Projekt B: Emulator

6. Januar 2023

Ziel dieser Projektaufgabe ist es, ein Programm zu entwickeln, dass die Ausführung von Maschinenbefehlen eines einfachen Prozessors nachbildet (emuliert). Die Spezifikation der Prozessorarchitektur und des Befehlssatzes wird hierzu im Folgenden dargestellt. Wichtige Hinweise finden Sie ebenfalls auf dem Aufgabenblatt – selbstverständlich können Sie sich auch darüber hinaus zu dem Thema informieren. Beachten Sie außerdem die Anweisungen auf dem allgemeinen Informationsblatt zu den Projektaufgaben. Bitte reichen Sie die Ergebnisse Ihrer Meilensteine 2 Tage vor dem jeweiligen Termin mit Ihrem Tutor über Git ein.

Spezifikation

Das Start-Up-Unternehmen TechMic hat sich auf die Programmierung von Mikrocontrollern für eingebettete Systeme spezialisiert. Das noch recht junge Team an Informatiker:innen hat jetzt schon die Auftragsbücher gefüllt und ist daher auf schlanke und effiziente Abläufe angewiesen. Da insbesondere die Fehlersuche in Programmen für eingebettete Systeme aufwendig ist, soll hierfür eine elegante Methode entwickelt werden. Nach einem Team-Meeting, in dem diskutiert wurde, welche Tools zukünftig zur Verbesserung der Arbeit beitragen sollen, wurde ein Prozessor-Emulator als beste Lösung vorgeschlagen. Der Emulator soll die Funktionsweise des eingesetzten Prozessors nachbilden. Hierdurch können die entwickelten Programme effizienter erprobt, getestet und Fehler schneller gefunden werden, da das Übertragen der kompilierten Software auf das eingebettete System entfällt.

In dem wöchentlichen Team-Meeting wurde eine Anforderungsspezifikation ausgearbeitet, die folgende Punkte enthält:

1. (Vereinfachte) Nachbildung der Prozessorarchitektur bestehend aus
   - Programmzähler: 8 Bit / 1 Byte
   - 16 Register: je 8 Bit / 1 Byte
   - Programmspeicher: 512 Byte, Adressierung von je 2 Bytes
   - Datenspeicher: 256 Byte, byte-weise Adressierung
2. Abbildung des gesamten Instruktionssatzes
   - 16 Instruktionen
   - 2 Byte je Instruktion (bestehend aus Opcode und Operanden)
3. Darstellungsformen (Speicherformat) des Mikrocontroller-Programms
   - Textdateien mit Hexadezimaldarstellung des Programmes (Dateiendung *.hex*)
   - optional: Binärdateien als direktes Abbild des Programmspeichers (Dateiendung *.bin*)
4. Eingabe und Anfangszustand:
   - Der Dateiname des Mikrocontroller-Programms soll Ihrem Emulator als Kommandozeilen-Argument übergeben werden.
     Beispiel: `./emu add.hex`
   - Sofern Sie beide Eingabeformate (s.o.) ermöglichen, soll Ihr Programm automatisch

anhand der Dateiendung erkennen, welches Eingabeformat vorliegt.

- Alle Speicher (Daten, Programm), Register und der PC sind initial mit dem Wert 0 belegt. Das Mikrocontroller-Programm wird vollständig in den Programmspeicher beginnend bei Adresse 0 geschrieben. Dann wird dieses mit der ersten Instruktion (PC = 0) sofort ausgeführt.

