## 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.**

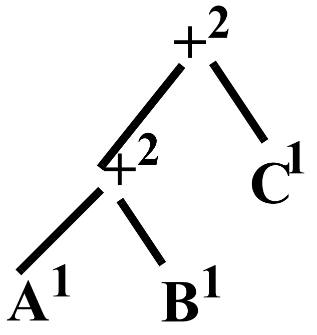
**If a register is used "too soon," the processor may stall execution until the register value becomes available.**

```
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.**

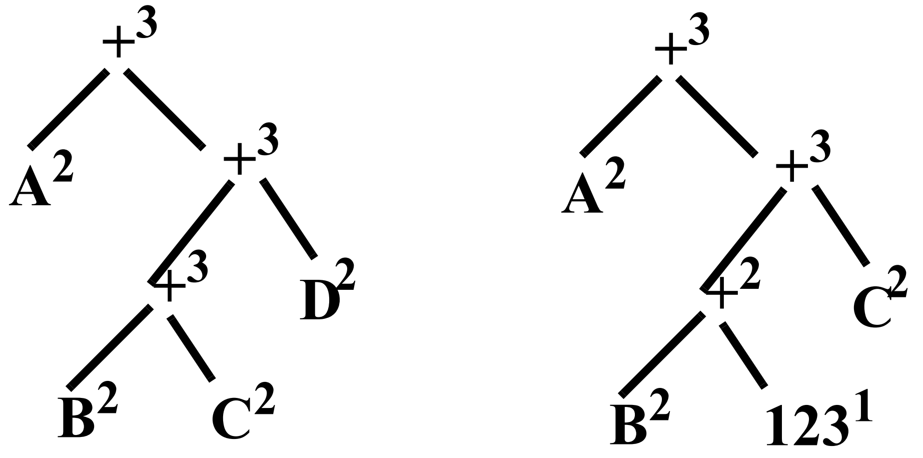
**We'll need to allocate an *extra register* to allow an independent instruction to fill each delay slot of a load.**

## 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:

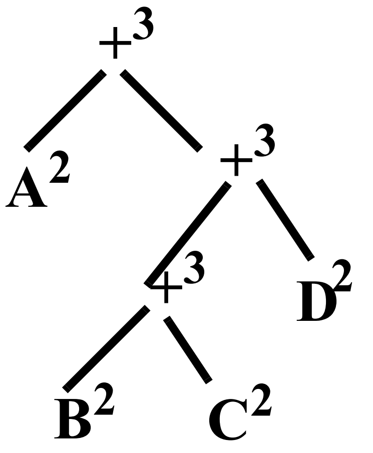
1. R load instructions
2. Pairs of Operator/Load instructions
3. Remaining operators

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.**

