“On the Fly” Local Register Allocation

Allocate registers as needed during code generation.
Partition registers into 3 classes.

- Allocatable
  Explicitly allocated and freed; used to hold a variable, literal or temporary.
  On SPARC: Local registers & unused In registers.

- Reserved
  Reserved for specific purposes by OS or software conventions.
  On SPARC: %fp, %sp, return address register, argument registers, return value register.
• Work

Volatile—used in short code sequences that need to use a register.
On SPARC: \( \%g1 \) to \( \%g4 \), unused out registers.

Register Targeting

Allow “end user” of a value to state a register preference in AST or IR.

or

Use Peephole Optimization to eliminate unnecessary register moves.

or

Use \textit{preferencing} in a graph coloring register allocator.
Register Tracking

Improve upon standard getReg/freeReg allocator by *tracking* (remembering) register contents.

Remember the value(s) currently held within a register; store information in a Register Association List.

Mark each value as *Saved* (in memory) or *Unsaved* (in memory).

Each value in a register has a *Cost*. This is the cost (in instructions) to restore the value to a register.
The cost of allocating a register is the sum of the costs of the values it holds.

\[ \text{Cost}(\text{register}) = \sum_{\text{values} \in \text{register}} \text{cost}(\text{values}) \]

When we allocate a register, we will choose the cheapest one.

If 2 registers have the same cost, we choose that register whose values have the most distant next use.

(Why most distant?)
Costs for the SPARC

0  Dead Value
1  Saved Local Variable
1  Small Literal Value (13 bits)
2  Saved Global Variable
2  Large Literal Value (32 bits)
2  Unsaved Local Variable
4  Unsaved Global Variable
Register Tracking Allocator

```c
reg getReg() {
    if ( ∃ r ∈ regSet and cost(r) == 0)
        choose(r)
    else {
        c = 1;
        while(true) {
            if ( ∃ r ∈ regSet and cost(r) == c) {
                choose r with cost(r) == c and
                    most distant next use of
                    associated values;
                break;
            }
            c++;
        }
    }
    Save contents of r as necessary;
    return r;
}
```
• Once a value becomes dead, it may be purged from the register association list without any saves.

• Values no longer used, but unsaved, can be purged (and saved) at \textit{zero} cost.

• Assignments of a register to a simple variable may be \textit{delayed}—just add the variable to the Register’s Association List entry as unsaved.

The assignment may be done later or made \textit{unnecessary} (by a later assignment to the variable)

• At the end of a basic block all unsaved values are stored into memory.
Example

```c
int a,b,c,d; // Globals
a = 5;
b = a + d;
c = b - 7;
b = 10;
```

Naive Code

```
mov   5,%10
st    %10,[a]
ld    [a],%10
ld    [d],%11
add   %10,%11,%11
st    %11,[b]
ld    [b],%11
sub   %11,7,%11
st    %11,[c]
mov   10,%11
st    %11,[b]
```

18 instructions are needed (memory references take 2 instructions)
## With Register Tracking

<table>
<thead>
<tr>
<th>Instruction Generated</th>
<th>%10</th>
<th>%11</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov 5,%10</td>
<td>5(S)</td>
<td></td>
</tr>
<tr>
<td>! Defer assignment to a d</td>
<td>5(S), a(U)</td>
<td></td>
</tr>
<tr>
<td>ld [d], %11</td>
<td>5(S), a(U)</td>
<td>d(S)</td>
</tr>
<tr>
<td>!d unused after next inst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add %10,%11,%11</td>
<td>5(S), a(U)</td>
<td>b(U)</td>
</tr>
<tr>
<td>!b is dead after next inst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub %11,7,%11</td>
<td>5(S), a(U)</td>
<td>c(U)</td>
</tr>
<tr>
<td>! %11 has lower cost</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st %11, [c]</td>
<td>5(S), a(U)</td>
<td></td>
</tr>
<tr>
<td>mov 10,%11</td>
<td>5(S), a(U)</td>
<td>b(U), 10(S)</td>
</tr>
<tr>
<td>! save unsaved values</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st %10, [a]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st %11,[b]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12 instructions (rather than 18)
Pointers, Arrays and Reference Parameters

When an array, reference parameter or pointed-to variable is read, all unsaved register values that might be aliased must be stored.

When an array, reference parameter or pointed-to variable is written, all unsaved register values that might be aliased must be stored, then cleared from the register association list.

Thus if \( a[3] \) is in a register and \( a[i] \) is assigned to, \( a[3] \) must be stored (if unsaved) and removed from the association list.
Optimal Expression Tree
Translation—Sethi-Ullman Algorithm


Goal: Translate an expression tree using the fewest possible registers.

