Compilation

0368-3133 (Semester A, 2013/14)

Lecture 9: Activation Records + Register Allocation

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv and Eran Yahav

Registers

- Most machines have a set of registers, dedicated memory locations that
  - can be accessed quickly,
  - can have computations performed on them, and
- Usages
  - Operands of instructions
  - Store temporary results
  - Can (should) be used as loop indexes due to frequent arithmetic operation
  - Used to manage administrative info
    - e.g., runtime stack

Registers

- Number of registers is limited
- Need to allocate them in a clever way
  - Using registers intelligently is a critical step in any compiler
    - A good register allocator can generate code orders of magnitude better than a bad register allocator

simple code generation

- assume machine instructions of the form
- `LD reg, mem`
- `ST mem, reg`
- `OP reg,reg,reg` (*)
- assume that we have all registers available for our use
  - Ignore registers allocated for stack management
  - Treat all registers as general-purpose

simple code generation

- assume machine instructions of the form
- `LD reg, mem`
- `ST mem, reg`
- `OP reg,reg,reg` (*)

Fixed number of Registers!

"Abstract" Code

- Instructions
  - Load: Memory ➔ Register
  - Store: Register ➔ Memory
  - Operation: R1 = R2 + R3 (*)

Fixed number of Registers

- Assume all registers are available
  - Ignore registers allocated for stack management
  - Treat all registers as general-purpose

Register allocation

- In TAC, there are an unlimited number of variables
- On a physical machine there are a small number of registers:
  - x86 has four general-purpose registers and a number of specialized registers
  - MIPS has twenty-four general-purpose registers and eight special-purpose registers
- Register allocation is the process of assigning variables to registers and managing data transfer in and out of registers

simple code generation

- assume machine instructions of the form
- `LD reg, mem`
- `ST mem, reg`
- `OP reg,reg,reg` (*)

Fixed number of Registers!

- We will assume that we have all registers available for any usage
  - Ignore registers allocated for stack management
  - Treat all registers as general-purpose

Simple approach

- Straightforward solution:
  - Allocate each variable in activation record
  - At each instruction, bring values needed into registers, perform operation, then store result to memory

x = y + z ➜

```
mov 16(%ebp), %eax
mov 20(%ebp), %ebx
add %ebx, %eax
mov %eax, 24(%ebx)
```

- Problem: program execution very inefficient– moving data back and forth between memory and registers

Plan

- Goal: Reduce number of temporaries (registers)
  – Machine-agnostic optimizations
    - Assume unbounded number of registers
  – Machine-dependent optimization
    - Use at most K registers
    - K is machine dependent

Generating Compound Expressions

- Use registers to store temporaries
  – Why can we do it?

- Maintain a counter for temporaries in c
- Initially: c = 0
- **cgen**(e_1 *op* e_2) = {
  Let A = **cgen**(e_1)
  c = c + 1
  Let B = **cgen**(e_2)
  c = c + 1
  Emit(_tc = A *op* B;) // _tc is a register
  Return _tc
  }

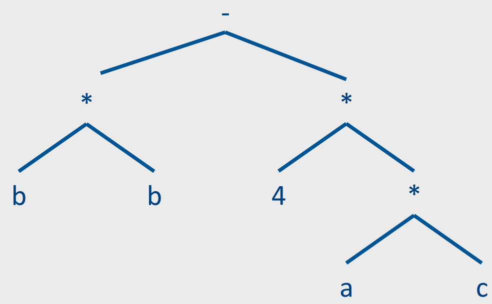
Improving **cgen** for expressions

- Observation – naïve translation needlessly generates temporaries for leaf expressions
- Observation – temporaries used exactly once
  – Once a temporary has been read it can be reused for another sub-expression
- **cgen**(e_1 *op* e_2) = {
  Let _t1 = **cgen**(e_1)
  Let _t2 = **cgen**(e_2)
  Emit(_t =_t1 *op* _t2;)
  Return t
  }
- Temporaries **cgen**(e_1) can be reused in **cgen**(e_2)

