# Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

02. Februar 2016

# Globale Register Allokation

Idee: Ein Teil der Register global, also über mehrere einfache Blöcke hinweg, einzelnen häufig benutzten Variablen zuordnet. Meist werden ein paar Register für die Variablen in inneren Schleifen reserviert und dann für diese Schleife fest zugeordnet.

- Der Programmier bestimmt die Variablen, etwa über eine `register`-Anweisung (in C möglich)
- Der optimierende Compiler bestimmt die Variablen.

# Usage Counts

Idee: Abschätzung des Zeitgewinns für jede in einer inneren Schleife $L$ auftretenden Variablen $v$, den man durch eine feste Zuordnung dieser Variablen zu einem Register erzielen kann.

Wenn wir die Variable $v$ für die Schleife $L$ einem Register `Rv` fest zuordnen, erhalten wir eine Kostenersparnis von 2 für jeden Gebrauch von $v$ in einem Block $B$ von $L$, in dem $v$ nicht vorher definiert wurde. Ausserdem sparen wir auch noch Kosten von 3, wenn $v$ am Ende des Blocks $B$ lebendig ist, da wir auf das Abspeichern des Werts von $v$ mit `ST v,Ri` verzichten können.

Wird also $v$ für die Schleife $L$ fest einem Register zugeordnet, sparen wir etwa

$$\sum_{B in L} 2 \cdot \mathrm{use}(v, B) + 3 \cdot \mathrm{live}(v, B)$$

Kosten, wobei

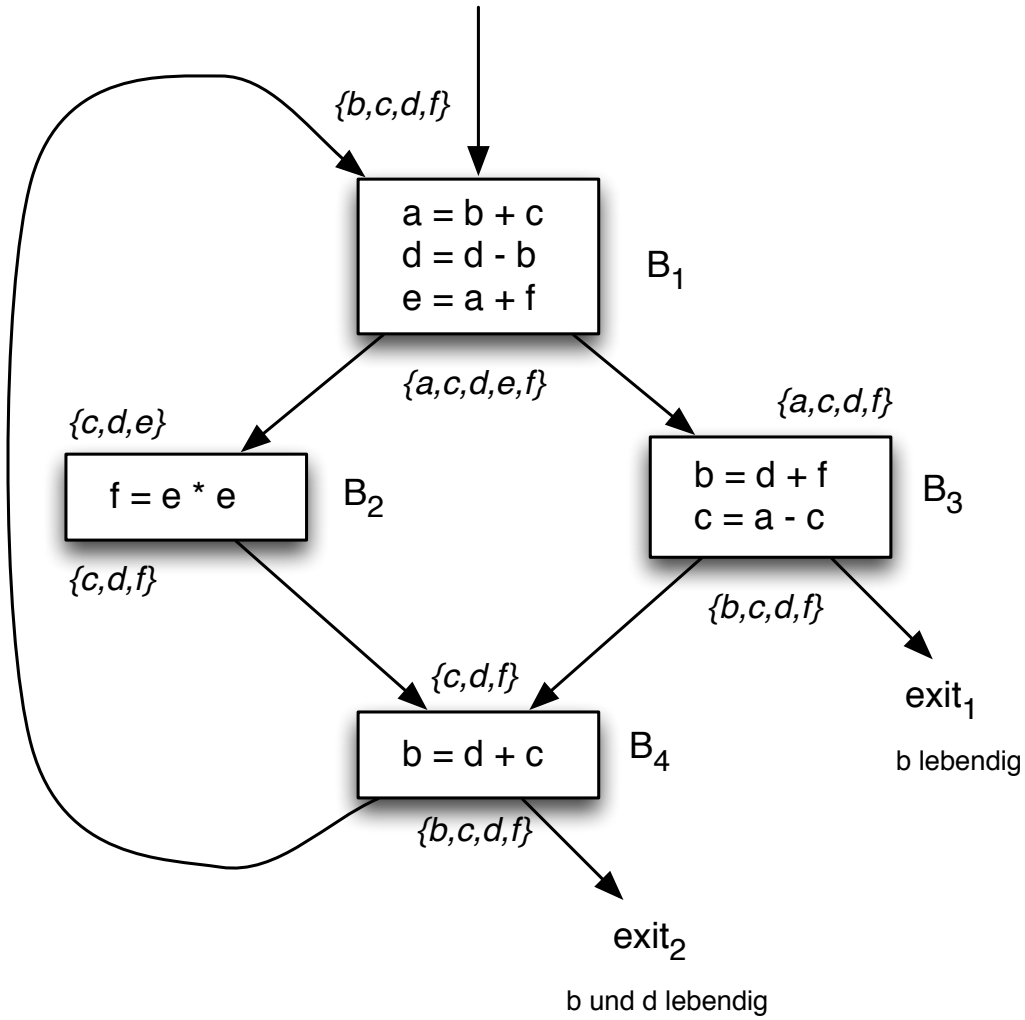
- $\mathrm{use}(v, b)$ die Anzahl des Gebrauchs von $v$ in $B$ vor einer Definition von $v$ in $B$ und
- $\mathrm{live}(v, B) =$
  $\begin{cases} 1 & \text{falls } v \text{ in } B \text{ definiert und lebendig am Ende von } B \\ 0 & \text{sonst.} \end{cases}$