Approach: Mark each tree node, N, with an Estimate of the minimum number of registers needed to translate the tree rooted by N.

Let $RN(N)$ denote the Register Needs of node N.
In a Load/Store architecture (ignoring immediate operands):
RN(leaf) = 1

RN(Op) =
    If RN(Left) = RN(Right)
        Then RN(Left) + 1
    Else Max(RN(Left), RN(Right))

Example:
Key Insight of SU Algorithm

Translate subtree that needs more registers \textit{first}.

Why?

After translating one subtree, we’ll need a register to hold its value.

If we translate the more complex subtree first, we’ll still have enough registers to translate the less complex expression (without \textit{spilling} register values into memory).
Specification of SU Algorithm

TreeCG(tree *T, regList RL);

Operation:

- Translate expression tree T using only registers in RL.
- RL must contain at least 2 registers.
- Result of T will be computed into head(RL).
Summary of SU Algorithm

if T is a node (variable or literal)
   load T into R1 = head(RL)
else (T is a binary operator)
   Let R1 = head(RL)
   Let R2 = second(RL)
   if RN(T.left) >= Size(RL) and
      RN(T.right) >= Size(RL)
      (A spill is unavoidable)
      TreeCG(T.left, RL)
      Store R1 into a memory temp
      TreeCG(T.right, RL)
      Load memory temp into R2
      Generate (OP R2,R1,R1)
   elsif RN(T.left) >= RN(T.right)
      TreeCG(T.left, RL)
      TreeCG(T.right, tail(RL))
      Generate (OP R1,R2,R1)
   else
      TreeCG(T.right, RL)
      TreeCG(T.left, tail(RL))
      Generate (OP R2,R1,R1)
Example (with Spilling)

Assume only 2 Registers;
RL = [\%10, \%11]

We Translate the left subtree first (using 2 registers), store its result into memory, translate the right subtree, reload the left subtree’s value, then do the final operation.
ld  [A], %l0
ld  [B], %l1
sub %l0,%l1,%l0
st  %l0, [temp]
ld  [C], %l0
ld  [D], %l1
add %l0,%l1,%l0
ld  [temp], %l1
add %l1,%l0,%l0
Larger Example

Assume 3 Registers;
RL = [%10,%11,%12]

Since right subtree is more complex, it is translated first.
ld  [C], %10
ld  [D], %11
add %10,%11,%10
ld  [E], %11
ld  [F], %12
mul %11,%12,%11
add %10,%11,%10
ld  [A], %11
ld  [B], %12
sub %11,%12,%11
add %11,%10,%10
Refinements & Improvements

- Register needs rules can be modified to model various architectural features.

For example, Immediate operands, that need not be loaded into registers, can be modeled by the following rule:

\[ \text{RN(literal)} = 0 \text{ if literal may be used as an immediate operand} \]

- Commutativity & Associativity of operands may be exploited:

\[ \begin{array}{c}
+3 \\
+2 \\
A^1 
\end{array} \quad \Rightarrow \quad \begin{array}{c}
+2 \\
+2 \\
A^1 \\
B^1 \\
C^1 \\
D^1 
\end{array} \]
Is Minimizing Register Use Always Wise?

SU minimizes the number of registers used but at the cost of reduced ILP.

Since only 2 registers are used, there is little possibility of parallel evaluation.
When more registers are used, there is often more potential for parallel evaluation:

Here as many as four registers may be used to increase parallelism.
Optimal Translation for DAGs is Much Harder

If variables or expression values may be shared and reused, optimal code generation becomes NP-Complete.

Example: \( a + b \cdot (c + d) + a \cdot (c + d) \)

We must decide how long to hold each value in a register. Best orderings may “skip” between subexpressions.

Scheduling Expression Trees


The Sethi-Ullman Algorithm minimizes register usage, without regard to code scheduling.

On machines with Delayed Loads, we also want to avoid stalls.
What is a Delayed Load?

Most pipelined processors require a delay of one or more instructions between a load of register R and the first use of R.

If a register is used “too soon,” the processor may stall execution until the register value becomes available.

```
ld   [a],%r1
add  %r1,1,%r1
```

We try to place an instruction that doesn’t use register R immediately after a load of R.
This allows useful work instead of a wasteful stall.

The Sethi-Ullman Algorithm generates code that will stall:

In fact, if we use the fewest possible registers, stalls are *Unavoidable!*
Why?

Loads increase the number of registers in use.

Binary operations decrease the number of registers in use (2 Operands, 1 Result).

