TECHNISCHE UNIVERSITÄT
CHEMNITZ

Professur Betriebssysteme

Algorithmen und Programmierung, WS 2023/2024

Belegaufgabe #5: **Rechner-Simulator**

Deadline: **22. Dezember 2023, 23.00 Uhr**
Erreichbare Punkte: **10**

1 Motivation

Ziel dieser Aufgabe ist…

- …das Verstehen und Umsetzen einer nichttrivialen textuellen Spezifikation;
- …der Umgang mit Bitoperatoren und `structs`;
- …der Umgang mit Konzepten wie Komplement oder Endianess;
- …der korrekte Umgang mit Speicher und Arrays.

2 Prozessoren

Die Harvard-Architektur[1] zeichnet sich dadurch aus, dass Befehls- und Datenspeicher weitgehend getrennt sind. In dieser Aufgabe geht es um ein Teil (eines Debugging-Tools) für einen fiktiven (stark vereinfachten) 4-Bit-Rechner mit einem Adressraum (sowohl der des Programm- als auch der des Datenspeichers) von jeweils 8 Bit. Sie sollen dafür einen Simulator schreiben, der Änderungen im Speicher zurückgibt.

Architektur

Der Rechner hat drei Register, siehe Abbildung 1:

- **Befehlszeiger PC**, 8 Bit, soll stets die Adresse des nächsten zu interpretierenden Befehls beinhalten (wenn keine Sprünge vorkommen also des Befehls, der dem aktuellen folgt, aufsteigend)

[1] `https://de.wikipedia.org/wiki/Harvard-Architektur`

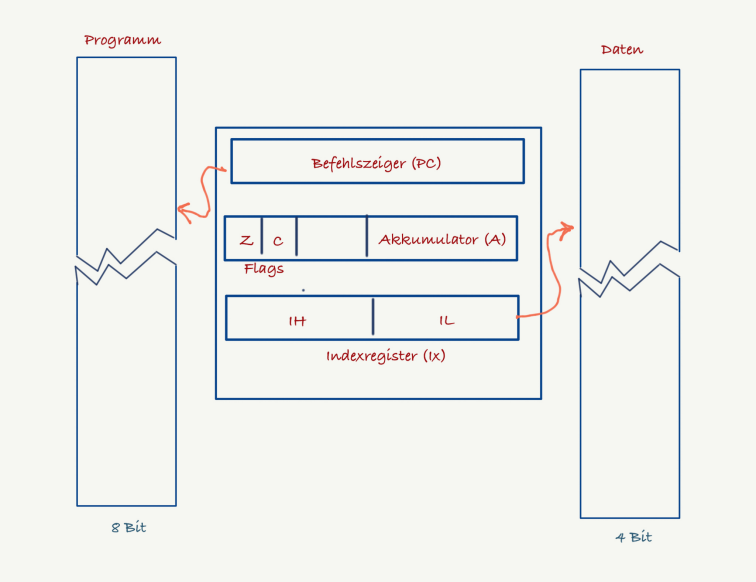


Abbildung 1: 4-Bit „Rechner“

- **Kombiregister**, 8 Bit, besteht aus den Flags **Z** (Zero, für Null) und **C** (Carry, für Übertrag) in den Bits 7 und 6, sowie aus dem 4-Bit-Akkumulator (**A**) in den Bits 0 bis 3. Der Akkumulator sammelt die Ergebnisse einer Befehlsabarbeitung, kann aber auch selbst Operand sein.
  Die Bits 4 und 5 werden nicht genutzt und dürfen nicht verändert werden.
- **Indexregister IX**, 8 Bit, dient vor allem zur indirekten Adressierung. Die jeweils höhere (**IH**) und niedrigere Hälfte (**IL**) können jedoch als einzelne 4-Bit-Register angesprochen werden.

