

Compilerbau

Wintersemester 2010 / 2011

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

Kapitel 7

Register-Allokation

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
 - Einführung
 - Lebendigkeitsanalyse
 - Register-Allokation durch Graph-Färbung
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Motivation

Speicher-Hierarchien (☞ Kapitel 8 – Scratchpad-Optimierungen):

Speicher sind um so effizienter bzgl. Laufzeit & Energieverbrauch...

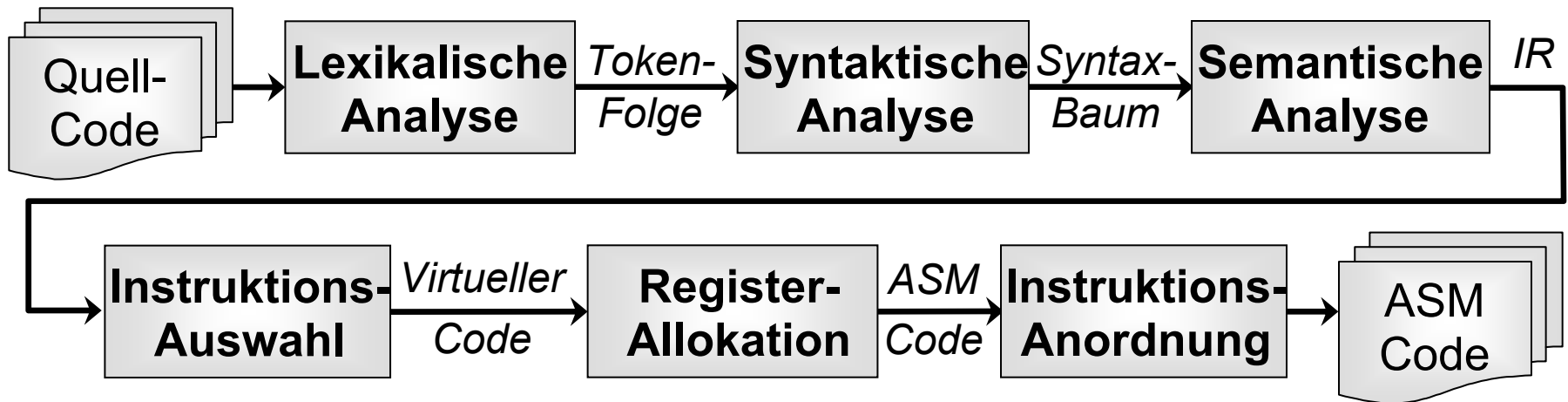
- ... je kleiner sie sind, und ...
- ... je näher sie am Prozessor platziert sind.

Register:

- Speicher-Hierarchien von Rechnern werden üblicherweise mit Festplatten, Hauptspeicher und Caches angegeben (z.B. in der Werbung).
- ☞ Aber: Register sind diejenigen Speicher, die unter allen Speichern am kleinsten und direkt im Prozessor enthalten sind.

☞ **Register sind die effizientesten Speicher schlechthin.**

Rolle der Register-Allokation



Register-Allokation (RA):

- Abbildung atomarer Daten der LIR auf physikalische Register
- Bestmögliche Ausnutzung der (knappen) Ressource von Prozessor-Registern
- ☞ *Wird als wichtigste Compiler-Optimierung überhaupt angesehen.*

Abhängigkeit RA \leftrightarrow Instruktionsauswahl

Möglichkeit 1: *Instruktionsauswahl generiert Virtuellen Code*

- LIR verwendet beliebige Anzahl virtueller Register (☞ Kapitel 2)
- ☞ RA muss jedes einzelne virtuelle Register auf ein physikalisches Register abbilden.

Möglichkeit 2: *Instruktionsauswahl generiert Stack-Zugriffe*

- Vor jedem USE eines Datums enthält die LIR einen Lade-Befehl, um das Datum vom Stack zu holen.
- Analog: Auf jedes DEF eines LIR-Datums folgt ein Schreib-Befehl
- ☞ RA muss jedes LIR-Datum auf ein Prozessor-Register abbilden und so diese Lade-/Schreib-Befehle weitestgehend entfernen.

