Reading Assignment

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)

“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 *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.

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

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

```
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 *zero* cost.
- Assignments of a register to a simple variable may be *delayed*—just add the variable to the Register's Association List entry as unsaved.

  The assignment may be done later or made *unnecessary* (by a later assignment to the variable)
- At the end of a basic block all unsaved values are stored into memory.

Example

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

Naive Code

```
mov   5,%l0
st    %l0,[a]
ld    [a],%l0
ld    [d],%l1
add   %l0,%l1,%l0
st    %l0,[b]
ld    [b],%l0
sub   %l0,7,%l0
st    %l0,[c]
mov   10,%l0
st    %l0,[b]
```

18 instructions are needed (memory references take 2 instructions)

With Register Tracking

Instruction Generated	%l0	%l1
mov 5,%l0	5(S)	
! Defer assignment to a	5(S), a(U)	
ld [d], %l1	5(S), a(U)	d(S)
!d unused after next inst		
add %l0,%l1,%l1	5(S), a(U)	b(U)
!b is dead after next inst		
sub %l1,7,%l1	5(S), a(U)	c(U)
! %l1 has lower cost		
st %l1, [c]	5(S), a(U)	
mov 10, %l1	5(S), a(U)	b(U), 10(S)
! save unsaved values		
st %l0, [a]		b(U), 10(S)
st %l1,[b]		

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

Reference: R. Sethi & J. D. Ullman, "The generation of optimal code for arithmetic expressions," Journal of the ACM, 1970.

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

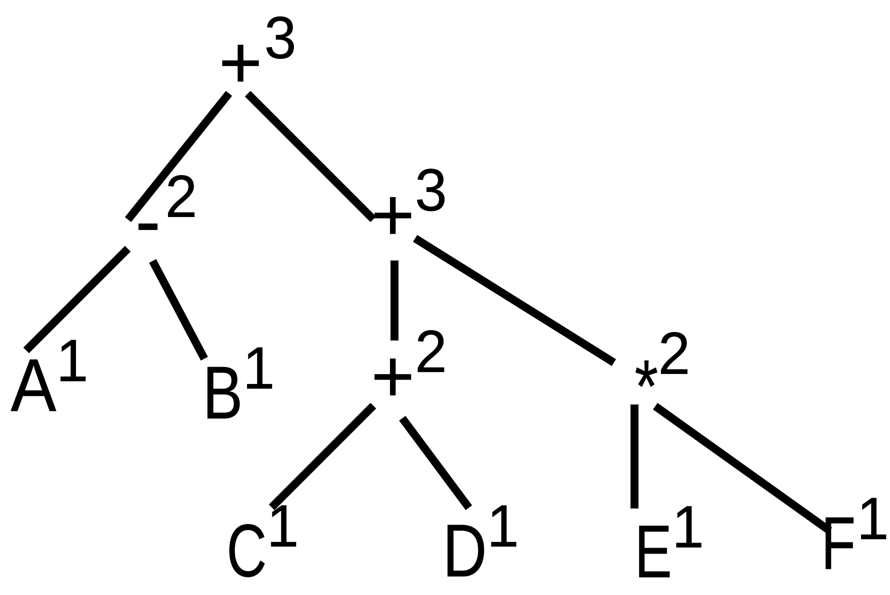
RN(Op) =

    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 *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 *spilling* register values into memory).

Specification of SU Algorithm

TreeCG(tree *T, regList RL);

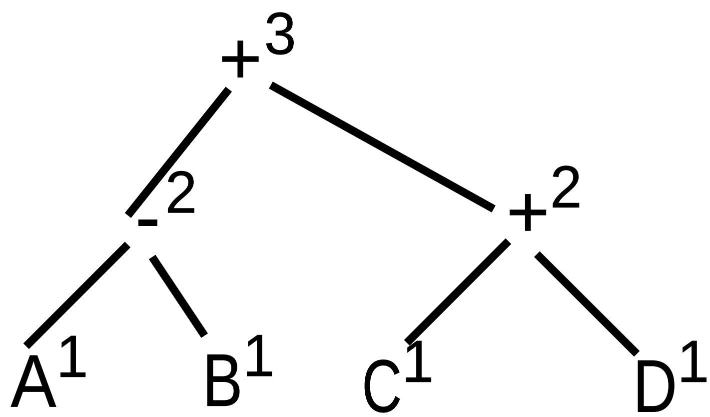
Operation:

- 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

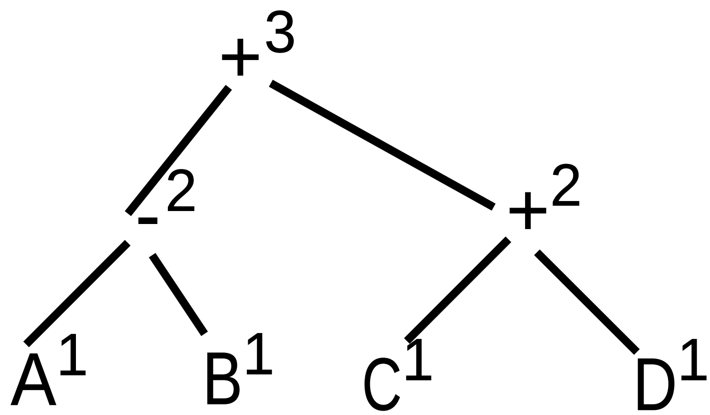
```
if T is a node (variable or literal)
     load T into R1 = head(RL)
else  (T is a binary operator)
   Let R1 = head(RL)
   Let R2 = second(RL)
   if  RN(T.left) >= Size(RL) and
       RN(T.right) >= Size(RL)
       (A spill is unavoidable)
        TreeCG(T.left, RL)
        Store R1 into a memory 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)



