Beobachtung:

- In gängigen Prozessoren kann man Werte von jedem Register in jedes andere schieben $\Longrightarrow$

  Die Kosten zwischen Registern differieren nur um eine Konstante :-)

- Komplexe rechte Seiten lassen sich i.a. mittels elementarerer Instruktionen simulieren $\Longrightarrow$

  Die Kosten zwischen Teilausdrücken und Registern differieren nur um eine Konstante :-))

- Die Kostenberechnung ist additiv $\Longrightarrow$

  Wir können statt mit absoluten Kosten-Angaben auch mit Kosten-Differenzen rechnen !!!

  Von diesen gibt es nur endlich viele :-)

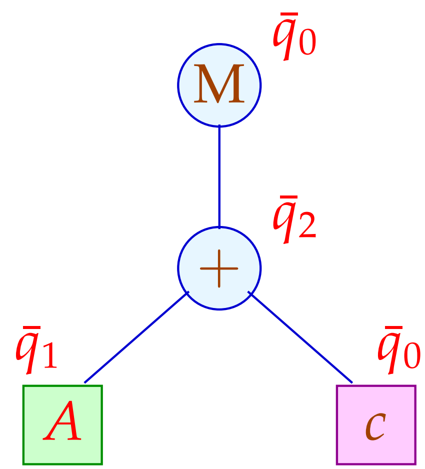
... im Beispiel:

$$\begin{array}{rcccl}
\delta_c & = & \{A \mapsto 1, D \mapsto 0\} & = & \bar{q}_0 \\
 & = & \delta_D & & \\
\delta_A & = & \{A \mapsto 0, D \mapsto 1\} & = & \bar{q}_1
\end{array}$$

$$\begin{array}{rcccl}
\delta_+(\bar{q}_1, \bar{q}_0) & = & \{A \mapsto 2, D \mapsto 1, A + A \mapsto 0\} & = & \bar{q}_2 \\
\delta_+(\bar{q}_0, \bar{q}_0) & = & \{A \mapsto 1, D \mapsto 0, A + A \mapsto 1\} & = & \bar{q}_3 \\
\delta_+(\bar{q}_1, \bar{q}_1) & = & \{A \mapsto 4, D \mapsto 3, A + A \mapsto 0\} & = & \bar{q}_4 \\
 & \ldots & & &
\end{array}$$

$$\begin{array}{rcccl}
\delta_M(\bar{q}_2) & = & \{A \mapsto 1, D \mapsto 0\} & = & \bar{q}_0 \\
 & = & \delta_M(\bar{q}_i) & , & i = 0, \ldots, 4
\end{array}$$

... das liefert die folgende Berechnung:



Für jede Konstanten-Klasse c und jedes Register R in δ_c tabellieren wir die zu wählende billigste Berechnung:

$$c \; : \quad \{A \mapsto 5, 3, D \mapsto 3\}$$

Analog tabellieren wir für jeden Operator a , jedes $\tau \in \bar{Q}^k$ und jedes R in $\delta_a(\tau)$:

M	select_M
$\bar{q}_0$	$\{A \mapsto 5, 1, D \mapsto 1\}$
$\bar{q}_1$	$\{A \mapsto 5, 1, D \mapsto 1\}$
$\bar{q}_2$	$\{A \mapsto 5, 0, D \mapsto 0\}$
$\bar{q}_3$	$\{A \mapsto 5, 1, D \mapsto 1\}$
$\bar{q}_4$	$\{A \mapsto 5, 0, D \mapsto 0\}$

Für "$+$" ist die Tabelle besonders einfach:

$+$	$\bar{q}_j$
$\bar{q}_i$	$\{A \mapsto 5, 3, D \mapsto 3\}$

Problem:

- Für reale Instruktionssätze benötigt man leicht um die 1000 Zustände.
- Die Tabellen für mehrstellige Operatoren werden riesig :-(

$\Longrightarrow$ Wir benötigen Verfahren der Tabellen-Komprimierung ...