☞ ***Im folgenden: Annahme virtuellen Codes (Möglichkeit 1).***

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
 - Einführung
 - Lebendigkeitsanalyse
 - Register-Allokation durch Graph-Färbung
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Motivation

Wann ist eine Abbildung virtueller auf physikalische Register gültig?

Zwei virtuelle Register r_0 und r_1 dürfen nur dann auf das gleiche physikalische Register abgebildet werden, wenn

✓ r_0 und r_1 niemals gleichzeitig „in Nutzung“ sind.

Wann ist ein virtuelles Register „in Nutzung“, wann nicht?

- Lebendigkeitsanalyse (*life time analysis, LTA*) ermittelt, wann die Lebenszeiten von virtuellen Registern beginnen und enden.
- Ein virtuelles Register ist lebendig, wenn es einen Wert enthält, der in Zukunft noch gebraucht werden könnte.
- LTA basiert auf Kontrollflussgraph und DEF/USE-Informationen.

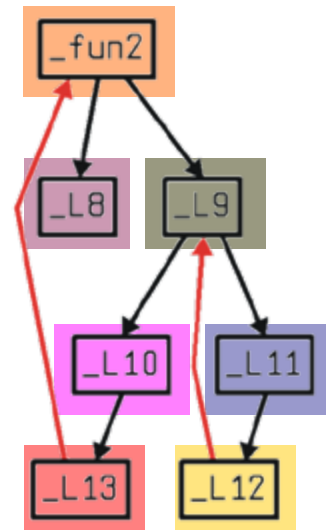
Kontrollflussgraphen

- **Kontrollfluss-Graph (control flow graph, CFG):**
Grundlegende Datenstruktur im Compilerbau
- **Definition:** $CFG = (V, E)$ ist ein gerichteter Graph mit
 $V = \{ b_1, \dots, b_n \}$ (Menge aller Basisblöcke, \rightarrow Kapitel 5)
 $E = \{ (b_i \rightarrow b_j) \mid \text{Basisblock } b_j \text{ ist direkter Nachfolger von } b_i \}$

```

int fun2() {
    for (; a1<30; a1++ ) {
        for (; b1<a1; b1++ )
            printf( "%d\n", b1 );
        printf( „%d\n“, a1 );
    }
    return a1; }

