Michael Lehn

High Performance Computing

Grundlagen für Cache-optimierte numerische Verfahren

Springer

Version 1
Mai 2015

Lektorat: Barbara Lühker, Bianca Alton
Satz: Markus Kohm, Steingraeber Satztechnik GmbH
Herstellung: Ute Kreutzer

Inhaltsverzeichnis

1 ULM - Ulm Lecture Machine

Übersicht

Anhand des fiktiven Computers *ULM (Ulm Lecture Machine)* soll in diesem Kapitel die prinzipielle Funktionsweise heutiger Computer beschrieben werden. Die Architektur der ULM orientiert sich an Konzepten der sogenannten *RISC (Reduced Instruction Set Computer)* Architektur. An vielen Stellen werden aber Bezeichnungen benutzt, die von der Intel64-Architektur stammen. Obwohl wir uns in diesem Kapitel mit einer sehr hardwarenahen Thematik beschäftigen, liegt ein weiterer Schwerpunkt darin, zu vermitteln, wie Algorithmen zunächst als Pseudo-Code beschrieben und dann in einer Programmiersprache umgesetzt werden können.

1.1 Aufbau der ULM-Rechnerarchitektur

In Abbildung 1.1 zeigen wir die wesentlichen Komponenten und den schematischen Aufbau der Architektur. Für den sogenannten *Hauptprozessor* des Computers verwenden wir meist die gängige Bezeichnung *CPU (central processing unit)*. Der *Hauptspeicher (Memory)* enthält Befehle und Daten, die über den *Datenbus (Bus)* in die CPU geladen oder von dort in den Speicher zurückgeschrieben werden können. Die Programmierung des Computers basiert auf dem sogenannten *Von-Neumann-Zyklus*. Dieser legt fest, wie Befehle sequentiell aus dem Hauptspeicher in die CPU geladen und dort ausgeführt werden. Der Datenbus unserer Architektur sei außerdem mit einer Tastatur und einem Terminal verbunden, die jeweils als *Eingabe- und Ausgabegeräte* dienen. In der Abbildung sind diese mit *I/O (input/output device)* angedeutet.

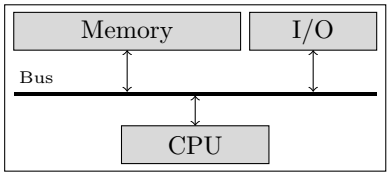

Abb. 1.1: Konzeptionelle Darstellung der Architektur.

1.1.1 Computerzahlen und Zeichencodierung

Wie bei allen modernen Computern werden bei der ULM-Architektur Befehle und Daten durch eine Folge von Nullen und Einsen codiert. Mit einem *Bit (binary digit)* bezeichnen wir eine Zahl $b \in \{0, 1\}$ und mit *Bitmuster* eine Liste einzelner Bits. Ein Bitmuster kann decodiert werden als *Unsigned-Integer (nicht-negative ganze Zahl)*, *Signed-Integer (ganze Zahl)*, *Floating-Point-Number (Fließkommazahl)* oder *Character (Textzeichen)*.

Mit einem *Byte* bezeichnen wir ein Bitmuster mit acht Bits. Wie in Tabelle 1.1 aufgelistet, werden weitere Größen für Bitmuster als *Word* (2 Bytes), *Long-Word* (4 Bytes) und *Quad-Word* (8 Bytes)

Unsigned-Integer

Für die Codierung nicht-negativer ganzer Zahlen betrachten wir zunächst deren Zahlendarstellung bezüglich einer beliebigen Basis. Daraus kann anschließend leicht die Umwandlung von Dezimalzahlen in Binärzahlen und deren kompaktere Darstellung als Hexadezimalzahlen hergeleitet werden.

Definition 1.1 (b-adische Darstellung für Unsigned-Integer)
Seien $b, n \in \mathbb{N}$ mit $b \geq 2$. Für beliebige Zahlen $a_0, \ldots, a_{n-1} \in \{0, \ldots, b-1\}$ definieren wir

$$(a_{n-1}, \ldots, a_0)_b := \sum_{\ell=0}^{n-1} a_\ell b^\ell. \tag{1.1}$$

Tab. 1.1: Bezeichnungen und Vereinbarungen für Datenbreiten.

Bits	Bytes	Beschreibung	Code
8	1	Byte	b
16	2	Word	w
32	4	Long-Word	l
64	8	Quad-Word	q

Wir sagen, der in (1.1) definierte Ausdruck ist eine n-stellige Darstellung einer nicht-negativen ganzen Zahl, kurz Unsigned-Integer, zur Basis b mit Ziffern $a_{n-1}, \ldots, a_0$. ♦

Mit (1.1) können nur Zahlen $z \in \mathbb{U}_{n,b} := \{0, \ldots, b^n - 1\}$ dargestellt werden, denn es gilt

$$0 \leq \sum_{\ell=0}^{n-1} a_\ell b^\ell \leq \sum_{\ell=0}^{n-1} (b-1) b^\ell = b^n - 1. \tag{1.2}$$

Für jede Zahl z aus diesem Bereich existiert eine eindeutige Darstellung, die durch

$$a_\ell = \left\lfloor \frac{z}{b^\ell} \right\rfloor \bmod b \quad \text{für} \quad 0 \leq \ell < n \tag{1.3}$$

festgelegt ist.

Soll ein Byte, Word, Long-Word oder Quad-Word als Unsigned-Integer interpretiert werden, sagen wir jeweils verkürzt *Unsigned-Byte*, *Unsigned-Word*, *Unsigned-Long-Word* oder *Unsigned-Quad*. Bei der Bezeichnung ihrer Wertebereiche geben wir die Basis $b = 2$ nicht explizit an, sondern definieren

$$\begin{aligned} \mathbb{U}_8 &:= \mathbb{U}_{8,2} &&= \{0, \ldots, 255\}, \\ \mathbb{U}_{16} &:= \mathbb{U}_{16,2} &&= \{0, \ldots, 2^{16} - 1\}, \\ \mathbb{U}_{32} &:= \mathbb{U}_{32,2} &&= \{0, \ldots, 2^{32} - 1\}, \\ \mathbb{U}_{64} &:= \mathbb{U}_{64,2} &&= \{0, \ldots, 2^{64} - 1\}. \end{aligned} \tag{1.4}$$

Umwandlung von Dezimalzahlen in Binärzahlen Für eine nicht-negative Dezimalzahl z kann zunächst mit (1.2) durch

$$n = \left\lceil \frac{\log(z+1)}{\log 2} \right\rceil$$

die minimale Anzahl an notwendigen Stellen für die Binär-Darstellung berechnet werden. Die einzelnen Ziffern können dann mit (1.3) und $b = 2$ berechnet werden.

Darstellung von Binär- und Hexadezimalzahlen Bei Hexadezimalzahlen (Basis $b = 16$) werden die Werte von 10 bis 16 durch die Buchstaben 'A' bis 'F' als einzelne Ziffern codiert. Vier Bits können so durch eine einzelne Ziffer codiert werden:

$$\begin{aligned} &0 = (0,0,0,0)_2,\ 1 = (0,0,0,1)_2,\ 2 = (0,0,1,0)_2,\ 3 = (0,0,1,1)_2, \\ &4 = (0,1,0,0)_2,\ 5 = (0,1,0,1)_2,\ 6 = (0,1,1,0)_2,\ 7 = (0,1,1,1)_2, \\ &8 = (1,0,0,0)_2,\ 9 = (1,0,0,1)_2,\ A = (1,0,1,0)_2,\ B = (1,0,1,1)_2, \\ &C = (1,1,0,0)_2,\ D = (1,1,0,1)_2,\ E = (1,1,1,0)_2,\ F = (1,1,1,1)_2. \end{aligned}$$

Ein Byte kann dann durch eine zweistellige Hexadezimalzahl dargestellt werden, denn

$$\begin{aligned}(h_1,h_0)_{16} &= h_1\cdot 2^4+h_0=(b_7,b_6,b_5,b_4)_2\cdot 2^4+(b_3,b_2,b_1,b_0)_2\\ &= 2^4\cdot\sum_{\ell=0}^{3}b_{4+\ell}2^\ell+\sum_{\ell=0}^{3}b_\ell 2^\ell=\sum_{\ell=0}^{7}b_\ell 2^\ell\\ &= (b_7,\ldots,b_0)_2\end{aligned}$$

Dies kann natürlich leicht für eine beliebige Stellenzahl verallgemeinert werden.

