## Discussion

- Although the example program is not in SSA form, all live ranges still form tree fragments :-)
- The intersection of tree fragments is again a tree fragment !
- A set $C$ of tree fragments forms a clique iff their intersection is non-empty !!!
- The greedy algorithm will find an optimal coloring ...

# Proof of the Intersection Property

(1) Assume $I_1 \cap I_2 \neq \emptyset$ and $v_i$ is the root of $I_i$. Then:

$$v_1 \in I_2 \quad \text{or} \quad v_2 \in I_1$$

(2) Let $C$ denote a clique of tree fragments.
Then there is an enumeration $C = \{I_1, \ldots, I_r\}$ with roots $v_1, \ldots, v_r$ such that

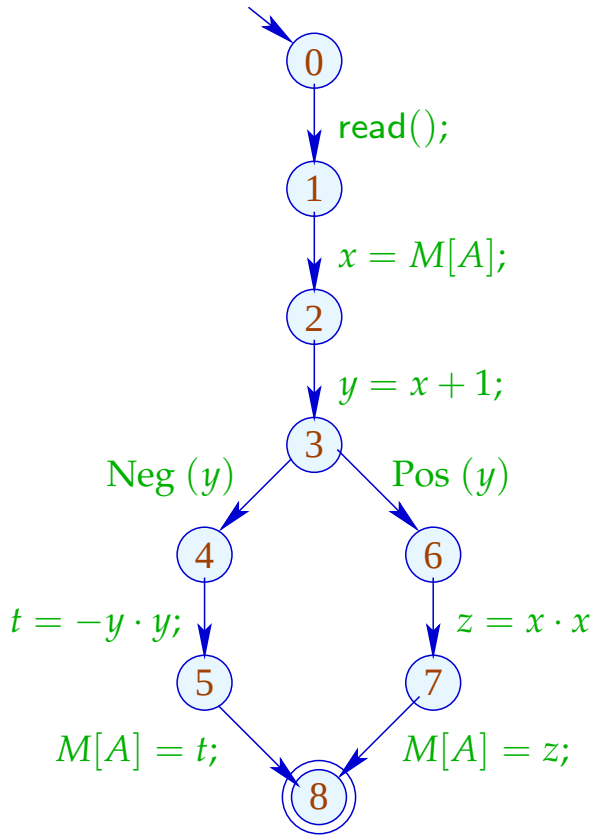
$$v_i \in I_j \qquad \text{for all} \quad j \leq i$$

In particular, $v_r \in I_i$ for all $i$. :-)