```

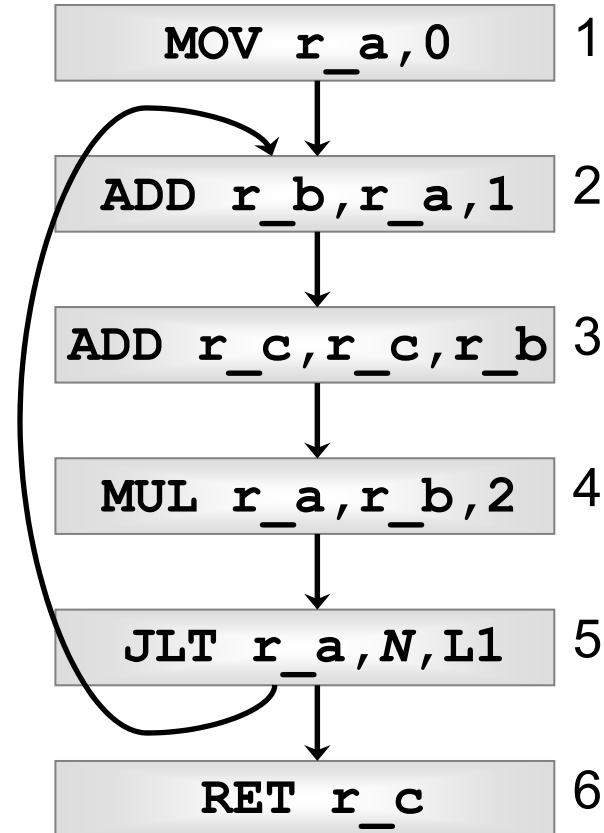


Beispiel-Kontrollflussgraph

```

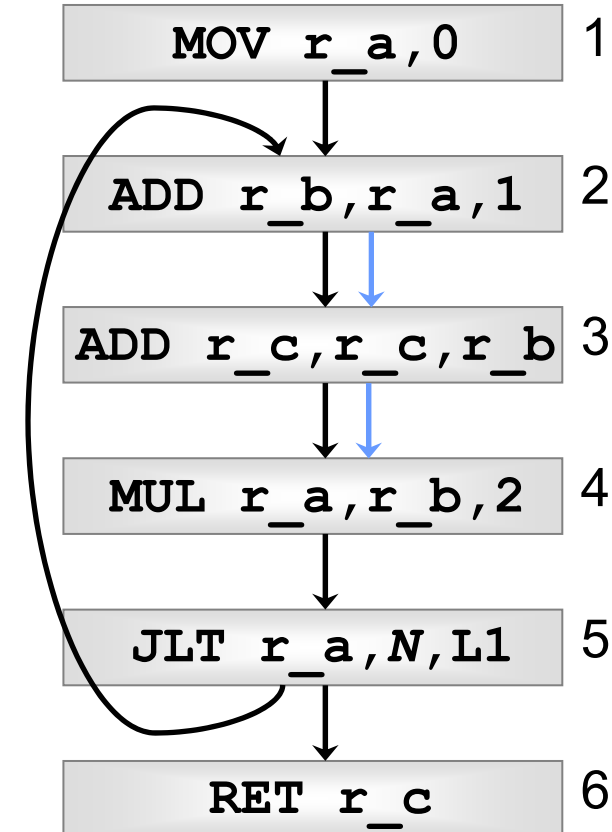
      MOV r_a,0;           # r_a = 0
L1:   ADD r_b,r_a,1;       # r_b = r_a+1
      ADD r_c,r_c,r_b;     # r_c = r_c+r_b
      MUL r_a,r_b,2;       # r_a = r_b*2
      JLT r_a,N,L1;        # if r_a<N goto L1
L2:   RET r_c;             # return r_c
  
```

- *Beachte:* CFG-Knoten repräsentieren hier einzelne Instruktionen anstatt Basisblöcke
- Code enthält drei virtuelle Register:
`r_a, r_b, r_c`



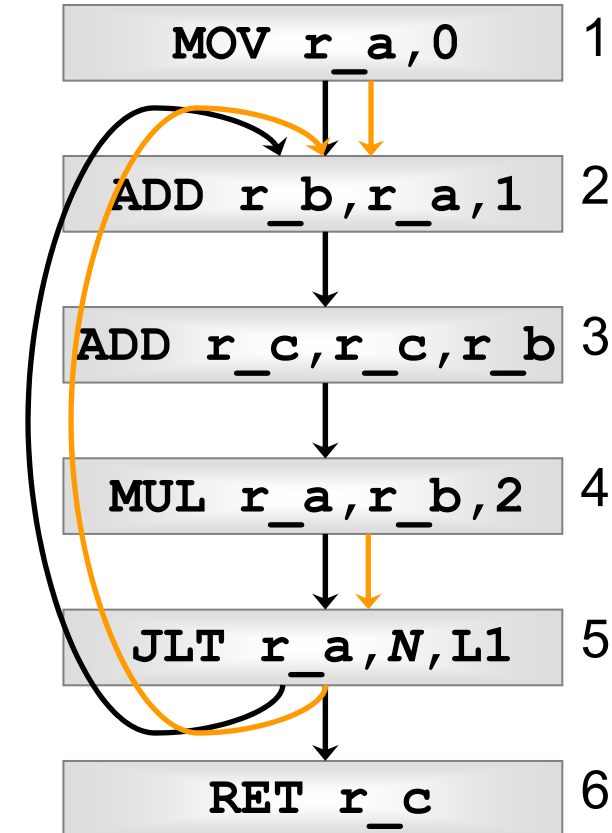
Beispiel: Lebenszeit von `r_b`

- Ein Register ist lebendig, wenn es in Zukunft noch gebraucht werden könnte.
- 👉 LTA arbeitet „von der Zukunft“ in „die Vergangenheit“
- Letztes USE von `r_b`: Knoten 4
- 👉 `r_b` ist entlang Kante (3, 4) lebendig
- Knoten 3 ist kein DEF von `r_b`
- 👉 `r_b` ist entlang Kante (2, 3) auch lebendig
- Knoten 2 definiert `r_b`:
- 👉 Inhalt von `r_b` für Knoten 2 irrelevant
- 👉 `r_b` ist entlang Kante (1, 2) nicht lebendig