Dieser Wert ist natürlich nur eine grobe Näherung der eingesparten Kosten.

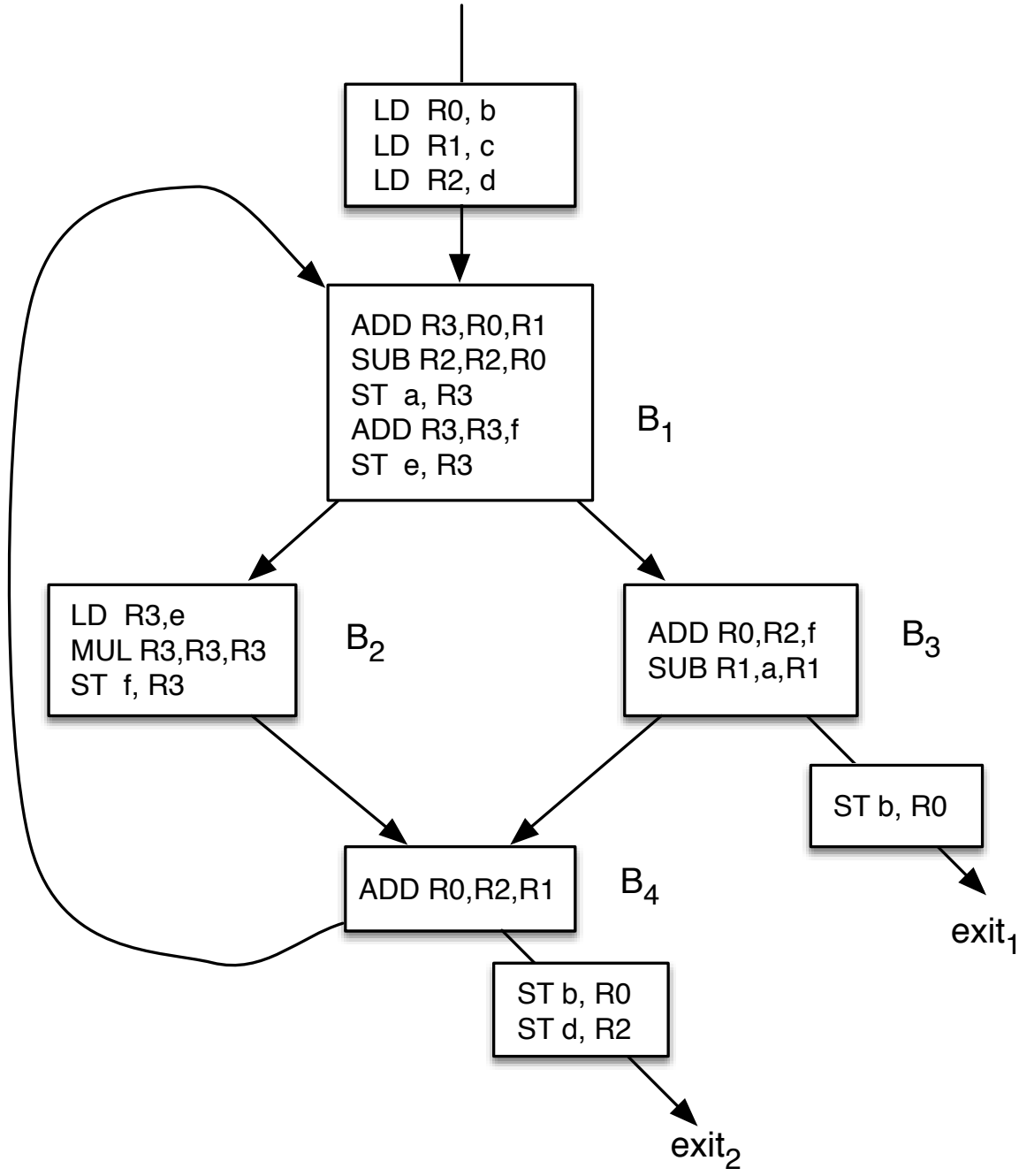
**Bemerkung:** Diese Strategie zur festen Zuordnung von Registern zu Variablen lässt sich natürlich auch für eine $L$ umfassende Schleife $L'$ verallgemeinern, wobei eventuell Registerwerte beim Verlassen von $L$ gespeichert werden müssen.