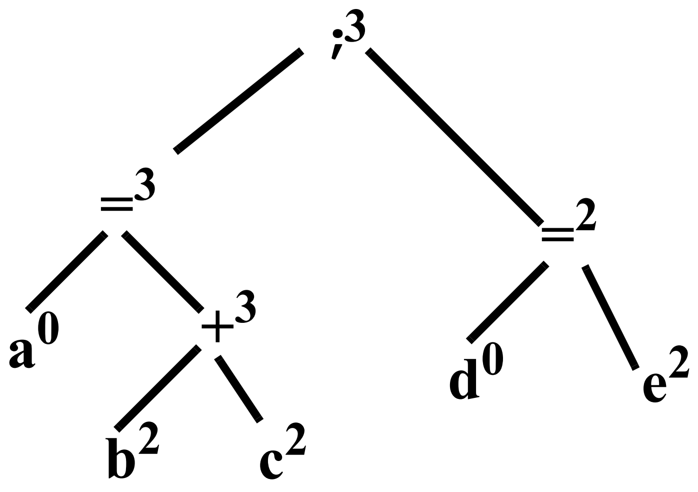
## How Do We Schedule Small Expressions?

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 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**

   $$\textbf{DefsOut(b)} = \{b\}$$

2. **If there is no definition to V in b then**

   $$\textbf{DefsOut(b)} = \textbf{DefsIn(b)}$$

3. **For the First Basic Block, $b_0$:**

   $$\textbf{DefsIn}(b_0) = \phi$$

4. **For all Other Basic Blocks**

   $$\textbf{DefsIn(b)} = \bigcup_{p \in \mathrm{Preds}(b)} \mathrm{DefsOut}(p)$$

## 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:**

**LiveIn(b) = true if V is Live (will be used before it is redefined) at the beginning of b.**

**LiveOut(b) = true if V 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
        LiveOut(b) = false
2. For all Other Basic Blocks
        $LiveOut(b) = \bigvee_{s \in Succ(b)} LiveIn(s)$
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.**

**∀ Basic Blocks, b, that contain a definition of V:**

$$\mathbf{Range(b)} = \{b\} \cup \{k \mid b \in \mathrm{DefsIn}(k)\ \&\ \mathrm{LiveIn}(k)\}$$

**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.**

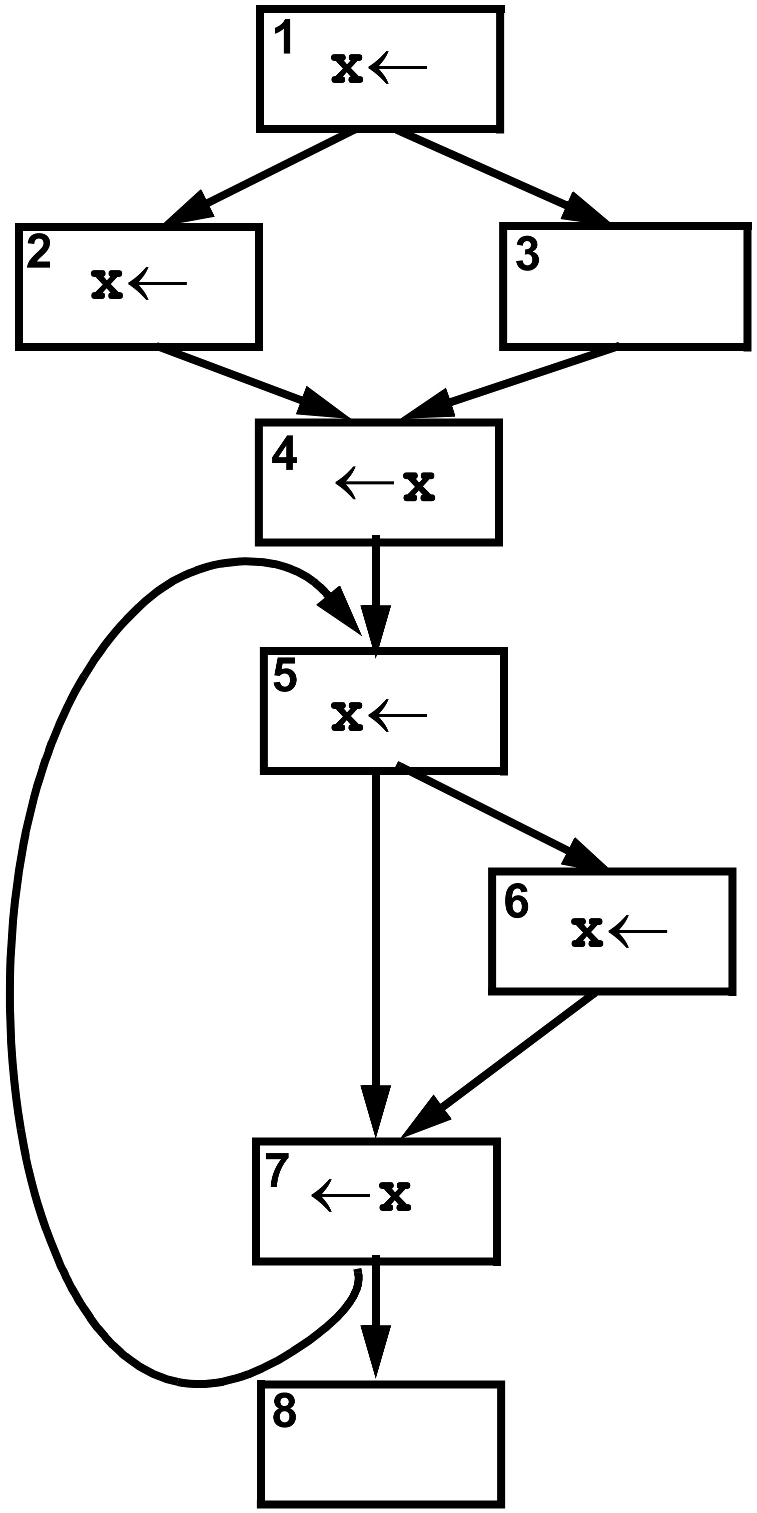
If two Live Ranges overlap (have one of more Basic Blocks in common), they *must* share the same register too. (Why?)