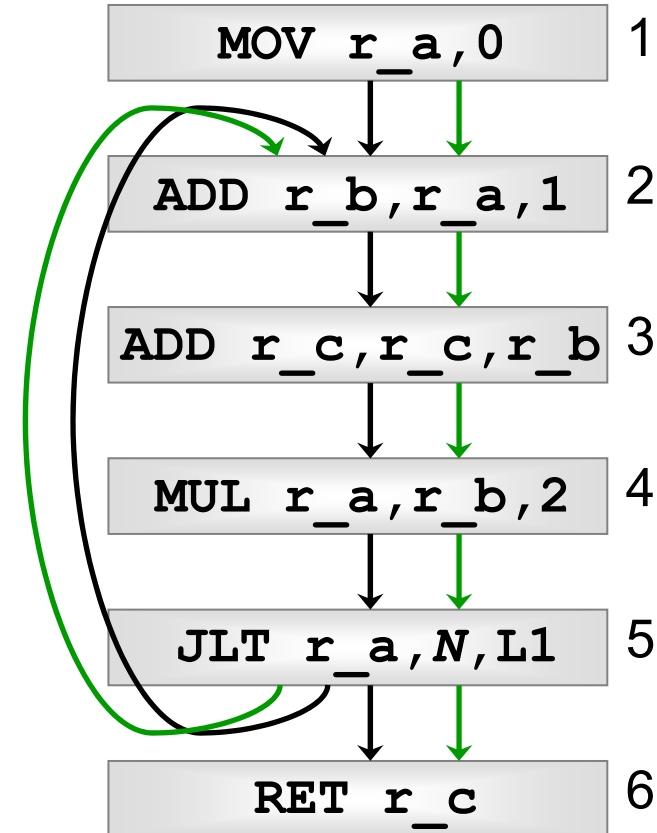
Beispiel: Lebenszeit von `r_a`

- Letztes USE von `r_a`: Knoten 5
- 👉 `r_a` ist entlang Kante (4, 5) lebendig
- Knoten 4 definiert `r_a`
- 👉 `r_a` ist entlang Kante (3, 4) nicht lebendig
- Knoten 2 ist ein USE von `r_a`:
- 👉 `r_a` wird in Knoten 1 definiert
- 👉 `r_a` ist entlang Kante (1, 2) lebendig
- 👉 `r_a` ist aber auch entlang (5, 2) lebendig, da das DEF von `r_a` in Knoten 4 über den Schleifen-Rücksprung auch Knoten 2 erreicht.