# Beispiel

Gegeben sei der folgende Flußgraph einer Schleife, bei dem die Sprungbefehle und Label weggelassen wurden. Die lebendigen Variablen am Blockanfang und -ende sind jeweils angegeben.



Wenn man jetzt mit dieser Information die Kostenersparnis für die Variable $a$ berechnen will, stellt man fest, dass $a$ nur in $B_3$ ohne vorige Definition benutzt wird und nur am Ende von $B_1$ lebendig ist. Also wäre die Ersparnis 5. Die entsprechenden Werte für $b$, $c$, $d$. $e$ und $f$ sind 10, 9, 9, 7 und 7. Hat man etwa die drei Register `R0`, `R1` und `R2` zur festen Zuordnung zur Verfügung und das Register `R3` zur freien Verfüging, so könnt man die ersten drei Register den Variablen $b$, $c$ und $d$ fest zuordnen. Dann würde sich mit der Maschinencode-Erzeugung aus dem vorigen Abschnitt folgendes Programmfragment ergeben:


LD R0, b
LD R1, c
LD R2, d
ADD R3,R0,R1
SUB R2,R2,R0
ST a, R3
ADD R3,R3,f
ST e, R3
B1
LD R3,e
MUL R3,R3,R3
ST f, R3
B2
ADD R0,R2,f
SUB R1,a,R1
B3
ST b, R0
exit1
ADD R0,R2,R1
B4
ST b, R0
ST d, R2
exit2

# Register Allokation durch Graph-Färbung

Wenn für eine Berechnung ein Register benötigt wird, aber im Moment alle Register in Gebrauch sind, also Werte von Variablen enthalten, die noch benötigt werden, muss ein Register ausgewählt werden und dessen Inhalt in den Speicher zurückgeschrieben werden (*spill*).
Eine systematische Methode, für einen Flussgraphen Register zu allokieren und das Zurückschreiben zu organisieren, beruht auf der Färbung von Graphen.

Diese Methode läuft in zwei Phasen ab. In der Phase 1 wird der Zwischencode so Maschinenbefehle übersetzt, als ob eine beliebige Zahl von (symbolischen) Registern zur Verfügung stehen würde. Die Namen von Variablen werden häufig einfach als Registerbezeichnungen interpretiert.

In einer zweiten Phase werden die vorhandenen freien Register den symbolischen Registern zugeordnet, wobei das Ziel ist, die Anzahl der Rückspeicherungen von Werten möglichst klein zu halten.