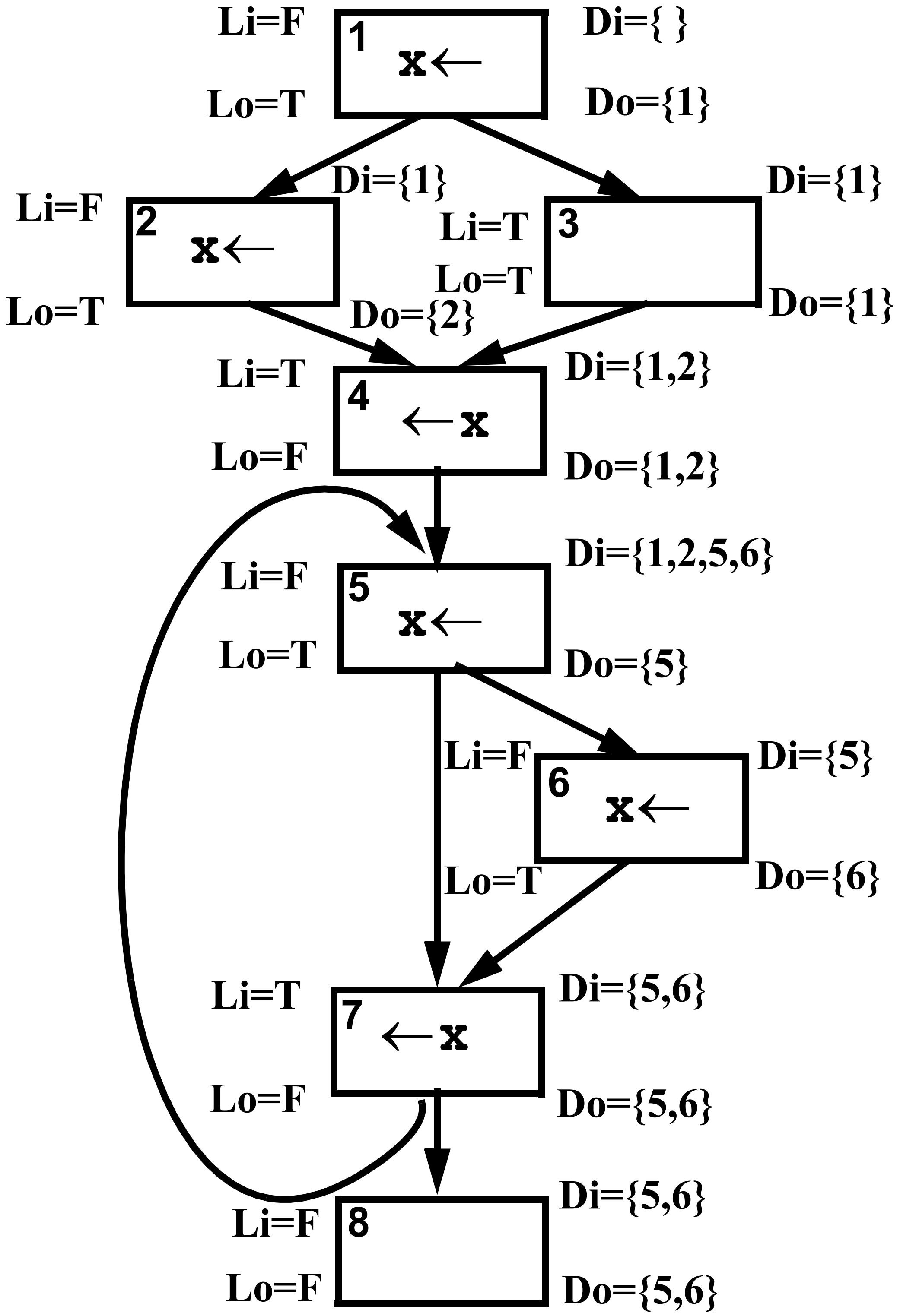
Therefore,

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

Then replace
  $\text{Range}(b_1)$ and $\text{Range}(b_2)$
  with $\text{Range}(b_1) \cup \text{Range}(b_2)$

# Example




Li=F
1
x←
Lo=T
Di={ }
Do={1}
Li=F
2
x←
Lo=T
Di={1}
Do={2}
Li=T
3
Lo=T
Di={1}
Do={1}
Li=T
4
←x
Lo=F
Di={1,2}
Do={1,2}
Li=F
5
x←
Lo=T
Di={1,2,5,6}
Do={5}
Li=F
6
x←
Lo=T
Di={5}
Do={6}
Li=T
7
←x
Lo=F
Di={5,6}
Do={5,6}
Li=F
8
Lo=F
Di={5,6}
Do={5,6}

## 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

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

Ranges 5 and 6 overlap, so

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

## Interference Graph

**An *Interference Graph* represents interferences between Live Ranges.**

**Two Live Ranges *interfere* if they share one or more Basic Blocks in common.**

**Live Ranges that interfere *must* be allocated different registers.**

**In an Interference Graph:**

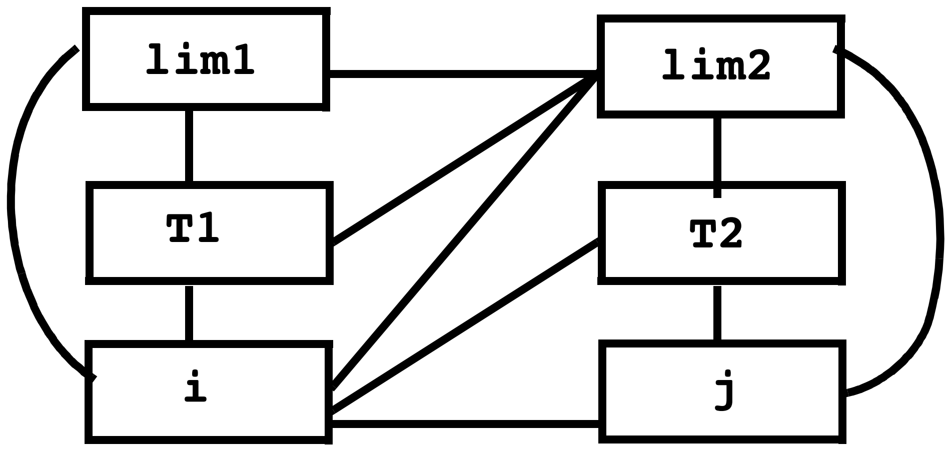
- **Nodes are Live Ranges**
- **An undirected arc connects two Live Ranges if and only if they interfere**

## Example

```
int p(int lim1, int lim2) {
  for (i=0; i<lim1 && A[i]>0;i++){}
  for (j=0; j<lim2 && B[j]>0;j++){}