Signed-Integer

Ganze Zahlen (d. h. mit eventuell negativem Vorzeichen) werden als Binärzahlen mit der Definition

$$\mathrm{s}(a_{n-1},\ldots,a_0)_2 := (a_{n-1},\ldots,a_0)_2 - a_{n-1}\cdot 2^n \tag{1.5}$$

als sogenanntes *Zweierkomplement* codiert. Wir nennen diese Interpretation des Bitmusters als n-tesllige *Signed-Integer*. Offensichtlich gilt

$$z=\mathrm{s}(a_{n-1},a_{n-2}\ldots,a_0)_2=\begin{cases}(0,a_{n-2},\ldots,a_0)_2, & a_{n-1}=0,\\ (1,a_{n-2},\ldots,a_0)_2-2^n, & a_{n-1}=1.\end{cases}$$

Wegen

$$(0,a_{n-2},\ldots,a_0)_2\in\mathbb{U}_{n-1,2}=\{0,\ldots,2^{n-1}-1\}$$

und

$$\begin{aligned}(1,a_{n-2},\ldots,a_0)_2-2^n &= 2^{n-1}+(a_{n-2},\ldots,a_0)_2-2^n\\ &= (a_{n-2},\ldots,a_0)_2-2^{n-1}\\ &\in \{-2^{n-1},\ldots,-1\}\end{aligned}$$

ist der darstellbare Wertebereich

$$\mathbb{S}_n := \{-2^{n-1},\ldots,-1,0,\ldots,2^{n-1}-1\}$$

und es gilt

$$z\geq 0 \quad\Longleftrightarrow\quad a_{n-1}=0.$$

Das Vorzeichen der dargestellten Zahl kann somit am höchstwertige Bit a_{n-1} abgelesen werden und wird deshalb als *Sign-Bit* einer Signed-Integer bezeichnet.

In Abbildung 1.2 sollen für den Fall $n=3$ die Interpretation eines Bitmusters als Unsigned-Integer oder Signed-Integer veranschaulicht und verglichen werden:

- Bei einer Unsigned-Integer (links) werden die darstellbaren Zahlen $0,\ldots,7$ gleichmäßig auf dem Einheitskreis verteilt, in dem wir durch

  $$\varphi(k)=k\cdot\frac{2\pi}{8},\quad k\in\mathbb{U}_3=\{0,\ldots,7\}$$

  jeder Zahl einen Winkel aus dem Intervall $[0,2\pi)$ zuordnen.

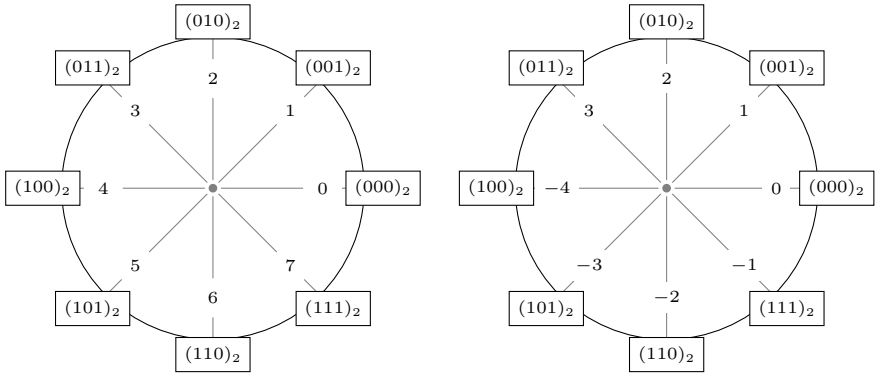

Abb. 1.2: Interpretation einer dreistelligen Binärzahl als Unsigned-Integer (links) und Signed-Integer (rechts).

- Bei einer Signed-Integer (rechts) wird durch

$$\varphi(k) = k \cdot \frac{2\pi}{8}, \quad k \in \mathbb{S}_3 = \{-4, \ldots, -1, 0, \ldots, 3\}$$

  stets ein Winkel im Intervall $[-\pi, \pi)$ zugeordnet.

Die Addition und Subtraktion entspricht in beiden Fällen dem Addieren und Subtrahieren der zugeordneten Winkel:

- Sind $k, \ell \in \mathbb{U}_3$ dann gilt

$$\varphi(k) + \varphi(\ell) = \varphi(k + \ell), \quad \text{falls } k + \ell \in \mathbb{U}_3,$$

  und

$$\varphi(k) - \varphi(\ell) = \varphi(k - \ell), \quad \text{falls } k - \ell \in \mathbb{U}_3.$$

  Dass eine Summe oder Differenz nicht darstellbar ist, ist durch

$$\begin{aligned} k + \ell \notin \mathbb{U}_3 &\iff \varphi(k) + \varphi(\ell) \notin [0, 2\pi) \\ k - \ell \notin \mathbb{U}_3 &\iff \varphi(k) - \varphi(\ell) \notin [0, 2\pi) \end{aligned}$$

  charakterisiert.
- Für $k, \ell \in \mathbb{S}_3$ dann gilt analog

$$\begin{aligned} \varphi(k) + \varphi(\ell) &= \varphi(k + \ell), \quad \text{falls } k + \ell \in \mathbb{S}_3, \\ \varphi(k) - \varphi(\ell) &= \varphi(k - \ell), \quad \text{falls } k - \ell \in \mathbb{S}_3. \end{aligned}$$

  und

$$\begin{aligned} k + \ell \notin \mathbb{S}_3 &\iff \varphi(k) + \varphi(\ell) \notin [-\pi, \pi), \\ k - \ell \notin \mathbb{S}_3 &\iff \varphi(k) - \varphi(\ell) \notin [-\pi, \pi). \end{aligned}$$

Floating-Point-Number

Die p niedrigstwertigen Bits werden für die Mantisse, die r höherwertigen Bits für den Exponenten und das höchstwertige Bit für das Vorzeichen verwendet. Durch $n = p + r + 1$ Bits wird eine Fließkommazahl durch

$$(b_{p+r}, \ldots, b_0)_2^f := s \cdot (1 + M/2^p) \cdot 2^{E-B}$$

mit

$$\begin{aligned} M &:= (b_{p-1}, \ldots, b_0)_2 \\ E &:= (b_{r+p-1}, \ldots, b_p)_2 \\ B &:= 2^{r-1} - 1 \\ s &:= (-1)^{b_{r+p}} \end{aligned}$$

definiert. Der IEEE-754-Standard definiert die Aufteilung der n Bits in die Länge der Mantisse und des Exponenten. Bei der ULM-Architektur unterstützen wir die beiden Formate:

- *Single-Precision:* $n = 32$ Bits werden in eine Mantisse mit $p = 23$ Bits und einen Exponenten mit $r = 8$ Bits aufgeteilt.
- *Double-Precision:* $n = 64$ Bits werden in eine Mantisse mit $p = 52$ Bits und einen Exponenten mit $r = 11$ Bits aufgeteilt.

Character-Tabellen

Etwas zur ASCII-Tabelle:

- 'A' wird als 65, 'B' als 66, ..., 'Z' als 80 codiert.
- 'a' wird als 97, 'b' als 98, ..., 'z' als 112 codiert.
- Mit 0 wird das Ende einer Zeichenkette (String) markiert.
- '!' wird mit 33 codiert

Für ein *Hello world!* Programm reicht das...

1.1.2 Hauptprozessor (CPU)

Bild muss noch angepasst werden...