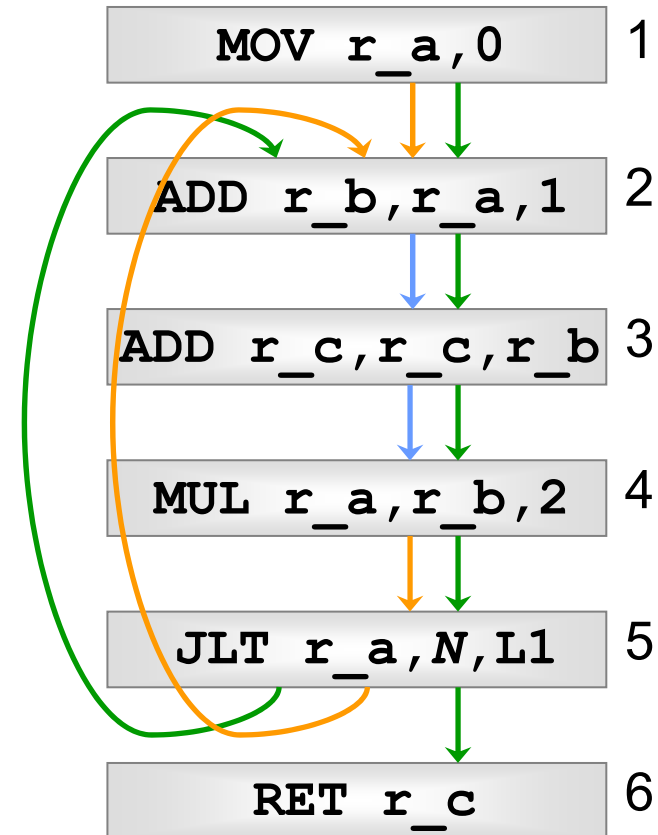
Beispiel: Lebenszeit von `r_c`

- Letztes USE von `r_c`: Knoten 6
- ☞ `r_c` ist entlang (5, 6) lebendig
- ☞ `r_c` ist entlang (4, 5) und (3, 4) lebendig
- Weiteres USE von `r_c`: Knoten 3
- ☞ DEF von `r_c` in Knoten 3 erreicht über Schleifen-Rücksprung das USE in Knoten 3
- ☞ `r_c` ist entlang (2, 3) und (5, 2) lebendig
- Zusätzlich:
- ☞ Da `r_c` außerhalb der Schleife nicht initialisiert ist: `r_c` ist entlang (1, 2) auch lebendig



Beispiel: Vollständige LTA

- Lebenszeit von `r_c` überlappt sowohl die von `r_a` als auch die von `r_b`
- ☞ `r_c` darf nicht das gleiche physikalische Register wie `r_a` erhalten
- ☞ `r_c` darf nicht das gleiche physikalische Register wie `r_b` erhalten
- Lebenszeiten von `r_a` und `r_b` sind disjunkt
- ☞ `r_a` und `r_b` dürfen sich das selbe physikalische Register teilen



Definitionen zur LTA

- Sei v ein Knoten des Knotenflussgraphen.
 - $pred_v / succ_v$ Menge aller Vorgänger/Nachfolger von v
 - def_v / use_v Menge aller von v definierten/benutzten virtuellen Register
- Ein virtuelles Register r ist entlang einer CFG-Kante lebendig, wenn im CFG ein gerichteter Pfad von dieser Kante zu einer Benutzung von r existiert, der keine weitere Definition von r enthält.
 - Register r ist *live-in* an einem CFG-Knoten v , wenn r entlang irgendeiner in v eingehenden Kante lebendig ist.
 - Register r ist *live-out* an v , wenn r entlang irgendeiner aus v ausgehenden Kante lebendig ist.

Lebendigkeit von Virtuellen Registern

- Sei v ein CFG-Knoten, r ein virtuelles Register.
 - Wenn $r \in use_v$: r ist live-in an v
 - Wenn r an v live-in ist: r ist live-out an allen $w \in pred_v$
 - Wenn r an v live-out ist, und $r \notin def_v$: r ist live-in an v

- Datenflussgleichungen zur Lebendigkeitsanalyse:
 - $in_v =$ Menge aller Register, die an v live-in sind (out_v analog)
 - $in_v = use_v \cup \{ out_v \setminus def_v \}$
 - $out_v = \bigcup_{s \in succ_v} in_s$

Algorithmus zur LTA

Gegeben: Kontrollflussgraph $G = (V, E)$ auf Instruktionsebene

Vorgehensweise: Iteratives Lösen der Datenflussgleichungen

- for (<alle Knoten $v \in V$ >)
 - $in_v = out_v = \emptyset$;
- do
 - for (<alle Knoten $v \in V$ in umgekehrt topologischer Folge>)
 - $in'_v = in_v$; $out'_v = out_v$;
 - $in_v = use_v \cup \{ out_v \setminus def_v \}$;
 - $out_v = \bigcup_{s \in succ_v} in_s$;
- while (($in_v \neq in'_v$) || ($out_v \neq out'_v$) für ein beliebiges $v \in V$)

Bemerkungen

Umgekehrt topologische Reihenfolge:

- Erzeuge Graph G' aus dem Kontrollflussgraphen G , indem alle Kanten herumgedreht werden.
- Führe Durchlauf durch G' in Tiefensuche (*depth first search, DFS*) durch, beginnend bei den Quellen von G' , d.h. den Knoten ohne eingehende Kanten.
- Kehre die Reihenfolge, in der Knoten von G' beim DFS-Durchlauf besucht werden, um.

Beispiel von Folie 10:

- DFS-Reihenfolge des „umgedrehten“ Graphen: 1, 2, 3, 4, 5, 6
- Umgekehrt topologische Reihenfolge: 6, 5, 4, 3, 2, 1

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
 - Einführung
 - Lebendigkeitsanalyse
 - Register-Allokation durch Graph-Färbung
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Graph-Färbung und Register-Allokation

Definition (*Graph-Färbung*):

Sei $G = (V, E)$ ein ungerichteter Graph, $K \in \mathbb{N}$.