  return i+j;
}
```

**We optimize array accesses by placing `&A[0]` and `&B[0]` in temporaries:**

```
int p(int lim1, int lim2) {
  int *T1 = &A[0];
  for (i=0; i<lim1 && *(T1+i)>0;i++){}
  int *T2 = &B[0];
  for (j=0; j<lim2 && *(T2+j)>0;j++){}
  return i+j;
}
```



## Register Allocation via Graph Coloring

**We model global register allocation as a Coloring Problem on the Interference Graph**

**We wish to use the fewest possible colors (registers) subject to the rule that two connected nodes can’t share the same color.**

## Optimal Graph Coloring is NP-Complete

**Reference:**

**"Computers and Intractability," M. Garey and D. Johnson, W.H. Freeman, 1979.**

**We'll use a Heuristic Algorithm originally suggested by Chaitin et. al. and improved by Briggs et. al.**

**References:**

**"Register Allocation Via Coloring," G. Chaitin et. al., Computer Languages, 1981.**

**"Improvement to Graph Coloring Register Allocation," P. Briggs et. al., PLDI, 1989.**

## Coloring Heuristic

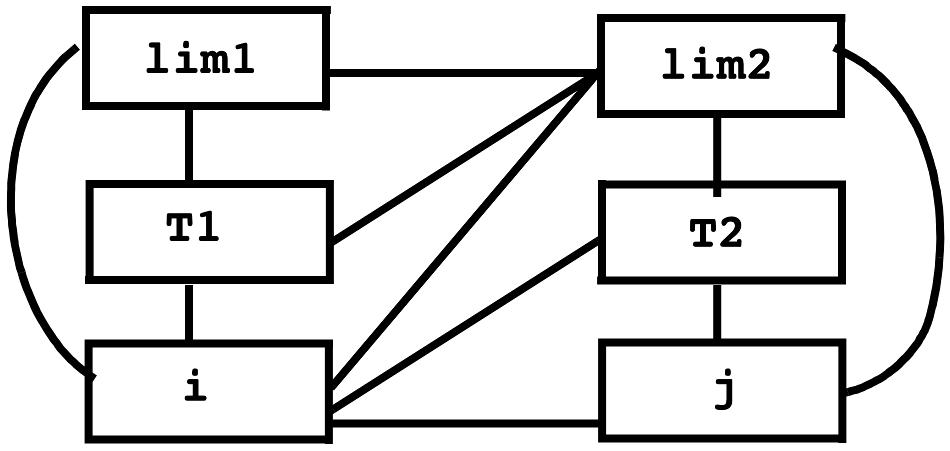
**To R-Color a Graph (where R is the number of registers available)**

```
1. While any node, n, has < R
   neighbors:
       Remove n from the Graph.
