

7.5 Globale Register Allokation

Da die Geschwindigkeit des Prozessors häufig um Größenordnungen schneller als die des Speichers ist, spielt die effiziente Nutzung der Register eine wichtige Rolle bei der Maschinencode-Erzeugung in einem optimierenden Compiler. Unser einfacher Maschinencode-Erzeuger aus dem vorigen Abschnitt hat aus diesem Grund die berechneten Werte so lange wie möglich in den Registern belassen. Allerdings werden diese Werte am Ende eines einfachen Blocks immer in den Speicher zurückgeschrieben. Es stellt sich somit die Frage, ob man nicht besser einen Teil der Register global, also über mehrere einfache Blöcke hinweg, einzelnen häufig benutzten Variablen zuordnet.

Welche Variablen einem Register über Blockgrenzen hinweg zugeordnet werden soll, kann bei einigen Programmiersprachen der Programmierer bestimmen, etwa durch die Deklaration einer Variablen mit der `register`-Anweisung in C, die ein Hinweis für den Compiler ist, falls möglich der deklarierten Variablen ein freies Register zuzuordnen. Heutzutage wird diese Aufgabe aber besser durch optimierende Compiler gelöst. Meist werden ein paar Register für die Variablen in inneren Schleifen reserviert und dann für diese Schleife fest zugeordnet. Die Frage bleibt natürlich, welche Variablen auf diese Weise einem Register zugeordnet werden sollen.

7.5.1 Usage Counts

Diese Methode beruht auf der Idee, für alle in einer inneren Schleife L auftretenden Variablen v , den Zeitgewinn abzuschätzen, den man durch eine feste Zuordnung dieser Variablen zu einem Register erzielen kann. Wir wollen annehmen, dass die Maschinencode-Erzeugung für einen einfachen Block wie im vorigen Abschnitt durchgeführt wird und wir wollen die Abarbeitungszeit der Befehle wie im Abschnitt 7.2 durch Kosten approximieren.

Wenn wir die Variable v für die Schleife L einem Register R_v fest zuordnen, erhalten wir (für unser Maschinenmodell) eine Kostenersparnis von 2 für jeden Gebrauch von v in einem Block B von L , in dem v nicht vorher definiert wurde (statt `ADD ...,v,...` hätte man den Befehl `ADD ...,Rv,...`). 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

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

Kosten, wobei

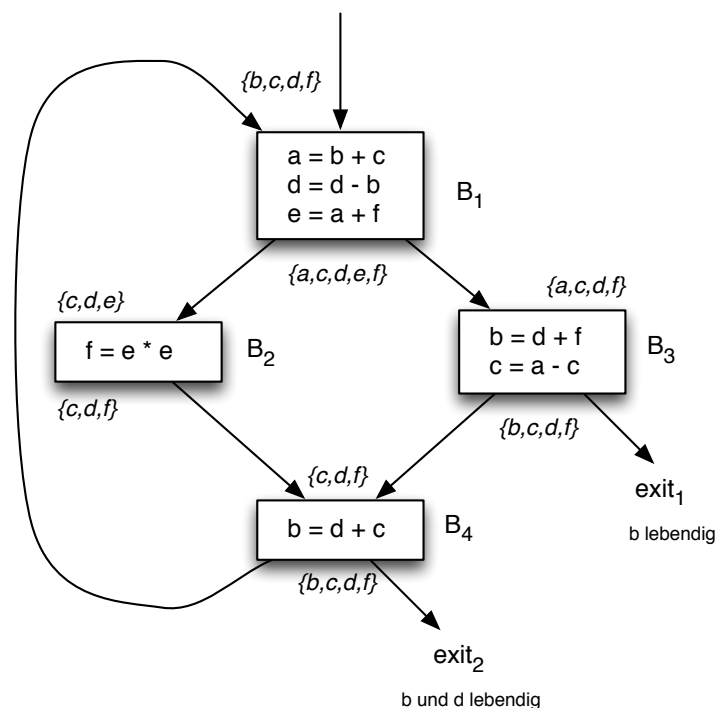
- $\text{use}(v, b)$ die Anzahl des Gebrauchs von v in B vor einer Definition von v in B und
- $\text{live}(v, B) = \begin{cases} 1 & \text{falls } v \text{ in } B \text{ definiert wurde und lebendig am Ende von } B \text{ ist} \\ 0 & \text{sonst.} \end{cases}$

Dieser Wert stellt natürlich nur eine grobe Näherung der eingesparten Kosten dar, weil verschiedenen Blöcke in der Schleife unterschiedlich häufig durchlaufen werden können. Auch ignorieren wir den Aufwand für das Laden und Speichern von v beim Eintritt und Austritt aus der Schleife, da dieser Aufwand nur einmal anfällt, wogegen wir annehmen können, dass die Blöcke in der Schleife häufiger durchlaufen werden.

