Georg-Simon-Ohm-Hochschule, Fakultät Informatik

Berichterstattung zum IT-Projekt

UMach, eine einfache virtuelle Maschine

Simon Beer Liviu Beraru Willi Fink
Werner Linne

18. Januar 2013

Inhaltsverzeichnis

1 Projektvorstellung

Das IT-Projekt wurde während des SS 2012 und WS 2012/13 durchgeführt. Ziel dieses Projektes war die Spezifikation und Implementieung einer virtuellen Maschine namens UMach. Dazu gehören ein Assembler und ein Debugger für diese Maschine. Der gewünschte Workflow bei der Fertigstellung des Projektes war der folgende:

1. Der Benutzer schreibt eine Assembler-Datei. Diese enthält Anweisungen an die UMach-Maschine wie die folgenden:

   ```
   SET  R2 0x1A
   MULI R2 4
   CP   R2 LO
   SET  R1 0
   SW   R1 R2
   ```

   Wir nehmen an, die Datei heißt „test.uasm“.

2. Diese Datei wird in eine Bytecode-Datei assembliert mit dem folgenden Befehl:

   ```
   ./uasm -o test.umx test.uasm
   ```

   Ergebnis ist eine Bytecode-Datei namens „test.umx“. Diese Bytecode-Datei enthält UMach-Instruktionen in binärer Form. Die oben angegebenen Instruktionen werden wie folgt assembliert:

   ```
   0x10 0x02 0x00 0x1A
   0x3A 0x02 0x00 0x04
   0x11 0x02 0x2B 0x00
   0x10 0x01 0x00 0x00
   0x15 0x01 0x02 0x00
   ```

3. Die Bytecode-Datei wird von der UMach-Maschine ausgeführt mit dem folgenden Befehl:

   ```
   ./umach test.umx
   ```

   Alternativ kann man die Maschine in Debugg-Modus starten mit dem Befehl

   ```
   ./umach -d test.umx
   ```

   Alternativ steht ein getrennter Qt-Debugger zur Verfügung.

Diesen Workflow betrachten wir als implementiert. Alle diese Schritte sind möglich und liefern die gewünschten Ergebnisse. Das Projekt enthält auch eine Reihe von fertiggeschriebenen Assembler-Programmen, die gleich getestet werden können.

1.1 Teilaufgaben

Im Rahmen dieses Projektes wurden die folgenden Teilaufgaben übernommen und gelöst.

1. Konzeptioneller Entwurf der virtuellen Maschine.
2. Spezifikation der virtuellen Maschine als PDF Dokument (Liviu Beraru).
3. Implementieung der Maschine in C99 für Linux (Liviu Beraru).
4. Assembler für die Maschine in C99 für Linux (Werner Linne).
5. Qt-Debugger (Simon Beer).
6. Demonstrationsprogramme in der UMach Assemblersprache `uasm` (Willi Fink).
7. Anweisungen zur Verwendung der UMach virtuellen Maschine, als PDF Dokument.

Die späteren Abschnitte gehen auf die Lösungswege der einzelnen Teilaufgabe ein. Jeder Abschnitt wurde vom zuständigen Gruppenmiglied geschrieben.

1.2 Organisation

Die Kommunikation unter den Gruppenmiglieder lief über Email, persönlichen Gespräche, Telefonaten und TODO-Dateien. Für die Verwaltung aller LaTeX und C Dateien wurde das Versionsverwaltungssystem GIT verwendet. Für die zentralle Verwaltung aller Dateien wurde ein GIT-Account auf Github eingerichtet und dort ein Projekt angelegt. Das Projekt ist öffentlich zugänglich, Schreiberechte aber haben nur die Gruppenmiglieder. Das Projekt kann unter der Adresse

`https://github.com/Malkavian/UMachVM`

nachgeschlagen werden. Dort können alle Dateien heruntergeladen werden.

2 Spezifikation und Implementierung

Liviu Beraru

2.1 Spezifikation

Die Spezifikation der virtuellen Maschine UMach – dessen Name von „sat*u*rn *mach*ine“ kommt, da sie als großes entferntes Ziel wahrgenommen wurde – umfasst mehrere Themen, die in einem umfangreichen Dokument behandelt wurden. Das Dokument ist Teil der Abgabe und wird sowohl aus PDF-Dokument als auch schriftlich dem Projekt-Betreuer übergeben. Die PDF-Datei heißt „UMachVM-Spec.pdf“. Der Titel lautet „UMach Spezifikation“. Die Themen dieses Dokumentes werden im folgenden kurz vorgestellt.

Architektur Die Architektur der UMach Maschine orientiert sich stark an der RISC Architektur. Sie ist registerbasiert und hat eine feste Instruktionslänge von 4 Byte. Die Byte-Reihenfolge ist „little endian“. Die UMach Maschine verwendet Port I/O. Es werden 32 Allzweckregister und 13 Spezialregister definiert. Bestimmte Spezialregister sind schreibgeschützt.

Instruktionssatz Es wurden für die UMach-Maschine 69 Instruktionen festgelegt. Für jede Instruktion wird eine Befehlsnummer, einen Assemblernamen, die Argumente und ein Instruktionsformat angegeben. Zudem werden für die meisten Instruktionen Verwendungsbeispiele und Fehlerquellen gegeben. Die Instruktionen werden in mehreren Kategorien unterteilt: arithmetische, logische, Speicher-, Vergleichs-, Sprunginstruktionen usw. Die Spezifikation wird nach diesen Kategorien strukturiert.

Jede Instruktion besteht aus einer Befehlsnummer im ersten Byte und aus Befehlsargumenten in den folgenden 3 Bytes. Zur Interpretieung der Instruktionsargumente wurden Instruktionsformate definiert. Gemäß dieser Formate, können die 3 Argumentenbytes entweder als Registernummer, oder direkte nummerische Angaben interpretiert werden, die 1, 2 oder 3 Bytes groß sind.

Speichermodell Die UMach Maschine verwendet nur absoluten Speicheradressen. Der gesamte Adressraum wird für den Speicher verwendet (kein Mapping für I/O-Ports).

Der Speicher wird in 5 Segmenten eingeteilt: Interrupttabelle, Code-Segment, Datensegment, Heap und Stack. Zur Kennzeichnung der Segmentgrenzen dienen Registerwerte. Ausnahme macht die Interrupttabelle, die die Adressen 0 bis 255 belegt.

Lese- und Schreibeoperationen werden durch geeignete *load*-, *store*-, *push*- und *pop*-Instruktionen ausgeführt. Zudem können die Grenzen der Speichersegmente durch Änderung bestimmter Registerwerte verändert werden, insbesondere durch die Register `HE` (heap end) und `SP` (stack pointer).

I/O Modell Die UMach Maschine verwendet Port I/O, als Gegenteil zu Memory Mapped I/O. Die Spezifikation beschreibt die Struktur der I/O-Einheit, die für die Eingabe und Ausgabe verantwortlich ist. Sie beinhaltet 8 Ausgabeports und 8 Eingabeports. Für Eingabe und Ausgabe werden spezielle `IN` und `OUT` Instruktionen zur Verfügung gestellt. Die aktuelle Implementierung der UMach Maschine bindet alle Eingabeports an die standard Eingabe (`stdin`) und alle Ausgabeports an die standard Ausgabe (`stdout`).

Interruptmodell Für die UMach Maschine wurde ein Interrupt-Mechanismus konzipiert und entwickelt. Die Spezifikation beschreibt diesen Mechanismus im Detail. Interrupts sind Signale, die entweder von der Maschine selbst oder vom Programmierer generiert werden können. Jedem Interrupt wird eine Interruptnummer vergeben. Von der Maschine werden sie in Fehlerfälle, vom Programmierer werden die mit der Instruktion `INT` generiert. Ein Interrupt kann abgefangen werden indem in der Interrupttabelle die Adresse einer entsprechenden Routine eingetragen wird.

2.2 Implementierung

Die UMach Maschine wurde in C99 implementiert. Das Programm heißt `umach` und wurde für Linux bzw. POSIX geschrieben. Beim Start dieses Programms können verschiedene Argumente übergeben werden. Siehe dazu das Dokument UMachVM-Verwendung.pdf.