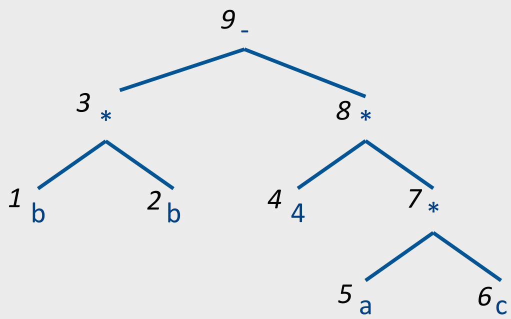
Sethi-Ullman translation

- Algorithm by Ravi Sethi and Jeffrey D. Ullman to emit optimal TAC
  - Minimizes number of temporaries for a **single expression**

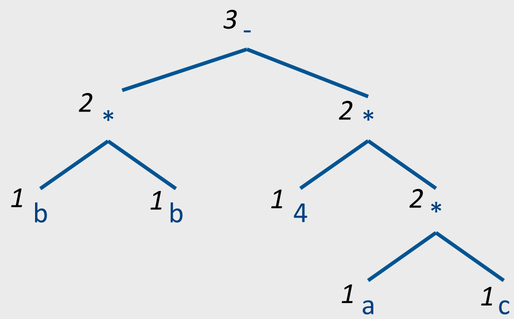
Example: b*b-4*a*c



Example (simple): b*b-4*a*c



Example (optimized): b*b-4*a*c



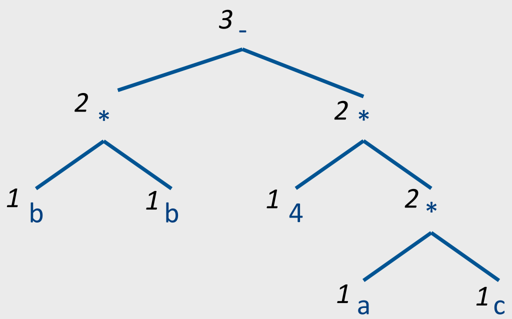
Spilling

- Even an optimal register allocator can require more registers than available
- Need to generate code for every correct program
- The compiler can save temporary results
  - Spill registers into temporaries
  - Load when needed
- Many heuristics exist

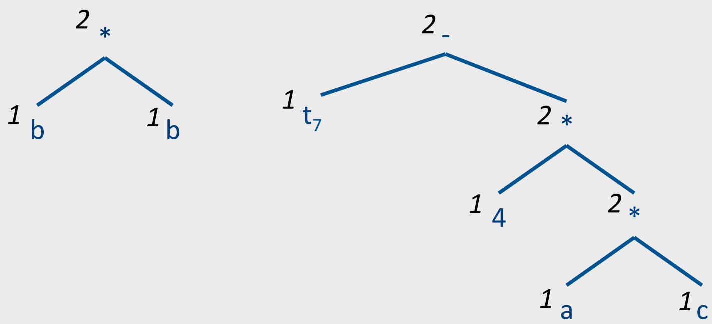
Simple Spilling Method

- Heavy tree – Needs more registers than available
- A `heavy' tree contains a `heavy' subtree whose dependents are 'light'
- Generate code for the light tree
- Spill the content into memory and replace subtree by temporary
- Generate code for the resultant tree

Example (optimized): b*b-4*a*c



Example (spilled): x := b*b-4*a*c




```
t7 := b * b
x := t7 - 4 * a * c
```

Simple Spilling Method

```
PROCEDURE Generate code for large trees (Node, Target register):
    SET Auxiliary register set TO
        Available register set \ Target register;

    WHILE Node /= No node:
        Compute the weights of all nodes of the tree of Node;
        SET Tree node TO Maximal non_large tree (Node);
        Generate code
            (Tree node, Target register, Auxiliary register set);

        IF Tree node /= Node:
            SET Temporary location TO Next free temporary location();
            Emit ("Store R" Target register ",T" Temporary location);
            Replace Tree node by a reference to Temporary location;
            Return any temporary locations in the tree of Tree node
                to the pool of free temporary locations;
        ELSE Tree node = Node:
            Return any temporary locations in the tree of Node
                to the pool of free temporary locations;
            SET Node TO No node;

FUNCTION Maximal non_large tree (Node) RETURNING a node:
    IF Node .weight <= Size of Auxiliary register set: RETURN Node;
    IF Node .left .weight > Size of Auxiliary register set:
        RETURN Maximal non_large tree (Node .left);
    ELSE Node .right .weight >= Size of Auxiliary register set:
        RETURN Maximal non_large tree (Node .right);
```