Befehlsabarbeitung

Der Rechner führt den Standardzyklus der Befehlsabarbeitung aus:

1. Er liest einen Befehl dessen Adresse durch den Befehlszeiger PC gegeben aus dem Programmspeicher.
2. Er erhöht den Befehlszeiger um 1, so dass er auf den nächsten Befehl zeigt.
3. Er führt den gelesenen Befehl durch und ändert dabei ggf. Register, Flags oder Speicher.
4. Gehe zu Schritt 1.

Ein Befehl des Rechners besteht aus jeweils einem 8-Bit-Datenwort. Damit man leichter den Sinn erfassen kann, werden Befehlsmuster auch mit Kürzeln (sogenannten Mnemonics) dargestellt, denen ggf. ein oder zwei Argumente folgen. Bei zwei Argumenten (das kommt nur bei Transfer vor) steht das erste immer für das Ziel der Operation, das zweite für die Quelle.

Ein Teil des Befehls beschreibt die Art der Operation, andere die Argument. In Tabelle 1 sind alle Befehle mit ihren Bitmustern aufgelistet. Die Spalten für Flags geben dabei durch ein „X“ an, ob durch die Befehlsausführung die Flags geändert werden. Ist das der Fall, so wird das Zero-Flag immer gesetzt (erhält den Wert eins), wenn das Ergebnis 0 ist, sonst rückgesetzt. Das Carry-Flag wird bei Additionsoperationen gesetzt, wenn das Ergebnis „überläuft“, also ein füntes Bit zur Ergebnisdarstellung benötig werden würde.

Die Tabellen 2 und 3 geben an, wie die Argumente bzw. die Bedingungen für die bedingten Sprünge kodiert sind.

Tabelle 1: Befehlssatz des Rechners

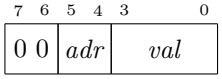
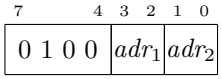
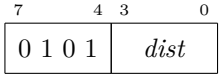
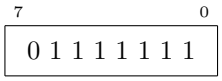
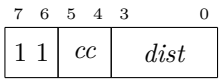
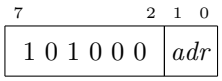
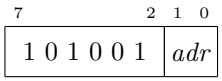
Mnemonic/ Bedeutung	**Bitmuster**	Flags Z	Flags C	**Wirkung**
`ldi` *adr*, *val* *load immediate*				Lädt den im Befehl enthaltenen 4-Bit-Wert *val* an die durch *adr* angegebene Stelle.
`mv` adr_1, adr_2 *move*				Kopiert den Wert in adr_2 nach adr_1.
`jr` *dist* *jump relative*				Ändert den Wert von PC um den Wert *dist*. Dabei ist *dist* im Zweierkomplement; eine Sprungweite von 0 „springt“ zum folgenden Befehl.
`halt` *halt execution*				Hält die Ausführung des Rechners an.
`b`*cc dist* *branch when condition*				Ändert bei erfüllter Bedingung (siehe Tabelle 3) den Wert von PC um den Wert *dist*. Dabei ist *dist* im Zweierkomplement.
`add` *adr* *add*		X	X	Addiert die Werte im Akkumulator und in *adr* und speichert das Ergebnis im Akkumulator.
`adc` *adr* *add with carry*		X	X	Addiert die Werte im Akkumulator, in *adr* sowie das Übertragsflag (C) und speichert das Ergebnis im Akkumulator.

Tabelle 1: Befehlssatz des Rechners (Fortsetzung)

Mnemonic/ Bedeutung	**Bitmuster**	Flags Z C	**Wirkung**
`and` *adr* *and*	7 … 2 1 0 1 0 1 0 1 0 | *adr*	X	Verküpft den Wert im Akkumulator und den Wert aus *adr* bitweise mit UND und speichert das Ergebnis in Akkumulator.
`or` *adr* *or*	7 … 2 1 0 1 0 1 0 1 1 | *adr*	X	Verküpft den Wert im Akkumulator und den Wert aus *adr* bitweise mit ODER und speichert das Ergebnis in Akkumulator.
`xor` *adr* *exclusive or*	7 … 2 1 0 1 0 1 1 0 0 | *adr*	X	Verküpft den Wert im Akkumulator und den Wert aus *adr* bitweise mit Exklusiv-ODER und speichert das Ergebnis in Akkumulator.
`neg` *adr* *negation*	7 … 2 1 0 1 0 1 1 0 1 | *adr*	X	Bildet vom Wert in *adr* das Einerkomplement.
`cpl` *adr* *complement*	7 … 2 1 0 1 0 1 1 1 0 | *adr*	X	Bildet vom Wert in *adr* das Zweierkomplement