Tabellen-Kompression:

Die Tabelle für "$+$" sieht im Beispiel so aus:

$+$	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
$\bar{q}_0$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_4$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$

Die meisten Zeilen / Spalten sind offenbar ganz ähnlich ;-)

Idee 1: Äquivalenzklassen

Wir setzen $q \equiv_a q'$, genau dann wenn

$$\forall\, p : \quad \delta_a(q,p) = \delta_a(q',p) \quad \wedge \quad \delta_a(p,q) = \delta_a(p,q')$$
$$\wedge \quad \mathsf{select}_a(q,p) = \mathsf{select}_a(q',p) \quad \wedge \quad \mathsf{select}_a(p,q) = \mathsf{select}_a(p,q')$$

Im Beispiel:

$$Q_1 = \{\bar{q}_0, \bar{q}_2, \bar{q}_3, \bar{q}_4\}$$
$$Q_2 = \{\bar{q}_1\}$$

mit:

$+$	Q_1	Q_2
Q_1	$\bar{q}_3$	$\bar{q}_2$
Q_2	$\bar{q}_2$	$\bar{q}_4$

Idee 2: Zeilenverschiebung

Sind viele Einträge gleich (im Beispiel etwa default $= \bar{q}_3$), genügt es, die übrigen Einträge zu speichern ;-)

Im Beispiel:

$+$	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
$\bar{q}_0$		$\bar{q}_2$			
$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
$\bar{q}_2$		$\bar{q}_2$			
$\bar{q}_3$		$\bar{q}_2$			
$\bar{q}_4$		$\bar{q}_2$			

Dann legen wir:

(1) gleiche Zeilen übereinander;

(2) verschiedene (Klassen von) Zeilen auf Lücke verschoben übereinander:

	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
class	0	1	0	0	0

	0	1
disp	0	2

	0	1	2	3	4	5	6
A	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
valid	0	0	1	1	1	1	1

Für jeden Eintrag im ein-dimensionalen Feld A vermerken wir in valid , zu welcher Zeile der Eintrag gehört ...

Ein Feld-Zugriff $\delta_+(\bar{q}_i, \bar{q}_j)$ wird dann so realisiert:

$$\begin{array}{lll} \delta_+(\bar{q}_i, \bar{q}_j) & = & \mathsf{let} \quad c \; = \; \mathsf{class}[\bar{q}_i]; \\ & & \qquad\; d \; = \; \mathsf{disp}[c]; \\ & & \mathsf{in} \;\; \mathsf{if} \;\; (\mathsf{valid}[d+j] \equiv c) \\ & & \qquad \mathsf{then} \;\; \mathsf{A}[d+j] \\ & & \qquad \mathsf{else} \;\; \mathsf{default} \\ & & \mathsf{end} \end{array}$$

Reinhard Wilhelm, Saarbrücken

Diskussion:

- Die Tabellen werden i.a. erheblich kleiner.
- Dafür werden Tabellenzugriffe etwas teurer.
- Das Verfahren versagt in einigen (theoretischen) Fällen.
- Dann bleibt immer noch das dynamische Verfahren ...

  möglicherweise mit Caching der einmal berechneten Werte, um unnötige Mehrfachberechnungen zu vermeiden :-)

3.3 Instruction Level Parallelität

Moderne Prozessoren führen nicht eine Instruktion nach der anderen aus.

Wir betrachten hier zwei Ansätze:

(1) VLIW (Very Large Instruction Words)
(2) Pipelining

VLIW:

Eine Instruktion führt simultan bis zu k (etwa 4:-) elementare Instruktionen aus.

Pipelining:

Instruktionsausführungen können zeitlich überlappen.