Generalizations

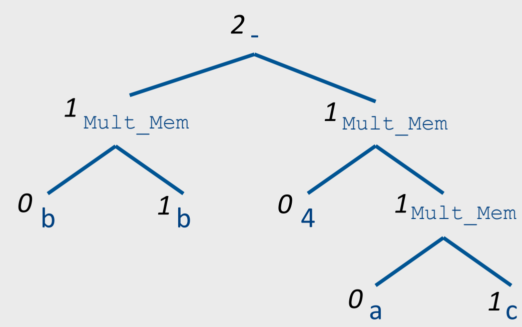
- More than two arguments for operators
  - Function calls
- Register/memory operations
- Multiple effected registers
  - Multiplication
- Spilling
  - Need more registers than available

Register Memory Operations

- Add_Mem X, R1
- Mult_Mem X, R1
- No need for registers to store right operands

Hidden Registers

Example: b*b-4*a*c



Can We do Better?

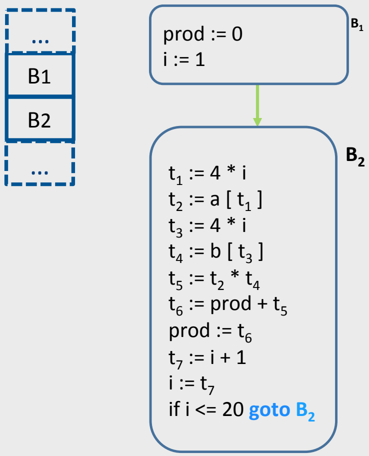
- Yes: Increase view of code
  - Simultaneously allocate registers for multiple expressions
- But: Lose per expression optimality
  - Works well in practice

Basic Blocks

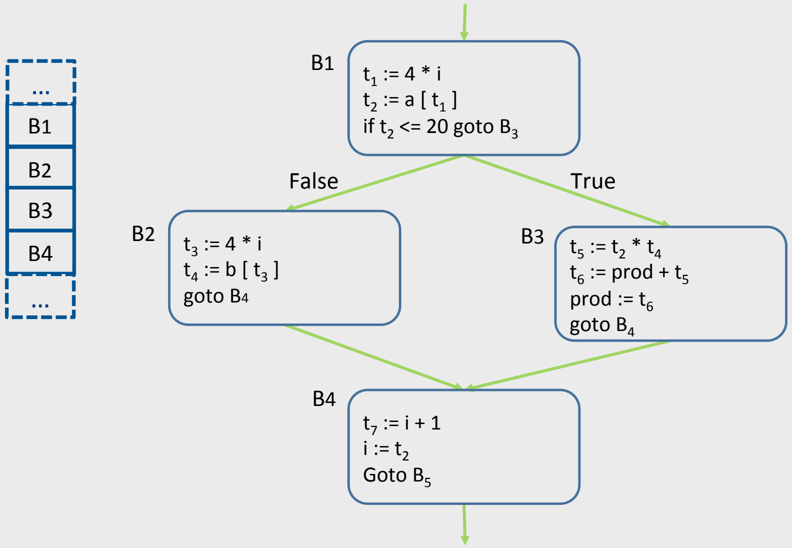
- **basic block** is a sequence of instructions with
  - **single entry** (to first instruction), no jumps to the middle of the block
  - **single exit** (last instruction)
  - code execute as a sequence from first instruction to last instruction without any jumps
- edge from one basic block B1 to another block B2 when the last statement of B1 may jump to B2

control flow graph

- A directed graph G=(V,E)
- nodes V = basic blocks
- edges E = control flow
  - (B1,B2) ∈ E when control from B1 flows to B2



