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:

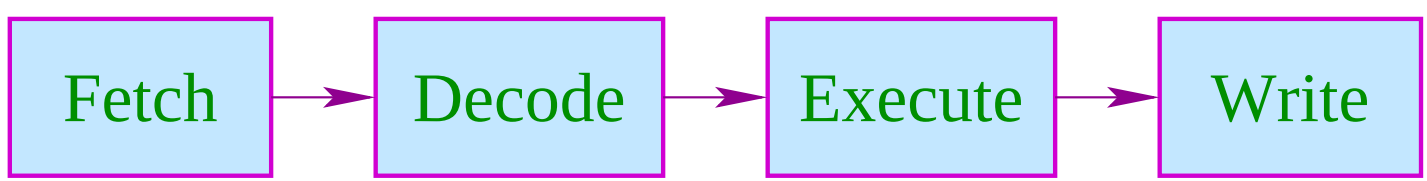
Instruktionsausführungen können zeitlich überlappen.

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:



- 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 :-)

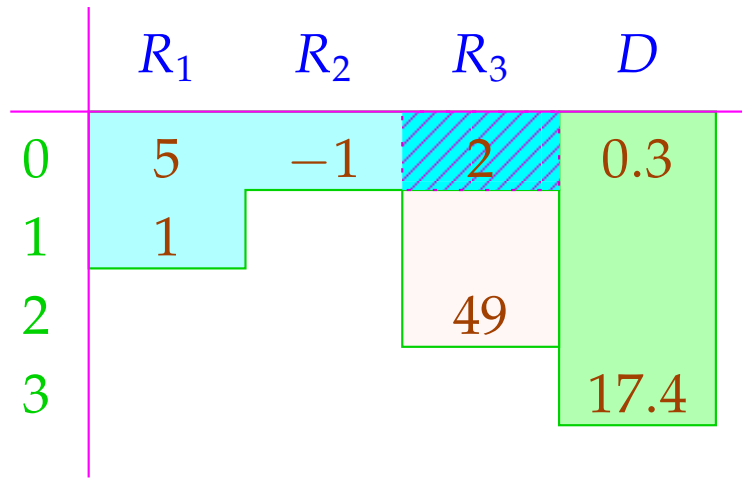
Beispiele für Restriktionen:

(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:

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

Mögliche Abhängigkeiten:

Definition	$\rightarrow$	Use	//	Reaching Definitions
Use	$\rightarrow$	Definition	//	???
Definition	$\rightarrow$	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 :-)

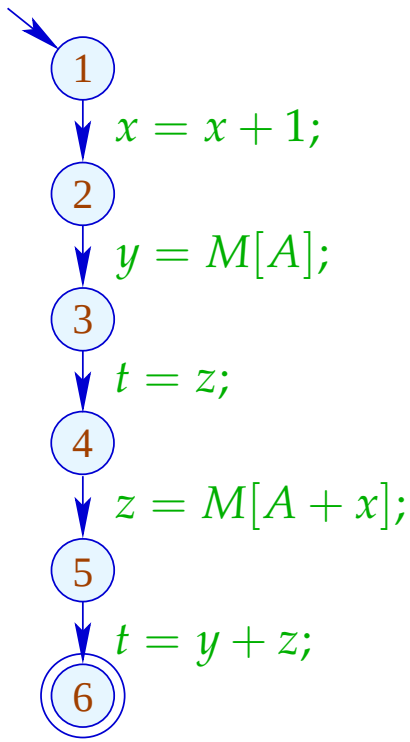
Die Transfer-Funktionen:

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

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:



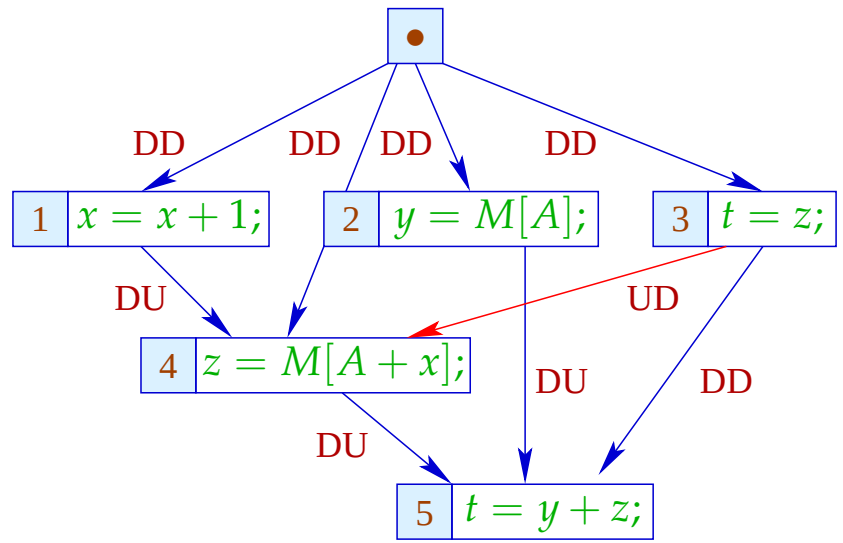
	$\mathcal{R}$
1	$\{\bullet_x, \bullet_y, \bullet_z, \bullet_t\}$
2	$\{1, \bullet_y, \bullet_z, \bullet_t\}$
3	$\{1,2, \bullet_z, \bullet_t\}$
4	$\{1,2,3, \bullet_z\}$
5	$\{1,2,3,4\}$
6	$\{1,2,4,5\}$

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	$t = y + z;$	$\{t\}$	$\{y, z\}$



Die UD-Kante $(3, 4)$ haben wir eingefügt, um zu verhindern, dass z vor der Benutzung überschrieben wird :-)

Im nächsten Schritt versehen wir jede Instruktion mit (ihren benötigten Resourcen, insbesondere) ihrer Zeit.

Wir wollen eine möglichst parallele korrekte Wortfolge bestimmen.

Dazu verwalten wir den aktuellen System-Zustand:

$$\Sigma : \mathit{Vars} \to \mathbb{N}$$

$$\Sigma(x) \;\hat{=}\; \text{zu wartende Zeit, bis } x \text{ vorliegt}$$

Am Anfang:

$$\Sigma(x) \;=\; 0$$

Wir müssen als Invariante garantieren, dass alle Operationen bei Betreten des Basisblocks abgeschlossen sind :-)

Dann füllen wir sukzessive die Slots der Wort-Folge:

- Wir beginnen bei den minimalen Knoten des Abhängigkeitsgraphen.
- Können wir nicht alle Slots eines Worts füllen, fügen wir ; ein :-)
- Nach jeder eingefügten Instruktion berechnen wir Σ neu.

Achtung:

→ Die Ausführung zweier VLIWs kann überlappen !!!

→ Die Berechnung einer optimalen Folge ist NP-hart ...

Beispiel: Wortbreite $k = 2$

Wort		Zustand			
1	2	x	y	z	t
		0	0	0	0
$x = x + 1$	$y = M[A]$	0	1	0	0
$t = z$	$z = M[A + x]$	0	0	1	0
		0	0	0	0
$t = y + z$		0	0	0	0

In jedem Takt beginnt die Ausführung eines neuen Worts.

Im Zustand brauchen wir uns nur merken, wieviele Takte auf das Ergebnis noch gewartet werden muss :-)

Beachte:

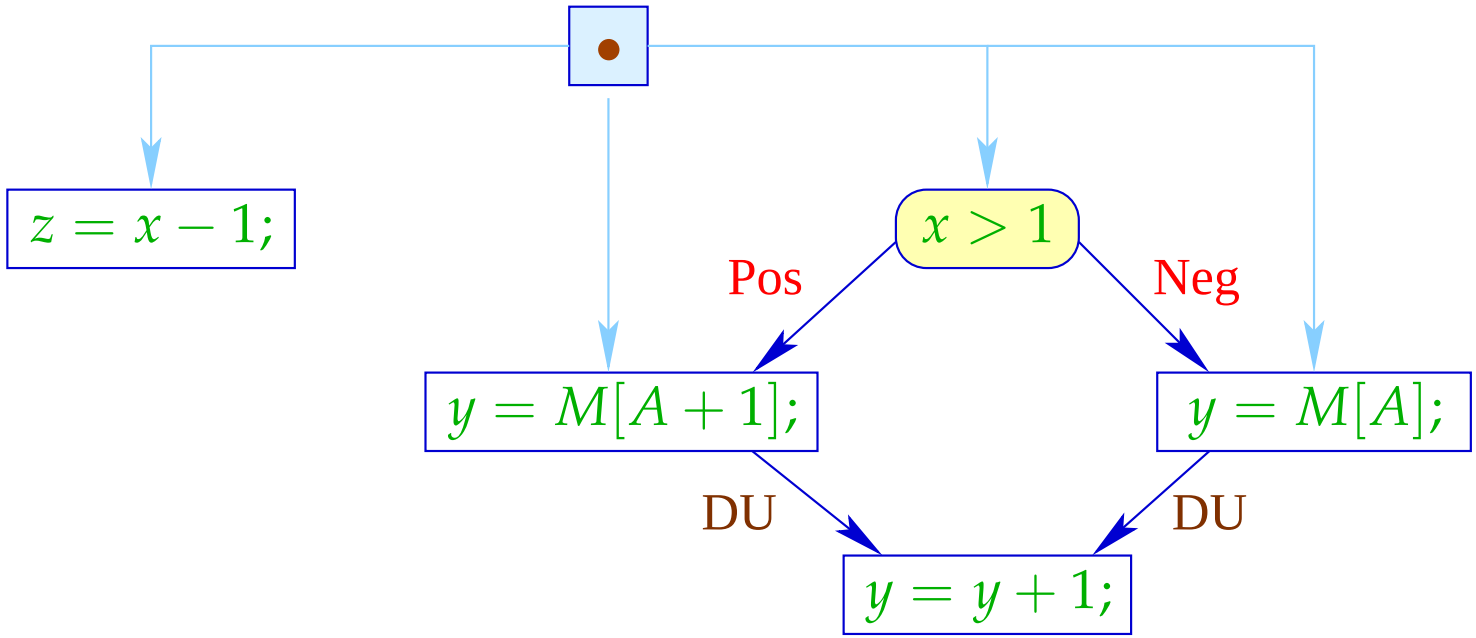
- Wenn Instruktionen zukünftiger Wortwahl weitere Restriktionen auferlegen, vermerken wir diese ebenfalls in Σ .
- Trotzdem unterscheiden wir nur endlich viele System-Zustände :-)
- Die Berechnung des Effekts eines VLIW auf Σ lässt sich in einen endlichen Automaten compilieren !!!
- Dieser Automat könnte allerdings sehr groß sein :-(
- Die Qual der billigsten Auswahl erspart er uns nicht :-(
- Basis-Blöcke sind leider i.a. nicht sehr groß

  $\Longrightarrow$ die Möglichkeiten zur Parallelisierung sind beschränkt :-((

Erweiterung 1: Azyklischer Code

$$
\begin{array}{l}
\textsf{if}\ \ (x > 1)\ \ \{ \\
\qquad y = M[A]; \\
\qquad z = x - 1; \\
\}\ \ \textsf{else}\ \ \{ \\
\qquad y = M[A+1]; \\
\qquad z = x - 1; \\
\} \\
y = y + 1;
\end{array}
$$

Im Abhängigkeitsgraph müssen wir zusätzlich die Kontroll-Abhängigkeiten vermerken ...



