Schriftliche Hausarbeit zur Abschlussprüfung der erweiterten Studien für Lehrer im Fach Informatik

Weiterbildungskurs VIII Informatik des
Hessischen Kultusministeriums
in Zusammenarbeit mit der Fernuniversität Hagen

Entwicklung eines Registermaschinen-Simulationsprogramms zum Einsatz im Informatikunterricht der Jahrgangsstufe 13

vorgelegt von
Dirk von Sierakowsky
aus Bad Hersfeld
im August 2004

Inhaltsverzeichnis

1 Einleitung

Als ehemaliger Student der Mathematik kann ich mich noch sehr gut an meine allererste Vorlesung, Analysis I, erinnern: Von der ersten Minute an wurden komplizierte Beweise geführt, fremd anmutende Symbolfolgen notiert, und es gab Hausaufgaben, von denen ich die Aufgabenstellung erst nach drei Tagen intensiven Grübelns aufgefasst hatte. Die Frage, wozu ich als Lehramtsstudent den Stoff auf solch abstraktem Niveau lernen sollte, begleitete mich damals durch das komplette erste Semester.

An diese Zustände fühlte ich mich sofort wieder erinnert, als ich einige Jahre später, es war im Wintersemester 2001/2002, eine Vorlesung über Theoretische Informatik an der Fernuniversität Hagen belegt hatte und mich mit den ersten Kapiteln dieses Kurses beschäftigte[1]. Wenngleich ich mich aufgrund der genannten Vorerfahrungen dieses Mal weitaus weniger abschrecken ließ, kämpfte ich mich doch wieder mit viel Mühe durch ein theorie- und mathematiklastiges Skript. Und auch die Frage, was diese Vorlesung auf dem Studienplan für Lehramtsstudenten verloren hat, kam in mir erneut auf.

Meine Verwunderung sollte einige Zeit später weiter zunehmen, stellte ich doch fest, dass dieses auf mich so schülerfremd wirkende Thema im Lehrplan des Hessischen Kultusministeriums für die Jahrgangsstufe 13.1 vorgeschrieben war. „Wer hatte sich das ausgedacht?“ „Welchem Schüler würde nach diesem Thema wohl nicht die Lust auf Informatik vergehen?“ Solche Fragen schossen mir damals durch den Kopf und sollten noch einige Zeit unbeantwortet bleiben. Die Theoretische Informatik hat ohne Zweifel für Fachwissenschaftler ihre Berechtigung - mir war jedoch völlig unklar, wie ich als Lehrer das Thema didaktisch reduzieren sollte. Insofern war ich froh, noch keinen Kurs in der Jahrgangsstufe 13.1 unterrichten zu müssen.

Erst im Rahmen einer Präsenzveranstaltung zum Weiterbildungskurs VIII Informatik, zu dem ich im Sommer 2002 zugelassen wurde, kam ich dann wieder mit der Thematik in Verbindung: Simulationsprogramme für endliche Automaten, Kellerautomaten und Turingmaschinen ließen mich zum ersten Mal Möglichkeiten erkennen, einzelne Themen methodisch ansprechend für den Unterricht aufzuarbeiten. Ich war erstaunt, wie viele Möglichkeiten zu praxisorientiertem Arbeiten sich doch unter dem Mantel der Theoretischen Informatik verbargen.

Leider konnte man uns in dem genannten Lehrgang kein vergleichbar gutes Simulationsprogramm für Registermaschinen nennen. Die Software, die wir damals verwendeten, wies einige Mängel auf, die ich später noch näher erläutern werde. Gerade aufgrund meiner Vorliebe für das Registermaschinen-Konzept war ich über diese Tatsache sehr enttäuscht, und die Idee zur Erstellung des Registermaschinen-Simulators (ReSi) war schnell geboren.

Eine überschaubare und intuitiv leicht zu bedienende Benutzeroberfläche, ein komfortables Hilfesystem, eine gute Visualisierung der simulierten Maschine und die Möglichkeit, erstellte Registermaschinenprogramme speichern und laden zu können - dies waren nur einige der Anforderungen, die ich im Vorfeld an das Programm hatte. Während des Programmierens überlegte ich mir außerdem eine Reihe von Beispielprogrammen, die ich für sinnvoll erachtete und daher dem Projekt hinzufügte. Aus einzelnen Ideen,

[1] Ich studierte bereits vor der Teilnahme am Weiterbildungskurs VIII in Hagen auf eigene Initiative.

wie ReSi inklusiv dieser Beispielprogramme sinnvoll in den Unterricht integriert werden könnte, entwickelte ich anschließend zwei kleine Unterrichtsreihen.

Nach Abschluss all dieser Arbeiten kann ich nun behaupten, zumindest für das kleine Kapitel „Registermaschine“ einige Möglichkeiten zu kennen, wie es sich für Schüler anschaulich und hoffentlich gewinnbringend unterrichten lässt. Ich hoffe sehr, dass Kollegen mit ähnlichen Problemen, wie ich sie einmal mit der konkreten Umsetzung dieses Themas hatte, von meiner Arbeit profitieren können. Gerüstet mit dieser Arbeit sowie einigen anderen Konzepten aus dem genannten Wochenlehrgang freue ich mich nun jedenfalls auf meinen ersten Informatikkurs in der Jahrgangsstufe 13.1.

2 Registermaschinen in der Theoretischen Informatik

2.1 Aufbau einer Registermaschine

Eine Registermaschine ist ein sehr einfaches Modell eines Rechners, das aus den folgenden Komponenten besteht: Akkumulator, Register, Befehlszähler und Programm[2]. Es wurde 1963 von den amerikanischen Mathematikern J. C. Shepherdson und H. E. Sturgis entwickelt. Weil es sich bei der Registermaschine um ein theoretisches Rechnerkonzept handelt, geht man von den idealisierten Annahmen aus, dass es abzählbar unendlich viele Registerzellen gibt, von denen jede wiederum eine (beliebig große) natürliche Zahl speichern kann.

Als Akkumulator bezeichnet man die Registerzelle 0. Sie ist dadurch ausgezeichnet, dass alle arithmetischen Berechnungen und Vergleiche in ihr ausgeführt werden können.

Das Programm legt die Arbeitsweise der Registermaschine fest. Es besteht aus einer Folge von Befehlen (s. 3.1.2), die mit Zeilennummern versehen untereinander stehen. Bei [L1] wird der Befehlszähler beim Übergang zur nächsten Programmzeile immer genau um den Wert Eins erhöht, d.h. alle Zeilen müssen fortlaufend nummeriert sein. Im Unterschied dazu bin in dieser Arbeit aus praktischen Gründen dazu übergegangen, den Befehlszähler stets mit der Zeilennummer der momentanen Programmzeile zu füllen. Damit ist die Nummer gemeint, die der Programmierer am Beginn einer jeden Programmzeile vergeben muss. So wird es möglich, die Programmzeilen mit beliebigen Abständen zu nummerieren und auch nachträglich noch problemlos Zeilen einfügen zu können. Dementsprechend hat der Befehlszähler beim Start den Wert der Zeilennummer der ersten Programmzeile.

Die meisten Registermaschinen[3] beherrschen alle der Grundrechenarten Addition („add“), Subtraktion („sub“), Multiplikation („mult“) und Division („div“) sowohl mit Konstanten als auch direkt oder indirekt adressiert. Aufgrund der Tatsache, dass Registerinhalte immer aus $IN_0 := IN \cup \{0\}$ sind, ergeben sich aber folgende, gegenüber der Alltagsmathematik einschränkende Definitionen für die Subtraktion

$$a - b := \max\{a - b, 0\}$$

und für die Division

$$a \; div \; b := \lfloor a / b \rfloor$$

($\lfloor n \rfloor$ ist die größte ganze Zahl N mit $N \leq n$).

Um den Akkumulator zu laden und zu speichern, existieren die Befehle „load“ und „store“. Auch sie können auf die verschiedenen beschriebenen Arten adressiert werden[4]. Sprünge im Quelltext lassen sich bedingt („jzero“) oder unbedingt („goto“)

[2] Das beschriebene Konzept wurde an das von [L1] angelehnt.
[3] In der Literatur kursieren verschiedene Realisierungen von Registermaschinen, die sich hauptsächlich im Befehlssatz unterscheiden. Die in dieser Arbeit verwendete Registermaschine gehört zu denjenigen mit mächtigem Befehlssatz – vgl. 3.1.2.
[4] Einschränkung: „store“ darf nicht mit einer Konstante verwendet werden.

realisieren. Der theoretischen Vollständigkeit halber existiert noch der Befehl „end“, der den Befehlszähler auf unendlich setzt und das Programm terminieren lässt.

Für den praktischen Einsatz ist der Befehlssatz einer höheren Programmiersprache wie Java oder Delphi sicherlich wesentlich komfortabler als der Registermaschinenbefehlssatz. Die Bedeutung von Registermaschinen ist in der Theoretischen Informatik zu finden, was nun näher begründet werden soll.

2.2 Berechenbarkeit

Um die Bedeutung von Registermaschinen für die Theoretische Informatik erläutern zu können, ist der Begriff „Berechenbarkeit partieller Funktionen“ von großer Bedeutung. Zunächst soll in Anlehnung an [L1] definiert werden, was darunter zu verstehen ist:

Definition 1:

(i) Unter einer partiellen Funktion $f : IN^k \to IN$ versteht man eine Teilmenge $R \subseteq IN^{k+1}$, sodass zu jedem k-Tupel $(n_1, n_2, \ldots, n_k) \in IN^k$ höchstens ein $n \in IN$ mit $(n_1, n_2, \ldots, n_k, n) \in R$ existiert.

(ii) Alle Elemente $m \in IN^k$, denen kein Funktionswert zugeordnet wird, heißen Undefiniertheitsstellen von f. Man schreibt in diesem Fall $f(m) = \perp$.

(iii) Die partielle Funktion $f : IN^k \to IN$ heißt berechenbar, wenn es einen Algorithmus gibt, der für jedes k-Tupel $(n_1, n_2, \ldots, n_k) \in IN^k$, das keine Undefiniertheitsstelle ist, terminiert. Für Undefiniertheitsstellen wird unabhängig von einer möglichen Terminierung keine Ausgabe bereitgestellt.

Die Definition der Berechenbarkeit ist in dieser Form nicht hinreichend exakt, weil der Begriff „Algorithmus“ nicht definiert wurde. Dennoch lassen sich bereits partielle Funktionen angeben, die man mit konkreten Programmen der Programmiersprache „Turbo-Pascal“ berechnen kann:

(a) $\exp : IN^2 \ni (n,m) \to n^m \in IN$

```
function exp(n,m : integer) : integer;
var i,j : integer;
begin
  j:=1;
  for i:=1 to m do j:=j*n;
  exp:=j
end;
```

(b) $ggT : IN^2 \ni (a,b) \to ggT(a,b) \in IN$

```
function ggT(a,b : integer) : integer;
var i : integer;
begin
  if a < b then i := a else i:= b;
  while (a mod i <> 0) or (b mod i <> 0)
    do dec(i);
  ggT := i
end;
```

(c) $fak : IN \ni n \to n! \in IN$

```
function fak(n : integer) : integer;
begin
  if n=1 then
    fak:=1
  else
    fak:=n*fak(n-1)
end;
```

Wenn die noch etwas unklare Definition (iii) sinnvoll sein soll, dann müssen diese Funktionen zur Klasse der berechenbaren Funktionen gehören.

Der Vollständigkeit halber sei an dieser Stelle auch auf die folgenden Klassen von partiellen Funktionen hingewiesen, die jeweils anhand eines Beispiels erläutert werden:

(i) **Berechenbare partielle Funktionen, zu denen man keinen Algorithmus kennt, der sie berechnen kann**

Die Funktion $f(n) := \begin{cases} 0 & falls\ es\ Yetis\ gibt \\ 1 & sonst \end{cases}$ ist entweder konstant gleich 1 oder konstant gleich 0 und somit in jedem Fall berechenbar. Welchen Wert sie hat, weiß allerdings niemand mit Sicherheit.

(ii) **Partielle Funktionen, von denen man nicht weiß, ob sie berechenbar sind oder nicht**

Die Ulam-Folge einer natürlichen Zahl *n* ist folgendermaßen rekursiv definiert:

$a_0 := n$ und $a_{n+1} = \begin{cases} 1 & falls\ a_n = 1 \\ \frac{a_n}{2} & falls\ a_n\ gerade\ ist \\ 3a_n + 1 & sonst \end{cases}$ (*für n>0*).

Es ist nicht schwierig, ein Programm zu schreiben, das für eine Zahl *n* die Glieder ihrer Ulam-Folge berechnet. Es ist aber nicht bekannt, ob es natürliche Zahlen gibt, deren Ulam-Folgen niemals den Wert 1 annehmen. Hält nämlich ein solches Programm für eine Eingabe *n* nicht an, so kann nicht entschieden werden, ob man nicht lange genug gewartet hat oder ob die Folge tatsächlich niemals den Wert 1 annimmt. Es gibt leider keinen bekannten Algorithmus, der eine Aussage über diese Terminierung machen kann.

Bezogen auf die Funktion $f(n) := \begin{cases} 1 & falls\ der\ Wert\ 1\ angenommen\ wird \\ 0 & sonst \end{cases}$

lässt sich also leicht ein Algorithmus angeben, der prinzipiell *f(n)=1* berechnen kann (ein Pascal-Programm findet sich in [L2], ein Registermaschinenprogramm liegt dieser Arbeit bei). Allerdings ist nicht klar, ob der Funktionswert 0 jemals angenommen wird, und insbesondere kennt man keinen Algorithmus, der *f(n)=0* berechnen könnte. Somit ist nicht klar, ob die Funktion *f* berechenbar ist oder nicht.

(iii) **Partielle Funktionen, die nicht berechenbar sind**

Das wohl bekannteste Beispiel ist das sog. Halteproblem: Gibt es ein Programm *A*,

welches für ein beliebiges anderes Programm *B* mit wiederum einer beliebigen Eingabe *x* entscheiden kann, ob *B* nach endlich vielen Schritten terminiert oder nicht? Man hat bewiesen, dass es ein solches universelles Programm *A* nicht geben kann. Der Beweis und die Formulierung des Halteproblems als partielle Funktion finden sich z.B. in [L5].

2.3 Leistungsfähigkeit von Registermaschinen

Im letzten Abschnitt wurde bereits das Problem der bislang intuitiven Algorithmus-Definition genannt. Um den Begriff „Berechenbarkeit“ genauer definieren zu können, muss man sich auf die Programmiersprache und die Hardware festlegen. Eine Möglichkeit ist es, der Definition als Hardware Registermaschinen mit dem üblichen Befehlssatz (vgl. 3.1.1) zugrunde zu legen:

Definition 2: *Eine partielle Funktion* $f : IN^k \to IN$ *heißt Registermaschinen-berechenbar, wenn es ein Registermaschinenprogramm gibt, das diese Funktion berechnet.*

Mit Hilfe des Simulationsprogramms ReSi (siehe. Kapitel 3) kann man sich leicht davon überzeugen, dass die drei partiellen Funktionen *exp*, *ggT* und *fak* aus 2.2 zur Klasse der Registermaschinen-berechenbaren Funktionen gehören. Entsprechende Registermaschinenprogramme finden sich in 5.2.

Natürlich kann man nun analoge Definitionen des Berechenbarkeitsbegriffs für andere Rechner und Programmiersprachen angeben. Eine große Bedeutung für die Theoretische Informatik haben beispielsweise die Turingmaschinen[5] und darauf aufbauend die Klasse der Turingmaschinen-berechenbaren Funktionen. Aber auch moderne Computer mit höheren Programmiersprachen wie Java definieren völlig analog eigene Klassen berechenbarer partieller Funktionen.

Es stellen sich nun viele Fragen: In welcher Beziehung stehen alle genannten Klassen berechenbarer Funktionen zueinander? Gibt es z.B. partielle Funktionen, die sich zwar mit Turing-, nicht jedoch mit Registermaschinen berechnen lassen? Wie ist eine Funktion einzuordnen, für deren Berechnung man kein Registermaschinen-Programm findet? Hat man nicht lange genug danach gesucht, sind Programmiersprache oder Rechner nicht genügend ausgereift, oder gibt es zurzeit gar keine Hard- und Software, die sie berechnen kann? Falls dies der Fall ist: Können partielle Funktionen, die heute nicht berechenbar sind, vielleicht in ein paar Jahren mit einem höher entwickelten Rechner doch als berechenbar gelten? Eine Antwort auf diese Fragen gibt die Churchsche These[6], die in der Formulierung von [L6] lautet:

Churchsche These: *„Jede (vernünftige) Präzisierung des Begriffs Algorithmus führt auf die gleiche Menge von berechenbaren Funktionen“*

Die Churchsche These ist ein Erfahrungssatz und hat damit den Charakter eines Naturgesetzes. Bislang wurde also immer festgestellt, dass man unabhängig vom zugrunde liegenden Algorithmuskonzept stets auf dieselbe Menge der berechenbaren Funktionen stößt. Anstatt von „Algorithmus“ könnte man also in Definition 1 (iii) völlig äquivalent z.B.

[5] Nach Alan Turing, 1912-1954. Turingmaschinen sollen hier jedoch nicht näher betrachtet werden, weil sie für diese Arbeit keine zentrale Rolle spielen. Eine Untersuchung findet sich z.B. in [L1].
[6] Nach Alonzo Church, 1903-1995

von „Registermaschinenprogramm“ oder „Turingmaschinenprogramm“ reden, was nachträglich diese auf den ersten Blick etwas unsaubere Definition legitimiert.

Bezogen auf die Leistungsfähigkeit von Registermaschinen sagt die Churchsche These also aus, dass alle partiellen Funktionen, die mit noch so weit entwickelten Computern und Programmiersprachen berechnet werden können, sich auch mit Registermaschinen berechnen lassen (im Sinne des Programmieraufwands sind moderne Programmiersprachen natürlich „leistungsfähiger“ als die der Registermaschinen). Diese Äquivalenz wird plausibel, wenn man sich verdeutlicht, dass letztlich auch jeder moderne Prozessor nur auf einen Satz elementarer Befehle, ähnlich den Registermaschinenbefehlen, zurückgreifen kann. Auch zukünftige Rechner werden also, die Gültigkeit der Churchschen These immer vorausgesetzt, genau dieselbe Menge an partiellen Funktionen berechnen können wie Registermaschinen.

Jetzt wird klar, weshalb die Bedeutung von Registermaschinen für die Theoretische Informatik so groß ist: Wenn man die Grenzen der Leistungsfähigkeit heutiger und zukünftiger Computer untersuchen will, muss man keine Rechner mit komplizierter Architektur oder hoch entwickelter Programmiersprache untersuchen. Man kann sich je nach Geschmack auf die einfach aufgebauten Register- oder Turingmaschinen beschränken.

Gibt es vielleicht noch einfachere Maschinen, die dieselbe Klasse berechenbarer Funktionen liefern? Eine Möglichkeit der Vereinfachung wäre es, die unendlich große Speicherkapazität der Registermaschinen aufzugeben, was zur Theorie der endlichen Automaten führt. Man kann jedoch zeigen, dass es beim Übergang zu endlichen Automaten zu einem echten Verlust an Leistungsfähigkeit kommt. Als Beispiel sei die partielle Funktion $f : IN \to IN$ genannt, die genau dann den Funktionswert 1 besitzt, wenn ihr Argument *n* die Dezimaldarstellung $n = \underbrace{1\ldots1}_{k\ mal}\underbrace{0\ldots0}_{k\ mal}$ $(k \geq 1)$ besitzt. Sie ist nicht mit endlichen Automaten, wohl aber mit Registermaschinen berechenbar[7]. Registermaschinen (mit einem minimalen Satz an Befehlen) lassen sich also nicht weiter vereinfachen, ohne die Klasse der von ihnen berechenbaren Funktionen zu verkleinern.

2.4 Zusammenhang zu konkreten Problemstellungen

Was hat nun die Berechnung von partiellen Funktionen mit der Lösung von Problemen zu tun? Welche „Problemstellungen“ können Registermaschinen verglichen mit modernen Rechnern prinzipiell lösen?

Um diese Fragen zu beantworten, muss man zunächst einmal das umgangssprachliche Wort „Problem“ etwas genauer definieren. Kein Rechner wird ein „Problem“ wie das der steigenden Weltbevölkerung lösen können. In der Informatik sind daher stets nur solche Problemstellungen gemeint, die sich auch mathematisch formulieren lassen, denn nur sie können überhaupt von Computern bearbeitet werden.

Betrachten wir als konkretes Beispiel für eine solche Problemstellung das Sortieren von *n* (natürlichen) Zahlen, denn dieses hat auf den ersten Blick nichts mit der Berechnung einer partiellen Funktion $f : IN^k \to IN$ zu tun. Der Computer erhält eine Folge von *n* unsortierten Zahlen $(a_1, a_2, \ldots, a_n)$ als Eingabe und soll daraus eine sortierte Folge

[7] Ein Beweis soll an dieser Stelle nicht geführt werden, er findet sich in [L1], S.251ff.

berechnen und ausgeben. Diese Ausgabe könnte man jedoch als eine einzige natürliche Zahl auffassen, indem man z.B. alle einzelnen Zahlen hintereinander schreibt: $a = a_{18}a_3\ldots$[8]. Das Lösen des Problems „Sortieren von *n* Zahlen“ wäre also dasselbe wie die Berechnung der Funktion $f : IN \ni (a_1, a_2, \ldots, a_n) \to a \in IN$.

Die Idee dieses Beispiels, mehrere Zahlen zu einer einzigen zusammenzufassen, lässt sich noch ausweiten: Was eine Ausgabe auch immer ist (Texte, Zahlen, Bilder etc.), auf der untersten Ebene ist sie immer eine Folge aus 0 und 1 und somit die Binärdarstellung einer einzigen natürlichen Zahl. In dieser Sichtweise kann ein Computer prinzipiell nur natürliche Zahlen ausgeben. Dasselbe gilt natürlich auch für alle möglichen Eingaben, die ein Computer erhalten kann. Jedes Problem kann daher aus Berechnung einer partiellen Funktion $f : IN^k \to IN$ aufgefasst werden. Mit den Ergebnissen aus 2.3 ergibt sich, dass diejenige Klasse von Problemen, die mit jedem modernen Computer prinzipiell gelöst werden können, mit der Klasse von Problemen, die mit einer Registermaschine prinzipiell zu lösen sind, identisch ist.

[8] Technische Einzelheiten sollen hier nicht diskutiert werden, es geht lediglich um prinzipielle Ideen.

3 Das Registermaschinen-Simulationsprogramm (ReSi)

3.1 Leistungsumfang

3.1.1 Anforderungen und Zielsetzungen

Wie bereits im Vorwort dargelegt, erhielt ich den Anstoß, ein Simulationsprogramm für Registermaschinen zu schreiben auf einem der Wochenseminare im Rahmen des Weiterbildungskurses VIII Informatik. Das Thema „Didaktik der Theoretischen Informatik“, das damals auf der Tagesordnung stand, faszinierte mich schon seit Einführung des neuen Lehrplans. Ich konnte mir damals nicht vorstellen, wie eine Thematik, die auf den ersten Blick doch notwendigerweise ohne konkreten Computereinsatz auskommen müsste, bei Schülern[9] auf Gegenliebe stoßen sollte. Schließlich ist es doch die praktische Arbeit am Rechner, die die Schüler - nicht nur im Informatikunterricht – stark motiviert.

Dass sich auch bei diesem Thema der Computer sehr sinnvoll im Unterricht einsetzen lässt, durfte ich in dem genannten Lehrgang schnell feststellen. Neben der Möglichkeit, die Programmiersprache „Prolog“ zu verwenden, fiel mir besonders das Simulationsprogramm [S1] für endliche Automaten auf. Diese Software, die von unseren Ausbildern G. Röhner und Dr. J. Poloczek stammt, gestattet es zwar, Turingmaschinen und endliche Automaten schülergerecht zu simulieren - Registermaschinen gehören jedoch leider nicht zum Funktionsumfang. Weil ich selbst aus verschiedenen Gründen heraus (vgl. 4.2) das Konzept „Registermaschine“ gegenüber dem Konzept „Turingmaschine“ im Unterricht vorziehen, hätte ich eine solche Möglichkeit als sehr hilfreich empfunden.

In dem genannten Lehrgang wurde ein Java-Applet[10] als Simulationsprogramm für Registermaschinen eingesetzt. Mir wurde jedoch schnell klar, dass dies keine handhabbare Lösung für die Schule darstellt, denn der größte Nachteil war, dass sich Registermaschinen-Programme nicht speichern ließen, ohne dabei den Java-Quelltext zu verändern. Ein weiterer Nachteil, der die Erstellung von Programmen erschwert, war die unkomfortable Eingabe der Registermaschinenprogramme. Hier fiel mir besonders negativ auf, dass das nachträgliche Einfügen von Programmzeilen oder von Kommentaren in einen bereits bestehenden Quelltext nicht möglich war. Die fehlenden Möglichkeiten, Programme oder Registerprotokolle direkt auszudrucken, komplettierten meinen Eindruck, dass dieses Applet für den Schuleinsatz nicht besonders geeignet ist.