Das Problem der *Graph-Färbung* besteht darin, jedem Knoten $v \in V$ eine eindeutige Farbe $k_v \in \{1, \dots, K\}$ zuzuweisen, so dass gilt:

$$\forall e = \{v, w\} \in E: k_v \neq k_w$$

(Keine zwei benachbarten Knoten dürfen die gleiche Farbe haben)

Idee einer Register-Allokation mit Graph-Färbung:

- Erzeuge Graphen G mit einem Knoten v pro virtuellem Register.
- Färbe G mit K Farben, wobei der betrachtete Ziel-Prozessor über K physikalische Register verfügt.
- Die Farbe k_v gibt an, welches physikalische Register das zu v gehörende virtuelle Register belegt.

Graph-Färbung und Register-Allokation

Definition (*Interferenzgraph*):

Für eine gegebene LIR sei $R_v = \{r_1, \dots, r_n\}$ die Menge aller virtuellen Register, $R_p = \{R_1, \dots, R_K\}$ die Menge aller physikalischen Register. Der *Interferenzgraph* ist ein ungerichteter Graph $G = (V, E)$ mit

- $V = R_v \cup R_p$ und
- $e = \{v, w\} \in E$ wenn v und w niemals das gleiche physikalische Register haben dürfen, d.h. wenn v und w *interferieren*.

Register-Interferenz: Zwei Register r_i und ...

- ... r_j interferieren, wenn sich ihre Lebenszeiten überlappen.
- ... R_j interferieren, wenn eine LIR-Operation op r_i verwendet, op aber nicht das physikalische Register R_j adressieren kann.

Graph-Färbung und Register-Allokation

Spezialfall Register-Transfers:

```

MOV r0, r1;           /* DEF: r0, USE: r1 */
...
ADD ri, rj, r0;      /* USE: r0 */
...
MUL rk, r1, r1;      /* USE: r1 */

```

- Lebenszeiten von **r0** und **r1** überlappen sich: streng genommen müsste eine Kante $\{r0, r1\}$ in G eingefügt werden.
- *Aber:* $\{r0, r1\}$ ist unnötig, da **r0** und **r1** den gleichen Wert enthalten. **r0** und **r1** dürfen gleiches physikalisches Register haben.
- 👉 Es wird in diesem speziellen Fall keine Kante $\{r0, r1\}$ erzeugt.
- 👉 Falls später **r0** und **r1** gleiche Farbe k haben, ist die MOV-Operation überflüssig und kann ohne weiteres entfernt werden.

Erzeugung des Interferenzgraphen (1)

Gegeben:

- LIR L
- Menge $R_p = \{R_1, \dots, R_K\}$ aller physikalischen Register

Basis-Algorithmus:

- $G = (V, E) = (R_v \cup R_p, \emptyset)$;
 - for (<alle Funktionen $f \in L$ >)
 - for (<alle Basisblöcke $b \in f$ >)
 - for (<alle Instruktionen $i \in b$ >)
 - $live = out_i \cup def_i$;
 - for (<alle Paare (r_j, r_k) mit $r_j \in def_i$ und $r_k \in live, j \neq k$ >)
 - if ($!isMOV(i) \parallel (isMOV(i) \ \&\& \ (r_k \notin use_i))$)
- $E = E \cup \{ \{r_j, r_k\} \}$;

Erzeugung des Interferenzgraphen (2)

Erweiterung des Basis-Algorithmus:

Maschinenabhängige Interferenzen müssen nach Ablauf des Basis-Algorithmus explizit in G nachgetragen werden, z.B. wenn

- eine LIR-Operation nicht sämtliche Register adressieren kann.
- Konventionen zum Aufruf von / Rücksprung aus Funktionen erzwingen, dass Funktionsparameter bzw. Rückgabewerte in ganz bestimmten Registern vorliegen (sog. *Calling Conventions*).
- die Nutzung von extended Registern (☞ Kapitel 1) weitere Einschränkungen nach sich zieht: Für ein virtuelles extended Register \mathbb{E}_0 , bestehend aus den Teilen \mathbf{d}_1 und \mathbf{d}_2 , muss erreicht werden, dass stets \mathbf{d}_1 auf ein gerades und \mathbf{d}_2 auf das nachfolgende ungerade physikalische Register abgebildet wird.