- Zunächst bestimmt man für jeden Programmpunkt die lebendigen Variablen.
- Als nächstes konstruiert man den Register Inference Graphen (RIG).
  Dies ist ein ungerichteter Graph, der als Knotenmenge die symbolischen Register hat.
- Zwei Knoten sind verbunden, wenn an einem Punkt im Programm beide symbolischen Register gleichzeitig lebendig sind.

Folgerung: die minimale Zahl von Registern, die notwendig ist um den Flussgraphen oder Zurückspeichern auszuwerten, ist gleich der minimalen Zahl von Farben, mit denen man den RIG zulässig färben kann.

- Hat man also $k$ Register zur Verfügung, kann man eine Register-Allokation ohne Zurückspeicherung finden, falls der RIG $G$ $k$-färbbar ist.
- Leider ist die Frage, ob ein gegebener Graph mit $k$ Farben zulässig zu färben ist, NP-Vollständig.

# Heuristischer Färbe-Algorithmus

1. Hat ein Knoten $n$ in $G$ weniger als $k$ Nachbarn, entfernt man diesen Knoten und die anhängenden Kanten aus $G$. Man erhält so einen Graphen $G'$. Ist $G'$ $k$-färbbar, so kann man diese Färbung leicht zu einer $k$-färbung für $G$ erweitern.
2. Erhält man durch Iteration dieses Schritts einen leeren Graphen, so kann man eine k-Färbung vom Originalgraphen $G$ konstruieren, indem man die entfernten Knoten in umgekehrter Reihenfolge ihrer Entfernung wieder einfügt und färbt.

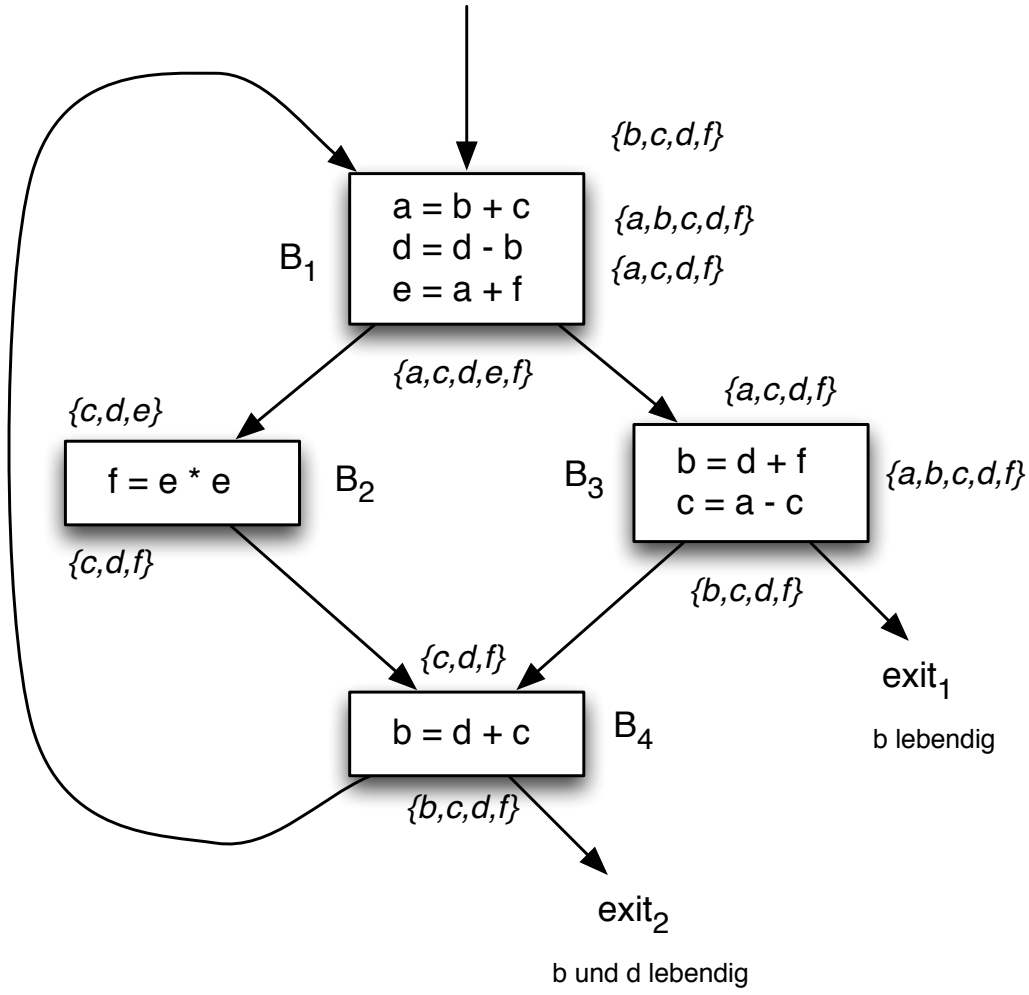
3. Erhält man durch Iteration von Schritt 1 einen Graphen $\tilde{G}$, in dem jeder Knoten $\geq k$ Nachbarn, muss man ein Register durch Zurückspeichern frei machen.
   - wähle eines der symbolischen Register nach heuristischen Regeln aus.
   - füge eine Ladeoperation an jeder Stelle im Programm ein, an der diese Variable gebraucht wird.
   - füge eine Speicheroperation an jeder Stelle im Programm ein, an der diese Variable definiert wird.
   - aktualisiere die Information über die Lebendigkeit der Variablen und damit den RIG

