Computersysteme

Stacks
Anwendung in der Assembler-Programmierung

Unterprogramme

Betrachten wir zunächst folgendes Programm m_mod_n :

```
/Berechne m modulo n für positive Integerwerte m und n.
/Beim Programmstart sollen m in R1 und n in R2 stehen.
/R3 wird als Hilfsregister für Vergleiche verwendet.
/Am Programmende soll das Ergebnis in R1 stehen.
m_mod_n:            /Schleifenbeginn
SLT     R3,R1,R2    /R3 wird genau dann gesetzt, wenn R1<n .
BNEZ    R3,Fertig   /Ggf. steht in R1 das Ergebnis m modulo n => Fertig
SUB     R1,R1,R2    /R1 soll solange um n reduziert werden, bis R1<0
J       m_mod_n     /Springe zum Schleifenanfang
Fertig:             /Programmende
HALT
```

Welche Probleme treten auf, wenn m_mod_n als Unterprogramm verwendet werden soll?

Rücksprungadresse?
Registerbelegung?
Parameterübergabe?

Stacks

Ein Stack (Kellerspeicher) ist ein Speicher, auf den nur auf bestimmte Weise zugegriffen werden kann.

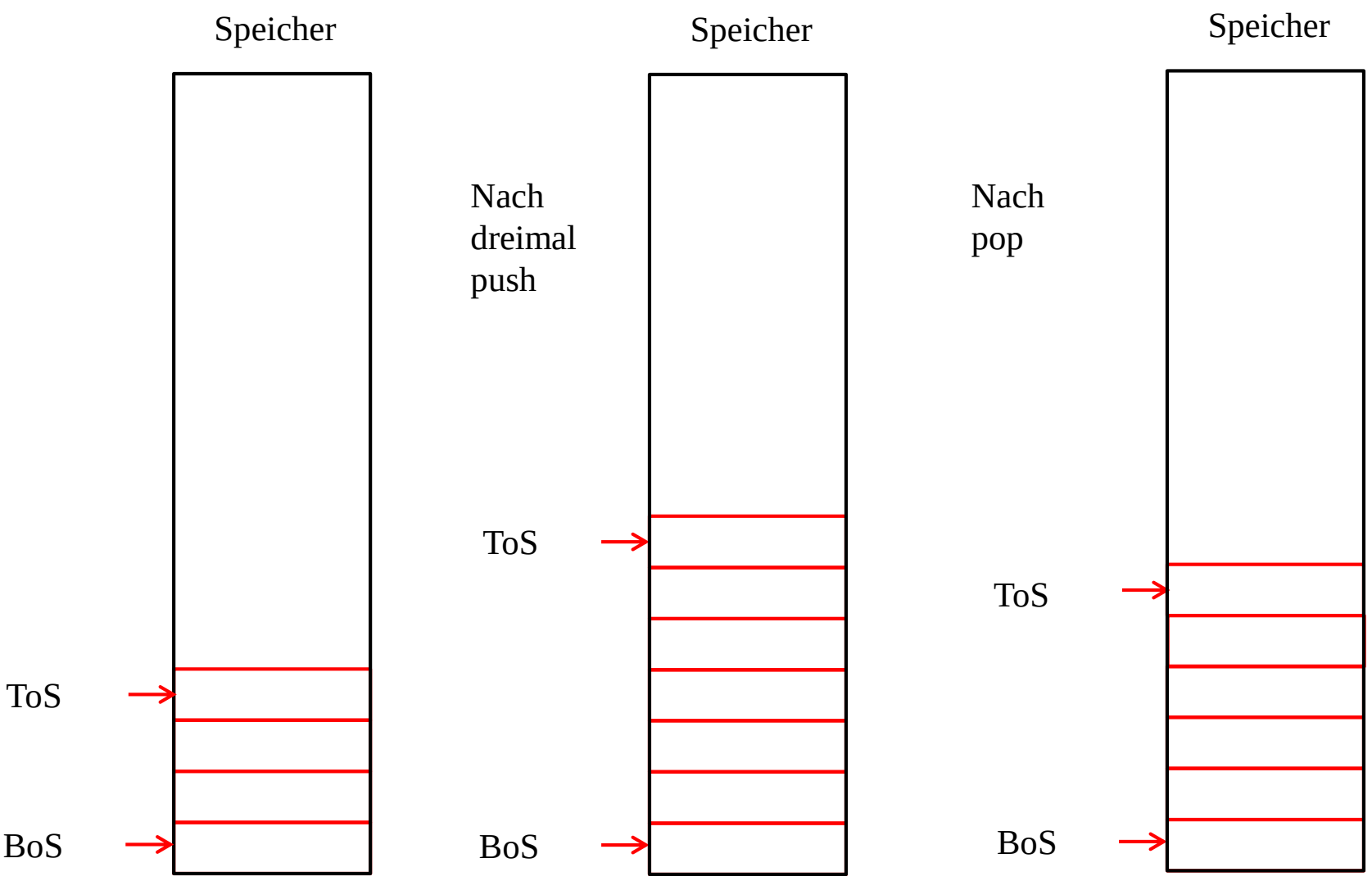
Mit dem push-Befehl wird ein neuer Wert auf den Stack gelegt (von oben in den Keller gelegt)**, zusätzlich zu den bereits im Stack befindlichen Werten.**

