Vorwort

„Aha, das große Spiel der Handsimulation. Spielen wir jetzt so etwas wie Pacman oder Ego Shooter? Das alles schon im ersten Semester? Das geht ja ratz fatz hier!“ Na ja, ganz so ist es nicht. Die Syntax einer Programmiersprache, also all die Buchstaben, Ziffern und sonstigen sonderbaren Zeichen, aus denen ein Programm besteht, sind ein wichtiger Teil eines Programms, auf den sich auch viele Studenten im Laufe des Kurses konzentrieren. Die Syntax einer Programmiersprache ist, ohne jeden Zweifel, sehr wichtig. Sie ist aber *nur* ein Teil des Ganzen. Ebenso wichtig ist die Art und Weise, wie das Programm von der CPU (also dem Rechner) ausgeführt wird, was man auch Berechnungsmodell nennt. Wir müssen also auch wissen, *wie* ein Programm ausgeführt wird.

Eine Möglichkeit, sich das „wie“ eines Programms zu erschließen, besteht im Durchführen von Handsimulationen. Derartige Handsimulationen waren in den alten Zeiten von Lochkarten und Magnetbändern normal und die effizienteste Methode, das Verhalten eines Programms herauszubekommen. In der heutigen Zeit sind Handsimulationen insbesondere bei Studenten sehr verpönt. Die Antwort lautet dann häufig: „Handsimulation ist blöd, ich kann doch das Programm direkt eintippen und testen!“ *„Ja, stimmt doch auch! Heutzutage ist Eintippen einfach und schnell, und ich sehe dann doch sofort, ob das Programm das Richtige macht oder nicht. Den alten Krimskrams braucht heute niemand mehr! Wir sind im digitalen und nicht analogen Zeitalter, aber (ältere) Dozenten verstehen das halt nicht mehr so richtig...“*

Diese Ausreden sind leider so gar nicht richtig. *„Wieso, ist doch so! Einfach Eintippen und schon ist alles klar!“* Damit kann man maximal feststellen, ob ein Programm beim Testen das richtige Resultat produziert hat. Aber das hat nicht damit zu tun, dass man weiss, *wie* ein Programm funktioniert. Vom Autofahren lernt man auch nicht, wie ein Getriebe funktioniert. *„Doch, beim Schalten werden je nach Gang verschiedene Zahnräder hin und her geschoben.“* Eben, sag ich doch, man lernt es nicht, denn es sind nämlich immer alle Zahnräder miteinander verbunden. *„Ja klar und der Kürbis ist eine Beerenfrucht.“* Na bitte, geht doch.

Nun aber mal wieder zuück zu unseren Handsimulationen. Programmieren lernt man nur, wenn man auch weiß, wie ein Programm arbeitet und was es macht. Hierfür ist es unablässig, dass man in der Lage ist, eine Handsimulation durchführen kann. Das ist auch nicht peinlich oder so. Selbst die Betreuer führen bei euren Fragen eine kleine Handsimulation

durch, um herauszufinden, was an euren Programmen jeweils falsch läuft. Nur machen sie dies aufgrund ihrer Erfahrung eben mal schnell im Kopf. Aber sie machen es.

Dieses kleine Büchlein ist dem Thema Handsimulation gewidmet. Am Anfang legen wir uns auf einen sehr einfachen Prozessor fest und klären, was dieser so abarbeiten kann. Danach zeigen wir anhand einiger Beispiele, wie die einfachen Sprachelemente von diesem Prozessor abgearbeitet werden. Dass heißt, wir führen entsprechende Handsimulationen durch und erläutern die einzelnen Schritte. Der Rest dieses Büchleins ist dann mit begleitenden Übungen versehen.

„Das klingt wieder nach Arbeit, viel Arbeit, obwohl das Wort Spiel so vollmundig im Titel steht.“ Ja, das hat auch seine Berechtigung. Wir *spielen* die Handsimulation. Dazu benötigen wir jeweils fünf Akteure, die die Funktionen der einzelnen Komponenten eines Rechners übernehmen. Diese Akteure spielen dann die einzelnen Anweisungen, was manchmal viel Spaß macht, zumindest nach *einem* Bierchen.

Kapitel 1

Das Simulationsspiel: Einführung

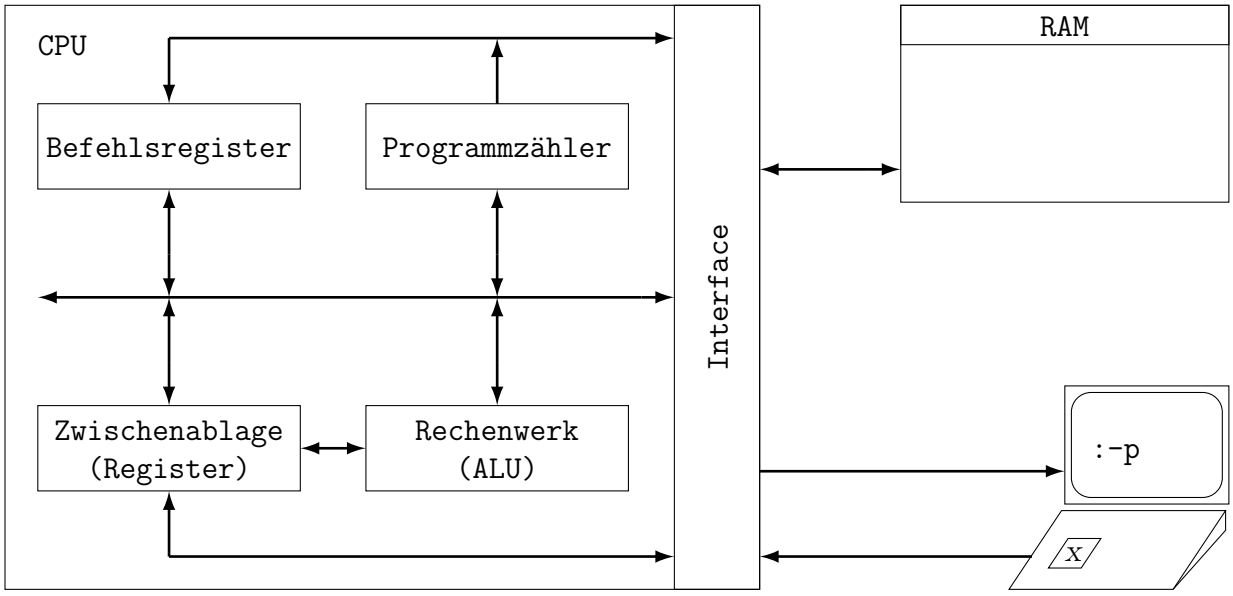
Um unser Simulationsspiel richtig spielen zu können, müssen wir zunächst zwei Dinge klären: Auf was für einer Hardware laufen unsere Programme und wie sehen die Regeln aus? Diese beiden Fragen werden wir in den ersten beiden Abschnitten klären. Danach werden wir im dritten Abschnitt ein paar einfache Anweisungen beispielhaft durch spielen, damit jeder eine konkrete Vorstellen davon bekommt, was wir von ihm erwarten.

1.1 Die verwendete Hardware-Konfiguration

In den ersten Kapiteln des Skriptes haben wir den Aufbau eines Rechners und des eigentlichen Rechenkerns (CPU und ALU) auf unterschiedlichen Abstraktionsebenen erklärt. Für unser Simulationsspiel sind zunächst einmal nur die folgenden Komponenten relevant: Zentrale (CPU), Rechenwerk (ALU) Zwischenablage (Register), Arbeitsspeicher (RAM), Eingabekanal (Tastatur), und Ausgabekanal (Bildschirm); die übrigen Komponenten wie Befehlsregister (IR), Programmzähler (PC), Grafikkarte, USB-Schnittstellen, Magnetbänder, Netzteil, Netzwerkinterface etc. sind (zunächst) einmal nebensächlich. Das für uns relevante Zusammenwirken haben wir in der folgenden Grafik (etwas vereinfacht) der folgenden Seite illustriert.

Die einzelnen Komponenten haben folgende Eigenschaften:

Zentrale (CPU): Die Zentrale ist, wie der Name schon sagt, dasjenige Element, das alle Abläufe steuert, in dem es die Tätigkeiten aller Komponenten koordiniert. Mit anderen Worten: hier laufen alle Fäden zusammen. In ihr (durch sie) wird jedes einzelne Programm abgearbeitet. Dazu holt sie sich Befehl für Befehl aus dem Arbeitsspeicher. Jeder neu geladene Befehl wird dekodiert und anschließend zur Ausführung gebracht (abgearbeitet), nach dem möglicherweise benötigte Parameter aus dem Arbeitsspeicher nachgeladen wurden. Für diese Tätigkeit besitzt die CPU diverse Hilfsmittel wie einen Programmzähler, ein Befehlsregister, ein Mikroprogramm etc., die wir hier aber



nicht weiter betrachten. Für uns ist relevant, dass die CPU jeden einzelnen Befehl aus dem Arbeitsspeicher holt und zur Ausführung bringt.

Rechenwerk (ALU): Das Rechenwerk (ALU, Arithmetic Logic Unit) ist eine hochspezialisierte Komponente, die Teil der Zentrale ist und alle anfallenden Rechenoperationen ausführt. Unsere ALU kann in jedem Fall die arithmetischen Grundrechenarten `plus`, `minus`, `mal`, `geteilt` und `modulo` sowie die logischen Verknüpfungen `und`, `oder` und `nicht`. Die ALU muss jeweils wissen, welche Operation sie ausführen soll und wo sie die jeweils benötigen Operanden her bekommt.

Zwischenablage (Register): Die Zwischenablage dient dem Datenaustausch *zwischen* den einzelnen Komponenten. Entsprechend besteht sie aus einer kleinen Zahl von Speicherelementen, die üblicherweise Register genannt und daher von uns mit `R1`, `R2`, ..., `Rn` bezeichnet werden. In dieser Zwischenablage können einzelne Werte, Variableninhalte, Konstanten, Adressen und dergleichen *kurzfristig* zwischengespeichert werden. In den Registern werden beispielsweise die Parameter (Operanden) einer arithmetischen Operation der ALU zur Verfügung gestellt, die ihrerseits das Ergebnis in einer der Register ablegt.

Arbeitsspeicher (RAM): Für das dauerhafte Speichern des eigentlichen Programms (also der einzelnen übersetzten Anweisungen und Ausdrücke) sowie der anfallenden Daten ist der Arbeitsspeicher zuständig, der oft auch als RAM bezeichnet wird. Der Arbeitsspeicher besteht aus einer sehr großen Zahl von Speicherplätzen, die beliebige Werte und alle Anweisungen aufnehmen können. Jede einzelne Speicherstelle besteht üblicherweise aus ein bis acht Bytes und hat eine eindeutige, ihr spezifisch zugewiesene Adresse. Heutzutage bestehen Adressen üblicherweise aus vier oder acht Bytes. Wichtig ist unter anderem, dass man zu einem gegebenen Zeitpunkt immer

nur auf *eine einzige* Speicherstelle zugreifen und *entweder* nur lesen *oder* nur schreiben kann. Eine Speicherstelle kann somit entweder von der CPU ausgelesen oder mit einem neuen Wert beschrieben werden; das Kopieren einzelner Werte innerhalb des Arbeitsspeichers ist somit nicht möglich.