Graph-Färbung per Vereinfachung (1)

1. **Initialisierung:** Aufbau des Interferenzgraphen G
2. **Vereinfachung:** Sukzessiv werden solche Knoten v aus G entfernt und auf einen Stack S gelegt, die kleineren Grad als K haben, d.h. die höchstens $K-1$ Nachbarn haben.
 (*☞ Solche Knoten sind immer K -färbbar, da es in der Nachbarschaft von v immer eine freie Farbe für v geben muss.*)
3. **Spilling:** Schritt 2 stoppt, wenn alle Knoten Grad $\geq K$ haben. Ein Knoten v wird ausgewählt, als *potentieller Spill* markiert, aus G entfernt und auf S gelegt.
 (*☞ Spilling = Ein-/Auslagern von Registerinhalten in den/aus dem Speicher, falls kein physikalisches Register mehr frei ist.*)
4. Wiederhole Vereinfachung und Spilling bis $G = \emptyset$.

Graph-Färbung per Vereinfachung (2)

- 5. Färbung:** Sukzessiv werden Knoten v vom Stack S entfernt und wieder in G eingefügt. Ist v kein potentieller Spill, so *muss* v färbbar sein. Ist v ein potentieller Spill, so *kann* v färbbar sein. Weise v in beiden Fällen eine freie Farbe k_v zu. Ist ein potentieller Spill nicht färbbar, wird v als *echter Spill* markiert.
- 6. Spill-Code-Generierung:** Für jeden echten Spill v wird vor jedem USE von v eine Lade-Operation eingefügt und nach jedem DEF von v eine Schreib-Operation.
(☞ Damit zerfällt die Lebenszeit des virtuellen Registers v in viele kleine Intervalle, die hoffentlich anschließend färbbar sind.)
- 7. Neustart:** Falls G ungefärbte Knoten enthält, gehe zu Schritt 1.
- 8. MOV-Operationen** mit Quellregister = Zielregister werden entfernt.

Coalescing (1)

Register-Transfers:

- Beim Aufbau des Interferenzgraphen wurden für MOV-Operationen keine künstlichen Kanten eingefügt, in der Hoffnung, dass Ziel und Quelle der MOV-Operation zusammenfallen.
- Aber: Der Graphfärbungsalgorithmus von Folien 24 & 25 erzwingt nicht, dass Ziel und Quelle auch tatsächlich zusammenfallen.

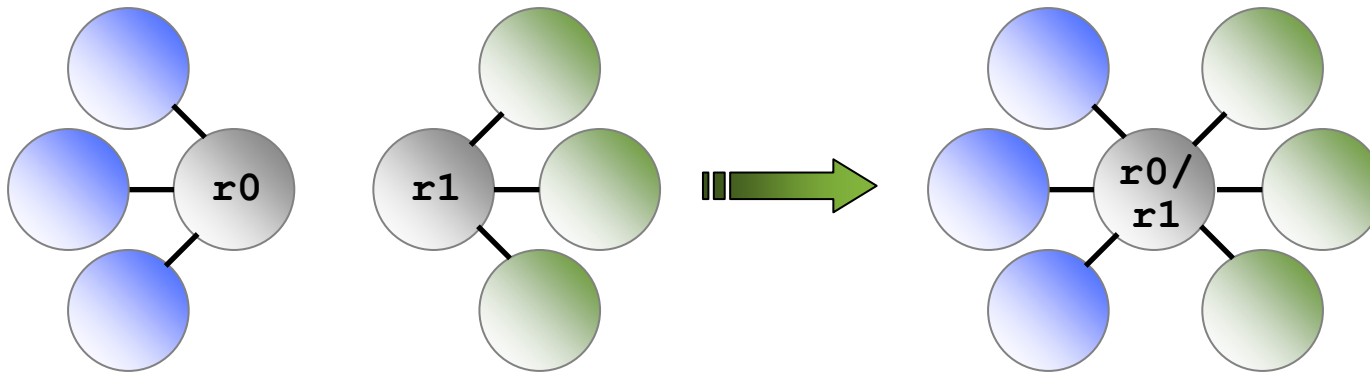
Register-Coalescing:

- Für einen Register-Transfer `MOV r0, r1` mit nicht interferierenden Quell- und Ziel-Registern vereinigt *Coalescing* im Interferenzgraph die Knoten von `r0` und `r1`, so dass bei anschließender Graph-Färbung `r0` und `r1` zwingend die gleiche Farbe erhalten.