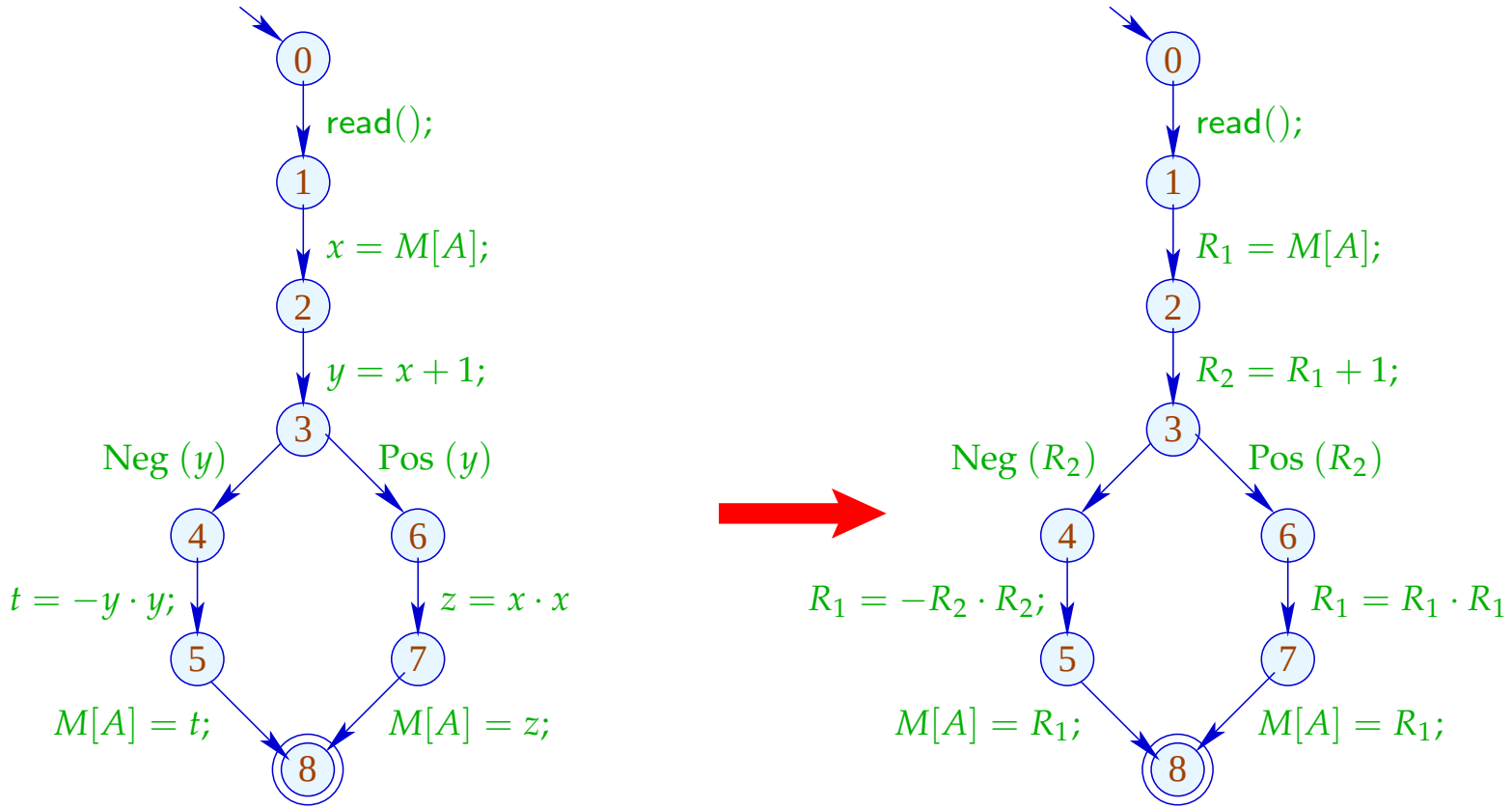
# The Greedy Algorithm

$$
\begin{array}{l}
\textbf{forall}\ (u \in \mathit{Nodes})\ \mathit{visited}[u] = \textbf{false};\\
\textbf{forall}\ (x \in \mathcal{L}[\mathit{start}])\ \Gamma(x) = \mathsf{extract}(\mathit{free});\\
\mathsf{alloc}(\mathit{start});\\
\\
\textbf{void}\ \mathsf{alloc}\ (\textbf{Node}\ u)\ \{\\
\quad \mathit{visited}[u] = \textbf{true};\\
\quad \textbf{forall}\ ((\mathit{lab}, v) \in \mathit{edges}[u])\\
\qquad \textbf{if}\ (\neg \mathit{visited}[v])\ \{\\
\qquad\quad \textbf{forall}\ (x \in \mathcal{L}[u] \backslash \mathcal{L}[v])\ \mathsf{insert}(\mathit{free}, x);\\
\qquad\quad \textbf{forall}\ (x \in \mathcal{L}[v] \backslash \mathcal{L}[u])\ \Gamma(x) = \mathsf{extract}(\mathit{free});\\
\qquad\quad \mathsf{alloc}\ (v);\\
\qquad\quad \}\\
\quad \}
\end{array}
$$

# Example



# Example


0
read();
1
$x = M[A];$
2
$y = x + 1;$
3
Neg $(y)$
Pos $(y)$
4
6
$t = -y \cdot y;$
$z = x \cdot x$
5
7
$M[A] = t;$
$M[A] = z;$
8
0
read();
1
$R_1 = M[A];$
2
$R_2 = R_1 + 1;$
3
Neg $(R_2)$
Pos $(R_2)$
4
6
$R_1 = -R_2 \cdot R_2;$
$R_1 = R_1 \cdot R_1$
5
7
$M[A] = R_1;$
$M[A] = R_1;$
8

## Remark:

- Intersection graphs for tree fragments are also known as cordal graphs ...
- A cordal graph is an undirected graph where every cycle with more than three nodes contains a cord :-)
- Cordal graphs are another sub-class of perfect graphs :-))

- Cheap register allocation comes at a price:

  when transforming into SSA form, we have introduced parallel register-register moves :-(

## Problem

The parallel register assignment:

$$\psi_1 = R_1 = R_2 \mid R_2 = R_1$$

is meant to exchange the registers $R_1$ and $R_2$ :-)

There are at least two ways of implementing this exchange ...

## Problem

The parallel register assignment:

$$\psi_1 = R_1 = R_2 \mid R_2 = R_1$$

is meant to exchange the registers $R_1$ and $R_2$ :-)

There are at least two ways of implementing this exchange ...