Eingabekanal: Über den Eingabekanal kann unser Programm, das von der CPU abgearbeitet wird, mit neuen Werten versorgt werden. In unseren Simulationen ist der Eingabekanal immer die Tastatur. Bei den Eingabewerten kann es sich um Zahlen oder einzelne Zeichen handeln, je nach dem wie es die CPU möchte. Wie bei der Kommunikation mit der ALU müssen die eingehenden Werte zunächst in einem der verfügbaren Register abgelegt werden.

Ausgabekanal: Der Ausgabekanal ist bei uns einfach der Bildschirm, über den wir mit dem Nutzer kommunizieren. Auch hier können wir entweder Zahlen oder Zeichen und damit auch Texte darstellen. Wiederum ist es so, dass die Ausgaben immer zuerst über eines der Register in der CPU zwischengelagert werden müssen.

Zusammenfassung: Die von uns betrachtete Hardware besteht aus einer CPU, die neben einem Befehlszähler und einem Befehlsregister noch eine ALU (Rechenwerk) sowie eine Zwischenablage besitzt. Weiterhin ist die CPU mit einem Arbeitsspeicher (RAM) sowie einem Ein- und einem Ausgabekanal verbunden. Die einzelnen Komponenten können ausschließlich mit der Zwischenablage kommunizieren, wobei die CPU die Kontrolle innehat.

1.2 Die Regeln

Alle Handsimulationen werden gemäß folgender Regeln abgearbeitet:

Regel 1: Wir benötigen fünf Mitspieler, je einen für die CPU, die ALU, den Arbeitsspeicher sowie den Ein- und den Ausgabekanal. Stehen nur vier Mitspieler zur Verfügung, kann einer sowohl den Ein- als auch den Ausgabekanal bedienen. Steht ein sechster Mitspieler zur Verfügung, übernimmt er die Zwischenablage.

Regel 2: Die CPU ist der Boss. Sie koordiniert alle Aktivitäten. Sie bestimmt, wer, wann, was zu erledigen hat. Unter anderem sorgt sie dafür, dass sich alle benötigen Parameter in der Zwischenablage befinden und dass die hereinkommenden Ergebnisse in den richtigen Registern abgelegt werden.

Regel 3: Alle Komponenten kommunizieren miteinander einzig und alleine über die Zwischenablage.

Regel 4: Der Arbeitsspeicher kann immer nur einen gegebenen Wert an einer Adresse abspeichern (schreiben) oder den Inhalt einer angewählten Adresse bereitstellen (lesen), aber niemals beides gleichzeitig.

Regel 5: Auf Anforderung reserviert der Arbeitsspeicher den benötigten Speicherplatz für die verwendeten Variablen.

Regel 6: Wir betrachten immer Ausdrücke oder Anweisungen, die sich vollständig im Befehlsregister der CPU befinden. Entsprechend verweist der Programmzähler auf den nächsten Ausdruck bzw. die nächste Anweisung des Programms.

Regel 7: In jedem Register der Zwischenablage können sich „beliebige“ Werte befinden. Hierzu zählen unter anderem Zahlen, Zeichen und Adressen. Die Zahl der verfügbaren Register wird in jeder Aufgabe definiert.

Regel 8: Alle Register und Speicherplätze des Arbeitsspeichers haben bei Programmstart undefinierte Werte, was wir mittels Fragezeichen kennzeichnen.

Anmerkungen: Die beiden Regeln 5 und 6 basieren auf Annahmen, die nicht gerade realitätsnah sind. Die konkreten Verhältnisse sind *wesentlich* komplizierter. Doch helfen diese Vereinfachungen dem Programmieranfänger dabei, sich mit der Programmiersprache C vertraut zu machen.

Bei der Illustration der Handsimulation richten wir uns nach folgenden Richtlinien:

1. Jeder Simulationsschritt beginnt mit dem Aufschreiben des betrachteten Ausdrucks bzw. der betrachteten Anweisung.
2. Sollte einem Register oder einer Variablen ein Wert zugewiesen werden, notieren wir dies nochmals explizit am rechten Rand.
3. Am Ende eines Simulationsschritts fassen wir die aktuelle Speicherplatzbelegung einschließlich der Zwischenablage nochmals zusammen.

1.3 Einige illustrierende Beispiele

Die Regel des letzten Abschnitts mögen ein wenig abstrakt und unhandlich anmuten. Daher werden wir in diesem Abschnitt obige Regeln anhand einiger ausgewählter *Beispiele* anwenden. Bei allen Beispielen nehmen wir an, dass wir genau zwei Register in der Zwischenablage benötigen.

Variablendefinition: Nehmen wir an, wir brauchen zwei Variablen vom Typ `int`. Dann hätten wir vielleicht folgende Variablendefinition: `int i, j`. Die Handsimulation obiger Regel würde wie folgt aussehen:

```
5: int i, j
CPU an RAM: Speicher für die Variablen i und j vom Typ int anlegen
```

R1 = ?	R2 = ?	i = ?	j = ?

Ausgabe von Texten: Für Ausgaben ist in der Programmiersprache C die `printf()`-Anweisung zuständig, die wir wie folgt simulieren:

```
 6: printf( "Bitte einen Wert fuer i eingeben: " )
    CPU an OUT: Drucke den Text: Bitte einen Wert fuer i
                eingeben:
```

Eingabe: Als nächstes betrachten wir die Eingabe von Werten, die naturgemäß ein bisschen komplizierter ist. Wir gehen davon aus, dass Nutzer eine `7` eintippen wird. Die zuständige `scanf()`-Anweisung können wir wie folgt simulieren:

```
 7: scanf( "%d", & i )
    CPU an IN : Lies eine int Zahl, Eingabe in R1 ablegen
    IN  an CPU: Eine 7 liegt in R1                          R1 =  7
    CPU an RAM: Speichere den Wert 7 aus R1 in der Variablen i      i =  7
   ---------------------------------------------------------------------
    R1 = 7      R2 = ?      i = 7      j = ?
   ---------------------------------------------------------------------
```

Wertzuweisung: Eine Zuweisung wie `j = -3` sieht wie folgt aus:

```
 8: j = -3
    CPU       : Die Konstante -3 in R2 ablegen              R2 = -3
    CPU an RAM: Speichere den Wert -3 aus R2 in der Variablen j     j = -3
   ---------------------------------------------------------------------
    R1 = 7     R2 = -3     i = 7     j = -3
   ---------------------------------------------------------------------
```

Ausdrücke: Nun brauchen wir noch die Berechnung eines arithmetischen oder logischen Ausdruckes der Form `j = i * j`. Aufgrund der bisherigen Anweisungen befindet sich die Werte der Variablen `i` und `j` bereits in den Registern `R1` und `R2`:

```
 9: j = i * j
    CPU an ALU: Multiplikation von R1 (7) und R2 (-3), Ergebnis in R2
                ablegen
    ALU an CPU: Das Ergebnis ist -21 und liegt in R2          R2 = -21
    CPU an RAM: Speichere den Wert -21 aus R2 in der Variablen j    j = -21
   ---------------------------------------------------------------------
    R1 = 7     R2 = -21     i = 7     j = -21
   ---------------------------------------------------------------------
```

Ausgabe von Werten: Die Ausgabe von Werten ist eigentlich nichts Besonderes, da dies in C auch durch die `printf()`-Anweisung bewerkstelligt wird. Nur müssen wir noch für die Ersetzung der Formatierungen durch die aktuellen Werte sorgen:

```
10: printf( "j=%d\n", j )
    CPU       : Zunächst muss ich %d durch den Wert von j ersetzen,
                der sich in R3 befindet
    CPU an OUT: Drucke den Text: j=-21
```

Programmende: Wenn wir das Ende der `main()`-Funktion erreicht haben, ist auch das Programm zu Ende:

```
11: Ende des Hauptprogramms
    CPU        : Das war's für mich, ich mach' jetzt Pause :-)
```

Ohne dass es vielleicht allen Lesern bewusst geworden ist, haben wir gerade folgendes Programm einer Handsimulation unterzogen:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5         int i, j;
 6         printf( "Bitte einen Wert fuer i eingeben: " );
 7         scanf( "%d", & i );
 8         j = -3;
 9         j = i * j;
10         printf( "j=%d\n", j );
11         return 0;
12     }
```

Somit haben wir alle Basisoperationen zusammen. Später müssen wir teilweise noch kleinere Modifikationen bzw. Erweiterungen vornehmen. Aber zur Erinnerung: Es geht nicht um eine originalgetreue Simulation der tatsächlichen Vorgänge sondern um ein vereinfachtes Spiel, um sich mit den einzelnen C-Anweisungen vertraut zu machen.

1.4 Übung: ein bildlicher Vergleich

Bevor wir jetzt mit den Übungen anfangen, wollen wir noch ein wenig bei unserem Rechnermodells verweilen. Damit wollen wir euer Verständnis der Funktionsweise der einzelnen Komponenten stärken. Der folgende bildliche Vergleich ist nicht tierisch ernst gemeint, sondern erfordert von allen Beteiligten ein kleines Augenzwinkern ;-)

Krankenhaus: Als Bildnis stellen wir uns einen einfachen Krankenhausbetrieb vor. Wir betrachten die folgenden „Akteure“: Chirurg, Notaufnahme, Patientenentlassung, Krankenstation, Registratur und leitende Oberschwester. Ordne nun diese sechs Akteure den von uns betrachteten sechs Komponenten eines Rechners zu. Erläutere die Gemeinsamkeiten und begründe deine Zuordnung.

1. Komponente: CPU

   Krankenhaus:

   Begründung :

2. Komponente: Eingabekanal

   Krankenhaus:

   Begründung :

3. Komponente: Ausgabekanal

   Krankenhaus:

   Begründung :

4. Komponente: ALU

   Krankenhaus:

   Begründung :

5. Komponente : Zwischenablage

   Krankenhaus:

   Begründung :

6. Komponente : Arbeitsspeicher

   Krankenhaus:

   Begründung :

Zwei weitere Bildnisse: Wenn man sich die Funktionsweise und das Zusammenspiel der einzelnen Komponenten eines Rechners genauer anschaut, findet man im täglichen Leben viele ähnliche Gleichnisse. Suche dir eine oder zwei Organisationen bzw. Unternehmen des täglichen Lebens, in denen sich vergleichbare Arbeitsabläufe bzw. Organisationsstrukturen befinden und ordne sie den Komponenten eines Rechners zu.

Rechner	Organisation:	Unternehmen: .....
CPU	.....	.....
Eingabekanal	.....	.....
Ausgabekanal	.....	.....
ALU	.....	.....
Zwischenablage	.....	.....
Arbeitsspeicher	.....	.....

Kapitel 2

Einige grundlegende Simulationen

Nun geht's endlich los. Damit sich aber niemand von euch abgeschreckt fühlt, fangen wir mit ganz einfachen Simulationen an. Da wir keine Programm sondern nur Einzeiler betrachten, schreiben wir die entsprechenden C-Anweisungen direkt in die Simulationen. Obwohl wir die prinzipielle Herangehensweise bereits in Abschnitt 1.3 besprochen haben, wird jeweils die erste Aufgabe in Form eines kleinen Tutorials besprochen. So, dann mal viel Spaß bei den ersten Handsimulationen :-) !

2.1 Variablendefinitionen

Motivation und Ziele: Zunächst kümmern wir uns um die Definition von Variablen, ohne die man nicht viel machen kann. Bisher haben wir nur den Datentyp `int` für ganzzahlige Werte, was aber nichts weiter macht. Jeder der folgenden Variablendefinitionen ist mit einem Speicherbildchen abzuschließen, wobei die Zwischenablage aus genau drei Registern bestehen soll.

Frage: Welche Werte haben Variablen nach ihrer Definition?

Aufgabe: Simuliere die folgenden Variablendefinition: `int i, j, k, l, m;`