Beispiel:

$$w = (R_1 = R_2 + R_3 \mid D = D_1 * D_2 \mid R_3 = M[R_4])$$

Achtung:

- Instruktionen belegen Hardware-Einrichtungen.
- Instruktionen greifen auf die gleichen Register zu $\Longrightarrow$ Hazards
- Ergebnisse einer Insgtruktion liegen erst nach einiger Zeit vor.
- Während dieser Zeit wechselt i.a. die benutzte Hardware:

Fetch $\rightarrow$ Decode $\rightarrow$ Execute $\rightarrow$ Write

- Während Execute bzw. Write werden evt. unterschiedliche interne Register/Busse/Alus benutzt.

Wir schließen:

Aufteilung der Instruktionsfolge in Wörter und ihre Aufeinanderfolge ist Restriktionen unterworfen ...

Im folgenden ignorieren wir die Phasen Fetch und Decode :-)

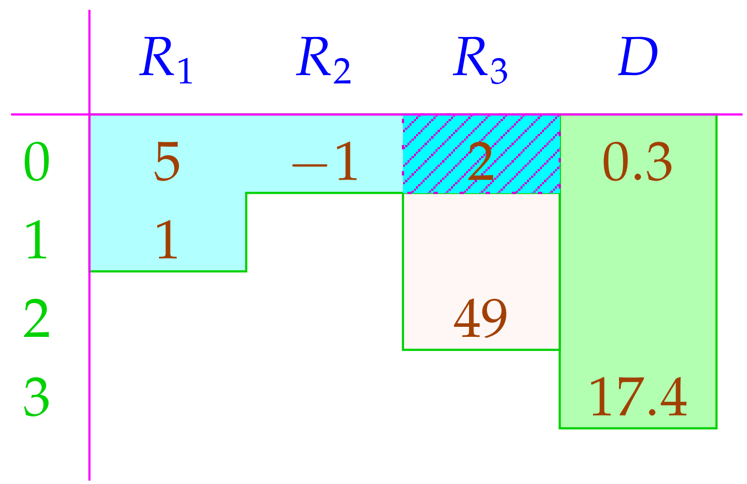
Beispiele für Restriktionen:

(1) maximal ein Load/Store pro Wort;
(2) maximal ein Jump;
(3) maximal ein Write in das selbe Register.

Timing:

Gleitkomma-Operation	3
Laden/Speichern	2
Integer-Arithmetik	1

Timing-Diagramm:



R_3 wird überschrieben, nachdem die Addition 2 abgeholte :-)

Wird auf ein Register mehrfach zugegriffen (hier: R_3), wird eine Strategie zur Konfliktlösung benötigt ...

Konflikte:

Read-Read: Ein Register wird mehrfach ausgelesen.

$\Longrightarrow$ i.a. unproblematisch :-)

Read-Write: Ein Register wird in einer Instruktion sowohl gelesen wie geschrieben.

Lösungsmöglichkeiten:

- ... verbieten!
- Lesen wird verzögert (stalls), bis Schreiben beendet ist!
- Lesen zeitlich vor dem Schreiben liefert den alten Wert!

  Gleichzeitiges Lesen wird verzögert/verboten/bevorzugt.

Write-Write: Ein Register wird mehrfach beschrieben.

$\Longrightarrow$ i.a. unproblematisch :-)

Lösungsmöglichkeiten:

- ... verbieten!
- ...

In unseren Beispielen ...

- erlauben wir gleichzeitiges Lesen;
- verbieten wir gleichzeitiges Schreiben bzw. Schreiben und Lesen;
- fügen wir keine Stalls ein.

Wir betrachten erst mal nur Basis-Blöcke, d.h. Folgen von Zuweisungen ...