Auch nach längerer Suche im Internet konnte ich jedoch kein besseres Simulationsprogramm finden, und so war schnell die Idee zu einem eigenen Simulationsprogramm geboren. Der Anforderungskatalog ergab sich dann fast zwangsläufig aus den Funktionen sowie den bereits genannten Nachteilen des Java-Applets:

- Die Erstellung von Programmen sollte mit Hilfe eines Editors erfolgen, sodass das nachträgliche Einfügen von Programmzeilen problemlos möglich ist. Die

[9] Wenn ich hier und im Folgenden vereinfacht von „Schülern“ spreche, sind dabei selbstverständlich stets alle Schülerinnen mit einbezogen.
[10] Siehe [S2].

Möglichkeit für Kommentare sollte gegeben sein, weil Registermaschinenprogramme von Natur aus schlecht lesbar sind.

- Die erstellten Programme sollten ohne Umwege gespeichert, geladen und gedruckt werden können.

- Der Ablauf von Registermaschinenprogrammen sollte gut visualisiert werden können und alle Registerinhalte sollten jederzeit einsehbar sein.

- Die Simulation der Maschine sollte sehr langsam möglich sein, um so Schülern die Möglichkeit zu geben, den Ablauf eines Programms in Ruhe nachvollziehen zu können. Ein Einzelschrittmodus sollte ebenfalls eingebaut sein. Für lange Berechnungen erschien es wiederum erforderlich, Programme sehr schnell ablaufen lassen zu können.

- Ein Protokoll über die wechselnden Registerbelegungen sollte bei Bedarf automatisch angefertigt werden, weil sich daraus sinnvolle didaktische Möglichkeiten ergeben (vgl. 4.3).

- Typische Programmierfehler der Schüler sollten erkannt werden und nicht zum Absturz des Simulators führen.

- Ein Schrittzähler sollte es ermöglich, verschiedene Programme zu demselben Problem untereinander bzgl. ihrer Effizienz vergleichen zu können.

- Zu guter Letzt wollte ich eine möglichst übersichtliche und intuitive Bedienung des Programms erreichen. Eine kleine Hilfe sollte die Erlernung der Bedienung unterstützen.

Nachdem ich eine grobe Vorstellung davon gewonnen hatte, was das Programm leisten sollte, fiel auch die Wahl der Programmiersprache nicht mehr schwer. Obwohl seit Jahren an unserer Schule Java als objektorientierte Programmiersprache gelehrt wird, erschien es mir aufgrund der benötigten graphischen Benutzeroberfläche zu aufwändig, das Projekt in dieser Sprache zu realisieren. Weil zwei der vorangegangenen Wochenseminare des Weiterbildungskurses VIII bereits vom Thema „Delphi“ handelten, hatte ich zu diesem Zeitpunkt schon einen kleinen Einblick in diese Sprache, und sie schien gut für meine Anforderungen geeignet zu sein. Bestärkt wurde diese Entscheidung außerdem durch die Tatsache, dass Delphi in vielen Schulen als objektorientierte Programmiersprache eingesetzt wird und es somit möglich wäre, das Programm im Unterricht mit Schülern analysieren und weiterentwickeln zu können.

3.1.2 Befehlssatz

In der Literatur kursieren verschiedene Realisierungen von Registermaschinen, die sich im Befehlssatz bzw. dessen Syntax zum Teil erheblich unterscheiden. Die Registermaschine von [L1] gehört zu den mächtigen Maschinen, unterstützt sie doch sowohl Befehle mit indirekter Adressierung als auch solche für Multiplikation und Division. Eine Registermaschine, die weniger Möglichkeiten der Programmierung bietet, findet sich z.B. in [L4]. Dennoch liefert sie dieselbe Klasse berechenbarer Funktionen, weil sich alle zusätzlichen Befehle der Maschinen mit mächtigeren Befehlssätzen durch einfa-

chere Befehle simulieren lassen. Als Beispiel sei auf die Multiplikation hingewiesen, die man mit Hilfe mehrerer Additionen ausdrücken kann.

Ich habe mich für die Simulation einer Maschine mit mächtigem Befehlssatz entschieden, weil die Maschine dadurch gut programmierbar wird. Die Syntax von [S2] erschien mir intuitiv leicht verständlich, weshalb ich sie für Schüler als gut geeignet empfinde und in ReSi verwende.

Die folgende Übersicht[11] stellt einen Zusammenhang zwischen der von mir gewählten Syntax und der von [L1] her und erläutert alle Befehle:

Befehl in ReSi	***Auswirkungen auf Register und Befehlszähler b***	***Befehl in [L1]***
load n	*R(0) := R(n), b:= N(b)*[12]	load n
cload n	*R(0) := n, b:= N(b)*	load #n
iload n	*R(0) := R(R(0)), b:= N(b)*	load *n
store n	*R(n) := R(0) , b:= N(b)*	store n
istore n	*R(R(n)) := R(0)*[13] *, b:= N(b)*	store *n
add n	*R(0) := R(0) + R(n) , b:= N(b)*	add n
cadd n	*R(0) := R(0) + n, b:= N(b)*	add #n
iadd n	*R(0) := R(0) + R(R(0)), b:= N(b)*	add *n
sub n	*R(0) :=* max *{R(0) - R(n), 0}, b:= N(b)*	sub n
csub n	*R(0) :=* max *{R(0) – n, 0}, b:= N(b)*	sub #n
isub n	*R(0) :=* max *{R(0) - R(R(0)), 0}, b:= N(b)*	sub *n
mult n	*R(0) := R(0) * R(n) , b:= N(b)*	mult n
cmult n	*R(0) := R(0) * n, b:= N(b)*	mult #n
imult n	*R(0) := R(0) * R(R(n)) , b:= N(b)*	mult *n
div n	*R(0) :=* $\lfloor R(0)/R(n) \rfloor$[14]*, b:= N(b)*	div n
cdiv n	*R(0) :=* $\lfloor R(0)/n \rfloor$*, b:= N(b)*	div #n
idiv n	*R(0) :=* $\lfloor R(0)/R(R(n)) \rfloor$*, b:= N(b)*	div *n
goto n	*b := n*	goto n
jzero n	*b :=* $\begin{cases} n & falls\ R(0)=0 \\ N(b) & sonst \end{cases}$	jzero n
end	*b :=* ∞	end

[11] vgl. [S2]
[12] *N(b)* bezeichnet hier die Zeilennummer der im Quelltext auf *b* folgenden Programmzeile, vgl. 2.1.
[13] Im Unterschied zu [L1] akzeptiert das Simulationsprogramm den Befehl „istore n“ mit R(n)=0.
[14] Die Division durch Null führt selbstverständlich zum Programmabbruch.

3.1.3 Syntax der Programmzeilen

Wie bereits erwähnt bestand eine der selbst formulierten Anforderungen darin, den Quellcode mit Kommentaren versehen zu können. Dies ist in Form ganzer Zeilen möglich, jeweils beginnend mit zwei Slashes (‚//'). Beim Programmablauf werden diese Zeilen dann automatisch übersprungen.

Mit Ausnahme der Kommentare muss jede Zeile des Quelltextes folgende Syntax besitzen:

ZEILENNUMMER REGISTERMASCHINENBEFEHL

Jede Zeilennummer muss eine natürliche Zahl sein, und jeder Befehl muss aus dem genannten Befehlssatz stammen. Überflüssige Leerzeichen werden vom Simulator ignoriert.

3.1.4 Register

Selbstverständlich lässt sich die theoretische Vorgabe, dass eine Registermaschine abzählbar unendlich viele Register besitzt, von denen jedes eine (beliebig große) natürliche Zahl speichern kann, nicht auf einen realen Rechner übertragen. Sowohl die Zahl der Register als auch deren Wertebereich mussten also in ReSi beschränkt werden, ohne dabei zu stark an Leistungsfähigkeit einzubüßen. Für die meisten Programme, die im Rahmen des Unterrichts angefertigt werden, würden vermutlich zehn Register völlig ausreichen. Gerade bei der Programmierung von Sortieralgorithmen, einem Standardthema des Informatikunterrichts, benötigt man jedoch ganz nach Anzahl der zu sortierenden Zahlen einen etwas größeren Satz an Registern, weshalb die Anzahl auf 50 Register plus Akkumulator beschränkt wurde.

Die obere Grenze für die Registerinhalte wurde vom Pascal-Datentyp „integer" übernommen und liegt daher bei 2147483647. Negative ganze Zahlen sind gemäß Theorie als Registerinhalte nicht vorgesehen und werden daher von ReSi nicht unterstützt.

3.1.5 Fehlererkennung

Folgende Fehler werden zur Laufzeit der Registermaschinenprogramme von ReSi erkannt und führen zum Abbruch des simulierten Registermaschinenprogramms, wobei die für den Abbruch verantwortliche Zeile markiert bleibt:

Fehler	***Ursache***
Division durch Null	*cdiv 0*, *div n* mit *R(n)=0* oder *idiv n* mit *R(R(n))=0*.
Ungültige Zeilennummer	Eine Programmzeile (Kommentare ausgenommen) beginnt nicht mit einer natürlichen Zahl - vgl. 3.1.3
Unbekanntes Sprungziel	*goto n* oder *jzero n* mit nicht existierender Zeilennummer *n*
Unbekannter Befehl	Der Befehlssatz bzw. die Syntax der Befehlszeilen (vgl. 3.1.3) wurden missachtet.
Ungültige Registernummer	Ein Befehl adressiert eine Registernummer, die nicht im Bereich von 0 bis 50 liegt.
Akkumulatorüberlauf	Das Ergebnis einer Berechnung wäre größer als die obere Grenze 2147483647 für Registerinhalte.

Diese Übersicht wurde knapp gehalten, weitere Informationen zu den Fehlermeldungen finden sich im gleichnamigen Kapitel des Hilfepakets (vgl. 5.3).

3.2 Bedienung

Wie bereits erwähnt, war eine intuitive Bedienung eines der selbst gestellten Kriterien für das Simulationsprogramm ReSi. Anhand eines Beispielprogramms zur Berechnung der Potenzfunktion $\exp : IN^2 \ni (n,m) \to n^m \in IN$ soll die Bedienung in den folgenden Abschnitten näher erläutert werden. Hierbei wird kein Wert auf Vollständigkeit gelegt, denn eine vollständige Erklärung aller Menüpunkte befindet sich in dem integrierten Hilfepaket (vgl. 5.3).

3.2.1 Systemvoraussetzungen und Installation

Die Anforderungen, die das Simulationsprogramm an die zugrunde liegende Hardware stellt, sind denkbar gering. Es wurde auf folgenden Betriebssystemen erfolgreich getestet: Microsoft Windows XP, Microsoft Windows 2000, Microsoft Windows 98 und Microsoft Windows 95. Das Programm benötigt inklusive Hilfedatei und Beispielprogramme weniger als zwei Megabyte Festplattenspeicher.

Zur Installation muss lediglich das File *ReSi.zip* von der mitgelieferten CD entpackt werden. Eine weitere Installation ist nicht notwendig.

Gestartet wird *ReSi.exe* nach dem Entpacken per Doppelklick, woraufhin sich das in Abbildung 1 gezeigte Startfenster öffnet. Dieses besteht aus den folgenden Elementen:

- Hauptmenü (oben)
- Symbolleiste (oben)
- Programmeditor (links)
- Registereditor (rechts)
- Taktratenschieberegler (unten links)
- Schrittzähler (unten rechts)

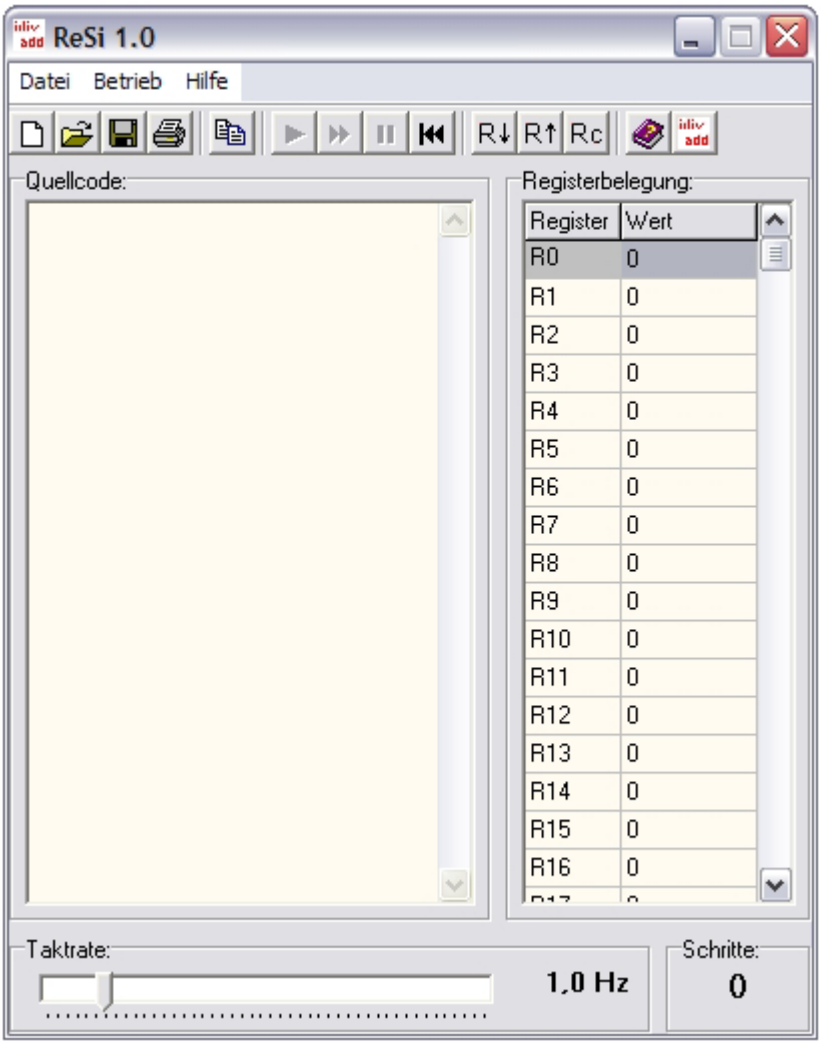


Abbildung 1: Startfenster

3.2.2 Erstellen von Registermaschinenprogrammen

Sofort nach dem Erscheinen des Startfensters kann mit dem Programmieren der Registermaschine begonnen werden. Zunächst erstellt man zweckmäßigerweise den Quelltext im Programmeditor, wobei lediglich auf den zur Verfügung stehenden Befehlssatz gemäß 3.1.2 sowie die Syntax der Zeilen gemäß 3.1.3 geachtet werden muss. Abbildung 2 zeigt das fertige Programm (vgl. [L1], Seite 24) samt Kommentarzeilen zu Beginn.

Spätestens nach dem Erstellen sollte das Programm gespeichert werden. Dies geschieht entweder mit dem Menüeintrag *Datei → Speichern* oder über das entsprechende Icon auf der Symbolleiste. Nach dem Speichern wird der Dateiname in der Kopfzeile des Programms angezeigt.

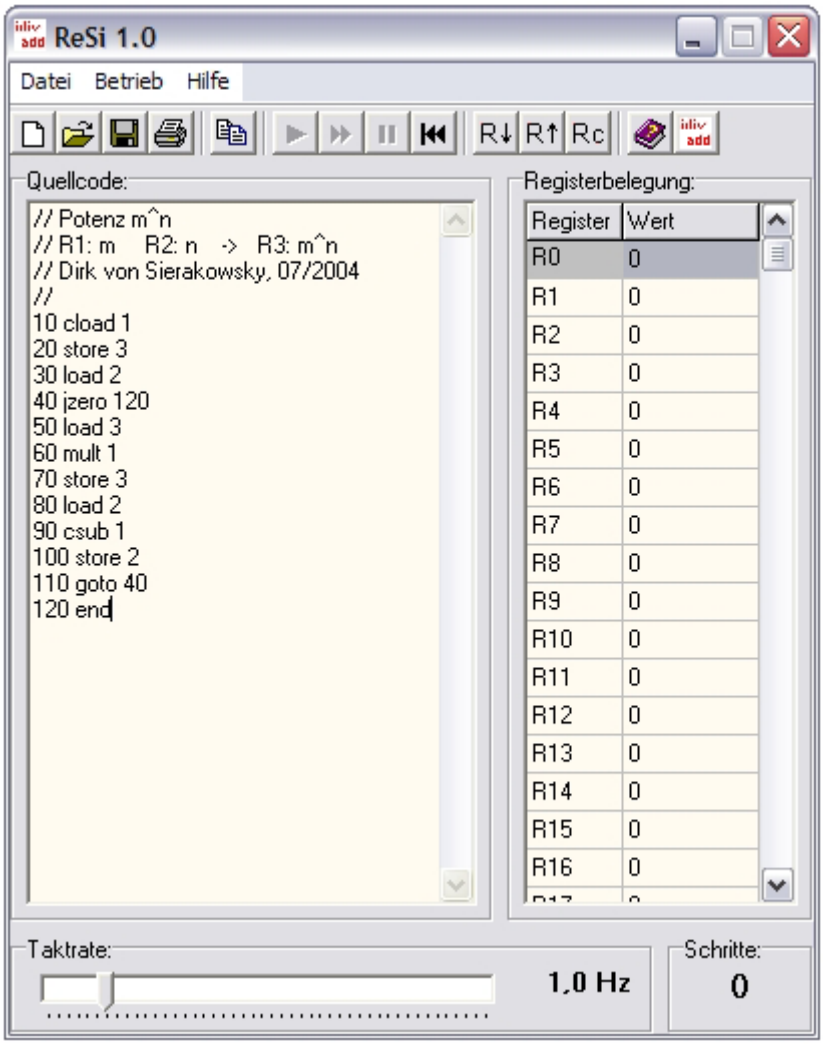


Abbildung 2: Programm zur Potenzberechnung

3.2.3 Ausführen von Programmen

Vor dem Ausführen des Programms muss noch die Registervorbelegung eingegeben werden. Dies geschieht rechts im Editor für die Registerbelegung durch Anklicken der entsprechenden Zellen (Abbildung 3). Für das Programm „Potenz“ muss die Basis *m* im Register R1 und der Exponent *n* im Register R2 stehen. Das Ergebnis findet sich dann später in R3.

ReSi verfügt über einen Zwischenspeicher für die Registerinhalte, der genau eine Registerbelegung speichern kann. Die eingegebene Registerbelegung kann in diesem mit Hilfe des Befehls *Betrieb → Register zwischenspeichern* gesichert werden. Bei einem erneuten Programmstart würde sie mit dem Befehl *Betrieb → Register wiederherstellen* wieder zur Verfügung stehen. Die aktuelle Registerbelegung und der Registerspeicher lassen sich mit dem Befehl *Betrieb → Register löschen* bequem wieder gemeinsam löschen.

Als letzte Vorbereitung müssen vor dem Start des Programms noch der Befehls- und der Schrittzähler der Registermaschine in den Anfangszustand versetzt werden. Dies geschieht durch Anwahl des Menüpunkts *Betrieb → Zurücksetzen*. Nun wird die erste

Programmzeile im Quelltext-Editor markiert, und die Menüpunkte *Betrieb → Einzelschritt* sowie *Betrieb → Start* sind aktiv (siehe Abbildung 4).

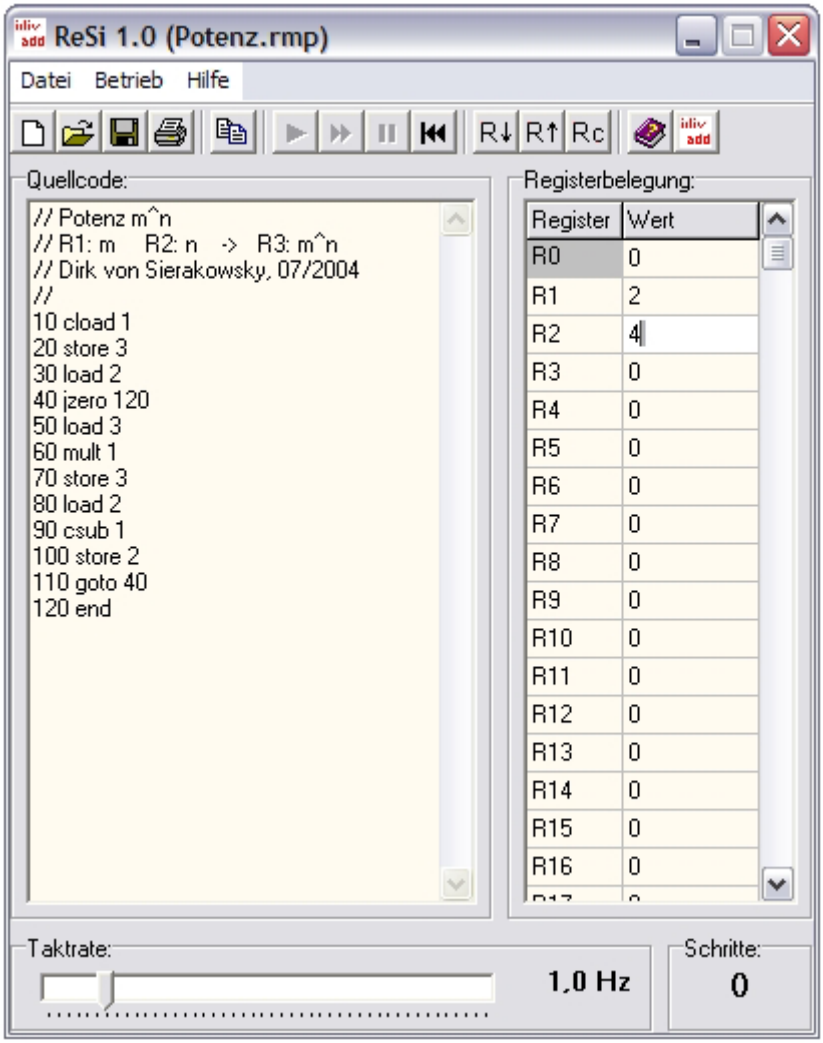


Abbildung 3: Eingabe der Registerbelegung zur Berechnung der Potenz 2^4

Prinzipiell stehen zwei Betriebsmodi zur Verfügung: Im Einzelschrittmodus (*Betrieb → Einzelschritt*) kann das Programm zeilenweise abgearbeitet werden, im Ablaufmodus (*Betrieb → Start*) wird es Befehl für Befehl automatisch abgearbeitet, bis es entweder erfolgreich terminiert ist oder durch eine Fehlermeldung bzw. manuell gestoppt wurde (*Betrieb → Abbruch*). Im Ablaufmodus kann die Frequenz der Registermaschine mit dem logarithmischen Schieberegler für die Taktrate zwischen *0.1* und *100 Hertz* variiert werden[15]. Zwischen beiden Modi kann während der Programmausführung beliebig gewechselt werden.

In beiden Modi werden stets die aktuelle Programmzeile sowie das zuletzt veränderte Register hervorgehoben. Dieses Markieren wurde aus didaktischer Sicht eingebaut, um es den Schülern zu ermöglichen, die Wirkungsweise der einzelnen Befehle besser nachvollziehen zu können. Unter *Betrieb → Optionen* lässt es sich je nach Geschmack abschalten.

[15] Es wurde kein Wert auf die exakte Einhaltung der angegebenen Frequenz gelegt. Je nach zugrunde liegendem System weicht die tatsächliche Frequenz davon erheblich ab.

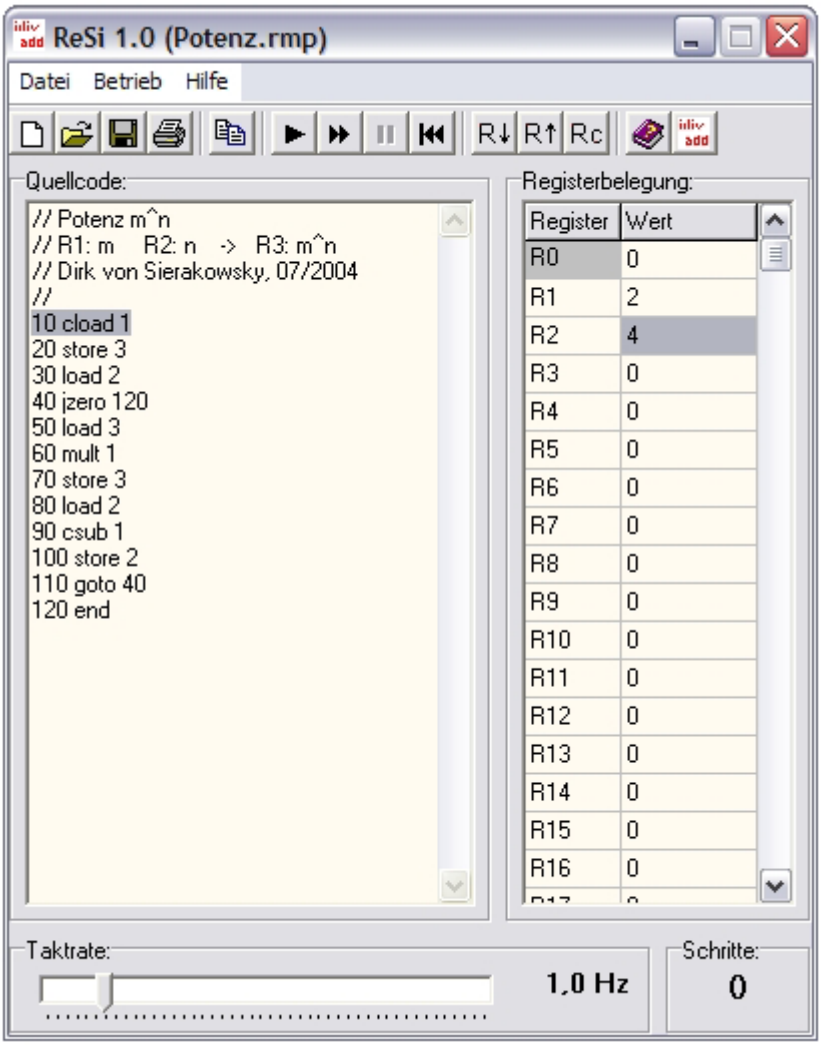


Abbildung 4: Nach dem Zurücksetzen des Befehls- und Schrittzählers

Hat das Programm erfolgreich terminiert (Abbildung 5), so gibt der Schrittzähler Auskunft über die Anzahl der benötigten Schritte. Das Ergebnis der Berechnung von 2^4 ist wie erwartet 16 und steht im Register R3. Für einen erneuten Programmstart müssten, wie oben erläutert, zunächst Befehls- und Schrittzähler zurückgesetzt werden. Ein Zurücksetzen ist ebenfalls nötig, wenn während des Programmlaufs manuelle Änderungen am Quelltext oder an den Registerinhalten vorgenommen werden.