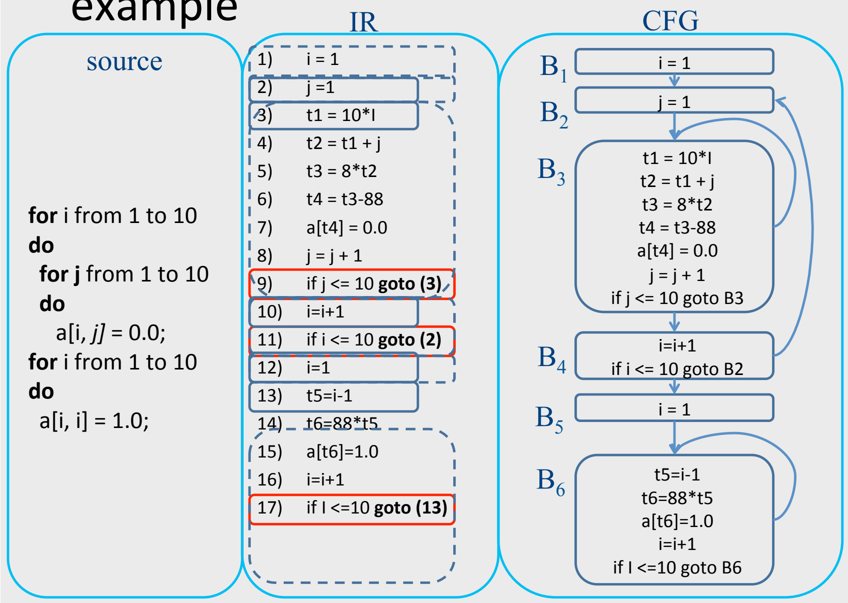
Another Example



Creating Basic Blocks

- **Input**: A sequence of three-address statements
- **Output**: A list of basic blocks with each three-address statement in exactly one block
- **Method**
  - Determine the set of **leaders** (first statement of a block)
    - The first statement is a leader
    - Any statement that is the target of a jump is a leader
    - Any statement that immediately follows a jump is a leader
  - For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

example



Example: Code Block

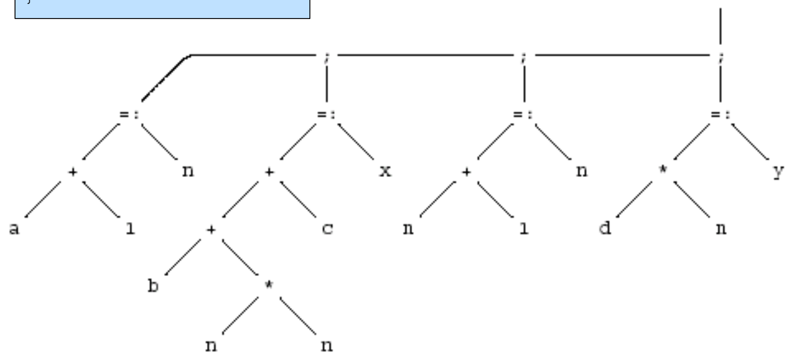
```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

Example: Basic Block

```
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

AST of the Example

```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```



Optimized Code (gcc)

```
{
  int n;
  n := a + 1;
  x := b + n * n + c;
  n := n + 1;
  y := d * n;
}
```

```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg     R1,R2
Mult_Reg     R1,R2
Add_Mem      b,R2
Add_Mem      c,R2
Store_Reg    R2,x
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg    R1,y
```

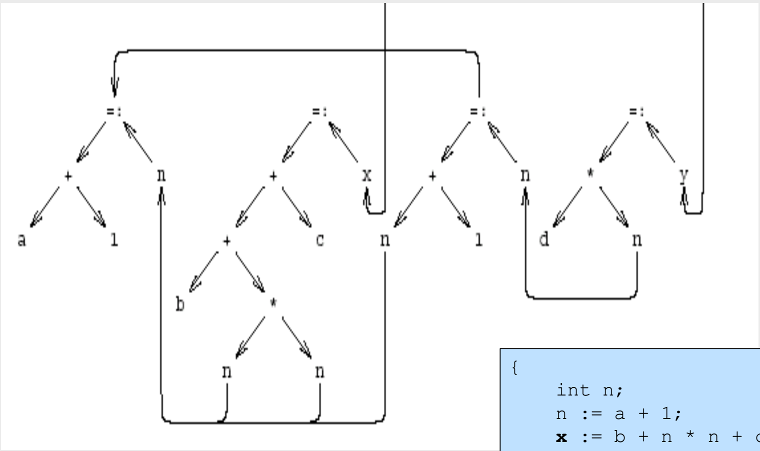
Register Allocation for B.B.

- Dependency graphs for basic blocks
- Transformations on dependency graphs
- From dependency graphs into code
  - Instruction selection
    - linearizations of dependency graphs
  - Register allocation
    - At the basic block level

Dependency graphs

- TAC imposes an order of execution
  - But the compiler can reorder assignments as long as the program results are not changed
- Define a partial order on assignments
  - a < b ⇔ a must be executed before b
  - Represented as a directed graph
    - Nodes are assignments
    - Edges represent dependency
  - Acyclic for basic blocks

Running Example



