**Summary of SU Algorithm**

if $T$ is a node (variable or literal)
  load $T$ into $R1 = \text{head}(RL)$
else (T is a binary operator)
  Let $R1 = \text{head}(RL)$
  Let $R2 = \text{second}(RL)$
  if $\text{RN}(T.\text{left}) \geq \text{Size}(RL)$ and
     $\text{RN}(T.\text{right}) \geq \text{Size}(RL)$
     (A spill is unavoidable)
     $\text{TreeCG}(T.\text{left}, RL)$
     Store $R1$ into a memory temp
     $\text{TreeCG}(T.\text{right}, RL)$
     Load memory temp into $R2$
     Generate (OP $R2,R1,R1$)
  elsif $\text{RN}(T.\text{left}) \geq \text{RN}(T.\text{right})$
     $\text{TreeCG}(T.\text{left}, RL)$
     $\text{TreeCG}(T.\text{right}, \text{tail}(RL))$
     Generate (OP $R1,R2,R1$)
  else
     $\text{TreeCG}(T.\text{right}, RL)$
     $\text{TreeCG}(T.\text{left}, \text{tail}(RL))$
     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], \%10
ld [B], \%11
sub \%10,\%11,\%10
st \%10, [temp]
ld [C], \%10
ld [D], \%11
add \%10,\%11,\%10
ld [temp], \%11
add \%11,\%10,\%10
```

**Larger Example**

Assume 3 Registers;
$RL = [\%10,\%11,\%12]$
Since right subtree is more complex,
it is translated first.

```
ld [A], \%10
ld [B], \%11
sub \%10,\%11,\%10
st \%10, [temp]
ld [C], \%10
ld [D], \%11
add \%10,\%11,\%10
ld [temp], \%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:

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

- Commutativity & Associativity of operands may be exploited:

\[
\begin{align*}
A1 + B1 + C1 + D1 & \Rightarrow \\
A1 + B1 & \quad C1 + D1
\end{align*}
\]

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:

\[
\begin{align*}
A1 + B1 & \quad C1 + D1 & \Rightarrow \\
A1 + B1 & \quad C1 + D1
\end{align*}
\]

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: \( a+b*(c+d)+a*(c+d) \)
We must decide how long to hold each value in a register. Best orderings may "skip" between subexpressions


---

Reading Assignment

- Read Section 15.3 (Register Allocation and Temporary Management) from Chapter 15.
- Read Chaitin's paper, "Register Allocation via Coloring."

---

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.

\[
\text{ld } [a],%r1 \\
\text{add } %r1,1,%r1 \leftarrow \text{Stall!}
\]

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) =
If ERM(Left) = ERM(Right)
    Then ERM(Left) + 1
Else Max(ERN(Left), ERM(Right))

Example

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:

\[ \begin{align*}
& \text{ld} [B], PR1 \\
& \text{ld} [C], PR2 \\
& \text{ld} [D], PR4 \\
& \text{add} PR1, PR2, PR3 \\
& \text{ld} [A], PR6 \\
& \text{add} PR3, PR4, PR5 \\
& \text{add} PR6, PR5, PR7 \\
\end{align*} \]

(Before Register Assignment)

\[ \begin{align*}
& \text{ld} [B], %l0 \\
& \text{ld} [C], %l1 \\
& \text{ld} [D], %l2 \\
& \text{add} %l0, %l1, %l0 \\
& \text{ld} [A], %l1 \\
& \text{add} %l0, %l2, %l0 \\
& \text{add} %l1, %l0, %l0 \\
\end{align*} \]

(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: \( a=b; \)

\[ \begin{align*}
& \text{ld} [b], %l0 \\
& \text{st} %l0, [a] \\
\end{align*} \]

---

**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**

Example code:

```
a = b + c; d = e;
```

```
ld [b], PR1
ld [c], PR2
add PR1, PR2, PR3
st PR3, [a]
```

```
lst [b], %l0
ld [c], %l1
ld [e], %l2
add %l0, %l1, %l2
st %l0, [a]
```

```
ld [b], PR1
ld [c], PR2
add PR1, PR2, PR3
st PR3, [a]
```

```
lst [b], %l0
ld [c], %l1
ld [e], %l2
add %l0, %l1, %l2
st %l0, [a]
```

---

**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**

```plaintext
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.

**Granularity 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`:
- Let `Preds(b)` = the set of basic blocks that are Immediate Predecessors of `b` in the CFG.
- Let `Succ(b)` = the set of basic blocks that are Immediate Successors to `b` in the CFG.

**Control Flow Graphs**

A Control Flow Graph (CFG) models possible execution paths through a program.
Nodes are basic blocks and arcs are potential transfers of control.

For example,

```plaintext
if (a > 0)
    b = 1;
else    b = 2;
a = c + b;
```
For a Basic Block $b$ and Variable $V$: Let $\text{DefsIn}(b) = \{\text{basic blocks that contain definitions of } V \text{ that reach (may be used in) the beginning of Basic Block } b\}$.

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

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 $\text{Preds}$ and $\text{Succ}$ are derived from the structure of the CFG. They are given as part of the definition of the CFG.

$\text{DefsIn}$ and $\text{DefsOut}$ 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, $b$, does not mean it must be allocated to a register at $b$.

We also require that the definition be *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 $V$:

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

$L\text{iveOut}(b) = \text{true if } V \text{ is Live (will be used before it is redefined) at the end of } b$.

$L\text{iveIn}$ and $L\text{iveOut}$ are computed, using the following rules:

1. If Basic Block $b$ has no successors then $L\text{iveOut}(b) = \text{false}$
2. For all Other Basic Blocks $L\text{iveOut}(b) = \bigvee_{s \in \text{Succ}(b)} L\text{iveIn}(s)$
3. $L\text{iveIn}(b) = \text{true if } V \text{ is used before it is defined in Basic Block } b$
   Then $\text{true}$
   Elsif $V$ is defined before it is used in Basic Block $b$
   Then $\text{false}$
   Else $L\text{iveOut}(b)$
**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

\[ \text{Range}(b_1) \text{ and } \text{Range}(b_2) \]

with \( \text{Range}(b_1) \cup \text{Range}(b_2) \)