## (1) Using an auxiliary register:

$$\begin{aligned} R &= R_1; \\ R_1 &= R_2; \\ R_2 &= R; \end{aligned}$$

(2) XOR:

$$\begin{aligned} R_1 &= R_1 \oplus R_2; \\ R_2 &= R_1 \oplus R_2; \\ R_1 &= R_1 \oplus R_2; \end{aligned}$$

## (2) XOR:

$$\begin{aligned} R_1 &= R_1 \oplus R_2; \\ R_2 &= R_1 \oplus R_2; \\ R_1 &= R_1 \oplus R_2; \end{aligned}$$

But what about cyclic shifts such as:

$$\psi_k = R_1 = R_2 \mid \ldots \mid R_{k-1} = R_k \mid R_k = R_1$$

for $k > 2$ ??

## (2) XOR:

$$\begin{aligned} R_1 &= R_1 \oplus R_2; \\ R_2 &= R_1 \oplus R_2; \\ R_1 &= R_1 \oplus R_2; \end{aligned}$$

But what about cyclic shifts such as:

$$\psi_k = R_1 = R_2 \mid \ldots \mid R_{k-1} = R_k \mid R_k = R_1$$

for $k > 2$ ??

Then at most $k - 1$ swaps of two registers are needed:

$$\begin{aligned} \psi_k &= R_1 \leftrightarrow R_2; \\ & \quad R_2 \leftrightarrow R_3; \\ & \quad \ldots \\ & \quad R_{k-1} \leftrightarrow R_k; \end{aligned}$$

## Next complicated case: permutations.

- Every permutation can be decomposed into a set of disjoint shifts :-)
- Any permutation of $n$ registers with $r$ shifts can be realized by $n - r$ swaps ...

## Next complicated case: permutations.

- Every permutation can be decomposed into a set of disjoint shifts :-)
- Any permutation of $n$ registers with $r$ shifts can be realized by $n - r$ swaps ...

## Example

$$\psi = R_1 = R_2 \mid R_2 = R_5 \mid R_3 = R_4 \mid R_4 = R_3 \mid R_5 = R_1$$

consists of the cycles $(R_1, R_2, R_5)$ and $(R_3, R_4)$. Therefore:

$$\begin{aligned} \psi \quad = \quad & R_1 \leftrightarrow R_2; \\ & R_2 \leftrightarrow R_5; \\ & R_3 \leftrightarrow R_4; \end{aligned}$$

## The general case:

- Every register receives its value at most once.
- The assignment therefore can be decomposed into a permutation together with tree-like assignments (directed towards the leaves) ...

## Example

$$\psi = R_1 = R_2 \mid R_2 = R_4 \mid R_3 = R_5 \mid R_5 = R_3$$

The parallel assignment realizes the linear register moves for $R_1, R_2$ and $R_4$ together with the cyclic shift for $R_3$ and $R_5$:

$$\begin{aligned} \psi \quad = \quad & R_1 = R_2; \\ & R_2 = R_4; \\ & R_3 \leftrightarrow R_5; \end{aligned}$$

# Interprocedural Register Allocation:

→ For every local variable, there is an entry in the stack frame.

→ Before calling a function, these must be saved into the stack frame and be restored after the call.

→ Sometimes there is hardware support :-)

Then the call is transparent for all registers.

→ If it is our responsibility to save and restore, we may ...

- save only registers which are over-written :-)
- restore overwritten registers only.

→ Alternatively, we save only registers which are still live after the call — and then possibly into different registers $\Longrightarrow$ reduction of life ranges :-)

## 3.2 Instruction Level Parallelism

Modern processors do not execute one instruction after the other strictly sequentially.

Here, we consider two approaches:

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

## VLIW:

One instruction simultaneously executes up to $k$ (e.g., 4:-) elementary Instructions.

## Pipelining:

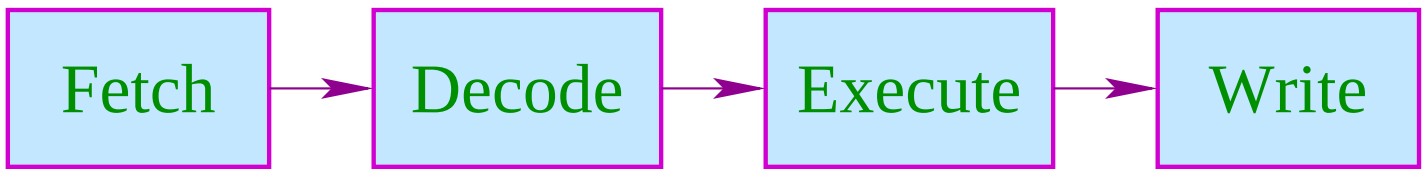
Instruction execution may overlap.

## Example:

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

## Warning:

- Instructions occupy hardware ressources.
- Instructions may access the same busses/registers $\Longrightarrow$ hazards
- Results of an instruction may be available only after some delay.
- During execution, different parts of the hardware are involved:




- During Execute and Write different internal registers/busses/alus may be used.