```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

Sources of dependency

- Data flow inside expressions
  - Operator depends on operands
  - Assignment depends on assigned expressions
- Data flow between statements
  - From assignments to their use
  - Pointers complicate dependencies

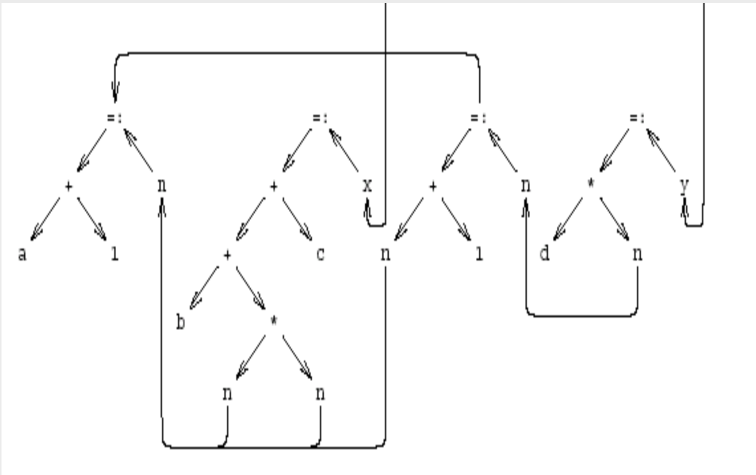
Sources of dependency

- Order of subexpresion evaluation is immaterial
  - As long as inside dependencies are respected
- The order of uses of a variable X are immaterial as long as:
  - X is used between dependent assignments
  - Before next assignment to X

Creating Dependency Graph from AST

- Nodes AST becomes nodes of the graph
- Replaces arcs of AST by dependency arrows
  - Operator → Operand
  - Create arcs from assignments to uses
  - Create arcs between assignments of the same variable
- Select output variables (roots)
- Remove ; nodes and their arrows

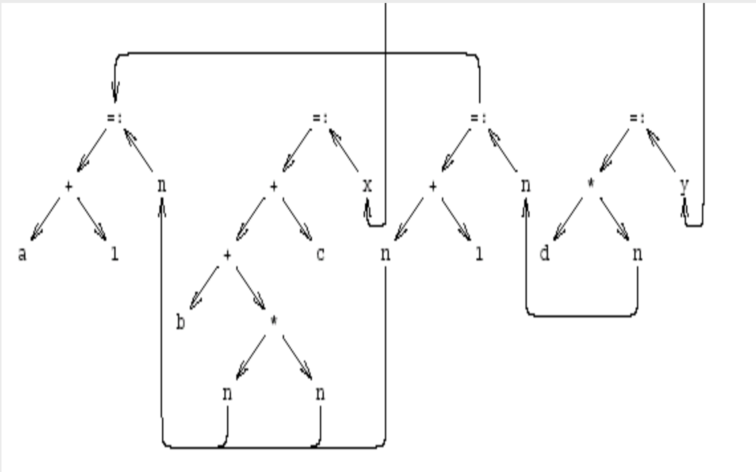
Running Example



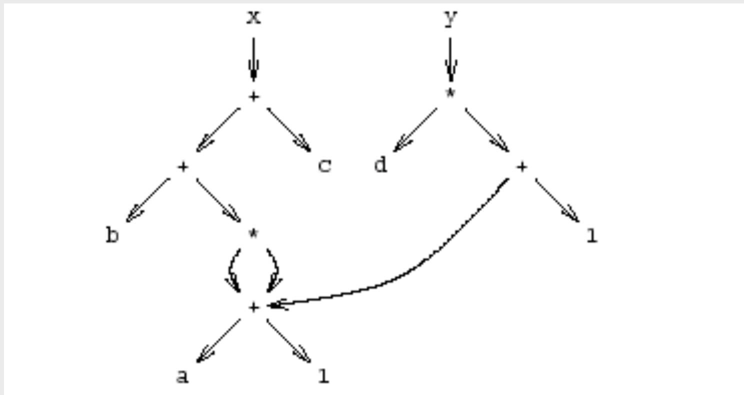
Dependency Graph Simplifications

- Short-circuit assignments
  - Connect variables to assigned expressions
  - Connect expression to uses
- Eliminate nodes not reachable from roots

