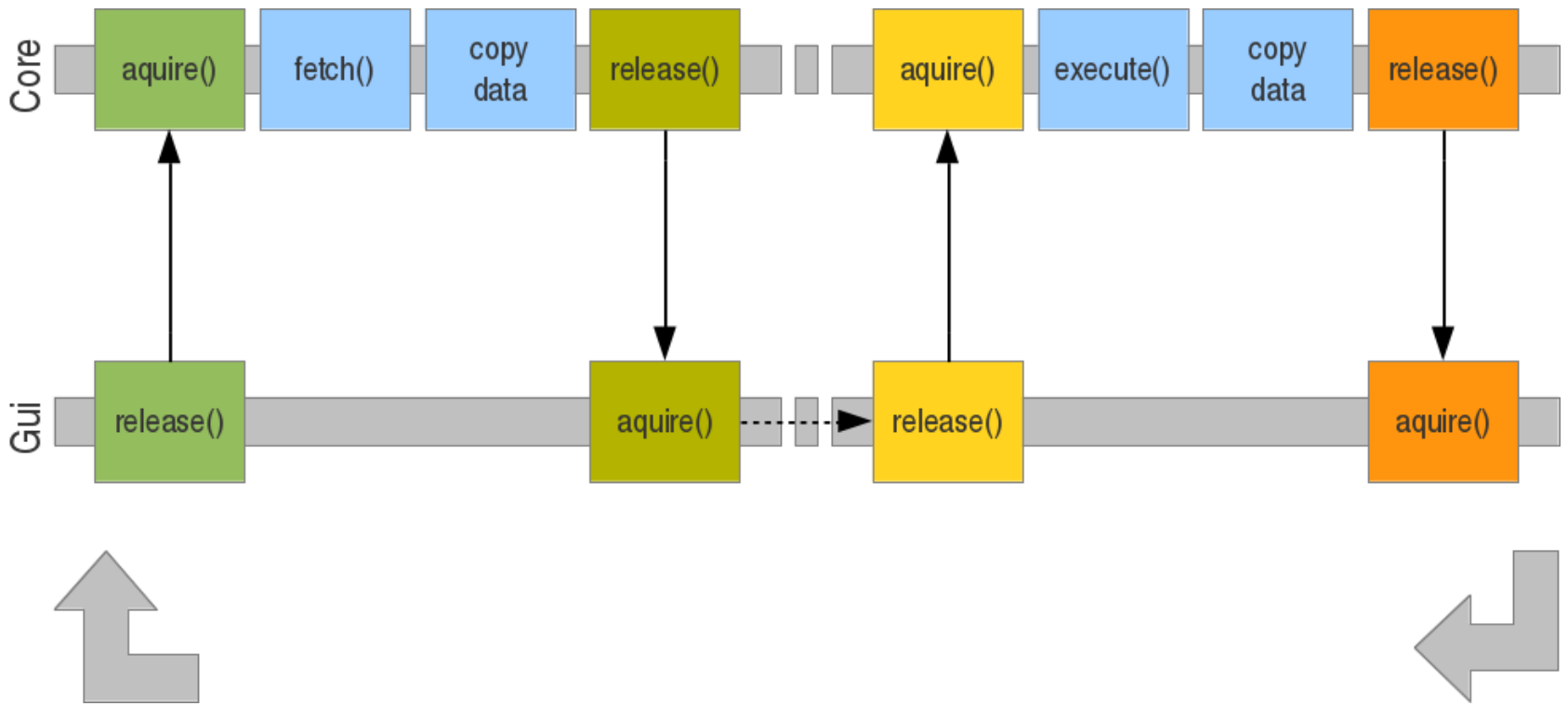
QSystemSemaphore

- Ressourcenzähler
- aquire()
  Ressource anfordern
- release()
  Ressource freigeben
- Blockiert Prozess wenn keine Ressource verfügbar
- Zugriff über ID

QSharedMemory

- Prozessübergreifender Speicher
- Erster Konstruktoraufruf erzeugt Shared Memory
- Zugriff über ID
- attach()
  Anhängen an den Prozess
- detach()
  Abhängen vom Prozess
- Letzter Aufruf von detach() zerstört QSharedMemory
- Sicherstellung der Freigabe

Kontrollzyklus



Warum QT?

