

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

Conflicts:

Read-Read: A register is simultaneously read.

⇒ 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.

⇒ 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:

(1) $x = x + 1;$

(2) $y = M[A];$

(3) $t = z;$

(4) $z = M[A + x];$

(5) $t = y + z;$

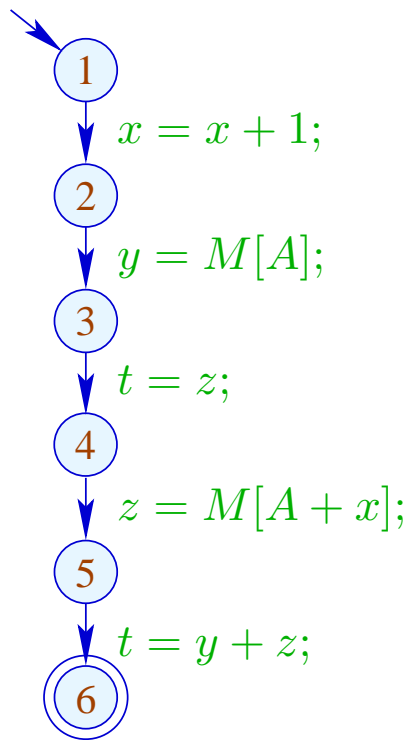
Possible Dependencies:

Definition	→	Use	//	Reaching Definitions
Use	→	Definition	//	???
Definition	→	Definition	//	Reaching Definitions

Reaching Definitions:

Determine for each u which definitions may reach u \implies 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	$\{\langle x, 2 \rangle, \langle y, 3 \rangle, \langle z, 5 \rangle, \langle t, 6 \rangle\}$

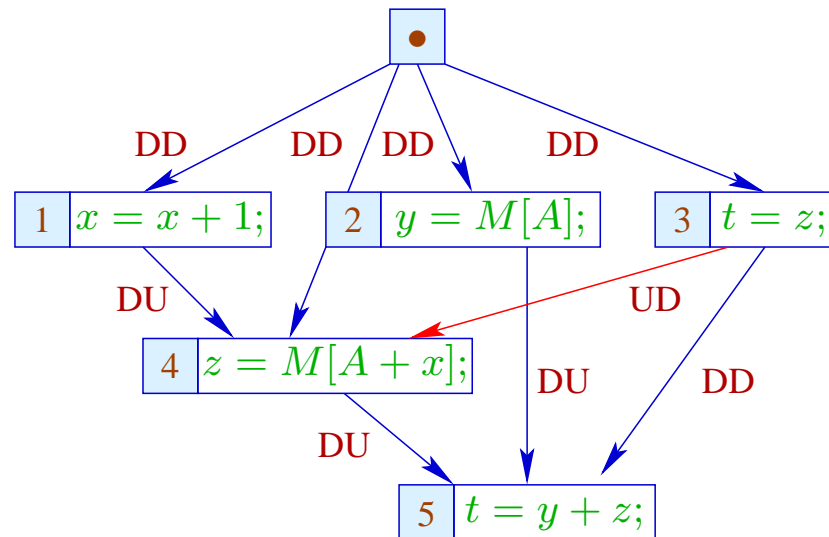
Let U_i, D_i denote the sets of variables which are used or defined at the edge outgoing from u_i . Then:

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

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

... in the Example:

		<i>Def</i>	<i>Use</i>
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 resources, in particular, its) execution time.

Our goal is a maximally parallel **correct** sequence of words.