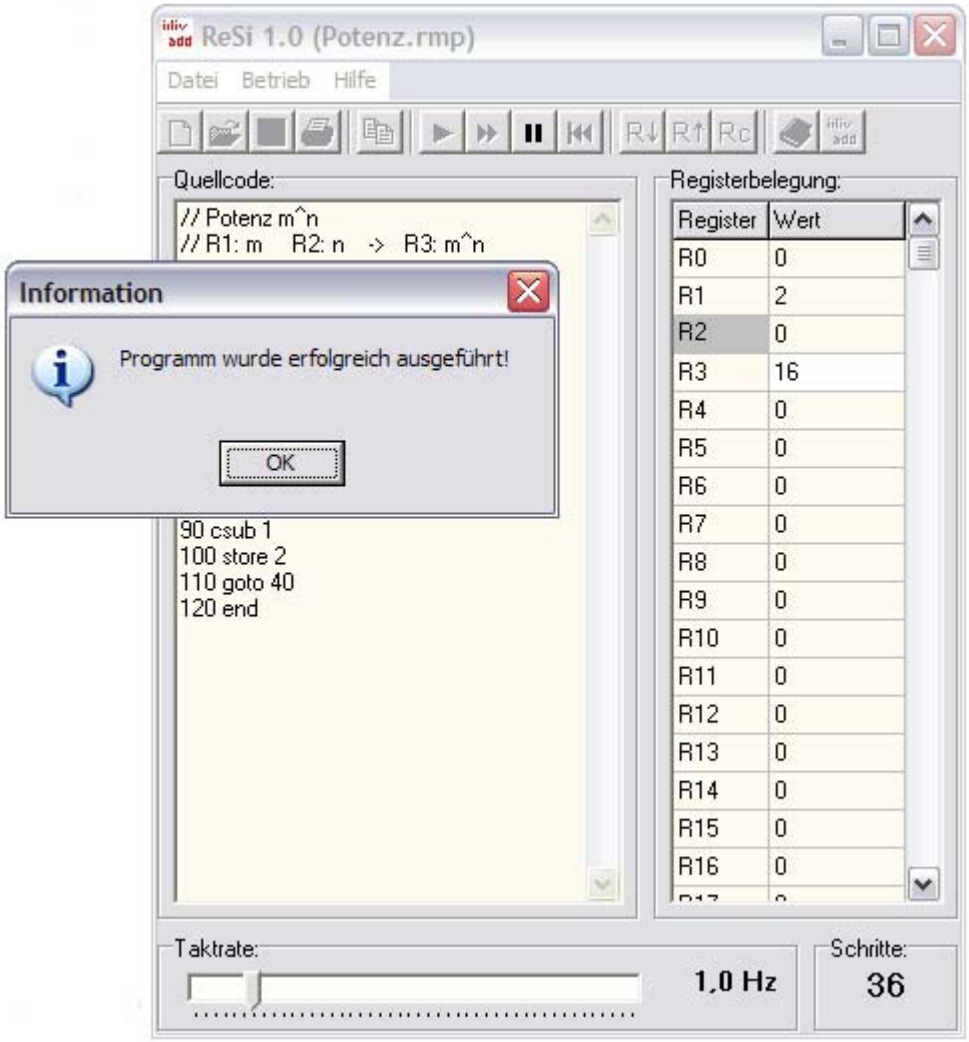


Abbildung 5: Erfolgsmeldung nach erfolgreicher Terminierung, das Resultat 16 steht im Register R3.

3.2.4 Protokoll des Registerverlaufs

Eine weitere Möglichkeit, um den Schülern den Ablauf des Programms verständlich zu machen, bietet die fortlaufende Protokollierung der Registerbelegung während der Simulation eines Registermaschinenprogramms. Ein solches Protokoll wird von ReSi automatisch in einem eigenen Protokollfenster erstellt. Es lässt sich unter *Betrieb → Protokoll anzeigen* sichtbar machen (Abbildung 6) und bleibt solange im Vordergrund, bis es wieder manuell geschlossen wird.

Der Übersichtlichkeit wegen, vor allem beim Drucken, werden standardmäßig nur die Register R0 bis R10 protokolliert. Unter *Betrieb → Optionen → Protokolleinstellungen* lässt sich die Anzahl der Register an das Problem anpassen. Auf demselben Formular kann die Protokollart verändert werden, um zusätzlich zu den Registerbelegungen die Programmzeilen mitzuprotokollieren.

Mit Hilfe des Schalters „Zwischenablage“ lässt sich das Protokoll dann über die Zwischenablage in Anwendungsprogramme wie Textverarbeitungen exportieren.

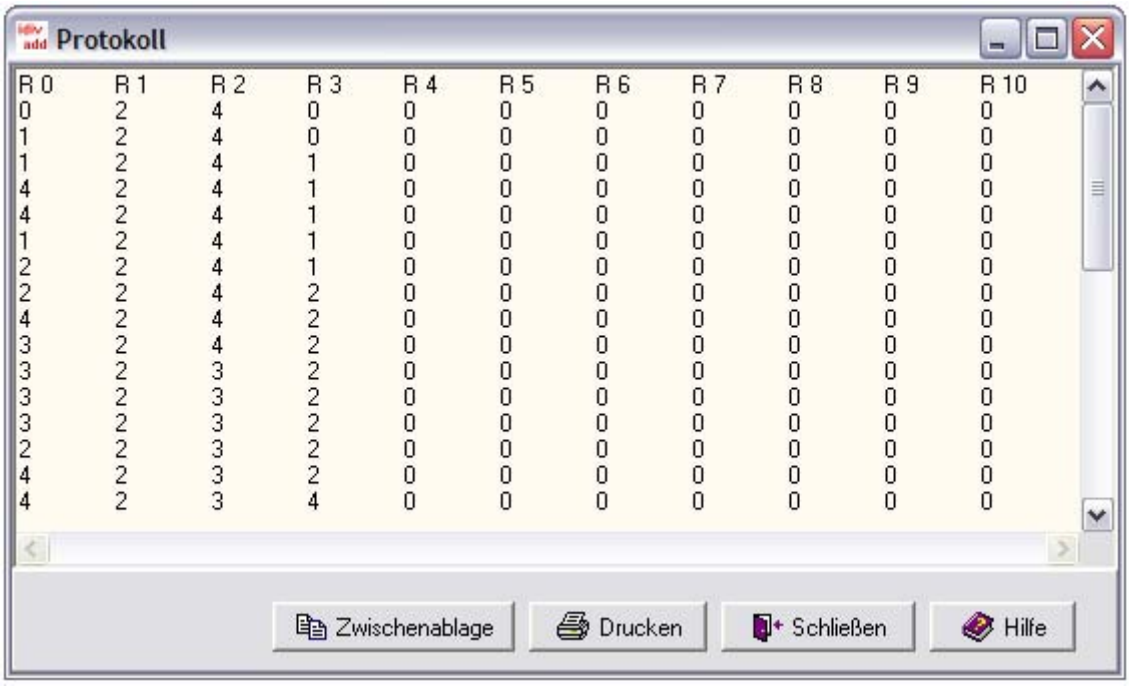


Abbildung 6: Registerprotokoll der Registerbelegungen von R0 bis R10

3.2.5 Gliederung des Hilfesystems

Um den Registermaschinen-Simulator in der Praxis leichter bedienbar zu machen, wurde ein ausführliches Hilfe-Paket integriert (*Hilfe → Anleitung*). Dieses wurde mit dem Microsoft Help Workshop erstellt und gliedert sich wie folgt:

- Der Registermaschinen-Simulator (ReSi) 1.0
  - ReSi 1.0: Allgemeine Hinweise
  - Info
  - Betrieb des Simulators
    - Beschränkung der Register
    - Syntax der Programmzeilen
    - Arithmetische Befehle
    - Sonstige Befehle
    - Beispielprogramm: Potenzberechnung
    - Protokollieren der Registerbelegung
  - Menüpunkte
    - Menüpunkt „Datei“
    - Menüpunkt „Betrieb“
  - Fehlermeldungen
    - Allgemeine Hinweise
    - Division durch Null
    - Unbekanntes Sprungziel
    - Unbekannter Befehl
    - Ungültige Registernummer
    - Ungültige Zeilennummer
    - Akkumulatorüberlauf

Alle Erklärungen im Hilfesystem wurden möglichst umgangssprachlich beschrieben, um die Hilfe für Schüler verständlich zu halten. Formale Exaktheit war daher nicht das

oberste Ziel - vgl. zum Beispiel die Erklärungen zum Befehlssatz. Das komplette Hilfesystem befindet sich im Anhang.

Die Hilfen zu den Fehlermeldungen wurden kontextsensitiv in das Simulationsprogramm integriert. Führt also während des Ablaufs eines Registermaschinenprogramms eine Fehlermeldung zum Abbruch, so steht durch Drücken des Buttons „Hilfe“ automatisch die entsprechende Hilfeseite zur Verfügung.

3.2.6 Beispielprogramme

Folgende Registermaschinenprogramme wurden bereits erstellt und dem Paket hinzugefügt. Sie befinden sich im Ordner „programme“:

Dateiname	***Beschreibung***	***Komplexität***
Bedingte Anweisung	Eine bedingte Anweisung wird im Registermaschinencode programmiert.	niedrig
Bruchaddition	Zwei Brüche werden addiert, das Ergebnis wird nicht gekürzt.	niedrig
Bubblesort 10	Zehn Zahlen werden per Bubblesort-Algorithmus sortiert.	sehr hoch
Fakultät	Die Fakultät einer natürlichen Zahl wird berechnet.	mittel
ggT nach Euklid	Der größte gemeinsame Teiler zweier natürlicher Zahlen wird nach dem Euklidschen Algorithmus berechnet.	hoch
kgV	Das kleinste gemeinsame Vielfache zweier natürlicher Zahlen wird berechnet.	mittel
Maximum 10	Berechnet das Maximum von 10 Zahlen.	hoch
Maximum 2	Berechnet das Maximum von 2 Zahlen.	mittel
Modulo	Berechnet *m mod n*.	niedrig
Multiplikation	Die Multiplikation zweier Zahlen wird auf mehrfache Additionen zurückgeführt.	mittel
Potenzberechnung	Berechnet m^n, siehe 3.2.2 ff.	mittel
Primzahltest	Erkennt, ob eine Zahl prim ist oder nicht.	hoch
Summe 10	Berechnet die Summe von 10 Zahlen.	hoch
Summe 2	Berechnet die Summe von 2 Zahlen.	niedrig
Ulam	Akzeptor für Ulam-Folgen, die den Wert 1 annehmen.	mittel

Im Kapitel 4.3 finden sich zu fast allen Programmen Ideen, wie diese sinnvoll in den Unterricht eingebaut werden können.

3.3 Das Delphi-Projekt

3.3.1 Überblick

Bevor detailliert auf die einzelnen Teile des Projekts eingegangen wird, soll zunächst ein grober Überblick erfolgen. In Abbildung 7 sind daher alle zum Projekt gehörigen Units in Beziehung zueinander gestellt. Eine besondere Stellung nehmen *RegisterEditor* und *RMProgramEditor* ein, weil sie eigene Delphi-Komponenten implementieren, die in die Unit *UReSi* eingebunden sind (siehe 3.3.5).

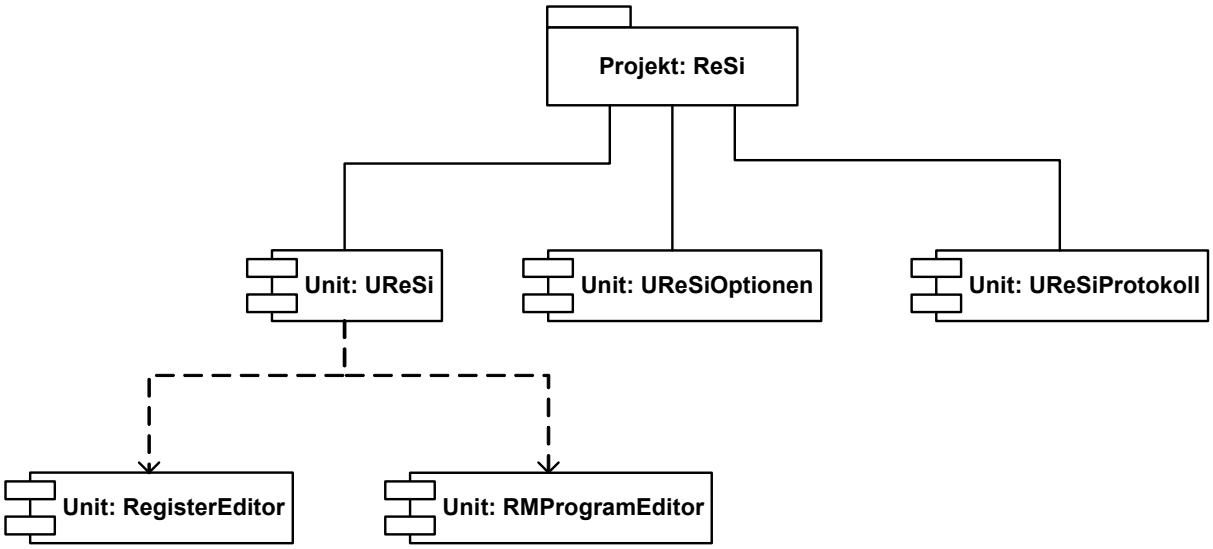


Abbildung 7: Übersicht über das Projekt ReSi

Es folgt eine funktionale Übersicht über die implementierten Klassen, die jeweils in eigenen Units realisiert wurden. Sie enthält eine Beschreibung der wesentlichen Umsetzung und geht beispielhaft auf die Verwendung von Klassen aus der Delphi-Bibliothek ein. Details zu den verwendeten Attributen und Methoden können den Quelltexten im Anhang entnommen werden.

3.3.2 Unit *UReSi*

Diese Unit implementiert die Klasse *THauptformular*, die die Benutzeroberfläche bereitstellt und von der Klasse *TForm* abgeleitet ist. Wie Abbildung 1 zu entnehmen, besteht die Benutzeroberfläche aus folgenden Elementen:

Element	***Enthaltene Typen***[16]
Hauptmenü	*TMainMenu, TMenuItem*
Symbolleiste	*TToolBar, TToolButton*
Icons für Symbolleiste und Hauptmenü	*TImageList*
Programmeditor	*TRMProgramEditor*
Registereditor	*TRegisterEditor*
Taktratenschieberegler	*TTrackBar, TLabel*
Schrittzähler	*TLabel*
Datei- und Druckdialoge	*TPrintDialog, TOpenDialog, TSaveDialog*

Viele der aufgeführten Typen sind mehrfach als Attribute in der Klasse enthalten, z.B. gibt es verschiedene Attribute vom Typ *TMenuItem*. Des Weiteren sind die meisten Attribute mit Voreinstellungen konfiguriert, die mit Hilfe des Delphi-Objektinspektors vorgenommen wurden. Beispielsweise wurde der Taktratenschieberegler so vorkonfiguriert, dass er beim Programmstart auf der Frequenz *1Hz* steht.

Die beiden Editoren für den Quelltext und die Registerbelegung wurden im Sinne einer Kapselung in eigenen Klassen realisiert (s. 3.3.5), was teilweise auch durch ihre

[16] Die Auflistung enthält nur die wichtigsten Typen.

umfangreiche Funktionalität gerechtfertigt erscheint. Die so entstandenen Klassen *TRMProgramEditor* und *TRegisterEditor* wurden als eigene Komponenten im Delphi-System registriert und dann in der Klasse *THauptformular* verwendet. Kleinere Teile wie z.B. der Schrittzähler wurden hingegen nicht als eigene Klassen realisiert, weil der Aufwand hierfür ungerechtfertigt hoch gewesen wäre.

Der Aufruf der implementierten Methoden erfolgt ereignisgesteuert durch die Benutzeraktionen. So wird beispielsweise die Methode *EinzelschrittClick()* beim Klicken auf den Menüpunkt „Einzelschritt“ aufgerufen.

Neben diesen Methoden, die allesamt die Endung „Click“ besitzen, existieren in dieser Klasse noch Hilfsmethoden. Ein Beispiel ist die Prozedur *updateFrequency()*, die die Beschriftung des Taktratenschiebereglers aktualisiert, sobald die Stellung des Schiebereglers verändert wird. Um dies zu erreichen, wurde die Hilfsmethode im Objektinspektor mit dem Ereignis *OnChange* des Attributs *TaktrateTrackBar* verbunden. Der Name „Hilfsmethode“ ist etwas irreführend, denn in diesen Programmteilen steckt die meiste gedankliche Arbeit. Die Methode *executeProgramRow()* implementiert beispielsweise die komplette Ausführung einer Zeile des Registermaschinen-Quelltextes. Sie ist die umfangreichste Methode des gesamten Projekts und enthält selbst mehrere Unterprozeduren.

3.3.3 Unit *UReSiOptionen*

Die Unit implementiert die Klasse *TOptionenFormular*, einen direkten Nachfahren der Delphi-Klasse *TForm*. Das Optionenformular besteht im Wesentlichen aus den beiden Karteikarten „Programmeinstellungen“ und „Protokolleinstellungen“:

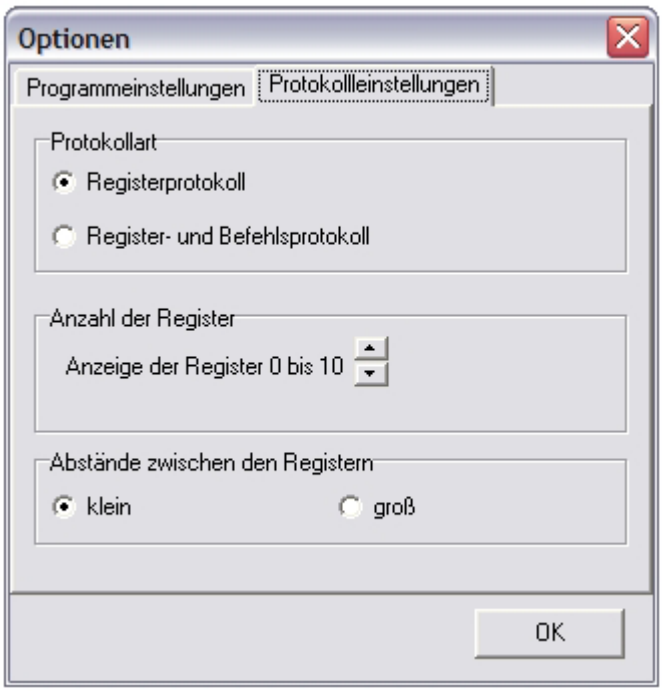


Abbildung 8: Das Optionenformular

Die Karteikarten sind Attribute vom Typ *TTabSheet*, die zu einem übergeordneten Attribut vom Typ *TPageControl* gehören und im Wesentlichen die folgenden Typen enthalten:

Karteikarte	*Enthaltene Typen*
Programmeinstellungen	*TCheckBox*
Protokolleinstellungen	*TRadioGroup, TLabel, TUpDown, TGroupBox*

Der Aufruf aller implementierten Methoden dieser Unit erfolgt Ereignis gesteuert durch die Benutzeraktionen - z.B. wird beim Verändern der Protokollart die Methode *ProtokollartRadioGroupClick()* aufgerufen.

Die Voreinstellungen wurden im Objektinspektor so gewählt, wie sie als sinnvoll erachtet werden – beispielsweise ist das Attribut *ProgrammzeileMarkierenCheckBox* standardmäßig auf *checked* gesetzt. Das dauerhafte Speichern der eingestellten Optionen über das Beenden von ReSi hinaus wurde nicht als notwendig erachtet und daher nicht realisiert.

3.3.4 Unit *UReSiProtokoll*

UReSiProtokoll implementiert die von *TForm* vererbte Klasse *TProtokollFormular*. Wie Abbildung 6 zeigt, besteht das Formular aus einem Protokollfenster und einem Panel des Typs *TPanel* mit verschiedenen Buttons:

Element	*Enthaltene Typen*
Protokollfenster	*TMemo*
Panel	*TBitButton*

Neben den Ereignis gesteuerten Methoden existieren die beiden Hilfsmethoden *actualizeProtocol()* und *clearProtocol()*, in denen die Funktionalität der Klasse im Wesentlichen implementiert wurde. Im Delphi-Optioneninspektor wurden wiederum Voreinstellungen vorgenommen, z.B. wurde bei allen Buttons die Eigenschaft *Anchors* auf *[akTop,akRight]* gesetzt, sodass diese beim Verändern der Fenstergröße ihre rechtsbündige Anordnung auf dem Panel behalten.

3.3.5 Units *RegisterEditor* und *RMProgramEditor*

Diese beiden Units implementieren die Typen *TRegisterEditor* und *TRMProgramEditor* und bilden zusammen das Delphi-Package *ReSiComponents*. Das eigene Package war notwendig, weil die genannten Typen als eigene Delphi-Komponenten realisiert werden mussten. Die Registrierung in der Registerkarte Beispiele erfolgt jeweils mittels der Prozedur *Register()*.

Der Typ *TRegisterEditor* wurde vom Typ *TValueListEditor* abgeleitet und an die Bedürfnisse des Simulationsprogramms angepasst. Die Beschriftung der Register von 0 bis 50 ist ebenso eingebaut wie ein Zwischenspeicher für eine komplette Registerbelegung. Des Weiteren wird über den Aufruf der Prozedur *init()* sichergestellt, dass der Benutzer nur natürliche Zahlen mit maximal zehn Stellen eingeben kann (vgl. 3.1.4).

Im Sinne einer guten Kapselung stehen noch weitere öffentliche Methoden zum Zugriff auf die Register zur Verfügung, z.B. können mittels *setRegisterValue()* Werte in ein Register geschrieben werden. Intern werden dabei die Zahlen in Strings umgewandelt und direkt in den jeweiligen Zellen gespeichert.

Hingewiesen sei auch auf die Ereignisbehandlungsroutine *DrawCell()*, die die geerbte Methode gleichen Namens überschreibt. Sie wird beim Eintreten des Ereignisses *OnDrawCell* aufgerufen und sorgt dafür, dass ein Register, dessen Inhalt verändert wurde, markiert gezeichnet wird.

Der Typ *TRMProgramEditor* ist direkter Nachfahre des Typs *TMemo*. Er enthält lediglich die beiden Methoden *getMaxProgramRow()* und *markProgramRow()*. Letztere sorgt dafür, dass eine Programmzeile des Registermaschinenprogramms markiert werden kann (vgl. 3.2.3).

3.3.6 Hilfesystem

Wie bereits in 3.1.1 beschrieben, war ein kleines Hilfesystem eine der Anforderungen, die ich im Vorfeld an das Projekt ReSi gestellt hatte. Als Tool zur Umsetzung erschien mir der Microsoft Help Workshop geeignet zu sein. Dieser liefert als Ergebnis die Hilfedatei *ReSi.hlp*, die sich mit nur wenigen Klicks über die Delphi-Projektoptionen in das Projekt integrieren lässt. Weitere Features wie die Unterstützung von Links innerhalb des Hilfesystems und die relativ umfangreichen Gestaltungsmöglichkeit der Hilfe-Texte sprechen ebenfalls für die Wahl dieser Software.

Der Hauptvorteil war für mich jedoch die Möglichkeit, ausgewählte Hilfeseiten kontextsensitiv zur Verfügung stellen zu können, denn dies wird sowohl von Delphi als auch vom MS Help Workshop unterstützt. Daher stehen nun beispielsweise bei auftretenden Fehlermeldungen gezielt die passenden Hilfeseiten zur Verfügung.

Das Hilfesystem besteht insgesamt aus den folgenden Dateien, von denen die letzten beiden nur zum Entwurf benötigt wurden:

Dateiname	***Beschreibung***
ReSi.hlp	Ausführbare Windows-Hilfedatei, wird vom Microsoft Help Workshop generiert.
ReSi.cnt	Enthält das Inhaltsverzeichnis des Hilfesystems, z.B. die vorhandenen Kapitel. Diese Datei muss mit Hilfe des Help Workshops manuell erstellt werden.
ReSi.hpj	Enthält Informationen wie die Pfadangabe der Datei *Hilfetexte ReSi.rtf*, die Zuordnung einzelner Hilfeseiten zu den Kontextnummern der kontextsensitiven Hilfe oder das Layout des Hilfefensters.
Hilfetexte ReSi.rtf	Enthält die eigentlichen Hilfeseiten, die mittels festgelegter Formatierungen an die in *ReSi.cnt* beschriebene Struktur angepasst wurden – siehe 5.3. Als Beispiel sei an dieser Stelle auf die notwendigen Fußnoten hingewiesen, die identisch zu den TopicID's sein müssen.