Das Statement $z = x - 1;$ wird mit immer den gleichen Argumenten in beiden Zweigen ausgeführt und modifiziert keine der sonst benutzten Variablen :-)

Wir hätten es ohnehin vor das if schieben können :-))

Als Code können wir deshalb erzeugen:

	$z = x - 1$	if $(!(x > 0))$ goto A
	$y = M[A]$	
	goto B	
$A:$	$y = M[A+1]$	
$B:$	$y = y + 1$	

Bei jedem Einsprung garantieren wir die Invariante :-(

Erlauben wir mehrere (bekannte) Zustände beim Betreten eines Teil-Basisblocks, können wir für diesen Code erzeugen, der allen diesen Bedingungen entspricht.

... im Beispiel:

	$z = x - 1$	if $(!(x > 0))$ goto A
	$y = M[A]$	goto B
$A:$	$y = M[A+1]$	
$B:$		
	$y = y + 1$	

Reicht uns diese Parallelität immer noch nicht, könnten wir versuchen, spekulativ Arbeit vorziehen ...

Dazu erforderlich:

- eine Idee, welche Alternative häufiger gewählt wird;
- die falsche Ausführung darf zu keiner Katastrophe d.h. Laufzeitfehlern führen (z.B. wegen Division durch 0);
- die falsch Ausführung muss rückgängig gemacht werden können (evt. durch verzögertes Commit) oder darf keinen beobachtbaren Effekt haben ...

... im Beispiel:

	$z = x - 1$	$y = M[A]$	if $(x > 0)$ goto B
	$y = M[A + 1]$		
$B:$			
	$y = y + 1$		

Im Fall $x \leq 0$ haben wir $y = M[A]$ zuviel ausgeführt.

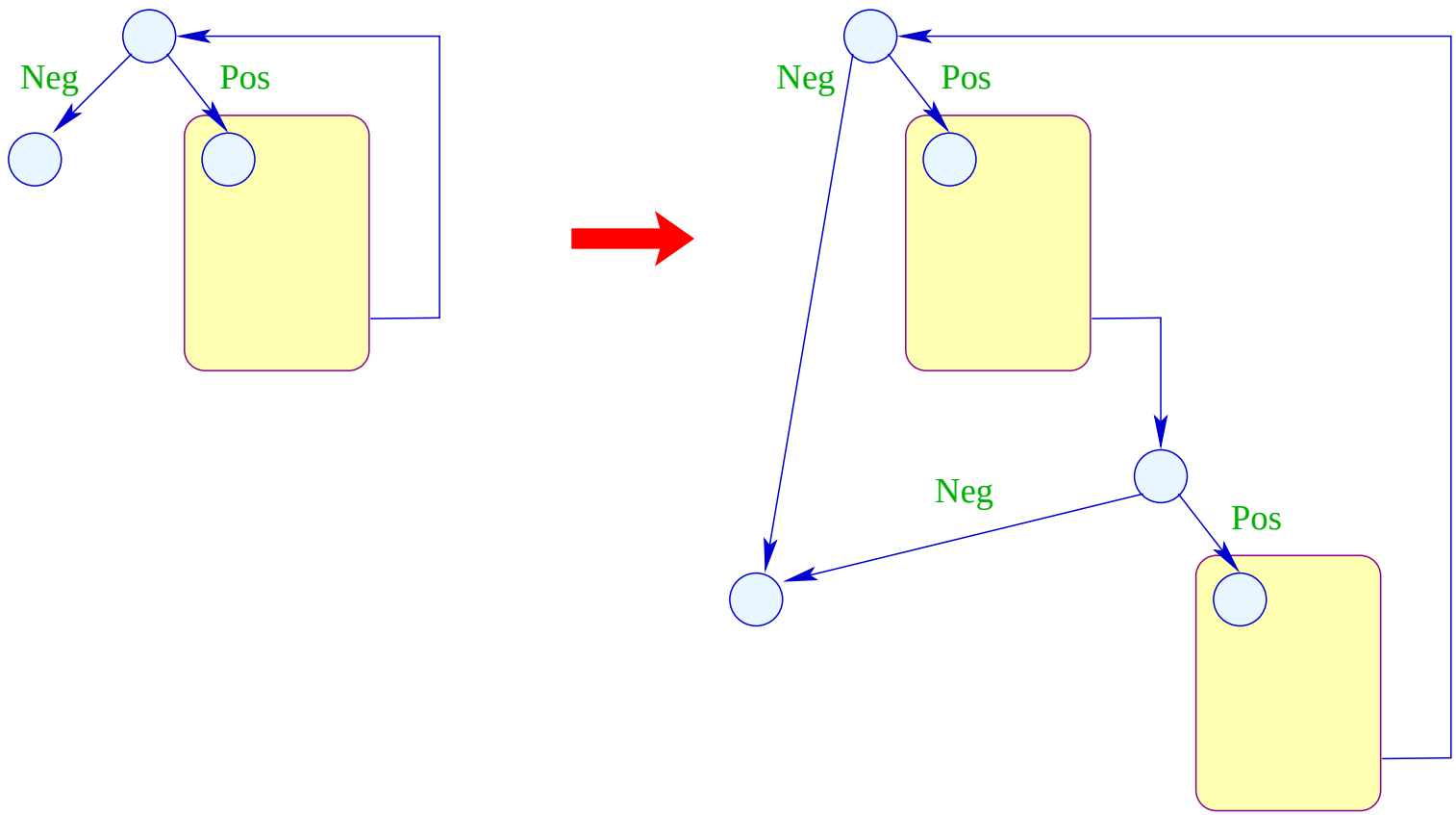
Dieser Wert wird aber im nächsten Schritt direkt überschrieben :-)