For that, we maintain the current system state:

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

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

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 **VLIW**s 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

:-)

Note:

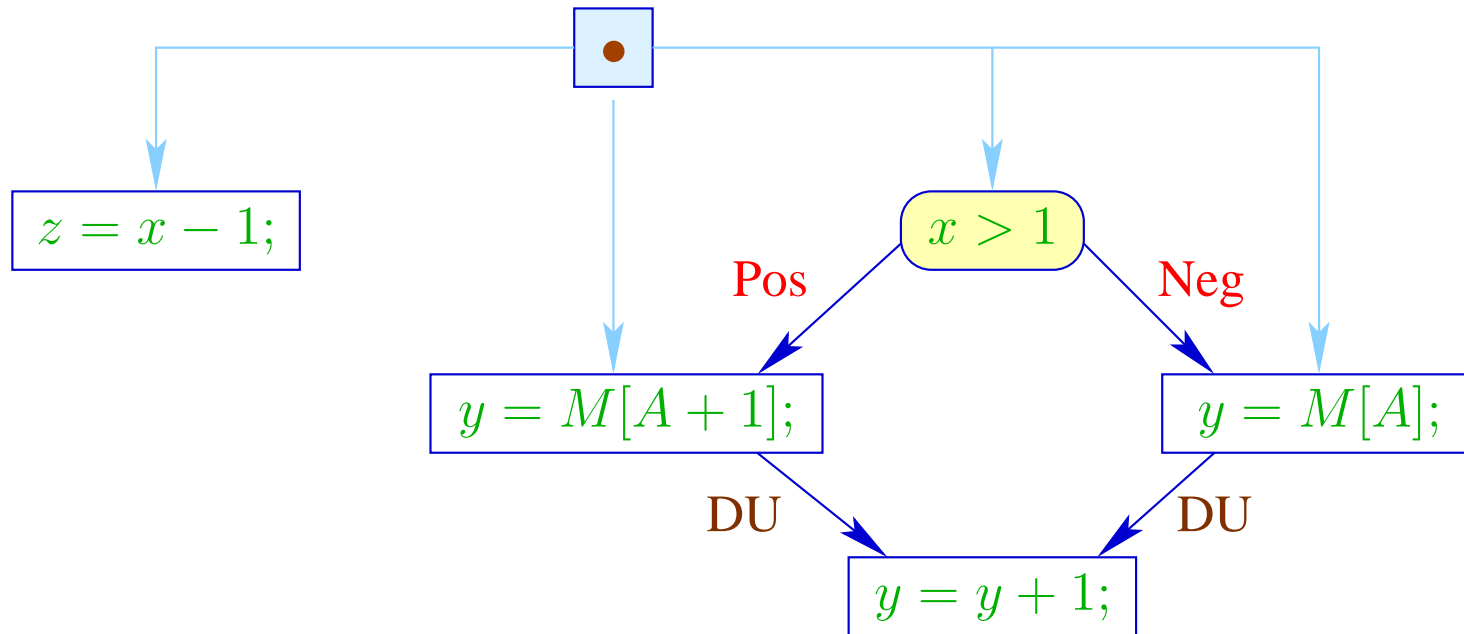
- If instructions put constraints on future selection, we also record these in Σ .
- Overall, we still distinguish just **finitely many** system states :-)
- The computation of the effect of a **VLIW** onto Σ can be compiled into a **finite automaton !!!**
- This automaton, though, could be quite huge :-((
- The challenge of making choices still remains :-((
- Basic blocks usually are not very large
 \implies opportunities for parallelization are limited :-(((

Extension 1: Acyclic Code

```
if ( $x > 1$ ) {  
     $y = M[A]$ ;  
     $z = x - 1$ ;  
} else {  
     $y = M[A + 1]$ ;  
     $z = x - 1$ ;  
}  
 $y = y + 1$ ;
```

The dependence graph must be enriched with extra control-dependencies

...



The statement $z = x - 1;$ is executed with the same arguments in both branches and does not modify any of the remaining variables :-)

We could have moved it **before** the if anyway :-))

The following code could be generated:

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

At every jump target, we guarantee the invariant $:-(\$

If we allow several (known) states on entry of a sub-block, we can generate code which complies with all of these.

... in the Example:

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

If this parallelism is not yet sufficient, we could try to speculatively execute possibly useful tasks ...