       Push n onto a Stack.
2. If the remaining Graph is non-
   empty:
       Compute the Cost of each node.
       The Cost of a Node (a Live Range)
       is the number of extra instructions
       needed if the Node isn't assigned a
       register, scaled by $10^{loop\_depth}$.
       Let NB(n) =
              Number of Neighbors of n.
       Remove that node n that has the
       smallest Cost(n)/NB(n) value.
```

```
    Push n onto a Stack.
    Return to Step 1.
3. While Stack is non-empty:
    Pop n from the Stack.
    If n’s neighbors are assigned fewer
       than R colors
    Then assign n any unassigned
  color
    Else leave n uncolored.
```

## Example

```
    int p(int lim1, int lim2) {
    int *T1 = &A[0];
    for (i=0; i<lim1 && *(T1+i)>0;i++){}
    int *T2 = &B[0];
    for (j=0; j<lim2 && *(T2+j)>0;j++){}
    return i+j;
  }
```



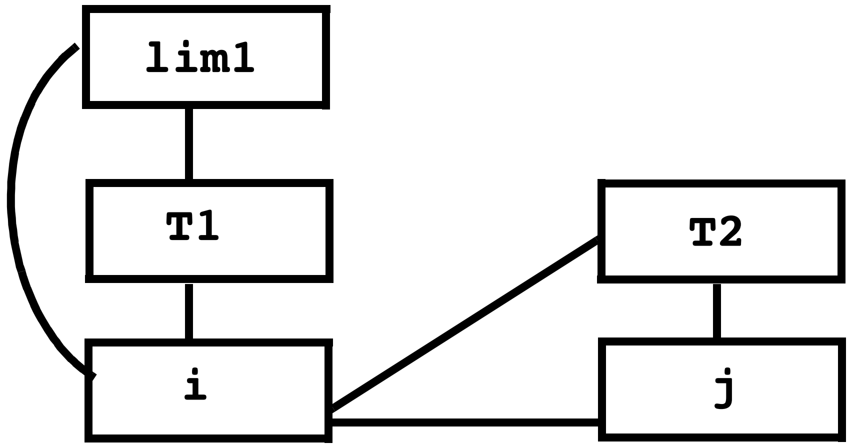
|  | `lim1` | `lim2` | `T1` | `T2` | `i` | `j` |
|---|---|---|---|---|---|---|
| **Cost** | **11** | **11** | **11** | **11** | **42** | **42** |
| **Cost/ Neighbors** | **11/3** | **11/5** | **11/3** | **11/3** | **42/5** | **42/3** |

**Do a 3 coloring**

Since no node has fewer than 3 neighbors, we remove a node based on the minimum Cost/Neighbors value.

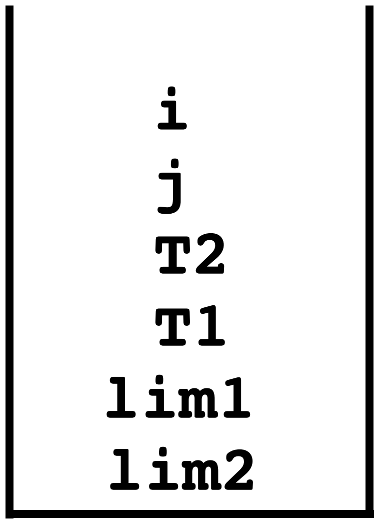
`lim2` is chosen.

