Codeerzeugung – Teil 2

Übersicht

- Instruktionsanordnung
- Basisblockgraph
  - Basisblockgraph
  - Abhängigkeitsgraphen für Basisblöcke
  - Setzungen und Benutzungen
- Abhängigkeiten zwischen Befehlen
  - Algorithmus BBA-Graph
  - Verkleinerung des Abhängigkeitsgraphen
- Befehlsfließband
  - Algorithmus FB-Anordnung
  - Realistische Befehlsfließbänder
- Lange Befehlswörter
  - Verallgemeinerter Abhängigkeitsgraph
  - Kompensationscode
  - Realistische VLIW-Rechner

Lernziele

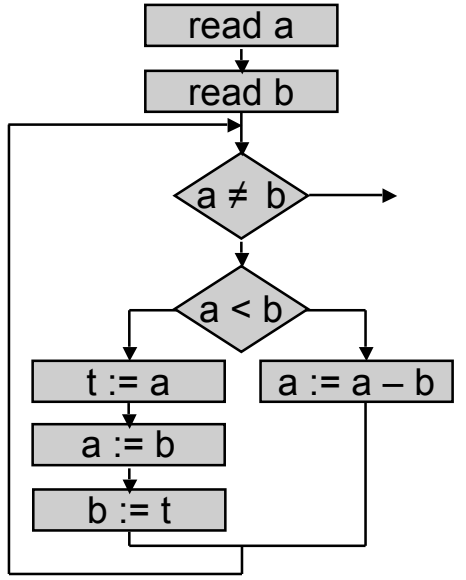
- Die Problematik der Instruktionsanordnung und das Lösungsprinzip mithilfe von Abhängigkeitsgraphen für Basisblöcke beschreiben können
- Probleme der Parallelverarbeitung und prinzipielle Lösungsmöglichkeiten für Befehlsfließbänder und „lange Befehlswörter“ wiedergeben können

Instruktionsanordnung

- Relevant für Prozessoren mit Parallelverarbeitung und/oder Befehlsfließband
- Umordnung der erzeugten Befehlsfolge, um parallel arbeitende Einheiten bzw. Befehlsfließband gut auszunutzen
- Problem
  - Semantik darf nicht verändert werden, insbesondere Abhängigkeiten zwischen den Berechnungen

Beispiel

```
read a; read b;
while a ≠ b do
   if a < b
      then t := a; a := b; b := t
      else a := a – b fi od
```



Darstellung semantischer Aspekte

- Kontrollfluss
  - (Prozedur-) Aufrufgraph (call graph)
    - Knoten: HP und Prozeduren; Kanten: Aufrufbeziehung
    - Benötigt für interprozedurale Analyse
  - Kontrollflussgraph
    - Knoten: primitive Anweisungen; Kanten: Ausführungsordnung
    - Benötigt für intraprozedurale Analyse
  - *Basisblockgraph* ☛
    - Abstraktion des Kontrollflussgraphen
- Semantische Abhängigkeiten: *Abhängigkeitsgraph* ☛

Basisblockgraph

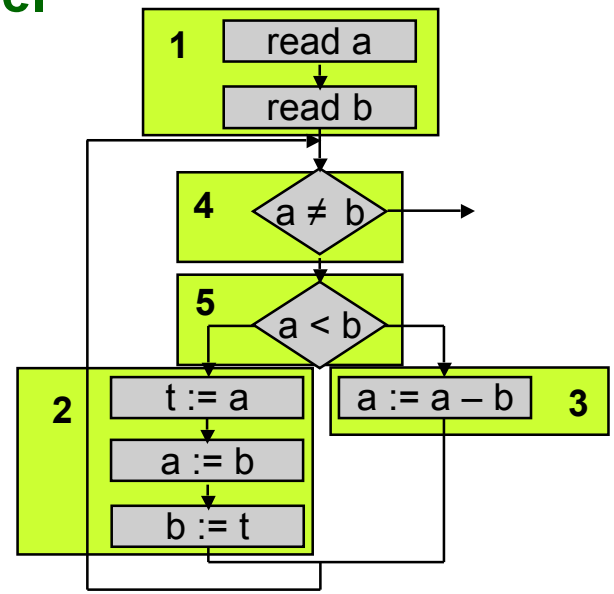
Basisblock (in einem Kontrollflussgraphen)

- Maximal langer Pfad, der höchstens am Anfang einen Knoten mit mehr als einem Vorgänger und höchstens am Ende einen Knoten mit mehr als einem Nachfolger hat

Basisblockgraph (kurz: BBG)

