Georg-Simon-Ohm-Hochschule, Fakultät Informatik

Berichterstattung zum IT-Projekt

UMach, eine einfache virtuelle Maschine

Simon Beer
077566

Liviu Beraru
077566

Willi Fink
077566

Werner Linne
077566

11. Januar 2013

Inhaltsverzeichnis

1 Projektumfang

Das IT-Projekt wurde während des SS 2012 und WS 2012/13 durchgeführt. Im Rahmen dieses Projektes wurden die folgenden Teilaufgaben übernommen und gelöst.

1. Konzeptioneller Entwurf der virtueller 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

1.1 Organisation

Die Sache war auf github. Die Verteilung der Aufgaben erfolgte als TODO Datei. Keiner hat die Datei gelesen, aber ok, scheiß egal, ich will raus aus diesen Loch.

Die Übergabe besteht aus:

1. ZIP mit Pass. AES256. Muhaha.

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 6 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 8 stellt die C-Dateien vor, die diese Komponenten ausmachen.

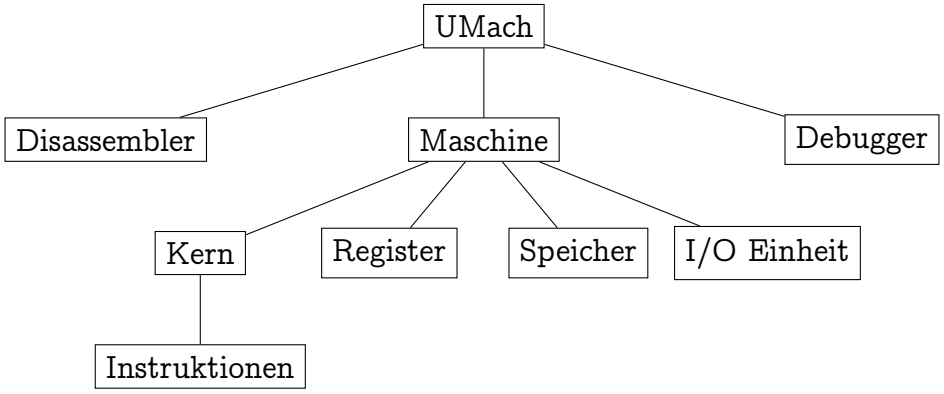

Abbildung 1: Komponenten-Struktur des `umach` Programms

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

4 Qt Debugger

Simon Beer

5 Demo-Programme

Willi Fink

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

5.1 Meist verwendete Befehle

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

- inputint
- printint
- putchar
- newline

5.3 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

5.4 Zahl raten

- Seed erzeugen
- Pseudozufallszahl generieren
- Formel: $X_{n+1} = (a + b \cdot X_n) \mod m$

- Spieler rät die Zahl
- Rückmeldung ob die geratene Zahl größer oder kleiner der gesuchten ist
- Anzahl der Versuche

5.5 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