Allgemein:

$x = e;$ hat keinen beobachtbaren Effekt in einem Zweig, falls x in diesem Zweig tot ist :-)

Erweiterung 2: Abwickeln von Schleifen

Wir wickeln wichtige, d.h. innere Schleifen mehrmals ab:



Nun ist auch klar, welche Seite bei Tests zu begünstigen ist:

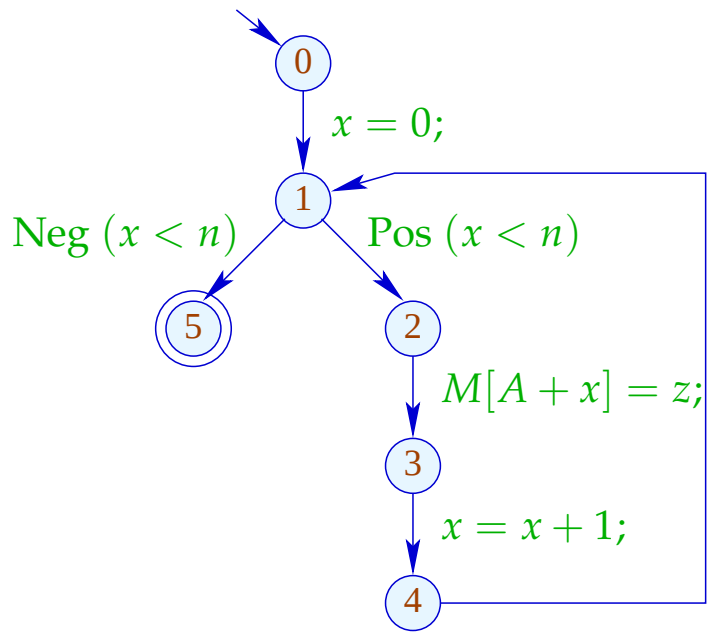
diejenige, die innerhalb des abgerollten Rumpfs der Schleife bleibt :-)

Achtung:

- Die verschiedenen Instanzen des Rumpfs werden relativ zu möglicherweise unterschiedlichen Anfangszuständen übersetzt :-)
- Der Code hinter der Schleife muss gegenüber dem Endzustand jedes Sprungs aus der Schleife korrekt sein!