Beschreibung von Registerinhalten

Die Inhalte der 64-Bit-Register bezeichnen wir in Anlehnung an GAS-Syntax für den Intel64-Assembler mit $\%0, \%1, \ldots, \%255$. An manchen Stellen wird es not-

wendig sein, auf den Wert eines einzelnen Bytes in einem Register zu verweisen. Ist

$$\%X = (h_{15}, \ldots, h_0)_{16} = (B_7, \ldots, B_0)_{256}$$

dann soll beispielsweise mit

$$\%X|_{3:0} := (h_7, \ldots, h_0)_{16} = (B_3, \ldots, B_0)_{256}$$

die Restriktion auf die niedrigstwertigen 4 Bytes bezeichnet werden.

Auffüllen von Bitmuster

Häufig werden wir ein Byte, Word oder Long-Word zu einem Quad-Word erweitern müssen. Stellt das Bitmuster eine ganze Zahl dar, so soll der ursprüngliche Wert erhalten bleiben. In diesem Fall müssen wir unterscheiden, ob durch das Bitmuster ein Unsigned-Integer oder ein Signed-Integer dargestellt wird. Im Folgenden sei $k \in \{1, 2, 4\}$ die Anzahl der Bytes, die zu einem Quad-Word erweitert werden soll.

Im Fall einer Unsigned-Integer $(h_{2k-1}, \ldots, h_0)_{16}$ erhalten wir das gesuchte Quad-Word, indem wir die $8 - k$ höchstwertigen Bytes auf Null setzen, denn

$$(h_{2k-1}, \ldots, h_0)_{16} = (0, \ldots, 0, h_{2k-1}, \ldots, h_0)_{16}. \tag{1.6}$$

Eine Erweiterung des Bitmusters dieser Art werden wir als *Zero-Padding* bezeichnen.

Bei einer Signed-Integer müssen die höchstwertigen Bytes abhängig vom Vorzeichen der Zahl aufgefüllt werden. Es gilt

$$z = S(h_{2k-1}, \ldots, h_0)_{16} = \begin{cases} S(0, \ldots, 0, h_{2k-1}, \ldots, h_0)_{16}, & z \geq 0, \\ S(F, \ldots, F, h_{2k-1}, \ldots, h_0)_{16}, & z < 0. \end{cases} \tag{1.7}$$

Wir werden dies als *Signed-Padding* bezeichnen.

Spezielle Register

- *Zero Register*
  Das Register 0 ist ein sogenanntes Zero-Register. Die 64 Bits dieses Registers sind stets Null. Dies kann auch nicht durch Rechenoperationen verändert werden.
- *Instruction Pointer*
  Das Register 1 enthält dir Adresse des Befehls, der gerade ausgeführt wird.
- *Instruction Register*
  Das Register 2 enthält in den unteren vier Bytes den 32-Bit Befehl, der gerade ausgeführt wird.

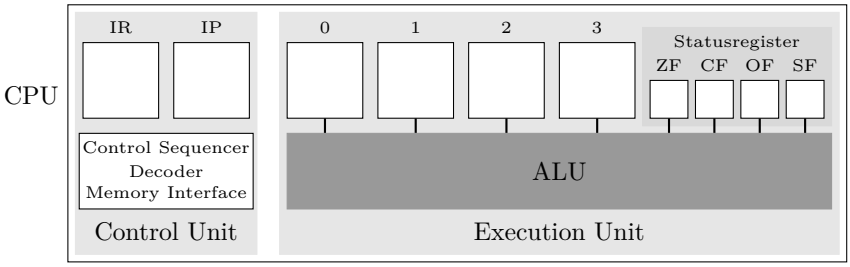

Abb. 1.3: Konzeptionelle Hervorhebung des Rechenwerks innerhalb einer CPU.

1.1.3 Speicher (Memory)

Der Speicher oder *RAM (Random-Access-Memory)* ist unterteilt in adressierbare *Speicherinhalte* oder *Speicherzellen*. Auf unserer Architektur besteht eine Zelle aus einem Byte. Der *Adressraum* ist die Menge der gültigen Adressen und ist bei der ULM der Wertebereich einer 64-Bit-Unsigned-Integer. Mit $M_1(k)$ bezeichnen wir den Speicherinhalt des Bytes bei Adresse k und mit

$$M_n(k) := (M_1(k), \ldots, M_1(k+n-1)) \tag{1.8}$$

die Werte der n Bytes, die beginnend bei Adresse k aufeinanderfolgend im Speicher liegen.

Für das Speichern von Daten, die aus mehr als einem Byte bestehen, verwendet die ULM das *Big-Endian-Format*. Das höchstwertige Byte wird dabei an der niedrigsten Adresse und das niedrigstwertige Byte an der höchsten Adresse abgelegt. Ein Speicherinhalt $M_8(k)$ kann somit als das Unsigned-Quad-Word

$$(M_1(k), \ldots, M_1(k+7))_{256}$$

oder das Signed-Quad-Word

$$(M_1(k), \ldots, M_1(k+7))^s_{256}$$

interpretiert werden.

1.1.4 Datenbus (Bus)

Mit dem Datenbus können von einer Adresse k ein einzelnes Byte ($n = 1$), ein Word ($n = 2$), ein Long-Word ($n = 4$) oder ein Quad-Word ($n = 8$) in ein Register geladen werden oder von dort in den Speicher zurückgeschrieben werden. Wir bezeichnen diese Operationen jeweils als *Load-Register* oder *Store-Register*. Technisch bedingt, können auf der ULM diese Operationen allerdings nur für Daten ausgeführt werden, deren Adresse durch n teilbar ist. Daten mit dieser Eigenschaft

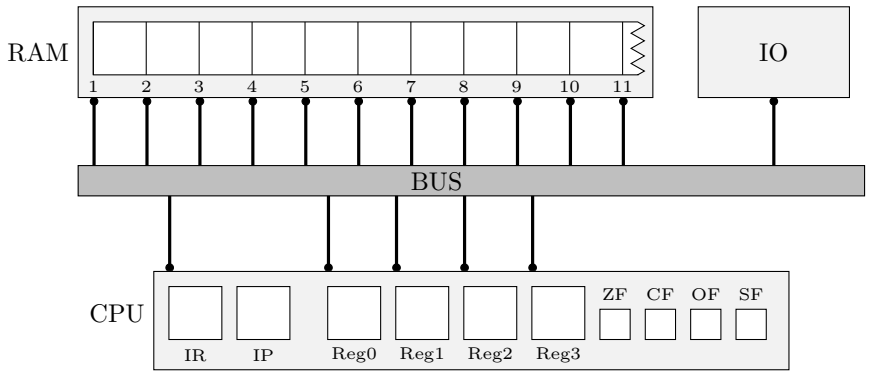

Abb. 1.4: Datenbus

bezeichnet man als *aligned* oder *im Speicher ausgerichtet*. Bei einem Load-Register oder Store-Register muss für den zu übertragenden Speicherinhalt $M_n(k)$ somit

$$k \in \{0, \ldots, 2^{64} - 1\}, \quad n \in \{1, 2, 4, 8\} \quad \text{und} \quad n \bmod k = 0. \tag{1.9}$$

gelten.

Load-Register

Beim Laden wird stets das gesamte 64-Bit-Register geschrieben. Sollen weniger als ein Quad-Word geladen werden, muss festgelegt werden wie die höherwertigen Bytes überschrieben werden sollen. Mit $M_n^z(k)$ und $M_n^s(k)$ bezeichnen wir jeweils die Erweiterung durch Zero-Padding oder Signed-Padding. Die zwei möglichen Varianten für das Laden eines Long-Word von Adresse k in Register X können wir somit durch $M_4^z \to \%X$ und $M_4^s \to \%X$ formal beschreiben.