Das `umach` Programm besteht aus mehreren Komponenten, die in der Abbildung 1 auf Seite 7 dargestellt sind. Diese Komponenten sind:

1. Maschine: Funktionen und Datenstrukturen, die den Kern, Speicher und I/O Einheit der Maschine realisieren. Die I/O Einheit ist nicht in einer Code-Datei vorhanden, sondern als Implementierung der I/O-Instruktionen. Eine zukünftige Version der UMach-Maschine sollte diese Einheit getrennt implementieren.

2. Disassembler: Ausgabe eines Bytecode-Datei in Assembler-Schreibweise.

3. Debugger: eingebauter einfacher Debugger.

Der Abschnitt 2.2.4 auf Seite 9 stellt die C-Dateien vor, die diese Komponenten ausmachen.

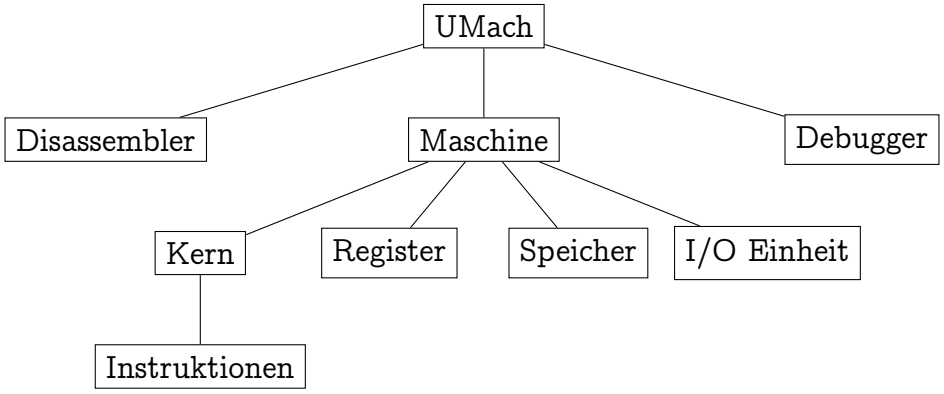

Abbildung 1: Komponenten-Struktur des `umach` Programms

2.2.1 Grundablauf

Die `main`-Funktion befindet sich in der Datei umach.c. Das Programm fängt gewöhnlich damit an, dass der UMach-Speicher gemäß Programmoptionen initialisiert wird. Dabei wird die Programmdatei, die als Programmargument übergeben wird, in den Code-Segment geladen und entsprechend die Segment-Register initialisiert. Danach wird aus `main` in das Modul `core` gesprungen (Datei core.c, Funktion `core_run_program`) und dort innerhalb einer Schleife alle Instruktionen aus dem Code-Segment ausgeführt. Das Programm stoppt in folgenden Fällen:

- Der Wert des Registers `PC` zeigt in den Datensegment (überschreitet den Code-Segment).

- Die Instruktion `EOP` (end of programm) wurde ausgeführt.

- Ein Interrupt wurde generiert, der von keiner Subroutine abgefangen wird.

Die Ausführung jeder Instruktion hat zwei Schritte, die sich am Von Neumann Zyklus orientieren:

1. fetch: die nächste Instruktion wir aus dem Code-Segment in ein globales 4-bytiges Array geladen.

2. execute: die jenige Funktion, die der Befehlsnummer im Byte 0 der Instruktion entspricht wird ausgeführt. Diese Funktion liest die Argumente aus dem globalen Array wo die Instruktion geladen wurde. Anschließend wird der Wert des PC Registers mit 4 inkrementiert.

2.2.2 Sprungtabelle

Ein wichtiges Konzept in der Implementierung der UMach Maschine ist die sogenannte Sprungtabelle. Sie besteht aus einem Array von Strukturen, die nach einem Schlüssel indiziert wird. Sie ist eine einfache Hashtabelle und wird verwendet, um die jenige Funktion zu finden, die eine bestimmte Instruktion implementiert. Schlüssel ist die Befehlsnummer der Instruktion.

Um das Konzept deutlicher zu machen, betrachte man als Beispiel die folgende Struktur:

```
typedef struct command
{
    int (*execute) (void);
}
command;
```

Sie besteht aus einem einzigen Feld: ein Funktionszeiger namens `execute`. Die gezeigte Funktion hat eine bestimmte Signatur: sie nimmt keine Argumente und gibt ein Integer zurück. Nun kann man eine Sprungtabelle wie folgt definieren:

```
command opmap[OPMAX] =
{
  [0x00] = { core_nop },
  [0x04] = { core_eop },
  [0x10] = { core_set },
  [0x11] = { core_cp  },
  [0x12] = { core_lb  },
  [0x13] = { core_lw  },
  [0x14] = { core_sb  },
  [0x15] = { core_sw  },
  ...
  [0xB8] = { core_out }
}
```

Alle Funktionsnamen auf der rechten Seite sind Namen von Funktionen, die wir implementiert haben. Die ausdrückliche Index-Zuweisung gehört zum C99-Standard und war einer der Gründe, warum C99 für dieses Projekt ausgewählt wurde. Möchte man

jetzt die Instruktion mit Nummer `0x13` ausführen, so schaut man in dieser Tabelle (Array) am Index `0x13` nach und falls der Eintrag nicht Null ist, führt man die Funktion aus:

```
command cmd = opmap[0x13];
if (cmd.execute) {
    cmd.execute();
}
```

Nach diesem Prinzip funktioniert der Kern der UMach Maschine, bzw. die Funktion `core_execute`, die den „execute"-Schritt implementiert und in der Datei core.c definiert ist. Die `command`-Struktur, die hier vereinfacht wurde, wird in der Datei command.h definiert, die Sprungtabelle selbst in command.c. Dort enthält die `command`-Struktur Felder für das Instruktionsformat, Name der Instruktion etc, die vom Assembler und Disassembler verwendet werden.

Diese Sprungtabelle ist eine Art, den „command pattern" in C zu implementieren.

2.2.3 Registertabelle

Für die Register der Maschine wurde ein ähnliches Konzept wie für die Sprungtabelle der Instruktionsfunktionen verwendet: eine Tabelle (Array) von Strukturen, die nach Registernummern indiziert wird. Die entsprechende Struktur `register` wird in der Datei register.h definiert und enthält Felder für den Registerwert, Zugriffsrechte und für den Registernamen, der vom Disassembler gebraucht wird. Um den Wert eines Registers mit Nummer x zu setzen, adressiert man die Tabelle an Index x und setzt das entsprechende Feld.

2.2.4 Übersicht der C Dateien

Die Implementierung erstreckt sich über 38 Dateien, die man wie folgt gruppieren kann:

Main Die `main` Funktion befindet sich in umach.c. Hier werden Programmargumente eingelesen, der UMach-Speicher initialisiert und das Programm in den Code-Segment geladen. Dann wird zum Modul `core` gesprungen und das Programm ausgeführt.

Kern Die Funktionen, die zum Kern (`core`) der UMach-Maschine gehören, sind in der Datei core.c implementiert und in der Headerdatei core.h deklariert. Dazu zählen Funktionen, die die Schritten fetch und execute implementieren. Hier findet man auch das globale Array `instruction`, das die aktuelle Instruktion enthält und den Integer `running`, der 0 wird, wenn die Maschine hält, sonst immer 1 ist. Die Funktion `core_run_program` führt die Schritte fetch und execute solange aus, bis die Variable `running` 0 wird.

Speicher Der Speichermodull (Dateien memory.c und memory.h) enthält Funktionen zum Speicher initialisieren und freigeben, Speicher lesen und schreiben, Laden der Programmdatei in den Code-Segment, und Funktionen, die die Befehle `PUSH` und `POP` realisieren.

Register Die Dateien register.c und register.h enthalten die Registertabelle und Funktionen zum Lesen und Schreiben der Registerinhalte und zum Setzen einzelner Bits in Registern. Zusätzlich werden Registernummern definiert.

Instruktionen Die UMach-Instruktionen werden in Kategorien unterteilt: arithmetische Instruktionen, logische Instruktionen usw. Diese Kategorien werden in der Spezifikation eingeführt. Für jede Kategorie von Instruktionen, wird eine getrennte C-Datei und Headerdatei verwendet. In dieser Datei werden alle Instruktionen implementiert, die der entsprechenden Kategorie gehören. Alle diese Funktionen wurden nach demselben Muster benannt: „core_x“. Wobei x ist der Assemblername der Instruktion, so wie in der Spezifikation festgelegt. Beispiele wären „core_add“, „core_sub“, „core_jmp“ usw.

