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

10. 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.

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 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 zusammengefasst.

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 festgelegt. 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 Abgrenzung der Segmente 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, insebesondere 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.

Interruptmodell Für die UMach Maschine wurde ein Interrupt-Mechanismus konzipiert und entwickelt. Die Spezifikation beschreibt diesen Mechanismus in Detail. Interrupts sind Signale, die entweder von der Maschine selber oder vom Programmierer generiert werden können. Jeder Interrupt wird eine Interruptnummer vergeben. Von der Maschine werden sie in Fehlerfälle generiert, vom Programmierer werden die mit der Instruktion `INT`. 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 geschrieben. Beim Start dieses Programms können verschiedene Argumente übergeben werden. Siehe dazu das Dokument UMachVM-Verwendung.pdf.

Die Implementierung erstreckt sich über 38 C-Dateien, die unten kurz vorgestellt werden.

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 nach dem 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 Instruktion implementiert. Schlüssel ist die Befehlsnummer.

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 Registernummer 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 setzte das entsprechende Feld.

3 Assembler

Werner Linne

4 Qt Debugger

Simon Beer

5 Demo-Programme

Willi Fink