Coalescing (2)

Effekte des Coalescing:

MOV r0, r1;



- 👉 **Vorteil des Coalescing:** Unnötige Register-Transfers entfallen, Maschinenbefehle legen ihre Resultate direkt in dem Register ab, in dem diese effektiv gebraucht werden.
- 👉 **Nachteil:** Verschmolzener Knoten hat höheren Grad als ursprüngliche Knoten, d.h. der Interferenzgraph kann nach Coalescing u.U. nicht mehr K -färbbar sein, während dies vorher u.U. zutraf.

Coalescing (3)

Sicheres Coalescing:

- Coalescing heißt *sicher*, wenn niemals ein vorher K -färbbarer Interferenzgraph nach dem Coalescing nicht mehr K -färbbar ist.
- ☞ Sicheres Coalescing entfernt somit u.U. nicht sämtliche möglichen Register-Transfers aus dem Code.
- ☞ Aber: verbleibende MOV-Operationen sind stets besser als Spill-Code-Generierung für nicht K -färbbaren Interferenzgraphen.

Ablauf:

- Coalescing wird nach Vereinfachung und vor Spilling ausgeführt. Ist Coalescing möglich, wird der Graph danach weiter vereinfacht.
- Vereinfachung entfernt nur solche Knoten v aus G , die weder Quelle noch Ziel einer MOV-Operation sind.

Coalescing (4)

Sicheres Coalescing nach Briggs:

- Verschmelze zwei Knoten v_0 und v_1 nur dann, wenn der resultierende Knoten v_0/v_1 weniger als K Nachbarn mit $\text{Grad} \geq K$ hat.
 - Hat v_0/v_1 nach Coalescing weniger als K Nachbarn mit $\text{Grad} \geq K$, so müssen vor dem Coalescing v_0 und v_1 jeweils für sich betrachtet auch weniger als K Nachbarn mit $\text{Grad} \geq K$ haben.
 - Ist der Interferenzgraph vor dem Coalescing K -färbbar, so ist er es hinterher auch, da v_0/v_1 in einer nachfolgenden Vereinfachung entfernt wird.
- 👉 Coalescing nach Briggs ist sicher.

Coalescing (5)

Sicheres Coalescing nach George:

- Verschmelze zwei Knoten v_0 und v_1 nur dann, wenn für jeden Nachbarn v_i von v_0 gilt: entweder interferiert v_i mit v_1 oder v_i hat Grad kleiner als K .
 - Nachbarn v_i mit Grad $< K$ haben auch nach dem Coalescing Grad $< K$ und werden somit in nachfolgender Vereinfachung entfernt.
 - Andere Nachbarn v_i , die vor dem Coalescing mit v_1 interferieren, haben per Definition zwei Kanten $\{v_i, v_0\}$ und $\{v_i, v_1\}$.
 Nach dem Coalescing fallen diese beiden Kanten zu einer Kante $\{v_i, v_0/v_1\}$ zusammen, so dass die Grade der beteiligten Knoten nur geringer werden können.
- 👉 Coalescing nach George ist sicher.

Literatur

Lebendigkeitsanalyse und Register-Allokation:

- Andrew W. Appel, *Modern compiler implementation in C*, Cambridge University Press, 1998.
ISBN 0-521-58390-X

Zusammenfassung

■ Lebendigkeitsanalyse

- Ermittlung des Beginns/Endes der Lebenszeiten von Registern
- Virtuelle Register dürfen sich nur dann ein physikalisches Register teilen, wenn sie nicht gleichzeitig lebendig sind
- Iteratives Lösen von Datenflussgleichungen

■ Register-Allokation durch Graph-Färbung

- Interferenzgraph G modelliert überlappende Lebenszeiten virtueller Register sowie zusätzliche Randbedingungen
- Färbung von G repräsentiert Abbildung virtueller auf physikalische Register
- Graph-Färbung durch iteriertes Vereinfachen, Spillen & Färben
- Sicheres Coalescing zum Entfernen von Register-Transfers