For that, we require:

- an idea which alternative is executed more frequently;
- the wrong execution may not end in a **catastrophy**, i.e., run-time errors such as, e.g., division by 0;
- the wrong execution must allow roll-back (e.g., by delaying a **commit**) or may not have any observational effects ...

... in the Example:

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

In the case $x \leq 0$ we have $y = M[A]$ executed in advance.

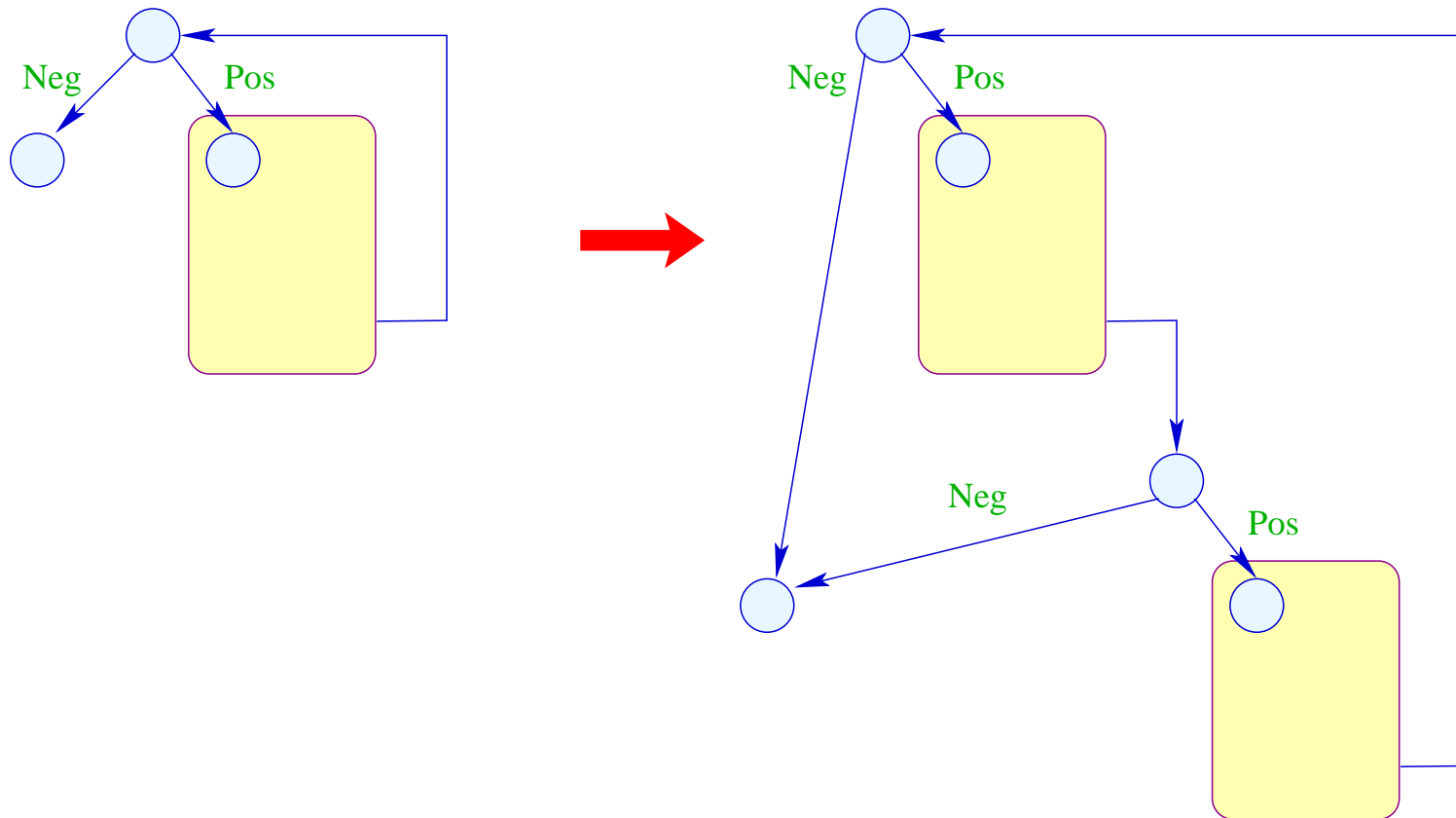
This value, however, is overwritten in the next step :-)