Tabelle 2: Kodierung der Adressen der Argumente (*adr*) in den Befehlen

Bits	**Kürzel**	**adressierter Wert**
`0 0`	`a`	Inhalt des Akkumulators (registerdirekte Adressierung)
`1 0`	`ih`	Inhalt des Registers `IH` (registerdirekte Adressierung)
`0 1`	`il`	Inhalt des Registers `IL` (registerdirekte Adressierung)
`1 1`	`(ix)`	Inhalt der Speicherstelle, deren Adresse in `IX` steht (indirekte Adressierung)

Tabelle 3: Kodierung der Bedingungen in bedingten Sprüngen

Bits	**Küzel**	**Bedingung**
`0 0`	`nz`	Das Zero-Flag ist nicht gesetzt ($Z = 0$)
`0 1`	`z`	Das Zero-Flag ist gesetzt ($Z = 1$)
`1 0`	`nc`	Das Übertragsflag ist nicht gesetzt ($C = 0$)
`1 1`	`c`	Das Übertragsflag ist gesetzt ($C = 1$)

3 Aufgabe

Schreiben Sie eine Funktion

```
char* simcpu(struct regs *registers, uint8_t data[128], uint8_t cmd[256]);
```

die die Ausführung von Befehlen der beschriebenen Architektur simuliert. Die Parameter von `simcpu()` haben folgende Bedeutung:

- `registers`: Zeiger auf eine Datenstruktur, die die Register des Rechners beschreibt und folgendermaßen definiert ist:

  ```
  struct regs {
    uint8_t pc; // Befehlszeiger
    uint8_t fa; // Flags + Akkumulator
    uint8_t ix; // Indexregister
  };
  ```

- `data`: Ein `uint8_t`-Array der Größe 128, das den Datenspeicher darstellt. Beachten Sie, dass jedes Element des Arrays für **zwei** Speicherstellen steht, da ein Speicherwort ja nur vier Bits groß ist. Dabei stehen die oberen (höherwertigen) vier Bits für die größere Adresse.[2]

- `cmd`: Ein `uint8_t`-Array der Größe 256, das den Befehlsspeicher darstellt.

Die Funktion `simcpu()` soll solange Befehle ausführen, bis entweder

- der Befehl `halt` ausgeführt wird *oder*
- ein ungültiger Befehlscode gefunden wird *oder*
- einhundert (100) Befehle ausgeführt wurden.

Nach der Ausführung von `simcpu()` sollen `registers` und `data` die der Ausführung entsprechenden Werte enthalten. Als Rückgabe gibt `simcpu()` einen C-String mit einer *aufsteigend nach Adressen geordneten* Liste von *geänderten* – d.h. vom Ausgangszustand abweichenden – Speicherstellen zurück, die für jede geänderte Adresse eine Zeile im folgendem Format enthält:[3]

- Zweistellige Adresse in Hexadezimalschreibweise mit Großbuchstaben ohne Prefix (aber ggf. mit führender Null)
- Leerzeichen
- Einstelliger Wert in Hexadezimalschreibweise mit Großbuchstaben ohne Prefix
- Newline („`\n`“)

Es wird nur der Endzustand dokumentiert, so dass jede Adresse maximal einmal vorkommen kann.

Einschränkungen

Bei der Programmierung von `simcpu()` unterliegen Sie einigen Einschränkungen:

[2] Dies entspricht etwa einem inversen Little-Endian-Format, vgl. `https://de.wikipedia.org/wiki/Byte-Reihenfolge#Little-Endian-Format`

[3] Siehe auch das Beispiel auf Seite 6.