```
1: int i, j, k, l, m;
```

`CPU an RAM:` Speicher für die Variablen `i, j, k, l` und `m` vom Typ `int` anlegen

R1=?	R2=?	R3=?	i=?	j=?	k=?	l=?	m=?

Aufgabe: Simuliere die folgende Variablendefinition: `int Seite_A, Seite_B, Umfang;`

Aufgabe: Simuliere die folgende Programmzeile: `int a; int b; int c, d; int e;`

2.2 Zuweisungen, Berechnungen und Ausdrücke

Motivation und Ziele: In diesem Abschnitt wollen wir unseren Variablen konstante Werte sowie ganze Ausdrücke zuweisen. Hierzu müssen wir einerseits mit den Konstanten richtig umgehen und andererseits die Zugriffe auf den Arbeitsspeicher richtig ausführen. Alle Beispiele dieses Abschnitts greifen auf die Variablen `i, j` und `k` vom Typ `int` zu. Zu Beginn jeder Aufgabe besitzen diese Variablen undefinierte Werte. Unsere Zwischenablage besteht wieder aus drei Registern, sodass wir jeweils mit folgendem Speicherbildchen beginnen:

R1 = ?	R2 = ?	R3 = ?	i = ?	j = ?	k = ?

Aufgabe: Simuliere die Zuweisung `i = 13;`

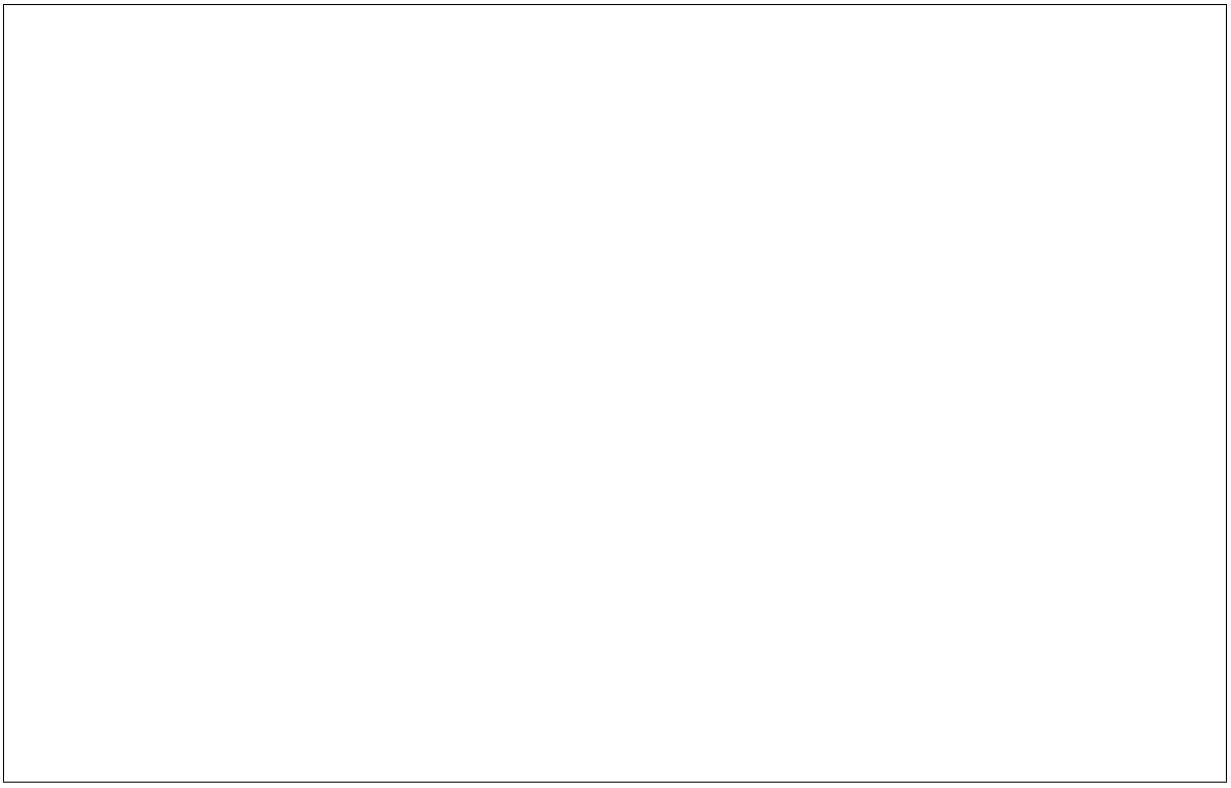
```
1: i = 13;
   CPU        : Zuerst muss ich die 13 in eines der Register bringen, um sie
                anschließend in den Arbeitsspeicher zu schreiben
   CPU        : Die Konstante 13 in R1 ablegen                          R1 = 13
   CPU an RAM: Speichere den Wert 13 aus R1 in der Variablen i          i  = 13
```

R1 = 13	R2 = ?	R3 = ?	i = 13	j = ?	k = ?

Aufgabe: Simuliere die Zuweisung `j = 9 * 7;`

Aufgabe: Simuliere die Zuweisung `k = 19 - i;`

Aufgabe: Simuliere die Zuweisung `i = -3 * k + j;`



2.3 Ausgaben

Motivation und Ziele: Da ein Programm ab und an auch etwas ausgeben sollte, beschäftigen wir uns in diesem Abschnitt mit der Simulation von Ausgabeanweisungen. Da wir keine Variablen verändern, verzichten wir diesmal auf das Darstellen von Speicherbildchen. Für das Bereitstellen von Zwischenergebnissen steht uns wiederum eine Zwischenablage mit drei Registern zur Verfügung.

Aufgabe: Simuliere die Ausgabe `printf( "Hello, world\n" )`

```
1: printf( "Hello, world\n" )
   CPU an OUT: Drucke den Text:  Hello, world\n
```

Aufgabe: Simuliere folgende Ausgabe: `printf( "Heute ist Montag\n" );`

Aufgabe: Simuliere folgende Ausgabe: `printf( "Ich wiege %d kg\n", 100 );`

Aufgabe: Simuliere folgende Ausgabe: `printf( "Das Ergebnis lautet %d\n", a );`

2.4 Eingaben

Motivation und Ziele: Zum Schluss simulieren wir noch zwei Eingabeanweisungen, die auf die beiden Variablen `i` und `j` vom Typ `int` zugreifen können, sodass wir mit unseren drei üblichen Registern bereits folgendes Speicherbildchen haben:

R1 = ?	R2 = ?	R3 = ?	i = ?	j = ?

Aufgabe: Simuliere zunächst die folgenden Eingabeanweisungen: `scanf( "%d", & i )`

```
1: scanf( "%d", & i )
   CPU an IN : Lies eine int Zahl, Eingabe in R2 ablegen
   IN  an CPU: Eine 4711 liegt in R2                          R2 = 4711
   CPU an RAM: Speichere den Wert 4711 aus R2 in der Variablen i   i  = 4711
```

R1 = ?	R2 = 4711	R3 = ?	i = 4711	j = ?

Aufgabe: Simuliere die folgenden Eingabeanweisungen: `scanf( "%d", & i )`

```
2: scanf( "%d", & j )
   CPU an IN : Lies eine int Zahl, Eingabe in R1 ablegen
   IN  an CPU: Eine -1 liegt in R1                            R1 =  -1
   CPU an RAM: Speichere den Wert -1 aus R1 in der Variablen j     j =  -1
```

R1 = -1	R2 = 4711	R3 = ?	i = 4711	j = -1

Aufgabe: Simuliere die folgenden Eingabeanweisungen: `scanf( "%d", & j )`

```
2: scanf( "%d", & j )
```

`CPU an IN :`	Lies eine `int` Zahl, Eingabe in `R1` ablegen	
`IN an CPU:`	Eine `-1` liegt in `R1`	`R1 = -1`
`CPU an RAM:`	Speichere den Wert `-1` aus `R1` in der Variablen `j`	`j = -1`

`R1 = -1`	`R2 = 4711`	`R3 = ?`	`i = 4711`	`j = -1`

Kapitel 3

Fläche eines Rechtecks

Motivation und Ziele: Nach dem wir im vorherigen Kapitel einzelne Anweisungen simuliert haben, wollen wir jetzt ein komplettes Programm für verschiedene Eingaben simulieren. Hierfür greifen wir auf unser erstes Beispielprogramm zurück (siehe auch Skriptkapitel 7), das wir für die Berechnung der Fläche eines Rechtecks entwickelt haben:

```
1  #include <stdio.h>
2
3  int main( int argc, char ** argv )
4      {
5          int a, b, F;
6          printf( "Bitte Seite a eingeben: " );
7          scanf( "%d", & a );
8          printf( "Bitte Seite b eingeben: " );
9          scanf( "%d", & b );
10         F = a * b;
11         printf( "Der Flaecheninhalt betraegt F=%d m*m\n", F );
12     }
```

In den folgenden Aufgaben geht es immer um die Simulation dieses Rechteckprogramms unter verschiedenen Eingaben, die wir jeweils zu Beginn angegeben werden.

Aufgabe: Simuliere obiges Programm mittels drei Registern und der Eingabe `3 40`:

```
 5: int a, b, F
```
`CPU an RAM:` Speicher für die Variablen `a, b` und `F` vom Typ `int` anlegen

R1 = ?	R2 = ?	R3 = ?	a = ?	b = ?	F = ?

```
 6: printf( "Bitte Seite a eingeben: " )
```
`CPU an OUT:` Drucke den Text: `Bitte Seite a eingeben:`

```
 7: scanf( "%d", & a )
```
`CPU an IN :` Lies eine `int` Zahl, Eingabe in `R1` ablegen
`IN  an CPU:` Eine `3` liegt in `R1` — `R1 = 3`
`CPU an RAM:` Speichere den Wert `3` aus `R1` in der Variablen `a` — `a = 3`

R1 = 3	R2 = ?	R3 = ?	a = 3	b = ?	F = ?

```
 8: printf( "Bitte Seite b eingeben: " )
```
`CPU an OUT:` Drucke den Text: `Bitte Seite b eingeben:`

```
 9: scanf( "%d", & b )
```
`CPU an IN :` Lies eine `int` Zahl, Eingabe in `R2` ablegen
`IN  an CPU:` Eine `40` liegt in `R2` — `R2 = 40`
`CPU an RAM:` Speichere den Wert `40` aus `R2` in der Variablen `b` — `b = 40`

R1 = 3	R2 = 40	R3 = ?	a = 3	b = 40	F = ?