4 Didaktik der Registermaschinen

4.1 Legitimation

Seit einigen Jahren hat das Themengebiet „Theoretische Informatik“ eine deutliche Aufwertung im Lehrplan [L7] erhalten, indem es verbindlich für die Jahrgangsstufe 13.1 vorgeschrieben wurde. Die Legitimation zur Behandlung von Registermaschinen innerhalb dieses Themengebietes ergibt sich im Lehrplan hauptsächlich aus den Begriffen „Registermaschine“ und „registerberechenbar“ (S. 20). Für Leistungskurse werden Registermaschinen dabei als gleichberechtigte Alternative zu Turingmaschinen zur Wahl gestellt, während diese Themen für Grundkurse als fakultative Inhalte angegeben sind.

Ebenfalls fakultativ sollen Aspekte der Technischen Informatik bereits in der 13.1 unterrichtet werden[17]. Der Begriff „Assemblersprache“, der hier genannt wird, stellt aus meiner Sicht eine weitere Legitimation für den Einsatz von Registermaschinen dar, denn deren Programmiersprache erinnert stark an Assemblersprachen realer Mikroprozessoren.

„Die Behandlung theoretischer Fragestellungen darf nicht isoliert von den Anwendungen und auf Vorrat erfolgen“ (S. 18). Dieser allgemeine methodische Hinweis des Lehrplans fordert geradezu den Einsatz von Simulationsprogrammen in der Theoretischen Informatik. In dieselbe Richtung zielt auch die Aufforderung „Im Grundkurs sollen die Fachbegriffe und Zusammenhänge möglichst anschaulich eingeführt werden, vertiefte mathematische Formalismen und Methoden sind zu vermeiden. Soweit möglich sind die theoretischen Konzepte mit konkreten Anwendungen zu verbinden“ (S.18).

Neben den genannten Legitimationen aus dem Lehrplan halte ich das Thema für sinnvoll, weil der Einsatz von Registermaschinen eine Vorbereitung für die Jahrgangsstufe 13.2 darstellt, sofern man sich dort für das Wahlthema „Technische Informatik“ entscheidet. Aber auch ohne diese Entscheidung ist es ein wichtiges Thema, denn die Schüler lernen anhand von Registermaschinen - die als Modell realer Prozessoren verstanden werden können - Computer auf einer elementaren Ebene zu verstehen. Hieraus sollte sich auch Motivation für das Thema ergeben.

4.2 Didaktischer Vergleich: Turing- oder Registermaschine?

Wie bereits erwähnt, stellt der Lehrplan Turing- und Registermaschinen sowie ihre Berechenbarkeitsbegriffe als alternative Konzepte nebeneinander. Ich möchte deshalb hier beide Möglichkeiten aus didaktischer Sicht vergleichen und beginne mit den Vorteilen des Konzepts „Turingmaschine“.

Zweifellos sind Turingmaschinen einfacher aufgebaut als Registermaschinen: Die Hardware der einfachsten Turingmaschinen[18] besteht nur aus einem einzigen unbegrenzten Band als Speicher sowie einem in beide Richtungen verschiebbaren Lese-/Schreibkopf. Auch der Befehlssatz ist leichter aufzufassen als der von Registermaschinen, weshalb Turingmaschinen sicher als das überschaubarere Konzept einzustufen sind. Ein weiteres Argument für die Verwendung von Turingmaschinen kann die Behandlung von Kellerautomaten oder endlichen Automaten im vorausgehenden Unter-

[17] „Technische Informatik“ ist zudem als Wahlthema für die Jahrgangsstufe 13.2 vorgesehen.
[18] vgl. z.B. [L1], S.36ff.

richt darstellen. Diese Maschinen arbeiten nämlich zustandsorientiert und sind daher sehr ähnlich zu programmieren – im Gegensatz zur imperativen Programmierweise der Registermaschinen. Nicht zuletzt sind Turingmaschinen sicher auch aus historischer Sicht bedeutender als Registermaschinen. Die Tatsache, dass Alan Turing maßgeblich an der Dechiffrierung der Enigma im zweiten Weltkrieg beteiligt war, bietet die Möglichkeit für kleine historische Abstecher im Unterricht.

Demgegenüber stehen allerdings auch zahlreiche Vorteile des Konzepts „Registermaschine“. In erster Linie ist für den Unterricht sicher relevant, dass Registermaschinenprogramme schon bei einfachen Aufgaben viel handhabbarer und deutlich weniger kompliziert sind als Turingmaschinenprogramme, die das Gleiche leisten. Der Grund dafür liegt in dem mächtigeren Befehlssatz der Registermaschinen, der durch den etwas komplexeren Aufbau möglich wird. Der Nachteil der aufwändigeren Bauweise erweist sich noch in einem weiteren Punkt letztlich als Vorteil, denn die Architektur der Registermaschine liegt im Gegensatz zur Architektur der Turingmaschine sehr nahe am realen Mikroprozessor, weshalb Registermaschinen als Modell für CPUs dienen können. Die Schüler erhalten durch die Registermaschinen also nebenbei eine Anschauung von realen Prozessoren und der Assemblerprogrammierung. Hinzu kommt ein weiterer Pluspunkt für diejenigen Schüler, die bereits über Erfahrungen im Programmieren mit imperativen Sprachen verfügen, denn für sie sind Registermaschinenprogramme intuitiv leicht verständlich. Weil gemäß Lehrplan in den Jahrgangsstufen 11.2 und 12.1 solche Kenntnisse vermittelt werden, schätze ich den benötigten Zeitbedarf für das Erstellen von Registermaschinenprogrammen geringer ein als den für das Erstellen von Turingmaschinenprogrammen. Gerade für Grundkurse erscheint daher die Behandlung von Registermaschinen besser geeignet als die von Turingmaschinen.

Zusammenfassend bin ich der Ansicht, dass im Allgemeinen die Behandlung von Registermaschinen sinnvoller ist als die Behandlung von Turingmaschinen. Verfügen die Schüler allerdings bereits über Kenntnisse von endlichen Automaten und Kellerautomaten, so kann die Behandlung von Turingmaschinen eine sinnvolle Fortführung dieser Maschinen darstellen.

4.3 Vorschläge zur Verwendung von ReSi im Unterricht

Wie bereits im zweiten Kapitel erwähnt, spielen Registermaschinen eine wichtige Rolle in der Theoretischen Informatik. Im Schulunterricht lässt sich der Registermaschinen-Simulator daher vor allem im Halbjahr 13.1 („Konzepte und Anwendungen der Theoretischen Informatik“) sinnvoll verwenden. Allerdings sehe ich auch die Möglichkeit zu einem Einsatz im zweiten Halbjahr der Jahrgangsstufe 13 im Rahmen des Wahlthemas „Technische Informatik“. Laut Lehrplan soll nämlich der Kurs „mit der Analyse einiger beispielhafter Assemblerprogramme abgerundet werden“. An dieser Stelle könnte die Simulationsumgebung mit den mitgelieferten Beispielprogrammen als Modell für eine Assemblersprache zum Einsatz kommen. Die methodischen Möglichkeiten des Simulationsprogramms (vgl. 4.3.3) sprechen sicherlich dafür.

In den folgenden beiden Abschnitten sollen zwei Möglichkeiten der Integration in der 13.1 erörtert werden, die beide an das Konzept [L2] anknüpfen. Keiner dieser Zugänge konnte bislang erprobt werden, weil ich über keinerlei Unterrichtserfahrungen in der Jahrgangsstufe 13 in Informatik verfüge. Meine Ausführungen sind daher als Ideen aufzufassen, wie sich das Thema „Registermaschine“ möglichst anschaulich in den Unterrichtsgang integrieren lässt. Dabei wurde versucht, die Inhalte auf Doppelstunden auf-

zuteilen, wobei von einem zweistündigen Grundkurs ausgegangen wurde. Für dreistündige Grundkurse oder gar Leistungskurse sollte es kein Problem darstellen, das Thema entsprechend zu vertiefen oder die Inhalte passend aufzuteilen.

Weil das Simulationsprogramm frei vervielfältigt werden darf, kann es insbesondere jedem Schüler zur Verfügung gestellt werden. Damit bietet sich die Möglichkeit, Hausaufgaben unter Einbeziehung von ReSi zu stellen, sofern alle Schüler Zugang zu einem geeigneten Rechner besitzen. Weil die Systemvoraussetzungen sehr niedrig sind, erscheint dies nicht unwahrscheinlich.

4.3.1 Zugang 1: Technische Informatik

Wie bereits erwähnt wird die Technische Informatik mit den Begriffen „Rechnerarchitektur“ und „Assemblersprache“ als fakultatives Thema für die 13.1 vorgeschlagen. Ich könnte mir dieses Thema bereits als Einstieg in das gesamte Halbjahr vorstellen. Als Motivation dient die Frage: „Wie kann ein Computer, der doch eigentlich nur 0 und 1 kennt, komplizierte Programme einer höheren Programmiersprache ausführen?“. Dies den Schülern exemplarisch zu verdeutlichen ist zugleich die Intention des hier beschriebenen Zugangs.

Voraussetzungen:
Hilfreich wäre es, wenn die Schüler die Binärdarstellung von natürlichen Zahlen sowie die logischen Funktionen UND, ODER und NOT bereits kennen. Diese Themen lassen sich aber bei Bedarf auch in die erste Doppelstunde der Unterrichtsreihe integrieren.

Doppelstunde 1:

Lernziele: Die Schüler sollen…
- die logische Funktionsweise von UND-, ODER- und NOT-Schaltgattern angeben können
- die Binärdarstellung natürlicher Zahlen kennen
- den Aufbau des Halbaddierers angeben und seine Wahrheitstafel aufstellen können
- die Funktionsweise des Addierwerks für 2-Bit-Dualzahlen exemplarisch nachvollziehen können
- den Zusammenhang zwischen dem Addierwerk und dem Rechnen mit natürlichen Zahlen erläutern können
- den Aufbau des Vergleichers angeben und seine Wahrheitstafel aufstellen können

In den ersten beiden Doppelstunden wird der Frage nachgegangen, wie es möglich ist, maschinell zu rechnen. Als Einführungsbeispiel dient ein sog. Halbaddierer, d.h. ein Addierer für zwei 1-Bit-Binärzahlen A und B. Dieser sollte, wie in Abbildung 9 zu sehen, ausschließlich aus UND-, ODER- und NOT-Schaltgattern aufgebaut sein[19]. Da es nur darum geht anzudeuten, wie ein Computer rechnet, sollten keine anderen Schaltgatter, ICs oder Flip-Flops verwendet werden.

[19] Die Schaltsymbole wurde aus [L8] entnommen.

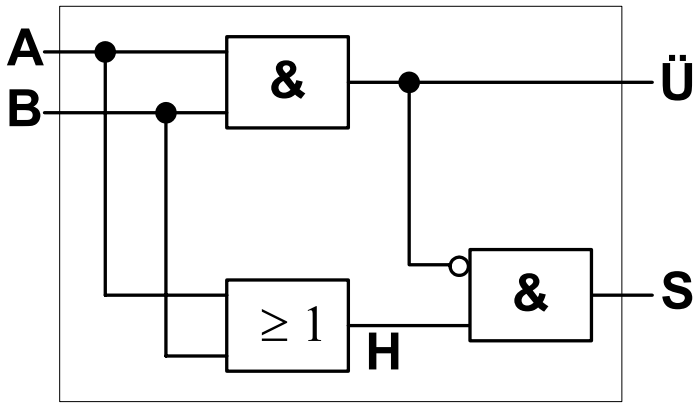


Abbildung 9: Aufbau eines Halbaddierers mit zwei Eingängen A und B und zwei Ausgängen S[20] und Ü

Der Addierer sollte nach Möglichkeit mit Hilfe von Logikbausteinen, die sich in den meisten Physiksammlungen befinden, real aufgebaut und vorgeführt werden - alternativ bieten sich natürlich Computersimulationen an.

Nach dem Vorführen werden ggf. zunächst die verwendeten Schaltgatter erklärt. Die Funktionsweise der gesamten Schaltung wird anschließend mit Hilfe der Wahrheitstafel

Eingaben			***Ergebnis***	
A	***B***	***H***	***Ü***	***S***
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

besprochen, ebenso wird der Zusammenhang zur Binärdarstellung von natürlichen Zahlen thematisiert.

Folgende Aufgabe schließt sich nun an: „Stelle die Wahrheitstafel der folgenden Schaltung auf und versuche herauszufinden, welchen Zweck sie haben könnte! Beginne zunächst mit der Wahrheitstafel des oberen Halbaddierers und erweitere diese Stück für Stück!“

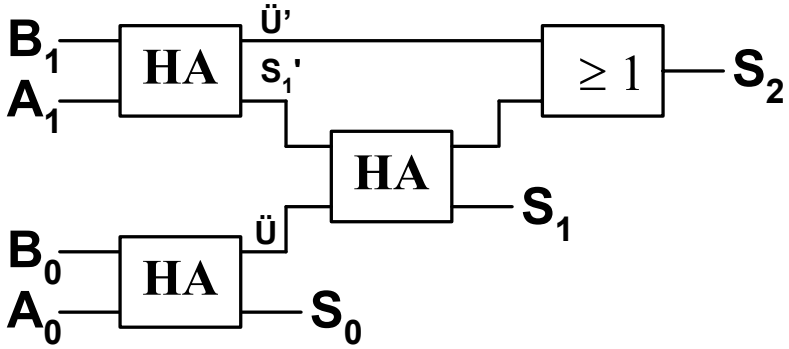


Abbildung 10: Addierwerk aus drei Halbaddierern für die beiden 2-Bit-Binärzahlen $A=(A_1A_0)_2$ und $B=(B_1B_0)_2$

[20] Die Bit-Bezeichnungen S für Summe und Ü für Übertrag stammen aus [L8], H steht für Hilfsbit.

Je nach Zeitbedarf und Vorkenntnissen der Schüler ist es möglich, dass nur ein Teil der Wahrheitstafel besprochen wird und der Rest als Hausaufgabe dient. Sollte die Aufgabe im Unterricht beendet werden, so bietet sich die Analyse einer weiteren Schaltung als Hausaufgabe an. Denkbar wäre z.B. ein Vorwärtszähler für eine 3-Bit-Dualzahl - vgl. [L8], Seite 25.

Doppelstunde 2:

Lernziele: Die Schüler sollen…

- den Aufbau der Vergleicherschaltung für 1-Bit-Dualzahlen angeben und die Wahrheitstafel aufstellen können
- die Funktionsweise des Vergleichers für 2-Bit-Dualzahlen exemplarisch nachvollziehen können
- den Aufbau einer Registermaschine erläutern können
- das Programm *Multiplikation.rmp* erklären und exemplarisch ein Registerprotokoll anfertigen können

Nachdem die Schüler in der ersten Doppelstunde gelernt haben, wie die Addition zweier Dualzahlen auf Bit-Ebene technisch realisiert werden kann, schließt nun der Vergleich von Dualzahlen den technischen Prolog ab. Entsprechende Vergleicher-Schaltungen für 1-Bit- und 2-Bit-Dualzahlen finden sich in [L8] auf Seite 12 ff und werden daher hier nicht explizit angegeben. Wiederum sollten die Schaltungen mit Hilfe von Logikbausteinen real vorgeführt werden.

Je nach Zeit könnte sich noch die Besprechung von Multiplizierer-Schaltwerken anschließen. Dies ist jedoch nicht unbedingt notwendig, weil man den Schülern verdeutlich kann, dass sich Multiplikationen durch wiederholtes Additieren ausführen lassen (s. u.). Außerdem würde es den Zeitrahmen dieser Doppelstunde sprengen.

Die Schüler sollten bisher erkannt haben, dass sich Additionen und Vergleiche technisch ausführen lassen. Weil die auftretenden Schaltungen jedoch mit wachsender Stellenzahl schnell kompliziert werden, macht es wenig Sinn, Schaltwerke für größere Dualzahlen zu betrachten. Daher erfolgt nun die Besprechung des Aufbaus einer Registermaschine, die als Modell für einen Mikroprozessor vorgestellt wird. Die Thematisierung der Funktionsweise einer realen CPU würde den Zeitrahmen sprengen und ist daher nicht vorgesehen. Anhand des Programms *Multiplikation.rmp* wird den Schülern nun einerseits die Syntax der Programmiersprache verdeutlicht und andererseits erfahren sie, wie die Multiplikation zweier Zahlen auf Additionen zurückgeführt werden kann (vgl. 5.2).

Weitere Aufgabe zur Binnendifferenzierung oder als Hausaufgabe:

- „Erstelle anhand des Quelltextes ein Protokoll der Registerinhalte!“
- „Erstelle ein Registermaschinenprogramm, das die Potenzierung auf die Multiplikation zurückführt! Gehe davon aus, dass die Befehle *mult n* und *cmult n* im Befehlssatz vorhanden sind!“
- „Erstelle anhand des Quelltextes[21] ein Protokoll der Registerinhalte und finde heraus, was das Programm leistet!“

[21] Gemeint ist der des Programms „Potenzberechnung.rmp“.

- „Erstelle ein Registermaschinenprogramm, das das Maximum zweier natürlicher Zahlen berechnet!“

Die Programmieraufgaben sind so gewählt, dass der bisherige Befehlssatz dazu ausreicht. Der Registermaschinen-Simulator muss den Schülern zu Hause noch nicht vorliegen - die Protokollaufgaben wären sonst auch nicht sinnvoll.

Doppelstunde 3:

Lernziele: Die Schüler sollen…
- die Arbeitsweise des Pascal-Programms „fakultaet“ erläutern können
- ein Registermaschinenprogramm zur Berechnung der Fakultät erstellen können
- eine bedingte Anweisung in ein Registermaschinenprogramm übersetzen können
- die Simulationsumgebung bedienen können

Die Stunde beginnt mit der Hausaufgabenbesprechung. Einer oder mehrere Schüler geben ihr Programm in die Simulationsumgebung ein bzw. testen ihr Protokoll mit der Simulationsumgebung. Danach wird der komplette Befehlssatz der Registermaschine besprochen. Insbesondere stellen die Schüler fest, dass die typischen Befehle einer höheren Programmiersprache wie Schleifen, bedingte Anweisungen etc. fehlen. Daher drängt sich eine weitere wichtige Frage auf: „Wie schafft es ein Mikroprozessor, trotz des einfachen Befehlssatzes, Pascal[22]-Programme auszuführen?“

Um diese Frage zu beantworten, soll exemplarisch ein Pascal-Beispielprogramm in Registermaschinencode übersetzt werden. Hierzu eignet sich z.B. die folgende Pascal-Funktion zur iterativen Berechnung der Fakultät, denn sie enthält bereits eine Schleife:

```
function fakultaet(n : integer) : integer;
var i,fak : integer;
begin
  fak := 1;
  for i:=2 to n do
    fak := fak*i;
  fakultaet := fak
end;
```

Nachdem der Pascal-Quellcode mit den Schülern besprochen wurde, wird gemeinsam versucht, ein Registermaschinenprogramm mit gleicher Funktionalität zu entwerfen. Dieses könnte letztlich wie das mitgelieferte Programm *Fakultät.rmp* aussehen:

```
// Berechnung der Fakultät
// R1:n  ->  R2:n!
// Dirk von Sierakowsky 08/2004
10 cload 1
20 mult 1
30 store 2
40 load 1
50 csub 1
60 store 1
70 jzero 100
80 load 2
```

[22] Selbstverständlich könnte hier jede andere höhere Programmiersprache stehen.

```
90 goto 20
100 end
```

Besonderer Wert sollte auf das Verständnis der Übersetzung der Schleife (Zeilen 40-70) gelegt werden.

Als Nächstes erfolgt die Übersetzung von bedingten Abfragen in Registermaschinencode. Relativ einfach ist eine Abfrage der Form „IF a=2 THEN …“ zu übertragen, etwas aufwändiger sind „IF a>b THEN …“ (vgl. *Bedingte Anweisung.rmp*) oder „IF a=b THEN …“ umzusetzen. Die Schüler erhalten eine dieser Programmieraufgaben im Unterricht und eine andere als Übung für zu Hause, wobei ihnen die Simulationsumgebung nun zur Verfügung stehen sollte.

Zur Differenzierung bietet es sich an, schnellen Schülern etwas komplexere Probleme zu stellen. Möglich wäre die Berechnung des Maximums von zwei Zahlen (*Maximum 2.rmp*) oder des kleinsten gemeinsamen Vielfachen von zwei Zahlen (*kgV.rmp*).

Die Schüler sollten nach dieser Stunde ein Gefühl dafür erlangt haben, wie ein Computer komplizierte Programme einer höheren Programmiersprache ausführen kann - womit die anfangs gestellte Frage beantwortet ist.

Doppelstunde 4

Lernziele: Die Schüler sollen…

- wissen, was eine berechenbare Funktion[23] ist
- die Definition eines Problems als Funktion $f : IN^k \to IN$ erläutern können
- wissen, dass die Formulierung eines Problems als Funktion nichts über deren Berechenbarkeit aussagt
- den Begriff „Algorithmus“ definieren können und wissen, dass Registermaschinenprogramme eine Möglichkeit der Konkretisierung des Begriffs darstellen
- ein Registermaschinenprogramm zur Berechnung der Funktion *n mod m* erstellen können
- die Churchsche These nennen können
- wissen, dass Registermaschinen und moderne Rechner dieselbe Klasse berechenbarer Funktionen besitzen

Hauptziel dieser Stunde ist die Formulierung der Churchschen These. Vorbereitend dazu werden zunächst die Begriffe „Berechenbarkeit“ (vgl. 2.2) und „Algorithmus“ definiert, wobei unter den mehreren verschiedenen Möglichkeiten zur Algorithmusdefinition das Registermaschinenprogramm hervorgehoben wird.

Zur Übung werden für einige der bereits besprochenen Registermaschinenprogramme die berechenbaren Funktionen, die sich hinter den Problemen verbergen, herausgearbeitet. Zur Festigung der bisher gelernten Programmierkenntnisse dient die Aufgabe, zur Berechnung einer weiteren vorgegebenen Funktion ein Registermaschinenprogramm zu schreiben, z.B. für $f(n,m) := n \bmod m$ (vgl. *Modulo.rmp*).

[23] Analog zum Konzept [L2] wird auch hier anstatt von „partiellen Funktionen“ didaktisch vereinfacht von „Funktionen“ gesprochen werden.

Zum Schluss wird die Churchsche These formuliert, und ihre Bedeutung für die Registermaschinen-Berechenbarkeit wird diskutiert.

Ausblick

Es bietet sich an, die Untersuchung von Funktionen anzuschließen, die prinzipiell nicht berechenbar sind oder von denen nicht klar ist, ob sie berechenbar sind :

- Das Registermaschinenprogramm *Ulam.rmp* kann dazu dienen, eine Funktion zu behandeln, von der zur Zeit nicht bekannt ist, ob sie berechenbar ist
- Das Halteproblem als Beispiel für eine nicht berechenbare Funktion lässt sich gemäß [L2] im Unterricht besprechen
- Das Thema „Entscheidbarkeit von Mengen“ lässt sich gut in die Thematik integrieren. Mit Hilfe des Programms *Primzahltest.rmp* kann die Menge aller Primzahlen entscheidbar klassifiziert werden. Mit Hilfe von *Ulam.rmp* kann diskutiert werden, dass die Menge aller Zahlen, deren Ulam-Folgenglieder 1 werden, semientscheidbar ist – und dass nicht klar ist, ob sie sogar entscheidbar ist.

Letztlich könnte die Überleitung zu dem Thema „Endlichen Automaten“ mit folgender Fragestellung gelingen: „Besitzen Maschinen, die einfacher aufgebaut sind als Registermaschinen, dieselbe Klasse berechenbarer Funktion?“

4.3.2 Zugang 2: Berechenbarkeitstheorie

Auch bei einem Einstieg über die Theorie der Berechenbarkeit lassen sich Registermaschinen sinnvoll in den Unterricht einbauen. Mein Vorschlag dazu stellt eine Erweiterung der Unterrichtsreihe [L2] dar, die die folgenden Intentionen verfolgt:

- Die ersten fünf Doppelstunden von [L2] sind sehr theorielastig. Durch das Erstellen von Registermaschinenprogrammen erhalten die Schüler die Möglichkeit, sich zwischendurch mit praktischen Dingen zu beschäftigen. Auch für die Konzeption der ersten Kursarbeit ist dies von Vorteil, denn so finden sich leichter konkrete Aufgaben aus den Anforderungsebenen I und II.
- Der Gedanke, dass ein Problem einer Funktion entspricht, wird mit Hilfe der Registermaschinenprogramme anhand einiger konkreter Beispiele vertieft.
- Ein konkreter und zudem recht einfacher Algorithmus-Begriff steht den Schülern durch die Registermaschinenprogramme zur Verfügung.
- Die Schüler lernen ein Modell eines Mikroprozessors sowie der Programmiersprache „Assembler“ kennen. Auch im Hinblick auf das Wahlthema „Technische Informatik“ in der 13.2 werden so Zusammenhänge geschafft.