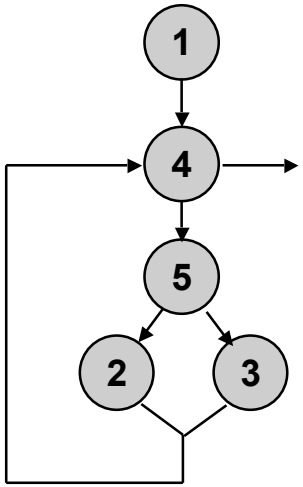
- Entsteht aus Kontrollflussgraphen durch Zusammenfassen von Basisblöcken zu Knoten

Beispiel



Kontrollflussgraph

Basisblöcke



Basisblockgraph

Abhängigkeitsgraphen für Basisblöcke

Charakteristika eines Basisblocks

- Befehlsfolge linear geordnet (insbesondere keine Verzweigungen)
- Wird vollständig und in der angegebenen Reihenfolge durchlaufen (wenn Kontrolle an ersten Befehl gegeben wird)

Umordnung der einzelnen Befehle

- Möglich, wenn „Gesamteffekt“ der Befehlsfolge unverändert bleibt
- Abhängig von Abhängigkeiten zwischen den Befehlsarten
  - Setzungen (definitions): Schreibende Zugriffe (dargestellt als X :=)
  - Benutzungen (uses): Lesende Zugriffe (dargestellt als := X)

Beispiel

- Befehl D1 := ADD M[A1 + D1.W], D5
  - Benutzungen: Register A1, D1, D5; Speicher M
  - Setzungen: Register D1; verschiedene Bits im Bedingungscode
- Offensichtlich: Keine Umordnungsmöglichkeiten in der Befehlsfolge

```
a: X :=
b: X :=
c:   := X
d: X :=
```

Setzungen und Benutzungen

Mögliche Setzungen in Maschinenprogrammen

- Änderungen von Registerinhalten durch Ladebefehle, durch Resultatablage (nach Operationen) oder durch (automatische) In- bzw. Dekrementierung (von Adressregistern)
- Abspeichern von Werten in Speicherzellen
- Modifikation eines Kellerzeigers (durch Kellern und Entkellern von Werten und durch Befehle zur Prozedurorganisation)
- Setzen von Überlauf-, Unterlauf- oder Übertragsbits und Vergleichsergebnissen im Bedingungscode

Mögliche Benutzungen

- Benutzung von Registerinhalten in Operationen, zur Adressierung oder zum Abspeichern
- Laden des Inhalts von Speicherzellen
- Abfragen von Teilen des Bedingungscodes in bedingten Sprüngen

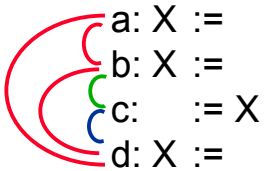
Abhängigkeiten zwischen Befehlen (1/2)

Abhängigkeiten zwischen Befehlen (die sequentielle Anordnung erfordern)

- Betreffen *Maschinenressourcen*

  (Einzelnes Register, Hauptspeicher, Bits des Bedingungscodes)

- *Arten von Abhängigkeiten*
  - Setzung - Setzung („definition - definition“; kurz: d-d)
  - Setzung - Benutzung („definition - use“; kurz: d-u)
  - Benutzung - Setzung („use - definition“; kurz: u-d)



Extraktion von Abhängigkeiten aus Befehlsfolge

- Unmittelbar bei Registern (da explizit benannt)
- Probleme bei Speicherzellen: **alias-Problem**

  (zwei Adressierungsarten mit dynamischen Anteilen in zwei verschiedenen Befehlen können Zugriff auf die gleiche Speicherzelle beschreiben)
- Deswegen (zunächst): Ganzer Speicher als **ein** Objekt betrachtet

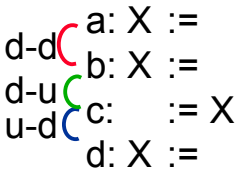
Darstellung der Abhängigkeiten

- Im *Abhängigkeitsgraph* ☛

Abhängigkeiten zwischen Befehlen (2/2)

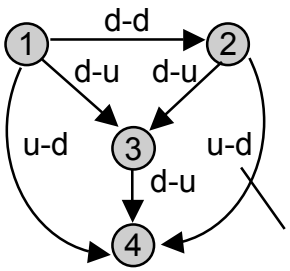
Abhängigkeitsgraph (eines Basisblocks)

- Knoten- und kantenmarkierter azyklischer gerichteter Graph
  - Knoten mit den Befehlen des Basisblocks markiert
  - Kante von a nach b, wenn a in der Befehlsfolge vor b steht und wenn
    - a = Setzung; b = Benutzung; Weg von a nach b setzungsfrei (d-u-Abhängigkeit)
    - a = Benutzung; b = Setzung; Weg von a nach b setzungsfrei (u-d-Abhängigkeit)
    - a = Setzung; b = Setzung; Weg von a nach b setzungs- und benutzungsfrei (d-d-Abhängigkeit)
  - Nur Kanten für direkte Abhängigkeiten (Indirekte Abhängigkeiten: transitive Hülle)