5. Ausgabe
   - Sehen Sie eine Möglichkeit vor, die Belegung der Register und des Speichers nach jeder ausgeführten Instruktion am Bildschirm auszugeben
   - Schreiben Sie den Inhalt der Register und des Speichers am Programmende in je eine Datei.
     - Nutzen Sie Textdateien (bzw. den Textmodus)
     - Schreiben Sie alle Werte im Hex-Format
     - Schreiben Sie die Registerinhalte in aufsteigender Registerreihenfolge in eine einzige Zeile. Trennen Sie die Werte mit einem einfachen Leerzeichen.
     - Schreiben Sie den Speicherinhalt in aufsteigender Adressreihenfolge mit 16 Werten (16 Bytes, d.h. 32 Hex-Ziffern) je Zeile. Trennen Sie die Werte mit einem einfachen Leerzeichen.
6. Optional: Assembler zur Konvertierung von Assembler-Code in Maschinenbefehle
   - Einlesen von Textdateien mit Assembler-Befehlen (zeilenweise)
   - Ausgabe im Textformat (hexadezimale Repräsentation der Maschinenbefehle) oder Binärformat (Abbild des Programms im Speicher der Maschine)

Der Instruktionssatz der Maschine ist in Table 1 dargestellt.

Hinweise

- Die Registerinhalte und alle mathematischen Operationen können Sie als ganzzahlig und vorzeichenlos auffassen.
- Alle Instruktionen haben eine Breite von 16 Bit Die höherwertigen 4 Bits stellen die Instruktion über den Opcode dar, danach folgen immer 12 Bits zur Darstellung der Operanden. Ungenutzte Bits sind typischerweise Nullen.
- Nach jeder Instruktion wird der Programmzähler (PC) um eins erhöht (insb. auch nach einem Sprung und einer `NOP` Instruktion).
- Die Funktion der Maschine ist nachfolgend näher erläutert.
- Das Ergebnis von logischen Operationen (Vergleiche, UND, ODER) ist immer 1 (Bedingung erfüllt) oder 0 (Bedingung nicht erfüllt).
- Sprünge zurück lassen sich unter Ausnutzung der Modulo-Eigenschaft des Programmspeichers bzw. des Programmzählers erzielen.

Assembler	Opcode	Operands		Operation	Beschreibung
	1. Byte	2. Byte			
	4 bit	4 bit	4 bit		
MOV Rn, addr	0000	Rn	addr	Rn ← M(addr)	Liest den Datenspeicher an Adresse `addr` und kopiert den Inhalt in Register `Rn`.
MOV addr, Rn	0001	Rn	addr	M(addr) ← Rn	Schreibt den Inhalt von Register `Rn` an die Adresse `addr` im Datenspeicher.
MOV Rn, @Rm	0010	Rn	Rm —	Rn ← M(Rm)	Liest den Datenspeicher an der Adresse, die in Register `Rm` abgelegt ist, und kopiert den Inhalt in Register `Rn`.
MOV @Rn, Rm	0011	Rn	Rm —	M(Rn) ← Rm	Schreibt den Inhalt von Register `Rm` im Datenspeicher an die Adresse, die in `Rn` abgelegt ist.
MOV Rn, #literal	0100	Rn	literal	Rn ← literal	Schreibt den Wert `literal` in das Register `Rn`.
MOV Rn, Rm	0101	Rn	Rm —	Rn ← Rm	Schreibt (kopiert) den Inhalt von Register `Rm` in das Register `Rn`.
ADD Rn, Rm	0110	Rn	Rm —	Rn ← Rn + Rm	Addiert die Inhalte der Register `Rn` und `Rm` und schreibt das Ergebnis nach `Rn`.
SUB Rn, Rm	0111	Rn	Rm —	Rn ← Rn - Rm	Subtrahiert den Inhalt des Registers `Rm` vom Inhalt des Registers `Rn` und schreibt das Ergebnis nach `Rn`.
MUL Rn, Rm	1000	Rn	Rm —	Rn ← Rn * Rm	Multipliziert die Inhalte der Register `Rn` und `Rm` und schreibt das Ergebnis nach `Rn`.
DIV Rn, Rm	1001	Rn	Rm —	Rn ← Rn / Rm	Dividiert den Inhalt des Registers `Rn` durch den Inhalt des Registers `Rm` und schreibt den ganzzahligen Anteil des Ergebnisses nach `Rn`.
AND Rn, Rm	1010	Rn	Rm —	Rn ← Rn AND Rm	Führt die logische Operation UND aus und schreibt das Ergebnis nach `Rn`.
OR Rn, Rm	1011	Rn	Rm —	Rn ← Rn OR Rm	Führt die logische Operation ODER aus und schreibt das Ergebnis nach `Rn`.
JZ Rn, rel	1100	Rn	rel	PC ← PC + rel (wenn Rn=0)	Sprung im Program um `rel`, wenn das Register `Rn` den Wert 0 hat.
CMP Rn, Rm	1101	Rn	Rm —	Rn ← Rn = Rm	Prüft die Inhalte der Register `Rn` und `Rm` auf Gleichheit und schreibt das Ergebnis nach `Rn`.
LESS Rn, Rm	1110	Rn	Rm —	Rn ← Rn < Rm	Prüft, ob der Inhalt des Registers `Rn` kleiner ist als der Inhalt von `Rm` und schreibt das Ergebnis nach `Rn`.
NOP	1111	—	— —	—	Es wird keine Operation ausgeführt.

Tabelle 1: Instruktionssatz des Prozessors

Funktionsprinzip

Im Folgenden wird das Funktionsprinzip des Prozessors dargestellt und die für die Implementierung relevanten Aspekte beleuchtet.

Ein Prozessor besteht aus mehreren Einheiten, wie in Abb. 1 vereinfacht dargestellt. Der von der Firma eingesetzte Prozessor basiert auf der sog. Harvard-Architektur, d.h. das zwei separate Speicher für das Programm und die Daten existiert. Die CPU besteht aus zwei miteinander verbundenen Funktionseinheiten, dem Steuerwerk und dem Rechenwerk. Das Steuerwerk wiederum ist mit Programmspeicher verbunden, in dem die Instruktionen des Programms abgelegt sind. Der Datenspeicher ist über das Bussystem nur vom Rechenwerk aus erreichbar.

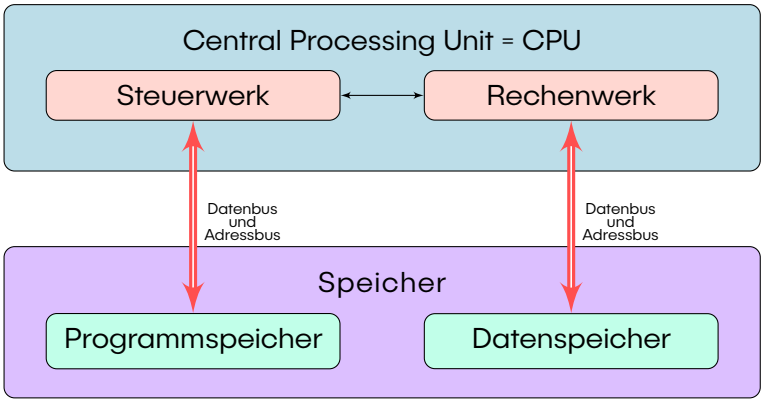


Abbildung 1: Vereinfachte Darstellung eines Prozessors nach dem Harvard-Prinzip

Die Hauptaufgabe des Steuerwerkes ist es, den Ablauf der Befehlsverarbeitung zu steuern. Das Steuerwerk verfügt über einen sog. Programmzähler (*Program Counter*, PC), der den Index (also die Position) der nächsten Instruktion im Programmspeicher speichert. Nach Ausführung *jeder* Instruktion wird der Programmzähler um eins erhöht. Sprungbefehle dienen dazu, die Bearbeitungsreihenfolge der Instruktionen zu verändern; dies ist zum Beispiel zur Implementierung von Schleifen notwendig. Bei Programmstart hat der PC den Wert 0, es wird also die erste Instruktion im Programmspeicher gelesen.

Das Rechenwerk führt u.a. die arithmetisch logischen Operationen aus, die als Operanden jedoch nur Register(inhalte) verwenden können. Sind die Daten im Datenspeicher abgelegt, müssen diese also vorher in Register geladen werden. Werden mehr als die verfügbaren Register benötigt, müssen Daten in den Datenspeicher ausgelagert werden. Aus diesem Grund verfügt das Rechenwerk über mehrere Instruktionen, um Daten zwischen den Registern und dem Datenspeicher transferieren zu können.