- Sie dürfen das Schlüsselwort `goto` **nicht** nutzen!
- Sie dürfen **keine** globalen Variablen benutzen!
- Sie dürfen die Signatur (d.h. die Deklaration) von `simcpu()` **nicht** ändern!
- Sie dürfen **ausschließlich** Bibliotheksfunktionen nutzen, die im **C99-Standard**[4] enthalten sind, keine zusätzlichen.
- Sie dürfen **keine** Memory-Leaks oder hängenden Zeiger (*dangling pointer*) verursachen! Außer dem von `simcpu()` zurückgegebenen Sting müssen daher alle im Code reservierten Speicherbereiche wieder freigegeben werden.
- Ihr Code darf **nichts** ausgeben.
- Schreiben Sie **keine** `main()`-Funktion![5]

Beispiel

Wenn jedes Element von `*registers` und alle Elemente von `data` mit Null initialisiert sind, dann führt der Aufruf von `simcpu()` bei einer Initialisierung

```
uint8_t cmd[256] = {
  [0] = 0x0f, // ldi a, 15     => a hat den Wert 15 bzw. -1
  [1] = 0x48, // mv ih, a      => ix enthält den Wert 240
  [2] = 0x44, // mv il, a      => ix hat den Wert 255
  [3] = 0x32, // ldi (ix), 2   => Adresse FF hat den Wert 2
  [4] = 0xa2, // add ih        => a hat den Wert 14 bzw. -2
  [5] = 0x44, // mv il, a      => ix hat den Wert 254
  [6] = 0x3a, // ldi (ix), 10  => Adresse FE hat den Wert A
  [7] = 0x7f, // halt          => in data[127] steht 42 (bzw. 0x2a)
};
```

zu folgender Rückgabe: „`FE A\nFF 2\n`“ bzw. bei Ausgabe im Terminal zu:

```
FE A
FF 2
```

4 Hinweise zur Aufgabe

- Ihnen wird unter `https://osg.informatik.tu-chemnitz.de/lehre/aup/support/cpu.h` eine Headerdatei `cpu.h` zur Verfügung gestellt, die die benötigten Typdeklarationen und die Funktionssignatur enthält.
- Lesen Sie die Architekturbeschreibung *gründlich* durch! Erstellen Sie eine Checkliste von zu beachtenden Anforderungen. Haken Sie bei der Implementation realisierte Anforderungen ab.
- Achten Sie bei den Befehlsbitmustern darauf, ob Sie Systematiken erkennen und nutzen Sie diese gegebenfalls – das könnte Ihren Code kürzer und übersichtlicher machen.
- Beachten Sie, dass ein Schreiben eines bereits im Speicher vorhandenen Wertes diesen *nicht* ändert.

[4] Siehe z.B. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf`
[5] Sie dürfen diese natürlich schreiben, aber sie soll nicht in der Abgabe enthalten sein.

5 Einreichung

- Erstellen Sie ein Archiv in Form einer .zip- oder einer .tar-Archivdatei, welche Ihre Lösungsdatei `cpu.c` enthält.
- Fügen Sie dem Archiv keine weiteren Dateien oder Ordner hinzu.
- Reichen Sie Ihre Lösung unter `https://osg.informatik.tu-chemnitz.de/submit` ein.
- Bis zum Abgabeende (22. Dezember 2023, 23.00 Uhr) können beliebig neue Lösungen eingereicht werden, die die jeweils älteren Versionen ersetzen.
- Ihr Code muss auf der Testmaschine mit den Optionen "`-std=c99 -Wall -Werror`" übersetzbar sein. Deren Details sind auf dem OpenSubmit-Dashboard verfügbar.
- Ihre Lösung wird automatisch auf die Einhaltung formaler Kriterien validiert. Sie werden über den Abschluss der Validierung per eMail informiert. Ihre Lösung wird nur bewertet, wenn die Validierung erfolgreich war.

8. Dezember 2023