Möchte man also den Quellcode einer bestimmten Instruktion ι finden, so schaut man in der Spezifikation, zu welcher Kategorie κ gehört die Instruktion ι. Dann schaut man in der C-Datei zur Kategorie κ nach einer Funktion mit dem Namen „core_ι“. Zum Beispiel die Instruktion `ADD` ist in der Kategorie „Arithmetische Instruktionen“, also schaut man in der Datei arithm.c nach einer Funktion core_add.

Hier eine Liste der Dateien:

C-Datei	Kategorie von Instruktionen
controll.c	Kontrollinstruktionen
loadstore.c	Lade- und Speicherbefehle
arithm.c	Arithmetische Instruktionen
logic.c	Logische Instruktionen
compare.c	Vergleichsinstruktionen
branch.c	Sprungbefehle
subroutine.c	Unterprogramminstruktionen
system.c	Systeminstruktionen
io.c	IO Instruktionen

Zu jeder C Datei gehört eine Headerdatei gleichen Namens, die hier nicht mehr gezeigt wurde. Die Wörter „Instruktion“ und „Befehl“ werden in der Benennung der einzelnen Kategorien gleichgesetzt.

Interruptnummern Die Interruptnummern sind in der Headerdatei interrupts.h deklariert.

Disassembler Die Dateien disassemble.c und disassemble.h enthalten die Implementierung eines einfachen Disassemblers. Der Disassembler liest eine Datei Instruktion für Instruktion aus, disassembliert jede davon und gibt sie auf die standard Ausgabe aus. Der Disassembler stoppt wenn er entweder die Datei bis zu Ende gelesen hat, oder wenn er eine spezielle Marke (Bytemuster) einliest, die den Start des Datensegments signalisiert. Diese Marke wird vom Assembler am Ende des Code-Segments geschrieben und wird auch vom Speicher-Modull verwendet, um das Ende dieses Segments zu erkennen.

Debugger Die UMach Maschine besitzt einen eingebauten einfachen Debugger, der in den Dateien debugger.c und debugger.h implementiert wird. Der Debugger zeigt immer eine disassemblierte Instruktion und wartet auf einen Befehl vom Benutzer. Nach der Ausführung einer Funktion wartet zeigt er die nächste Instruktion und wartet auf einen neuen Befehl. Der Debugger benutzt den Disassembler-Modull um die einzelnen Instruktionen zu disassemblieren. Seine Verwendung wird in der Dokumentationsdatei UMachVM-Verwendung.pdf beschrieben.

Optionen Die Dateien options.h und options.c deklarieren eine Optionenstruktur, die die Programmoptionen enthält. Sinn davon ist, dass diese Optionen an mehreren Stellen in Programm verwendet werden, was eine globale Deklaration notwendig macht.

Die Optionen werden in umach.c gesetzt, wenn die Programmargumente ausgelesen werden.

Logging Das Modul `logmsg` (Dateien logmsg.c und logmsg.h) enthält ein einfaches Mechanismus zur bedingten Ausgabe von Nachrichten. Jede Nachricht wird dem Modul mit einem „level“ übergeben. Je nach Wert des Feldes `verbose` in der Optionenstruktur, wird die Nachricht ausgegeben oder nicht.

Utils Die Datei strings.c enthält ein paar nützliche Funktionen zur Bearbeitung von Zeichenketten: zersplittern, Leerzeichen überspringen und Zahlen parsen.

Makefile Das Makefile dient zur Kompilierung des `umach` Programms. Man kann das Programm kompilieren, indem man in das Quellenverzeichniss wechselt und dort `make` Eingibt.

3 Assembler

Werner Linne

3.1 Zielsetzung

Aufgabe des Assemblers Der *uasm* Assembler übersetzt Quelltext in UMach Bytecode und erstellt Debuginformationen.

Erwünschte Eigenschaften Die aus meiner Sicht wichtigsten Eigenschaften des Assemblers sind:

- **Effektive Syntax:** Die Syntax des UMach Assemblers soll verständlich und komfortabel sein. Dadurch wird das Codieren eines UMach Programms erleichtert.
- **Performance:** Der Assembler soll auch größere Mengen an Quelltext mit wenig CPU-Zeit und wenig Arbeitsspeicherbedarf übersetzten können.
- **Aussagekräftige Fehlermeldungen:** Bei syntaktischen Fehlern im Quelltext soll der Benutzer möglichst genau über Art und Position des gefundenen Fehlers informiert werden.
- **Nützliche Debuginformationen:** Der Assembler generiert bei der Übersetzung des Quelltextes (optional) Debuginformationen. Art und Format dieser Informationen wurden maßgeschneidert auf die Bedürfnisse des UMach Debuggers festgelegt.

3.2 Bedienung

Der Assembler wird über die Befehlszeile bedient. Die Aufrufsyntax ist folgendermaßen definiert:

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

Die Bedeutung der einzelnen Elemente wird im Folgenden genauer erklärt:

Das *-o* Flag ist optional und benötigt, falls es gesetzt wurde, genau ein Argument (*outfile*). Das Argument *outfile* steht für den Namen der Datei, in welche der vom

Assembler erzeugte UMach Bytecode geschrieben wird. Ist das *-o* Flag nicht gesetzt, wird die Datei namens *u.out* verwendet.

Das *-g* Flag steuert die Generierung von Debuginformationen. Ist das Flag gesetzt, werden Debuginformationen generiert, ansonsten nicht.

Das *-w* Flag veranlasst den Assembler im Fehlerfall erst dann zu Terminieren, nachdem der Benutzer die Eingabetaste gedrückt hat. Dieses Flag wird hauptsächlich von dem Debugger genutzt.

Das Argument *file(s)* steht für ein oder mehrere Dateien und wir zwingend benötigt. Der Assembler übersetzt alle in *file(s)* genannten Quelltextdateien in den UMach Bytecode.

3.3 Syntax

Die Syntax von UMach Quelltextdateien ist relativ einfach gestaltet. Wir unterscheiden zwischen den Abschnitten *Code Section* und *Data Section*. Jede Quelltextdatei beginnt implizit mit der *Code Section*. Die *Code Section* beinhaltet Befehle, Sprungmarken, Kommentare und Leerzeilen. Die *Data Section* ist optional und wird durch die Textzeile "*.data*" eröffnet. Die *Data Section* beinhaltet Definitionen von Variablen, Kommentare sowie Leerzeilen.

Zur Veranschaulichung folgt ein fiktives Beispiel einer UMach Quelltextdatei:

```
1  SET R1 hello
2  myloop:
3      CALL println #this function is implemented somewhere else
4      DEC R2
5      CMP R2 ZERO
6  BNE myloop
7  #lines containing only a comment or nil are ignored
8  SET R1 9001
9  EOP
10
11 .data #begin of data definitions
12 .string hello  "Hello World!\n" #C-style escaping in strings is OK
13 .int    answer 42
14 .int    drink  0xCAFE
```

In diesem Beispiel bilden die Zeilen 1 bis 10 die *Code Section* und die *Data Section* wird in Zeile 11 eingeleitet und beinhaltet Zeile 12 bis 14.

Zu beachten sind folgende syntaktischen Regeln:

- Es sind beliebig viele Whitespaces (‘\t’ und ‘␣’) vor und nach Tokens zulässig
- Nur Symbolbezeichner (Sprungmarken und Definitionen von Variablen) sind case-sensitiv
- Symbolbezeichner dürfen keine Zahlenwerte sein
- Symbolbezeichner bestehen aus genau einem Wort
- Die Definition einer Sprungmarke endet mit ‘:’ und steht in einer eigenen Zeile im Quelltext
- Für alle Dateien gilt der gleiche Namensraum, d.h. namensgleiche Symbolbezeichner in verschiedenen Quelltextdateien sind nicht erlaubt
- Ab ‘#’ beginnt ein Kommentar bis einschließlich ‘\n’

3.4 Implementierung

Der *uasm* Assembler ist ein 2-pass Assembler, welcher in der Programmiersprache C99 implementiert ist. Das Programm benötigt lediglich die GNU Standard C Library (*glibc*) sowie die GNOME *GLib* zur Compile- und Laufzeit. Der *uasm* Assembler ist ausgelegt und getestet für die Nutzung auf 32- & 64-bit Systemen.

Für die Erzeugung des Programms *uasm* benötigt man einen C99 konformen Compiler (bspw. aktuelle Versionen von *GCC* oder *clang*), das Programm *GNU make* und die oben genannten Libraries.