   Dann iteriert man das Verfahren weiter.

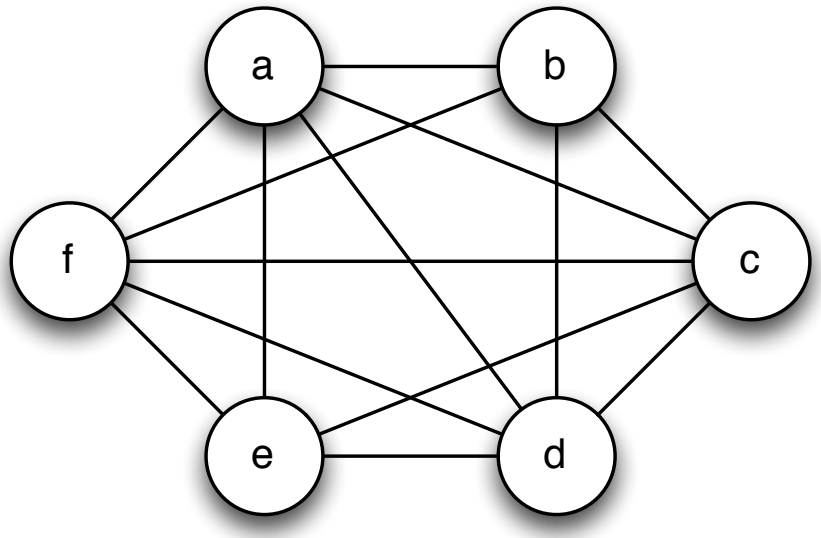
## Beispiel

Betrachten wir den Flussgraph aus dem vorigen Beispiel. Es stehen die vier Register `R0`, `R1`, `R2` und `R3` zur Verfügung. In unserem Fall können die Variablennamen direkt als Namen der symbolischen Register benutzt werden. Für jeden Punkt im Flussgraph bestimmt man die lebendigen Variablen.