Store-Register

Mit $\%X \to M_8(k)$ drücken wir aus, dass der gesamte Registerinhalt $\%X$ in den Speicher geschrieben wird. Weitere Varianten erlauben das Speichern des niedrigstwertige Long-Word, Word oder Byte. Allgemein können alle Varianten für $n \in \{1, 2, 4, 8\}$ zusammengefasst werden mit $\%X|_{n:0} \to M_n(k)$.

1.1.5 Rechenwerk (ALU)

Wir betrachten zunächst nur die Addition, Subtraktion und Multiplikation. Die Operationen können im Unsigned oder Signed Modus ausgeführt werden. Abhängig vom Ergebnis der Operation werden in der ALU folgende Statusbits gesetzt:

- *ZF (Zero Flag)*
  Beim Ergebnis sind alle Bits auf Null gesetzt.
- *SF (Sign Flag)*
  Beim Ergebnis ist das höchstwertige Bit gesetzt.
- *OF (Overflow Flag)*
  Das Ergebnis einer Addition oder Subtraktion von zwei Signed-Integer a und b liegt nicht im Wertebereich $\{-2^{63}, \ldots, 2^{63}-1\}$ einer 64-Bit Signed-Integer.
- *CF (Carry Flag)* Das Ergebnis einer Addition oder Subtraktion von zwei Signed-Integer a und b liegt nicht im Wertebereich $\{0, \ldots, 2^{64}-1\}$ einer 64-Bit Unsigned-Integer.

Subtraktion $b-a$	CF	ZF
$a=b$	0	1
$a<b$	1	0
$a>b$	0	0

Diese Statusbits bleiben gesetzt, bis sie von einer weiteren Rechenoperation überschrieben werden.

1.1.6 Datenbus (Bus)

Der Datenbus stellt die Operationen *Fetch* und *Store* zur Verfügung, um Daten vom Speicher jeweils in Register zu laden oder zurückzuschreiben. Beide Operationen werden durch die drei Argumente *Startadresse*, *Anzahl Bytes* und *Register-Name* festgelegt:

- Es können mit einer Fetch oder Store Operation ein Byte, Word, Long-Word oder ein Quad-Word zwischen Speicher und Register bewegt werden.
- Die Adresse im Speicher muss durch die Anzahl der Bytes teilbar sein.

1.1.7 Übungsaufgaben

1. Bestimmen Sie jeweils die Hexadezimal-Darstellung von

   a) 123 als Unsigned-Byte, Signed-Byte, Unsigned-Word und Signed-Word.

2. Zeigen Sie: Für die Erweiterung der n-stelligen Hexadezimal-Darstellung einer Signed-Integer gilt

$$z = (h_{n-1}, \ldots, h_0)_{16} = \begin{cases} (0, \ldots, 0, h_{n-1}, \ldots, h_0)_{16}, & z \geq 0, \\ (\mathrm{F}, \ldots, \mathrm{F}, h_{n-1}, \ldots, h_0)_{16}, & z < 0. \end{cases}$$

3. ...

1.2 Befehlssatz der ULM

Ein Maschinenbefehl besteht aus 32 Bits, die zur Ausführung in die unteren vier Bytes des Befehlsregisters geladen werden. Diese vier Bytes bezeichnen wir der Einfachheit halber mit

$$\%2|_{3:0} = (OP, X, Y, Z)_{256}.$$

Das höchstwertige Byte OP codiert die Operation, die restlichen drei Bytes können als Operanden benutzt werden. Wir unterschieden drei Typen von Operanden:

- *Inhalt eines Register*
  Ein Operand wie X soll als Register-Nummer interpretiert werden, und die Operation den Registerinhalt $\%X$ verwenden.
- *Immediate-Value*
  Ein Operand wie X soll als Zahlenwert $(X)_{256}$ oder $(X)^s_{256}$ benutzt werden. Bei der Beschreibung der Maschinenbefehle deuten wir dies mit X oder X^s an. In manchen Fällen soll mehr als ein Operand benutzt werden, um einen konstanten Wert zu codieren. Wir deuten dies beispielsweise mit XY für $(X, Y)_{256}$ oder XYZ^s für $(X, Y, Z)^s_{256}$ an.
- *Speicherinhalt*
  Letztlich können Operanden auch als Adressen interpretiert werden. Durch den Op-Code des Maschinenbefehl wird dann außerdem festgelegt, wie viele Bytes an dieser Adresse referenziert werden. Mit $M_8(\%X + Y)$ deuten wir an, dass sich die Adresse als Summe des Registerinhaltes $\%X$ und des Immediate-Value Y berechnet. An der Operation beteiligt sind dann die acht darauf folgenden Bytes.

1.2.1 Laden und Speichern von Daten

Laden eines Quad-Word in ein Register

Unsere Architektur stellt zwei Befehle zur Verfügung, um aus dem Speicher ein Quad-Word in ein Register zu laden.

Op	Funktion	Assembler
10	$M_8(\%X + \%Y) \to \%Z$	*movq (%X,%Y), %Z*
11	$M_8(\%X + Y^s) \to \%Z$	*movq Y(%X), %Z*

Laden eines Long-Word, Word oder Byte

Sollen weniger als acht Byte in ein Register geladen werden, so werden diese in die niederwertigen Bytes geschrieben. Die höherwertigen Bytes werden abhängig

davon, ob die Zahl als Signed- oder Unsigned-Integer interpretiert werden soll, mit Nullen oder Einsen gefüllt. Im Fall einer Unsigned-Integer werden die oberen Bytes stets mit Nullen überschrieben. Bei einer Signed-Integer dagegen werden die oberen Bytes mit dem Vorzeichen-Bit aufgefüllt.

Store-Befehle: Register-Inhalte in RAM schreiben

Schreiben eines Quad-Word:

Op	Funktion	Assembler
40	$\%X \rightarrow M_8(\%Y + \%Z)$	*movq %X, (%Y,%Z)*
41	$\%X \rightarrow M_8(\%Y + Z^s)$	*movq %X, Z(%Y)*

1.2.2 Integer-Arithmetik

Addition und Subtraktion

Die Addition und Subtraktion wird sowohl für Signed-Integer als auch Unsigned-Integer ausgeführt. Alle Status-Register werden entsprechend gesetzt. Zu beachten ist, dass sich bei diesen Operationen das Bitmuster des Ergebnisses nicht unterscheidet. Soll die Operation für Unsigned-Integer interpretiert werdeni, sind nur das Carry-Flag und das Zero-Flag relevant. Bei Signed-Integer Operationen sind nur das Overflow-Flag, Sign-Flag und Zero-Flag relevant.

Op	Funktion	Assembler
60	$\%X + \%Y \rightarrow \%Z$	*addq %X, %Y, %Z*
61	$X^s + \%Y \rightarrow \%Z$	*addq $X, %Y, %Z*
62	$\%Y - \%X \rightarrow \%Z$	*subq %X, %Y, %Z*
63	$\%Y - X^s \rightarrow \%Z$	*subq $X, %Y, %Z*

Multiplikation und Division

Zunächst nur eine Multiplikation von Unsigned-Integer ...

Op	Funktion	Assembler
70	$\%X \cdot \%Y \rightarrow \%Z$	*mulq %X, %Y, %Z*
71	$X \cdot \%Y \rightarrow \%Z$	*mulq $X, %Y, %Z*

1.2.3 Floating-Point-Arithmetik

1.2.4 Bit-Operation

Dinge wie bitweise Und-Verknüpfung, Oder-Verknüpfung, Bit-Shifts, ...