Mit dem pop-Befehl wird der oberste Wert vom Stack gelesen, wobei er im Moment des Lesens aus dem Stack gelöscht wird (von oben aus dem Keller geholt)**.**

Man kann sich das auch so wie eine runde Tablettendose vorstellen, deren Querschnitt die Form der Tabletten hat: Wenn man eine neue Tablette hinein tut, kann man zuerst nur diese wieder heraus holen. Erst wenn man die oberste Tablette heraus genommen (gelesen) hat kann man auf die nächste, darunter liegende Tablette zugreifen.

Stacks werden in Computern realisiert als Teile von physikalisch vorhandenem RAM. Dabei benutzt man als Hilfsvariable einen Pointer (Zeiger) auf den **Top-of-Stack (ToS)**, also auf die Speicherzelle, in die zuletzt ein Element hineingeschrieben worden ist. Darüber hinaus merkt man sich ggf. in einer weiteren Hilfsvariablen den **Bottom-of-Stack**, also die Speicherstelle des ersten Elements, das im Stack steht, damit man eine Prüfmöglichkeit hat, wann der Stack durch wiederholtes Lesen leer geworden ist. Dieser Bottom-of-Stack verändert sich nie, während der Top-of-Stack bei jeder push-Operation hoch- und bei jeder pop-Operation heruntergezählt wird.

In der folgenden Grafik ist die Aktivität eines Stacks dargestellt:
Es sind vier Elemente im Stack gespeichert. Danach werden drei weitere Elemente in den Stack geschrieben, jeweils indem der ToS erhöht wird und das neue Element an der Adresse des neuen ToS gespeichert wird. Danach wird ein Element gelesen, indem die Speicherzelle, auf die der ToS zeigt, gelesen wird und danach der ToS verringert wird. Der Stack kann byte-, halbwort-, oder wortorientiert sein, wodurch sich ergibt, dass push und pop den ToS in eine byte-adressierbaren Speicher jeweils um 1, 2 oder 4 verändern.


Speicher
ToS
BoS
Speicher
Nach
dreimal
push
ToS
BoS
Speicher
Nach
pop
ToS
BoS

Stacks werden in der Informatik an vielen Stellen verwendet. Ein Beispiel dafür finden wir in der Assembler-Programmierung, wenn wir geschachtelte Unterprogramme implementieren sollen, die von unterschiedlichen Stellen aufgerufen werden sollen.

Wir müssen uns dann nämlich beim Aufruf eines Unterprogramms die Stelle im Programm merken, von der wir weggesprungen sind, um dorthin zurück gelangen zu können, wenn das Unterprogramm seine Berechnung ausgeführt hat. Dies geschieht zweckmäßigerweise mit einem JAL-Befehl (JAL steht für **J**ump **A**nd **L**ink). Dieser vollzieht einen Sprung an die angegebene Adresse und merkt sich den NPC, also die Speicherstelle des folgenden Befehls im Programmspeicher im Register R31. Am Ende des Unterprogramms kann man dann einfach mit einem JR R31 (JR steht für **J**ump **R**egister) an diese Stelle zurückspringen und somit kann das aufrufende Programm fortgesetzt werden.

Wenn nun aber R31 bereits belegt ist und ein neuer Aufruf eines Unterprogramms findet statt (z.B. weil ein Unterprogramm sich selbst aufruft), dann müssen die Inhalte der Register, die das aufrufende Programm benötigt, gerettet werden, um sie für den Rücksprung wieder herzustellen. Dies betrifft insbesondere das Register R31, das ja für den Rücksprung gebraucht wird.