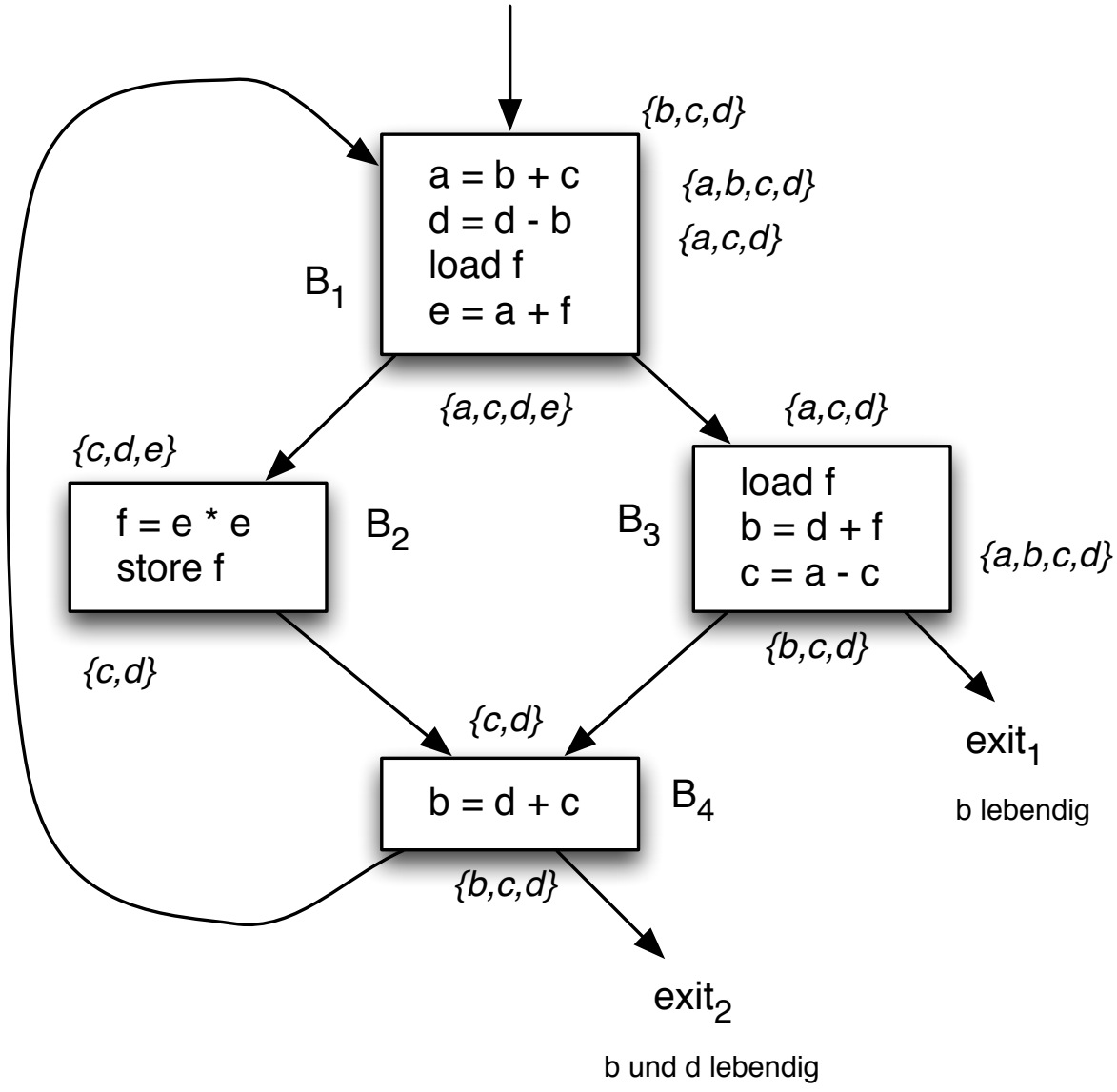
Man erhält:


{b,c,d,f}
a = b + c
d = d - b
e = a + f
B1
{a,b,c,d,f}
{a,c,d,f}
{a,c,d,e,f}
{a,c,d,f}
{c,d,e}
f = e * e
B2
B3
b = d + f
c = a - c
{a,b,c,d,f}
{c,d,f}
{b,c,d,f}
{c,d,f}
exit1
b lebendig
b = d + c
B4
{b,c,d,f}
exit2
b und d lebendig