- Einfach zu verwenden, gute Dokumentation
- Plattformunabhängigkeit
- Flache Hierarchie - Einfach zu erweitern
- Vorhandene Erfahrungswerte

Signal & Slot Prinzip

- Alternative zu Callback
- Einfache Syntax
- Minimal langsamer als Callback
- Benötigt Meta Object Compiler

Signal

```
signals:
  void requestOpenFileInTab(IFile *file);
```

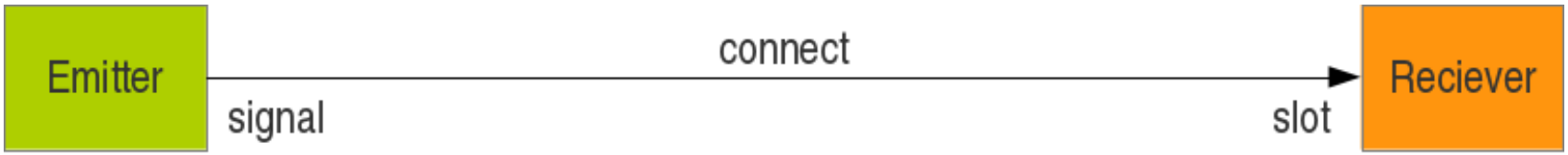
Emit

```
if (!asmFiles[i]->isOpen()) {
  emit requestOpenFileInTab(asmFiles[i]);
}
```

Slot

```
private slots:
  void openFileInTab(IFile *file);
```

Connect




```
connect (m_project,
         SIGNAL(requestOpenFileInTab(IFile)),
         this,
         SLOT(openFileInTab(IFile)));

m_menu = new QMenu(this);
m_actionOpen = m_menu->addAction("Open");
connect (m_actionOpen, SIGNAL(triggered()),
         this, SLOT(openFile()));
```

Projektdatei

- Projektdatei .umproject
  - Zugehörige Assemblerdateien
  - Speicherung von Einstellungen und Haltepunkten
- Benötigt Make-Routine

Demo

- fibonacci.umx
- Haltepunkte
- Nächste Instruktion
- Registerinhalt

Ausblick

- Speichern von Optionen
- Symbolinformation
- Manipulation Daten & Zustand
- Weitere Ideen
  - Speicheranzeige
  - Graphische Darstellung Speicherbelegung
  - ...

Teil V

Demos

Inhalt I

SET

`SET R1 5` oder `SET R1 label`
Setzt das Register R1 auf den angegebenen Wert. Labels werden durch Adressen ersetzt.

Arithmetische Befehle

`ADD R1 R2 R3`	Addiert $R1 \leftarrow R2 + R3$
`SUB R1 R2 R3`	Subtrahiert $R1 \leftarrow R2 - R3$
`INC R1`	Inkrementiert $R1++$
`DEC R1`	Dekrementiert $R1--$
`MUL R1 R2`	Multiplikation
`DIV R1 R2`	Division durch 0 führt zu Interrupt

und die entsprechenden Immediate-Varianten

Bedingte Sprünge

- `CMP R1 R2` Vergleicht das Register R1 mit R2.
- `BL label` Springt zum angegebenen Label, falls R1 kleiner R2 ist. Weitere Möglichkeiten: `BLE`, `BG`, `BGE`, `BE`

Unbedingte Sprünge

- `JMP label`
- `CALL funktion`

IO-Befehle

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

Hilfsfunktionen

- inputint
- printint
- putchar
- newline

Hello World

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

Fibonacci Zahlen

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

- Unterscheidung zwischen $n = 1$, $n = 2$ und $n \geq 3$ notwendig

Zahl raten

- Seed erzeugen
- Pseudozufallszahl generieren
- Formel: $X_{n+1} = (a + b * X_n) \mod m$
- Spieler rät die Zahl
- Rückmeldung ob die geratene Zahl größer oder kleiner der gesuchten ist
- 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