Running Example



Cleaned-Up Data Dependency Graph



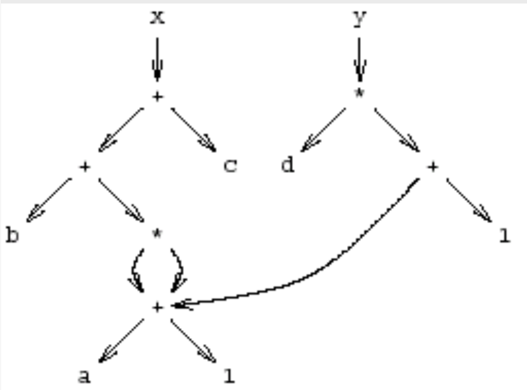
Common Subexpressions

- Repeated subexpressions
- Examples
  x = a * a + 2 * a * b + b * b;
  y = a * a – 2 * a * b + b * b;
  n[i] := n[i] +m[i]
- Can be eliminated by the compiler
  - In the case of basic blocks rewrite the DAG

From Dependency Graph into Code

- Linearize the dependency graph
  - Instructions must follow dependency
- Many solutions exist
- Select the one with small runtime cost
- Assume infinite number of registers
  - Symbolic registers
  - Assign registers later
    - May need additional spill
  - Possible Heuristics
    - Late evaluation
    - Ladders

Pseudo Register Target Code




```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,X1

Load_Reg    X1,R1
Mult_Reg    X1,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x

Load_Reg    X1,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

Non optimized vs Optimized Code

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,X1

Load_Reg    X1,R1
Mult_Reg    X1,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x

Load_Reg    X1,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

```
{
  int n;
  n := a + 1;
  x := b + n * n + c;
  n := n + 1;
  y := d * n;
}
```

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,R2

Load_Reg    R2,R1
Mult_Reg    R2,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x

Load_Reg    R2,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,R2

Mult_Reg    R1,R2
Add_Mem     b,R2
Add_Mem     c,R2
Store_Reg   R2,x

Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

Register Allocation

- Maps symbolic registers into physical registers
  - Reuse registers as much as possible
  - Graph coloring (next)
    - Undirected graph
    - Nodes = Registers (Symbolic and real)
    - Edges = Interference
    - May require spilling

Register Allocation for Basic Blocks

- Heuristics for code generation of basic blocks
- Works well in practice
- Fits modern machine architecture
- Can be extended to perform other tasks
  - Common subexpression elimination
- But basic blocks are small
- Can be generalized to a procedure

Problem	Technique	Quality
Expression trees, using register-register or memory-register instructions	Weighted trees; Figure 4.30	
with sufficient registers:		Optimal
with insufficient registers:		Optimal
Dependency graphs, using register-register or memory-register instructions	Ladder sequences; Section 4.2.5.2	Heuristic
Expression trees, using any instructions with cost function	Bottom-up tree rewriting; Section 4.2.6	
with sufficient registers:		Optimal
with insufficient registers:		Heuristic
Register allocation when all interferences are known	Graph coloring; Section 4.2.7	Heuristic

Global Register Allocation

Variable Liveness

- A statement x = y + z
  - **defines** x
  - **uses** y and z
- A variable x is live at a program point if its value (at this point) is used at a later point

y = 42	x undef, y live, z undef
z = 73	x undef, y live, z live
x = y + z	x is live, y dead, z dead
print(x);	x is dead, y dead, z dead

(showing state after the statement)

Computing Liveness Information

- between basic blocks – dataflow analysis (next lecture)
- within a single basic block?
- idea
  - use symbol table to record next-use information
  - scan basic block backwards
  - update next-use for each variable

Computing Liveness Information

- INPUT: A basic block B of three-address statements. symbol table initially shows all non-temporary variables in B as being live on exit.
- OUTPUT: At each statement i: x = y + z in B, liveness and next-use information of x, y, and z at i.
- Start at the last statement in B and scan backwards
  - At each statement i: x = y + z in B, we do the following:
  1. Attach to i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
  2. In the symbol table, set x to "not live" and "no next use.“
  3. In the symbol table, set y and z to "live" and the next uses of y and z to i

Computing Liveness Information

- Start at the last statement in B and scan backwards
  - At each statement i: x = y + z in B, we do the following:
  1. Attach to i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
  2. In the symbol table, set x to "not live" and "no next use.“
  3. In the symbol table, set y and z to "live" and the next uses of y and z to i