Op	Funktion	Assembler
80	$\%X \vee \%Y \to \%Z$	*orq* *%X, %Y, %Z*
81	$X \vee \%Y \to \%Z$	*orq* *$X, %Y, %Z*
82	$\%X \wedge \%Y \to \%Z$	*andq* *%X, %Y, %Z*
83	$X \wedge Y \to \%Z$	*andq* *$X, %Y, %Z*
84	$\%Y \ll \%X \to \%Z$	*shlq* *%X, %Y, %Z*
85	$\%Y \ll X \to \%Z$	*shlq* *$X, %Y, %Z*
86	$\%Y \gg \%X \to \%Z$	*shrq* *%X, %Y, %Z*
87	$\%Y \gg X \to \%Z$	*shrq* *$X, %Y, %Z*

1.2.5 Sprungbefehle

Unbedingte Sprünge

Op	Funktion	Assembler
$9A$	$\%1 + 4 \cdot XYZ^s \to \%1$	*jmp* *$XYZ*
$9B$	$\%1 + 4 \to \%Y$, $\%X \to \%1$	*jmp* *%X*, %Y
$9C$	$\%X \to \%1$	*jmp* %X

Bedingte Sprünge

Bei bedingten Sprüngen wird ein Sprung in Abhängigkeit der Status-Flags durchgeführt. Sprünge sind relativ zur Adresse des Sprungbefehles. Die Weite des Sprunges wird durch ein Immediate-Value festgelegt, das als Signed-Integer interpretiert wird. Da ein Befehl nur an einer durch vier teilbaren Adresse liegen kann, wird die Sprungweite implizit mit vier multipliziert. Es kann also sowohl 2^{28} Bytes vor- oder zurückgesprungen werden.

Op	Bedingung	Funktion	Assembler
90	ZF = 1	$\%1 + 4 \cdot XYZ^s \to \%1$	*jz* *$XYZ*
91	ZF = 0	$\%1 + 4 \cdot XYZ^s \to \%1$	*jnz* *$XYZ*

Op	Bedingung	Funktion	Assembler
92	SF $\neq$ OF	$\%1 + 4 \cdot XYZ^s \rightarrow \%1$	*jl* *$XYZ*
93	SF = OF	$\%1 + 4 \cdot XYZ^s \rightarrow \%1$	*jge* *$XYZ*
94	ZF $\vee$ (SF $\neq$ OF)	$\%1 + 4 \cdot XYZ^s \rightarrow \%1$	*jle* *$XYZ*
95	$\neg$ZF $\wedge$ (SF = OF)	$\%1 + 4 \cdot XYZ^s \rightarrow \%1$	*jg* *$XYZ*
Op	Bedingung	Funktion	Assembler
96	CF	$\%1 + 4 \cdot XYZ^s \rightarrow \%1$	*jb* *$XYZ*
97	$\neg$CF	$\%1 + 4 \cdot XYZ^s \rightarrow \%1$	*jnb* *$XYZ*
98	CF $\vee$ ZF	$\%1 + 4 \cdot XYZ^s \rightarrow \%1$	*jbe* *$XYZ*
99	$\neg$CF $\wedge$ $\neg$ZF	$\%1 + 4 \cdot XYZ^s \rightarrow \%1$	*ja* *$XYZ*

1.2.6 Übungsaufgaben

1. *Assemblierung*

   Bestimmen Sie die Maschinencodes für die Assemblerbefehle

```
orq     %5,     %0,     %12
movq    %5,     (%16,%7)
movq    %5,     7(%16,7)
subq    $2,     %5,     %0
jl      $-3
addq    $-2,    %5,     %8
```

2. *Disassemblierung*

   Folgende Zeilen stellen jeweils einen Maschinenbefehl als Hexadezimalzahl dar:

```
81 2A 00
05 81 02
00 06 70
05 06 07
```

   Bestimmen Sie jeweils den dazugehörigen Assemblerbefehl.

3. Status-Flags

   Bezeichne a und b jeweils die Inhalte von Register 5 und 6. Geben Sie mögliche Werte für a und b an, so dass nach

```
subq    %5,     %6,     %0
```

für die Status-Flags jeweils gilt

a) SF $\neq$ OF

b) SF = OF

1.3 Sequentielle Programmausführung

Abbildung 1.5 zeigte den Zustand der ULM unmittelbar bevor der erste CPU-Takt ausgeführt wird. Beginnend bei Adresse 0 liegt ein Maschinenprogramm

```
63 01 00 05 92 00 00 02 61 01 05 05
```

im Speicher. Die Bits des Register IP sind alle auf Null gesetzt. Bis auf das Zero-Register enthalten alle andere Register undefinierte Werte und werden hier nicht dargestellt.

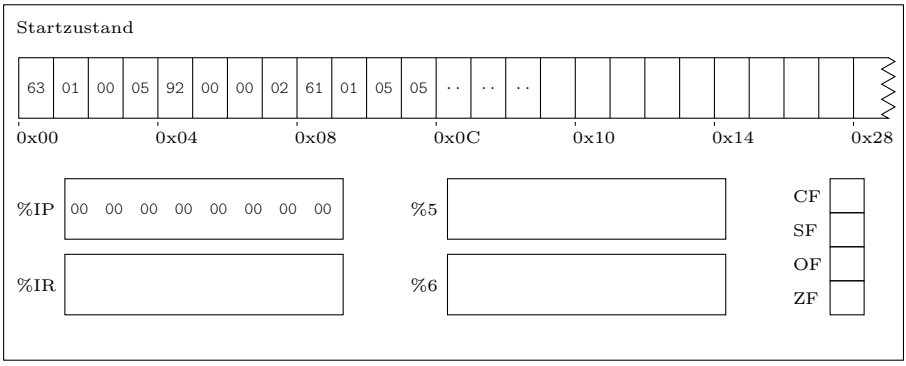

Abb. 1.5: Zustand der ULM vor dem ersten CPU-Takt.

1.3.1 Von-Neumann-Zyklus

Ein Programm wird ausgeführt, indem das Steuerwerk mit einer gewissen Taktrate ein Befehl nach dem anderen ausführt. Innerhalb dieser sogenannten *CPU Taktrate* wird der Befehl wiederum schrittweise ausgeführt. Dies Schritte werden in diesem Kontext die *Stufen* oder *Stages* der Befehlsausführung genannt und mit dem *Von-Neumann Taktzyklus* wie folgt festgelegt:

(A) **Fetch**

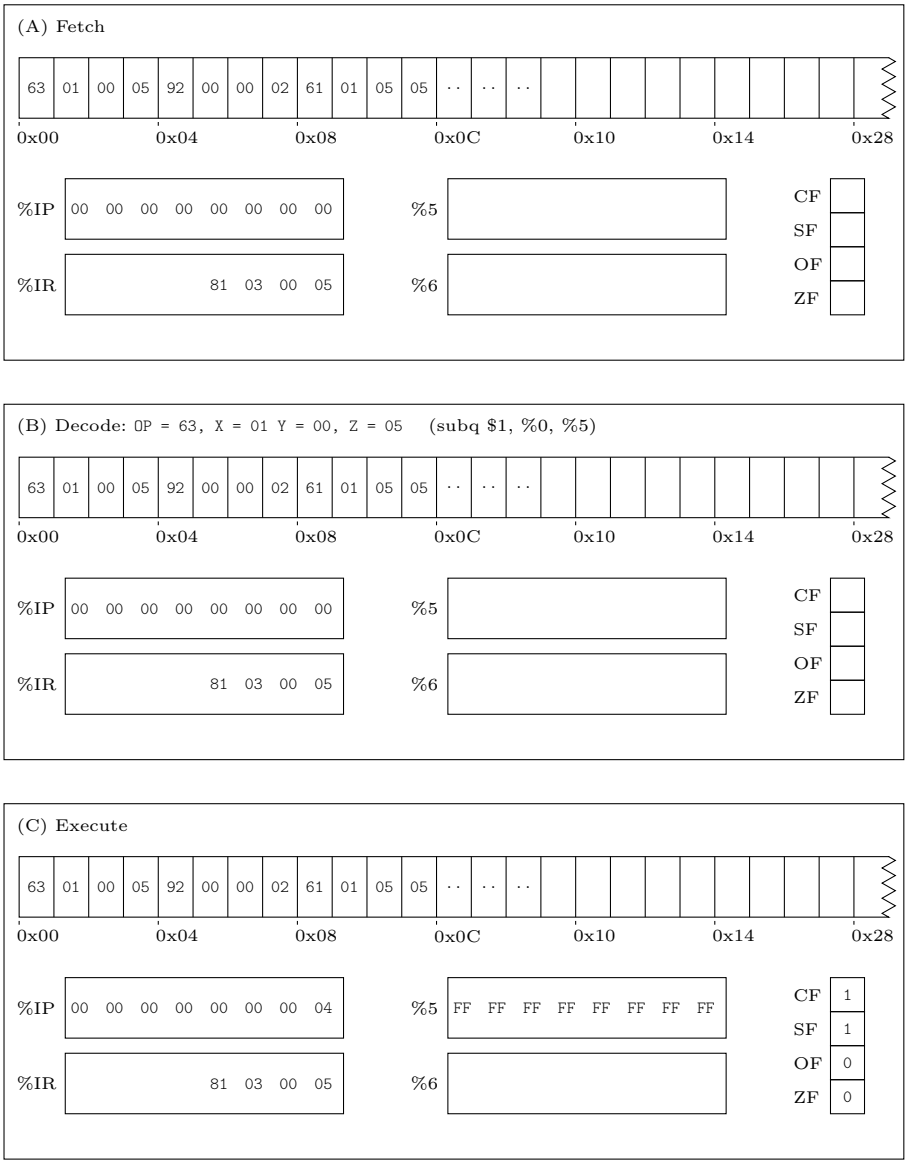

Abb. 1.6: Durchlaufen eines Von-Neumann-Zyklus

Von der Adresse %IP wird ein 32-Bit in das Instruction-Register IR geladen:

$$M_4^z\,(\%\text{IP}) \rightarrow \%\text{IR}$$

Im ersten Takt wird somit stets der Befehl in $M_4^z(0)$ ausgeführt.

(B) **Decode**

Der ins **IR** Register geladene Befehl wird in einen internen Befehl für die Architektur decodiert:

$$\%\text{IR} \rightarrow (\text{OP}, \text{X}, \text{Y}, \text{Z})$$

(C) **Execute**

Der Befehl $(\text{OP}, \text{X}, \text{Y}, \text{Z})$ wird ausgeführt. Handelt es sich dabei nicht um einen Sprungbefehl, der den Inhalt %IP von Register IP verändert hat, wird %IP um Eins erhöht.

Diese Schritte werden solange wiederholt, bis die Programmausführung durch einen Halt-Befehl ($\text{OP} = 00$) beendet wird.

1.3.2 Der ULM-Simulator

Mit dem Simulator für die ULM kann ein Programm schrittweise ausgeführt werden. Analog zu den obigen graphischen Darstellungen, kann dabei der Inhalt von Speicherflächen und Registern beobachtet werden.

Listing 1.1: ULM-Maschinencode zur Berechnung der Fakultät

```
1  81 28 00 05    # 0000 :  orq     $40,    %0,     %5
2  11 05 00 05    # 0004 :  movq    (%5),   %5
3  81 02 00 06    # 0008 :  orq     $2,     %0,     %6
4  90 00 00 03    # 000C :  jmp     $3
5  70 05 06 06    # 0010 :  mulq    %5,     %6,     %6
6  63 01 05 05    # 0014 :  subq    $1,     %5,     %5
7  63 01 05 00    # 0018 :  subq    $1,     %5,     %0
8  95 FF FF FD    # 001C :  jg      $-3
9  41 06 01 08    # 0020 :  movq    %6,     8(%1)
10 00 00 00 00    # 0024 :  halt
11 00 00 00 00    # 0028
12 00 00 00 03    # 002C
```

1.3.3 Übungsaufgaben

1. Die ULM enthalte beginnend bei Adresse 0 folgenden Zahlen .... Führen Sie das Programm zunächst mit Papier und Bleistift aus. Kontrollieren Sie dann mit dem ULM-Simulator ihre Schritte.

1.4 Der ULM-Assembler

Mit dem ULM-Assembler kann aus einem *Assembler-Programm* auf relativ einfache Weise Maschinencode erzeugt werden. Ein Assembler-Programm besteht dabei aus Zeilen, die entweder einen *Assembler-Befehl* oder eine *Assembler-Anweisung* enthalten. Mit Assemblerbefehl bezeichnen wir eine Zeile, wie etwa

```
subq    $8,     %3,     %3
```

die anhand der Tabellen in Abschnitt 1.2 direkt in den entsprechenden Maschinencode

```
64 08 03 03
```

umgewandelt werden kann. Mit den *Assembler-Anweisungen* dagegen kann der ULM-Assembler selbst gesteuert werden. Beispielsweise kann mit der Anweisung *.equ RSP, %3* vereinbart werden, dass das Symbol *RSP* in allen folgenden Zeilen durch %3 zu ersetzen ist. Bevor der Assembler letztlich Maschinencode erzeugt, werden zuerst alle Assembler-Anweisungen ausgeführt. Der dadurch erzeugte Programmtext enthält dann nur noch Assemblerbefehle, die dann sukzessive codiert werden können. Um Anweisungen und Befehle besser voneinander unterscheiden zu können, beginnen Assembler-Anweisungen stets mit einem Punkt.

1.4.1 Substitutionen und Macros

Wie zuvor erwähnt können mit der *.equ* Anweisung spezielle Zeichenketten ersetzt werden. Dies kann benutzt werden, um beispielsweise festzulegen, dass die Register *1* und *3* jeweils als *RIP (Instruction-Pointer)* und *RSP (Stack-Pointer)* verwendet werden sollen:

```
.equ   RIP, 1
.equ   RSP, 3
```

Diese Definition wird dann in einer Zeilen wie

```
subq    $8,     %RSP,     %RSP
```

Mit Hilfe von *Assembler-Macros* kann festgelegt werden, dass eine Zeile

```
pushq   %5
```

mit den Zeilen

```
subq    $8,     %RSP,     %RSP
movq    %5,   (%RSP)
```

expandiert werden soll. Die Zeilen, mit denen ein Macro ersetzt werden sollen, werden in einem durch die Anweisungen *.macro* und *.endm* begrenzten Block festgelegt. Durch Macro-Parameter kann außerdem festgelegt werden, dass in den einzufügenden Zeilen weitere Substitutionen durchgeführt werden sollen. Das Macro *pushq* wurde etwa definiert durch:

```
.macro pushq  PARAM
    subq    $8,     %RSP,     %RSP
    movq    \PARAM, (%RSP)
.endm
```

Hier wurde mit *PARAM* der Parameter für das Macro bezeichnet. Stellen im Macro-Block, die mit dem Wert von *PARAM* substituiert werden sollen, werden durch ein vorangestelltes "\"-Zeichen markiert. Der ULM-Assembler führt alle so definierten Substitutionen rekursiv aus.

Soll ein Macro mehr als einen Parameter erhalten, können diese wie in

```
.macro cmpq  PARAM1, PARAM2
    subq    \PARAM1, \PARAM2,  %0
.endm
```

durch eine Parameterliste festgelegt werden.

1.4.2 Erzeugen von Daten

Mit den Anweisungen *.byte*, *.word*, *.long* und *.quad* können Konstanten in Bitmuster entsprechender Breite umgewandelt werden. Beispielsweise wird mit der Anweisung *.byte 166* ein Bitmuster erzeugt, das der Hexadezimalzahl $(A, 6)_{16}$ entspricht.

1.4.3 Text- und Datensegment

Vom ULM-Assembler erzeugter Maschinencode setzt sich aus einem *Textsegment* und einem *Datensegment* zusammen. Das Textsegment enthält den Maschinencode, der aus Assemblerbefehlen generiert wurde. Das darauf folgende Datensegment enthält vorinitialisierte Daten. Dabei wird das notwendige Alignment berücksich-

tigt, so dass beispielsweise mit *.quad* erzeugte Daten an einer durch acht teilbaren Adresse liegen.