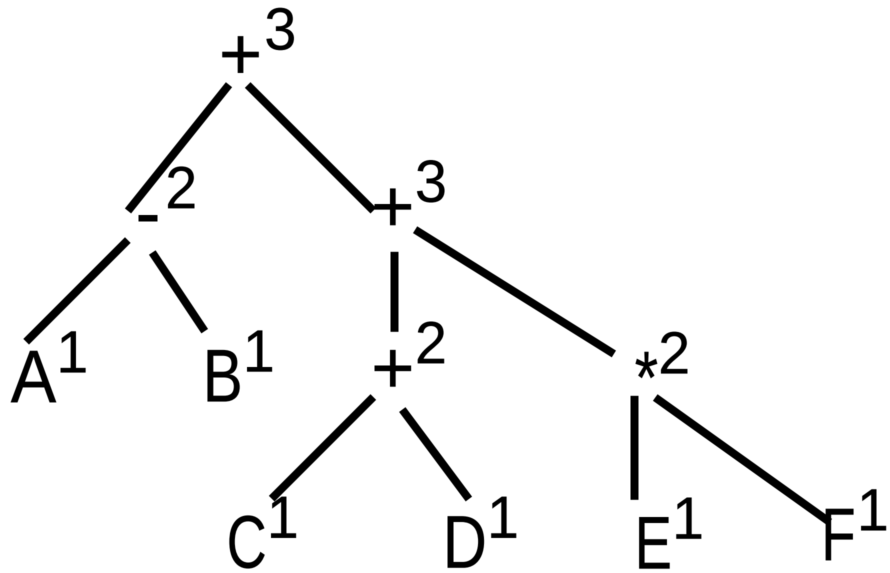
Assume only 2 Registers;
RL = [**%l0**,**%l1**]

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.



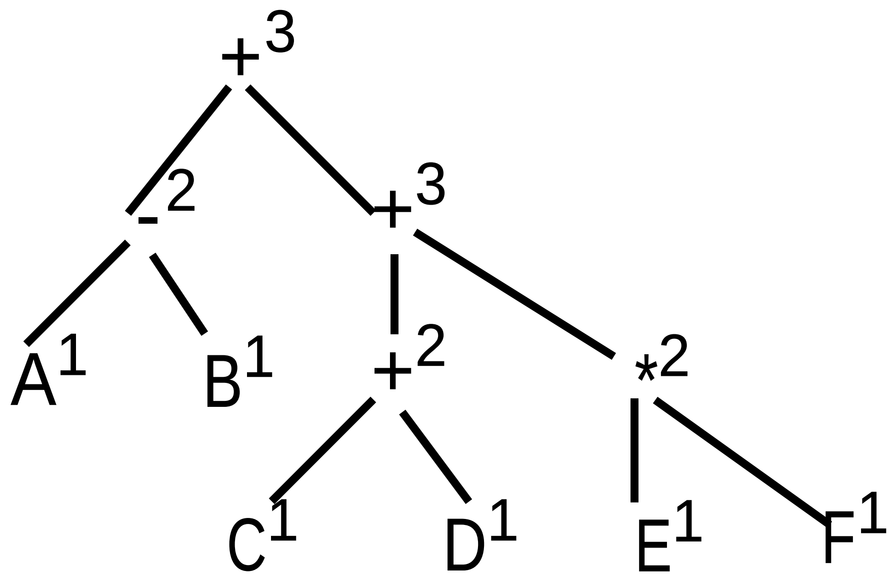

```
ld  [A], %l0
ld  [B], %l1
sub %l0,%l1,%l0
st  %l0, [temp]
ld  [C], %l0
ld  [D], %l1
add %l0,%l1,%l0
ld  [temp], %l1
add %l1,%l0,%l0
```