In general:

$x = e;$ has no observable effect in a branch if x is **dead** in this branch :-)

Extension 2: Unrolling of Loops

We may unrole **important**, i.e., inner loops several times:



Now it is clear which side of tests to prefer:

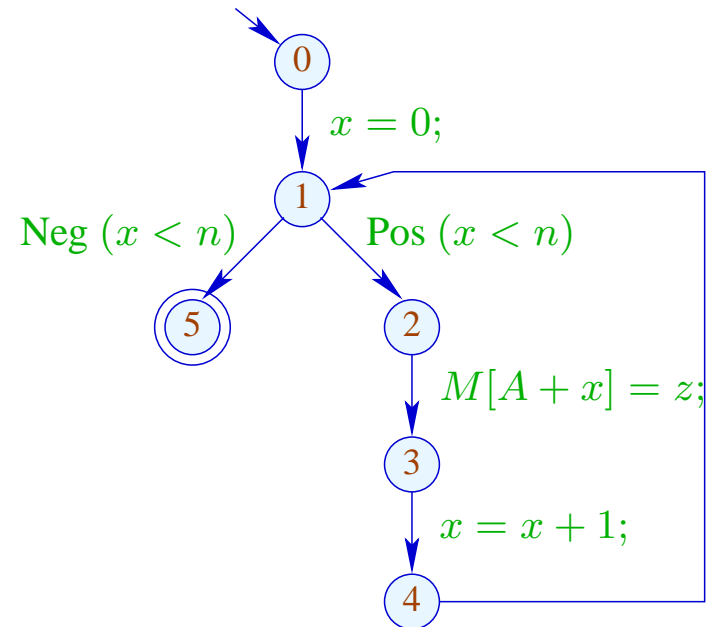
the side which stays within the unrolled body of the loop :-)

Warning:

- The different instances of the body are translated relative to possibly different initial states :-)
- The code behind the loop must be correct relative to the exit state corresponding to every jump out of the loop!

Example:

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

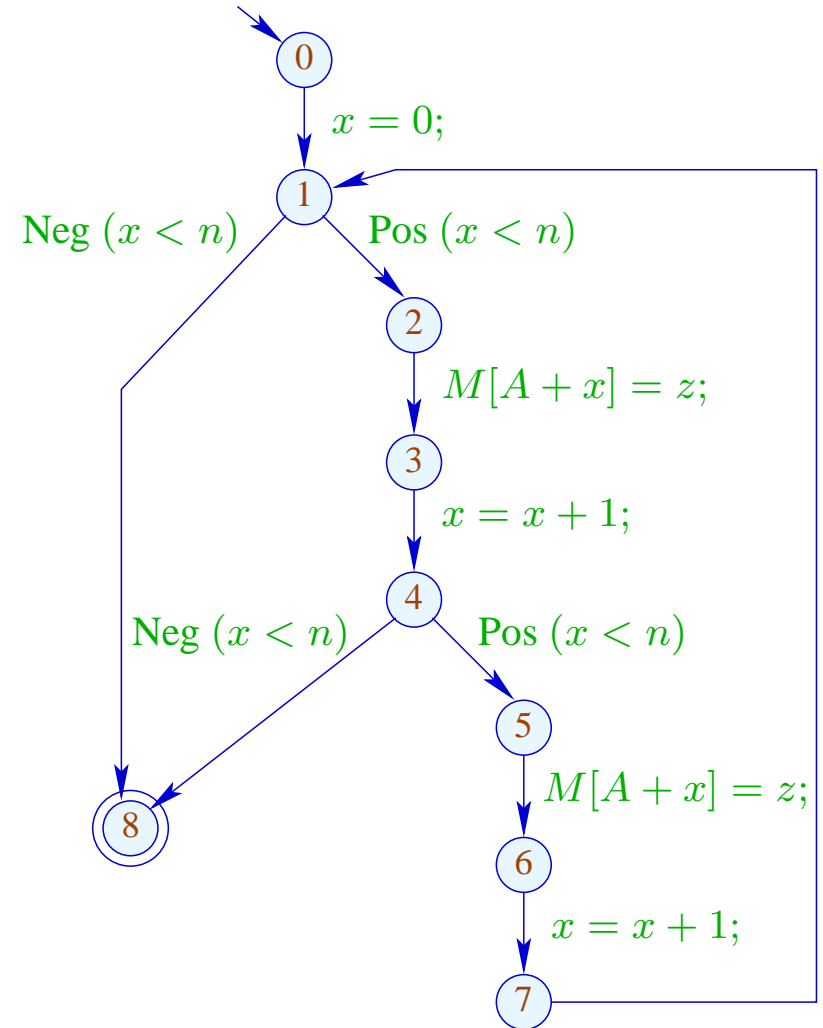


Duplication of the body yields:

```

for ( $x = 0; x < n; x++$ ) {
     $M[A + x] = z;$ 
     $x = x + 1;$ 
    if ( $!(x < n)$ ) break;
     $M[A + x] = z;$ 
}

```



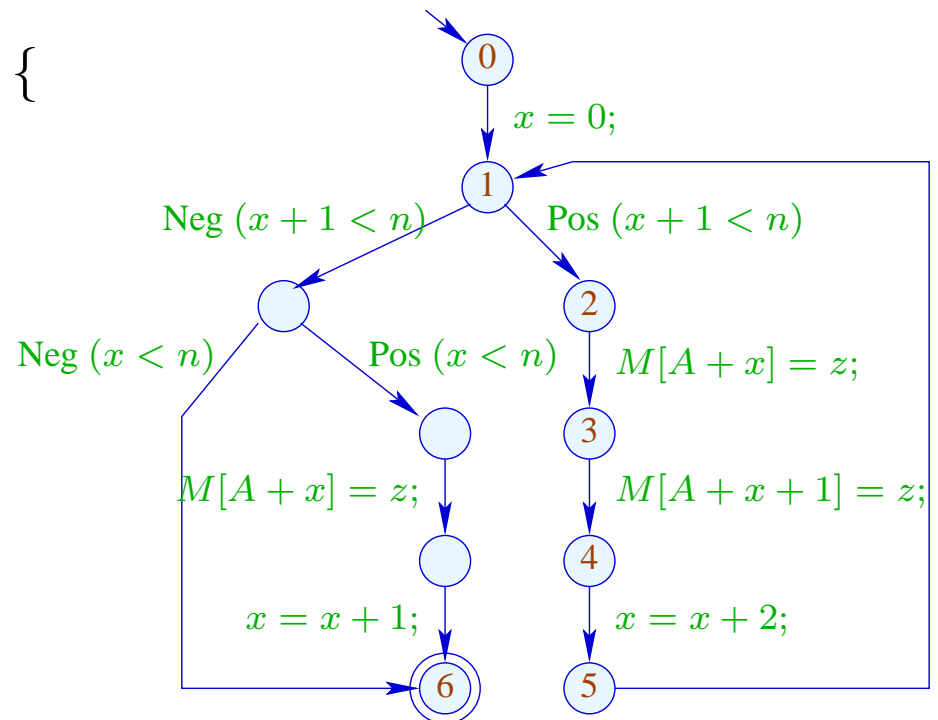
It would be better if we could remove the assignment $x = x + 1$; together with the test in the middle — since these serialize the execution of the copies !!