Idee: Datenabhängigkeitsgraph

Knoten	Instruktionen
Kanten	Abhängigkeiten

Beispiel:

$$
\begin{array}{ll}
(1) & x = x + 1; \\
(2) & y = M[A]; \\
(3) & t = z; \\
(4) & z = M[A + x]; \\
(5) & t = y + z;
\end{array}
$$

Mögliche Abhängigkeiten:

Definition	$\longrightarrow$	Use	//	Reaching Definitions
Use	$\longrightarrow$	Definition	//	???
Definition	$\longrightarrow$	Definition	//	Reaching Definitions

Reaching Definitions: Ankommende Definitionen

Ermittle für jedes u , welche Variablen-Definitionen ankommen

$\Longrightarrow$ mithilfe Ungleichungssystem berechenbar :-)

Der abstrakte Bereich:

$\mathbb{R} = 2^{Nodes}$ // Man hätte auch Kanten nehmen können :-)

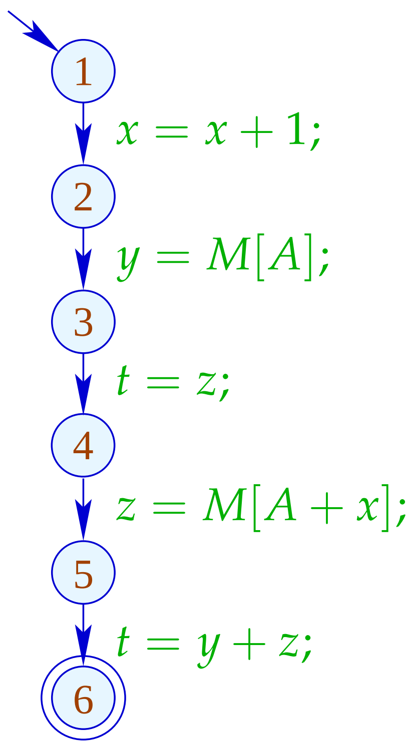
Die Transfer-Funktionen:

$$
\begin{array}{lll}
[\![(\_, ;, \_)]\!]^\sharp\, R & = & R \\
[\![(\_, Pos(e), \_)]\!]^\sharp\, R & = & [\![(\_, Neg(e), \_)]\!]^\sharp\, R \quad = \quad R \\
[\![(u, x = e;, \_)]\!]^\sharp\, R & = & (R \backslash Defs_x) \cup \{u\} \qquad \text{wobei} \\
 & & Defs_x \text{ die Menge der Definitionen von } x \text{ ist} \\
[\![(u, x = M[A];, \_)]\!]^\sharp\, R & = & (R \backslash Defs_x) \cup \{u\} \\
[\![(\_, M[A] = x;, \_)]\!]^\sharp\, R & = & R
\end{array}
$$

Die Information wird offenbar vorwärts propagiert, wobei die Ordnung auf dem vollständigen Verband $\mathbb{R}$ "$\subseteq$" ist :-)

Vor Programm-Ausführung ist die Menge der ankommenden Definitionen $d_0 = \{\bullet_x \mid x \in \mathit{Vars}\}$.

... im Beispiel:



	$\mathcal{R}$
1	$\{\bullet_x, \bullet_y, \bullet_z, \bullet_t\}$
2	$\{1, \bullet_y, \bullet_z, \bullet_t\}$
3	$\{1,2, \bullet_z, \bullet_t\}$
4	$\{1,2,3, \bullet_z\}$
5	$\{1,2,3,4\}$
6	$\{1,2,4,5\}$

Seien U_i, D_i die Mengen der an einer von u_i ausgehenden Kante benutzten bzw. definierten Variablen. Dann gilt:

$$(u_1, u_2) \in DD \qquad \text{falls} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap D_2 \neq \emptyset$$
$$(u_1, u_2) \in DU \qquad \text{falls} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap U_2 \neq \emptyset$$

... im Beispiel:

		Def	*Use*
1	$x = x + 1;$	$\{x\}$	$\{x\}$
2	$y = M[A];$	$\{y\}$	$\{A\}$
3	$t = z;$	$\{t\}$	$\{z\}$
4	$z = M[A + x];$	$\{z\}$	$\{A, x\}$
5	$t = y + z;$	$\{t\}$	$\{y, z\}$