The load that brings the number of registers in use up to the minimum number needed must be followed by an operator that uses the just-loaded value. This implies a stall.

We’ll need to allocate an extra register to allow an independent instruction to fill each delay slot of a load.
Extended Register Needs

Abbreviated as $ERN$

ERN(Identifier) = 2
ERN(Literal) = 1
ERN(Op) =

\[
\text{If } ERN(\text{Left}) = ERN(\text{Right}) \\
\text{Then } ERN(\text{Left}) + 1 \\
\text{Else Max}(ERN(\text{Left}), ERN(\text{Right}))
\]
Example

```
<table>
<thead>
<tr>
<th>+3</th>
<th>D^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td></td>
</tr>
<tr>
<td>A^2</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>+3</th>
<th>C^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2</td>
<td></td>
</tr>
<tr>
<td>A^2</td>
<td></td>
</tr>
</tbody>
</table>
```

**Idea of the Algorithm**

1. Generate instructions in the same order as Sethi-Ullman, but use Pseudo-Registers instead of actual machine registers.

2. Put generated instructions into a “Canonical Order” (as defined below).

What are Pseudo-Registers?

They are unique temporary locations, unlimited in number and generated as needed, that are used to model registers prior to register allocation.

Canonical Form for Expression Code

(Assume R registers will be used)

Desired instruction ordering:
1. R load instructions
2. Pairs of Operator/Load instructions
3. Remaining operators
This canonical form is obtained by “sliding” load instructions upward (earlier) in the original code ordering.

Note that:

• Moving loads upward is always safe, since each pseudo-register is assigned to only once.

• No more than R registers are ever live.
Example

Let $R = 3$, the minimum needed for a delay-free schedule.

Put into Canonical Form:

1. $ld [B], PR1$
2. $ld [C], PR2$
3. $add PR1,PR2,PR3$
4. $ld [D], PR4$
5. $add PR3,PR4,PR5$
6. $ld [A], PR6$
7. $add PR6,PR5,PR7$

(Before Register Assignment)

1. $ld [B], %10$
2. $ld [C], %11$
3. $ld [D], %12$
4. $add %10,%11,%10$
5. $ld [A], %11$
6. $add %10,%12,%10$
7. $add %11,%10,%10$

(After Register Assignment)

No Stalls!
Does This Algorithm Always Produce a Stall-Free, Minimum Register Schedule?

Yes—if one exists!

For very simple expressions (one or two operands) no stall-free schedule exists.

For example: \(a = b\);

\[
\begin{align*}
\text{ld} & \quad [b], \%10 \\
\text{st} & \quad \%10, \ [a]
\end{align*}
\]
Why Does the Algorithm Avoid Stalls?

Previously, certain “critical” loads had to appear just before an operation that used their value.

Now, we have an “extra” register. This allows critical loads to move up one or more places, avoiding any stalls.
How Do We Schedule Small Expressions?

Small expressions (one or two operands) are common. We’d like to avoid stalls when scheduling them.

Idea—Blend small expressions together into larger expression trees, using “,” and “;” like binary operators.
Example

\[ a = b + c; \quad d = e; \]

\[ a^0 \]

\[ b^2 \]

\[ c^2 \]

\[ d^0 \]

\[ e^2 \]

Original Code

\[
\begin{align*}
&\text{ld \ [b], PR1} \\
&\text{ld \ [c], PR2} \\
&\text{add \ PR1,PR2,PR3} \\
&\text{st \ PR3, \ [a]} \\
&\text{ld \ [e], PR4} \\
&\text{st \ PR4, \ [d]} \\
\end{align*}
\]

In Canonical Form

\[
\begin{align*}
&\text{ld \ [b], \ %10} \\
&\text{ld \ [c], \ %11} \\
&\text{ld \ [e], \ %12} \\
&\text{add \ %10,%11,%10} \\
&\text{st \ %10, \ [a]} \\
&\text{st \ %12, \ [d]} \\
\end{align*}
\]

After Register Assignment
Global Register Allocation

Allocate registers across an entire subprogram.

A Global Register Allocator must decide:

- What values are to be placed in registers?
- Which registers are to be used?
- For how long is each Register Candidate held in a register?
Live Ranges

Rather than simply allocate a value to a fixed register throughout an entire subprogram, we prefer to split variables into Live Ranges.

What is a Live Range?

It is the span of instructions (or basic blocks) from a definition of a variable to all its uses.

Different assignments to the same variable may reach distinct & disjoint instructions or basic blocks.

