Reading Assignment

- Read Section 15.3 (Register Allocation and Temporary Management) from Chapter 15
- Get Class Handout 3 from DOIT.
- Read Chaitin's paper, "Register Allocation via Coloring."

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

Reference: R. Sethi, "Complete Register Allocation Problems," *SIAM Journal of Computing*, 1975.

Scheduling Expression Trees

Reference: S. Kurlander, T. Proebsting and C. Fischer, "Efficient Instruction Scheduling for Delayed-Load Architectures," *ACM Transactions on Programming Languages and Systems*, 1995. (Linked from class Web page)

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.

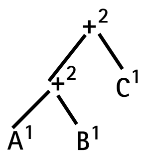
If a register is used "too soon," the processor may stall execution until the register value becomes available.

```
ld   [a],%r1
                  ← Stall!
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:




```
ld  [A], %l0
ld  [B], %l1      <-- Stall!
add %l0,%l1,%l0
ld  [C], %l1      <-- Stall!
add %l0,%l1,%l0
```

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

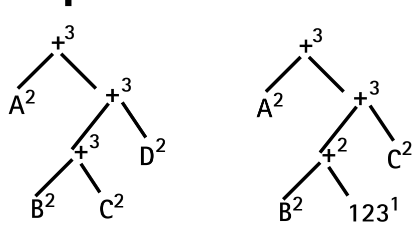
ERN(Op) =

 If ERN(Left) = ERN(Right)

  Then ERN(Left) + 1

  Else Max(ERN(Left), ERN(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).
3. Map pseudo-registers to actual machine registers.

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:

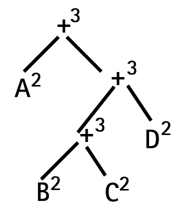
1. R load instructions
2. Pairs of Operator/Load instructions
3. Remaining operators

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




```
ld   [B], PR1
ld   [C], PR2
add PR1,PR2,PR3
ld   [D], PR4
add PR3,PR4,PR5
ld   [A], PR6
add PR6,PR5,PR7
```

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

Put into Canonical Form:

```
ld   [B], PR1
ld   [C], PR2
ld   [D], PR4
add PR1,PR2,PR3
ld   [A], PR6
add PR3,PR4,PR5
add PR6,PR5,PR7
```

(Before Register Assignment)

```
ld   [B], %l0
ld   [C], %l1
ld   [D], %l2
add %l0,%l1,%l0
ld   [A], %l1
add %l0,%l2,%l0
add %l1,%l0,%l0
```

(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;`

```
ld   [b], %l0
st   %l0, [a]
```


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.

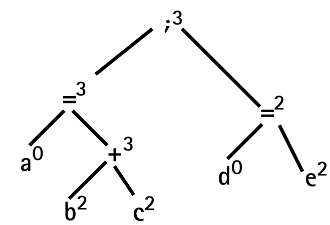
How Do We Schedule Small Expressions?

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; d=e;
```




```
ld  [b], PR1
ld  [c], PR2
add PR1,PR2,PR3
st  PR3, [a]
ld  [e], PR4
st  PR4, [d]
```

Orginal Code

```
ld  [b], PR1
ld  [c], PR2
ld  [e], PR4
add PR1,PR2,PR3
st  PR3, [a]
st  PR4, [d]
```

In Canonical Form

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

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

```
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:

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.

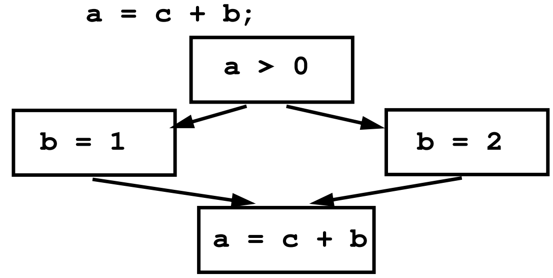
Control Flow Graphs

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,

```
if (a > 0)
       b = 1;
else   b = 2;
a = c + b;
```



For a Basic Block b and Variable V:

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

Let DefsOut(b) = the set of basic blocks that contain definitions of V 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:
   $$\text{DefsIn}(b_0) = \phi$$
4. For all Other Basic Blocks
   $$\text{DefsIn}(b) = \bigcup_{p \in \text{Preds}(b)} \text{DefsOut}(p)$$