```
x = 1
y = x + 3
z = x * 3
x = x * z
```

can we change the order between 2 and 3?

simple code generation

- translate each TAC instruction separately
- For each register, a **register descriptor** records the variable names whose current value is in that register
  - we use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty
  - As code generation progresses, each register will hold the value of zero or more names
- For each program variable, an **address descriptor** records the location(s) where the current value of the variable can be found
  - The location may be a register, a memory address, a stack location, or some set of more than one of these
  - Information can be stored in the symbol-table entry for that variable

simple code generation

For each three-address statement `x := y op z`,

1. Invoke ***getreg*** (`x := y op z`) to select registers R_x, R_y, and R_z
2. If Ry does not contain y, issue: `LD` R_y`, y'` for a location y' of y
3. If Rz does not contain z, issue: `LD` R_z`, z'` for a location z' of z
4. Issue the instruction `OP` R_x, R_y, R_z
5. Update the address descriptors of x, y, z, if necessary
   - R_x is the only location of x now, and
     R_x contains only x (remove R_x from other address descriptors)

*The function **getreg** is not defined yet, for now think of it as an oracle that gives us 3 registers for an instruction*

Find a register allocation

variable	register
a	?
b	?
c	?

register

eax

ebx

b = a + 2

c = b * b

b = c + 1

return b * a

Is this a valid allocation?

variable	register
a	eax
b	ebx
c	eax

register

eax

ebx

b = a + 2	ebx = eax + 2
c = b * b	eax = ebx * ebx
b = c + 1	ebx = eax + 1
return b * a	return ebx * eax

Overwrites previous value of 'a' also stored in eax

Is this a valid allocation?

variable	register
a	eax
b	ebx
c	eax

register

eax

ebx

```
b = a + 2

c = b * b

b = c + 1

return b * a
```

```
ebx = eax + 2

eax = ebx * ebx

ebx = eax + 1

return ebx * eax
```

Value of ‘c’ stored in eax is not needed anymore so reuse it for ‘b’

Main idea

- For every node n in CFG, we have out[n]
  - Set of temporaries live out of n
- Two variables *interfere* if they appear in the same out[n] of any node n
  - **Cannot be allocated to the same register**
- Conversely, if two variables do not interfere with each other, they can be assigned the same register
  - We say they have disjoint live ranges
- How to assign registers to variables?

Interference graph

- Nodes of the graph = variables
- Edges connect variables that interfere with one another
- Nodes will be assigned a color corresponding to the register assigned to the variable
- Two colors can’t be next to one another in the graph

Interference graph construction