```
10: F = a * b
```
`CPU an ALU:` Multiplikation von `R1` (3) und `R2` (40), Ergebnis in `R3` ablegen
`ALU an CPU:` Das Ergebnis ist `120` und liegt in `R3` — `R3 = 120`
`CPU an RAM:` Speichere den Wert `120` aus `R3` in der Variablen `F` — `F = 120`

R1 = 3	R2 = 40	R3 = 120	a = 3	b = 40	F = 120

```
11: printf( "Der Flaecheninhalt betraegt F=%d m*m\n", F )
```
`CPU :` Zunächst muss ich `%d` durch `120` (Wert von `F` in `R3`) ersetzen
`CPU an OUT:` Drucke den Text: `Der Flaecheninhalt betraegt F=120`

```
12: Ende des Hauptprogramms
```
`CPU :` Das war's für mich, ich mach' jetzt Pause :-)

Aufgabe: Simuliere obiges Programm mittels drei Registern und der Eingabe `8 7`:

Aufgabe: Simuliere obiges Programm mittels drei Registern und der Eingabe `-2 -5`:

Kapitel 4

Die einfache Fallunterscheidung: `if`

In diesem Kapitel beschäftigen wir uns mit der einfachen Fallunterscheidung, die in der Programmiersprache C in Form von `if-else` angeboten wird. Eine detailierte Beschreibung der `if`-Anweisung befindet sich in Skriptkapitel 24

4.1 Stoffwiederholung

Die einfache Fallunterscheidung dient dazu, das Programm in Abhängigkeit von Eingaben oder Berechnungen in unterschiedlicher Art und Weise fortzuführen. In der Programmiersprache C sieht die Grundform wie folgt aus:

```
1 if ( Ausdruck )
2    Anweisung_1 ;
3 else Anweisung_2 ;
4 // erste Zeile hinter der if-Anweisung
```

Die wichtigsten Regeln waren wie folgt:

1. Zuerst wird die Bedingung (Ausdruck) ausgewertet, die in runden Klammern `()` nach dem Schlüsselwort `if` steht.
2. Für den Fall, dass die Bedingung den Wert wahr (also einen Wert ungleich null) liefert, wird `Anweisung_1` ausgeführt. Anschließend wird hinter die `if`-Anweisung „gesprungen“, was in unserem Fall die Zeile `4` ist.
3. Liefert die Bedingung den Wert falsch (also den Wert null), wird die nächste Anweisung „übersprungen“ (Zeile `2` in unserem Beispiel) und mit `Anweisung_2` fortgefahren, die hinter dem Schlüsselwort `else` steht.
4. Sollen in einem der Zweige mehr als eine Anweisung ausgeführt werden, müssen geschweifte Klammern `{}` verwendet werden.

4.2 Einfache Handsimulationen

Motivation und Ziele: In diesem Abschnitt starten wir mit der Simulation einfacher, isolierter `if`-Anweisungen. Zu diesem Zweck verwenden wir folgendes Programmschnipsel, in dem das Wort `Bedingung` Platzhalter für eine konkrete Bedingung steht:

```
1 if ( Bedingung )
2       printf( "Bedingung erfuellt\n" );
3 else printf( "Bedingung nicht erfuellt\n" );
4 printf( "Ende Fallunterscheidung\n" );
```

Da es um die Abarbeitung der eigentlichen `if`-Anweisungen geht, benötigen wir zunächst keine Speicherbildchen. Zur Erinnerung Nur die ALU kann Ausdrücke auswerten, arithmethische Vergleiche durchführen und logische Verknüpfungen bilden.

Anmerkung: Es kann gut sein, dass sich an dieser Stelle fortgeschrittene Programmierer etwas wundern. In der Regel ist es nämlich so, dass der Compiler erkennt, wenn Konstanten miteinander verknüft werden, rechtet das Ergebnis selbstständig aus und modiziert gegebenenfalls das resultierende Programm. Beispielsweise würden die meisten Compiler einen Ausdruck wie `3+4` durch eine `7` ersetzen. Für unsere Zwecke, dem Einüben einfacher Handsimulationen zum Zwecke eines tieferen Verständnisses, ist es seht sinnvoll, die Ausdrücke möglichst einfach zu halten, sodass wir obige Leistungsfähigkeit heutiger Compiler einfach ignorieren.

Aufgabe: Simuliere obiges Programm. Verwende dabei drei Register und setze `1 <= 2` als `Bedingung` ein:

`1: if ( 1 <= 2 )`
- `CPU` : Die Konstanten `1` und `2` in `R1` und `R2` ablegen
- `CPU an ALU`: Test auf $\leq$ von `R1` (`1`) und `R2` (`2`), Ergebnis in `R3` ablegen
- `ALU an CPU`: Das Ergebnis ist `1` und liegt in `R3`
- `CPU` : Das Ergebnis ist `1` (`wahr`), weiter mit Zeile `2`

`2: printf( "Bedingung erfuellt\n" );`
- `CPU an OUT`: Drucke den Text: `Bedingung erfuellt\n`
- `CPU` : Ende des `then`-Teils, weiter mit Zeile `4`

`4: printf( "Ende Fallunterscheidung\n" );`
- `CPU an OUT`: Drucke den Text: `Ende Fallunterscheidung\n`

Aufgabe: Wiederhole obige Simulation und setze `4 > 5` als `Bedingung`:

Aufgabe: Simuliere diesmal mit `13 != 4711` als `Bedingung`:

Aufgabe: Beim letzten Versuch betrachten wir die `Bedingung 1 == 2`:

4.3 Ein Vollständiges Programm

Motivation und Ziele: Nun wollen wir unsere Handsimulationen im Rahmen eines einfachen, aber dennoch vollständigen Programms üben. Hierfür verwenden wir das folgende Miniprogramm:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5         int i, j;
 6         scanf( "%d", & i );
 7         scanf( "%d", & j );
 8         if ( Bedingung )
 9              printf( "Bedingung erfuellt\n" );
10         else printf( "Bedingung nicht erfuellt\n" );
11         return 0;
12     }
```

Je nach Aufgabe werden wir die `Bedingung` (Zeile 8) sowie die beiden Eingaben für die Variablen `i` und `j` festlegen. Um die Handsimulationen nicht unnötig aufzublähen, haben wir auf die Eingabeaufforderungen verzichtet.

Aufgabe: Simuliere obiges Programm. Verwende dabei drei Register und `i <= j` als `Bedingung`. Die Eingabe lautet: `2 4`:

5: `int i, j`
`CPU an RAM:` Speicher für die Variablen `i` und `j` vom Typ `int` anlegen

R1 = ?	R2 = ?	R3 = ?	i = ?	j = ?

6: `scanf( "%d", & i )`
`CPU an IN :` Lies eine `int` Zahl, Eingabe in `R1` ablegen
`IN an CPU:` Eine 2 liegt in `R1` — `R1 = 2`
`CPU an RAM:` Speichere den Wert 2 aus `R1` in der Variablen `i` — `i = 2`

R1 = 2	R2 = ?	R3 = ?	i = 2	j = ?

7: `scanf( "%d", & j )`
`CPU an IN :` Lies eine `int` Zahl, Eingabe in `R2` ablegen
`IN an CPU:` Eine 4 liegt in `R2` — `R2 = 4`
`CPU an RAM:` Speichere den Wert 4 aus `R2` in der Variablen `j` — `j = 4`

R1 = 2	R2 = 4	R3 = ?	i = 2	j = 4

```
 8: if ( i <= j )
```
CPU an ALU: Test auf <= von R1 (2) und R2 (4), Ergebnis in R3 ablegen
ALU an CPU: Das Ergebnis ist 1 und liegt in R3 — R3 = 1
CPU : Das Ergebnis ist 1 (wahr), weiter mit Zeile 9

R1 = 2	R2 = 4	R3 = 1	i = 2	j = 4

```
 9: printf( "Bedingung erfuellt\n" );
```
CPU an OUT: Drucke den Text: Bedingung erfuellt\n
CPU : Ende des then-Teils, weiter mit Zeile 11

```
11: return 0
```
CPU : Programmende

Aufgabe: Verwende diesmal die Bedingung `i < j` und die Eingabe `42 -4`:

Aufgabe: Diesmal wird es komplizierter. Die `Bedingung` lautet `(i - 711) != (40 * j)` und als Eingabe sollen `4711` und `100` verwendet werden:

Kapitel 5

Die mehrfache Fallunterscheidung: `switch`

In diesem Kapitel schreiten wir voran und behandeln mehrfachen Fallunterscheidung, die in C als `switch` bekannt ist. Eine detailierte Beschreibung der `switch`-Anweisung befindet sich in Skriptkapitel 25

5.1 Stoffwiederholung

Die mehrfache Fallunterscheidung dient dazu, das Programm in Abhängigkeit von Eingaben oder Berechnungen an verschiedenen Stellen fortzuführen. Im weiteren Sinne handelt es sich um eine erweiterte `if`-Anweisung, die nicht nur auf wahr und falsch sondern „gleichzeitig“ auf unterschiedliche Werte prüft. In der Programmiersprache C sieht die Grundform wie folgt aus:

```
1  switch( Ausdruck )
2  {
3     case Konstante_1: Anweisungen_1;
4                       break;
5     case Konstante_2: Anweisungen_2;
6                       break;
7     ................: ............;
8                       break;
9     case Konstante_n: Anweisungen_n;
10                      break;
11    default         : Anweisungen_default;
12                      break;
13 }
14 // erste Zeile hinter der switch-Anweisung
```

Die wichtigsten Regeln waren wie folgt:

1. Zuerst wird die Bedingung (Ausdruck) ausgewertet, die in runden Klammern `()` nach dem Schlüsselwort `switch` steht. Dieser Ausdruck *muss* ein Ganzzahltyp, also `int` oder `char` sein.
2. Anschließend wird überprüft, ob einer der in den `case`-Verzweigungen aufgeführten *Konstanten* mit dem Wert des `switch`-Ausdrucks übereinstimmt. Ist dies der Fall, wird an dieser Stelle mit der Programmabarbeitung forgefahren. Ist dies nicht der Fall, wird zur `default`-Alternative „gesprungen“. Sollte eine derartige `default`-Alternative nicht vorhanden sein, Wird hinter die gesamte `switch`-Anweisung verzweigt, was in unserem Fall die Zeile `14` ist.
3. Hinter jedem `case` bzw. hinter `default` können auch mehrere Anweisungen stehen, ohne dass man diese mittels geschweifter Klammern zusammenfassen muss.
4. Wird in eine der `case/default`-Alternativen verzweigt, werden der Reihe nach alle Anweisungen ausgeführt, bis die CPU auf eine `break`-Anweisung trifft, was einen Sprung hinter die `switch`-Anweisung nach sich zieht.
5. Die `default`-Alternative muss nicht zu unterst sondern kann überall innerhalb der geschweiften Klammern der `switch`-Anweisung stehen.

5.2 Einfache Handsimulationen

Motivation und Ziele: Die folgenden Handsimulationen zum Thema `switch`-Anweisung beziehen sich immer auf das folgende Programm, das oben auf der nächsten Seite abgedruckt ist. In diesem Programm ist das Wort `Bedingung` ein Platzhalter, der je nach Aufgabenstellung durch einen konkreten Ausdruck zu ersetzen ist. Da wir an verschiedenen Stellen Speicherzugriffe haben, illustrieren wir die Simulation wieder mit kleinen Speicherbildchen. Die Zwischenablage stellt uns wieder drei Register zur Verfügung.

