Handbuch zum 2a-emulator

Malte Tammena

2. Dezember 2019

Inhaltsverzeichnis

1 Einleitung

Der `2a-emulator` ist ein für den Minirechner 2a umgesetzter Emulator. Der Minirechner 2a wird im Hardwarepraktikum der Universität Leipzig verwendet. Informationen zu ihm finden sich zum Beispiel in „Der Minirechner 2a. Handbuch und Bedienungsanleitung“ von Max Braungardt und Dr. Thomas Schmid. Der `2a-emulator` soll der Fehlerfindung während der Umsetzung von Programmieraufgaben für den Minirechner 2a dienen und wurde im Zuge der Bachelorarbeit „Emulatoren als Testumgebung am Beispiel des Minirechner 2a“ von Malte Tammena umgesetzt.

2 Installation

Vorkompiliertes Programm Die einfachste Variante um den Emulator zu installieren, ist der Download eines der bereits kompilierten Programme[1]. Nach dem Herunterladen kann das Programm in der Konsole ausgeführt werden.

Manuelle Installation Die manuelle Installation erfordert zwei zusätzliche Programme und eine funktionierende Internetverbindung.

`git` Git[2] kann unter Linux mit Hilfe der Paketverwaltung installiert werden. Für Ubuntu-Nutzer findet sich eine Anleitung in der Ubuntu-Hilfe[3]. Unter Windows kann Git zum Beispiel in Form von „Git for Windows“[4] installiert werden.

`rustup` Rustup[5] dient der Verwaltung von Rust-Compiler-Versionen. Er ist der einfachste Weg den Rust-Compiler zu installieren. Anleitungen für alle Plattformen können auf der offizielen Seite[6] gefunden werden.

Für die manuelle Installation muss zunächst der Quellcode[7] mittels git geklont werden. Hierfür werden folgende Befehle (ohne `>`) in der Konsole ausgeführt.

```
> git clone https://v4.git.tammena.rocks/2a-emulator/2
    a-emulator.git
> cd 2a-emulator
```

Um den Emulator zu kompilieren und auszuführen, nutzen wir das Programm `cargo`[8]. Dieses wurde automatisch mit dem Compiler installiert. Folgender Aufruf in der Konsole kompiliert den Emualtor und führt ihn aus:

```
> cargo run --locked --release
```

[1] https://v4.git.tammena.rocks/2a-emulator/2a-emulator/releases
[2] https://git-scm.com/
[3] https://help.ubuntu.com/lts/serverguide/git.html
[4] https://gitforwindows.org/
[5] https://rustup.rs/
[6] https://rustup.rs/
[7] https://v4.git.tammena.rocks/2a-emulator/2a-emulator
[8] https://github.com/rust-lang/cargo/

Das Programm kann lokal installiert werden, so dass ein Aufruf von `2a-emulator` ausreicht um den Emulator zu starten. Folgender Aufruf installiert den Emulator.

```
> cargo install --locked --path . -f
```

Eventuell muss die `PATH`-Variable angepasst werden. Unter Linux wird der Emulator in den Ordner `/.cargo/bin` installiert. Unter Windows befindet sich das Programm nach Installation in `C:\Users\BENUTZERNAME\.cargo\bin`. Diese Pfade können abweichen. Der Aufruf von `cargo install` sollte jedoch nach erfolgreicher Ausführung Informationen über den Installationspfad ausgeben.

3 Benutzung

Die Ausführung von `2a-emulator -help` zeigt Hinweise zur Verwendung des Emulators. Informationen zu möglichen Tastenkombinationen und Eingaben für die interaktive Bedienung des Emulators finden sich im unteren rechten Bereich der interaktiven Darstellung.

Interaktive Oberfläche Um eine interaktive Session zu starten, kann der Emulator mit der Kommandozeilenoption `-interactive` oder ohne Optionen gestartet werden. Zum Beispiel mittels `2a-emulator`. Um optional direkt ein Assembler-Programm zu laden, kann der Aufruf `2a-emulator PFAD/ZUM/PROGRAM.asm` verwendet werden. Sollte sich das Programm im derzeitigen Pfad befinden, reicht somit folgender Aufruf:

```
> 2a-emulator PROGRAM.asm
```

Testausführung Um einen einfachen Text gegen ein Assembler-Programm auszuführen wird der Emulator wie folgt aufgerufen:

```
> 2a-emulator -t PFAD/ZUM/TEST PFAD/ZUM/PROGRAM.asm
```

4 Tests

Um wiederholt bestimmte Eigenschaften von Assembler-Programmen zu überprüfen, können Tests geschrieben werden, welche vom Emulator gegen ein Programm ausgeführt werden. Eine Testdatei besteht aus einem oder mehreren Tests. Ein Test hat immer die gleiche Struktur, die im folgenden Beispiel zu erkennen ist:

```
TEST '1␣+␣1'
WITH
    FC = 0x01,
    FD = 0x01
FOR 1000
```

```
EXPECT
    NO STOP,
    FF = 0x02
END
```

