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 \( R_5 \):

\[
\psi = \begin{align*}
R_1 &= R_2; \\
R_2 &= R_4; \\
R_3 &\leftrightarrow R_5;
\end{align*}
\]

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 \(\Rightarrow\) reduction of life ranges \(^{(+)}\)
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

\[ \psi = R_1 = R_2 \mid R_2 = R_4 \mid R_5 = R_6 \mid R_6 = R_3 \]

The parallel assignment realizes the linear register moves for \( R_1, R_2 \) and \( R_4 \) together with the cyclic shift for \( R_5 \) and \( R_6 \):

\[ \psi = \begin{cases} R_1 = R_2; \\ R_2 = R_4; \\ R_5 \leftrightarrow R_6; \end{cases} \]

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 \( \rightarrow \)
  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 \( \rightarrow \)
  - restore overwritten registers only.
- Alternatively, we save only registers which are still live after the call — and then possibly into different registers \( \rightarrow \) reduction of life ranges \( \rightarrow \)
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:
$$w = (R) = R_2 + R_6 \quad D = D_1 \times D_2 \quad R_6 = M[R_4]$$

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

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

- During Execute and Write different internal registers/busses/alu 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 $\implies$)

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.
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:

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point Operation</td>
<td>3</td>
</tr>
<tr>
<td>Load/Store</td>
<td>2</td>
</tr>
<tr>
<td>Integer Arithmetic</td>
<td>1</td>
</tr>
</tbody>
</table>

Timing Diagram:

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>-1</td>
<td>2</td>
<td>0.3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>17.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Example Timing:

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point Operation</td>
<td>3</td>
</tr>
<tr>
<td>Load/Store</td>
<td>2</td>
</tr>
<tr>
<td>Integer Arithmetic</td>
<td>1</td>
</tr>
</tbody>
</table>

Timing Diagram:

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>-1</td>
<td>2</td>
<td>0.3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>17.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$R_3$ is over-written, after the addition has fetched 2.
VLIW:
One instruction simultaneously executes up to \( k \) (e.g., 4) elementary Instructions.

Pipelining:
Instruction execution may overlap.

Example:
\[
\begin{align*}
we &= (R_1 = R_2 + R_3 \mid D = D_1 \times D_2 \mid R_3 = M[R_4])
\end{align*}
\]

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

Conflicts:

Read-Read: A register is simultaneously read.
\[\implies \text{in general, unproblematic} \implies \]

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

Conflict Resolution:

\[
\begin{itemize}
  \item \ldots \text{ruled out!}
  \item \text{Read is delayed (stalls), until write has terminated!}
  \item \text{Read before write returns old value!}
\end{itemize}
\]

Example Timing:

<table>
<thead>
<tr>
<th>Floating-point Operation</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store</td>
<td>2</td>
</tr>
<tr>
<td>Integer Arithmetic</td>
<td>1</td>
</tr>
</tbody>
</table>

Timing Diagram:

\[
\begin{array}{cccc}
R_1 & R_2 & R_3 & D \\
0 & 5 & -1 & 2.0 \\
1 & 1 & 0.0 & 0.3 \\
2 & 0.0 & 17.4 & 0.0 \\
3 & 0.0 & 0.0 & 0.0 \\
\end{array}
\]

\( R_3 \) is over-written, after the addition has fetched 2 \( \implies \) 

Write-Write: A register is simultaneously written to.
\[\implies \text{in general, unproblematic} \implies \]

Conflict Resolutions:

\[\begin{itemize}
  \item \ldots \text{ruled out!}
  \item \ldots
\end{itemize}\]

In Our Examples ...

\[\begin{itemize}
  \item \text{simultaneous read is permitted;}
  \item \text{simultaneous write/read and write/write is ruled out;}
  \item \text{no stalls are injected.}
\end{itemize}\]

We first consider basic blocks only, i.e., linear sequences of assignments ...
Idea: Data Dependence Graph

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:

1. $x = x + 1$
2. $y = M[A];$
3. $z = z;$
4. $z = M[A + x];$
5. $t = y + z;$

Possible Dependencies:

Definition → Use // Reaching Definitions

Use → Definition // Reaching Definitions

Definition → Definition // Reaching Definitions

Reaching Definitions:

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

... in the Example:

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

$(u_1, u_2) \in DD$ if $u_1 \in R[u_2] \land D_1 \cap D_2 \neq \emptyset$

$(u_1, u_2) \in DU$ if $u_1 \in R[u_2] \land D_1 \cap U_2 \neq \emptyset$

... in the Example:

<table>
<thead>
<tr>
<th></th>
<th>Def</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x = x + 1;$</td>
<td>${x}$</td>
</tr>
<tr>
<td>2</td>
<td>$y = M[A];$</td>
<td>${y}$</td>
</tr>
<tr>
<td>3</td>
<td>$z = z;$</td>
<td>${z}$</td>
</tr>
<tr>
<td>4</td>
<td>$z = M[A + x];$</td>
<td>${z}$</td>
</tr>
<tr>
<td>5</td>
<td>$t = y + z;$</td>
<td>${t}$</td>
</tr>
</tbody>
</table>

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:

$\Sigma : Vars \rightarrow N$

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

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 $\Sigma$.

**Warning:**

$\rightarrow$ The execution of two VLIWs can overlap !!!
$\rightarrow$ Determining an optimal sequence, is NP-hard ...

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 $\Sigma$.

**Warning:**

$\rightarrow$ The execution of two VLIWs can overlap !!!
$\rightarrow$ Determining an optimal sequence, is NP-hard ...