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 cordal
  -:
- 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:

\[ \psi_1 = R_1 = R_2 | R_2 = R_1 \]

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:

\[
\begin{align*}
R & = R_1; \\
R_1 & = R_2; \\
R_2 & = R_i;
\end{align*}
\]
(2) XOR:

\[
R_1 = R_1 \oplus R_2; \\
R_2 = R_1 \oplus R_2; \\
R_3 = R_1 \oplus R_2;
\]

But what about cyclic shifts such as:

\[
\psi_k = R_1 = R_2 | \ldots | R_{k-1} = R_k | R_k = R_1
\]

for \( k > 2 \) ??

But what about cyclic shifts such as:

\[
\psi_k = R_1 = R_2 | \ldots | R_{k-1} = R_k | R_k = R_1
\]

for \( k > 2 \) ??

Then at most \( k - 1 \) swaps of two registers are needed:

\[
\psi_k = R_1 \leftrightarrow R_2; \\
R_3 \leftrightarrow R_4; \\
\ldots \\
R_{k-1} \leftrightarrow R_k;
\]
(2) XOR:
\[
\begin{align*}
R_1 &= R_1 \oplus R_2; \\
R_2 &= R_1 \oplus R_2; \\
R_1 &= R_1 \oplus R_2;
\end{align*}
\]

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:
\[
\begin{align*}
\psi_k &= R_1 \leftrightarrow R_2; \\
R_2 &= R_3; \\
\ldots \\
R_{k-1} &= R_k;
\end{align*}
\]

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
\[
\psi = R_1 = R_2 \mid R_2 = R_5 \mid R_3 = R_4 \mid R_4 = R_5 \mid R_5 = R_1
\]
consists of the cycles \((R_1, R_2, R_5)\) and \((R_3, R_4)\). Therefore:
\[
\begin{align*}
\psi &= R_1 \leftrightarrow R_2; \\
R_2 &= R_5; \\
R_3 &= R_4;
\end{align*}
\]

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 \( R_1, R_3 \) and \( R_4 \) together with the cyclic shift for \( R_2 \) and \( R_5 \):
\[
\begin{align*}
\psi &= R_1 = R_2; \\
R_2 &= R_4; \\
R_3 &= R_5;
\end{align*}
\]
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 \( R_2 \) and \( R_5 \):

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

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 \( \implies \) reduction of life ranges :)

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 \( \implies \) reduction of life ranges :)

\( \text{PBQP} \)
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:

$$ w = (R_1 = R_2 + R_3) \quad D = D_1 \cdot D_2 \quad R_5 = M[R_4] $$
Warning:
- Instructions occupy hardware resources.
- Instructions may access the same busses/registers → hazards
- Results of an instruction may be available only after some delay.
- During execution, different parts of the hardware are involved:

  Fetch → Decode → Execute → Write

- During Execute and Write different internal registers/busses/alu's may be used.

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

Pipelining:
Instruction execution may overlap.

Example:
$$w = (R_1 = R_2 + R_3) \quad D = D_1 + D_2 \quad R_5 = M(R_4)$$

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

  Fetch → Decode → Execute → Write

- During Execute and Write different internal registers/busses/alu's 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 and 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.

-------

Warning:

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

\[
\begin{array}{c}
\text{Fetch} \\
\downarrow \\
\text{Decode} \\
\downarrow \\
\text{Execute} \\
\downarrow \\
\text{Write}
\end{array}
\]

- During Execute and Write different internal registers/busses/alu's may be used.

\[
\begin{array}{c}
2 \\
3 \\
4
\end{array}
\]

-------

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:

\[
\begin{array}{cccc}
R_1 & R_2 & R_3 & D \\
0 & 5 & -1 & 2 & 0.3 \\
1 & 1 &  &  &  \\
2 &  & 49 &  & 17.4 \\
3 &  &  &  & \\
\end{array}
\]

\(R_3\) 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:

\[
\begin{align*}
&\text{Example:} \\
&\text{Example Timing:} \\
&\text{Timing Diagram:}
\end{align*}
\]

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

\[
R_3 \text{ is over-written, after the addition has fetched } 2 \implies
\]

\[
\begin{array}{cccc}
\text{Floating-point Operation} & 3 \\
\text{Load/Store} & 2 \\
\text{Integer Arithmetic} & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
R_1 & R_2 & R_3 & D \\
0 & 5 & -1 & 22 & 0.3 \\
1 & 1 & 49 & 17.4 \\
\end{array}
\]

\[
\begin{array}{cccc}
& & & \\
0 & 5 & -1 & 22 & 0.3 \\
1 & 1 & 49 & 17.4 \\
\end{array}
\]

\[
R_3 \text{ is over-written, after the addition has fetched } 2 \implies
\]

\[
\begin{array}{cccc}
\text{Floating-point Operation} & 3 \\
\text{Load/Store} & 2 \\
\text{Integer Arithmetic} & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
R_1 & R_2 & R_3 & D \\
0 & 5 & -1 & 22 & 0.3 \\
1 & 1 & 49 & 17.4 \\
\end{array}
\]

\[
\begin{array}{cccc}
& & & \\
0 & 5 & -1 & 22 & 0.3 \\
1 & 1 & 49 & 17.4 \\
\end{array}
\]

\[
R_3 \text{ is over-written, after the addition has fetched } 2 \implies
\]
If a register is accessed simultaneously (here: $R_0$), a strategy of conflict solving is required ...

Conflicts:

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

$\implies$ 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.

$\implies$ 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

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

**Example:**

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

---

**Idea:** Data Dependence Graph

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

**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 \( u \) 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:

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

... 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>( t = z; )</td>
<td>{t}</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>
Let $U_i, D_i$ denote the sets of variables which are used or defined at
the edge outgoing from $u_i$. Then:

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

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

... in the Example:

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

[Diagram of variables and edges]