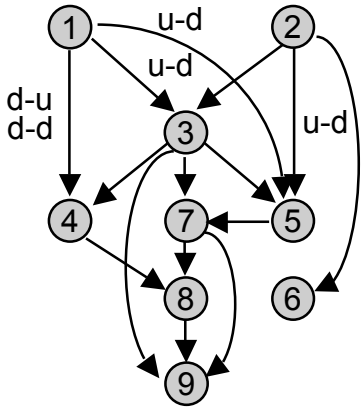
Beispiele

```
1: (CC, D1)   := M[A1+ 4].W;
2: (CC, D2)   := M[A1 + 6].W;
3: (CC, D1)   := D1+ D2;
4: M[A1 + 4]  := D1.W;
```



Beachte:
nicht vollständig annotiert

```
1: D1         := M[A1+ 4];
2: D2         := M[A1 + 6];
3: A1         := A1 + 2;
4: D1         := D1+ A1;
5: M[A1]      := A1;
6: D2         := D2 + 1;
7: D3         := M[A1 + 12];
8: D3         := D3 + D1;
9: M[A1 + 6]  := D3;
```



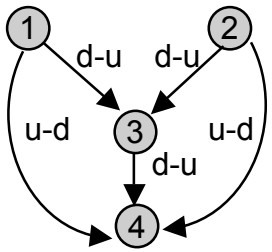
Algorithmus BBA-Graph

```
var     LS, EB: set of pair (Resource, Befehlsnummer);
        akt_Befehl: Befehlsnummmer;
begin
    akt_Befehl := letzter Befehl des Basisblocks;
    erzeuge_Knoten (akt_Befehl);
    LS := Setzungen(akt_Befehl);
    EB := Benutzungen(akt_Befehl);
    while Vorgänger(akt_Befehl) definiert do
        akt_Befehl := Vorgänger(akt_Befehl);
        erzeuge_Knoten(akt_Befehl);
        foreach Resource r die in akt_Befehl gesetzt/benutzt wird do
            foreach (r, b) ∈ LS ∪ EB do
                case Konflikt(r, akt_Befehl, b) of
                    d-d: if ∄(r, α) ∈ EB then ziehe_Kante(akt_Befehl → b) fi;
                    d-u: ziehe_Kante(akt_Befehl → b);
                    u-d: ziehe_Kante(akt_Befehl → b) endcase od od;
        foreach Resource r‘ die in akt_Befehl gesetzt wird do
            LS := LS \ {(r‘, α) ∈ LS} ∪ {(r‘, akt_Befehl)};
            EB := EB \ {(r‘, β) ∈ EB} od;
        foreach Resource r‘ die in akt_Befehl benutzt wird do
            EB := EB ∪ {(r‘, akt_Befehl)} od od
end
```

Basisblock

```
1: D1          := M[A1+ 4].W;
2: D2          := M[A1 + 6].W;
3: D1          := D1+ D2;
4: M[A1 + 4]   := D1.W;
```

Abhängigkeitsgraph



Akt_Bef = 1;
LS = {(M,4), (D1,3), (D2,2)}
EB = {(A1,4), (A1,2), (D1,3), (M,2)}
r = {D1, M, A1}
    für (D1,3): Konflikt(D1,1,3) = d-u
    für (M,4): Konflikt(M,1,4) = u-d
    für (M,2): Konflikt(M,1,2) = false
    für (A1,4): Konflikt(A1,1,4) = false
    für (A1,2): Konflikt(A1,1,2) = false
LS = {(M,4), (D1,1), (D2,2)}
EB = {(M,2), (A1,4), (A1,2)}
EB = {(M,2), (M,1), (A1,4), (A1,2), (A1,1)}

Verkleinerung des Abhängigkeitsgraphen

Eigenschaften des Abhängigkeitsgraphen

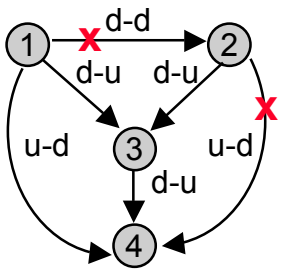
- Stellt mögliche Freiheitsgrade für die Umordnung von Befehlsfolgen dar
- Gültige Anordnungen durch *Topologisches Sortieren* ☛
  - gesteuert durch Heuristiken
  - Eventuell zusätzliche Einschränkungen (z.B. vorgeschriebene Abstände zwischen voneinander abhängigen Befehlen)

Verkleinerung des Abhängigkeitsgraphen (Elimination von Kanten)