This is possible if we substitute $x + 1$ for x in the second copy, transform the condition and add a compensation code:

```

for ( $x = 0$ ;  $x + 1 < n$ ;  $x = x + 2$ ) {
     $M[A + x] = z$ ;
     $M[A + x + 1] = z$ ;
}
if ( $x < n$ ) {
     $M[A + x] = z$ ;
     $x = x + 1$ ;
}

```



Discussion:

- Elimination of the intermediate test together with the fusion of all increments at the end reveals that the different loop iterations are in fact independent :-)
- Nonetheless, we do not gain much since we only allow one store per word :-)
- If right-hand sides, however, are more complex, we can interleave their evaluation with the stores :-)

Extension 3:

Sometimes, one loop alone does not provide enough opportunities for parallelization :-)

... but perhaps two successively in a row :-)

Example:

```
for ( $x = 0; x < n; x++$ ) {  
     $R = B[x];$   
     $S = C[x];$   
     $T_1 = R + S;$   
     $A[x] = T_1;$   
}
```

```
for ( $x = 0; x < n; x++$ ) {  
     $R = B[x];$   
     $S = C[x];$   
     $T_2 = R - S;$   
     $C[x] = T_2;$   
}
```

In order to fuse two loops into one, we require that:

- the iteration schemes coincide;
- the two loops access different data.

In case of individual variables, this can easily be verified.

This is more difficult in presence of arrays.

Taking the source program into account, accesses to distinct statically allocated arrays can be identified.

An analysis of accesses to the same array is significantly more difficult ...

Assume that the blocks A, B, C are distinct.

Then we can combine the two loops into:

```
for ( $x = 0; x < n; x++$ ) {  
     $R = B[x];$             $R = B[x];$   
     $S = C[x];$             $S = C[x];$   
     $T_1 = R + S;$         $T_2 = R - S;$   
     $A[x] = T_1;$         $C[x] = T_2;$   
}
```

The first loop may in iteration x not read data which the second loop writes to in iterations $< x$.

The second loop may in iteration x not read data which the first loop writes to in iterations $> x$.

If the index expressions of jointly accessed arrays are **linear**, the given constraints can be verified through **integer linear programming ...**

$$\begin{array}{ll} i \geq 0 & x_{\text{write}} = i \\ i \leq x - 1 & x_{\text{read}} = x \\ & x_{\text{read}} = x_{\text{write}} \end{array}$$

// x_{read} read access to C by 1st loop

// x_{write} write access to C by 2nd loop

... obviously has no solution :-)

General Form:

$$s \geq t_1$$

$$t_2 \geq s$$

$$y_1 = s_1$$

$$y_2 = s_2$$

$$y_1 = y_2$$

for linear expressions s, t_1, t_2, s_1, s_2 over i and the iteration variables.

This can be simplified to:

$$0 \leq s - t_1 \quad 0 \leq t_2 - s \quad 0 = s_1 - s_2$$

What should we do with it ???

Simple Case:

The two inequations have no solution over \mathbb{Q} .

Then they also have no solution over \mathbb{Z} :-)

... in Our Example:

$$x = i$$

$$0 \leq i = x$$

$$0 \leq x - 1 - i = -1$$

The second inequation has no solution :-)

One Variable:

The inequations where x occurs positive, provide **lower bounds**.

The inequations where x occurs negative, provide **upper bounds**.

If G, L are the greatest lower and the least upper bound, respectively, then all (integer) solution are in the interval $[G, L]$:-)

Example:

$$\begin{array}{l} 0 \leq 13 - 7 \cdot x \\ 0 \leq -1 + 5 \cdot x \end{array} \iff \begin{array}{l} x \leq \frac{13}{7} \\ x \geq \frac{1}{5} \end{array}$$

The only **integer** solution of the system is $x = 1$:-)

Discussion:

- Solutions only matter within the bounds to the iteration variables.
- Every **integer** solution there provides a conflict.
- Fusion of loops is possible if **no** conflicts occur :-)
- The given special case suffices to solve the case one variable over \mathbb{Z} :-)
- The number of variables in the inequations corresponds to the nesting-depth of for-loops \implies in general, is quite **small** :-)