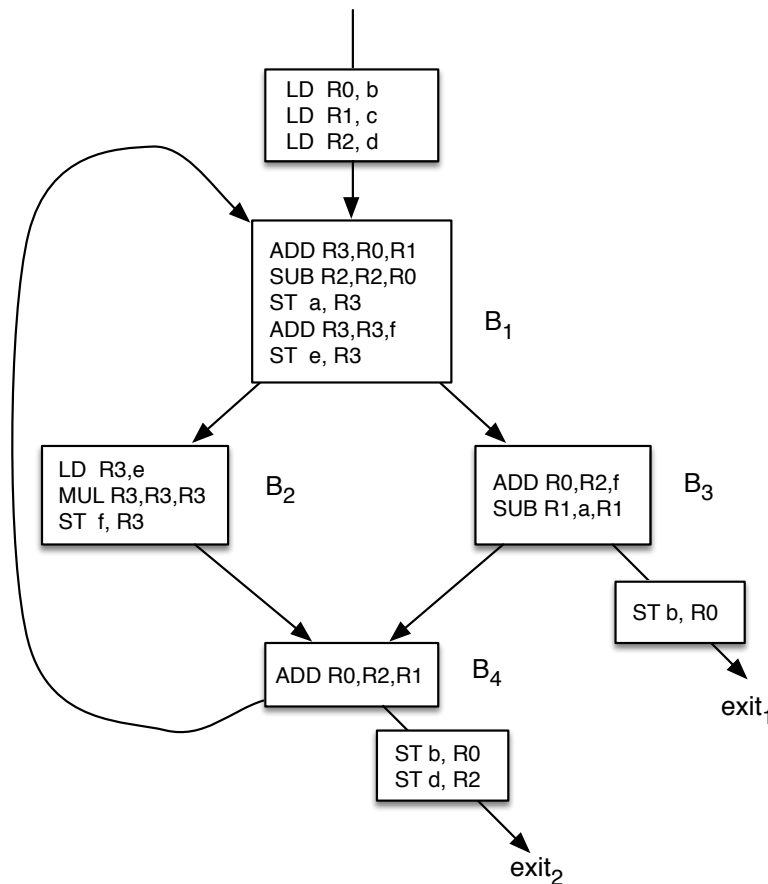
Bemerkung: Diese Strategie zur festen Zuordnung von Registern zu Variablen lässt sich natürlich auch für eine L umfassende Schleife L' verallgemeinern, wobei eventuell Registerwerte beim Verlassen von L gespeichert werden müssen.

Beispiel 7.9:

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 Verfügung, so könnte 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 (die Marken und Sprungbefehle sind wieder nicht angegeben):



7.5.2 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 der Inhalt in den Speicher zurückgeschrieben werden (*spill*). Eine systematische Methode, für einen Flussgraphen Register zu allokalieren 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 einfach als Registerbezeichnungen interpretiert und meist kann man die Drei-Adress-Befehle direkt als Maschinenbefehle interpretieren. Falls ein Zugriff auf Werte über spezielle Register, wie etwa über den Stackpointer oder Framepointer geschieht, nehmen wir an, dass es spezielle Register für diese Aufgaben gibt, die nicht weiter betrachtet werden.

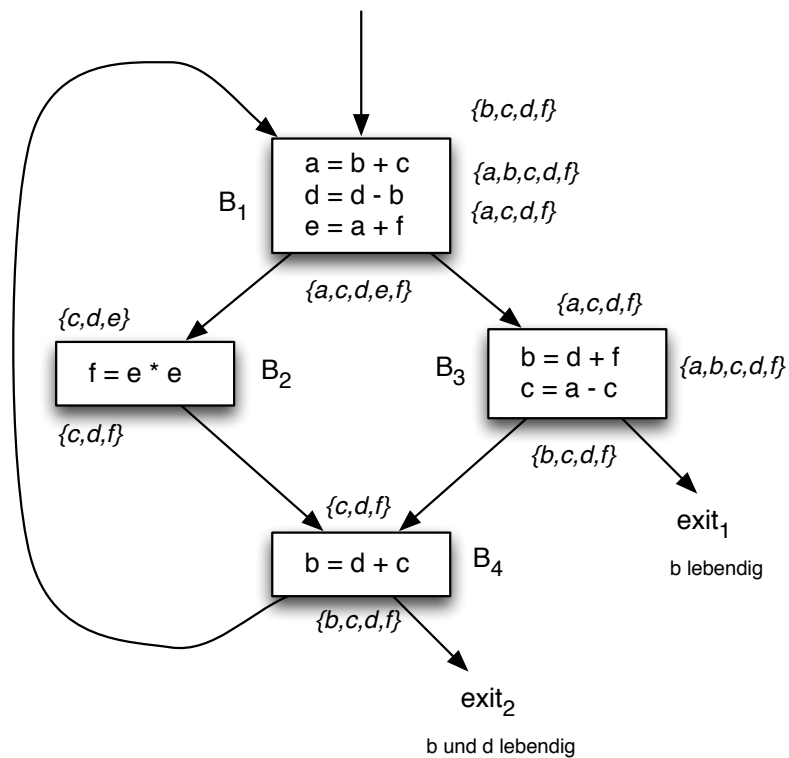
In einer zweiten Phase werden die vorhandenen freien Register den symbolischen Registern zugeordnet, wobei das Ziel ist, die Anzahl der Rückspeicherungen von Werten möglichst klein zu halten. Zunächst bestimmt man für jeden Programmpunkt die lebendigen Variablen. Als nächstes konstruiert man den Register 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. Das bedeutet, dass die Werte dieser beiden symbolischen Register nicht dem gleichen realen Register zugeordnet werden können.