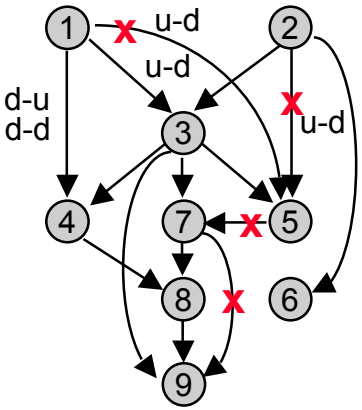
- *Aliasanalyse*: Verschiedenheit von Speicheradressen ermitteln
- *Lebendigkeitsanalyse*
  - Ermitteln, ob Ressourcen nach Setzung noch gebraucht werden (evtl. in nachfolgenden BBs)
  - Tritt häufig bei CC auf

Beispiele

```
1: (CC, D1)   := M[A1+ 4].W;
2: (CC, D2)   := M[A1 + 6].W;
3: (CC, D1)   := D1+ D2;
4: M[A1 + 4]  := D1.W;
```




```
1: D1          := M[A1+ 4];
2: D2          := M[A1 + 6];
3: A1          := A1 + 2;
4: D1          := D1+ A1;
5: M[A1]       := A1;
6: D2          := D2 + 1;
7: D3          := M[A1 + 12];
8: D3          := D3 + D1;
9: M[A1 + 6]   := D3;
```



Befehlsfließband

Zielmaschine

- Prozessor mit k-stufigem Befehlsfließband
- Vereinfachende Annahme:
  Abhängige Befehle stets nach Verzögerung von einem Zyklus ausführbar

Ziel der Instruktionsanordnung

- Umordnung der Befehlsfolgen innerhalb der Basisblöcke unter Beachtung der Abhängigkeiten und Fließbandbedingungen in möglichst kurze Befehlsfolgen
- Dabei
  - Ursprüngliche Befehle bleiben erhalten
  - Werden evtl. umgeordnet (unter Beachtung der Abhängigkeiten)
  - Zusätzliche NOP-Befehle (wo erforderlich)

Grundidee des Verfahrens

- *Topologisches Sortieren* ☛ des azyklischen Abhängigkeitsgraphen

Algorithmus FB-Anordnung (1/2)

Topologisches Sortieren (allgemeine Idee)

- Geg.: partielle Ordnung R auf Menge X
- Ges.: totale Ordnung T auf X, die mit R verträglich ist
- Vorgehensweise
  - Initialisierung: minimale Elemente von R als Menge K (von Kandidaten)
  - Nach Wahl und Anordnung eines Elements x aus K: Entfernung von x (und seinen Kanten) aus K und R (damit: neue minimale Elemente in R \ {x}, die zu K hinzugefügt werden)

Besonderheit des Algorithmus FB-Anordnung

- Allgemeines topologisches Sortieren:
  nichtdeterministische Auswahl unter den Kandidaten
- Hier: Heuristik, die Kollisionen berücksichtigt
  - Nur Kandidaten, die mit den zuletzt angeordneten Befehlen keine Kollision haben
  - Falls keine solchen existieren: Einfügen einer NOP-Instruktion
  - Falls solche existieren: Auswahl nach den (nach Priorität geordneten) Kriterien
    - Befehle, die mit vielen nachfolgenden Befehlen Kollisionen erzeugen können
      (sind i.a. Befehle, die die meisten Nachfolger im Abhängigkeitsgraphen haben)
    - Befehle, die auf jeweils längsten Pfaden zu den Blättern des Abhängigkeitsgraphen liegen

Algorithmus FB-Anordnung (2/2)


```
var Kands, rKands, potKolls: set of BefNr;
begin
    Kands := {Minimale El. von BBA-Graph};
    potKolls := {letzte Bef. in Anordnungen der Vorgänger-BBs};
    repeat
        rKands : = Kands \ kollidierend(Kands, potKolls);
        if rKands ≠ ∅
            then wähle bestes b ∈ rKands (gemäß Heuristik);
                 ordne b an;
                 Kands := Kands / {b};
                 entferne b und seine Kanten aus BBA-Graph;
                 Kands := Kands ∪ {neue Quellen in BBA-Graph};
                 potKolls := {b}
            else ordne NOP an;
                 potKolls := ∅ fi
    until Kands = ∅
end
```

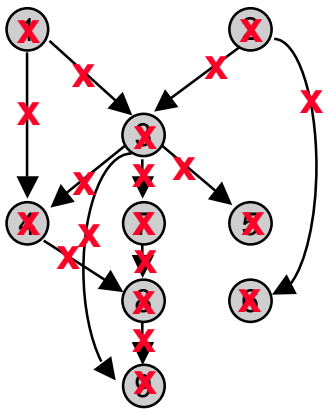
Angeordnete Befehle

```
2: D2        := M[A1 + 6];
1: D1        := M[A1 + 4];
6: D2        := D2 + 1;
3: A1        := A1 + 2;
   NOP;
7: D3        := M[A1 + 12];
4: D1        := D1 + A1;
5: M[A1]     := A1;
8: D3        := D3 + D1;
   NOP;
9: M[A1 + 6] := D3;
```