3.4.1 Interne Datenstrukturen

Eine Hauptaufgabe des Assemblers ist die Übersetzung von Zeichenketten in die jeweilige UMach Bytecode Repräsentation. Ein solcher Übersetzungsvorgang kann mithilfe von Hashtables fast immer in $O(1)$ ausgeführt werden. Bei der Implementierung wurde die Datenstruktur *GHashTable* der *GLib* verwendet.

Insgesamt kommen drei Hashtables zum Einsatz:

1. Eine statische Hashtable zur Übersetzung von Befehlen

2. Eine statische Hashtable zur Übersetzung von Registern

3. Eine dynamische Hashtable zur Auflösung von Symbolbezeichner

Bei der Speicherung der Initialisierungswerten von Variablen wird eine einfach verkettete Liste eingesetzt (*GSList* der *GLib*). Hierbei ist zu beachten, dass nur performante Funktionen wie "Einfügen am Anfang" sowie "Iteration über alle Elemente" benötigt werden. Der Vorteil beim Einsatz der verketteten Liste ist, dass die Funktion "Iteration über alle Elemente" in $O(n)$ realisierbar ist. Bei Verwendung von *GHashTable* würde der Aufwand für diese Operation hingegen in $O(n^2)$ liegen.

3.4.2 Interne Datentypen

In diesem Abschnitt werden die wichtigsten internen Datentypen des *uasm* Assemblers kurz vorgestellt.

Befehle Befehle werden auf den Datentyp `command_t` abgebildet und bestehen aus vier Attributen.

opcode Der OpCode des Befehls entsprechend der UMach Spezifikation

opname Der Name des Befehls entsprechend der UMach Spezifikation

format Das Befehlsformat entsprechend der UMach Spezifikation

has_label Ein Flag das anzeigt, ob der Befehl mit Sprungmarken oder Variablen verwendet werden kann

Der `command_t` Datentyp ist im C-Quelltext folgendermaßen definiert:

```
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 Register werden auf den Datentyp `register_t` abgebildet und bestehen aus zwei Attributen.

regcode Die Nummer des Registers entsprechend der UMach Spezifikation

regname Der Name des Registers entsprechend der UMach Spezifikation

Der `register_t` Datentyp ist im C-Quelltext folgendermaßen definiert:

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

Symbole Symbole werden auf den Datentyp `symbol_t` abgebildet und bestehen aus drei Attributen.

sym_name Der vom Benutzer gewählte Name eines Symbols

sym_type Die Art des Symbols (Sprungmarke, Integer Variable, String Variable)

sym_addr Die Zieladresse, auf die ein Symbol verweist

Der `symbol_t` Datentyp ist im C-Quelltext folgendermaßen definiert:

```
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;
```

3.4.3 Assemblierung

Der Assembler parst jede Quelltextdatei zweimal (2-pass Assembler). Mithilfe der im ersten Durchgang gesammelten Informationen ist es dem Assembler möglich, im zweiten Durchgang den Quelltext Zeile für Zeile zu übersetzen, ohne an manchen Stellen "vorauslesen" zu müssen.

Hier eine genauere Beschreibung der beiden Durchgänge:

Assembler Pass 1 Im Pass 1 werden folgende Aufgaben erledigt:

- Sprungmarken im Quelltext werden erkannt und deren Namen sowie deren Zieladressen in die Symboltabelle eingetragen.
- Die Codegröße (Summe des Speicherplatzbedarfs aller Befehle) wird berechnet.
- Die Initialisierungswerte von Variablen werden zwischengespeichert.
- Die Adressen von Variablen werden berechnet.
- Namen und Adressen von Variablen werden in die Symboltabelle eingetragen.

Assembler Pass 2 Im Pass 2 werden folgende Aufgaben erledigt:

- Der UMach Bytecode wird generiert und in das *outfile* geschrieben.
- Die (optionale) Generierung der Debuginformationen.
- Initialisierungswerte von Variablen werden in das *outfile* geschrieben.

3.4.4 Performance

Durch die sorgfältige Auswahl der Datenstrukturen und dem Fokus auf performanten C-Quelltext zeigt der *uasm* Assembler gute Laufzeiteigenschaften. Die Auflösung von Symbolen erfolgt (mit Ausnahme des Sonderfalls von Kollisionen in der Hashtable) unabhängig von der Anzahl der gespeicherten Symbolen in konstanter Zeit, sprich $O(1)$. Des Weiteren wächst die Gesamtlaufzeit einer Assemblierung praktisch linear mit der Anzahl der zu assemblierenden Quelltextzeilen, sprich $O(n)$. Der Arbeitsspeicherbedarf wächst linear mit der Anzahl von definierten Symbolen.

In absoluten Zahlen bedeutet dies einen Durchsatz von ca. $1.4 \times 10^6 \frac{Zeilen}{Sekunde}$, gemessen auf einem AMD Athlon II X2 250 System mit 3 GHz.

3.4.5 Fehlerdiagnose

Falls der vom Benutzer geschriebene Assemblercode syntaktische Fehler enthält, wird eine Fehlermeldung ausgegeben und die Assemblierung abgebrochen.

Einige Fehlersituationen werden hier anhand von Beispielen genauer diskutiert:

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

Bsp. 1: Ein unbekannter Befehl "SQRT" wurde verwendet.

Bsp. 2: Die Anzahl der Operanden für den Befehl "CMP" ist ungültig.

Bsp. 3: Es wurde versucht, an die unbekannte Sprungmarke "getinput" zu springen.

Bsp. 4: Ein unbekanntes Register "R77" wurde als Operand verwendet.

Bsp. 5: Die Sprungmarke "get_input" wurde ein zweites mal definiert.

Bsp. 6: Bei der Definition der Variable "myint" wurde kein Initialisierungswert angegeben.

3.4.6 Quelltextstruktur

Der *uasm* Quelltext ist in insgesamt 8 C-Dateien zuzüglich deren Header-Files aufgeteilt. In der folgenden Tabelle sind die Module genauer spezifiziert.

C-Datei	Funktionalität
uasm.c	*main()* Funktion und Benutzerschnittstelle
assemble.c	Hauptmodul für das Assemblieren
asm_formats.c	Modul für das Parsen der verschiedenen Befehlsformate
symbols.c	Modul für die Verwaltung der Symboltabelle
collect_data.c	Modul für die Verwaltung der Inhalte von Variablen
commands.c	Modul für die Zuordnung Befehlsname ⇒ `command_t`
registers.c	Modul für die Zuordnung Registername ⇒ `register_t`
str_func.c	Hilfsfunktionen zur Manipulation von Zeichenketten

3.5 Debuginformationen

Die Debuginformationen ermöglichen dem Debugger das Setzen von Haltepunkten auf Sprungmarken sowie auf beliebige Zeilen mit Befehlen im Assemblercode. Außerdem können anhand der Debuginformationen Inhalte von Variablen ausgelesen und verändert werden.

Ist das generieren von Debuginformationen aktiviert, werden die Dateien *outfile*.`fmap`, *outfile*.`sym` und *outfile*.`debug` erstellt. Deren Bedeutung wird in den folgenden Abschnitten beschrieben.

3.5.1 File Map

Die sogenannte File Map wird in die Datei *outfile*.`fmap` geschrieben. Sie dient der Indizierung jeder Assemblercode-Datei mit einer ID. Technisch gesehen handelt es sich um eine ASCII-Textdatei mit n Zeilen, in welchen jeweils eine 1:1 Relation (File-ID, File-Name) steht.

Beispiel:

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

In diesem Beispiel erhält also die Datei "tictactoe.uasm" die ID 0 und die Dateien mit Hilfsfunktionen erhalten die IDs 1 bis 4.

3.5.2 Debug File

Das Debug File wird in die Datei *outfile*.`debug` geschrieben. Es dient dazu, dem Debugger mitzuteilen, auf welche Adressen die Befehle aus den Assemblercode-Dateien abgebildet werden. Technisch gesehen handelt es sich um eine Binärdatei, die n 32Bit-Datentripel (File-ID, Line-No, Address) im Big-Endian Format enthält.

Beispiel (hexadezimale Notation):