Auch für andere Register kann die Rettung der Registerinhalte wichtig sein.
Im bereits vorgestellten Programm m_mod_n verwenden wir R1 bis R3.
Wenn wir nun aus diesem Programmstück ein Unterprogramm entwickeln wollen, dass von beliebigen Stellen aufgerufen werden kann, müssen wir uns Gedanken darüber machen, ob R1 bis R3 vom aufrufenden Programm anders verwendet wurden und die Registerinhalte von R1 bis R3 beim Aufruf des Unterprogramms zunächst gerettet und bei der Beendigung wieder hergestellt werden müssen.

Der Speicherbereich, in dem die Register gerettet werden, ist sinnvollerweise als Stack organisiert:

- Unmittelbar nach dem Aufruf des Unterprogramms werden die Registerinhalte mit wiederholten push-Operationen auf den Stack geschrieben.
- Unmittelbar vor dem Rücksprung aus dem Unterprogramm werden die Registerinhalte durch pop-Operationen vom Stack geholt und in den Registern wieder hergestellt.

Des weiteren kann mit Hilfe des Stacks die Parameterübergabe erfolgen:
Bei unserem Beispiel m_mod_n kann das aufrufende Programm die Parameter m und n auf den Stack pushen und zusätzlich einen Speicherplatz für das Ergebnis reservieren.
Das Unterprogramm kann dann die Parameter m und n vom Stack holen und am Ende des Unterprogramms das Ergebnis an die reservierte Stelle des Stacks schreiben.

Beispiel:

```
/Berechne R5:=(R2 mod R1) + (R4 mod R3) mit Hilfe
/ des angepassten Unterprogramms Up_m_mod_n.
/Voraussetzungen:
/ In R1 bis R4 stehen positive Integerwerte. Ergebnis < 2^31.
/ R5: Ergebnis
/ R6: Hilfsregister
/ R7 bis R29 werden nicht überschrieben
/ R30: ToS (=Top of Stack), zu Beginn ToS=1000
```

Hauptprogramm zum Unterprogramm Up_m_mod_n
/Programmbeschreibung und Registerbelegung s. vorangegangene Folie

```
Hp_Start:               /Start des Hauptprogramms
ADDI  R30,R0,#1000      /ToS wird mit 1000 initialisiert
SW    0(R30),R2         /Speichern der Übergabeparameter auf dem Stack
SW    4(R30),R1
SW    8(R30),R0         /Reservierung für das Ergebnis und Initialisierung
ADDI  R30,R30,#12       /Erhöhe ToS
JAL   Up_m_mod_n        /Berechne R2 mod R1
SUBI  R30,R30,#12       /Reduziere ToS
LW    R5, 8(R30)        /Hole das Ergebnis (R2 mod R1) vom Stack und speichere es in R5
SW    0(R30),R4         /Speichern der Übergabeparameter auf dem Stack
SW    4(R30),R3
SW    8(R30),R0         /Reservierung für das Ergebnis und Initialisierung
ADDI  R30,R30,#12       /Erhöhung ToS
JAL   Up_m_mod_n        /Berechne R4 mod R3
SUBI  R30,R30,#12       /Reduziere ToS
LW    R6, 8(R30)        /Hole das Ergebnis (R4 mod R3) vom Stack und speichere es in R6
ADD   R5,R5,R6          /Berechne das Gesamtergebnis R5=(R2 mod R1) + (R4 mod R3)
HALT                    /Ende des Hauptprogramms
```

Angepasstes Unterprogramm Up_m_mod_n (1. Teil)

```
Up_m_mod_n:       /Start des Unterprogramms
/Berechnet m modulo n für positive Integerwerte m und n.
/m und n werden vom Stack geholt und in R1 bzw. R2 gespeichert.
/R3 wird als Hilfsregister für Vergleiche verwendet.
/Am Programmende wird das Ergebnis auf den Stack geschrieben.
SW    0(R30),R1     /Retten der Registerinhalte R1 bis R3, die vom Unterprogramm
SW    4(R30),R2     /überschrieben werden.
SW    8(R30),R3
SW    12(R30),R31   /Rücksprungadresse sichern, falls innerhalb des Unterprogramms
                    /weitere Unterprogramme aufgerufen werden (könnte hier entfallen)
LW    R1,-12(R30)   /Holen der Übergabeparameter vom Stack
LW    R2,-8(R30)
ADDI  R30,R30,#16   /Erhöhen des Stackpointers
```