Kands potKolls

{1,2} ∅

BBA-Graph



rKands	Bester	zu elim. Kanten	Kands	potKolls
{1,2} \ ∅ = {1,2}	2	2-3, 2-6	{1,6}	{2}
{1,6} \ {6} = {1}	1	1-4, 1-3	{6,3}	{1}
{6,3} \ {3} = {6}	6	∅	{3}	{6}
{3} \ ∅ = {3}	3	3-4, 3-5, 3-7, 3-9	{4,5,7}	{3}
{4,5,7} \ {4,5,7} = ∅				∅
{4,5,7} \ ∅ = {4,5,7}	7	7-8	{4,5}	{7}
{4,5} \ ∅ = {4,5}	4	4-8	{5,8}	{4}
{5,8} \ {8} = {5}	5	∅	{8}	{5}
{8} \ ∅ = {8}	8	8-9	{9}	{8}
{9} \ {9} = ∅				∅
{9} \ ∅ = {9}	9	∅	∅	{9}

Realistische Befehlsfließbänder

Probleme bei realistischen Befehlsfließbändern

- Spezielle Verzögerungen (> 1) zwischen Befehlen erforderlich
  (neue Datenstruktur(en) statt der Menge potKoll im Algorithmus FB-Anordnung)

Z.B. Schlange(n) fester Länge(n)

- „Hot spots“
  Werte im Fließband, die genau einen Zyklus später verarbeitet werden müssen, nachdem sie produziert wurden

Durch beide Aspekte

- Weitere Einschränkungen für einen Instruktionsanordner
- Entsprechende Vergrößerung des Aufwands

Lange Befehlswörter (1/3)

Problem

- Codeerzeugung für einen Prozessor mit n parallel arbeitenden funktionalen Einheiten und (n Befehle breitem) langem Befehlswort

n häufig > 10

- Vereinfacht: Umordnung einer Befehlsfolge in n (parallele) Befehlsfolgen (oder in eine neue n Befehle breite Folge von langen Befehlsworten)
- Dazu notwendig: Befehlsanordnung über Basisblockgrenzen hinaus (da möglicher Parallelismus innerhalb von Basisblöcken sehr beschränkt ist)

Bei numerischen Programmen: Faktor 2-3; Sonst kleiner

Prinzipielles Verfahren

- Anordnen von „Pfaden“ (trace scheduling) anstelle von Basisblöcken
- Grundidee (Berücksichtigung der relativen „Durchlaufhäufigkeit“)
  - Um mögliche Parallelität zu erhöhen:
    Gemeinsame Anordnung von Instruktionen aus aufeinander folgenden Basisblöcken, die während der Programmausführung „häufig“ unmittelbar nacheinander ausgeführt werden
  - Häufigkeitsinformation aus Messungen oder heuristischen Schätzungen

Lange Befehlswörter (2/3)

Zerlegung des Kontrollflussgraphen (einer Prozedur)

- Bestimmung des ersten Teilpfades
  - Zunächst betrachtet: am häufigsten durchlaufener Basisblock
  - Dann: Entscheidung darüber, ob und welche vorangehenden oder nachfolgenden Basisblöcke gemeinsam mit diesem angeordnet werden sollen
  - Dabei verwendet: Geeignete Strategie auf der Basis der relativen Durchlaufhäufigkeit
- Für den gewonnenen Teilpfad wird dann eine *Folge langer Befehlswörter konstruiert* ☛
- Bestimmung der weiteren Teilpfade
  - Von den noch nicht angeordneten Basisblöcken dann wieder den am häufigsten durchlaufenen auswählen und w.o. verfahren
- Ende: alle Basisblöcke der Prozedur sind (im Rahmen von Teilpfaden) angeordnet

Lange Befehlswörter (3/3)

Konstruktion von Folgen langer Befehlswörter für Teilpfade

- Teilpfade können als verlängerte Basisblöcke gesehen werden
  (aus denen Verzweigungen hinaus- und in die Zusammenführungen hineinführen)
- Ziel: Packen der Befehle des Teilpfads in möglichst kurze Folge langer Befehlswörter
  (ohne Abhängigkeiten und durch Kontrollfluss gegebene semantische Bedingungen zu verletzen)
- Annahme
  - Unbeschränkt viele funktionale Einheiten
  - Beliebig lange Befehlswörter
- Dann gilt
  - Kürzeste mögliche Folge von langen Befehlswörtern so lang wie der längste (kritische) Pfad des *verallgemeinerten Abhängigkeitsgraphen* ☛