If so, the live ranges are Independent, and may be assigned Different registers.
Example

```c
a = init();
for (int i = a+1; i < 1000; i++){
    b[i] = 0; }
a = f(i);
print(a);
```

The two uses of variable \( a \) comprise independent live ranges.
Each can be allocated separately.

If we insisted on allocating variable \( a \) to a fixed register for the whole subprogram, it would conflict with the loop body, greatly reducing its chances of successful allocation.
Granulatity of Live Ranges

Live ranges can be measured in terms of individual instructions or basic blocks.

Individual instructions are more precise but basic blocks are less numerous (reducing the size of sets that need to be computed).

We’ll use basic blocks to keep examples concise.

You can define basic blocks that hold only one instruction, so computation in terms of basic blocks is still fully general.
Computation of Live Ranges

First construct the Control Flow Graph (CFG) of the subprogram.

For a Basic Block $b$ and Variable $V$:

Let $\text{DefsIn}(b) = \text{the set of basic blocks that contain definitions of } V \text{ that reach (may be used in) the beginning of Basic Block } b$.

Let $\text{DefsOut}(b) = \text{the set of basic blocks that contain definitions of } V \text{ that reach (may be used in) the end of Basic Block } b$. 
If a definition of $V$ reaches $b$, then the register that holds the value of that definition must be allocated to $V$ in block $b$.

Otherwise, the register that holds the value of that definition may be used for other purposes in $b$.

The sets Preds and Succ are derived from the structure of the CFG.

They are given as part of the definition of the CFG.
DefsIn and DefsOut must be computed, using the following rules:

1. If Basic Block \( b \) contains a definition of \( V \) then
   \[
   \text{DefsOut}(b) = \{b\}
   \]

2. If there is no definition to \( V \) in \( b \) then
   \[
   \text{DefsOut}(b) = \text{DefsIn}(b)
   \]

3. For the First Basic Block, \( b_0 \):
   \[
   \text{DefsIn}(b_0) = \emptyset
   \]

4. For all Other Basic Blocks
   \[
   \text{DefsIn}(b) = \bigcup_{p \in \text{Preds}(b)} \text{DefsOut}(p)
   \]
Liveness Analysis

Just because a definition reaches a Basic Block, \( b \), \textit{does not mean it} must be allocated to a register at \( b \).

We also require that the definition be \textit{Live} at \( b \). If the definition is dead, then it will no longer be used, and register allocation is unnecessary.

For a Basic Block \( b \) and Variable \( V \):

\[
\text{LiveIn}(b) = \text{true if } V \text{ is Live (will be used before it is redefined) at the beginning of } b.
\]

\[
\text{LiveOut}(b) = \text{true if } V \text{ is Live (will be used before it is redefined) at the end of } b.
\]
LiveIn and LiveOut are computed, using the following rules:

1. If Basic Block b has no successors then
   \[ \text{LiveOut}(b) = \text{false} \]

2. For all Other Basic Blocks
   \[ \text{LiveOut}(b) = \bigvee_{s \in \text{Succ}(b)} \text{LiveIn}(s) \]

3. LiveIn(b) =
   If V is used before it is defined in Basic Block b
   Then true
   Elsif V is defined before it is used in Basic Block b
   Then false
   Else LiveOut(b)
Merging Live Ranges

It is possible that each Basic Block that contains a definition of \( v \) creates a distinct Live Range of \( V \).

\[ \forall \text{ Basic Blocks, } b, \text{ that contain a definition of } V: \]

\[ \text{Range}(b) = \{b\} \cup \{k \mid b \in \text{DefsIn}(k) \& \text{LiveIn}(k)\} \]

This rule states that the Live Range of a definition to \( V \) in Basic Block \( b \) is \( b \) plus all other Basic Blocks that the definition of \( V \) reaches and in which \( V \) is live.
If two Live Ranges overlap (have one of more Basic Blocks in common), they *must* share the same register too. (Why?)

Therefore,

If $\text{Range}(b_1) \cap \text{Range}(b_2) \neq \emptyset$

Then replace

$\text{Range}(b_1)$ and $\text{Range}(b_2)$

with $\text{Range}(b_1) \cup \text{Range}(b_2)$
Example
The Live Ranges we Compute are

Range(1) = \{1\} U \{3,4\} = \{1,3,4\}

Range(2) = \{2\} U \{4\} = \{2,4\}

Range(5) = \{5\} U \{7\} = \{5,7\}

Range(6) = \{6\} U \{7\} = \{6,7\}

Ranges 1 and 2 overlap, so

\text{Range}(1) = \text{Range}(2) = \{1,2,3,4\}

Ranges 5 and 6 overlap, so

\text{Range}(5) = \text{Range}(6) = \{5,6,7\}