/Fortsetzung auf der nächsten Folie

Angepasstes Unterprogramm Up_m_mod_n (2. Teil)

```
m_mod_n:
SLT     R3,R1,R2      /m<n?
BNEZ    R3, Fertig    /Falls m<n steht in R1 das Ergebnis m modulo n
SUB     R1,R1,R2      /m soll solange um n reduziert werden, bis m<n
J       m_mod_n
Fertig:
SUBI    R30,R30,#16   /Reduziere Stackpointer
SW      -4(R30),R1    /Speichere das Ergebnis auf dem Stack
LW      R1,0(R30)     /Hole gerettete Registerinhalte vom Stack
LW      R2,4(R30)
LW      R3,8(R30)
LW      R31,12(R30)   /Hole Rücksprungadresse vom Stack (könnte hier entfallen, s.o.)
JR      R31           /Rücksprung zum aufrufenden Programm
```

Ein weiteres Beispiel für ein Programm mit Stack sehen wir in dem folgenden Programm zur Berechnung der Funktion **Summe(n)**, welches die Summe der Zahlen von 1 bis n berechnen soll für eine natürliche Zahl n. Die Berechnung von Summe(n) erfolgt rekursiv:

– Wenn n=1 ist, wird das Ergebnis auf 1 gesetzt.
– Wenn n>1 ist, wird das Ergebnis als n+Summe(n-1) ermittelt, wobei Summe(n-1) mit demselben Unterprogramm berechnet wird.

```
int Summe (int n)
{           if (n==1)
                        return (1)
            else
                        return (n + Summe(n - 1))
}
```

Wie in dem vorangegangenen Beispiel wird der Stack verwendet für

- Übergabe der Funktionsparameter an das Unterprogramm
- sichern und wiederherstellen der lokalen Registerinhalte einschließlich der Rücksprungadresse
- Übergabe des Ergebnisses an das aufrufende Programm

Das Unterprogramm Summe wird zunächst für k=n aufgerufen, falls k=1 ist, wird das Ergebnis auf 1 gesetzt und zum aufrufenden Programm zurückgesprungen.
Falls k>1 ist, wird das Unterprogramm Summe mit k-1 erneut aufgerufen.
Dies wird solange wiederholt, bis schließlich das Unterprogramm Summe mit k=1 aufgerufen wird.
Bei jedem Aufruf des Unterprogramms Summe werden jedes Mal wieder alle oben genannten Daten auf den Stack geschrieben, so dass der verwendete Stackbereich immer weiter anwächst.
Erst wenn k=1 erreicht ist, wird mit der Addition begonnen und der ToS wieder reduziert.

Die maximale Größe des Stacks kann häufig erst zur Laufzeit des Programms ermittelt werden, da man erst dann feststellt, wie oft solche rekursiven Aufrufe ausgeführt werden. Man spricht daher von einem Laufzeitstack, der im Speicher so angelegt werden muss, dass er auch im schlechtesten Fall genügend freie Speicherzellen hat, um das Programm korrekt auszuführen.

Da das Hüser-Tool ein Browser-basiertes Simulationstool ist, das von einem ehemaligen Kommilitonen entwickelt wurde, sollten Sie Programme dort zunächst nur mit kleinen Stackgrößen testen und Kopien Ihrer Programme auch außerhalb des Tools sichern.

```
int Summe (int n)
{           if (n==1)
                    return (1)
            else
                    return (n + Summe(n - 1))
}
```

Registerbelegung:

R1:	**n**
R2:	**n-1**
R3:	**Hilfsvariable**
R4:	**Summe(n-1)**
R5:	**Endergebnis Summe(n)**
R30:	**ToS = aktueller Wert des Stackpointer**
R31:	**Rücksprungadresse**

Hauptprogramm zur Berechnung von Summe(n)
/Programmbeschreibung und Registerbelegung s. vorangegangene Folien