Abhängigkeiten bleiben erhalten

Möglicher (optimaler) Algorithmus

- Aus topologischem Sortieren ableitbar
  - Füllung eines neuen Befehlsworts mit der Menge von Kandidaten
    (= Befehle, die keine Vorgänger mehr haben)
  - Dabei: Verschieben von Befehlen über Verzweigungen hinweg mithilfe *semantikerhaltender Transformationen* (die für Erzeugung des notwendigen *Kompensationscodes* ☛ sorgen)

Verallgemeinerter Abhängigkeitsgraph

- Darstellung von Abhängigkeiten in Teilpfaden
- Im folgenden betrachtet: Bedingte Anweisungen und Schleifen

Bedingte Anweisungen (s_0; **if** b **then** s_1 **else** s_2 **fi**; s_3)

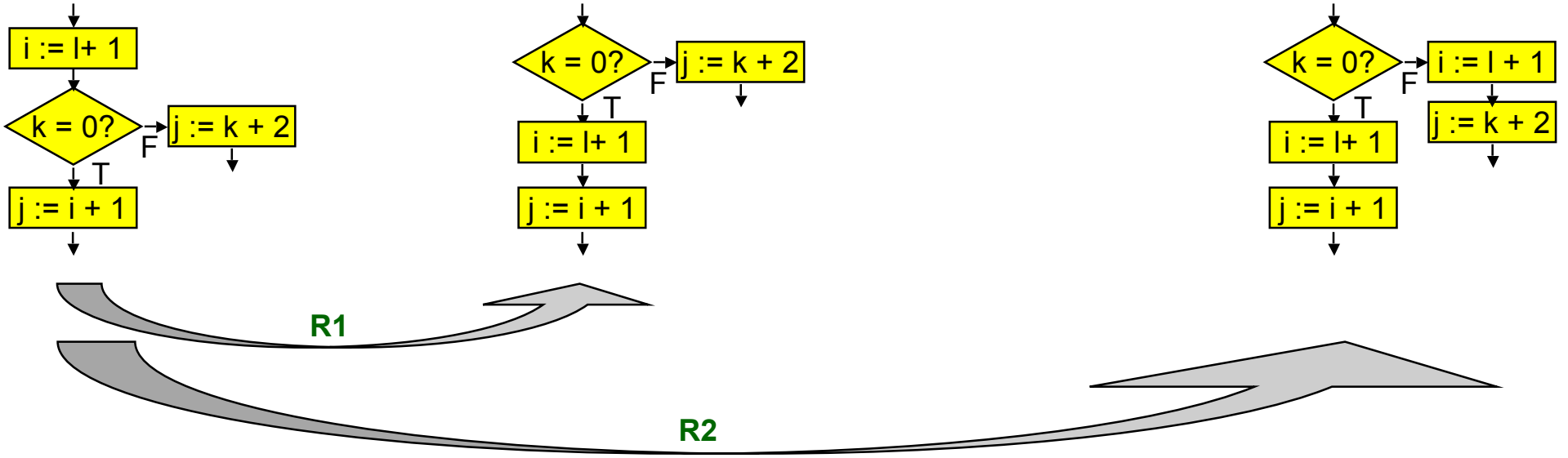
- Hier
  - Zwei mögliche Wege, nämlich s_0; b; s_1; s_3 und s_0; b; s_2; s_3
  - Verallgemeinerter Abhängigkeitsgraph enthält alle Abhängigkeiten zwischen Benutzungen und Setzungen auf diesen beiden Wegen
- Vorgehensweise: Modifikation des Algorithmus BBA-Graph
  - Beide Wege (s_1 und s_2) werden von hinten mit denselben Werten von LS und EB betreten
  - Danach: Vereinigung der aktuellen Werte von LS und EB und Verarbeitung der Bedingung

Schleifen (s_0; **while** b **do** s_1 **od**; s_2)

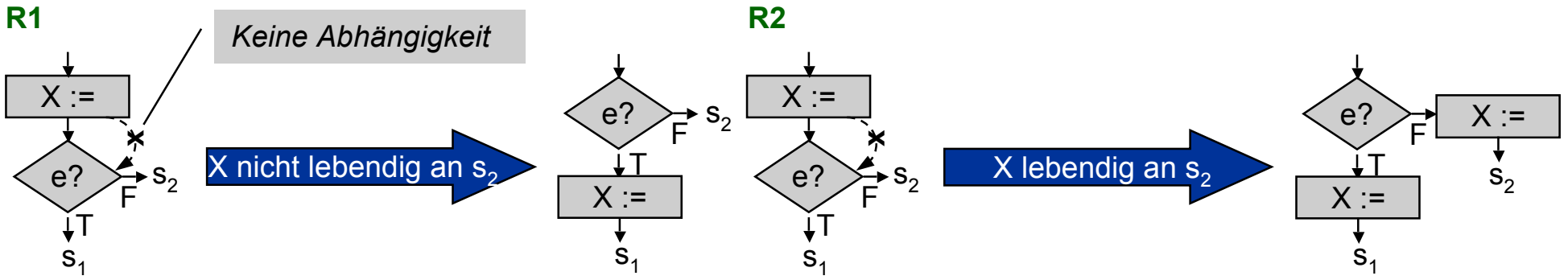
- Hier
  - Potentiell unendlich viele Wege, nämlich s_0; b; (s_1; b)* s_2
  - Für die Berechnung der Abhängigkeiten ausreichend: s_0; b; s_2 und s_0; b; s_1; b; s_1; b; s_2
- Vorgehensweise: Modifikation von BBA-Graph
  - Jede Schleife wird zweimal durchlaufen
    - 1. mit den hinter ihr gültigen Werten von LS und EB
    - 2. mit den neuen Werten, die sich nach Bearbeitung der Schleifenbedingung ergeben haben