We now have:



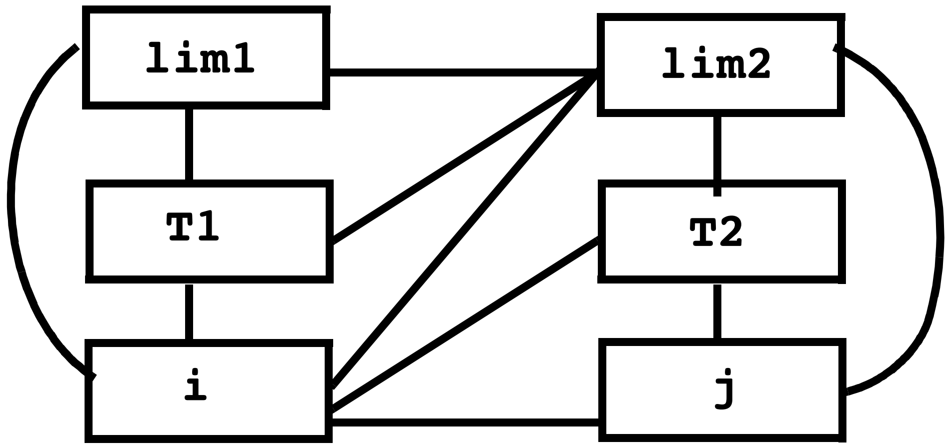
Remove (say) `lim1`, then `T1`, `T2`, `j` and `i` (order is arbitrary).

## The Stack is:



## Assuming the colors we have are R1, R2 and R3, the register assignment we choose is

**`i`:R1, `j`:R2, `T2`:R3, `T1`:R2, `lim1`:R3, `lim2`:spill**



## Color Preferences

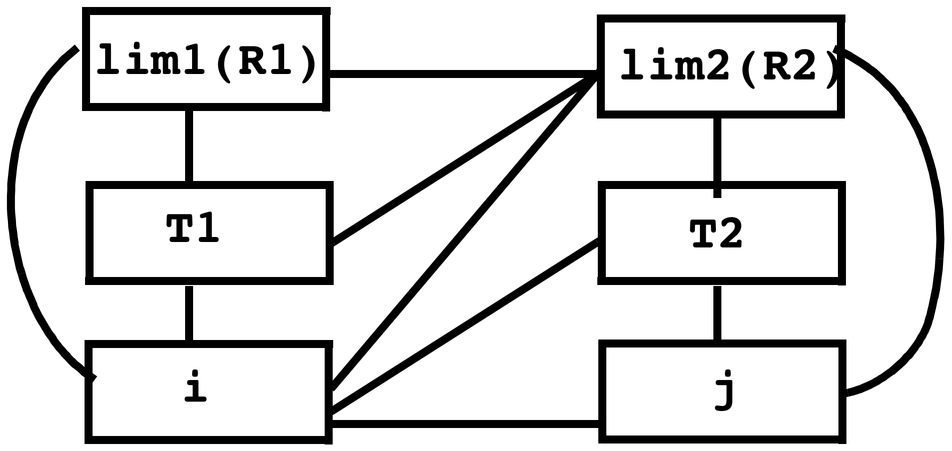
**Sometimes we wish to assign a particular register (color) to a selected Live Range (e.g., a parameter or return value)** *if possible*.