```
00 00 00 00 | 00 00 00 05 | 00 00 01 00
00 00 00 00 | 00 00 00 08 | 00 00 01 04
00 00 00 00 | 00 00 00 09 | 00 00 01 08
00 00 00 00 | 00 00 00 0f | 00 00 01 14
00 00 00 00 | 00 00 00 15 | 00 00 01 1c
...
```

Die erste Zeile aus dem Beispiel sagt aus, dass der Befehl aus Zeile 5 der Assemblercode-Datei mit der ID 0 an der Adresse 256 im Arbeitsspeicher der UMach VM steht. Die weiteren Zeilen aus dem Beispiel kann man dem entsprechend interpretieren.

3.5.3 Symbol File

Das Symbol File wird in die Datei *outfile*.sym geschrieben. Es dient dazu, dem Debugger mitzuteilen, auf welche Adressen die Symbole aus den Assemblercode-Dateien verweisen. Technisch gesehen handelt es sich um eine Textdatei, die n Datentripel (Address, Symbol-Type, Symbol-Name) enthält.

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
```

Dieses Beispiel sagt unter Anderem aus, dass die Sprungmarke “start_inputint” auf den Befehl an der Adresse $000005e0_{16}$ im Arbeitsspeicher der UMach VM zeigt. Der Inhalt der String-Variable “promptdraw” ist beginnend an der Adresse 00000784_{16} gespeichert. Analog befindet sich an der Adresse 00000794_{16} der Inhalt der Integer-Variable “newln”.

4 Qt Debugger

Simon Beer

4.1 Einleitung

Zielsetzung Ziel der Entwicklung war ursprünglich nur die Bereitstellung eines graphischen Debuggers. Dabei sollen folgende Funktionalitäten mindestens unterstützt werden:

- Setzen von Haltepunkten, an denen die Maschine die Ausführung unterbrechen soll und auf eine Benutzereingabe wartet.
- Einzelschritt nach einem Haltepunkt durch die Instruktionsfolge
- Anzeigen der entsprechenden Codestelle bei Auftreten eines Haltepunktes oder bei Einzelschritt
- Inspizieren und Modifizieren der Register, Daten und des Maschinenzustandes.

Eine weitere Anforderung besteht darin, dass die graphische Oberfläche und der Maschinenkern als eigenständige Prozesse laufen sollen und die Kommunikation und Steuerung somit per Inter-Prozess-Kommunikation realisiert ist. Dies hat insofern den Vorteil, das der Maschinenkern nicht als Teil der GUI umgesetzt werden muss. Ändert sich z.B. etwas an der internen Maschinenkernimplementierung muss keine Änderung oder ein Neu-Kompilierung der graphischen Oberfläche vorgenommen werden.

Ein sekundäres Ziel besteht darin, den graphischen Debugger plattform-unabhängig zu gestalten, oder zumindest eine unproblematische Umsetzung für andere gängige Betriebssysteme (Windows, MacOS) abseits der verwendeten Entwicklungsplattform Linux zu ermöglichen.

Grundlegende Designentscheidungen Im Verlauf der Planung zeigte sich jedoch, dass alle für die Umsetzung der Zielstellung an den graphischen Debugger notwendigen Elemente im Ansatz schon zu einer rudimentären Entwicklungsumgebung geführt haben, vor allem die Tatsache, dass an Haltepunkten die entsprechende Zeile im Quelltext angezeigt werden sollte. Die dafür in *Qt* verwendeten graphischen Oberflächenelemente erlauben jedoch auch die Edition des angezeigten Text. Dieser Umstand hatte zur

Folge, dass der Debugger als eine minimalistische Entwicklungsumgebung implementiert wurde, welche hauptsächlich Funktionalitäten zum Debuggen der Programme bereitstellt.

Für die Verwendung von *Qt* sprach die damit zu erreichende Plattform-Unabhängigkeit, nicht nur in Anbetracht der graphischen Elemente sondern auch im Bereich der Inter-Prozess-Kommunikation. Hier stellt *Qt* eigene Klassen bereit, die auf allen Systemen gleichermaßen eingesetzt werden können, ohne dass in Abhängigkeit vom Betriebssystem spezifische Kenntnisse notwendig sind.

4.2 Debugging

Es gibt verschiedene Möglichkeiten, die Steuerung der Maschine beim Debuggen zu realisieren. Daher soll eine kleine Übersicht über mögliche Verfahren gegeben werden und eine Begründung für die Wahl des in der Realisierung des Debuggers verwendeten Verfahrens.

Haltepunkte Es gibt insgesamt zwei nennenswerte Verfahren um Haltepunkte zu realisieren. Eine Möglichkeit besteht darin, die entsprechende Instruktion, an der ein Halt der Maschine stattfinden soll, durch einen speziellen Interrupt zu ersetzen. Tritt dieser ein, wird dem Debugger dies durch die Maschine signalisiert. Diese Möglichkeit ist aber aufgrund verschiedener Aspekte nicht umgesetzt worden. Vor allem hätten dadurch Änderungen in der Maschinenspezifikation vorgenommen worden müssen, welche aber nicht erwünscht waren.

Daher kam letztendlich eine weitere Methode zur Anwendung: dem Abgleich der aktuellen Instruktions-Adresse. Hier wird die aktuelle Adresse mit einer Liste von Adressen verglichen, an denen ein Halten der Maschine erwünscht ist. Dies kann entweder durch die Maschine selbst geschehen oder direkt durch den Debugger. Letzteres wurde schließlich auch zur Umsetzung realisiert. Dabei wird nach jeder Instruktions-Ausführung die aktuelle Adresse vom Debugger mit Hilfe der Inter-Prozess-Kommunikation aus der Maschine ausgelesen und abgeglichen.

Adresstabelle Als Haltepunkte können vom Benutzer entweder Zeilennummern oder Sprunglabels angegeben werden. Um diese den entsprechenden Instruktions-Adressen zuordnen zu können, wird vom Assembler beim Assemblieren eine entsprechende Datei mit einer Tabelle erzeugt, welche beim Starten des Debugging-Vorganges durch den Debugger eingelesen wird. Da der Abgleich zwischen Zeilennummer und Sprunglabel erst beim Start abgeglichen werden kann, und in der GUI keine zusätzliche Prüfung

vorhanden ist, können vom Benutzer ungültige Zeilen oder Labels angegeben werden. Diese werden beim Debugging-Vorgang ignoriert.

Einzelschritt Standardmäßig läuft die Maschine, wenn sie durch den Debugger kontrolliert wird, in einer Art Einzelschrittmodus, um dem Debugger die Möglichkeit zu geben, an bestimmten definierten Zeitpunkten die Maschine zu manipulieren um z.B. die aktuelle Instruktions-Adresse auslesen zu können.

In der Regel bleibt dies jedoch vom Benutzer unbemerkt, da nur auf direkten Wunsch des Benutzers dieser automatische Zyklus zwischen Debugger und Maschine unterbrochen und auf Eingaben vom Benutzer gewartet wird. Dies ist nach Haltepunkten oder einem vom Benutzer explizit angestoßenen Einzelschritt der Fall. Auf die explizite Implementierung dieses Zyklus wird im folgendem Kapitel *Inter-Prozess-Kommunikation* im Detail eingegangen.

4.3 Inter-Prozess-Kommunikation

Die Steuerung der Maschine und der Datenaustausch zwischen Maschine und Debugger wurde, wie schon erwähnt, durch Inter-Prozess-Kommunikation realisiert. Im diesem Kapitel soll im Detail darauf eingegangen werden, welche Mittel dazu verwendet worden sind, um dies zu realisieren. Weiter soll auch der Ablauf der Kommunikation im Detail geschildert wird.

QSystemSemaphore Bei der *QSystemSemaphore* handelt es sich um eine von der *Qt*-Bibliothek bereitgestellte Semaphore. Eine Semaphore ist ein Ressourcenzähler, der prozess-übergreifend eingesetzt werden kann. Dieser kann mit einem beliebigen Startwert initialisiert werden. Weiterhin werden zwei Funktionen bereit gestellt, mit denen Ressourcen angefordert oder freigegeben werden können.

Die Besonderheit liegt hier darin, dass, wenn nicht mehr die Angeforderte Anzahl von Ressource verfügbar sind, d.h. der Ressourcenzähler einen Stand von null oder weniger erreichen würde, der anfordernde Prozess solange blockiert wird, bis ein weiterer Prozess ausreichend zusätzliche Ressourcen freigibt. Wird immer nur jeweils eine Ressource freigegeben oder angefordert, so kann das Semaphore als ein Mutex verwendet werden, um die Abläufe von zwei Prozessen miteinander zu synchronisieren.