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5        int i;
 6        printf( "Bitte einen Wert fuer i eingeben: " );
 7        scanf( "%d", & i );
 8        switch( Ausdruck )
 9        {
10           case  0: printf( "null\n" );
11                    break;
12           case  1: printf( "eins\n" );
13                    break;
14           case  2: printf( "zwei\n" );
15           case  3: printf( "drei\n" );
16                    break;
17           default: printf( "sonstiges\n" );
18                    break;
19           case  5: case  6: case  7:
20           case  8: printf( "huhu, wie geht es dir?\n" );
21                    printf( "Zahl: %d\n", i );
22                    break;
23        }
24        return 0;
25     }
```

Aufgabe: Simuliere obiges Programm. Die `Bedingung` lautet `i` und die Eingabe ist `0`:

`5: int i;`
`CPU an RAM:` Speicher für die Variable `i` vom Typ `int` anlegen

R1=?	R2=?	R3=?	i=?

`6: printf( "Bitte einen Wert fuer i eingeben: " );`
`CPU an OUT:` Drucke den Text: `Bitte einen Wert fuer i eingeben:`

`7: scanf( "%d", & i )`
`CPU an IN :` Lies eine `int` Zahl, Eingabe in `R1` ablegen
`IN  an CPU:` Eine `0` liegt in `R1` — `R1= 0`
`CPU an RAM:` Speichere den Wert `0` aus `R1` in der Variablen `i` — `i = 0`

R1=0	R2=?	R3=?	i=0

`8: switch( i )`
`CPU` : `i` befindet sich bereits in `R1`
`CPU` : Der Ausdruck ergibt `0`, weiter mit Zeile `10`

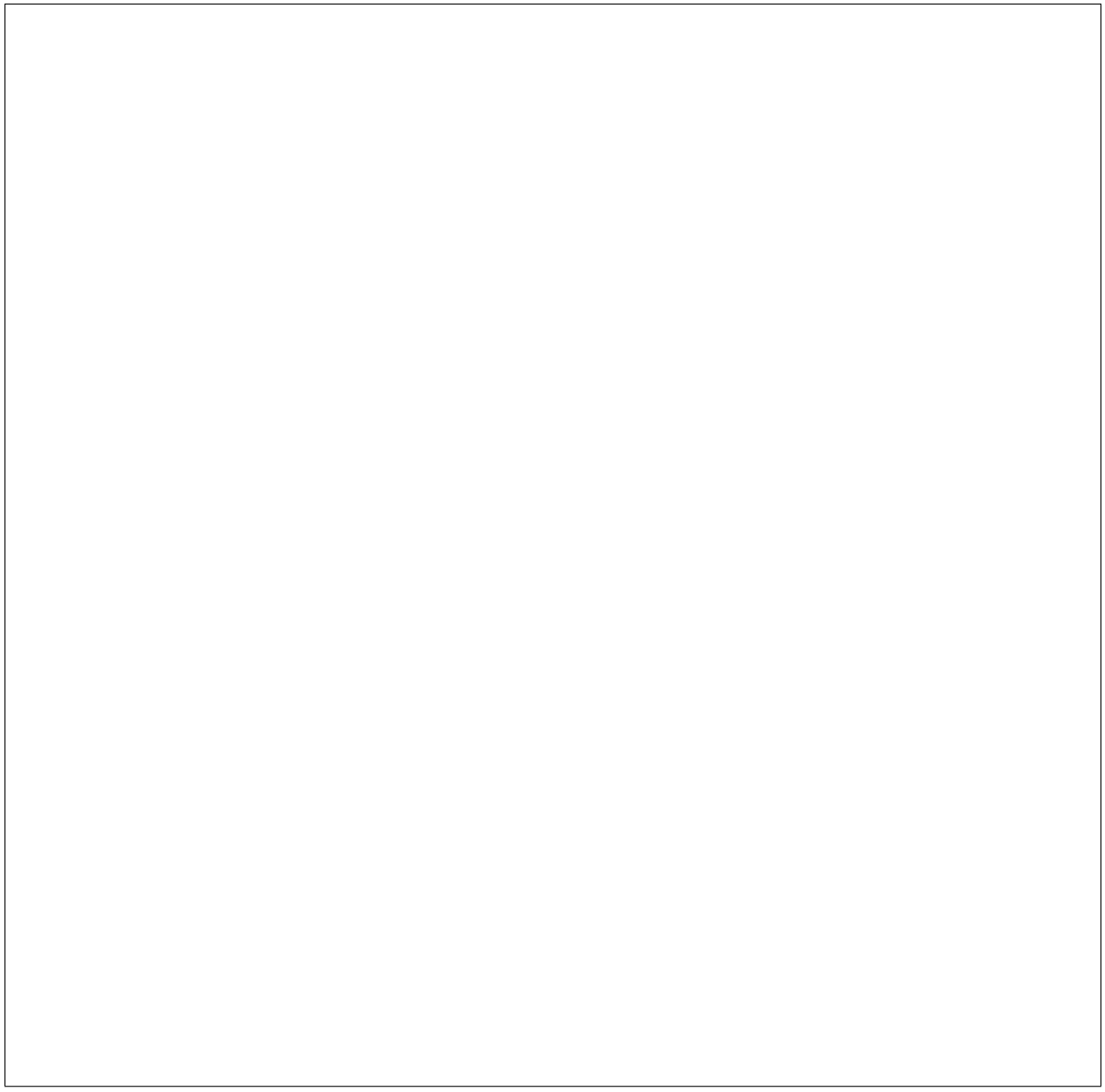
`10: printf( "null\n" );`
`CPU an OUT:` Drucke den Text: `null\n`

`11: break;`
`CPU` : Die `switch`-Anweisung ist zu Ende, weiter mit Zeile `24`

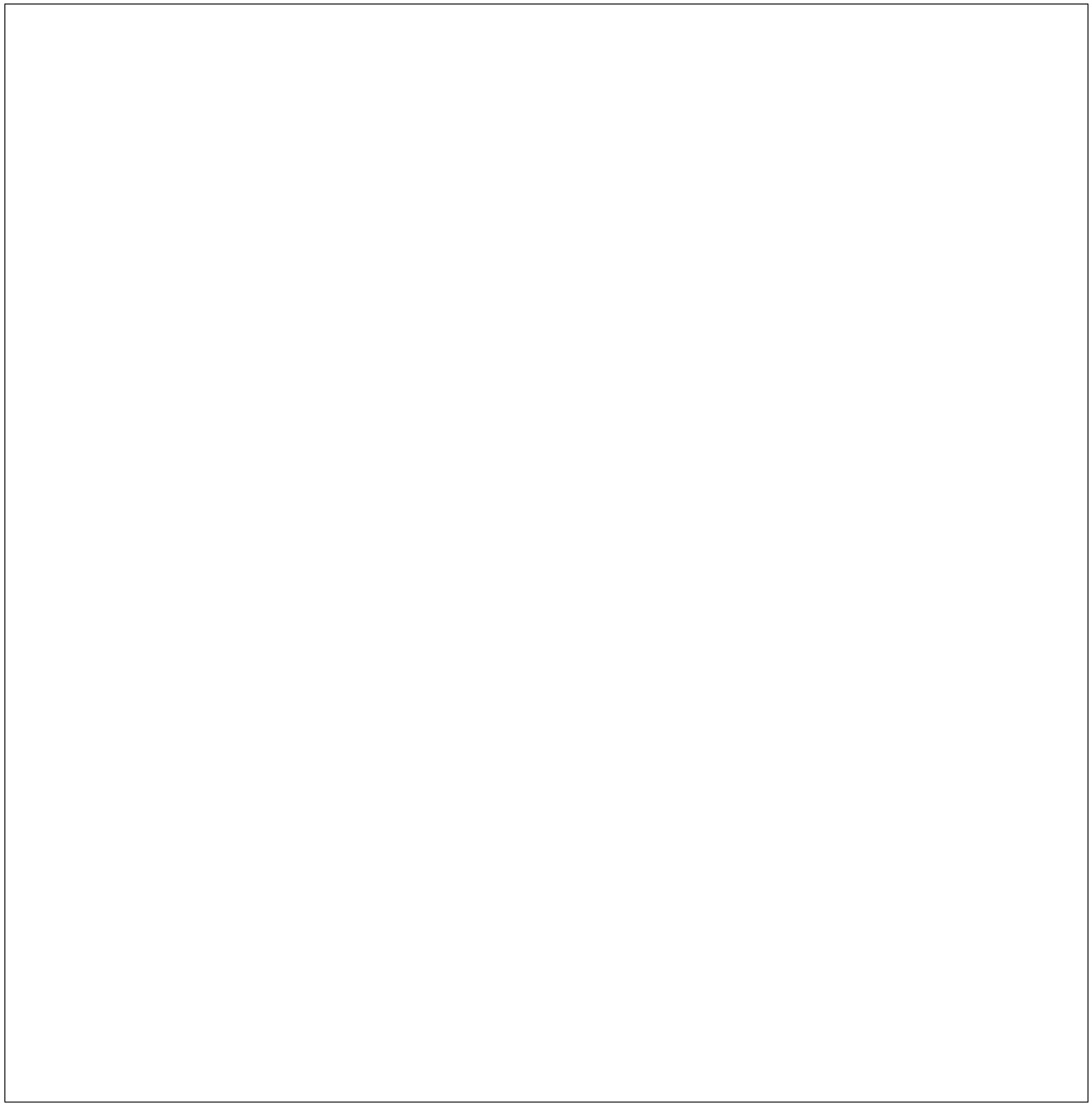
`24: return 0;`
`CPU` : Das Programm ist zu Ende

Aufgabe: Simuliere obiges Programm mit der `Bedingung i + 1` und der Eingabe `2`:

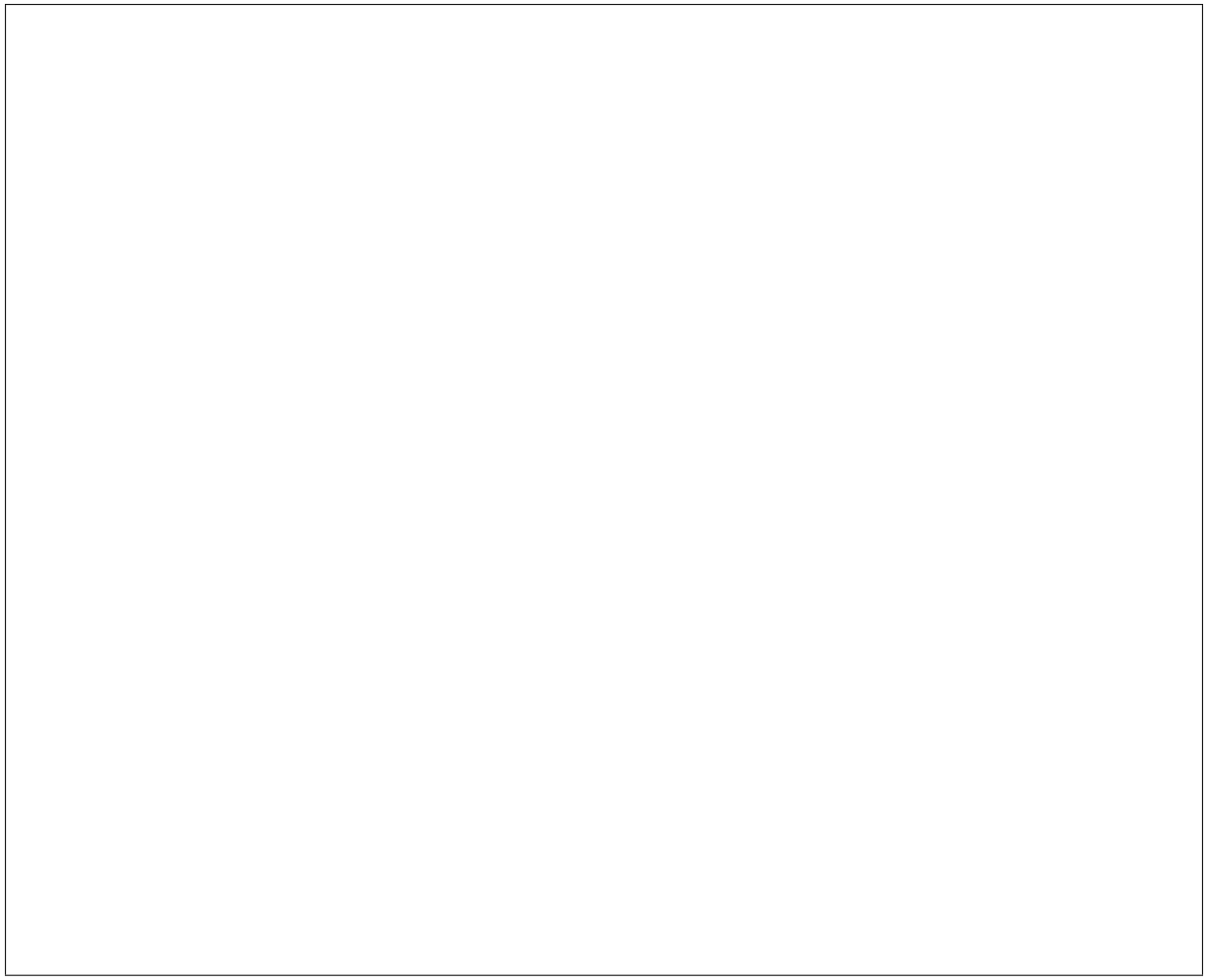
Aufgabe: Dieses Mal lautet die `Bedingung i / 2` und der Eingabe `5`:



Aufgabe: Nun wollen wir die `Bedingung i + 8` und die Eingabe `-2` simulieren:



Aufgabe: Beim letzen Mal simulieren wir die `Bedingung i` und die Eingabe `10`:



Kapitel 6

Die `while`-Schleife

Schleifen sind neben den Fallunterscheidungen die wohl gängigten Strukturierungsmitteln in prozeduralen Programmiersprachen. In der Programmiersprache C werden drei unterschiedliche Schleifenkonstruktionen angeboten, von denen die `while`-Schleife die einfachste ist und in Skriptkapitel 26 detailiert beschrieben wird.

6.1 Stoffwiederholung

Schleifen dienen dazu, Anweisungen wiederholt auszuführen, ohne sie explizit mehrmals hinzuschreiben. In der Programmiersprache C sieht die `while`-Schleife wie folgt aus:

```
1 while( Ausdruck )
2     Anweisung;
3 // erste Zeile hinter der while-Schleife
```

Die wichtigsten Regeln waren wie folgt:

1. Zuerst wird *immer* die Bedingung (Ausdruck) ausgewertet, die in runden Klammern nach dem Schlüsselwort `while` steht.
2. Für den Fall, dass die Bedingung den Wert wahr (also einen Wert ungleich null) liefert, wird der Anweisungsteil ausgeführt. Anschließend wird die Bedingung der `while`-Schleife erneut ausgewertet, um gegebenenfalls wieder mit dem Anweisungsteil fortzufahren. Der Vorgang „Bedingung auswerten und Anweisung ausführen“ wird solange wiederholt, bis die Bedingung den Wert falsch (also den Wert null) liefert.
3. Liefert die Bedingung den Wert falsch (den Wert null), wird hinter die Anweisung „gesprungen“, was in unserem Beispiel die Zeile `3` ist.
4. Sollen in der Schleife mehr als eine Anweisung ausgeführt werden, müssen diese in geschweifte Klammern `{}` gesetzt werden.

6.2 Einfache Handsimulationen

Motivation und Ziele: In diesem Abschnitt starten wir mit der Simulation einfacher, isolierter `while`-Schleifen. Zu diesem Zweck verwenden wir folgendes Programmschnipsel, in dem das Wort `Bedingung` Platzhalter für eine konkrete Bedingung steht:

```
1 while( Bedingung )
2    printf( "Schleifenrumpf\n" );
3 printf( "Ende der While-Schleife\n" );
```

Da es zunächst um die Abarbeitung der eigentlichen `while`-Anweisungen geht, benötigen wir erst einmal keine Speicherbildchen. Zur Erinnerung Nur die ALU kann Ausdrücke auswerten, arithmethische Vergleiche durchführen und logische Verknüpfungen bilden. Bezüglich der Verwendung konstanter Ausdrücke wie beispielsweise `3 > 2` sei auf das in Kapitel 4.2 Gesagte verwiesen.

Aufgabe: Simuliere obiges Programm. Verwende dabei drei Register und setze `1 > 2` als `Bedingung` ein:

`1:` `while ( 1 > 2 )`
`CPU` : Die Konstanten `1` und `2` in `R1` und `R2` ablegen
`CPU an ALU:` Test auf $>$ von `R1` (`1`) und `R2` (`2`), Ergebnis in `R3` ablegen
`ALU an CPU:` Das Ergebnis ist `0` und liegt in `R3`
`CPU` : Das Ergebnis ist `0` (`falsch`), weiter mit Zeile `3`

`3:` `printf( "Ende der While-Schleife\n" );`
`CPU an OUT:` Drucke den Text: `Ende der While-Schleife\n`

Aufgabe: Wiederhole obige Simulation und setze `4 < 5` als `Bedingung` ein:

Es lohnt sich, nochmals einen kurzen Blick auf die beiden Beispielsimulationen zu werfen: Im ersten Fall wurde der Anweisungsteil der `while`-Schleife kein einziges Mal ausgeführt, wohingegen es im zweiten Fall zu einer Endlosschleife kam.

„Na toll, ganz großes Kino! Entweder nie, oder für alle Ewigkeit. Das ist doch alles total sinnlos! Was soll ich bloß damit anfangen?“ Das ist tatsächlich eine gute Frage! Auf eine sinnvolle Antwort kommen wir, wenn die Frage wie folgt umformulieren: Woran liegt es denn, dass es im zweiten Fall zu einer Endlosschleife kommt? *„Klar, jetzt kommt bestimmt so ein der Student ist einfach zu blöd oder so.“* Das kann im Einzelfall natürlich der Fall sein, aber natürlich nicht bei unseren Studenten, oder? Im Ernst, schau dir doch noch einmal die Bedingung in Zeile `1` an. Was fällt dir auf? *„Ok, wenn ich mir das in Ruhe anschaue, dann sehe ich, dass sich die Bedingung nie verändern kann, denn sie ist ja ein konstanter Ausdruck, der entweder wahr oder falsch ist. Wenn sich nichts ändern kann, kann die Schleife bzw. die CPU auch nichts merken. Also muss ich irgendwie und irgendwo durch den Schleifenrumpf etwas ändern bzw. potentiell ändern können, damit die Schleife auch irgendwann mal aufhört zu arbeiten.“* Sehr gut, du hast es erfaßt! Bei einer sinnvollen Schleifenkonstruktion muss sich beispielsweise eine Variable durch eine Eingabe oder eine ausgeführte Berechnung (potientell) ändern können, dann könnte es klappen. Diese Dinge üben wir per Handsimulation im nächsten Abschnitt.

6.3 Ein Vollständiges Programm

Motivation und Ziele: Mit dem folgenden sehr einfachen aber dennoch vollständigen Miniprogramm wollen wir die Handsimulation der `while`-Schleife weiter üben:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5         int i, j;
 6         scanf( "%d", & i );
 7         scanf( "%d", & j );
 8         while( Bedingung )
 9         {
10             printf( "Schleifendurchlauf\n" );
11             Anweisung;
12         }
13         return 0;
14     }
```

Je nach Aufgabe werden wir die `Bedingung` in Zeile `8`, die `Anweisung` in Zeile `11` sowie die Eingaben für die beiden Variablen `i` und `j` festlegen. Um die Handsimulationen nicht unnötig aufzublähen, haben wir wiederum auf die Eingabeaufforderungen verzichtet.

Aufgabe: Diesmal verwenden wir drei Register, die Zuweisung `i = i + 1` als `Anweisung`, `i < j` als `Bedingung` und die Zahlen `1 3` als Eingabe für die beiden Variablen `i` und `j`.

```
 5: int i, j
    CPU an RAM: Speicher für die Variablen i und j vom Typ int anlegen
```

R1 = ?	R2 = ?	R3 = ?	i = ?	j = ?

```
 6: scanf( "%d", & i )
    CPU an IN : Lies eine int Zahl, Eingabe in R1 ablegen
    IN  an CPU: Eine 1 liegt in R1                                  R1 = 1
    CPU an RAM: Speichere den Wert 1 aus R1 in der Variablen i      i  = 1
```

R1 = 1	R2 = ?	R3 = ?	i = 1	j = ?

```
 7: scanf( "%d", & j )
    CPU an IN : Lies eine int Zahl, Eingabe in R2 ablegen
    IN  an CPU: Eine 3 liegt in R2                                  R2 = 3
    CPU an RAM: Speichere den Wert 3 aus R2 in der Variablen j      j  = 3
```

R1 = 1	R2 = 3	R3 = ?	i = 1	j = 3

```
 8: while( i < j )
    CPU an ALU: Test auf < von R1 (1) und R2 (3), Ergebnis in R3 ablegen
    ALU an CPU: Das Ergebnis ist 1 und liegt in R3                  R3 = 1
    CPU       : Das Ergebnis ist wahr, also weiter mit Zeile 10
```

R1 = 1	R2 = 3	R3 = 1	i = 1	j = 3

```
10: printf( "Schleifendurchlauf\n" );
    CPU an OUT: Drucke den Text: Schleifendurchlauf\n

11: i = i + 1;
    CPU       : Die Konstante 1 in R3 ablegen                       R3 = 1
    CPU an ALU: Addition von R1 (1) und R3 (1), Ergebnis in R3 ablegen
    ALU an CPU: Das Ergebnis ist 2 und liegt in R3                  R3 = 2
    CPU an RAM: Speichere den Wert 2 aus R3 in der Variablen i      i  = 2
    CPU       : Weiter mit Zeile 8
```