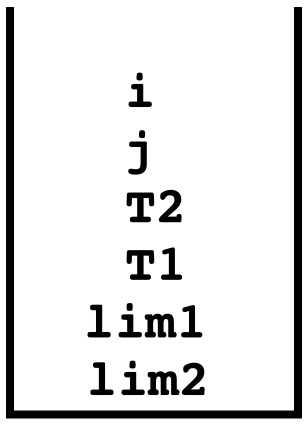
**We can mark a node in the Interference Graph with a** *Color Preference*.

**When we unstack nodes and assign colors, we will avoid choosing color c if an uncolored neighbor has indicted a preference for it. If only color c is left, we take it (and ignore the preference).**

## Example

**Assume in our previous example that lim1 has requested register R1 and lim2 has requested register R2 (because these are the registers the parameters are passed in).**





**Now when `i`, `j` and `T1` are unstacked, they respect `lim1`'s and `lim2`'s preferences:**

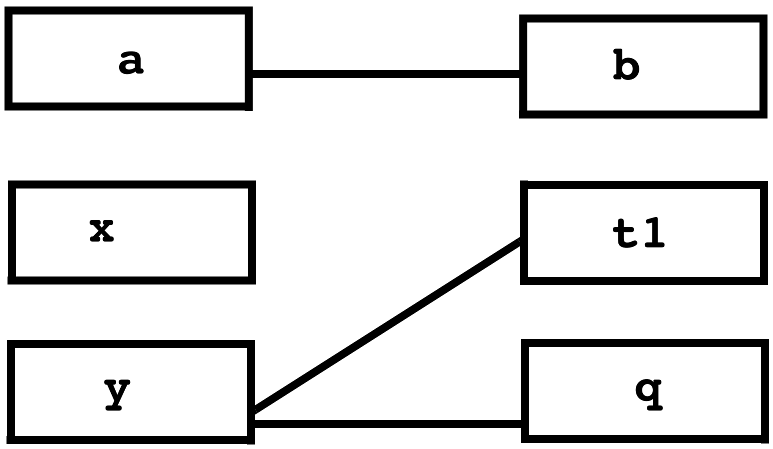
**`i`:R3, `j`:R1, `T2`:R2, `T1`:R2, `lim1`:R1, `lim2`:spill**

## Using Coloring to Optimize Register Moves

**A nice "fringe benefit" of allocating registers via coloring is that we can often *optimize away* register to register moves by giving the source and target the *same color*.**

**Consider**

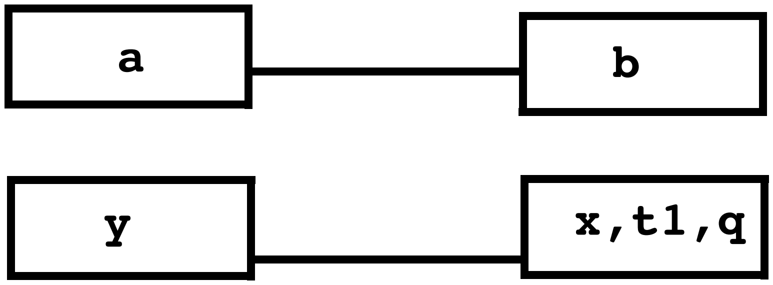
```
Live in: a,b
t1 = a + b
x = t1
y = x + 1
q = t1
Live out: y,q
```



**We'd like `x`, `t1` and `q` to get the same color. How do we "force" this?**

## We can “merge” `x`, `t1` and `q`

```
Live in: a,b
t1 = a + b
x = t1
y = x + 1
q = t1
Live out: y,q
```



together:

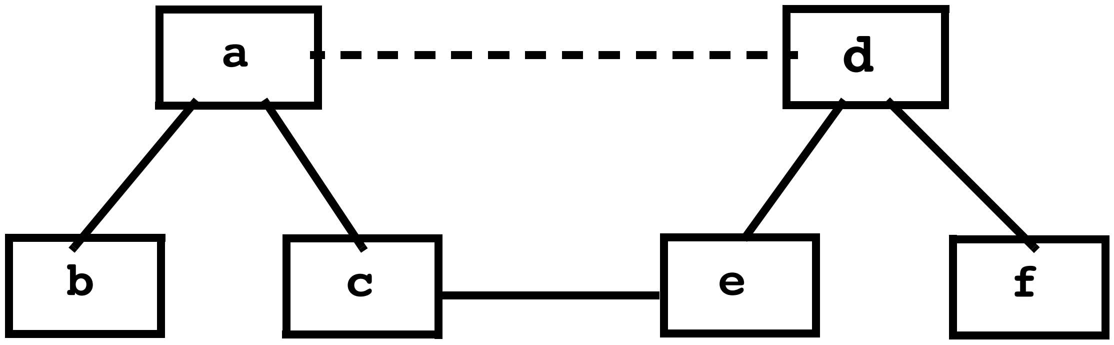
Now a 2-coloring that optimizes away both register to register moves is trivial.

## Reckless Coalescing

**Originally, Chaitin suggested merging *all* move-related nodes that don't interfere.**

**This is *reckless*—the merged node may not be colorable!**

**(Is it worth a spill to save a move??)**



**This Graph is 2-colorable before the reckless merge, but *not* after.**