Larger Example



Assume 3 Registers;
RL = [**%l0**,**%l1**,**%l2**]

Since right subtree is more complex, it is translated first.




```
ld  [C], %l0
ld  [D], %l1
add %l0,%l1,%l0
ld  [E], %l1
ld  [F], %l2
mul %l1,%l2,%l1
add %l0,%l1,%l0
ld  [A], %l1
ld  [B], %l2
sub %l1,%l2,%l1
add %l1,%l0,%l0
```

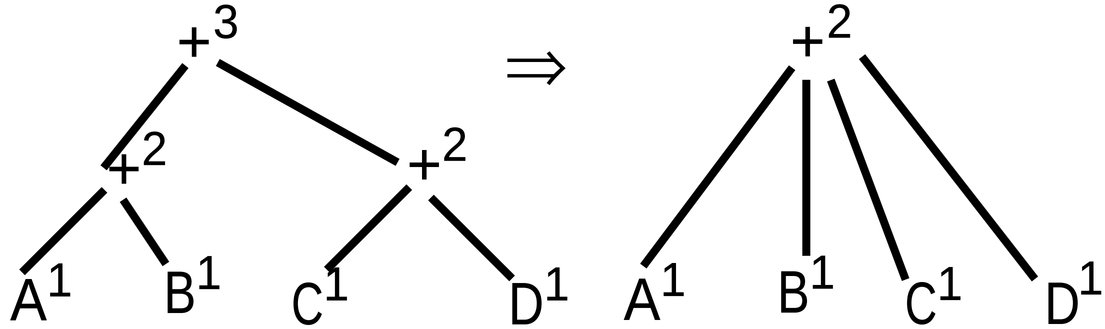
Refinements & Improvements

- 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:

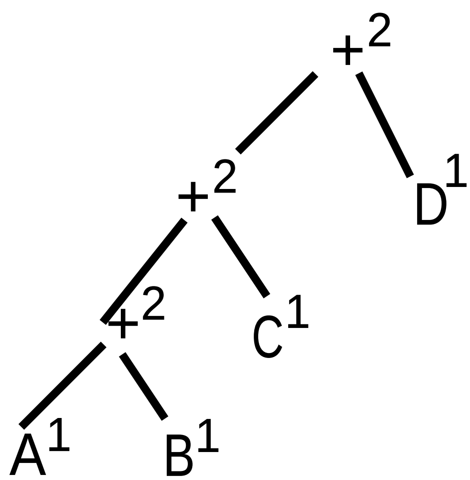
  RN(literal) = 0 if literal may be used as
  an immediate operand

- Commutativity & Associativity of operands may be exploited:



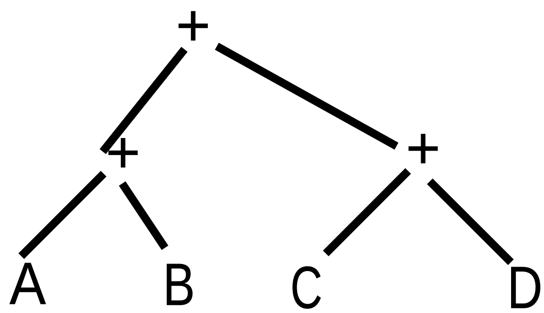
Is Minimizing Register Use Always Wise?

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*(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.