R1 = 2	R2 = 3	R3 = 2	i = 2	j = 3

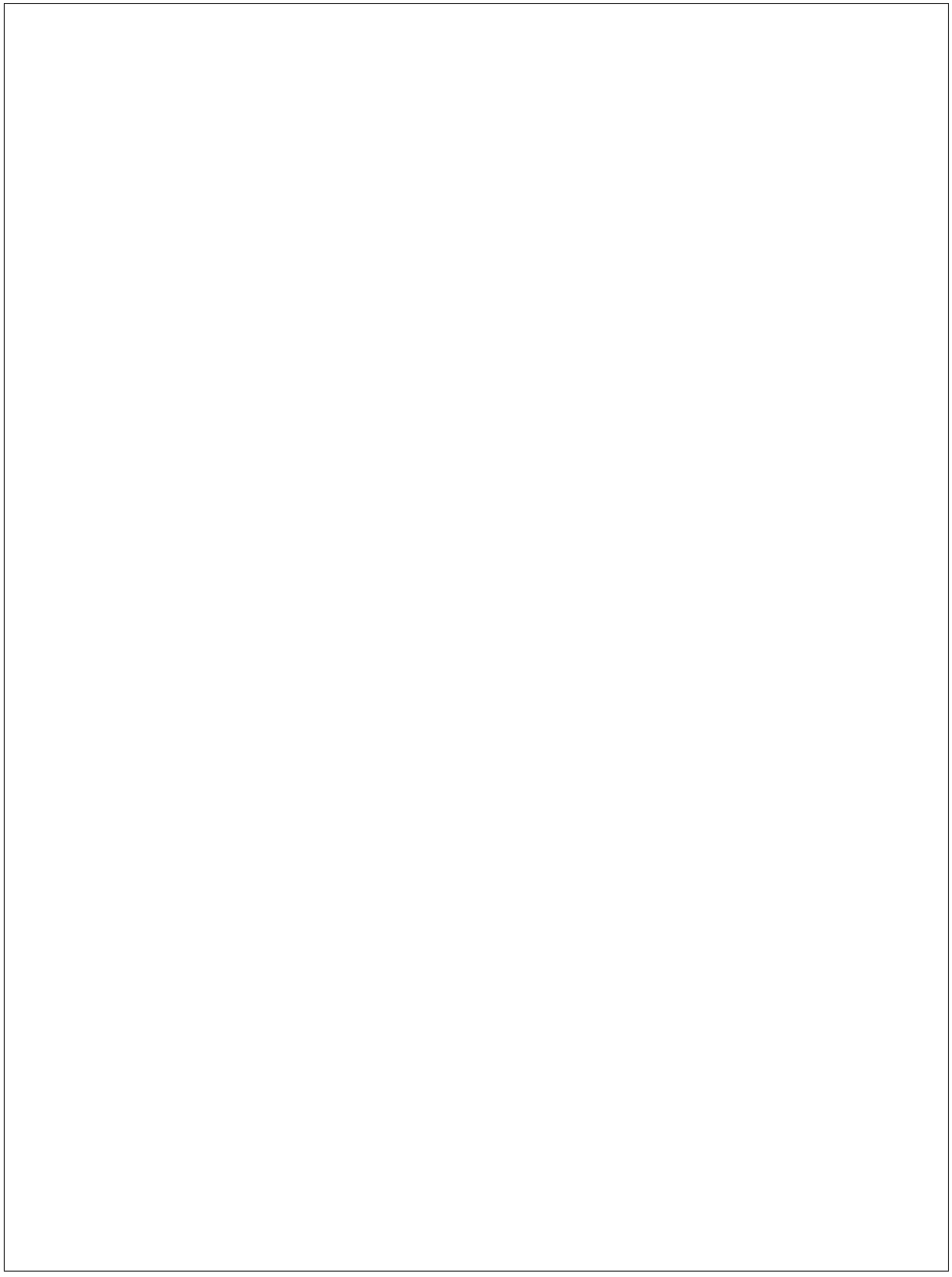
```
 8: while( i < j )
    CPU an ALU: Test auf < von R1 (2) und R2 (3), Ergebnis in R3 ablegen
    ALU an CPU: Das Ergebnis ist 1 und liegt in R3                  R3 = 1
    CPU       : Das Ergebnis ist wahr, also weiter mit Zeile 10
```

R1 = 2	R2 = 3	R3 = 1	i = 2	j = 3

```
10: printf( "Schleifendurchlauf\n" );
    CPU an OUT: Drucke den Text: Schleifendurchlauf\n

11: i = i + 1;
    CPU       : Die Konstante 1 in R3 ablegen                                  R3 = 1
    CPU an ALU: Addition von R1 (2) und R3 (1), Ergebnis in R3 ablegen
    ALU an CPU: Das Ergebnis ist 3 und liegt in R3                             R3 = 3
    CPU an RAM: Speichere den Wert 3 aus R3 in der Variablen i                 i = 3
    CPU       : Weiter mit Zeile 8
    ---------------------------------------------------------------------------
    R1 = 3          R2 = 3          R3 = 3          i = 3          j = 3
    ---------------------------------------------------------------------------
 8: while( i < j )
    CPU an ALU: Test auf < von R1 (3) und R2 (3), Ergebnis in R3 ablegen
    ALU an CPU: Das Ergebnis ist 0 und liegt in R3                             R3 = 0
    CPU       : Das Ergebnis ist falsch, also weiter mit Zeile 13
    ---------------------------------------------------------------------------
    R1 = 3          R2 = 3          R3 = 0          i = 3          j = 3
    ---------------------------------------------------------------------------
13: return 0
    CPU       : Programmende
```

Aufgabe: Für die folgende Handsimulation nehmen wir die vorherige Aufgabe, wandeln sie aber geringfügig ab: Die `Bedingung` lautet diesmal `i >= j`, die `Anweisung` verändern wir zu `i = i - 1` und als Eingabe für die beiden Variablen `i` und `j`. stehen die beiden Zahlen `2 1` zur Verfügung.



Aufgabe: In der dritten Handsimulation ändern wir obiges Progamm wie folgt ab: Die `Bedingung` lautet diesmal `i != j`, als `Anweisung` nehmen wir `i = i + 3; j = j + 2` und als Eingabe für die beiden Variablen `i` und `j`. stehen diesmal die Zahlen `1` und `3` zur Verfügung.

Damit hätten wir auch die letzte kleine `while`-Schleife per Hand simuliert womit wir dieses Kapitel auch abschließen.

Kapitel 7

Die `do-while`-Schleife

Bevor wir uns der wesentlich komplexeren `for`-Schleife beschäftigen, behandeln wir hier noch schnell die `do-while`-Schleife, da sie der `while`-Schleife sehr ähnelt.

7.1 Stoffwiederholung

Gemäß Skriptkapitel 28 sieht die `do-while`-Schleife wie folgt aus:

```
1 do Anweisung;
2 while( Ausdruck );
```

Die wichtigsten Regeln waren wie folgt:

1. Zuerst wird *immer* der Schleifenrumpf (in unserem Fall die `Anweisung`) ausgeführt. Erst *anschließend* wird die Schleifenbedingung (`Ausdruck`) ausgewertet.
2. Die `do-while`-Schleife wird solange ausgefürt, wie die Bedingung erfüllt ist, d.h. der `Ausdruck` einen Wert ungleich Null liefert. Die Klammern um den `Ausdruck` gehören zur Syntax der `do-while`-Schleife.
3. Sollen in der Schleife mehr als eine Anweisung ausgeführt werden, müssen diese wie üblich in geschweifte Klammern `{}` gesetzt werden.

Ein kurzer Vergleich zeigt, dass im Gegensatz zur `while`-Schleife die `do-while`-Schleife erst den Schleifenrumpf ausführt und dann die Bedingung überprüft. Entsprechend wird der Schleifenrumpf auch mindestens einmal ausgeführt, was bei der `while`-Schleife nicht unbedingt der Fall ist, da der `Ausdruck` in der Schleifenbedingung gleich den Wert Null annehmen kann. Beide Schleifen lassen sich aber leicht ineinander überführen:

```
1 if ( Ausdruck )            // ist identisch mit:
2    do Anweisung;           // while( Ausdruck )
3    while( Ausdruck );      //    Anweisung;
```

In diesem Beispiel wurde die `do-while`-Schleife einfach in eine Fallunterscheidung eingebettet. Der noch nicht so geübten Leser sollte am Ende dieses Kapitels anhand eines einfachen Beispiels zeigen, dass beide Schleifen identisch sind.

7.2 Einfache Handsimulationen

Motivation und Ziele: In diesem Abschnitt starten wir mit zwei einfachen Simulation, um zunächst die `do-while`-Schleife korrekt auszuführen. Hierzu verwenden wir den folgenden Programmausschnitt, in dem `i` eine Variable vom Typ `int` ist.

```
1 do {
2       printf( "Schleifendurchlauf\n" );
3       i = i + 1;
4     } while( i < 1 );
5 printf( "Ende der Do-While-Schleife\n" );
```

Aufgabe: Simuliere obiges Programm. Verwende dabei nur zwei Register. In dieser ersten Handsimulation gehen wir davon aus, dass sowohl die Variable `i` als auch die beiden Register mit dem Wert `0` initialisiert sind. Gemäß obiger Voraussetzungen fangen wir mit folgendem Speicherbildchen an:

R1 = 0	R2 = 0	i = 0

2: `printf( "Schleifendurchlauf\n" );`

`CPU an OUT`: Drucke den Text: `Schleifendurchlauf\n`

3: `i = i + 1;`

`CPU` :	Die Konstante `1` in `R2` ablegen	`R2 = 1`
`CPU an ALU`:	Addition von `R1` (`0`) und `R2` (`1`), Ergebnis in `R2` ablegen	
`ALU an CPU`:	Das Ergebnis ist `1` und liegt in `R2`	`R2 = 1`
`CPU an RAM`:	Speichere den Wert `1` aus `R2` in der Variablen `i`	`i = 1`

R1 = 1	R2 = 1	i = 1

4: `while( i < 1 );`

`CPU` :	Die Konstante `1` in `R2` ablegen	`R2 = 1`
`CPU an ALU`:	Test auf `<` von `R1` (`1`) und `R2` (`1`), Ergebnis in `R2` ablegen	
`ALU an CPU`:	Das Ergebnis ist `0` und liegt in `R2`	`R2 = 0`
`CPU` :	Das Ergebnis ist `falsch`, also weiter mit Zeile `5`	

R1 = 1	R2 = 0	i = 1

5: `printf( "Ende der Do-While-Schleife\n" );`

`CPU an OUT`: Drucke den Text: `Ende der Do-While-Schleife\n`

Aufgabe: In dieser Aufgabe soll obige Simulation wiederholt werden. Nur gehen wir diesmal davon aus, dass die Variable `i` mit dem Wert `1` initialisiert ist und dass die Bedingung in Zeile `4` wie folgt lautet: `i <= 2`.

7.3 Ein Vollständiges Programm

Motivation und Ziele: Mit dem folgenden sehr einfachen aber dennoch vollständigen Miniprogramm wollen wir die Handsimulation der `do-while`-Schleifer weiter üben:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5        int i, j;
 6        scanf( "%d", & i );
 7        scanf( "%d", & j );
 8        do {
 9             printf( "i=%d\n", i );
10             Anweisung;
11           } while( Bedingung );
12        return 0;
13     }
```

Je nach Aufgabe werden wir wie üblich die `Anweisung` in Zeile `10`, die `Bedingung` in Zeile `11` sowie die Eingaben für die beiden Variablen `i` und `j` festlegen. Um die Handsimulationen nicht unnötig aufzublähen, haben wir wiederum auf die Eingabeaufforderungen verzichtet. Und wie schon so oft stehen für alle anfallenden Berechnungen drei Register zur Verfügung.

Aufgabe: In dieser Aufgabe sollen die Zahlen `i, i+1, ..., i+j` ausgegeben. Unabhängig von der Zahl `j` soll mindestens die Zahl `i` ausgegeben werden. Hierzu setzen wir als `Anweisung: i = i + 1; j = j - 1;` und als `Bedingung: j > 0`. Als Eingabe stehen die beiden Zahlen `3 2` zur Verfügung.

`5:` `int i, j`
`CPU an RAM:` Speicher für die Variablen `i` und `j` vom Typ `int` anlegen

`R1=?`	`R2=?`	`R3=?`	`i=?`	`j=?`

`6:` `scanf( "%d", & i )`
`CPU an IN :` Lies eine `int` Zahl, Eingabe in `R1` ablegen
`IN an CPU:` Eine `3` liegt in `R1` — `R1= 3`
`CPU an RAM:` Speichere den Wert `3` aus `R1` in der Variablen `i` — `i = 3`

`R1=3`	`R2=?`	`R3=?`	`i=3`	`j=?`

`7:` `scanf( "%d", & j )`
`CPU an IN :` Lies eine `int` Zahl, Eingabe in `R2` ablegen
`IN an CPU:` Eine `2` liegt in `R2` — `R2= 2`
`CPU an RAM:` Speichere den Wert `2` aus `R2` in der Variablen `j` — `j = 2`

`R1=3`	`R2=2`	`R3=?`	`i=3`	`j=2`