Die ersten beiden Doppelstunden der genannten Unterrichtsreihe bleiben zunächst unverändert und werden daher nur kurz zusammengefasst:

Doppelstunde 1

Das Färbeproblem eines Graphen führt auf den Unterschied zwischen Algorithmen mit polynomialer Laufzeit (berechenbar und durchführbar) und Algorithmen mit exponentieller Laufzeit (berechenbar, aber nicht durchführbar).

Doppelstunde 2

Anhand des Beispiels „wundersame Zahlen“ - das sind solche Zahlen, deren Ulam-Folgenglieder irgendwann den Wert 1 annehmen - wird die Definition eines Problems als Funktion vorbereitet.

Doppelstunde 3

Lernziele: Die Schüler sollen…

- die Definition eines Problems als Funktion $f : IN^k \to IN$ am Beispiel „wundersame Zahlen“ erläutern können
- wissen, dass die Formulierung eines Problems als Funktion nichts über deren Berechenbarkeit aussagt
- den Begriff Algorithmus definieren können und wissen, dass Registermaschinenprogramme eine Möglichkeit der Konkretisierung dieses Begriffs darstellen
- die Wirkung der im Programm *Ulam.rmp* enthaltenen Registermaschinenbefehle angeben können

Im ersten Teil der Doppelstunde wird, in Anknüpfung an das Beispiel „wundersame Zahlen“, mit den Schülern erarbeitet, dass ein Problem stets als Funktion $f : IN^k \to IN$ aufgefasst werden kann[24]. Die Frage, wie die Funktionswerte berechnet werden, leitet über zum Thema „Registermaschine“, das als mehrstündiger Einschub gedacht ist. Bis zu dieser Stelle wurde das Konzept [L2] mit Ausnahme der Änderung der Funktionsdefinition nicht modifiziert.

Zunächst erhalten die Schüler eine informelle Algorithmusdefinition („Ein Algorithmus ist eine endliche Folge von Anweisungen“). Es wird erarbeitet, dass es viele Arten der Beschreibung von Algorithmen geben kann, wobei die Registermaschinenprogramme als eine spezielle Möglichkeit vorgestellt werden. Als Übung wird das Programm (*Ulam.rmp*) mit Hilfe von ReSi untersucht. Dies sollte gemeinsam geschehen, weil es einen relativ umfangreichen Quelltext besitzt, der die Schüler sonst gleich überfordern würde. Die Wirkung der enthaltenen Befehle wird Schritt für Schritt erarbeitet, wobei Zusammenhänge zum bereits bekannten Pascal-Programm aus Doppelstunde 2 hergestellt werden.

Hausaufgabe: „Erstelle anhand des ausgeteilten Quelltextes ein Protokoll der Registerinhalte. Beim Programmstart soll die Belegung (0,4,0,0,…) vorliegen.“

Doppelstunde 4

Lernziele: Die Schüler sollen…

- die Funktionsweise einfacher Registermaschinenprogramme nachvollziehen können
- in Kleingruppenarbeit einfache Registermaschinenprogramme zur Berechnung von Funktionen verfassen können
- die bisher vorkommenden Registermaschinenbefehle kennen

[24] In der nächsten Stunde werden Funktionen berechnet, deren Definitionsbereich IN^2 ist. Daher wird die Definition an dieser Stelle etwas allgemeiner als in [L2] gegeben.

Um den Umgang mit Registermaschinenprogrammen zu üben, werden in dieser Stunde nach der Hausaufgabenbesprechung einige einfache Funktionen mit Registermaschinenprogrammen berechnet. Die Programme sollten überschaubar sein und noch keine Befehle mit indirekter Adressierung beinhalten. Als Vorbereitung wird der bisher bekannte Befehlssatz wiederholt und ggf. etwas erweitert. Mögliche Funktionen wären z.B.

- $f_1(n,m) := n + m$, siehe *Summe 2.rmp*
- $f_2(n,m) := n \bmod m$, siehe *Modulo.rmp*
- $f_3(n) := \begin{cases} 1 & falls\ n > 5 \\ 0 & sonst \end{cases}$, ähnlich wie *Bedingte Anweisung.rmp*
- $f_4(n.m) := \max(n,m)$, siehe *Maximum 2.rmp*

Die Schüler werden in Gruppen eingeteilt, weil die Anforderungen, die die neue Programmiersprache stellt, zunächst für viele sehr hoch sind. Jede Gruppe erhält eine Funktion mit angemessenem Schwierigkeitsgrad, im zweiten Teil der Doppelstunde werden die erarbeiteten Lösungen präsentiert. Haben Gruppen verschiedene Lösungen zu einem Problem erarbeitet, so können diese bzgl. der benötigten Schritte beim Ablauf verglichen werden. Sehr leistungsstarke Schüler erhalten als Aufgabe, sich über den Euklidschen Algorithmus zu informieren und mit diesem die Funktion $f_5(m,n) := ggT(m,n)$ zu berechnen (siehe *ggT nach Euklid.rmp*).

Als Hausaufgabe sollen die Schüler zur Vertiefung mit Hilfe von ReSi die Addition zweier Bruchzahlen (ohne anschließendes Kürzen) berechnen (vgl. *Bruchaddition.rmp*).

Doppelstunde 5

Lernziele: Die Schüler sollen…
- die Wirkungsweise des Programms *Summe 10.rmp* erläutern können
- die Befehle mit indirekter Adressierung kennen
- den Quelltext zum Programm *Summe 15.rmp* erstellen können

Nach der Besprechung der Hausaufgabe wird gemeinsam das Programm *Summe 10.rmp* untersucht:

```
// Berechne die Summe von R1 bis
// R10, Ergebnis in R13
// Dirk von Sierakowsky, 08/2004
10 cload 1
20 store 12
// Addieren der nächsten Zahl
30 iload 12
40 add 13
50 store 13
// Zähler erhöhen
60 load 12
70 cadd 1
80 store 12
// Grenze erreicht?
90 csub 10
100 jzero 30
// Aufräumen
110 store 12
120 end
```

Die Aufgaben der einzelnen Register werden besprochen, insbesondere wird jedoch der neue Befehl „iload 12“ untersucht. Die weiteren Befehle mit indirekter Adressierung werden den Schülern der Vollständigkeit halber mitgeteilt. Sie erhalten die Aufgabe, ein analoges Programm *Summe 15.rmp* zu schreiben, das im Anschluss besprochen wird.

Je nach Zeit können noch weitere Programme mit indirekter Adressierung behandelt werden, vgl. z.B. *Maximum 10.rmp*. Je nach Leistungsfähigkeit der Schüler und vorhandener Unterrichtszeit wäre es auch möglich, einige einfache Programmieraufgaben zu verteilen (z.B. Umsetzen von bedingten Abfragen, vgl. Doppelstunde 3 aus 4.3.1). Die Erstellung eines Sortier-Programms (vgl. *BubbleSort.rmp*) ist anspruchsvoll und eignet sich zur Vergabe als kleine Präsentation.

Mit dieser Doppelstunde ist der Einschub über die Registermaschinen beendet. Je nach Geschmack ließe sich in der folgenden Doppelstunde noch die Churchsche These, die für den Grundkurs ohnehin nur fakultativ im Lehrplan vorgesehen ist, besprechen.

Mit der Frage, ob es nicht-berechenbare Funktionen gibt, wird die Fortführung des ursprünglichen Unterrichtsganges motiviert. Die Reihe [L2] wird exakt an der Stelle fortgeführt, an der sie unterbrochen wurde, und führt schließlich zum Halteproblem.

4.3.3 Methodische Variation der Aufgaben

Zum Abschluss dieses Kapitels möchte ich noch einige methodische Variationen für die Arbeit mit Registermaschinenprogrammen vorschlagen:

(i) **Vervollständigen von Quelltexten**: Gerade wenn die Schüler noch wenig Erfahrung im Umgang mit Registermaschinenprogrammen besitzen, bietet es sich an, einen Teil des Quelltextes vorzugeben und diesen vervollständigen zu lassen.
(ii) **Erstellen möglichst effektiver Registermaschinenprogramme**: Verschiedene Programme der Schüler können bezüglich ihrer Effizienz verglichen werden, indem dasjenige Programm gesucht wird, welches gemäß dem Schrittzähler der Registermaschine das Problem am schnellsten löst.
(iii) **Verfassen von Programmen mit ähnlicher Struktur**: Ein Beispiel wäre die Änderung des Programms *Bedingte Anweisung.rmp*, sodass es eine ähnliche bedingte Anweisung realisiert. Ähnlich aufgebaute Programme sind auch *Potenz.rmp* und *Multiplikation.rmp* sowie *Summe 10.rmp* und *Maximum 10.rmp*.
(iv) **Erstellen eines Protokolls der Registerinhalte**: Um das Verständnis für den Befehlssatz zu erhöhen, empfiehlt es sich, zu einem gegebenen Quelltext ein Protokoll der Registerinhalten anfertigen und damit die Aufgabe des Programms herausfinden zu lassen. Dies lässt sich später mit dem Simulationsprogramm kontrollieren (vgl. 3.2.4).
(v) **Herausfinden des zu mehreren Registerprotokollen gehörigen Quelltextes**: Eine Knobelaufgabe, die mehrere Protokolle desselben Programms mit unterschiedlichen Eingabewerten voraussetzt. Enthält der Quelltext Schleifen oder Sprünge, ist die Aufgabe fast unlösbar.
(vi) **Ergänzen von Kommentaren**: In einen vorgegebenen, unkommentierten Quelltext sollen sinnvolle Kommentare integriert werden.

5 Anhang

5.1 Quelltexte des Delphi-Projekts

Alle Quelltexte wurden in der Programmiersprache Delphi 6.0 der Firma Borland geschrieben und befinden sich ebenfalls auf der mitgelieferten CD.

1. ReSi.dpr

```
program ReSi;

(*********************************************)
(*  Registermaschinen-Simulator (ReSi)       *)
(*            (10.08.2004)                   *)
(*                                           *)
(*  Copyright ® 2004 by Dirk von Sierakowsky *)
(*  E-Mail: dirk.v.sierakowsky@gmx.de        *)
(*- - - - - - - - - - - - - - - - - - - - - - *)
(*  Ich übernehme keine Haftung für etwaige  *)
(*  Schäden, die durch dieses Programm       *)
(*  verursacht werden                        *)
(*- - - - - - - - - - - - - - - - - - - - - - *)
(*  Dieses Unit ist FREEWARE.                *)
(*  Alle Rechte vorbehalten                  *)
(*********************************************)

uses
  Forms,
  UReSi in 'UReSi.pas' {Hauptformular},
  UReSiProtokoll in 'UReSiProtokoll.pas' {ProtokollFormular},
  UReSiOptionen in 'UReSiOptionen.pas' {OptionenFormular};

{$R *.res}

begin
  Application.Initialize;
  Application.Title := 'ReSi 1.0';
  Application.HelpFile := 'hilfedateien\ReSi.hlp';
  Application.CreateForm(THauptformular, Hauptformular);
  Application.CreateForm(TProtokollFormular, ProtokollFormular);
  Application.CreateForm(TOptionenFormular, OptionenFormular);
  Application.Run;
end.
```