Das Steuerwerk instruiert das Rechenwerk auf Basis des aktuellen Befehls. Handelt es sich bei der aktuellen Instruktion z.B. um eine Addition, liest das Rechenwerk die Werte der beiden Operanden (Register), addiert diese, und speichert das Ergebnis im ersten Operanden (Register). Siehe hierzu Table 1 sowie die Beispiele am Ende des Aufgabenblattes.

In der Realität führt das Steuerwerk Instruktionen aus, solange die CPU mit Strom versorgt wird. Die Emulation soll jedoch beendet werden, sobald der PC die Anzahl der Instruktionen erreicht hat (wenn der PC also auf eine Instruktion zeigt, deren Index größer oder gleich der Programmlänge ist). Sie dürfen hierfür annehmen, dass der Programmspeicher nie vollständig ausgenutzt wird.

Beispiel 1: Addition von zwei Zahlen

Aufgabe: Es sollen zwei Zahlen 15 und 1 addiert und das Ergebnis im Speicher an Adresse 0 abgelegt werden.

Mit dem gegebenen Instruktionssatz werden zunächst die beiden Zahlen in (beliebige) Register geschrieben und diese dann addiert. Das Ergebnis wird dann in den Datenspeicher geschrieben.

Wir wählen hierzu die Register 0 und 1. Da beide Werte nach der Addition nicht weiter benötigt werden, können wir beide Register direkt in der Addition verwenden. Eine mögliche Lösung der Aufgabe mit dem Instruktionssatz lautet:

Assembler

```
1 MOV R0, #15 ; R0 <- 15
2 MOV R1, #1  ; R1 <- 1
3 ADD R0, R1  ; R0 <- R0 + R1
4 MOV 0, R0   ; M[0] <- R0
```

Binär-Darstellung

```
1 0100.0000 0000.1111
2 0100.0001 0000.0001
3 0110.0000 0001.0000
4 0001.0000 0000.0000
```

Hex-Darstellung

```
1 400F
2 4101
3 6010
4 1000
```

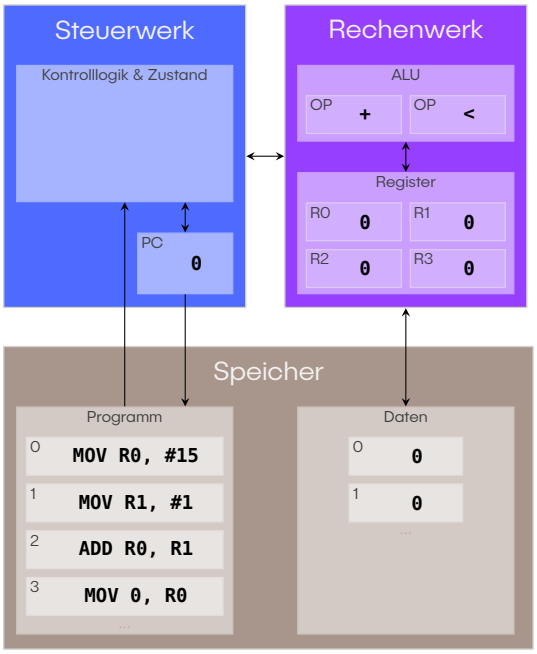
Der Übersichtlichkeit halber werden in obiger Darstellung die beiden Bytes mit einem Leerzeichen getrennt und die Bytes jeweils in Vierergruppen aufgeteilt und mit einem Punkt getrennt. Es wird ferner immer eine Instruktion pro Zeile dargestellt. Dies entspricht nicht dem erwarteten Datenformat Ihres Programms.