Im Assembler-Programm muss diese Trennung von Befehlen und Daten allerdings nur indirekt berücksichtigt werden. Mit den Anweisungen *.text* und *.data* kann festgelegt werden, ob darauf folgende Zeilen Assemblerbefehle oder Direktiven für das initialisieren von Daten enthalten. Dabei ist es möglich, diese beiden Anweisungen abwechselnd zu benutzen:

```
.data
    .quad 42
.text
    movq    (%6),   %6
    subq    %6,     %8,     %12
.data
    .quad 3
.text
    addq    %6,     %12,    %7
```

In einer Vorstufe zur Codegenerierung können so Zeilen für das Text- und Datensgement getrennt werden. Die Reihenfolge der Befehle bleibt dabei erhalten. Das Ergebnis dieser Verarbeitung sieht in diesem Fall wie folgt aus:

```
movq    (%6),   %6
subq    %6,     %8,     %12
addq    %6,     %12,    %7
.quad 42
.quad 3
```

1.4.4 Markieren von Befehlen und Daten

Durch ein *Label* kann ein darauf folgender Assemblerbefehl oder eine Assembler-Anweisung für Daten identifiziert werden. In Listing 1.2 werden in den Zeilen 8, 11 und 17 jeweils die Label *loop*, *check* und *n* definiert. Mit *loop* und *check* werden die jeweils darauf folgenden Assemblerbefehle *mulq* (Zeile 9) und *subq* (Zeile 12) markiert. Mit dem Label *n* wird das in Zeile 18 erzeugte Quad-Word gekennzeichnet. Die Unterscheidung zwischen einem Label zur Identifizierung von Befehlen (Text-Label) und Daten (Data-Label) ist wie folgt begründet:

- Ein Text-Label wird für Sprünge verwendet. Der ULM-Assembler muss dabei berücksichtigen, das bei Sprungbefehlen das angegebene Argument intern mit vier zu multiplizieren. In Zeile 7 wird deshalb das Label *check* nicht mit der

Listing 1.2: Assemblerprogramm zur Berechnung der Fakultät

```
1  .equ        RIP,        1
2
3  .text
4              orq         $n,         %0,         %5
5              movq        (%5),       %5
6              orq         $2,         %0,         %6
7              jmp         check
8  loop:
9              mulq        %5,         %6,         %6
10             subq        $1,         %5,         %5
11 check:
12             subq        $1,         %5,         %0
13             jg          loop
14             movq        %6,         n(%RIP)
15             halt
16 .data
17 n:
18             .quad 3
```

relativen Adresse 12 sondern 3 ersetzt. Analog wird in Zeile 13 das Label nicht mit -16 sondern -4 ersetzt.

- Ein Data-Label wird zum Laden oder Speichern von Daten verwendet. In diesem werden absolute und relative Adressen direkt verwendet. In Zeile 4 wird *$n* mit der absoluten Adresse 40 und in Zeile 14 mit der relativen Adresse 4 ersetzt.

Der in Listing 1.2 gezeigte Assembler-Code wird somit umgewandelt in

```
orq     $40,    %0,     %5
movq    (%5),   %5
orq     $1,     %0,     %6
jmp     $3
mulq    %5,     %6,     %6
subq    $1,     %5,     %5
subq    $1,     %5,     %0
jg      $-3
movq    %6,     8(%1)
halt
```

Anhand der Tabellen für den ULM-Befehlssatz kann der Assembler damit den in Listing 1.3 gezeigten Maschinencode generieren.

Algorithmus 1 Factorial

```
    f ← 1
do:
    if n < 2 then
        done
    end if
    f ← f · n
    n ← n − 1

    do
done:
    done
```

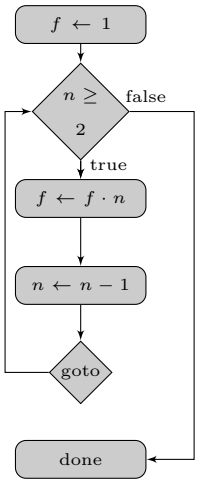

Abb. 1.7: Pseudo-Code und Ablaufdiagramm zur Berechnung der Fakultät.

1.4.5 Übungsaufgaben

1. Schreiben Sie folgende Macros für den ULM-Assembler:

1.5 Kontrollstrukturen

Durch *Kontrollstrukturen* wie *bedingten Anweisungen* und *Schleifen* können Algorithmen formuliert werden, die sich leicht als Computerprogramme umsetzen lassen. Als einfaches Beispiel dient dazu die Berechnung der Fakultät durch

$$n! = \prod_{k=1}^{n} k = \begin{cases} 2 \cdot \ldots \cdot n, & n \geq 2, \\ 1, & n = 0 \text{ oder } n = 1. \end{cases} \tag{1.10}$$

Ein Algorithmus für (1.10) muss so formuliert werden, dass es offensichtlich ist, wie die Berechnung mit den zur Verfügung stehenden Befehlen durchzuführen ist.

1.5.1 Goto und If-Goto

Verwendet man bereits bei der Formulierung eines Algorithmus Anweisungen für Sprünge, so kann dieser sehr leicht in Assembler-Code umgesetzt werden. Algorithmus 1 in Abbildung 1.7 beschreibt, wie einer Variable f zunächst der Wert 1 zugewiesen wird und dann im Fall $n \geq 2$ sukzessive mit

$$f \leftarrow f \cdot n \quad f \leftarrow f \cdot (n-1) \quad \ldots \quad f \leftarrow f \cdot 2$$

Listing 1.3: Assembler-Code zu Algorithmus

```
1  .equ    N,      %5
2  .equ    F,      %6
3
4  .text
5          orq     $3,     %0,     %N
6
7          orq     $1,     %0,     %F
8      do:
9          subq    $2,     %N,     %0
10         jl      done
11         mulq    %N,     %F,     %F
12         subq    $1,     %N,     %N
13         jmp     do
14     done:
15         halt
```

überschrieben wird, so dass schließlich $f = n!$ gilt. Durch das in Abbildung 1.7 ebenfalls dargestellte Ablaufdiagramm wird veranschaulicht, welche Operationen beim diesem Algorithmus sequentiell ausgeführt werden.

Eine mögliche Umsetzung in Assembler-Code zeigt Listing 1.3. In den Zeilen 1 und 2 wird vereinbart, dass mit N und F jeweils die Register 5 und 6 bezeichnet werden. Die eigentliche Umsetzung des Algorithmus beginnt, nachdem in Zeile 5 das Register N mit $n = 3$ belegt wurde:

- Die Rechenoperation $f \leftarrow 1$, $f \leftarrow f \cdot n$ und $n \leftarrow n - 1$ werden durch die Befehle *orq*, *mulq* und *subq* in den Zeilen 7, 11 und 12 umgesetzt.
- Um im Fall von $n < 2$ einen Sprung zur Marke *done* in Zeile 14 durchzuführen, wird in Zeile 9 der *subq* Befehl benutzt. Vom Wert in Register N wird eine 2 abgezogen, das Ergebnis aber verworfen, indem das Zero-Register als Ergebnis-Register verwendet wird. Nur im Fall von $n < 2$ wird von der ALU das Sign-Flag gesetzt, so dass der Befehl *jl* in Zeile 10 einen Sprung ausführt.
- Der unbedingte Sprung (goto) wird in Zeile 13 durch ein *jmp* umgesetzt.