## We conclude:

Distributing the instruction sequence into sequences of words is amenable to various constraints ...

In the following, we ignore the phases Fetch und Decode :-)

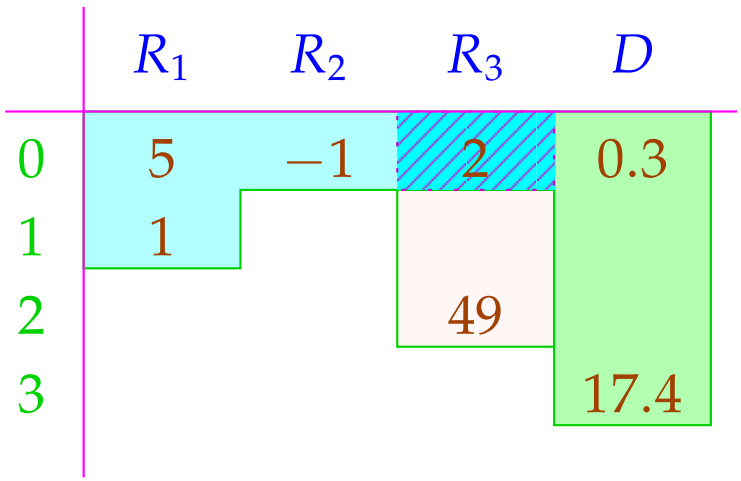
## Examples for Constraints:

(1) at most one load/store per word;
(2) at most one jump;
(3) at most one write into the same register.

## Example Timing:

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

## Timing Diagram:




$R_3$ is over-written, after the addition has fetched 2 :-)

If a register is accessed simultaneously (here: $R_3$), a strategy of conflict solving is required ...

## Conflicts:

**Read-Read:** A register is simulatneously read.

$\Longrightarrow$ in general, unproblematic :-)

**Read-Write:** A register is simultaneously read and written.

**Conflict Resolution:**

- ... ruled out!
- Read is delayed (stalls), until write has terminated!
- Read before write returns old value!

**Write-Write:** A register is simultaneously written to.

$\Longrightarrow$ in general, unproblematic :-)

**Conflict Resolutions:**

- ... ruled out!
- ...

## In Our Examples ...

- simultaneous read is permitted;
- simultaneous write/read and write/write is ruled out;
- no stalls are injected.

We first consider basic blocks only, i.e., linear sequences of assignments ...

Idea: Data Dependence Graph

| Vertices | Instructions |
|---|---|
| Edges | Dependencies |

Example:

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

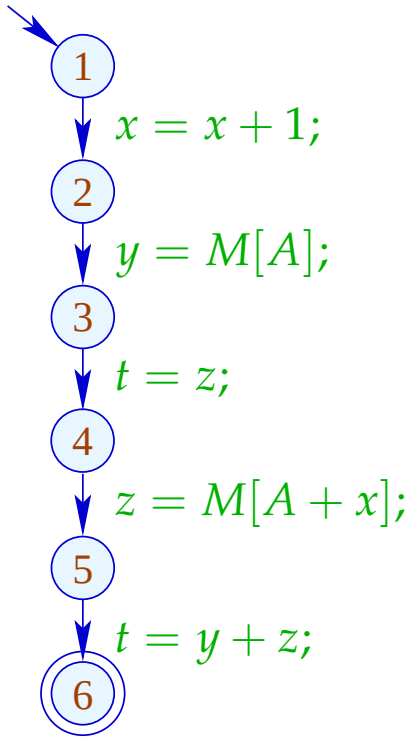
## Possible Dependencies:

| | | | | |
|---|---|---|---|---|
| Definition | $\rightarrow$ | Use | // | Reaching Definitions |
| Use | $\rightarrow$ | Definition | // | ??? |
| Definition | $\rightarrow$ | Definition | // | Reaching Definitions |

## Reaching Definitions:

Determine for each $u$ which definitions of may reach $\Longrightarrow$
can be determined by means of a system of constraints :-)

## ... in the Example:




| | $\mathcal{R}$ |
|---|---|
| 1 | $\{\langle x, 1\rangle, \langle y, 1\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$ |
| 2 | $\{\langle x, 2\rangle, \langle y, 1\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$ |
| 3 | $\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$ |
| 4 | $\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 1\rangle, \langle t, 4\rangle\}$ |
| 5 | $\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 5\rangle, \langle t, 4\rangle\}$ |
| 6 | $\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 5\rangle, \langle t, 6\rangle\}$ |

Let $U_i$, $D_i$ denote the sets of variables which are used or defined at the edge outgoing from $u_i$. Then:

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

## ... in the Example:

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