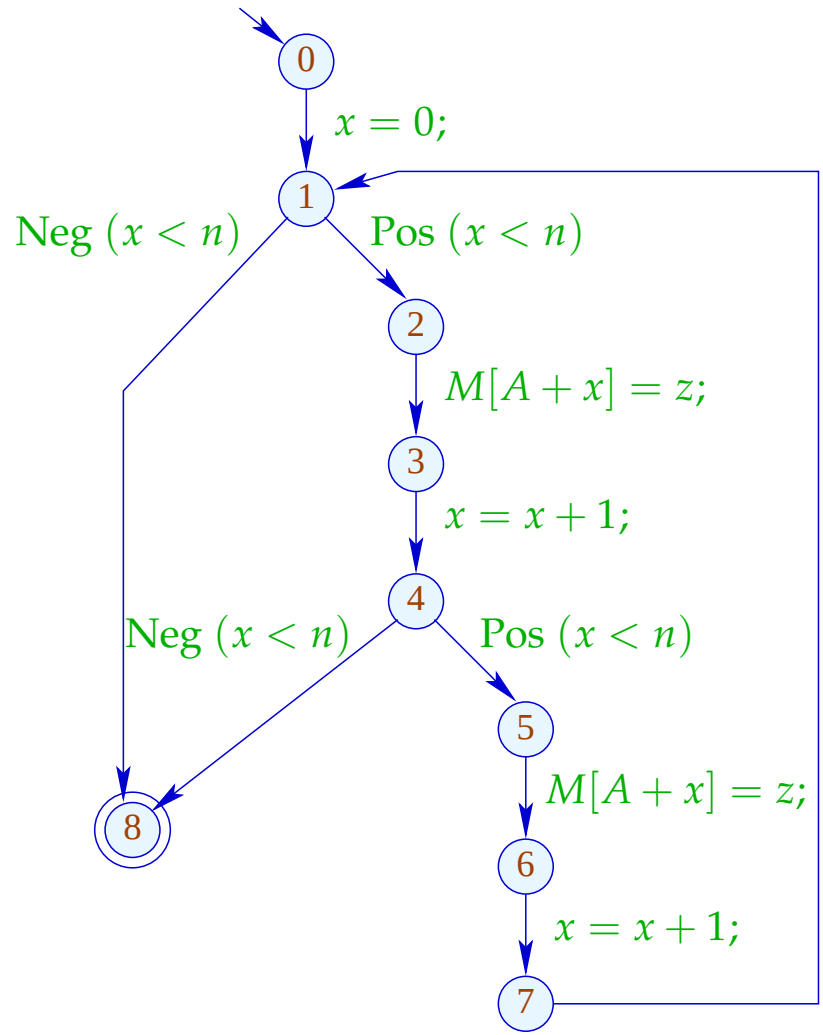
Beispiel:

$$\text{for } (x = 0; x < n; x{+}{+})$$
$$M[A + x] = z;$$



Verdoppelung des Rumpfs liefert:

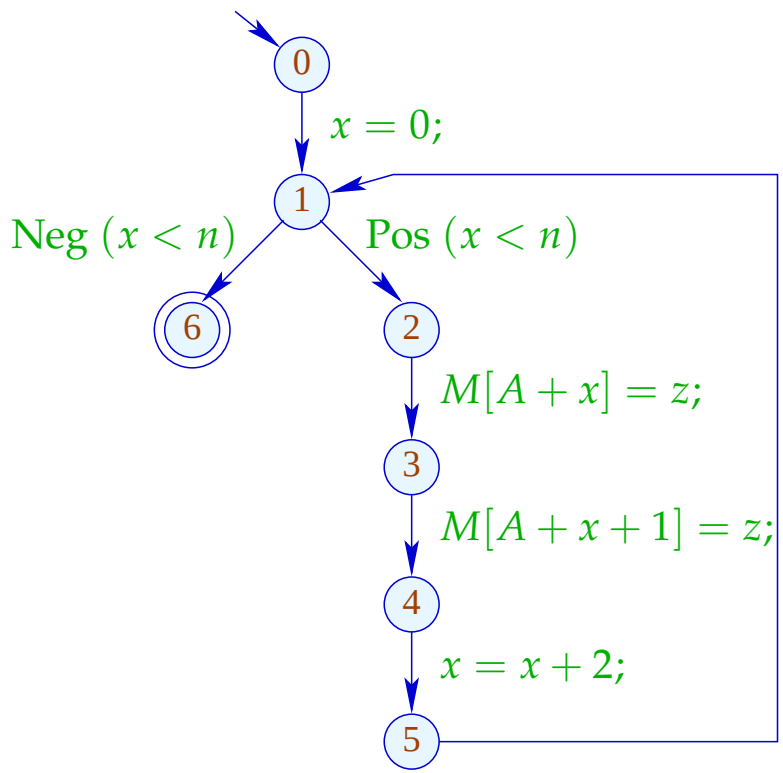
$$
\begin{aligned}
&\text{for}\ \ (x = 0; x < n; x{+}{+})\ \ \{\\
&\qquad M[A + x] = z;\\
&\qquad x = x + 1;\\
&\qquad \text{if}\ \ (!(x < n))\ \ \text{break};\\
&\qquad M[A + x] = z;\\
&\}
\end{aligned}
$$