1.5.2 While-Loop

Durch eine *While-Loop* kann der obige Algorithmus verständlicher durch

$f \leftarrow 1$
while $(n \geq 2)$ **do**
  $f \leftarrow f \cdot n$
  $n \leftarrow n - 1$
end while

Algorithmus 2 While-Loop

Instructions A
while cond **do**
    Instructions B
end while
Instructions C

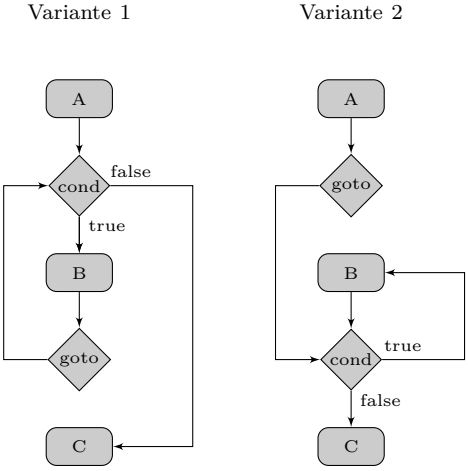

Abb. 1.8: Pseudo-Code und mögliche Ablaufdiagramme für eine While-Loop.

done

In Abbildung 1.8 werden durch Ablaufdiagramme zwei Möglichkeiten (Variante 1 und 2) skizziert, wie eine While-Loop mit *Durchlaufbedingung (cond)* durch Sprunganweisungen umgesetzt werden kann. Setzt man für obiges Beispiel, die Komponenten

A	$f \leftarrow 1$
cond	$n \geq$
B	$f \leftarrow f \cdot n$ $n \leftarrow n - 1$
C	done

ein, kann die Implementierung wieder dem in Abschnitt 1.5.1 gezeigten Schema erfolgen. Aus den Ablaufdiagrammen wird aber auch sichtbar, dass Algorithmus 1.7 gerade der Variante 1 der Ablaufdiagramme entspricht. In Listing 1.2 wurde bereits eine Implementierung gezeigt, die auf der Variante 2 basiert. Diese hat den Vorteil, dass pro Schleifendurchlauf ein Sprungbefehl weniger benötigt wird.

1.5.3 For-Loop

Mit einer *For-Loop* kann der Algorithmus so formuliert werden, dass er direkt die Verwendung des Produktzeichenes in (1.10) wiedergibt:

$f \leftarrow 1$
for $i = 2, \ldots, n$ **do**
    $f \leftarrow f \cdot i$

Algorithmus 3 For-Loop

Instructions A

for (init; cond; cont) **do**

  Instructions B

end for

Instructions C

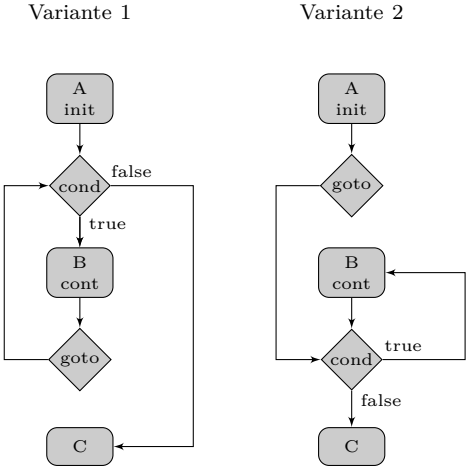

Abb. 1.9: Pseudo-Code und mögliche Ablaufdiagramme für eine For-Loop.

end for
done

Diese For-Loop ist dabei ein Spezialfall einer While-Loop mit

$f \leftarrow 1$
$i \leftarrow 2$
while $i \leq n$ **do**
  $f \leftarrow f \cdot i$
  $i \leftarrow i + 1$
end while
done

In Abbildung **??** zeigen wir eine allgemeinere Form der For-Loop, die zeigt, wie eine While-Loop um eine *Initialisierung (init)* und Anweisungen zur *Fortsetzung (cont)* ergänzt wird. Mit

Initialisierung (init)	$i \leftarrow 2$
Durchlaufbedingung (cond)	$i \leq n$
Fortsetzung (cont)	$i \leftarrow i + 1$
Loop-Body (B)	$f \leftarrow f \cdot i$

erhält man obige For-Loop. Durch Anpassen der Anweisungen zur Initialisierung und Fortsetzung kann ein Schleifenzähler wie in

$f \leftarrow 1$
for $i = n, \ldots, 2$ **do**
  $f \leftarrow f \cdot i$
end for

Algorithmus 4 If-Then-Else

```
Instructions A
if cond then
    Instructions B
else
    Instructions C
end if
Instructions D
```

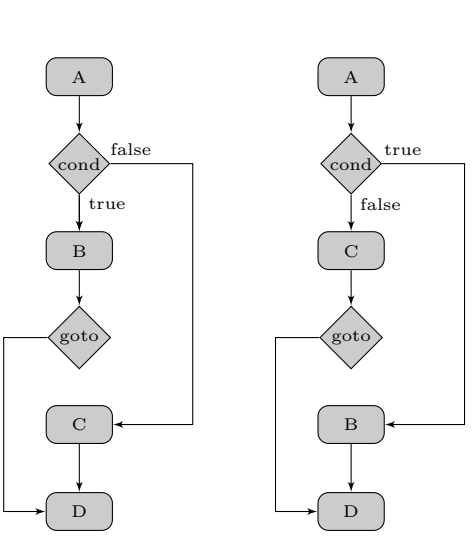

Abb. 1.10: Pseudo-Code und mögliche Ablaufdiagramme für ein If-Then-Else.

done

die Werte von 2 bis n in umgekehrter Reihenfolge durchlaufen.

1.5.4 If-Then-Else

Sprunganweisungen wurden in den bisher vorgestellten Kontrollstrukturen für die Realisierung von Schleifen verwendet. Das in Abbildung 1.10 dargestellte *If-Then-Else* Konstrukt beschreibt dagegen eine *Programmverzweigung* in Abhängigkeit einer Bedingung. Die Verzweigung beginnt, nachdem die Anweisungen A ausgeführt wurden. Ist die Bedingung (cond) erfüllt, werden die Anweisungen in B, und andernfalls die Anweisungen in C. In beiden Fällen wird nach dieser Verzweigung mit den Anweisungen in D fortgefahren.

1.5.5 Übungsaufgaben

1. *For-Loop*
   a) Assembler-Code für Fakultät mit For-Loop
   b) For-Loop rückwärts durchlaufen
2. *If-Then-Else*

Mit a, b und c seien im Folgenden jeweils die Inhalte der Register 5, 6 und 7 bezeichnet.

a) Gegeben sie der Pseudo-Code

if cond **then**
    $a \leftarrow 1$
else
    $a \leftarrow 0$
end if

Für die zusammengesetzten Bedingungen

$$(i) \quad \text{cond} = (a > b) \wedge (b > c),$$
$$(ii) \quad \text{cond} = (a > b) \vee (b > c)$$

soll der Pseudo-Code jeweils in Assembler-Code umgesetzt werden. Die Registerinhalte sollen dabei als Unsigned-Integer interpretiert werden.

b) In Aufgabe 2a) sollen nun die Registerinhalte als Signed-Integer interpretiert werden.

c) Beschreiben Sie zunächst den Pseudo-Code

if $(a > b) \wedge (b > c)$ **then**
    $n \leftarrow 1$
else if $(a > b) \vee (b > c)$ **then**
    $n \leftarrow 2$
else
    $n \leftarrow 0$
end if

mit einem Ablaufdiagramm. Schreiben Sie dann Assembler-Fragmente für die Fälle, dass Registerinhalte als Unsigned- oder Signed-Integer interpretiert werden.

3. *Größter gemeinsamer Teiler*

while $a \neq b$ **do**
    if $a > b$ **then**
        $a \leftarrow a - b$
    else
        $b \leftarrow b - a$
    end if
end while

4. *Summe von Vektorelementen* Mit For-Loop

5. *Repeat-Until-Loop*

Literaturverzeichnis

J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, Mar. 1988. ISSN 0098-3500. doi: 10.1145/42288.42291. URL http://doi.acm.org/10.1145/42288.42291.

J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Mar. 1990. ISSN 0098-3500. doi: 10.1145/77626.79170. URL http://doi.acm.org/10.1145/77626.79170.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3): 308–323, Sept. 1979. ISSN 0098-3500. doi: 10.1145/355841.355847. URL http://doi.acm.org/10.1145/355841.355847.

J. v. Neumann. First draft of a report on the edvac. Technical report, 1945.

A. S. Tanenbaum. *Structured Computer Organization.* Pearson Education, 6th edition, 2012. ISBN 0273769243, 9780273769248.