Angelegt werden *QSystemSemaphore* mit einem Konstruktor, der gleichzeitig eine eindeutige ID als String erwartet, sodass diese von verschiedenen Prozessen aus identifiziert und drauf zugegriffen werden kann. Ist eine *QSystemSemaphore* bereits an-

gelegt, so wird durch jeden weiteren Konstruktor-Aufruf mit Angabe der gleichen ID nur eine Referenz auf die bestehende Semaphore zurückgegeben.

QSharedMemory Beim *QSharedMemory* handelt es sich um einen eigenständigen Speicher, der den Zugriff aus verschiedenen Prozessen aus erlaubt, da in der Regel Prozesse nicht auf den anderer zugreifen können. Genau wie die *QSystemSemaphore* wird dieser beim ersten Konstruktor-Aufruf und der Vergabe einer eindeutigen ID erzeugt. Daraufhin kann mit einem Aufruf von *create()* der eigentliche Speicher unter Angabe einer Größe reserviert alloziert werden. Jeder weitere Konstruktor-Aufruf unter Angabe der gleichen ID liefert eine Referenz auf den bestehenden *QSharedMemory* zurück.

Um den so erzeugten Speicher in einem Prozess nutzbar machen, muss er mit dem Funktionsaufruf von *attach()* an den Prozess *angehängt* werden. Wird der Speicher nicht mehr benötigt, so kann er mir einem *detach()* wieder vom Prozess gelöst werden. Hierbei ist jedoch sicherzustellen, dass der Speicher erst komplett von allen Prozessen *abgehängt* wird, wenn er nicht mehr benötigt wird, da das Abhängen vom letzten Prozess automatisch den Destruktor aufruft und somit somit alle hinterlegten Daten verloren gehen.

Ein weiterer Aspekt, der beim Umgang mit *QSharedMemory* zu beachten ist, ist, dass wenn der Speicher nicht mehr benötigt wird, darauf geachtet werden muss, dass er von allen Prozessen abgehängt werden muss, damit die Freigabe sichergestellt wird. Dies hängt damit zusammen, dass unter Linux der Speicher das Prozessende aller zugreifenden Prozesse überlebt und als Speicherleiche bestehen bleibt. Besonders im Fehlerfall ist darauf zu achten.

Um den geordneten Zugriff zu gewährleisten und gleichzeitige Zugriffsversuche zu unterbinden, stellt *QSharedMemory* auch ein integriertes Mutex zur Verfügung, um den Speicher dagegen abzusichern.

Übersicht Zur Realisierung der *Inter-Prozess-Kommunikation* und zur Steuerung der Maschine durch den Debugger werden die oben beschriebenen *Qt*-Klassen *QSystemSemaphore* und *QSharedMemory* verwendet. Hierbei dienen die Semaphoren dazu, die Maschine durch den Debugger Steuern zu können, um so den Ablauf zu kontrollieren. Der *QSharedMemory* dient dazu, den Datenaustausch zwischen Debugger und Maschine zu realisieren. So werden Register- und Speicherinhalte durch die Maschine über den *QSharedMemory* dem Debugger zur Verfügung gestellt, damit dieser diesen anzeigen oder auch manipulieren kann. Auch werden über den gemeinsamen Speicher Daten zur Steuerung ausgetauscht, z.B. ob ein Abbruch durch den Benutzer vorliegt.

Realisiert ist dies durch ein *Struct*, welches in einer eigenen Header-Datei deklariert ist und sowohl vom Debugger als auch vom Kern eingebunden wird. Dies hat den Vorteil, dass die Größe eines *Structs* fest definiert ist und die benötigte Speichergröße beim Anlegen des *QSharedMemory* schon bekannt ist. Auch erlaubt dies einen einfachen Zugriff auf die Daten, da der *QSharedMemory* bei Zugriff nur einen *Void-Pointer* zurückliefert. Dieser muss nur durch einen *Cast* in einen *Pointer* auf das entsprechende *Struct* umgewandelt werden und der Zugriff auf alle Bestandteile ist möglich.

Kontrollzyklus Der Kontrollzyklus wurde mit vier *QSystemSemaphore* realisiert, die in diesem Fall in vereinfachter Verwendung als Mutex genutzt werden. Dies wird wie schon erläutert, dadurch erreicht, dass nur eine Ressource zur Verfügung gestellt wird. Der Gebrauch von *QSystemSemaphore* liegt auch darin begründet, dass von der Verwendeten *Qt*-Bibliothek kein prozess-übergreifendes Mutex zur Verfügung gestellt wird, aber dennoch für die Prozesssynchronisation nur *Qt*-Klassen verwendet werden sollten, um genannte Zielsetzungen wie Plattform-Unabhängigkeit nicht zu verletzen.

Betrachtet man den einen Zyklus des Maschinenkerns, so besteht dieser aus zwei grundlegenden Abläufen: Einem *Fetch*, gefolgt von einem *Execute*. Beide sollen nur auf expliziten Befehl des Debuggers ausgeführt werden. Dafür werden zwei der Semaphoren verwendet. Die Semaphoren werden mit null verfügbaren Ressourcen initialisiert. Bevor der Kern nun entweder das *Fetch* oder *Execute* ausführt, versucht er von der jeweiligen Semaphore eine Ressource anzufordern. Da in beiden Fällen noch keine Ressource vorhanden sind, müssen diese erst durch den Debugger freigegeben werden. Somit wird der Kern solange blockiert, bis dies geschieht. Aufgrund dessen ist es dem Debugger möglich, den genauen Zeitpunkt zu bestimmen, wann das *Fetch* oder *Execute* ausgeführt werden darf.

Die zwei weiteren Semaphoren dienen wiederum dazu, dem Kern die Möglichkeit zu geben, dem Debugger zu signalisieren, wann der *Fetch*- oder *Execute*-Vorgang beendet worden sind. In diesem Fall geschieht das Gegenteil wie oben und der Debugger fordert von den Semaphoren eine Ressource an. Ist der Kern mit seinen Aufgaben fertig, so gibt er diese Ressource frei, und der Debugger erkennt, dass der Vorgang abgeschlossen ist.

Weitere Aufgabe des Kerns ist es, vor einem *Fetch* oder *Execute* eventuell von dem Debugger manipulierte Daten aus dem *QSharedMemory* zu kopieren und danach die vom Kern veränderte Daten im *QSharedMemory* zu hinterlegen. Der Debugger hat so die Möglichkeiten, die Daten und Zustände der Maschine zu manipulieren, ohne direkten Zugriff darauf zu haben. Weiterhin erhält er so wichtige Informationen wie die aktuelle Instruktions-Adresse, anhand deren er z.B. entscheiden kann, ob ein Haltepunkt vorliegt. In diesem Fall kann er z.B. erst auf eine Benutzereingabe warten,

bis er anhand der Semaphore dem Kern die Freigabe zur weiteren Abarbeitung des geladenen Programmes gibt.

Dies geschieht zwischen *Fetch* und vor der nächsten Instruktions-Abarbeitung mittels *Execute*. Dadurch ist der als nächstes auszuführende Befehl anhand der Instruktions-Adresse dem Debugger bekannt, und anhand den vom Assembler erzeugten Zusatzinformationen auch die zugehörige Quelldatei und die Zeilennummer. So kann die entsprechende Codestelle vor deren Ausführung im Debugger angezeigt werden. Wird nun vom Benutzer ein Einzelschritt ausgeführt, so wird ihm nach dessen Ausführung die nächste auszuführende Instruktion angezeigt und der kann das Ergebnis der vorherigen Instruktions-Abarbeitung begutachten.

4.4 Qt

Wie schon häufiger erläutert, wurde für die Umsetzung des graphischen Debuggers auf die *Qt*-Bibliothek zurückgegriffen. Es sollen nun weitere Gründe für die Entscheidung abseits des schon genannten Vorteils der Plattform-Unabhängigkeit erläutert und auf weitere Eigenheiten von *Qt* eingegangen werden. Im Detail soll davon das *Signals & Slots* Prinzip von *Qt* erläutert werden, welches die Kommunikation zwischen den einzelnen Objekten realisiert.