Hauptprogramm:

```
ADDI  R2, R0, #42      / Initialiserung zu Demozwecken
ADDI  R3, R0, #24      / Initialiserung zu Demozwecken
ADDI  R4, R0, #66      / Initialiserung zu Demozwecken
ADDI  R30, R0, #1004 / ToS initialisieren mit 1004
SW    0(R30), R1       / n auf Stack pushen
ADDI  R30, R30, #4     / ToS erhöhen
SW    0(R30), R0       / Reservierung für das Ergebnis
ADDI  R30, R30, #4     / ToS erhöhen
JAL   Summe            / Summe(n) wird berechnet
SUBI  R30, R30, #4     / ToS reduzieren
LW    R5, 0(R30)        / Ergebnis vom Stack holen
SUBI  R30, R30, #4     / ToS reduzieren wegen n
SW    1000(R0),R5      / Ergebnis an Adresse 1000 speichern
HALT                   / Programmende
```

Unterprogramm zur Berechnung Summe(n) (1. Teil)

```
Summe:
SW      0(R30),R1       / R1 bis R4 retten, die vom Unterprogramm überschrieben werden.
ADDI    R30, R30, #4    / ToS erhöhen
SW      0(R30),R2
ADDI    R30, R30, #4
SW      0(R30),R3
ADDI    R30, R30, #4
SW      0(R30),R4
ADDI    R30, R30, #4
SW      0(R30), R31     / Rücksprungadresse auf den Stack pushen
ADDI    R30, R30, #4    / ToS erhöhen
LW      R1, -28(R30)    / Hol Übergabeparameter n vom Stack
SNEI    R3, R1, #1      / Wenn n<>1 ist,
BNEZ    R3, Rekursion   /  gehe zur Rekursion
ADDI    R4, R0, #1      / Sonst setze Ergebnis = 1
J       return          / und beende das Unterprogramm
```

/Fortsetzung auf der nächsten Folie

Unterprogramm zur Berechnung Summe(n) (2. Teil)

```
Rekursion:
SUBI   R2, R1, #1     / k := n - 1
SW     0(R30), R2     / k auf Stack pushen
ADDI   R30, R30, #4   / ToS erhöhen
SW     0(R30), R0     / Reservierung für das Ergebnis
ADDI   R30, R30, #4   / ToS erhöhen
JAL    Summe          / rekursiver Aufruf
SUBI   R30, R30, #4   / ToS reduzieren
LW     R4, 0(R30)     / Summe(n-1) vom Stack holen und in R4 speichern
SUBI   R30, R30, #4   / ToS reduzieren wegen k
ADD    R4, R1, R4     / Summe := n + Summe(n-1)
```

/Fortsetzung auf der nächsten Folie

Unterprogramm zur Berechnung Summe(n) (3. Teil)

```
return:
SW      -24(R30),R4   / Speichere Ergebnis auf dem Stack
SUBI    R30, R30, #4  / Reduziere ToS
LW      R31, 0(R30)   / Rücksprungadresse vom Stack holen
SUBI    R30, R30, #4  / R1 bis R4 wieder herstellen.
LW      R4, 0(R30)
SUBI    R30, R30, #4
LW      R3, 0(R30)
SUBI    R30, R30, #4
LW      R2, 0(R30)
SUBI    R30, R30, #4
LW      R1, 0(R30)
JR      R31           / Rücksprung zum aufrufenden Programm
```

Bei dem vorangegangenen Beispiel wird der Stack verwendet für

- Übergabe der Funktionsparameter an das Unterprogramm
- sichern und wiederherstellen der lokalen Registerinhalte einschließlich der Rücksprungadresse
- Übergabe des Ergebnisses an das aufrufende Programm

Dies hat den Vorteil, dass keine globalen Variablen verwendet werden, wodurch Hauptprogramm und Unterprogramm relativ unabhängig voneinander sind.
Nachteilig ist jedoch, dass das Programm umfangreicher wird.

Deshalb wird auf der folgenden Folie eine wesentlich kürzere Alternative vorgestellt:

Rekursives Programm zur Berechnung Summe(n) ohne Stack

```
/Registerbelegung
/R1: n
/R2: Laufvariable i
/R3: Hilfsvariable
/R4: Ergebnis

HP_Start:
ADD   R2, R1, R0    / Initialisiere Laufvariable i mit n
ADD   R4, R0, R0    / Initialisiere Ergebnis mit 0
J     Summe         / Aufruf des Unterprogramms Summe

Summe:
SEQI  R3, R2, #0    / Laufvariable i==0 ?
BNEZ  R3, Fertig    /  dann fertig.
ADD   R4, R4, R2    / Addiere i zum Ergebnis
SUBI  R2, R2, #1    / Reduziere Laufvariable i um 1
J     Summe         / Erneuter Aufruf des Unterprogramms Summe

Fertig:
SW    1000(R0),R4   / Speichere Ergebnis an Adresse 1000
HALT
```