Besser wäre es, wenn wir auf den Test in der Mitte verzichten könnten. Das ist möglich, wenn wir wissen, dass n stets gerade ist :-)

Dann haben wir:

$$
\begin{array}{l}
\text{for } (x = 0; x < n; x = x + 2) \ \{ \\
\qquad M[A + x] = z; \\
\qquad M[A + x + 1] = z; \\
\}
\end{array}
$$



Diskussion:

- Beseitigung der Zwischenabfrage zusammen mit Verschieben des Zwischen-Inkrements ans Ende zeigt, dass die verschiedenen Rumpf-Iterationen in Wahrheit unabhängig sind :-)
- Wir gewinnen trotzdem nicht viel, da wir nur maximal ein Store pro Wort gestatten :-(
- Sind die rechten Seiten allerdings komplizierter, könnten wir deren Auswertung mit je einem Store pro Takt verschränken :-)

Erweiterung 3:

Möglicherweise bietet eine Schleife allein nicht genug Möglichkeiten zur Parallelisierung :-(

... möglicherweise aber zwei aufeinander folgende :-)

Beispiel:

$$
\begin{array}{l}
\text{for } (x = 0; x < n; x{+}{+}) \ \{ \\
\qquad R = M[B + x]; \\
\qquad S = M[C + x]; \\
\qquad T_1 = R + S; \\
\qquad M[A + x] = T_1; \\
\}
\end{array}
\qquad
\begin{array}{l}
\text{for } (x = 0; x < n; x{+}{+}) \ \{ \\
\qquad R = M[B + x]; \\
\qquad S = M[C + x]; \\
\qquad T_2 = R - S; \\
\qquad M[C + x] = T_2; \\
\}
\end{array}
$$

Um beide Schleifen zu einer zusammen zu fassen, muss:

- das Iterations-Schema übereinstimmen;
- die beiden Schleifen greifen auf unterschiedliche Daten zu.

Im Falle von einzelnen Variablen lässt sich das leicht verfizieren.

Schwieriger ist das in Anwesenheit von Pointern oder Feldern.

Unter Rückgriff auf das Source-Programm kann man Zugriffe auf statisch allokierte disjunkte Felder erkennen.

Analyse von Zugriffen auf das gleiche Feld ist erheblich schwieriger ...

Nehmen wir für das Beispiel an, die Bereiche
$[A, A+n-1], [B, B+n-1], [C, C+n-1]$ überlappen nicht.

Offenbar können wir dann die beiden Schleifen kombinieren zu:

$$
\begin{array}{lll}
\textsf{for}\ \ (x = 0; x < n; x{+}{+})\ \ \{ & & \\
\qquad R = M[B + x]; & & R = M[B + x]; \\
\qquad S = M[C + x]; & & S = M[C + x]; \\
\qquad T_1 = R + S; & & T_2 = R - S; \\
\qquad M[A + x] = T_1; & & M[C + x] = T_2; \\
 & \} &
\end{array}
$$