Entscheidungsgründe für Qt Weiterer Grund für die Entscheidung für *Qt* waren die vorhandenen Erfahrungswerte in der Entwicklung mit *Qt*. Dies ist ein nicht zu unterschätzender Faktor, der deutliche Zeitersparnis bei der Entwicklung der Oberfläche mit sich gebracht hat. Dies ist auf zwei wesentliche Aspekte zurückzuführen. Einmal ist dies der Wegfall von einem gewissen Lernprozess, der immer notwendig ist, wenn sich in neue Konzepte eingearbeitet werden muss, wie z.B. einer Bibliothek zur Oberflächen-Programmierung. Weiterhin wird auch im Verlauf der Entwicklung Aufwand eingespart, da schon bekannte Problemlösungen angewendet und die Aneignung neuer entfällt.

Ein zusätzlicher Nutzen von *Qt* war in diesem Fall auch, dass *Qt* eine Bibliothek für die Programmiersprache *C++* ist. Dies hatte gerade den Vorteil, dass der Maschinenkern in *C* geschrieben ist. Da *C++* nativ das Einbinden von *C*-Quelltext erlaubt, konnte die "MainFunktion der Maschine unproblematisch um *C++* Anteile zur Inter-Prozess-Kommunikation erweitert werden.

Auch für die Verwendung von *Qt* sprach, dass *Qt* nicht nur eine Bibliothek zur reinen Oberflächenprogrammierung ist, sondern auch wie schon oben beschrieben, z.B. auch Klassen für die Inter-Prozess-Kommunikation mitbringt. Somit hat sich letztlich die

Entscheidung für *Qt* als sinnvoll erwiesen.

Signals & Slots Prinzip Beim *Signals & Slots* Prinzip handelt es sich um ein Konzept, um *Callback*-Funktionen zu verdecken, erweitern und deren Verwendung zu vereinfachen um Signale zwischen oder innerhalb von Objekten zu verschicken. In *Qt* dienen sie hauptsächlich in der Oberflächenprogrammierung dazu, Oberflächenereignisse weiterzuleiten und diese mit Funktionalität zu hinterlegen.

Das *Signals & Prinzips* umfasst fünf Komponenten:

- *Signals*, die zwischen den Objekten versendet werden.
- *Emit*, um ein Signal auszulösen.
- *Slot*, um Signale aufzunehmen und Funktionalität bereitzustellen.
- *Connect*, um das von einem Objekt gesendete Signal einem Slot zuzuweisen.
- *Dissconect*, um eine Signal-Slot Verknüpfung wieder aufzuheben.

Anhand eines Beispieles soll die Funktionsweise dieses Prinzip näher erläutert werden.

```
1  class  Debugger
2  {
3  signals:
4    void requestOpenFileInTab (IFile *file);
5  private:
6    void showCodeLine(int instructionAddress) {
7      ...
8      if (!asmFiles[i]->isOpen()) {
9        emit requestOpenFileInTab (asmFiles [i]);
10     ...
11   }
12 };
13
14 class CodeEditor
15 {
16 public slots:
17   void openFileInTab (IFile *file) {
18   ...
19   }
20 };
21
```

```
22 class UMachGui
23 {
24   UMachGui() {
25     mDebugger = new Debugger;
26     mCodeEditor = new CodeEditor;
27     connect (m_debugger, SIGNAL(requestOpenFileInTab(IFile)),
28              m_codeEditor, SLOT(openFileInTab(IFile)));
29   }
30 }
```

In diesem Beispiel wird mit Hilfe eines *connects()* das Signal *requestOpenFileInTab(IFile)* des Objekts *mDebugger* vom Typ Debugger mit dem Slot *openFileInTab(IFile)* des Objektes *mCodeEditor* vom Typ *CodeEditor* verbunden. Somit werden zukünftig alle Signale diesen Typs die vom Objekt *mDebugger* mit einem *Emit* emittiert werden, an den verbundenen Slot des Objektes *mCodeEditor* weitergeleitet.

Qt Meta-Object-Compiler Das *Signals & Slots* Beispiel zeigt auch, dass *Qt*-Schlüsselwörter einführt, welche von einem standardisierten *C++*-Compiler nicht verstanden werden. Um dieses Problem zu lösen, gibt es einen sogenannten *Meta-Object-Compiler*. Dieser wird vor dem eigentlichen Compilieren ausgeführt und wandelt den um *Qt*-Schlüsselwörter erweiterten Code in reinen *C++*-Code um.

4.5 GUI

Im letzten Kapitel soll eine kleine Übersicht über die Oberfläche und der von dieser verwendeten Projektdateien gegeben werden.

Bestandteile Die Software besteht aus einem Hauptfenster, welches einen Editor enthält und mehrere Dateien in Reiter geöffnet halten kann. Weiterhin steht eine Liste bereit, welche alle zu einem Projekt gehörigen Dateien auflistet. Alle dort aufgelisteten Dateien können im Editor geöffnet werden. Eine Toolbar gibt schnellen Zugriff auf häufig verwendete Funktionen, wie "Nächster Haltepunk", etc. Als extra Fenster gibt es eine Tabelle zum selektiven Anzeigen bestimmter Registerinhalte und ein Fenster zum Setzen von Haltepunkten anhand Zeilennummern oder Label. Auch ein Optionsfenster ist vorhanden, in dem momentan nur die Speichergröße der Maschine eingestellt werden kann. Eine Tabelle, welche die Daten der Variablen anzeigt und deren Manipulation erlaubt, ist im Hauptfenster untergebracht.

Projektdatei Die Notwendigkeit einer Projektdatei hat sich insofern gezeigt, da bei vorhandener Editierbarkeit des Quelltextes in der Entwicklungsumgebung auch ein *Build-Prozess* vor dem Ausführen notwendig wurde. Somit erfüllt diese momentan hauptsächlich die Aufgabe einer *Make*-Datei, in der alle notwendigen Quelldateien hinterlegt sind, da eine solche vom Assembler nicht direkt angeboten wird. In der Entwicklungsumgebung können neue oder bestehende Quelldateien einem Projekt hinzugefügt oder entfernt werden. Projektdateien können gespeichert, geladen oder neu angelegt werden. Aufbau der Projektdatei ist wie folgt: Diese besteht aus zweckspezifischen Abschnitten, die mit einem Marker eingeleitet werden, welcher mit einem *Punkt* beginnt, gefolgt von einem Schlüsselwort. Dies erlaubt zu einem späteren Zeitpunkt weitere Abschnitte einzuführen, in denen zusätzliche Informationen hinterlegt werden können, wie zum Beispiel gesetzte Haltepunkte, sodass diese beim Laden eines Projektes wieder automatisch zur Verfügung stehen.

5 Demo-Programme

Willi Fink

Um die Funktionalität der Virtuellen Maschine testen und demonstrieren zu können, mussten im Rahmen des Projekts Demo-Programme erstellt werden.

5.1 Meist verwendete Befehle

Die dabei am häufigsten verwendeten Befehle werden nun nähergebracht.

5.1.1 Wertzuweisung

`SET R1 5` oder `SET R1 label`

Setzt das Register R1 auf den angegebenen Wert. Labels werden durch Adressen ersetzt.

5.1.2 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

5.1.3 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`

5.1.4 Unbedingte Sprünge

- `JMP label`
  Sprung zu einem Label.

- `CALL funktion`
  Funktionsaufruf ähnelt dem JMP Befehl, mit dem Unterschied, dass nach der Ausführung der Funktion ins Hauptprogramm zurückgesprungen und der nächste Befehl ausgeführt wird.

5.1.5 IO-Befehle

- `IN R1 R2 ZERO`
  Liest `R2` viele Bytes vom Port `ZERO`(Konsole) und schreibt sie an die Adresse `R1`.

- `OUT R1 R2 ZERO`
  Schreibt `R2` viele Bytes aus dem Speicher, beginnend mit dem Byte an der Adresse `R1`, an den Port `ZERO` raus.

5.2 Hilfsfunktionen

Durch wiederkehrende Muster innerhalb der Demoprogramme, wurde schon früh die Notwendigkeit von Hilfsfunktionen deutlich. Folgende Hilfsfunktionen verrichten Aufgaben, die häufig bewältigt werden müssen:

- inputint
- printint
- putchar
- newline

Die Funktionen arbeiten nach dem Prinzip, dass der Callee die über den Stack überreichten Argumente hinter sich aufräumt.