Kompensationscode (1/2)

Beispiel (Codeumordnung mit Einfügen von Kompensationscode)

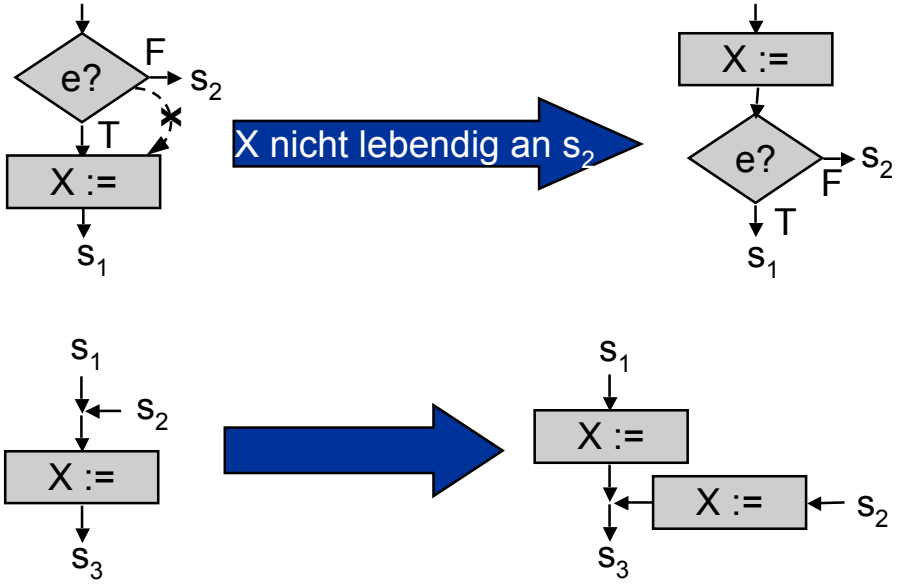


Umordnungsregeln

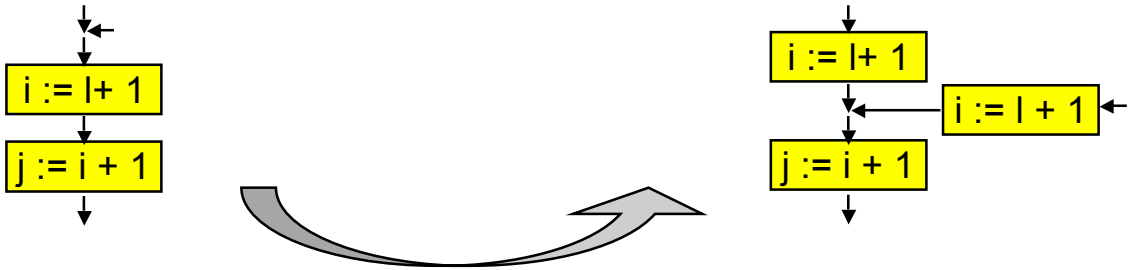


Kompensationscode (2/2)