2. *UReSi.pas*

```
unit UReSi;

(*********************************************)
(*  UReSi            (10.08.2004)            *)
(*                                           *)
(*  Unit des Projektes ReSi                  *)
(*  Copyright ® 2004 by Dirk von Sierakowsky *)
(*  E-Mail: dirk.v.sierakowsky@gmx.de        *)
(*- - - - - - - - - - - - - - - - - - - - - - *)
(*  Ich übernehme keine Haftung für etwaige  *)
(*  Schäden, die durch diese Unit            *)
(*  verursacht werden                        *)
(*- - - - - - - - - - - - - - - - - - - - - - *)
(*  Diese Unit ist FREEWARE.                 *)
(*  Alle Rechte vorbehalten                  *)
(*********************************************)


interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Menus, Buttons, ExtCtrls, Grids, ValEdit,
  RegisterEditor, ComCtrls, RMProgramEditor, Printers, ImgList, ToolWin;

const noOfCells = 50; // Beschränkung der Anzahl der Register auf 50 + Akkumulator

type
  // Ergebnistypen beim Ende oder Abbruch der Ausführung eines RM-Programms
  TResult = (ok, cancel, terminated, divThroughZero, unknownRow, unknownCommand,
             regNoOutOfRange, invalidRowNo, accuOverflowError);

  THauptformular = class(TForm)
    MainMenu: TMainMenu;
    QuellcodeGroupBox : TGroupBox;
    BelegungGroupBox : TGroupBox;
    Datei: TMenuItem;
    Hilfe: TMenuItem;
    Neu: TMenuItem;
    Oeffnen: TMenuItem;
    Speichern: TMenuItem;
    Speichernunter: TMenuItem;
    Drucken: TMenuItem;
    Beenden: TMenuItem;
    Anleitung: TMenuItem;
    Information: TMenuItem;
    SaveDialog: TSaveDialog;
    OpenDialog: TOpenDialog;
    RegisterEditor: TRegisterEditor;
    TaktrateTrackBar: TTrackBar;
    PrintDialog: TPrintDialog;
    TaktrateLabel: TLabel;
    StepsLabel: TLabel;
    ToolBar: TToolBar;
    NewButton: TToolButton;
    OpenButton: TToolButton;
    SaveButton: TToolButton;
    PrintButton: TToolButton;
    SpaceToolButton1: TToolButton;
    StepButton: TToolButton;
    ToolBarImageList: TImageList;
    RunButton: TToolButton;
    StopButton: TToolButton;
    ResetButton: TToolButton;
    RegSaveButton: TToolButton;
    RegLoadButton: TToolButton;
    RegClearButton: TToolButton;
    SpaceToolButton3: TToolButton;
    HelpButton: TToolButton;
    InfoButton: TToolButton;
    Betrieb: TMenuItem;
    Einzelschritt: TMenuItem;
    ProgrammStart: TMenuItem;
    ProgrammStop: TMenuItem;
    ProgrammReset: TMenuItem;
```

```
    RegisterSpeichern: TMenuItem;
    RegisterLaden: TMenuItem;
    RegisterClear: TMenuItem;
    Protokoll: TMenuItem;
    Protokolloptionen: TMenuItem;
    SpaceToolButton2: TToolButton;
    ProgramMemo: TRMProgramEditor;
    Kopieren: TMenuItem;
    CopyButton: TToolButton;
    SpaceToolButton: TToolButton;

    procedure SpeichernClick(Sender: TObject);
    procedure SpeichernunterClick(Sender: TObject);
    procedure OeffnenClick(Sender: TObject);
    procedure NeuClick(Sender: TObject);
    procedure InformationClick(Sender: TObject);
    procedure DruckenClick(Sender: TObject);
    procedure BeendenClick(Sender: TObject);
    procedure AnleitungClick(Sender: TObject);
    procedure EinzelschrittClick(Sender: TObject);
    procedure ProgrammStartClick(Sender: TObject);
    procedure ProgrammStopClick(Sender: TObject);
    procedure ProgrammResetClick(Sender: TObject);
    procedure RegisterLadenClick(Sender: TObject);
    procedure RegisterSpeichernClick(Sender: TObject);
    procedure RegisterClearClick(Sender: TObject);
    procedure ProtokollClick(Sender: TObject);
    procedure ProtokolloptionenClick(Sender: TObject);
    procedure RegisterEditorClick(Sender: TObject);
    procedure KopierenClick(Sender: TObject);

    procedure ProgramMemoChange(Sender: TObject);
    procedure exitProgram(Sender: TObject; var canClose : boolean);
    procedure updateFrequency(Sender: TObject);
    function getActFileName : TFileName;
  private
    { Private-Deklarationen }
    stopped : boolean;
    actFileName : TFileName;  // globale Variable für die momentan im Speicher befindliche
Programm-Datei
    actProgramRow : integer;  // globale Variable für die aktuell auszuführende interne
Programmzeile

    procedure executeProgramRow(var row : integer; var result : TResult);
    procedure refreshTitleBar;
    procedure showResult(result : TResult);
    procedure setCounter(number : integer);
    procedure incCounter;
    function getFirstProgramRow : integer;
    procedure enableButtons(new, open, save, print, copy, exit, run, stop, step, reset,
regSave, regLoad, regClear, opt, help, info : boolean);
    procedure forceReset;
  public
    { Public-Deklarationen }
    constructor Create(AOwner: TComponent); override;
  end;


var Hauptformular: THauptformular;


implementation

uses UReSiProtokoll, UReSiOptionen;

{$R *.dfm}
{$Q+}  // erzwingt die Überlaufprüfungen bei allen Rechnungen


// ----------------------------------------------------------------------------
// Methoden zur Ereignissteuerung
// ----------------------------------------------------------------------------


procedure THauptformular.ProgrammResetClick(Sender: TObject);
// versetzt das Programm in den Ausgangszustand, aus dem es gestartet werden kann
var row : integer;
begin
```

```
  if ProgramMemo.Lines[0] <> '' then
  begin
    row := getFirstProgramRow;
    actProgramRow := row;
    setCounter(0);
    if OptionenFormular.ProgrammzeilemarkierenCheckBox.Checked then
      ProgramMemo.markProgramRow(row);
    ProtokollFormular.clearProtocol;
    ProtokollFormular.actualizeProtocol(actProgramRow);
    enableButtons(true,true,true,true,true,
true,true,false,true,true,true,true,true,true,true,true)
  end
end;

procedure THauptformular.RegisterClearClick(Sender: TObject);
// löscht alle Registerinhalte
begin
  RegisterEditor.clear
end;

procedure THauptformular.RegisterSpeichernClick(Sender: TObject);
// legt alle Registerinhalte im Zwischenspeicher ab
begin
  RegisterEditor.storeRegisters
end;

procedure THauptformular.RegisterLadenClick(Sender: TObject);
// lädt alle Registerinhalte aus dem Zwischenspeicher
begin
  RegisterEditor.recallRegisters;
  forceReset
end;

procedure THauptformular.EinzelschrittClick(Sender: TObject);
// führt die aktuelle Programmzeile aus
var result : TResult;
begin
  executeProgramRow(actProgramRow,result);
  if result = ok then
    incCounter
  else
    showResult(result);
  ProtokollFormular.actualizeProtocol(actProgramRow)
end;

procedure THauptformular.ProgrammStartClick(Sender: TObject);
// started das Registermaschinenprogramm im Ablaufmodus
var i,
    timeToWait : integer;
    result : TResult;
begin
  stopped := false;

enableButtons(false,false,false,false,false,false,false,true,false,false,false,false,false,fal
se,false,false);

  //Durchlaufe alle Zeilen des Programms:
  executeProgramRow(actProgramRow,result);
  ProtokollFormular.actualizeProtocol(actProgramRow);
  while (result = ok) and not stopped do
  begin
    timeToWait := 1050 div Trunc(10*(exp(TaktrateTrackBar.Position/1000)-1)); //Logarithmische
Trackbar
    incCounter;
    for i := 1 to timeToWait do
    begin
      Application.ProcessMessages; //Ermöglich der Anwendung, auf Ereignisse zu reagieren
      sleep(10) //Wartet 10 ms
    end;
    Application.ProcessMessages;
    executeProgramRow(actProgramRow,result);
    ProtokollFormular.actualizeProtocol(actProgramRow)
  end;

  //Abbruch durch den Benutzer?
  if stopped then
  begin
    MessageDlg('Abbruch durch den Benutzer!',mtInformation,[mbOk],0);
```

```
enableButtons(true,true,true,true,true,true,true,false,true,true,true,true,true,true,true,true
);
    ProgramMemo.SetFocus
  end
  else
  begin
    showResult(result);

enableButtons(true,true,true,true,true,true,false,false,false,true,true,true,true,true,true,tr
ue)
  end
end;

procedure THauptformular.KopierenClick(Sender: TObject);
// speichert den Quelltext des Registermaschinenprogramms in der Windows-Zwischenablage
var cursorPos : integer;
begin
  cursorPos := ProgramMemo.SelStart;
  ProgramMemo.SelectAll;
  ProgramMemo.CopyToClipboard;
  ProgramMemo.SelStart := cursorPos
end;

procedure THauptformular.ProgrammStopClick(Sender: TObject);
// stopped das laufende Registermaschinenprogramm
begin
  stopped := true
end;

procedure THauptformular.InformationClick(Sender: TObject);
// ruft den Context Nr. 5 des Hilfesystems auf
begin
  Application.HelpContext(5)
end;

procedure THauptformular.BeendenClick(Sender: TObject);
// schließt das Programm
begin
  Close
end;

procedure THauptformular.SpeichernClick(Sender: TObject);
// ruft den Speichern-Dialog auf
begin
  ForceCurrentDirectory := true;
  if length(actFileName) > 0 then
    ProgramMemo.Lines.SaveToFile(actFileName)
  else
  begin
    if SaveDialog.Execute then
    begin
      ProgramMemo.Lines.SaveToFile(SaveDialog.FileName);
      actFileName := SaveDialog.FileName;
      refreshTitleBar
    end
  end
end;

procedure THauptformular.SpeichernunterClick(Sender: TObject);
// ruft den Speichernunter-Dialog auf
begin
  ForceCurrentDirectory := true;
  if SaveDialog.Execute then
  begin
    ProgramMemo.Lines.SaveToFile(SaveDialog.FileName);
    actFileName := SaveDialog.FileName;
    refreshTitleBar
  end
end;

procedure THauptformular.OeffnenClick(Sender: TObject);
// ruft den Öffnen-Dialog auf
begin
  ForceCurrentDirectory := true;
  if OpenDialog.Execute then
    ProgramMemo.Lines.LoadFromFile(OpenDialog.FileName);
  actFileName := OpenDialog.FileName;
```

```
  if OptionenFormular.ProgrammzeilemarkierenCheckBox.Checked then
    ProgramMemo.markProgramRow(0);
  refreshTitleBar;
  ProgrammResetClick(Sender)
end;

procedure THauptformular.NeuClick(Sender: TObject);
// bereitet ein neues Registermaschinenprogramm vor
var i : byte;
begin
  actFileName := '';
  RegisterEditor.clear;
  ProgramMemo.Clear;
  setCounter(0);
  for i:=1 to 10 do
    ProgramMemo.Lines.Add(IntToStr(10*i)+' ');
  ProgramMemo.SelStart := 4; //Setzt den Cursor
  refreshTitleBar;

enableButtons(true,true,true,true,true,true,false,false,false,true,true,true,true,true,true,tr
ue)
end;

procedure THauptformular.DruckenClick(Sender: TObject);
// ruft den Drucken-Dialog auf und druckt den Quelltext formatiert
var line: integer;
    printText: textFile;
begin
  if actFileName = '' then
    MessageDlg('Sie müssen die Datei zuerst speichern!!',mtInformation,[mbOk],0)
  else
    if printDialog.execute then
    begin
      AssignPrn(printText);
      Rewrite(printText);
      Printer.Canvas.Font := ProgramMemo.Font;
      Printer.Canvas.Font.Size := 11;
      writeln(printText,'');
      writeln(printText,'');

      //Ausgabe der Strings aus ProgramEdit
      for line := 0 to ProgramMemo.Lines.Count-1 do
        writeln(printText,#9+ProgramMemo.Lines[Line]);

      writeln(printText,'');
      writeln(printText,'');
      Printer.Canvas.Font.Size := 10;
      writeln(printText,#9+actFileName);
      CloseFile(printText)
    end
end;

procedure THauptformular.AnleitungClick(Sender: TObject);
// ruft das Hilfesystem auf
begin
  Application.HelpCommand(HELP_FINDER,0)
end;

procedure THauptformular.ProtokollClick(Sender: TObject);
// zeigt oder verbirgt das Protokoll-Fenster
begin
  if Protokoll.Checked then
  begin
    ProtokollFormular.Hide;
    Protokoll.Checked := false
  end
  else
  begin
    ProtokollFormular.Show;
    Protokoll.Checked := true
  end
end;

procedure THauptformular.ProtokolloptionenClick(Sender: TObject);
// zeigt das Optionen-Fenster
begin
  OptionenFormular.Show
end;
```

```
procedure THauptformular.ProgramMemoChange(Sender: TObject);
// erzwingt einen Neustart des Registermaschinenprogramms
begin
  forceReset
end;

procedure THauptformular.RegisterEditorClick(Sender: TObject);
// erzwingt einen Neustart des Registermaschinenprogramms
begin
  forceReset
end;

// ------------------------------------------------------------------------
// "Hilfsmethoden"
// ------------------------------------------------------------------------

procedure THauptformular.setCounter(number : integer);
// setzt den Schrittzähler
begin
  StepsLabel.Caption := intToStr(number)
end;

procedure THauptformular.incCounter;
// erhöht den Wert des Schrittzählers
begin
  StepsLabel.Caption := intToStr(strToInt(StepsLabel.Caption)+1)
end;

procedure THauptFormular.refreshTitleBar;
// aktualisiert die Titelzeile des Hauptfensters mit dem aktuellen Dateinamen
var displayedFileName : string;
    position : integer;
begin
  //Pfadangabe abschneiden
  position := LastDelimiter('\',actFileName);
  displayedFileName := Copy(actFileName,position+1,Length(actFileName)-position);

  //Überschrift anpassen
  if actFileName = '' then
    Caption := 'ReSi 1.0'
  else
    Caption := 'ReSi 1.0 (' + displayedFileName + ')'
end;

procedure THauptformular.showResult(result : TResult);
// gibt eine Fehler- bzw. Erfolgsmeldung aus, wenn das Registermaschinenprogramm
// beendet bzw. abgebrochenwurde
begin
  case result of
      terminated : MessageDlg('Programm wurde erfolgreich
ausgeführt!',mtInformation,[mbOk],0);
      divThroughZero : MessageDlg('Division durch Null, Abbruch!',mtError,[mbOK,mbHelp],1);
      unknownRow : MessageDlg('Unbekanntes Sprungziel, Abbruch!',mtError,[mbOK,mbHelp],2);
      unknownCommand : MessageDlg('Unbekannter Befehl, Abbruch!',mtError,[mbOK,mbHelp],3);
      regNoOutOfRange : MessageDlg('Ungültige Registernummer,
Abbruch!',mtError,[mbOK,mbHelp],4);
      invalidRowNo : MessageDlg('Ungültige Zeilennummer, Abbruch!',mtError,[mbOK,mbHelp],7);
      accuOverflowError : MessageDlg('Akkumulatorüberlauf, Abbruch!',mtError,[mbOK,mbHelp],8)
  end;
  enableButtons(true,true,true,true,true, true,false,false,false,true,true,true,true,
true,true,true)
end;

procedure THauptformular.enableButtons(new, open, save, print, copy, exit, run, stop, step,
reset, regSave, regLoad, regClear, opt, help, info : boolean);
// aktiviert oder deaktiviert einzelne Menüpunkte
begin
  NewButton.Enabled := new;
  Neu.Enabled := new;

  OpenButton.Enabled := open;
  Oeffnen.Enabled := open;

  SaveButton.Enabled := save;
  Speichern.Enabled := save;
  Speichernunter.Enabled := save;
```

```
  PrintButton.Enabled := print;
  Drucken.Enabled := print;

  CopyButton.Enabled := copy;
  Kopieren.Enabled := copy;

  Beenden.Enabled := exit;

  RunButton.Enabled := run;
  ProgrammStart.Enabled := run;

  StopButton.Enabled := stop;
  ProgrammStop.Enabled := stop;

  StepButton.Enabled := step;
  Einzelschritt.Enabled := step;

  ResetButton.Enabled := reset;
  ProgrammReset.Enabled := reset;

  RegSaveButton.Enabled := regSave;
  RegisterSpeichern.Enabled := regSave;

  RegLoadButton.Enabled := regLoad;
  RegisterLaden.Enabled := regLoad;

  RegClearButton.Enabled := regClear;
  RegisterClear.Enabled := regClear;

  Protokolloptionen.Enabled := opt;

  HelpButton.Enabled := help;
  Anleitung.Enabled := help;

  InfoButton.Enabled := info;
  Information.Enabled := info
end;

procedure THauptformular.updateFrequency(Sender: TObject);
// aktualisiert die Beschriftung des Taktratenschiebereglers
begin
  TaktrateLabel.Caption := floatToStrF(exp(TaktrateTrackBar.Position/1000)-1,ffFixed,5,1)+' Hz
'
end;

procedure THauptformular.executeProgramRow(var row : integer; var result : TResult);
// führt den Befehl der Programmzeile row aus und liefert die interne Nummer der folgenden
Zeile
// sowie eine Erfolgsmeldung zurück

type TRMCommand = (load, iload, cload, mult, imult, cmult, add, iadd, cadd, sub, goto_,
             isub, csub, div_, idiv, cdiv, store, istore, jzero, ende, error);

var rowNoString : string;
    command : TRMCommand;
    argument,
    nextRow : integer;

  function isValidRegister(number : integer) : boolean;
  // prüft, ob ein Zahl zwischen 0 und 50 liegt und damit eine Registernummer sein kann
  begin
    if (number < 0) or (number > noOfCells) then
      isValidRegister := false
    else
      isValidRegister := true
  end;

  function getNextProgramRow(row : integer; command : TRMCommand; argument : string) :
integer;
  // sucht die interne Nummer der auf row folgenden Programmzeile (Kommentare werden
übersprungen)
  var maxRow,
      i : integer;
  begin
    maxRow := ProgramMemo.Lines.Count;
    case command of
      error : begin
                getNextProgramRow := 0;
```

```
              exit
            end;
      goto_ : begin
              // suche die Nummer der Zielzeile (argument) des Sprungbefehls
               i := 0;
               while (copy(trim(ProgramMemo.Lines[i]),1,length(argument)) <> argument) and
(i <= maxRow) do
                 inc(i);
               getNextProgramRow := i
            end;

      jzero : if RegisterEditor.getRegisterValue(0) = 0 then
            begin
              // suche die Nummer der Zielzeile (argument) des Sprungbefehls
               i := 0;
               while (copy(trim(ProgramMemo.Lines[i]),1,length(argument)) <> argument) and
(i <= maxRow) do
                 inc(i);
               getNextProgramRow := i;
            end
            else
            begin
              // kein Sprung nötig
              inc(row);
              while (copy(trim(ProgramMemo.Lines[row]),1,2) = '//') and (row < maxRow) do
                inc(row);
              getNextProgramRow := row
            end

      else
      begin
        // kein Sprungbefehl
        inc(row);
        while (copy(trim(ProgramMemo.Lines[row]),1,2) = '//') and (row < maxRow) do
          inc(row);
        getNextProgramRow := row
      end
    end
  end;

  procedure parseRow(rowString : string; var rowNoString : string; var command : TRMCommand;
var argument : integer);
  // zerlegt eine Programmzeile in ihre Bestandteile
  var commandPos,argumentPos : integer;
      commandString : string;
  begin
    // suche erstes Leerzeichen
    commandPos := pos(' ',rowString);
    if commandPos > 0 then
    begin
      // Regulärer RM-Befehl
      rowNoString := trim(copy(rowString,0,commandPos));
      delete(rowString,1,commandPos);
      argumentPos := pos(' ',trim(rowString));

      if argumentPos > 0 then
      // Befehl mit Argument
      begin
        commandString := copy(rowString,0,argumentPos);
        delete(rowString,1,argumentPos);

        // Die folgende Fehlerbehandlung läuft nur dann korrekt,
        // wenn unter Tools -> Debugger-Optionen -> Sprach-Exceptions
        // "Bei Delphi-Exceptions stoppen" deaktiviert ist:
        try
          argument := strToInt(trim(rowString));
        except
          argument := -1;
          commandString := ''
        end
      end
      else
      // Befehl ohne Argument
      begin
        commandString := rowString;
        argument := -1
      end
    end;
```

```
    // string befehl in einen vom Typ TBefehl umwandeln
    command := error;

    if CompareText(trim(commandString),'load')=0 then command := load
    else if CompareText(trim(commandString),'iload')=0 then command := iload
    else if CompareText(trim(commandString),'cload')=0 then command := cload

    else if CompareText(trim(commandString),'iadd')=0 then command := iadd
    else if CompareText(trim(commandString),'add')=0 then command := add
    else if CompareText(trim(commandString),'cadd')=0 then command := cadd

    else if CompareText(trim(commandString),'isub')=0 then command := isub
    else if CompareText(trim(commandString),'sub')=0 then command := sub
    else if CompareText(trim(commandString),'csub')=0 then command := csub

    else if CompareText(trim(commandString),'imult')=0 then command := imult
    else if CompareText(trim(commandString),'mult')=0 then command := mult
    else if CompareText(trim(commandString),'cmult')=0 then command := cmult

    else if CompareText(trim(commandString),'idiv')=0 then command := idiv
    else if CompareText(trim(commandString),'div')=0 then command := div_
    else if CompareText(trim(commandString),'cdiv')=0 then command := cdiv

    else if CompareText(trim(commandString),'store')=0 then command := store
    else if CompareText(trim(commandString),'istore')=0 then command := istore

    else if CompareText(trim(commandString),'goto')=0 then command := goto_
    else if CompareText(trim(commandString),'jzero')=0 then command := jzero
    else if CompareText(trim(commandString),'end')=0 then command := ende
  end;

begin
  // Programmzeile zerlegen, enthaltenen Befehl ausführen
  result := ok;
  parseRow(trim(HauptFormular.ProgramMemo.Lines[row]),rowNoString,command,argument);

  // Ist die Zeilennummer eine natürliche Zahl?
  try
    if strToInt(rowNoString) < 0 then
    begin
      result := invalidRowNo;
      exit
    end
  except
    on Exception do
    begin
      result := invalidRowNo;
      exit
    end
  end;

  // nächste Programmzeile suchen (Kommentare werden übersprungen,
  // bei command = error wird 0 zurückgeliefert)
  nextRow := getNextProgramRow(row,command,intToStr(argument));

  // aktuelle Programmzeile ausführen
  try
   case command of
    load : if isValidRegister(argument) then
             RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(argument))
           else
              result := regNoOutOfRange;

    cload : RegisterEditor.setRegisterValue(0,argument);

    iload : if isValidRegister(argument) then
              if isValidRegister(RegisterEditor.getRegisterValue(argument)) then

RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(RegisterEditor.getRegisterVa
lue(argument)))
              else
                // Inhalt des Registers ist keine gültige Registernummer
                result := regNoOutOfRange
            else
              // Argument von iload ist keine gültige Registernummer
              result := regNoOutOfRange;
```

```
    add : if isValidRegister(argument) then

RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)+RegisterEditor.getRegiste
rValue(argument))
           else
             result := regNoOutOfRange;

    cadd : RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)+argument);

    iadd : if isValidRegister(argument) then
              if isValidRegister(RegisterEditor.getRegisterValue(argument)) then

RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)+RegisterEditor.getRegiste
rValue(RegisterEditor.getRegisterValue(argument)))
              else
                // Inhalt des Registers ist keine gültige Registernummer
                result := regNoOutOfRange
            else
              // Argument von iadd ist keine gültige Registernummer
              result := regNoOutOfRange;

    mult : if isValidRegister(argument) then

RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)*RegisterEditor.getRegiste
rValue(argument))
           else
              result := regNoOutOfRange;

    cmult : RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)*argument);

    imult : if isValidRegister(argument) then
              if isValidRegister(RegisterEditor.getRegisterValue(argument)) then

RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)*RegisterEditor.getRegiste
rValue(RegisterEditor.getRegisterValue(argument)))
              else
                // Inhalt des Registers ist keine gültige Registernummer
                result := regNoOutOfRange
            else
              // Argument von imult ist keine gültige Registernummer
              result := regNoOutOfRange;

    sub :  if isValidRegister(argument) then
             if RegisterEditor.getRegisterValue(0) < RegisterEditor.getRegisterValue(argument)
then
               RegisterEditor.setRegisterValue(0,0)
             else
              RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)-
RegisterEditor.getRegisterValue(argument))
           else
             result := regNoOutOfRange;

    csub : if RegisterEditor.getRegisterValue(0) < argument then
             RegisterEditor.setRegisterValue(0,0)
           else
             RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)-argument);

    isub : if isValidRegister(argument) then
              if isValidRegister(RegisterEditor.getRegisterValue(argument)) then
               if RegisterEditor.getRegisterValue(0) <
RegisterEditor.getRegisterValue(RegisterEditor.getRegisterValue(argument)) then
                 RegisterEditor.setRegisterValue(0,0)
               else
                 RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0)-
RegisterEditor.getRegisterValue(RegisterEditor.getRegisterValue(argument)))
              else
                // Inhalt des Registers ist keine gültige Registernummer
                result := regNoOutOfRange
            else
              // Argument von isub ist keine gültige Registernummer
              result := regNoOutOfRange;

    div_ : if isValidRegister(argument) then
             if RegisterEditor.getRegisterValue(argument) > 0 then
              RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0) div
RegisterEditor.getRegisterValue(argument))
             else
               result := divThroughZero
```

```
           else
              result := regNoOutOfRange;

    cdiv : if argument > 0 then
             RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0) div
argument)
           else
             result := divThroughZero;

    idiv : if isValidRegister(argument) then
             if isValidRegister(RegisterEditor.getRegisterValue(argument)) then
               if RegisterEditor.getRegisterValue(RegisterEditor.getRegisterValue(argument))
> 0 then
                  RegisterEditor.setRegisterValue(0,RegisterEditor.getRegisterValue(0) div
RegisterEditor.getRegisterValue(RegisterEditor.getRegisterValue(argument)))
               else
                  result := divThroughZero
             else
               // Inhalt des Registers ist keine gültige Registernummer
               result := regNoOutOfRange
            else
              // Argument von idiv ist keine gültige Registernummer
              result := regNoOutOfRange;

    goto_ : if nextRow > ProgramMemo.Lines.Count then
              result := unknownRow;

    store : RegisterEditor.setRegisterValue(argument,RegisterEditor.getRegisterValue(0));
    istore :
RegisterEditor.setRegisterValue(RegisterEditor.getRegisterValue(argument),RegisterEditor.getRe
gisterValue(0));

    jzero : if nextRow > ProgramMemo.Lines.Count then
              result := unknownRow;

    error: result := unknownCommand;

    ende : result := terminated
   end
  except
    on EIntOverflow do result := accuOverflowError
  end;

  // Nächste Programmzeile markieren und zurückgeben
  if result = ok then
  begin
    if OptionenFormular.ProgrammzeilemarkierenCheckBox.Checked then
      Hauptformular.ProgramMemo.markProgramRow(nextRow);
    row := nextRow
  end
end;

function THauptformular.getActFileName : TFileName;
// liefert Namen und Pfad des aktuellen Registermaschinenprogramms
begin
  getActFileName := actFileName
end;

procedure THauptformular.forceReset;
// deaktiviert die Menüeinträge Einzelschritt, Start und Abbruch
begin
  RunButton.Enabled := false;
  ProgrammStart.Enabled := false;

  StopButton.Enabled := false;
  ProgrammStop.Enabled := false;

  StepButton.Enabled := false;
  Einzelschritt.Enabled := false
end;

function THauptFormular.getFirstProgramRow : integer;
// sucht im Quelltext die interne Nummer der ersten Programmzeile
// (Kommentare werden übersprungen)
var row : integer;
begin
  row := 0;
  while copy(ProgramMemo.Lines[row],1,2) = '//' do
```

```
    inc(row);
  getFirstProgramRow := row
end;

procedure THauptFormular.exitProgram(Sender: TObject; var canClose : boolean);
// wird beim Versuch, das Programm zu beenden, aufgerufen
begin
  if OptionenFormular.SicherheitsabfrageCheckBox.Checked then
    if MessageDlg('Möchten Sie ReSi 1.0 wirklich beenden?', mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
      canClose := true
    else
      canClose := false
end;

// ----------------------------------------------------------------------------
// Konstruktor
// ----------------------------------------------------------------------------

constructor THauptformular.Create(AOwner: TComponent);
begin
  inherited;
  actFileName := '';
  actProgramRow := 0;
  setCounter(0);

enableButtons(true,true,true,true,true,true,false,false,false,true,true,true,true,true,true,tr
ue);
  RegisterEditor.init; // schränkt die einzugebenden Zeichen auf Zahlen ein
  RegisterEditor.setMarkingOfRegisters(true)
end;

end.
```

3. UReSiOptionen.pas

```
unit UReSiOptionen;

(*********************************************)
(*  UReSiOptionen   (10.08.2004)             *)
(*                                           *)
(*  Unit des Projektes ReSi                  *)
(*  Copyright ® 2004 by Dirk von Sierakowsky *)
(*  E-Mail: dirk.v.sierakowsky@gmx.de        *)
(*- - - - - - - - - - - - - - - - - - - - - *)
(*  Ich übernehme keine Haftung für etwaige  *)
(*  Schäden, die durch diese Unit            *)
(*  verursacht werden                        *)
(*- - - - - - - - - - - - - - - - - - - - - *)
(*  Diese Unit ist FREEWARE.                 *)
(*  Alle Rechte vorbehalten                  *)
(*********************************************)

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ComCtrls, StdCtrls, ExtCtrls;

type
  TOptionenFormular = class(TForm)
    OptionenPageControl: TPageControl;
    OptionenTabSheet: TTabSheet;
    ProtokollTabSheet: TTabSheet;
    SicherheitsabfrageCheckBox: TCheckBox;
    ProtokollartRadioGroup: TRadioGroup;
    AnzahlGroupBox: TGroupBox;
    AnzRegLabel: TLabel;
    UpDown: TUpDown;
    OKButton: TButton;
    AbstandRadioGroup: TRadioGroup;
    RegistermarkierenCheckBox: TCheckBox;
    ProgrammzeilemarkierenCheckBox: TCheckBox;

    procedure OKButtonClick(Sender: TObject);
    procedure UpDownClick(Sender: TObject; Button: TUDBtnType);
    procedure AbstandRadioGroupClick(Sender: TObject);
    procedure ProtokollRadioButtonClick(Sender: TObject);
    procedure ProtokollartRadioGroupClick(Sender: TObject);
    procedure RegistermarkierenCheckBoxClick(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  OptionenFormular: TOptionenFormular;

implementation

uses UReSiProtokoll, UReSi;

{$R *.dfm}

procedure TOptionenFormular.OKButtonClick(Sender: TObject);
// schließt das Formular
begin
  Close
end;

procedure TOptionenFormular.UpDownClick(Sender: TObject; Button: TUDBtnType);
// ändert die Anzahl der protokollierten Register
begin
  AnzRegLabel.Caption := 'Anzeige der Register 0 bis ' + intToStr(UpDown.Position);
  ProtokollFormular.clearProtocol
end;

procedure TOptionenFormular.AbstandRadioGroupClick(Sender: TObject);
// führt zum Löschen des Protokollmemos
begin
  ProtokollFormular.clearProtocol
```

```
end;

procedure TOptionenFormular.ProtokollRadioButtonClick(Sender: TObject);
// führt zum Löschen des Protokollmemos
begin
  Protokollformular.clearProtocol
end;

procedure TOptionenFormular.ProtokollartRadioGroupClick(Sender: TObject);
// führt zum Löschen des Protokollmemos
begin
  Protokollformular.clearProtocol
end;

procedure TOptionenFormular.RegistermarkierenCheckBoxClick(Sender: TObject);
// schaltet das Markieren der veränderten Register ein oder aus
begin
  Hauptformular.RegisterEditor.setMarkingOfRegisters(RegistermarkierenCheckBox.Checked)
end;

end.
```

4. UReSiProtokoll.pas

```
unit UReSiProtokoll;

(*********************************************)
(*  UReSiProtokoll   (10.08.2004)            *)
(*                                           *)
(*  Unit des Projektes ReSi                  *)
(*  Copyright ® 2004 by Dirk von Sierakowsky *)
(*  E-Mail: dirk.v.sierakowsky@gmx.de        *)
(*- - - - - - - - - - - - - - - - - - - - - *)
(*  Ich übernehme keine Haftung für etwaige  *)
(*  Schäden, die durch diese Unit            *)
(*  verursacht werden                        *)
(*- - - - - - - - - - - - - - - - - - - - - *)
(*  Diese Unit ist FREEWARE.                 *)
(*  Alle Rechte vorbehalten                  *)
(*********************************************)

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls, Buttons, Printers, ComCtrls;

type
  TProtokollFormular = class(TForm)
    ProtokollMemo: TMemo;
    Panel: TPanel;
    DruckButton: TBitBtn;
    ZwischenablageButton: TBitBtn;
    SchliessenButton: TBitBtn;
    HilfeButton: TBitBtn;

    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure DruckButtonClick(Sender: TObject);
    procedure ZwischenablageButtonClick(Sender: TObject);
    procedure SchliessenButtonClick(Sender: TObject);
    procedure HilfeButtonClick(Sender: TObject);

    procedure actualizeProtocol(var programRow : integer);
    procedure clearProtocol;
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  ProtokollFormular: TProtokollFormular;

implementation

uses UReSi, UReSiOptionen;

{$R *.dfm}

// ----------------------------------------------------------------------------
// Methoden zur Ereignissteuerung
// ----------------------------------------------------------------------------

procedure TProtokollFormular.FormClose(Sender: TObject; var Action: TCloseAction);
// löscht die Selektion des Menüeintrags Protokoll
begin
  Hauptformular.Protokoll.Checked := false
end;

procedure TProtokollFormular.SchliessenButtonClick(Sender: TObject);
// löscht die Selektion des Menüeintrags Protokoll und versteckt das Fenster
begin
  Hide;
  Hauptformular.Protokoll.Checked := false
end;

procedure TProtokollFormular.DruckButtonClick(Sender: TObject);
// ruft den Druck-Dialog auf und druckt daa Protokoll-Memo
var line: integer;
```

```
    printText: textFile;
begin
  if HauptFormular.printDialog.execute then
  begin
      AssignPrn(printText);
      Rewrite(printText);
      Printer.Canvas.Font.Size := 11;
      writeln(printText,'');
      writeln(printText,'');

      // Ausgabe der Strings aus ProgramEdit
      for line := 0 to ProtokollMemo.Lines.Count-1 do
        writeln(printText,#9+ProtokollMemo.Lines[Line]);

      writeln(printText,'');
      writeln(printText,'');
      Printer.Canvas.Font.Size := 10;
      writeln(printText,#9+Hauptformular.getActFileName);
      CloseFile(printText)
  end
end;

procedure TProtokollFormular.ZwischenablageButtonClick(Sender: TObject);
// speichert den Inhalt des Protokoll-Memos in der Windows-Zwischenablage
begin
  ProtokollMemo.SelectAll;
  ProtokollMemo.CopyToClipboard
end;

procedure TProtokollFormular.HilfeButtonClick(Sender: TObject);
// ruft den Eintrag 6 des kontextsensitiven Hilfesystems auf
begin
  Application.HelpContext(6)
end;

// ----------------------------------------------------------------------------
// "Hilfsmethoden"
// ----------------------------------------------------------------------------

procedure TProtokollFormular.clearProtocol;
// löscht das Protokoll-Memo und notiert die aktuelle Registerbelegung
var firstRow : string;
    i : integer;
begin
  if OptionenFormular.ProtokollartRadioGroup.ItemIndex = 0 then
    //Reines Registerprotokoll
    firstRow := ''
  else
    //Register- und Befehlsprotokoll
    firstRow := 'Befehl' + #9;

  if OptionenFormular.AbstandRadioGroup.ItemIndex = 1 then
    //Große Abstände
    for i:=0 to OptionenFormular.UpDown.Position do
      firstRow := firstRow + 'Register ' + intToStr(i) + #9
  else
    //Kleine Abstände
    for i:=0 to OptionenFormular.UpDown.Position do
      firstRow := firstRow + 'R ' + intToStr(i) + #9;

  ProtokollFormular.ProtokollMemo.Clear;
  ProtokollFormular.ProtokollMemo.Lines.Add(firstRow)
end;

procedure TProtokollFormular.actualizeProtocol(var programRow : integer);
// hängt die aktuelle Registerbelegung ans Protokoll an
var i : integer;
    spaceString, newRow, noString : string;
begin
  spaceString := '            ';
  if OptionenFormular.ProtokollartRadioGroup.ItemIndex = 0 then
    // Reines Registerprotokoll
    newRow := ''
  else
    // Register- und Befehlsprotokoll
    newRow := #9;

  for i:=0 to OptionenFormular.UpDown.Position do
```

```
  begin
    noString := intToStr(Hauptformular.RegisterEditor.getRegisterValue(i));
    if (length(noString) > 7) or (OptionenFormular.AbstandRadioGroup.ItemIndex = 0) then
      newRow := newRow + noString + #9
    else
      newRow := newRow + noString + #9 + #9
  end;
  ProtokollMemo.Lines.Add(newRow);

  // Programmzeilen ins Protokoll aufnehmen?
  if OptionenFormular.ProtokollartRadioGroup.ItemIndex = 1 then
    // Register- und Befehlsprotokoll
    ProtokollMemo.Lines.Add(Hauptformular.ProgramMemo.Lines[programRow])
end;

end.
```

5. RMProgramEditor.pas

```
unit RMProgramEditor;

(*********************************************)
(*  TRMProgramEditor (10.08.2004)            *)
(*                                           *)
(*  Copyright © 2004 by Dirk von Sierakowsky *)
(*  E-Mail: dirk.v.sierakowsky@gmx.de        *)
(*- - - - - - - - - - - - - - - - - - - - - - *)
(*  Ich übernehme keine Haftung für etwaige  *)
(*  Schäden, die durch diese Komponente      *)
(*  verursacht werden                        *)
(*- - - - - - - - - - - - - - - - - - - - - - *)
(*  Diese Komponente ist FREEWARE.           *)
(*  Alle Rechte vorbehalten                  *)
(*********************************************)

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, StdCtrls;

type
  TRMProgramEditor = class(TMemo)
  private
    { Private-Deklarationen }
  protected
    { Protected-Deklarationen }
  public
    { Public-Deklarationen }
    function getMaxProgramRow : integer;
    procedure markProgramRow(rowNo : integer);
  published
    { Published-Deklarationen }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Beispiele', [TRMProgramEditor])
end;

function TRMProgramEditor.getMaxProgramRow : integer;
// ermittelt die Zeilennummer der letzten Programmzeile
var lastRow : string;
begin
  lastRow := Lines[Lines.Count-1];
  getMaxProgramRow := strToInt(Copy(lastRow,1,Pos(' ',lastRow)-1))
end;

procedure TRMProgramEditor.markProgramRow(rowNo : integer);
// markiert die Zeile rowNo
begin
  SetFocus;
  SelStart := Pos(Lines[rowNo],Text)-1;
  SelLength := Length(Lines[rowNo])
end;

end.
```