Erläuterung

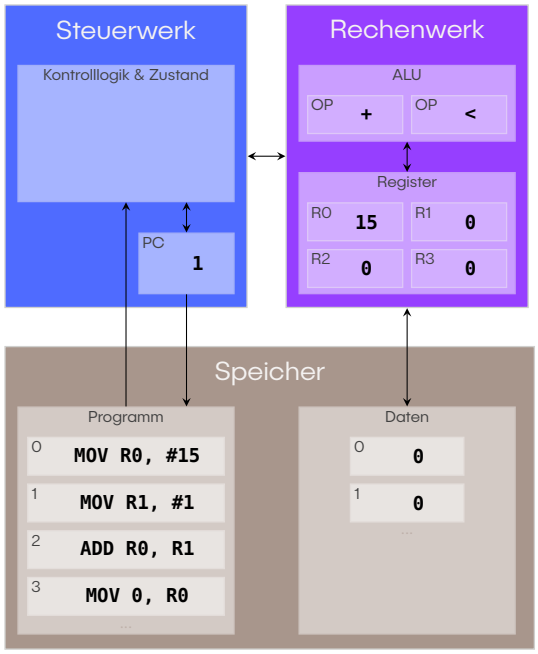
1. Die erste Assembler-Instruktion schreibt den Wert 15 nach Register `R0`. Der Opcode lautet demnach `0100`, die 4-Bit für das Register 0 sind `0000` und der Wert 15 ist binär `0000.1111`. Zusammengesetzt ergibt sich die erste Zeile in der Binärdarstellung, die dann unmittelbar in einen hexadezimalen Ausdruck umgewandelt werden kann.
2. Die zweite Assembler-Instruktion schreibt den Wert 1 nach Register `R1`. Der Opcode lautet demnach wieder `0100`, die 4-Bit für das Register 1 sind `0001` und der Wert 1 ist binär `0000.0001`.
3. Die Addition hat den Opcode `0110`, die 4-Bit für das erste Register 0 sind `0000` und für das zweite Register 1 `0001`. Die verbleibenden vier Bits werden mit Nullen aufgefüllt.
4. Die MOV-Operation zum Schreiben eines Registerinhalts in den Speicher besitzt den Opcode `0001`, die 4-Bit für das Register 0 sind `0000` und die 8-Bit für die Adresse 0 lauten `0000.0000`.

Ausführung

Die Ausführung des Beispiel-Programms durch den Prozessor wird nachfolgend detailliert erläutert. Es wird nur ein Ausschnitt des Mikrocontrollers dargestellt. Die Abbildungen zeigen jeweils den Zustand des Mikrocontrollers *nach* Ausführung der jeweiligen Instruktion.

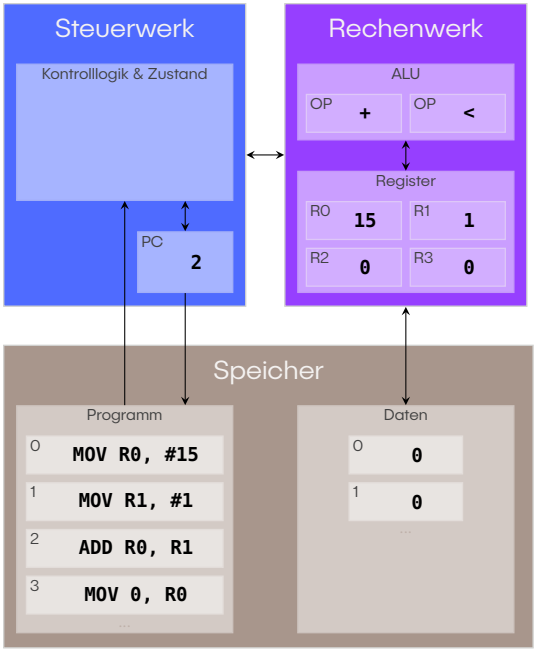


0. Der Programmzähler, Datenspeicher sowie alle Register haben den Anfangswert Null. Das Programm wurde in den Programmspeicher geladen.