5.2.1 inputint

Diese Hilfsfunktion liest 10 Zeichen aus der Eingabe der Konsole, wandelt diese in ein Integer um und pusht anschließend das Ergebnis auf den Stack. Dieses Ergebnis kann dann im Hauptprogramm mit einem PULL-Befehl in ein beliebiges Register gespeichert werden. Es werden nur 10 Zeichen gelesen, weil die größte Zahl, die ein Register darstellen kann, zehnstellig ist.

5.2.2 printint

Die über den Stack übergebene Zahl wird durch fortlaufende Divisionen und das Aufaddieren des ASCII-Wertes der Zahl 0 nach und nach in einen String umgewandelt. Jedes Zeichen wird auf den Stack gepusht. Sobald die Konvertierung fertig ist wird die "putchar"-Funktion entsprechend oft ausgeführt, um die Zeichen an die Konsole herauszuschreiben.

5.2.3 putchar

Das über den Stack übergebene Zeichen im ASCII-Format wird auf die Konsole herausgeschrieben.

5.2.4 newline

Diese Funktion schreibt das ASCII-Zeichen, das einen Zeilenumbruch symbolisiert, auf die Konsole heraus.

5.3 Die Demo-Programme

5.3.1 helloWorld.uasm

Schreibt ein "Hello!" an die Konsole raus.

5.3.2 echo.uasm

Ein Testprogramm das die folgende Zeile "Echo program. End with q" an die Konsole schreibt und durch die Eingabe des Zeichens 'q' beendet wirden kann.

5.3.3 99_bottles.uasm

Dieses Programm schreibt den Songtext des Lieds "99 Bottles of Beer" an die Konsole heraus. "99 Bottles of Beer" wurde in über 1500 verschiedenen Programmiersprachen als Ausgabe umgesetzt. Dieser Songtext und weitere Informationen sind auf der Webseite `www.99-bottles-of-beer.net` zu finden.

5.3.4 ggT.uasm

Der Benutzer gibt zwei Zahlen an, von diesen wird anschließend der größte gemeinsame Teiler mit Hilfe des euklidischen Algorithmus berechnet und ausgegeben.

5.3.5 fibonacci.uasm

Dieses Programm schreibt n Zahlen der Fibonacci-Folge, $X_n = X_{n-1} + X_{n-2}$ mit $X_1 = 1$ und $X_2 = 2$, an die Konsole heraus. Die Zahl n wird vom Benutzer durch die Hilfsfunktion "inputint" angegeben. Es ist eine Unterscheidung zwischen $n = 1$, $n = 2$ und $n \geq 3$ notwendig, da erst ab $n \geq 3$ alle Vorgänger für die Folge feststehen, zuvor muss man die Vorgänger aus der Definition entnehmen. Für den Fall $n \geq 3$ kommt eine Schleife zum Einsatz, die die Folge entsprechend der eingegebenen Zahl n berechnet. Am Ende jedes Durchlaufs des Schleifenrumpfs wird die berechnete Fibonacci Zahl ausgegeben.

5.3.6 zahl_raten.uasm

Hierbei handelt es sich um ein Spiel, bei dem es geht eine "zufällige" Zahl zu erraten. Die Vorgehensweise des Programms ist wie folgt:

1. Größtmögliche Zahl abfragen
   Der Benutzer wird zunächst gebeten die größtmögliche Zahl m einzugeben.

2. Seed erzeugen
   Der Benutzer wird gebeten einen Seed einzugeben. Der Seed wird in Form eines Strings angegeben. Der String wird mit der "inputint"-Funktion gelesen und in eine Zahl X_0 umgewandelt.

3. Pseudozufallszahl durch die Formel "$X_{n+1} = (a + b \cdot X_n) \mod m$" generieren
   Diese Formel wird n mal aufgerufen. Hierbei sind n, a und b im Code festgelegt. Um alle Zahlen erreichen zu können, sind a und b als Primzahlen gewählt.

4. Spieler rät die Zahl
   Der Spieler rät durch eingeben die Zahl. Dabei bekommt er jedes mal die Rückmeldung, ob die geratene Zahl größer, kleiner oder gleich der gesuchten ist. Die Anzahl der Versuche wird mitgezählt und bei jeder Rückmeldung mit ausgegeben. Hat der Spieler die gesuchte Zahl erraten, so ist das Spiel vorbei.

5.3.7 tictactoe.uasm

Tic Tac Toe ist ein Spiel für zwei Personen. Hierbei geht es darum, abwechselnd eins der neun im Quadrat angelegten Felder mit seiner Markierung zu versehen. Es gewinnt der Spieler, der drei Markierungen in einer Reihe, senkrecht, waagrecht, oder diagonal, angeordnet hat.

Für die Realisierung dieses Spiels werden die Spielinformationen in den Registern gespeichert:

- R1-R9: Spielfelder
  Für den Wertebereich eins Spielfeldregisters gilt: 0(von niemand belegt), 1(von Spieler 1 belegt) und 2(von Spieler 2 belegt)

- R10: Aktueller Spieler Für den Wertebereich eins Spielfeldregisters gilt: 1(Spieler 1 ist an der Reihe), 2(Spieler 2 ist an der Reihe)

- R20: Anzahl der Spielzüge
  Die Anzahl der Spielzüge wird festgehalten, um während der Auswertung ein Unentschieden festzustellen zu können.

Der Spielzyklus besteht aus drei wesentlichen Abschnitten:

1. Eingabe
   Der Spieler, der an der Reihe ist wird gebeten ein Feld anzugeben, in das er seine Markierung setzen will. Dafür gibt er eine Zahl zwischen 1 und 9 an. Nach der

Eingabe wird der Inhalt des Registers R10 in ein Register der Spielfelder(R1-R9) kopiert. Nach der Eingabe endet der Zug des Spielers.

2. Ausgabe

```
1|2|3
-+-+-
4|5|6
-+-+-
7|8|9
```

So sieht die Ausgabe der Spielfelder zu Beginn des Spiels aus. Hierbei wird für jedes Spielfeldregister abgefragt ob es den Wert 0 hat, oder nicht. Ist in einem Feld die 0, so wird eine Zahl herausgeschrieben um zu symbolisieren, dass dieses Feld frei ist und während der Eingabe mit der entsprechenden Zahl ausgewählt werden kann, anderenfalls wird der Inhalt auf den Stack gepusht und es wird eine Funktion aufgerufen. Diese Funktion schreibt auf die Konsole ein 'X' heraus falls das gepushte eine '1' ist. Für eine '2' schreibt es ein 'O'.

3. Auswertung
   Es gibt 8 Reihen die komplett mit der Markierung eines Spielers belegt werden können, damit ein Spieler gewinnt. Dem entsprechend gibt es 8 Abfragen die erfolgen müssen um zu prüfen ob jemand gewonnen hat. Für eine Abfrage werden die drei Register aus einer Reihe mit einander multipliziert. Es gibt nur drei relevante Äquivalenzklassen der Ergebnisse, die dabei berücksichtigt werden müssen. '0','2','4': Entweder eines der Felder ist nicht belegt, oder in einer Reihe treten sowohl '1'-en als auch '2'-en auf. '1': alle Register der Reihe enthalten eine '1', somit gewinnt Spieler 1. '8': alle Register der Reihe enthalten eine '2', somit gewinnt der Spieler 2. Steht ein Gewinner fest, so tritt sofort das Ende des Spiels ein. Falls kein Gewinner feststeht wird der Rundenzähler, das Register R20, um eins erhöht. Beinhaltet er die Zahl '9', so sind '9' Spielzüge vergangen, ohne dass ein Spieler gewonnen hat. Folglich sind alle Felder belegt und es tritt ein Unentschieden ein und somit das Ende des Spiels. Hat niemand gewonnen und es steht auch kein Unentschieden fest, so wird das Spielerregister entweder von '1' auf '2' oder von '2' auf '1' gesetzt und der Spielzyklus beginnt von vorn.

Am Ende des Spiels:

1. Ausgabe des Ergebnisses
   Es wird eine Nachricht ausgegeben, die entweder besagt welcher Spieler gewonnen hat, oder ob ein Unentschieden feststeht. Es folgt die Frage, ob eine neue Runde gespielt werden Soll. Ist die Eingabe ein 'q' so beendet sich das Programm, anderenfalls startet das Spiel neu.

2. bei neuer Runde aufräumen
   Vor dem Neustarten müssen alle Spielfeldregister geleert werden und das Spielerregister wird wieder auf '1' gesetzt.