6. RegisterEditor.pas

```
unit RegisterEditor;

(*********************************************)
(*  TRegisterEditor (10.08.2004)             *)
(*                                           *)
(*  Copyright © 2004 by Dirk von Sierakowsky *)
(*  E-Mail: dirk.v.sierakowsky@gmx.de        *)
(*- - - - - - - - - - - - - - - - - - - - - - *)
(*  Ich übernehme keine Haftung für etwaige  *)
(*  Schäden, die durch diese Komponente      *)
(*  verursacht werden                        *)
(*- - - - - - - - - - - - - - - - - - - - - - *)
(*  Diese Komponente ist FREEWARE.           *)
(*  Alle Rechte vorbehalten                  *)
(*********************************************)

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Grids, ValEdit, DB, Graphics;

const noOfCells = 50;

type
  TRegisterEditor = class(TValueListEditor)
  private
    { Private-Deklarationen }
    memory : array[0..noOfCells] of integer; //Registerspeicher
    actRegisterNo : integer;
    markRegisters : boolean;
  protected
    { Protected-Deklarationen }
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState); override;
  public
    { Public-Deklarationen }
    procedure init;
    function getRegisterValue(regNo : integer) : integer;
    procedure setRegisterValue(regNo, value : integer);
    procedure setMarkingOfRegisters(yesNo : boolean);
    procedure clear;
    procedure storeRegisters;
    procedure recallRegisters;
    constructor Create(AOwner: TComponent); override;
  published
    { Published-Deklarationen }
  end;

procedure Register;

implementation

procedure Register;
// registriert die Komponente "TRegisterEditor" in der Komponentenpalette unter "Beispiele"
begin
  RegisterComponents('Beispiele', [TRegisterEditor]);
end;

function TRegisterEditor.getRegisterValue(regNo : integer) : integer;
// liefert den Wert des Registers regNo
begin
  getRegisterValue := strToInt(Values['R'+intToStr(regNo)])
end;

procedure TRegisterEditor.setRegisterValue(regNo, value : integer);
// setzt den Wert des Registers regNo
begin
  actRegisterNo := regNo;  // Für die Ereignisbehandlung von OnDrawCell
  Values['R'+intToStr(regNo)] := intToStr(value)
end;

procedure TRegisterEditor.clear;
// setzt alle Register sowie den Registerspeicher auf 0
var i : integer;
begin
  for i:=0 to noOfCells do
  begin
```

```
    Values['R'+intToStr(i)] := intToStr(0);
    memory[i] := 0
  end
end;

procedure TRegisterEditor.storeRegisters;
//Speichert die Register zwischen
var i : integer;
begin
  for i:=0 to noOfCells do
    memory[i] := StrToInt(Values['R'+intToStr(i)])
end;

procedure TRegisterEditor.recallRegisters;
//Speichert die Register zwischen
var i : integer;
begin
  for i:=0 to noOfCells do
    Values['R'+intToStr(i)] := intToStr(memory[i]);
end;

procedure TRegisterEditor.init;
// beschränkt die Registereinträge auf numerische Zeichen
var i : integer;
begin
  for i := 0 to noOfCells do
    ItemProps['R'+intToStr(i)].EditMask := '0999999999;0; '
end;

procedure TRegisterEditor.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
// markiert das Register actRegisterNo, sofern markRegisters gesetzt ist
begin
  inherited;
  if (ARow = actRegisterNo + 1) and (ACol = 0) and markRegisters then
    with Canvas do
    begin
      Brush.Color := clLtGray;
      TextRect(ARect,ARect.Left+2,ARect.Top+1,'R'+intToStr(actRegisterNo))
    end
end;

procedure TRegisterEditor.setMarkingOfRegisters(yesNo : boolean);
// schaltet das Markieren von veränderten Registern ein
begin
  markRegisters := yesNo
end;

constructor TRegisterEditor.Create(AOwner: TComponent);
begin
  inherited;
  markRegisters := false;
  actRegisterNo := 0;
  clear
end;

end.
```

5.2 Quelltexte der Registermaschinenprogramme

Bei der Erstellung dieser Programme wurde kein Wert auf Eleganz oder Kürze der Quelltexte gelegt:

Name des Programms	***Quelltext***

Bedingte Anweisung.rmp	`// IF R1 > R2 THEN R3:=0`
	`// ELSE R3:=1`
	`// Dirk von Sierakowsky, 08/2004`
	`//`
	`10 load 2`
	`20 sub 1`
	`30 jzero 60`
	`// R1 < R2:`
	`40 cload 1`
	`50 store 3`
	`// Aufräumen`
	`60 cload 0`
	`70 end`

Bruchaddition.rmp	`// Addition zweier Brüche ohne Kürzen`
	`// R1/R2+R3/R4`
	`// Dirk von Sierakowsky, 08/2004`
	`//`
	`10 load 1`
	`20 mult 4`
	`30 store 5`
	`40 load 2`
	`50 mult 3`
	`60 store 6`
	`70 load 5`
	`80 add 6`
	`90 store 7`
	`100 load 2`
	`110 mult 4`
	`120 store 8`
	`130 end`

Bubblesort 10.rmp	`// Bubble-Sort für die Register`
	`// R1 bis R10`
	`// Dirk von Sierakowsky, 06/2004`
	`//`
	`2 cload 9`
	`4 store 14`
	`10 cload 1`
	`20 store 11`
	`30 cadd 1`
	`40 store 12`
	`// Vergleiche benachbarte Zahlen`
	`50 iload 11`
	`60 isub 12`
	`70 jzero 140`
	`// Vertausche beide Zahlen`
	`80 iload 11`
	`90 store 13`
	`100 iload 12`
	`110 istore 11`
	`120 load 13`
	`130 istore 12`
	`// Am Ende angelangt?`
	`140 load 14`

	150 sub 11
	160 jzero 200
	// Erhöhe R11 und R11
	170 load 11
	180 cadd 1
	190 goto 20
	// Am Ende angelangt?
	200 load 14
	205 csub 1
	210 jzero 250
	230 store 14
	240 goto 10
	// Lösche Müll:
	250 cload 0
	260 store 11
	270 store 12
	280 store 13
	290 store 14
	300 end

Fakultät.rmp	// Berechnung der Fakultät
	// R1:n -> R2:n!
	// Dirk von Sierakowsky 08/2004
	//
	10 cload 1
	20 mult 1
	30 store 2
	40 load 1
	50 csub 1
	60 store 1
	70 jzero 100
	80 load 2
	90 goto 20
	100 end

ggT nach Euklid.rmp	// ggT-Berechnung nach Euklidschem
	// Algorithmus
	// R1: a R2: b -> R1: ggT(a,b)
	// Dirk von Sierakowsky, 08/2004
	//
	// Bringe die größere Zahl in R1
	10 load 2
	20 sub 1
	30 jzero 120
	// R2 > R1: Vertauschen
	40 load 2
	50 store 3
	60 load 1
	70 store 2
	80 load 3
	90 store 1
	100 cload 0
	110 store 3
	// Eigentlicher Euklidscher Algorithmus
	// R1 := R1-R2
	120 load 1
	130 sub 2
	140 store 1
	// R1 = R2 ?
	150 sub 2
	170 jzero 190
	180 goto 10
	190 load 2
	200 sub 1

	210 jzero 230 220 goto 10 // ggT gefunden 230 end

kgV.rmp	// kgV-Berechnung
	// R1: a R2: b
	// R3: kgV(a,b)
	// Dirk von Sierakowsky, 08/2004
	//
	// Teste für alle Vielfachen von R1,
	// ob sie Vielfache von R2 sind:
	10 add 1
	20 store 3
	30 div 2
	40 mult 2
	50 store 4
	60 load 3
	70 sub 4
	80 jzero 110
	90 load 3
	100 goto 10
	110 store 4
	120 end

Maximum 10.rmp	// Berechnung des Maximums
	// der Zahlen R1 bis R10
	// Ergebnis steht in R12
	// Dirk von Sierakowsky, 07/2004
	//
	10 cload 11
	20 store 11
	30 load 11
	40 csub 1
	50 store 11
	60 jzero 130
	70 iload 11
	80 sub 12
	90 jzero 30
	// R12 := R(R11))
	100 iload 11
	110 store 12
	120 goto 30
	130 end

Maximum 2.rmp	// Berechnung des Maximums
	// von R1 und R2
	// Ergebnis in R3
	// Dirk von Sierakowsky, 08/2004
	//
	20 sub 2
	30 jzero 70
	// R1 war größer
	40 load 1
	50 store 3
	60 goto 90
	// R2 war größer
	70 load 2
	80 store 3
	// Akkumulator löschen
	90 cload 0
	100 end

Modulo.rmp	// Berechnung von n mod m
	// R1: n R2: m R3: n mod m
	// Dirk von Sierakowsky, 08/2004

	// 10 load 1 20 div 2 30 mult 2 40 store 3 50 load 1 60 sub 3 70 store 3 80 cload 0 90 end

Multiplikation.rmp	// Multiplikation durch Addition
	// R1: n R2: m -> R1: n*m
	// Dirk von Sierakowsky, 08/2004
	//
	10 load 2
	20 jzero 90
	30 csub 1
	40 store 2
	50 load 1
	60 add 1
	70 store 1
	80 goto 10
	90 end

Potenzberechnung.rmp	// Potenz m^n
	// R1: m R2: n -> R3: m^n
	// Dirk von Sierakowsky, 07/2004
	//
	10 cload 1
	20 store 3
	30 load 2
	40 jzero 120
	50 load 3
	60 mult 1
	70 store 3
	80 load 2
	90 csub 1
	100 store 2
	110 goto 40
	120 end

Primzahltest.rmp	// Primzahltest
	// R1: n R0: 1/0
	// Dirk von Sierakowsky, 08/2004
	//
	// R2:= n-1
	10 load 1
	20 store 2
	// WHILE R2>1 DO...
	30 load 2
	40 csub 1
	50 store 2
	60 jzero 160
	// R2 > 1, n mod R2 = 0?
	70 load 1
	80 div 2
	90 mult 2
	100 store 3
	110 load 1
	120 sub 3
	130 jzero 150
	// n mod R2 <> 0, dec(R2)
	140 goto 30
	// n mod R2 = 0, Keine Primzahl

	150 goto 170 // Primzahl 160 cload 1 170 end

Summe 10.rmp	// Berechne die Summe von R1 bis
	// R10, Ergebnis in R13
	// Dirk von Sierakowsky, 08/2004
	//
	10 cload 1
	20 store 12
	// Addieren der nächsten Zahl
	30 iload 12
	40 add 13
	50 store 13
	// Zähler erhöhen
	60 load 12
	70 cadd 1
	80 store 12
	// Grenze erreicht?
	90 csub 10
	100 jzero 30
	// Aufräumen
	110 store 12
	120 end

Summe 2.rmp	// Berechnet die Summe R1+R2
	// und gibt das Ergebnis in R3 aus
	// Dirk von Sierakowsky, 08/2004
	//
	10 load 1
	20 add 2
	30 store 3
	40 end

Ulam.rmp	// Akzeptor für Zahlen, deren Ulam-
	// Folge den Wert 1 enthält
	// R1: Eingabe n
	// Dirk von Sierakowsky, 08/2004
	//
	10 load 1
	// R0 mod 2 = 0 ?
	20 cdiv 2
	30 cmult 2
	40 store 2
	50 load 1
	60 sub 2
	70 jzero 130
	// R2 mod 2 <> 0: n:=3n+1
	80 load 1
	90 cmult 3
	100 cadd 1
	110 store 1
	120 goto 160
	// R2 mod 2 = 0: n:=n div 2
	130 load 1
	140 cdiv 2
	150 store 1
	// n=1?
	160 csub 1
	170 jzero 190
	180 goto 10
	// Zahl als wundersam erkannt
	190 store 2
	200 end

5.3 Quelltexte des Hilfesystems (*Hilfetexte ReSi.rtf*)

idiv
#$ add **ReSi 1.0: Allgemeine Hinweise**

Der Registermaschinen-Simulator (ReSi) stellt eine Simulationsumgebung für Registermaschinen zur Verfügung und verfolgt das Ziel, die Arbeitsweise solcher Maschinen zu verdeutlichen. Das Programm wurde im Rahmen einer Prüfungsarbeit für den Lehrerweiterbildungskurs VIII des Hessischen Kultusministeriums entwickelt.

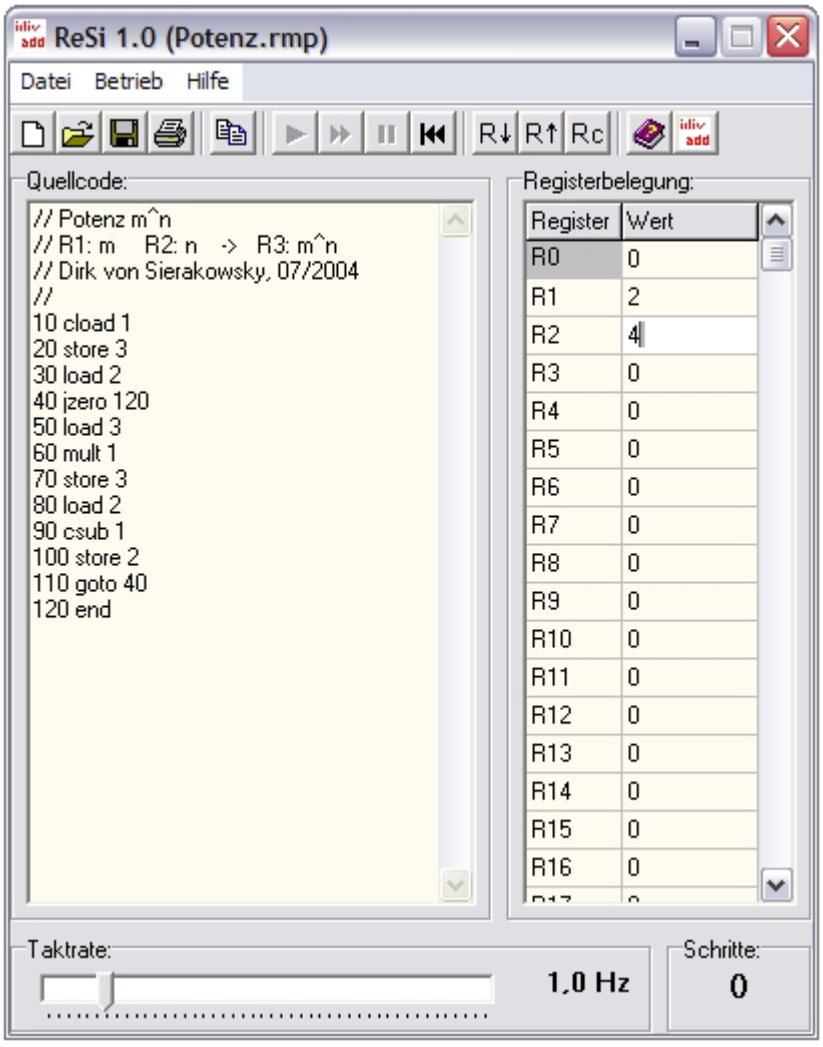


Abbildung 11: Hauptfenster des Programms

Der Funktionsumfang umfasst das Erstellen und Speichern von Registermaschinenprogrammen sowie die Visualisierung der Arbeitsweise der Maschine bei der Ausführung. Hierzu können Programme zeilenweise oder als Ganzes ausgeführt werden, wobei wiederum die Taktrate der Maschine von 0.1 bis 50 Hz gewählt werden kann.

Der Befehlssatz wurde an das Buch „Theoretische Informatik“ von Asteroth/Baier (ISBN 3-8273-7033-7) angelehnt, wobei die Syntax aus didaktischen Gründen leicht abgeändert wurde.

AllgemeineHinweise
$ Allgemeine Hinweise

idiv
#$ add **Info**

Die Benutzung des Registermaschinen-Simulators (ReSi) 1.0 im Rahmen von Forschung und Lehre ist unbegrenzt möglich und bedarf keiner weiteren Genehmigung, ebenso ist die Weitergabe der Software unbegrenzt gestattet.

Das Recht auf Veränderung bleibt davon unberührt und bedarf einer schriftlichen Genehmigung.

Ich übernehme keine Haftung für etwaige Schäden, die durch die Verwendung dieses Programms entstehen.

12. August 2004,
Dirk von Sierakowsky

Kontakt: dirk.v.sierakowsky@gmx.de

[#] Info
[$] Info

idiv
#$ add **Beschränkung der Register**

Weil sich die theoretische Vorgabe, dass eine Registermaschine beliebig viele Register besitzt, nicht in die Praxis umsetzen lässt, wurde die Anzahl der Register in dieser Simulationsumgebung auf 51 (Akkumulator + Register 1 bis 50) beschränkt. Dies sollte für den Gebrauch im Unterricht ausreichen.

Ebenfalls beschränkt werden musste der Wertebereich für die Inhalte der Register, jedes kann eine ganze Zahl aus dem Bereich von 0 bis 2147483647 aufnehmen. Negative Zahlen sind gemäß Theorie nicht möglich, Überläufe der Register werden nicht abgefangen.

BeschränkungderRegister
$ Beschränkung der Register

idiv
#$ add **Syntax der Programmzeilen**

1. Jede **Programmzeile** muss folgende Syntax besitzen:

   Zeilennummer Registermaschinenbefehl

   Die Zeilennummer muss eine natürliche Zahl sein! Beispielprogramm…

   Der Registermaschinenbefehl muss aus dem gültigen Befehlssatz stammen. Dieser gliedert sich in arithmetische Befehle und sonstige Befehle. Bei allen Befehlen ist die Beschränkung der Register zu beachten.

2. **Kommentarzeilen** besitzen die folgende Syntax:

   // Kommentar

[#] SyntaxderProgrammzeilen
[$] Syntax der Programmzeilen

idiv
#$ add **Arithmetische Befehle**

Folgende arithmetische Befehle stehen zur Programmierung der Registermaschine zur Verfügung:

1. **Addition**
   (a) **cadd n**: Addition der Konstanten n zum Akkumulator (Register 0)
   (b) **add n**: Addition des Inhalts von Register n zum Akkumulator
   (c) **iadd n**: Addition des Inhalts des Registers, dessen Nummer im Register n steht, zum Akkumulator

2. **Subtraktion**: Negative Ergebnisse führen stets zum Ergebnis 0 !!
   (a) **csub n**: Subtraktion der Konstanten n vom Akkumulatorinhalt
   (b) **sub n**: Subtraktion des Inhalts von Register n vom Akkumulatorinhalt
   (c) **isub n**: Subtraktion des Inhalts des Registers, dessen Nummer im Register n steht, vom Akkumulatorinhalt

3. **Multiplikation**:
   (a) **cmult n**: Multiplikation der Konstanten n mit dem Inhalt des Akkumulators
   (b) **mult n**: Multiplikation des Inhalts von Register n mit dem Inhalt des Akkumulators
   (c) **imult n**: Multiplikation des Inhalts des Registers, dessen Nummer im Register n steht, mit dem Inhalt des Akkumulators

4. **Division**: Das Ergebnis einer Division wird stets auf den Ganzzahlanteil reduziert !!
   (a) **cdiv n**: Division des Inhalts des Akkumulators durch die Konstante n
   (b) **div n**: Division des Inhalts des Akkumulators durch den Inhalt von Register n
   (c) **idiv n**: Division des Inhalts des Akkumulators durch den Inhalt des Registers, dessen Nummer im Register n steht

Wichtig: Jede Programmzeile (mit Ausnahme der Kommentarzeilen, s. sonstige Befehle) muss folgende Syntax besitzen:

Zeilennummer Registermaschinenbefehl

Die Zeilennummer muss eine natürliche Zahl sein! Beispielprogramm…

Bei allen Befehlen ist die Beschränkung der Register zu beachten. Weitere Befehle zur Programmierung finden sich unter sonstige Befehle.

ArithmetischeBefehle
$ Arithmetische Befehle

idiv
#$ add **Sonstige Befehle**

Neben den arithmetischen Befehlen stehen Befehle zum Laden und Speichern von Registern sowie zum Springen zur Verfügung:

1. **Laden des Akkumulators (Register 0)**
   (a) **cload n**: Laden der Konstanten n in den Akkumulator
   (b) **load n**: Laden des Inhalts von Register n in den Akkumulator
   (c) **iload n**: Laden des Inhalts des Registers, dessen Nummer im Register n steht, in den Akkumulator

2. **Speichern des Akkumulators**
   (a) **store n**: Speichern des Akkumulatorinhalts in das Register n
   (b) **istore n**: Speichern des Akkumulatorinhalts in das Register, dessen Nummer im Register n steht

3. **Sprungbefehle**
   (a) **goto n**: Unbedingter Sprung in die Programmzeile mit der Zeilennummer n
   (b) **jzero n**: Sprung in die Programmzeile mit der Zeilennummer n, sofern im Akkumulator eine 0 steht, sonst weiter zur folgenden Programmzeile

4. **Programmende**: **end** gibt das Ende des Registermaschinenprogramms an

5. **Kommentarzeilen** werden mit „//“ eingeleitet

Wichtig: Jede Programmzeile (mit Ausnahme der Kommentarzeilen) muss folgende Syntax besitzen:

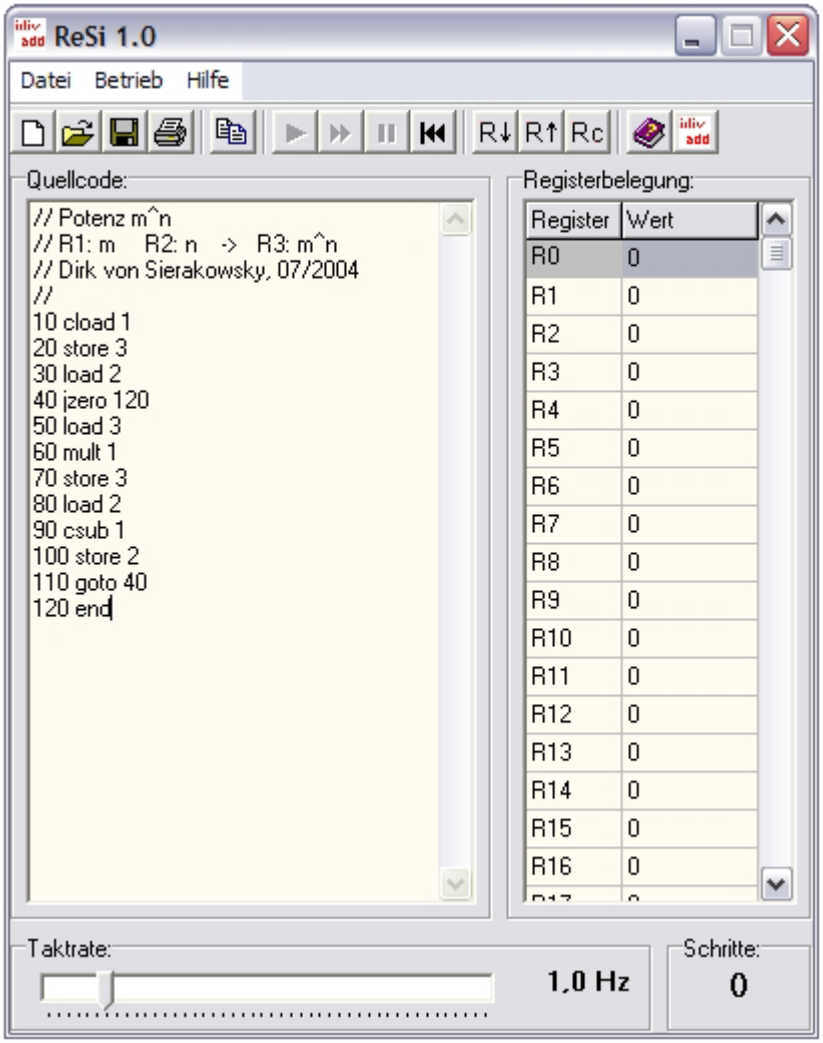
Zeilennummer Registermaschinenbefehl

Die Zeilennummer muss eine natürliche Zahl sein! Beispielprogramm…

SonstigeBefehle
$ Sonstige Befehle

#$ Beispielprogramm: Berechnung der Potenzfunktion

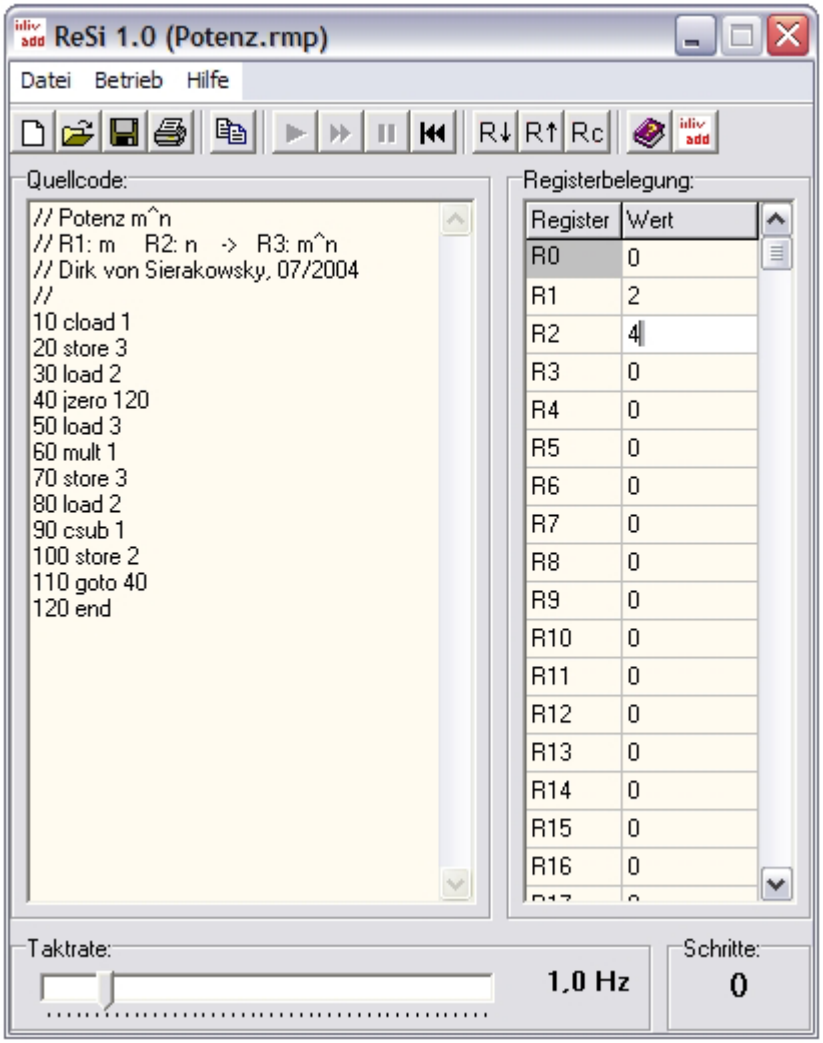
Sofort nach dem Erscheinen des Startfensters kann mit dem Programmieren der Registermaschine begonnen werden. Zunächst erstellt man zweckmäßigerweise den Quelltext im Programmeditor, wobei lediglich auf den zur Verfügung stehenden Befehlssatz sowie die Syntax der Zeilen geachtet werden muss. Die folgende Abbildung zeigt bereits das fertige Programm samt Kommentarzeilen zu Beginn:



Spätestens nach dem Erstellen sollte das Programm gespeichert werden. Dies geschieht entweder mit dem Menüeintrag *Datei -> Speichern* oder über das entsprechende Icon auf der Symbolleiste. Nach dem Speichern wird der Dateiname in der Kopfzeile des Programms angezeigt.