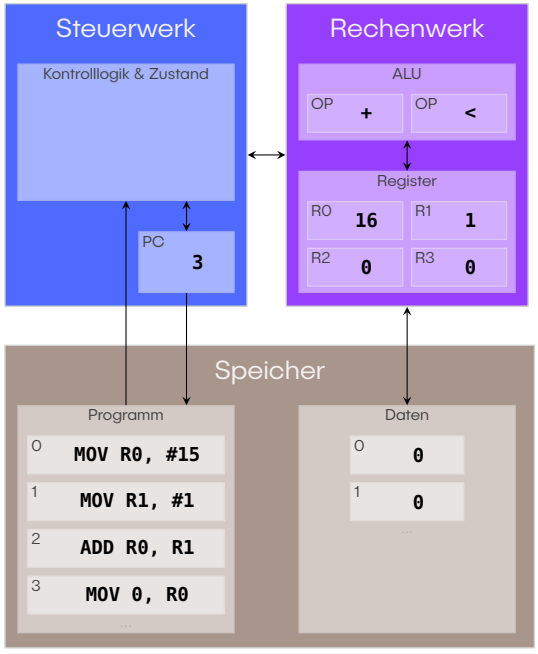
1. Die Instruktion an Adresse PC=0 wird ausgeführt

- Der Wert 15 wird nach R0 geschrieben.
- Der PC wird danach um eins erhöht.



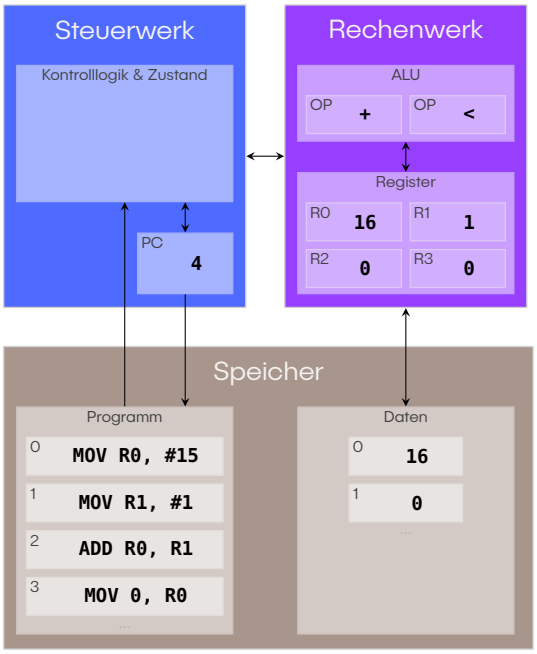
2. Die Instruktion an PC=1 wird ausgeführt

- Der Wert 1 wird nach R1 geschrieben.
- Der PC wird danach um eins erhöht.



3. Die Instruktion an PC=2 wird ausgeführt

- Die Inhalte von R0 und R1 werden addiert und das Ergebnis in R0 abgelegt.
- Der PC wird danach um eins erhöht.



4. Die Instruktion an PC=3 wird ausgeführt
   - Der Inhalt des Registers `R0` wird an die Adresse 0 im Datenspeicher geschrieben.
   - Der PC wird danach um eins erhöht.
   - Da der PC nun hinter die letzte Anweisung des Programms zeigt, wird die Emulation beendet.

Beispiel 2

Betrachten wir abschließend noch ein etwas komplexeres Beispiel, das auch Sprunganweisungen verwendet.

Berechnet werden soll die Summe von aufsteigenden Integern:

$$\text{sum} = \sum_{i=1}^{10} i$$

In der Programmiersprache `C` wäre eine Lösung

```c
uint8_t sum = 0;
for (uint8_t i = 1; i <= 10; i++) {
    sum += i;
}
```

Eine äquivalente Formulierung in Assembler mit dem oben beschrieben Instruktionssatz wäre:

```
1  MOV R0, #1  ; const. 1
2  MOV R1, #10 ; const. 10
3  MOV R3, #0  ; sum
4  MOV R4, #1  ; i
5  ADD R3, R4  ; sum += i
6  ADD R4, R0  ; i++
7  MOV R5, R1
8  LESS R5, R4 ; 10 < i
9  JZ R5, 251  ; -5 + 256
10 MOV 0, R3   ; sum -> mem
```