Damit ist aber klar, dass die minimale Zahl von Registern, die notwendig ist um den Flussgraphen oder Zurückspeichern auszuwerten, gleich der minimalen Zahl von Farben ist, mit denen man den RIG zulässig (d.h. verbundene Knoten haben unterschiedliche Farben) 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. Aber es gibt eine in der Praxis häufig erfolgreich eingesetzte heuristische Näherung für dieses Problem.

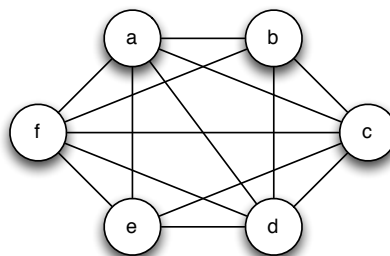
- 1) Hat ein Knoten n in G weniger als k Nachbarn, entfernt man diesen Knoten und die anhängenden Kanten aus G . Man erhält so einen Graphen G' . Ist G' k -färbbar, so kann man diese Färbung leicht zu einer k -färbung für G erweitern, in dem man den Knoten n mit einer der Farben färbt, die kein Nachbar von n hat.
- 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. Man wählt eines der symbolischen Register nach heuristischen Regeln aus, etwa eine Variable, die nur wenig gebraucht wird. Man sollte aber vermeiden, in inneren Schleifen eine Zurückspeicherung einzufügen. An jeder Stelle im Programm, an der diese Variable gebraucht wird, fügt man eine Lade-Operation ein und an jeder Stelle, an der diese Variable definiert wird, fügt man eine Speicher-Operation ein. Dann wird die Information über die Lebendigkeit der Variablen und damit ein RIG aktualisiert und man iteriert das Verfahren weiter.

Beispiel 7.10:

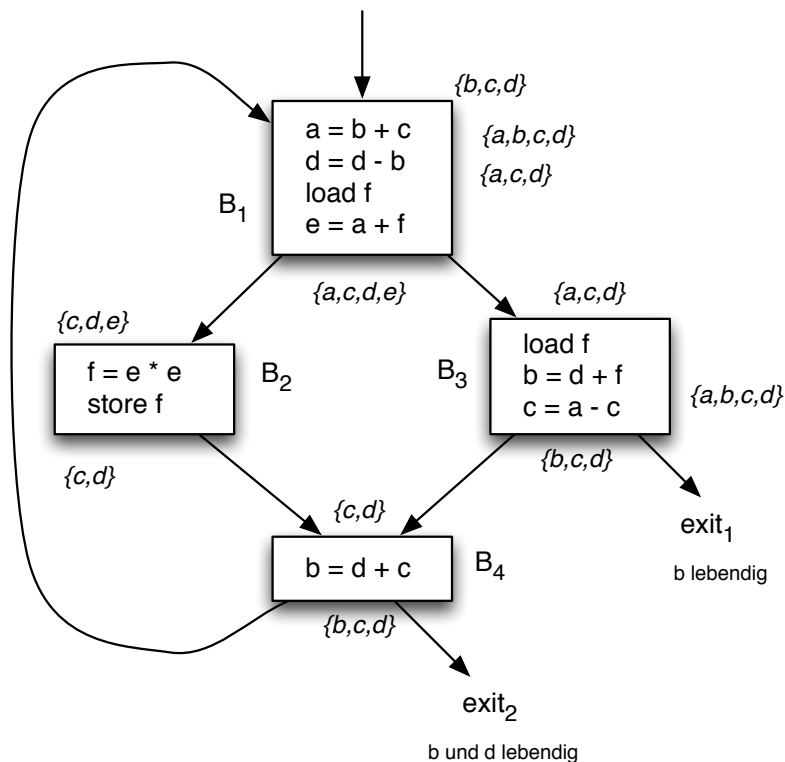
Betrachten wir den Flussgraph aus Beispiel 7.7 und nehmen wir an, dass auch wieder die vier Register R0, R1, R2 und R3 zur Verfügung stehen. In unserem Fall können die Variablennamen direkt als Namen der symbolischen Register benutzt werden. Für jeden Punkt im Flussgraph bestimmt man die lebendigen Variablen. Man erhält:



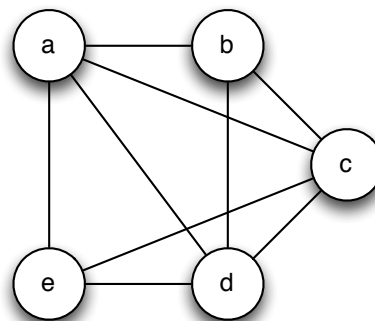
Mit diesen Informationen konstruiert man den Register Inference Graph (RIG):



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:



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:

