Interprocedural Register Allocation:

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

→ Before calling a function, the locals 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:

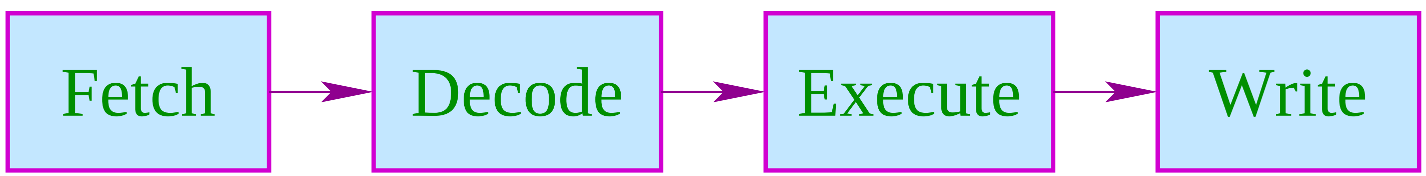
Instruction execution may overlap.

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 :-)

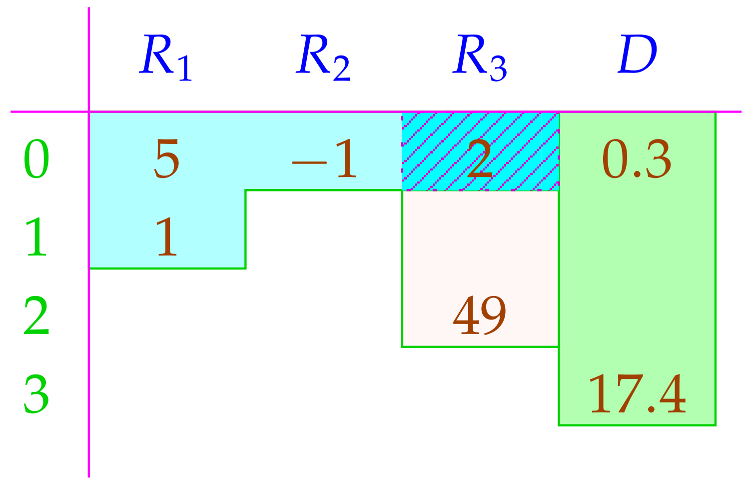
Examples for Constraints:

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

Example Timing:

Floating-point Operation	3
Load/Store	2
Integer Arithmetic	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:

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

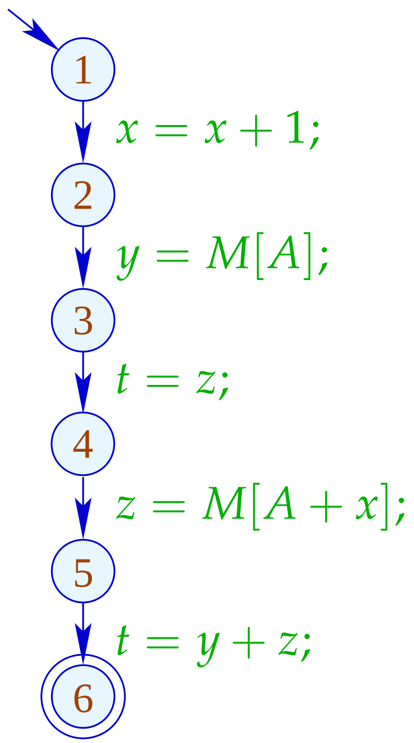
Possible Dependencies:

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

Reaching Definitions:

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

... in the Example:



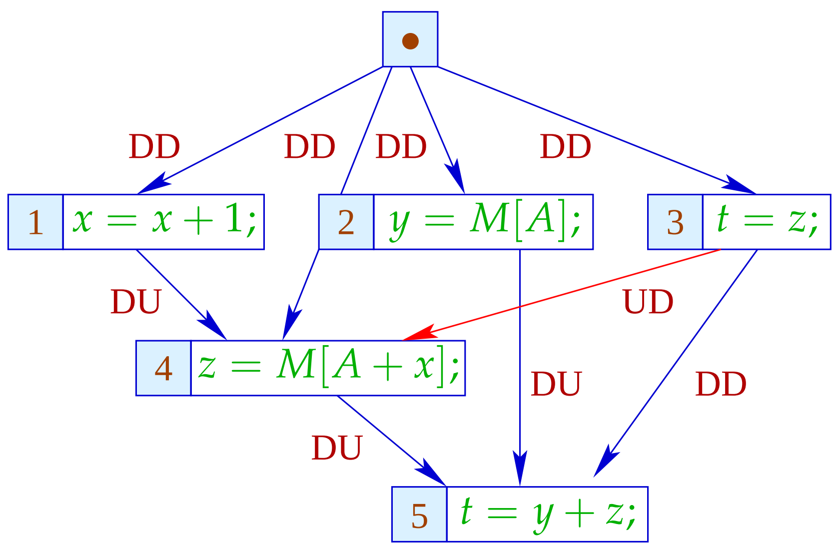
	$\mathcal{R}$
1	$\{\langle x, 1\rangle, \langle y, 1\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$
2	$\{\langle x, 2\rangle, \langle y, 1\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$
3	$\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$
4	$\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 1\rangle, \langle t, 4\rangle\}$
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:

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

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



The UD-edge $(3, 4)$ has been inserted to exclude that z is over-written before use :-)

In the next step, each instruction is annotated with its (required ressources, in particular, its) execution time.

Our goal is a maximally parallel correct sequence of words.

For that, we maintain the current system state:

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

$$\Sigma(x) \;\hat{=}\; \text{expected delay until } x \text{ is available}$$

Initially:

$$\Sigma(x) = 0$$

As an invariant, we guarantee on entry of the basic block, that all operations are terminated :-)

Then the slots of the word sequence are successively filled:

- We start with the minimal nodes in the dependence graph.
- If we fail to fill all slots of a word, we insert ; :-)
- After every inserted instruction, we re-compute Σ .

Warning:

→ The execution of two VLIWs can overlap !!!

→ Determining an optimal sequence, is NP-hard ...

Example: Word width $k = 2$

Word		State			
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 each cycle, the execution of a new word is triggered.

The state just records the number of cycles still to be waited for the result :-)