Der Programmablauf ist analog zum vorherigen Beispiel, beinhaltet aber einen bedingten Sprung. Hat das Register R5 — welches das Ergebnis des Vergleichs `R1 < R4` enthält (eigentlich `R5 < R4`, aber `R1` wird unmittelbar vorher nach R5 kopiert) — den Wert 0, soll die Ausführung mit Zeile 5 fortgesetzt werden. Hierbei sind nun zwei Aspekte zu berücksichtigen:

1. Da nach jeder Instruktion der PC um 1 erhöht wird, muss der Sprung einen zusätzlichen Schritt zurück machen, um dieses Verhalten zu kompensieren. Es muss hier also nach Zeile 4 bzw. um 5 Instruktionen zurückgesprungen werden.
2. Da alle Operanden als vorzeichenlos betrachtet werden, wird der Sprung in Zeile 9 statt 5 zurück, was -5 entspräche, 251 nach vorne gemacht. Hierbei wird ein Überlauf des Programmzählers (8 Bit) ausgenutzt, um die eigentliche Subtraktion in eine Addition umzuwandeln.

Damit ergibt sich die Instruktion in Zeile 9 dann zu `JZ R5, 251`. Wird der Sprung genommen (R5 = 0), ergibt sich

$$\text{PC} \leftarrow (\text{PC} + 251 + 1) \bmod 256 = (9 + 251 + 1) \bmod 256 = 261 \bmod 256 = 5$$

und es wird mit Zeile 5 fortgesetzt. Hat R5 einen von Null verschiedenen Wert, wird in Zeile 10 fortgesetzt.

Hinweise zur Implementierung

- Die Speicher und Register können elegant mit Arrays implementiert werden.
- Verwenden Sie ausschließlich Datentypen mit fester Bitbreite (z. B. `uint8_t`) aus *stdint.h*.
- Das Einlesen im hexadezimalen Format kann elegant mit `fscanf` und einem passenden Formatstring erledigt werden.
- Die Interpretation der Instruktionen (Auslesen von Bitgruppen an bestimmten Stellen) kann mit Division und Modulo-Operation erreicht werden[1]. Um z. B. das fünfte und sechste Bit (von rechts, rot markiert) aus `10101100` zu selektieren, können Sie zunächst durch $16 = 2^4$ teilen und dann den Rest der Division mit $4 = 2^2$ (Modulo-Operation) bestimmen. Das Ergebnis lautet dann `00000010`. Die selektierten Bits sind rot markiert. Sie wurden nach rechts verschoben (Division) und alle Bits zur Linken wurden genullt (Modulo-Operation).
- Die ausgeführte Operation (und Interpretation des/der Operanden) hängt (ausschließlich) vom Opcode der Instruktion ab. Sie können daher die Auswahl der Operation

[1] Noch besser lässt sich das Problem mit Bit-Operationen oder Bitfeldern angehen, aber diese werden in der Vorlesung nicht behandelt

umsetzen, indem Sie den Opcode in einer Sequenz aus Verzweigungen mit **`if-else`** oder einem **`switch`**-Block auswerten. Nutzen Sie hierbei eine Aufzählung (**`enum`**) anstelle von *Magic Numbers*. Alternativ können Sie die Auswahl auch (besonders elegant) mit einem Funktionszeiger-Array lösen.

- Sollten Sie auch das binäre Datenformat zur Eingabe implementieren, müssen Sie die beiden Bytes der Instruktionen (je nach CPU Ihres Rechners) nach dem Einlesen vertauschen. Lesen Sie hierzu beide Bytes als 2-Byte Integer ein und führen Sie die Konvertierung anschließend mit der Funktion `ntohs` aus. Diese ist im Header *arpa/inet.h* deklariert.