Weitere Umordnungsregeln

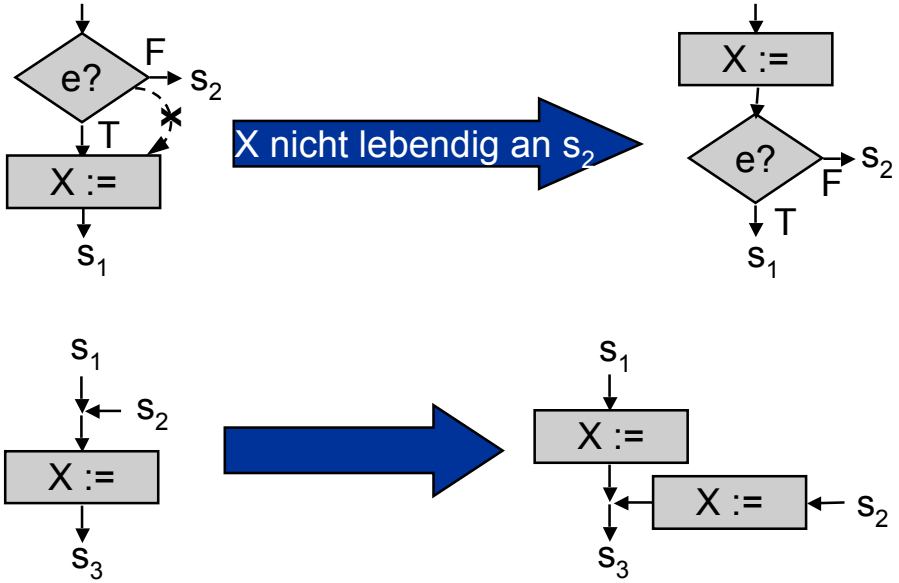


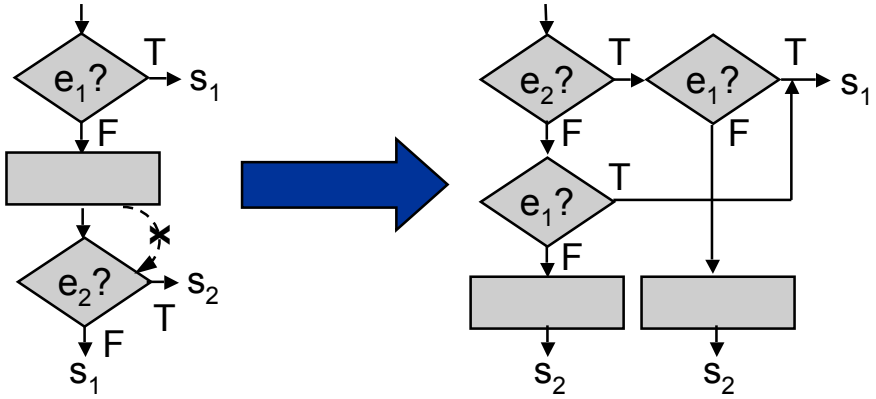
Beispiel (Anwendung der letzteren Regel)



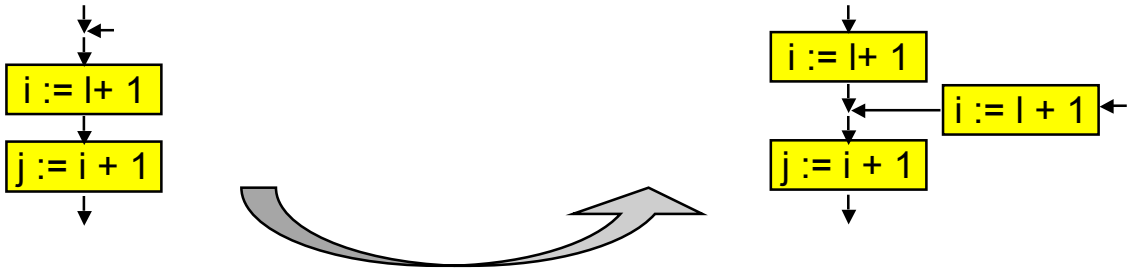
Kompensationscode (2/2)

Weitere Umordnungsregeln





Beispiel (Anwendung der letzteren Regel)



Realistische VLIW-Rechner

Zusätzliche Probleme für den Codeerzeuger

Aufgabe der bisherigen Annahmen

- Endliche Anzahl funktionaler Einheiten (FEs)
  - Problem: Bei der Anordnung des erweiterten Abhängigkeitsgraphen in möglichst wenigen Ebenen muss die endliche Breite der Ebenen (festgelegt durch die Anzahl der FEs) berücksichtigt werden
  - Heuristik: Bevorzugte Platzierung der Befehle auf dem längsten Pfad (wenn es mehr Kandidaten als funktionale Einheiten gibt)
- Unterschiedliche Ausführungszeiten von Befehlen
  - Markierung der Kanten des Abhängigkeitsgraphen mit nötigen Verzögerungen
  - Heuristik: Bevorzugte Anordnung der Befehle auf dem kritischen Pfad (= Pfad mit maximaler Summe seiner Verzögerungen)
- Beschränkte Bandbreite des Registersatzes und des Speichers
  - Da nicht jede FE zu jedem Zeitpunkt auf jedes Register/Speicherstelle zugreifen kann, Aufteilung von Registersatz und Speicher in mehrere „Bänke“ (die teilweise verbunden sind); jede FE nur mit Teilmenge der „Bänke“ verbunden
  - Neue Aufgaben des des Codeerzeugers
    - Lokalität zwischen Registern, Speicher und FEs berücksichtigen
    - „Transportkosten“ zwischen Bänken minimieren