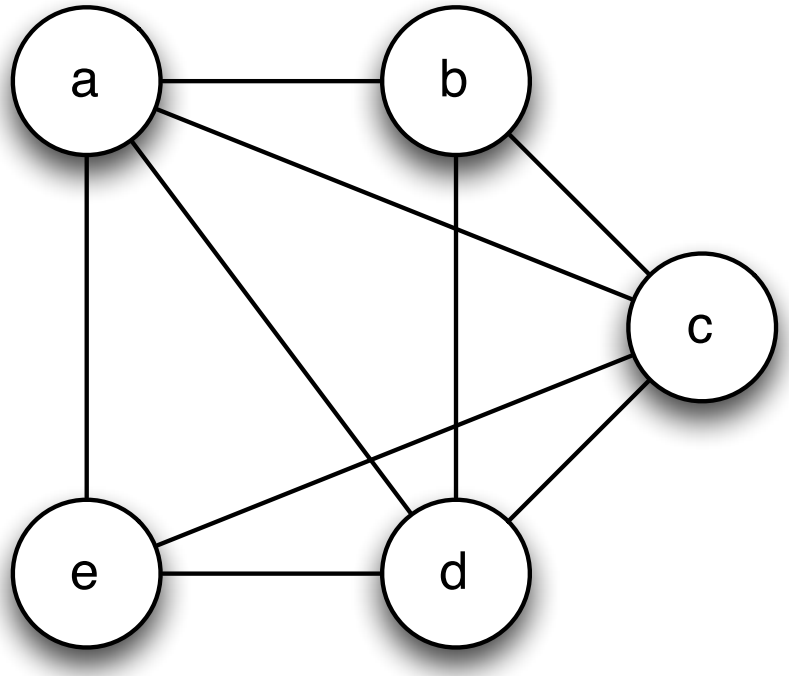
Mit diesen Informationen konstruiert man den Register Inference Graph (RIG):



Nun beginnt die zweite Phase, also die Zuweisung der vier Register zu den Variablen. Wie man sofort sieht, gibt es keinen Knoten mit weniger als vier Nachbarn, also muss eine der Variablen quasi „im Speicher leben“.
Wählt man zum Beispiel die Variable $f$ aus, da sie nur dreimal im Flussgraphen auftritt und viele Nachbarn hat, erhält man den folgenden geänderten Flussgraphen:

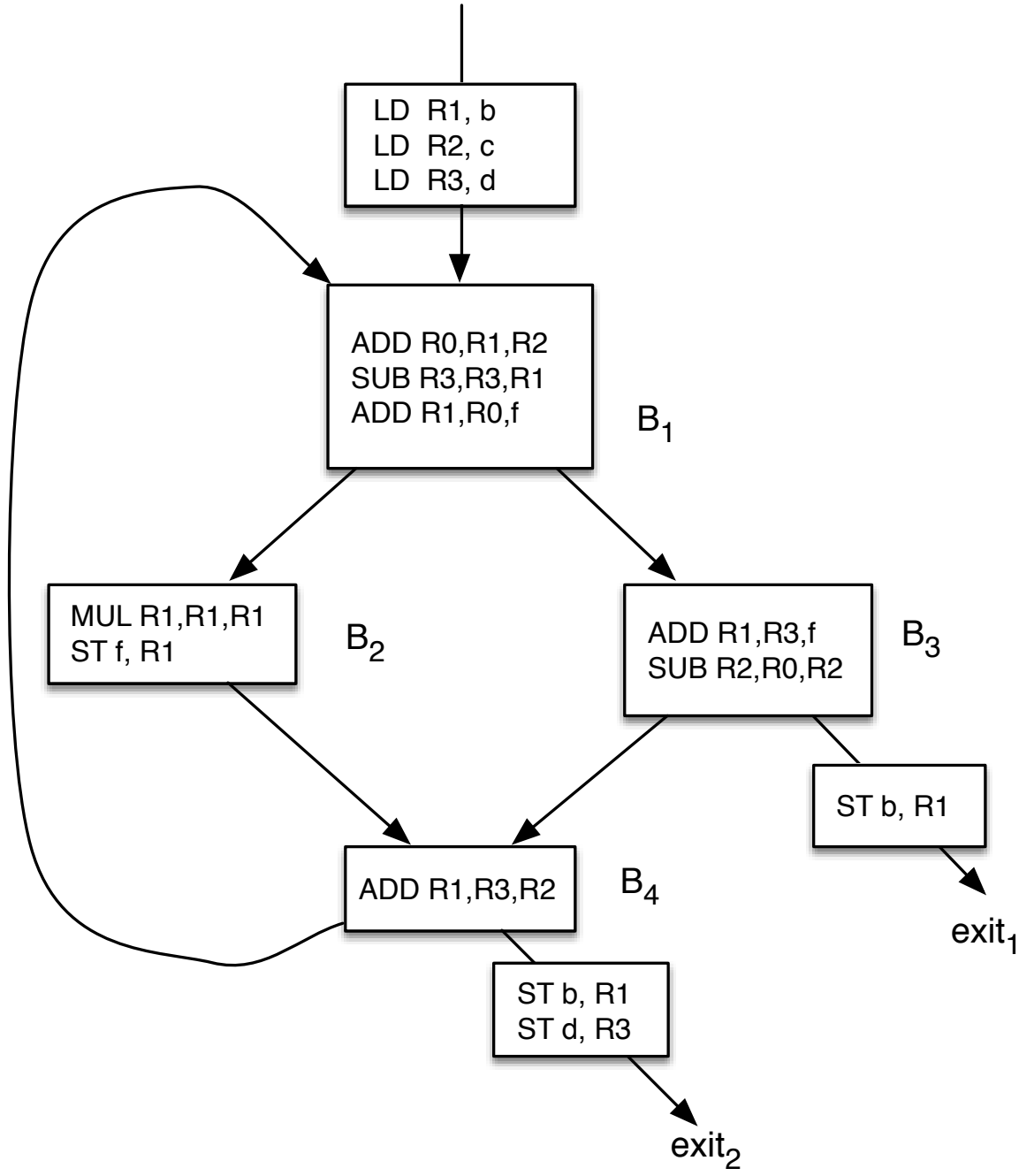

{b,c,d}
a = b + c
d = d - b
load f
e = a + f
{a,b,c,d}
{a,c,d}
B1
{a,c,d,e}
{a,c,d}
{c,d,e}
f = e * e
store f
B2
B3
load f
b = d + f
c = a - c
{a,b,c,d}
{c,d}
{b,c,d}
{c,d}
exit1
b lebendig
b = d + c
B4
{b,c,d}
exit2
b und d lebendig

und den folgenden RIG:



Nun können wir den Knoten $e$ auswählen, da er nur 3 Nachbarn besitzt. Dann könnte man die Knoten $d$, $a$, $b$ und $c$ (in dieser Reihenfolge) entfernen und landen beim leeren Graphen. Also ist dieser RIG 4-färbbar.
Man ordnet den Knoten in umgekehrter Reihenfolge jetzt Farben zu. Für die vier Knoten $a$, $b$, $c$ und $d$ benötigt man jeweils eine andere Farbe; der Knoten $e$ kann aber in der gleichen Farbe gefärbt werden, die Knoten $b$ hat.
Da die Farben Registern entsprechen, bedeutet dies, dass ein Register die Wert der Variablen $b$ und $e$ nacheinander ohne störende Interaktionen aufnehmen kann.

Ordnen wir der Variablen *a* das Register R0, *c* entsprechend R2, *d* R3 und den Variablen *b* und *e* das Register R1 zu und übersetzen die Lade- und Speicheroperationen durch entsprechende Adressierung, so erhält man folgendes Programmfragment:


LD R1, b
LD R2, c
LD R3, d
ADD R0,R1,R2
SUB R3,R3,R1
ADD R1,R0,f
B1
MUL R1,R1,R1
ST f, R1
B2
ADD R1,R3,f
SUB R2,R0,R2
B3
ST b, R1
exit1
ADD R1,R3,R2
B4
ST b, R1
ST d, R3
exit2