Der obige Test fordert von einem Programm, dass es bei Eingabe von 1 in die Eingaberegister `FC` und `FD` nach 1000 Takten die Ausgabe 2 in das Register `FF` schreibt. Das Programm darf währenddessen nicht anhalten, insbesondere darf kein Fehlerstopp aufgetreten sein. Das Programm erwartet weder, dass allgemeine Additionen korrekt ausgeführt werden noch, dass das Programm niemals anhält. Dem Element `TEST` folgt immer ein Testname. Dieser sollte aussagekräftig und einzigartig gewählt werden, damit er unter einer Menge von fehlgeschlagenen Tests leicht wiedererkannt werden kann. Der `WITH`-Block enthält Konfigurationen für den Testablauf. Folgende Einstellungen können durch Kommata getrennt verwendet werden:

`Fx = 0xNN`	Setze das Eingaberegister Fx auf den hexadezimalen Wert NN. Die Eingaberegister heißen `FC`, `FD`, `FE` und `FF`. Zum Beispiel `FC = 0x09`.
`RANDOM INPUT`	Setze die Eingaberegister auf zufällige Werte.
`RANDOM RESET`	Setze die Maschine an zufälligen Zeitpunkten zurück.
`RANDOM INTERRUPT`	Löse an zufällig gewählten Zeitpunkten Interrupts aus.
`INTERRUPT`	Löst nach Hälfte der Takte einmalig einen Interrupt aus.

Dem `FOR`-Element folgt die Zahl der emulierten Takte. Zuletzt kann nach `EXPECT` eine kommagetrennte Liste von Erwartungen angegeben werden. Möglich sind:

`STOP`	Die Maschine hat nach der vorgegebenen Anzahl an Takten angehalten. Dies kann sowohl ein Fehlerstopp sein, welcher, zum Beispiel, durch einen Stack Overflow ausgelöst wird, oder ein gewöhnlicher Stopp. Letzterer kann im Programm mittels `STOP` ausgelöst werden.
`NO STOP`	Weder Fehlerstopp noch Stopp sind aufgetreten.
`ERROR STOP`	Ein Fehlerstopp ist aufgetreten.
`NO ERROR STOP`	Kein Fehlerstopp ist aufgetreten. Soll auch ein gewöhnlicher Stopp verboten sein, kann `NO STOP` verwendet werden.
`Fx = 0xNN`	Das Ausgaberegister `Fx` enthält nach Ausführung die Hexadezimalzahl NN. Es kann aus den Ausgaberegistern `FE` und `FF` gewählt werden. Zum Beispiel: `FF = 0xF4`.

Da die Blöcke `WITH` und `EXPECT` optional sind, kann ein minimaler Test verfasst werden. Dieser akzeptiert jedes Programm nach 42 Takten:

```
TEST 'pretty useless test'
FOR 42
END
```

5 Beispiel

Das folgende Programm berechnet die Zahl 42 durch Addition von Einsen und schreibt diese auf das Ausgaberegister `FF`. Wird ein Interrupt ausgelöst, schreibt das Programm 41 in das Register `FF` und hält an.

```
#! mrasm

JR MAIN             ; Start at MAIN
JR INTERRUPT        ; Interrupt routine

MAIN:
    EI              ; Enable interrupts
    LDSP 0xEF       ; Load stack pointer, to enable calls
    BITS (0xF9), 1  ; Enable key edge interrupts
    CLR R0          ; Clear register 0
LOOP:               ; The 'Adding'-loop
    INC R0          ; Increase R0
    CMP R0, 42      ; Compare with the target value
    JZS OUTPUT      ; Jump to OUTPUT on R0 == 42
    JR LOOP         ; If R0 != 42, keep adding

OUTPUT:
    ST (0xFF), R0   ; Move R0 to output register
    CLR R0          ; Clear R0
    JR LOOP         ; Return to loop

INTERRUPT:          ; Interrupt handling
    MOV (0xFF), 41  ; Store 41 in output register
    STOP            ; Halt the machine
    RETI            ; If continued, return from interrupt
```

Um zu überprüfen, ob das Programm tatsächlich hält was es verspricht, kann es mit den folgenden Tests überprüft werden. `calc 42` überprüft, ob nach 1000 Takten die Ausgabe 42 vorhanden ist ohne, dass die Maschine angehalten hat. Tatsächlich braucht das obige Programm 962 Takte, bis die Antwort erstmalig auf das Ausgaberegister `FF` geschrieben wird. Der Test `on interrupt` überprüft ob, sofern ein Interrupt ausgelöst wird, das Programm ohne Fehler anhält, nachdem es 41 in die Ausgabe geschrieben hat. Das Programm besteht die Tests problemlos.

```
TEST 'calc 42'
FOR 1000
EXPECT FF = 0x2A, NO STOP
END

TEST 'on interrupt'
WITH INTERRUPT
FOR 1000
EXPECT FF = 0x29, STOP, NO ERROR STOP
END
```

6 Hinweise

UTF8 Der Emulator kann mit dem Feature `utf8` kompiliert werden, um weitere Unicodezeichen im interaktiven Emulator zu erlauben. Dafür können `cargo` die Parameter `--feature utf8` bei den obigen Aufrufen mitgegeben werden.