Beide Realisierungen der Berechnung der Summe von 1 bis n haben ihre Berechtigung und auch weitere Realisierungen sind denkbar, beispielsweise die folgende iterative:

```
/Registerbelegung s. vorangegangene Folie
ADD   R2, R0, R0      / R2 = 0, Zaehler
ADD   R4, R0, R0      / R4 = 0, Summe

loop:
ADD   R4, R4, R2      / Aufsummieren
ADDI  R2, R2, #1      / Zähler erhöhen
SLE     R3, R2, R1    / (Zaehler <= n)?
BNEZ  R3, loop        /   wiederhole ggf. die Schleife
SW      1000(R0), R4 / Speichere das Ergebnis an Adresse 1000
HALT
```

Bei dem folgenden Beispiel zum Selbststudium wird nur ein Teil der Parameter mit Hilfe des Stacks übergeben. Dies kann effizient sein, erfordert aber auch besondere Sorgfalt.

```
int fibonacci (int i)
{       if (i<2)
                return (1)
        else
                return (fibonacci (i-1)+fibonacci(i-2))
}
```

Registerbelegung:

R1:	**i, wobei die i-te Fibonacci Zahl berechnet werden soll**
R2:	**i-te Fibonacci-Zahl**
R3:	**Hilfsvariable**
R4:	**Hilfsvariable**
R5:	**Stackpointer (ToS)**
R31:	**Rücksprungadresse**

Hauptprogramm:

```
/Berechnet die i-te Fibonacci-Zahl F(i).
/Voraussetzungen:
/Zu Beginn steht i an Adresse 1512. i>=1, F(i) < 2^31.
/Am Ende soll F(i) an Adresse 1516 geschrieben werden.
/R1:        i, globale Variable
/R2:        F(i), globale Variable
/R3, R4:    Hilfsregister
/R30:       ToS (=Top of Stack), zu Beginn ToS=1000
/R31:       Rücksprungadresse

Hp_Start:                   / Start des Hauptprogramms
ADDI   R30, R0, #1000       / Initialisieren ToS
LW     R1, 1512(R0)         / Lesen der Zahl i
ADD    R3, R0, R0           / Initialisieren von R3
ADD    R4, R0, R0           / Initialisieren von R4
JAL    Fibonacci            / Sprung ins Unterprogramm Fibonacci
Rück1:                      / Label nur für Demonstrationszwecke
SW     1516(R0), R2         / Ergebnis wegschreiben
HALT                        / Ende des Programms
```

```
Fibonacci:                / Beginn des Unterprogramms
SW      0(R30), R3        / Retten von R3
ADDI    R30, R30, #4      / Hochzählen des ToS
SW      0(R30), R4        / Retten von R4
ADDI    R30, R30, #4      / Hochzählen des ToS
SW      0(R30), R31       / Retten von R31
ADDI    R30, R30, #4      / Hochzählen des ToS
ADDI    R2, R0, #1        / Ergebnis:= 1 (prophylaktisch, falls i<=2)
SLEI    R3, R1, #2        / (i<=2)?
BNEZ    R3, Ende          / Herausspringen, falls i<=2
ADD     R3, R1, R0        / Retten von i
SUBI    R1, R1, #1        / i: = i-1
JAL     Fibonacci         / Berechnen von F(i-1) in R2
Rück2:                    / Label nur für Demonstrationszwecke
ADD     R4, R0, R2        / Retten des Ergebnisses F(i-1) nach R4
SUBI    R1, R3, #2        / Bilden von i-2
JAL     Fibonacci         / Berechnen von F(i-2) in R2
Rück3:                    / Label nur für Demonstrationszwecke
ADD     R2, R4, R2        / F(i): = F(i-1) + F(i-2)
Ende:
LW      R31, -4(R30)      / Zurückholen von R31 vom Stack
SUBI    R30, R30, #4      / Herunterzählen den ToS
LW      R4, -4(R30)       / Zurückholen von R4 vom Stack
SUBI    R30, R30, #4      / Herunterzählen den ToS
LW      R3, -4(R30)       / Zurückholen von R3 vom Stack
SUBI    R30, R30, #4      / Herunterzählen den ToS
JR      R31               / Rücksprung
```