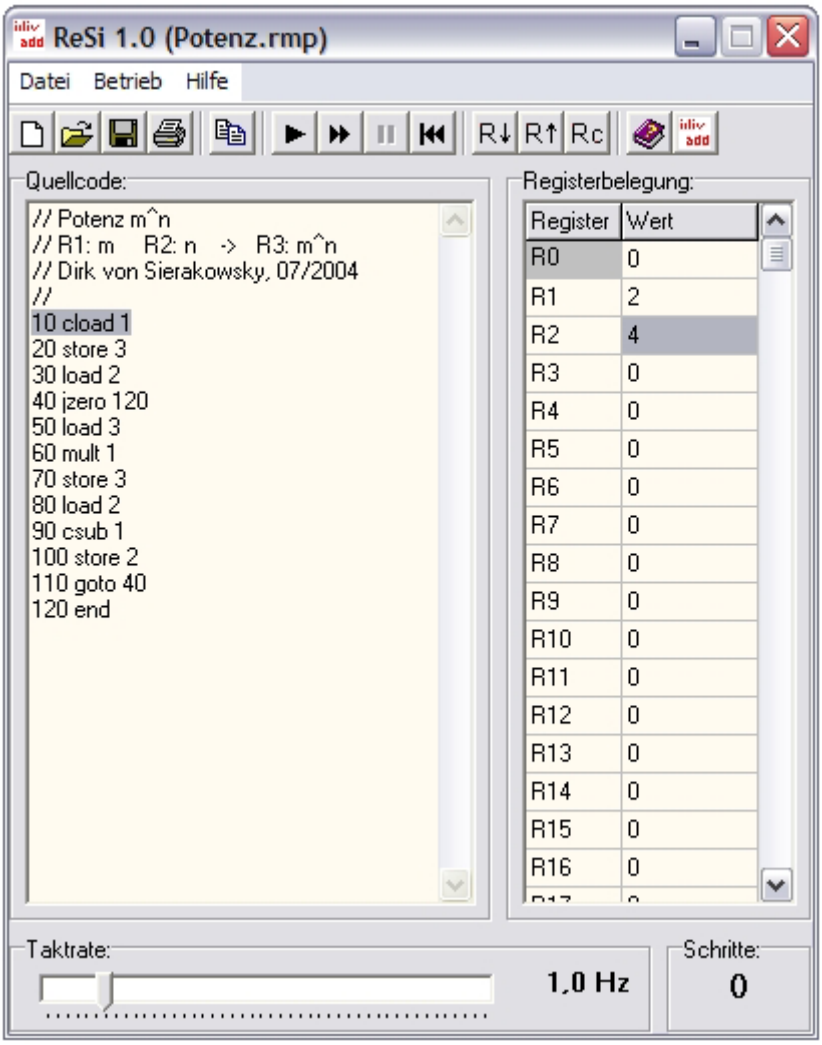
Vor dem Ausführen des Programms muss noch die Registervorbelegung eingegeben werden. Dies geschieht rechts im Editor für die Registerbelegung durch Anklicken der entsprechenden Zellen. Für das Programm „Potenz“ muss die Basis *m* im Register R1 und der Exponent *n* im Register R2 stehen. Das Ergebnis findet sich dann später in R3.

Beispielprogramm
$ Beispielprogramm



ReSi verfügt über einen Zwischenspeicher für die Registerinhalte, der genau eine Registerbelegung speichern kann. Die eingegebene Registerbelegung kann in diesem mit Hilfe des Befehls *Betrieb -> Register zwischenspeichern* gesichert werden. Bei einem erneuten Programmstart würde sie mit dem Befehl *Betrieb -> Register wiederherstellen* wieder zur Verfügung stehen. Die aktuelle Registerbelegung und der Registerspeicher lassen sich mit dem Befehl *Betrieb -> Register löschen* bequem wieder gemeinsam löschen.

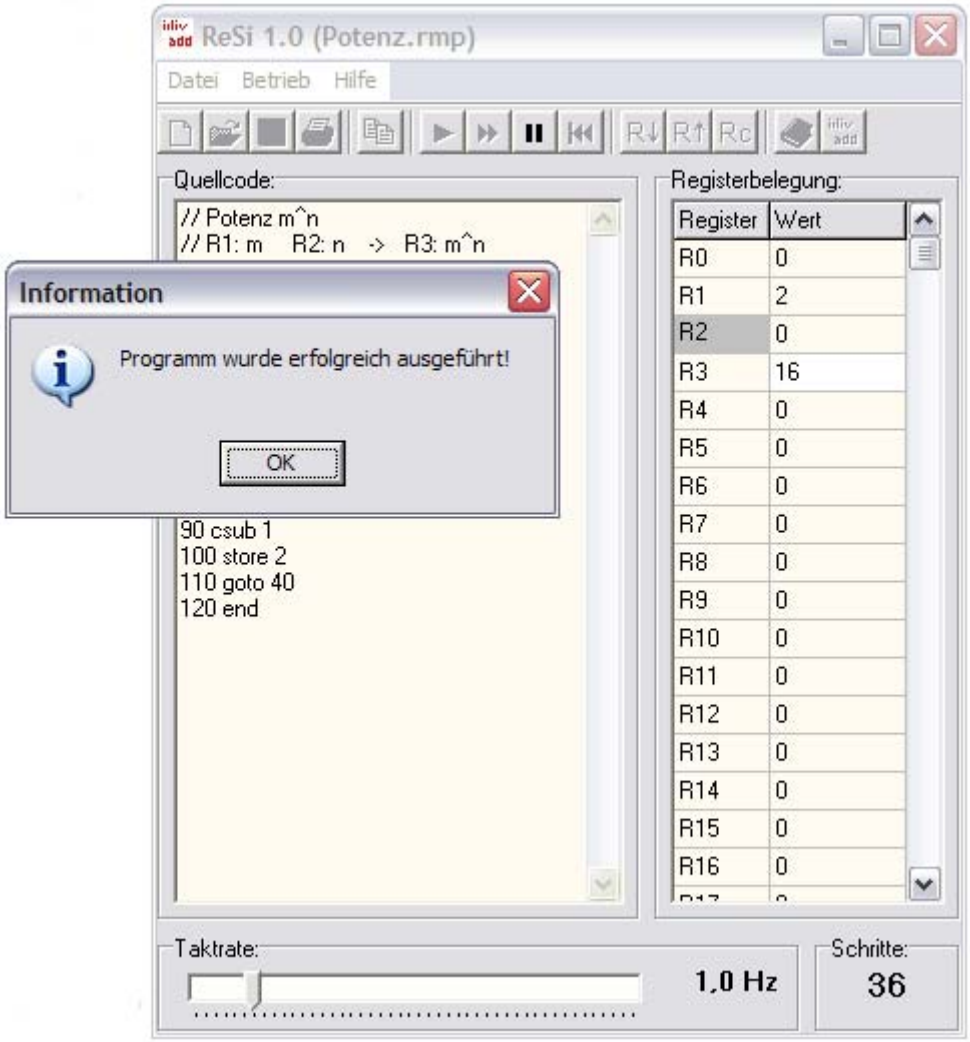
Als letzte Vorbereitung müssen vor dem Start des Programms noch der Befehls- und der Schrittzähler der Registermaschine in den Anfangszustand versetzt werden. Dies geschieht durch Anwahl des Menüpunkts *Betrieb -> Zurücksetzen*. Nun wird die erste Programmzeile im Quelltext-Editor markiert, und die Menüpunkte *Betrieb -> Einzelschritt* sowie *Betrieb -> Start* sind aktiv:



Prinzipiell stehen zwei Betriebsmodi zur Verfügung: Im Einzelschrittmodus (*Betrieb -> Einzelschritt*) kann das Programm zeilenweise abgearbeitet werden, im Ablaufmodus (*Betrieb -> Start*) wird es Befehl für Befehl automatisch abgearbeitet, bis es entweder erfolgreich terminiert ist oder durch eine Fehlermeldung bzw. manuell gestoppt wurde (*Betrieb -> Abbruch*). Im Ablaufmodus kann die Frequenz der Registermaschine mit dem logarithmischen Schieberegler für die Taktrate zwischen *0.1* und *100 Hertz* variiert werden. Zwischen beiden Modi kann während der Programmausführung beliebig gewechselt werden.

In beiden Modi werden stets die aktuelle Programmzeile sowie das zuletzt veränderte Register hervorgehoben. Dieses Markieren wurde aus didaktischer Sicht eingebaut, um es den Schülern zu ermöglichen, die Wirkungsweise der einzelnen Befehle besser nachvollziehen zu können. Unter *Betrieb -> Optionen* lässt es sich je nach Geschmack abschalten.

Hat das Programm erfolgreich terminiert, so gibt der Schrittzähler Auskunft über die Anzahl der benötigten Schritte. Das Ergebnis der Berechnung von 2^4 ist wie erwartet 16 und steht im Register R3 ablesen. Für einen erneuten Programmstart müssten, wie oben erläutert, zunächst Befehls- und Schrittzähler zurückgesetzt werden. Ein Zurücksetzen ist ebenfalls nötig, wenn während des Programmlaufs manuelle Änderungen am Quelltext oder an den Registerinhalten vorgenommen werden.


ReSi 1.0 (Potenz.rmp)
Datei Betrieb Hilfe
Quellcode:
// Potenz m^n
// R1: m R2: n -> R3: m^n
90 csub 1
100 store 2
110 goto 40
120 end
Information
Programm wurde erfolgreich ausgeführt!
OK
Registerbelegung:
Register Wert
R0 0
R1 2
R2 0
R3 16
R4 0
R5 0
R6 0
R7 0
R8 0
R9 0
R10 0
R11 0
R12 0
R13 0
R14 0
R15 0
R16 0
Taktrate:
1,0 Hz
Schritte:
36

#$ Protokollieren der Registerbelegung

ReSi 1.0 bietet die Möglichkeit, während des Ablaufs eines Programms die Registerinhalte fortlaufend zu protokollieren. Das entsprechende Protokollfenster lässt sich im Menüpunkt Betrieb einschalten und unter Optionen konfigurieren:

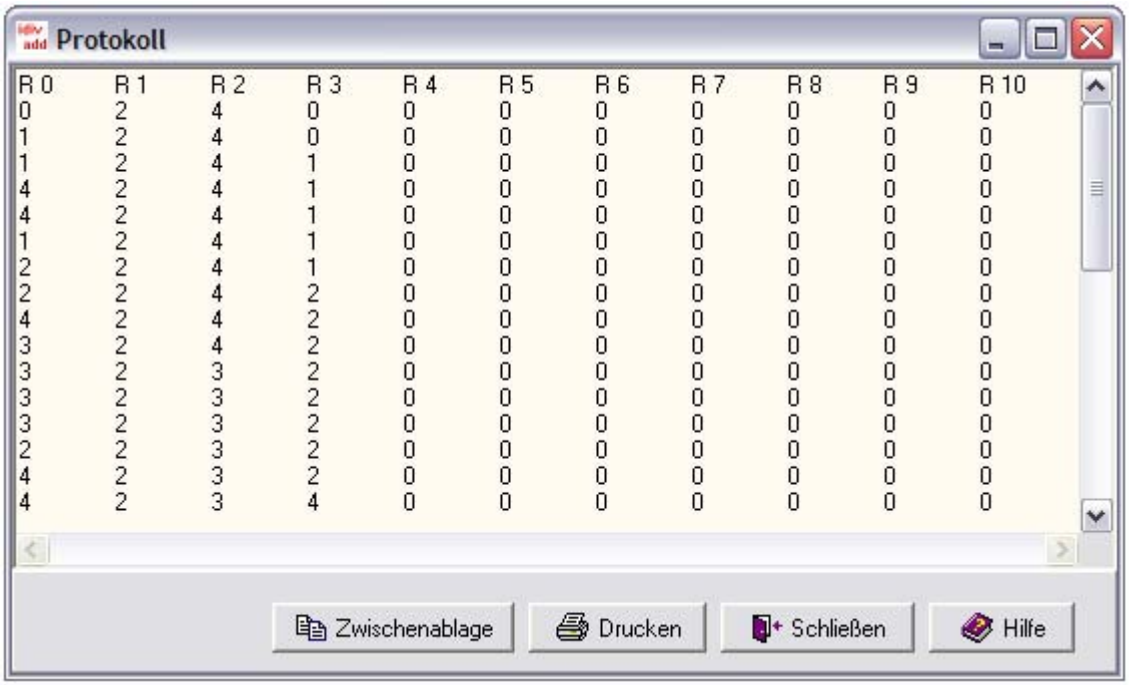


Abbildung 12: Protokollfenster

Zwischenablage: Der gesamte Inhalt des Fensters wird in die Zwischenablage übernommen.

Drucken: Der gesamte Inhalt wird gedruckt.

Protokollfenster
$ Protokollfenster

idiv
#$ add **Menüpunkt „Datei“**

Das Dateimenü enthält die folgenden Einträge:

Neu (): Ein neues Registermaschinenprogramm kann geschrieben werden

Öffnen… (): Laden eines Registermaschinenprogramms (*.rmp)

Speichern (): Speichern des aktuellen Registermaschinenprogramms

Speichern unter…: Speichern des aktuellen Registermaschinenprogramms unter Angabe des Namens

Drucken… (): Drucken des aktuellen Registermaschinenprogramms

In die Zwischenablage kopieren (): Kopiert den gesamten Quelltext in die Windows-Zwischenablage.

Beenden: Beenden von Registermaschine 1.0

[#] Datei
[$] Datei

idiv
#$ add **Menüpunkt „Betrieb“**

Dieser Menüpunkt enthält Befehle zur Steuerung des Programmablaufs:

Einzelschritt (): Die nächste Programmzeile wird abgearbeitet.

Start (): Das Programm wird von der aktuellen Position beginnend gestartet. Die Geschwindigkeit des Ablaufs kann mit Hilfe der Taktrate eingestellt werden.

Stopp (): Das Programm wird gestoppt. Es kann danach von der aktuellen Position aus erneut gestartet werden, ggf. auch im Einzelschrittmodus.

Reset (): Das Registermaschinenprogramm wird auf den ersten Befehl zurückgesetzt.

Mit Hilfe der folgenden Befehle können die Register zwischengespeichert werden:

Register zwischenspeichern (): Speichert die aktuelle Registerbelegung in einem Zwischenspeicher. Dieser kann genau eine Registerbelegung speichern.

Register wiederherstellen (): Überschreibt die aktuelle Registerbelegung mit dem Inhalt des Register-Zwischenspeichers.

Register löschen (): Löscht die aktuelle Registerbelegung sowie den Inhalt des Register-Speichers.

Die restlichen Befehle:

Protokoll anzeigen: Ermöglicht das Ein- und Ausschalten des Protokollfensters.

Optionen (): Ermöglicht Einstellungen zum Protokoll sowie zum Programm.

Karteikarte „Protokolleinstellungen“:

Betrieb
$ Betrieb

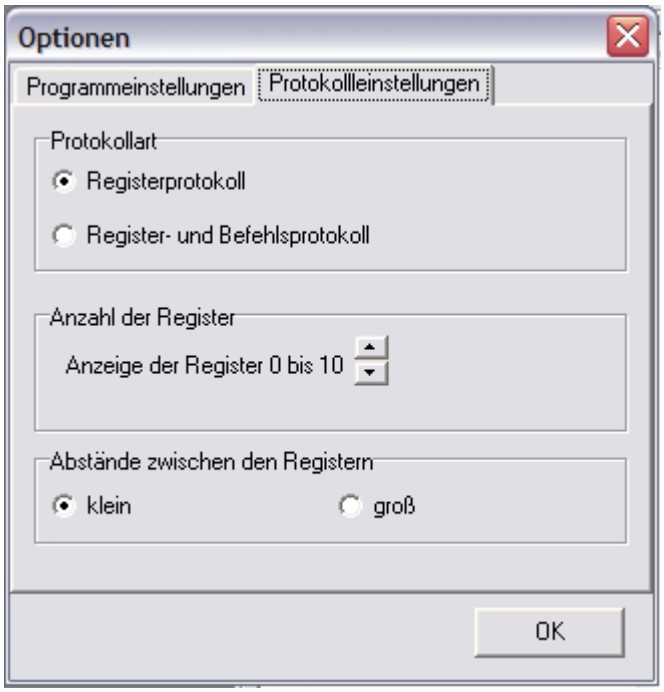


Abbildung 13: Optionenfenster

- **Protokollart**: Ein Registerprotokoll enthält nur die Registerbelegungen, ein Register- und Befehlsprotokoll enthält zusätzlich jeweils die dazugehörigen Registermaschinenbefehle.

- **Anzahl der Register**: Hier kann die Anzahl der Register eingestellt werden, die im Protokoll erscheinen. Unabhängig von dieser Protokolleinstellung arbeitet das Programm weiterhin mit allen 51 Registern.

- **Abstände zwischen den Registern**: Standardmäßig sind die Abstände klein gewählt. Sollten die Register sehr große Zahlen beinhalten, so kann es notwendig sein, die Standardeinstellung zu verlassen.

Karteikarte „Programmeinstellungen“:

- **Sicherheitsabfrage beim Beenden einblenden**: Kann je nach Geschmack ausgeschaltet werden.

- **Veränderte Register beim Programmlauf markieren**: Zusätzliche Visualisierung der Register bei Veränderung ihrer Inhalte.

- **Aktuelle Programmzeile hervorheben**: Ständige Visualisierung der aktuellen Programmzeile während der Ausführung eines Programms.

idiv
#$ add **Allgemeine Hinweise zur Fehlererkennung**

ReSi 1.0 erkennt eine Reihe von typischen Programmierfehlern zur Laufzeit der Programme:

Division durch Null
Ungültige Zeilennummer
Unbekanntes Sprungziel
Unbekannter Befehl
Ungültige Registernummer
Akkumulatorüberlauf

Alle Fehlermeldungen führen zwangsläufig zum Abbruch des Registermaschinenprogramms.

Nicht überprüft wird allerdings, ob die Zeilen eines Registermaschinenprogramms aufsteigend nummeriert sind. Der Programmierer sollte daher im Sinne gut lesbarer Quelltexte darauf achten, Zeilennummern immer aufsteigend zu vergeben.

AllgemeineHinweise
$ Allgemeine Hinweise zur Fehlererkennung

idiv
#$ add **Fehler: Division durch Null**

cdiv 0 ist ebenso verboten wie div n, wenn im Register n eine 0 steht. Bei indirekter Adressierung mittels idiv n tritt die verantwortliche Null etwas versteckt auf!

DivisiondurchNull
$ Division durch Null

idiv
#$ add **Fehler: Unbekanntes Sprungziel**

Ein jzero- oder goto-Befehl hat eine Sprungadresse, die nicht existiert!

[#] UnbekanntesSprungziel
[$] Unbekanntes Sprungziel

Fehler: Unbekannter Befehl

Ein unbekannter Befehl wurde verwendet oder die Syntax der Programmzeile ist nicht korrekt. Der gültige Befehlssatz teilt sich auf in arithmetische Befehle und sonstige Befehle.

Dieser Fehler tritt auch dann auf, wenn Leerzeichen zwischen Zeilennummer und Befehl bzw. zwischen Befehl und Argument vergessen wurden!

[#] UnbekannterBefehl
[$] Unbekannter Befehl

idiv
#$ add **Fehler: Ungültige Registernummer**

Eine Registernummer muss im Bereich von 0 bis 50 liegen!

Vorsicht: Bei indirekter Adressierung (iload, istore, iadd etc.) tritt dieser Fehler versteckt auf, die Registernummer steht selbst in einem Register!

[#] UngültigeRegisternummer
[$] Ungültige Rergisternummer

idiv
#$ add **Fehler: Ungültige Zeilennummer**

Zeilennummern dürfen nur natürliche Zahlen sein!

[#] UngültigeZeilennummer
[$] Ungültige Zeilennummer

idiv
#$ add **Fehler: Akkumulatorüberlauf**

Bei der aktuellen Berechnung würde der Akkumulator überlaufen, d.h. das Ergebnis der Rechnung wäre größer als 2147483647 und würde damit gegen die Beschränkung des Wertebereichs der Register verstoßen. Die Rechnung konnte daher nicht ausgeführt werden!

[#] Akkumulatorüberlauf
[$] Akkumulatorüberlauf

5.4 Literatur

[L1] A. Asteroth, C. Baier. Theoretische Informatik. Kösel-Verlag. Kempen 2002.

[L2] G. Battenfeld, Dr. J. Poloczek u.a. Theoretische Informatik. Planung eines Kurses in der Jahrgangsstufe 11. 1996
http://www.bildung.hessen.de/abereich/inform/skii/lk/theorie.pdf

[L3] N. Breier. Ein Plädoyer für die Registermaschine. in: LOG IN 9 (1989) Heft 1.

[L4] E. Kindler, S. Manthey. Automaten, Formale Sprachen und Berechenbarkeit I. Skript zur Vorlesung im WS 2001/2002 an der TU München.
http://wwwbrauer.informatik.tu-muenchen.de/lehre/fospr/WS0102/Skript/SkriptIII.pdf

[L5] S. Zimmer. Berechenbarkeit. Skript zur Vorlesung Informatik I an der Uni-Stuttgart. 2003
http://www.informatik.uni-stuttgart.de/ipvr/sgs/lehre/vorlesungen/info1_autip_ws03/V10_Berechenbarkeit/berechenbarkeit_p_2.pdf

[L6] H.-P. Gumm u.a. Einführung in die Informatik. Oldenbourg. 2004.

[L7] Hessisches Kultusministerium. Lehrplan Informatik Gymnasialer Bildungsgang.
http://lernarchiv.bildung.hessen.de/archiv/lehrplaene/gymnasium/informatik

[L8] W. Schneeweiss. Technische Informatik I. Kurseinheit 3. Skript der FernUniversität – Gesamthochschule Hagen. 2002.

5.5 Software

[S1] Dr. J. Poloczek, G. Röhner. Simulation endlicher Automaten. Frankfurt und Darmstadt 2004.
http://www.bildung.hessen.de/abereich/inform/skii/lk/Automaten.zip

[S2] I. Hook. Simulation einer Registermaschine per Java-Applet.
http://www.visualistik.de/java/belegaufgaben/register.html

[S3] H.-J. Philippi. 99 Steps to WinHelp.
http://members.aol.com/hjphilippi/99steps.htm

5.6 Compact Disc

Ordner	***Inhalt***
hilfedateien	Dateien, die zur Erstellung des Hilfesystems mit dem Microsoft Help Workshop benötigt wurden
icons	Bitmaps, die für die Symbolleisten verwendet wurden
quelltexte	Delphi-Quelltexte der Units *UResi*, *UReSiOptionen* und *UReSiProtokoll*
komponenten	Delphi-Quelltexte der Komponenten-Units *RegisterEditor* und *RMProgramEditor*
programme	Registermaschinenprogramme