```
 9: printf( "i=%d\n", i );
    CPU an OUT: Drucke den Text: i=3\n

10: i = i + 1; j = j -1;
    CPU       : Die Konstante 1 in R3 ablegen                                  R3 = 1
    CPU an ALU: Addition von R1 (3) und R3 (1), Ergebnis in R1 ablegen
    ALU an CPU: Das Ergebnis ist 4 und liegt in R1                             R1 = 4
    CPU an RAM: Speichere den Wert 4 aus R1 in der Variablen i                 i  = 4
    CPU an ALU: Subtraktion von R2 (2) und R3 (1), Ergebnis in R2 ablegen
    ALU an CPU: Das Ergebnis ist 1 und liegt in R2                             R2 = 1
    CPU an RAM: Speichere den Wert 1 aus R2 in der Variablen j                 j  = 1
    ---------------------------------------------------------------------------------
    R1 = 4          R2 = 1          R3 = 1          i = 4          j = 1
    ---------------------------------------------------------------------------------

11: while( j > 0 );
    CPU       : Die Konstante 0 in R3 ablegen                                  R3 = 0
    CPU an ALU: Test auf > von R2 (1) und R3 (0), Ergebnis in R3 ablegen
    ALU an CPU: Das Ergebnis ist 1 und liegt in R3                             R3 = 1
    CPU       : Das Ergebnis ist wahr, also weiter mit Zeile 9
    ---------------------------------------------------------------------------------
    R1 = 4          R2 = 1          R3 = 1          i = 4          j = 1
    ---------------------------------------------------------------------------------

 9: printf( "i=%d\n", i );
    CPU an OUT: Drucke den Text: i=4\n

10: i = i + 1; j = j -1;
    CPU       : Die Konstante 1 in R3 ablegen                                  R3 = 1
    CPU an ALU: Addition von R1 (4) und R3 (1), Ergebnis in R1 ablegen
    ALU an CPU: Das Ergebnis ist 5 und liegt in R1                             R1 = 5
    CPU an RAM: Speichere den Wert 5 aus R1 in der Variablen i                 i  = 5
    CPU an ALU: Subtraktion von R2 (1) und R3 (1), Ergebnis in R2 ablegen
    ALU an CPU: Das Ergebnis ist 0 und liegt in R2                             R2 = 0
    CPU an RAM: Speichere den Wert 0 aus R2 in der Variablen j                 j  = 0
    ---------------------------------------------------------------------------------
    R1 = 5          R2 = 0          R3 = 1          i = 5          j = 0
    ---------------------------------------------------------------------------------

11: while( j > 0 );
    CPU       : Die Konstante 0 in R3 ablegen                                  R3 = 0
    CPU an ALU: Test auf > von R2 (0) und R3 (0), Ergebnis in R3 ablegen
    ALU an CPU: Das Ergebnis ist 0 und liegt in R3                             R3 = 0
    CPU       : Das Ergebnis ist falsch, also weiter mit Zeile 12
    ---------------------------------------------------------------------------------
    R1 = 5          R2 = 0          R3 = 0          i = 5          j = 0
    ---------------------------------------------------------------------------------

12: return 0
    CPU       : Programmende
```

Aufgabe: Diesmal wollen wir den Wert der Variablen `i` `j`-Mal verdoppeln. Dazu verändern wir die `Anweisung` zu `i = i * 2; j = j - 1;` und die `Bedingung` zu `j >= 0`. Unabhängig von der Zahl `j` soll mindestens die Zahl `i` ausgegeben werden. Eingegeben werden diesmal die beiden Zahlen `2 2`. Aus Platzgründen sparen wir uns die Zeile `12`.

Kapitel 8

Die `for`-Schleife

Nun widmen wir uns endlich der `for`-Schleife, die aufgrund ihrer Struktur um um einiges komplizierter als die beiden vorangegangenen Schleifen ist. Nichtsdesdotrotz werden versuchen, uns diesem Konstrukt schrittweise zu nähern, in dem wir eine größere Zahl von Handsimulation mit jeweils steigendem Schwierigkeitsgrad behandeln.

8.1 Stoffwiederholung

In Skriptkapitel 27 wurde ausführlich besprochen, dass die `for`-Schleife aus einem Schleifenkopf mit *bis zu drei* Ausdrücken und einem Schleifenrumpf besteht:

```
1 for( Ausdruck_1; Ausdruck_2; Ausdruck_3 )
2     Anweisung;
3 // erste Anweisung hinter der for-Schleife
```

Die wichtigsten Regeln waren wie folgt:

1. Es müsser Schleifenkopf enthält drei Ausdrücke, die jeweils durch ein *Semikolon* zu trennen sind. Jeder dieser Ausdrücke kann auch leer sein; das jeweilige Semikolon muss aber trotzdem gesetzt bleiben.

2. Als erstes wird immer der `Ausdruck_1` ausgewertet. *Unabhängig* vom Ergebnis dieser Auswertung wird immer mit dem nächsten Punkt fortgefahren.

3. Nun wird `Ausdruck_2` ausgewertet. Sollte das Ergebnis falsch sein (den Wert Null haben), wird hinter der `for`-Schleife weiter gemacht, was in obigem Beispiel die Zeile `3` ist. Im Falle eines wahren Ausdrucks (ein Ergebnis ungleich Null haben), wird mit der Ausführung der `Anweisung` sowie mit dem Auswerten von `Ausdruck_3` fortgefahren. Anschließend beginnt das Spiel dieses Punkte (Punkt **??** um Missverständnissen vorzubeugen) von vorne sowie der `Ausdruck_2` ausgewertet.

An vielen Literaturstellen wird die Funktionsweise der `for`-Schleife mittels der folgenden `while`-Schleife erläutert.

```
1  // Vielfach wird die unten stehende while-Schleife mit der
2  // folgender for-Schleife gleichgesetzt:
3  //
4  //    for( Ausdruck_1; Ausdruck_2; Ausdruck_3 )
5  //       Anweisung;
6
7  Ausdruck_1;
8  while( Ausdruck_2 )
9  {
10    Anweisung;
11    Ausdruck_3;
12 }
```

Diese Erklärung hilft vielen Studenten und ist auch korrekt, so lange im Schleifenrumpf keine `continue`-Anweisung vorkommt. Die `continue`-Anweisung veranlasst die CPU *sofort* den `Ausdruck_2` abzuarbeiten, *ohne* `Ausdruck_3` auszuwerten.

8.2 Einfache Handsimulationen

Motivation und Ziele: In diesem Abschnitt verwenden wir die `for`-Schleife wie wir sie eingangs aufgeschrieben haben. Dabei werden die drei Ausdrücke sehr einfach gestaltet sein und der Schleifenrumpf besteht aus einer einfachen Ausgabe. Je Aufgabe werden wir die Bestandteile ein wenig abändern. Ziel dieser ersten Handsimulationen ist es, die Abarbeitungsreihenfolge der einzelnen Bestandteile zu üben.

Aufgabe: Das folgende Programm soll zwei mal ausgeben, dass du im Unterricht nicht schlafen sollst. Hierzu verwenden wir die `for`-Schleife in Form einer einfachen Zählschleife. Hiezu nehmen wir eine Variable `i` vom Typ `int`. Entsprechend brauchen wir Speicher für eine Variable sowie zwei Register für unsere Berechnungen.

```
1  for( i = 0; i < 2; i = i + 1 )
2     printf( "Schlafe nicht im Unterricht!\n" );
3  // erste Zeile nach der for-Schleife
```

Somit ergibt sich also folgende Handsimulation:

```
1: j = 0;
   CPU        : Die Konstante 0 in R1 ablegen                    R1 = 0
   CPU an RAM: Speichere den Wert 0 aus R1 in der Variablen j    j  = 0
   ---------------------------------------------------------------------
   R1 = 0                        R2 = ?                          i = 0
   ---------------------------------------------------------------------
```

```
1: i < 2;
   CPU        : Die Konstante 2 in R2 ablegen                                  R2 = 2
   CPU an ALU : Test auf < von R1 (0) und R2 (2), Ergebnis in R2 ablegen
   ALU an CPU : Das Ergebnis ist 1 und liegt in R2                             R2 = 1
   CPU        : Das Ergebnis ist wahr, also weiter mit Zeile 2
   ---------------------------------------------------------------------------
   R1 = 0                              R2 = 1                          i = 0
   ---------------------------------------------------------------------------
2: printf( "Schlafe nicht im Unterricht!\n" );
   CPU an OUT : Drucke den Text: Schlafe nicht im Unterricht!\n

1: i = i + 1;
   CPU        : Die Konstante 1 in R2 ablegen                                  R2 = 1
   CPU an ALU : Addition von R1 (0) und R2 (1), Ergebnis in R2 ablegen
   ALU an CPU : Das Ergebnis ist 1 und liegt in R2                             R2 = 1
   CPU an RAM : Speichere den Wert 1 aus R2 in der Variablen i                 i  = 1
   ---------------------------------------------------------------------------
   R1 = 1                              R2 = 1                          i = 1
   ---------------------------------------------------------------------------
1: i < 2;
   CPU        : Die Konstante 2 in R2 ablegen                                  R2 = 2
   CPU an ALU : Test auf < von R1 (1) und R2 (2), Ergebnis in R2 ablegen
   ALU an CPU : Das Ergebnis ist 1 und liegt in R2                             R2 = 1
   CPU        : Das Ergebnis ist wahr, also weiter mit Zeile 2
   ---------------------------------------------------------------------------
   R1 = 1                              R2 = 1                          i = 1
   ---------------------------------------------------------------------------
2: printf( "Schlafe nicht im Unterricht!\n" );
   CPU an OUT : Drucke den Text: Schlafe nicht im Unterricht!\n

1: i = i + 1;
   CPU        : Die Konstante 1 in R2 ablegen                                  R2 = 1
   CPU an ALU : Addition von R1 (1) und R2 (1), Ergebnis in R2 ablegen
   ALU an CPU : Das Ergebnis ist 2 und liegt in R2                             R2 = 2
   CPU an RAM : Speichere den Wert 2 aus R2 in der Variablen i                 i  = 2
   ---------------------------------------------------------------------------
   R1 = 2                              R2 = 2                          i = 2
   ---------------------------------------------------------------------------
1: i < 2;
   CPU        : Die Konstante 2 in R2 ablegen                                  R2 = 2
   CPU an ALU : Test auf < von R1 (2) und R2 (2), Ergebnis in R2 ablegen
   ALU an CPU : Das Ergebnis ist 0 und liegt in R2                             R2 = 0
   CPU        : Das Ergebnis ist falsch, also weiter mit Zeile 3
   ---------------------------------------------------------------------------
   R1 = 2                              R2 = 0                          i = 2
   ---------------------------------------------------------------------------
```

Und schon sind wir mit unseren zwei Schleifendurchläufen fertig.

Aufgabe: Nun besteht die Aufgabe darin, die Jahreszahlen von 2000 bis 2002 (einschließlich) mittels einer `for`-Schleife auszugeben. Zu diesem Zweck nehmen wir eine Variable `jahr` vom `int`, die wir innerhalb der `for`-Schleife schrittweise hoch zählen.

```
1 for( i = 0; i < 2; i = i + 1 )
2    printf( "Schlafe nicht im Unterricht!\n" );
3 // erste Zeile nach der for-Schleife
```

R1 = 2000	R2 = 2	jahr = 2000
R1 = 20	R2 = 200	jahr = 200