```
b = a + 2

c = b * b

b = c + 1

return b * a
```

Interference graph construction

b = a + 2

c = b * b

b = c + 1

{b, a}

return b * a

Interference graph construction

b = a + 2

c = b * b

{a, c}

b = c + 1

{b, a}

return b * a

Interference graph construction

b = a + 2

{b, a}

c = b * b

{a, c}

b = c + 1

{b, a}

return b * a

Interference graph construction

{a}

b = a + 2

{b, a}

c = b * b

{a, c}

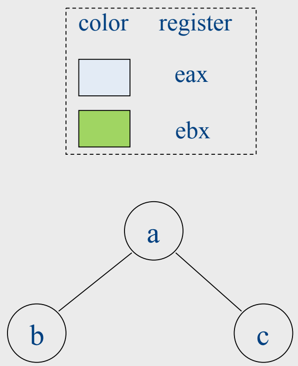
b = c + 1

{b, a}

return b * a

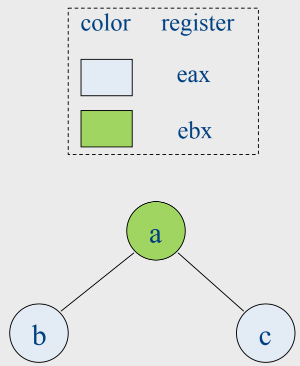
Interference graph

{a}
b = a + 2
{b, a}
c = b * b
{a, c}
b = c + 1
{b, a}
return b * a



Colored graph

{a}
b = a + 2
{b, a}
c = b * b
{a, c}
b = c + 1
{b, a}
return b * a



Graph coloring

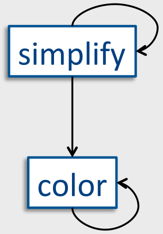
- This problem is equivalent to graph-coloring, which is NP-hard if there are at least three registers
- No good polynomial-time algorithms (or even good approximations!) are known for this problem
  - We have to be content with a heuristic that is good enough for RIGs that arise in practice

Coloring by simplification [Kempe 1879]

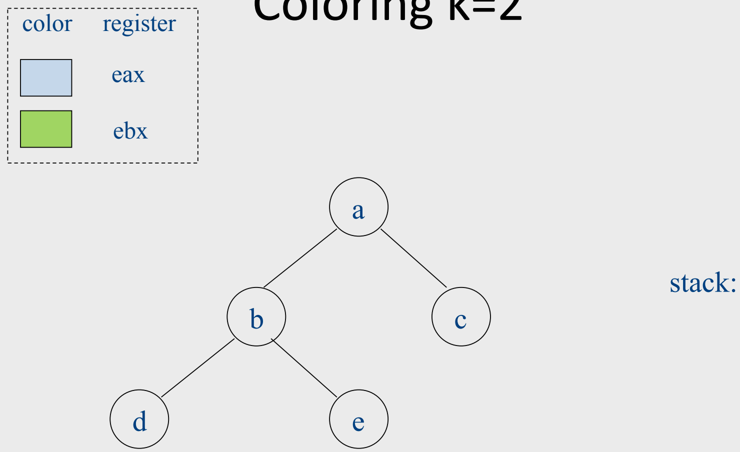
- How to find a **k**-coloring of a graph
- Intuition:
  - Suppose we are trying to ***k****-color a graph and find a node* with fewer than ***k*** *edges*
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in
  - Reason: fewer than *k neighbors* → *some color must be* left over

Coloring by simplification [Kempe 1879]

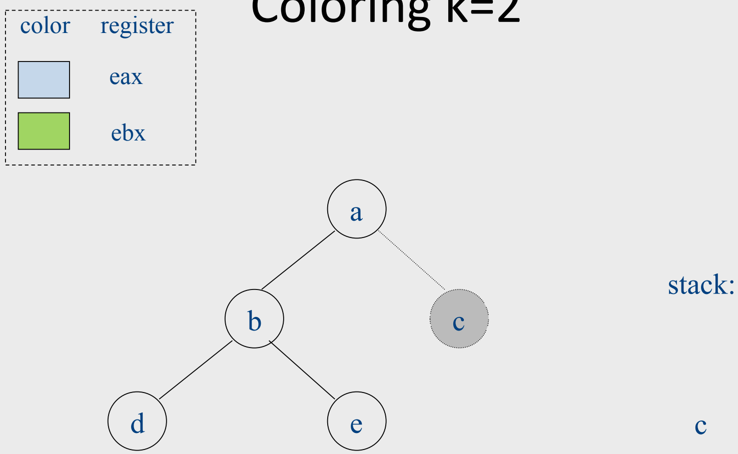
- How to find a k-coloring of a graph
- Phase 1: Simplification
  - Repeatedly simplify graph
  - When a variable (i.e., graph node) is removed, push it on a stack
- Phase 2: Coloring
  - Unwind stack and reconstruct the graph as follows:
  - Pop variable from the stack
  - Add it back to the graph
  - Color the node for that variable with a color that it doesn't interfere with



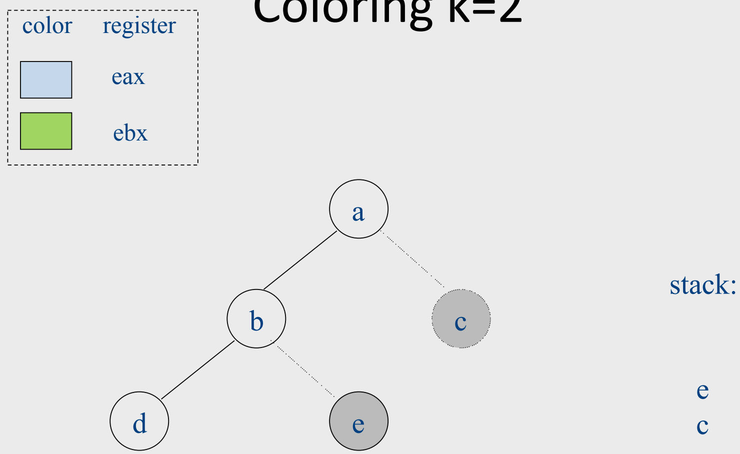
Coloring k=2

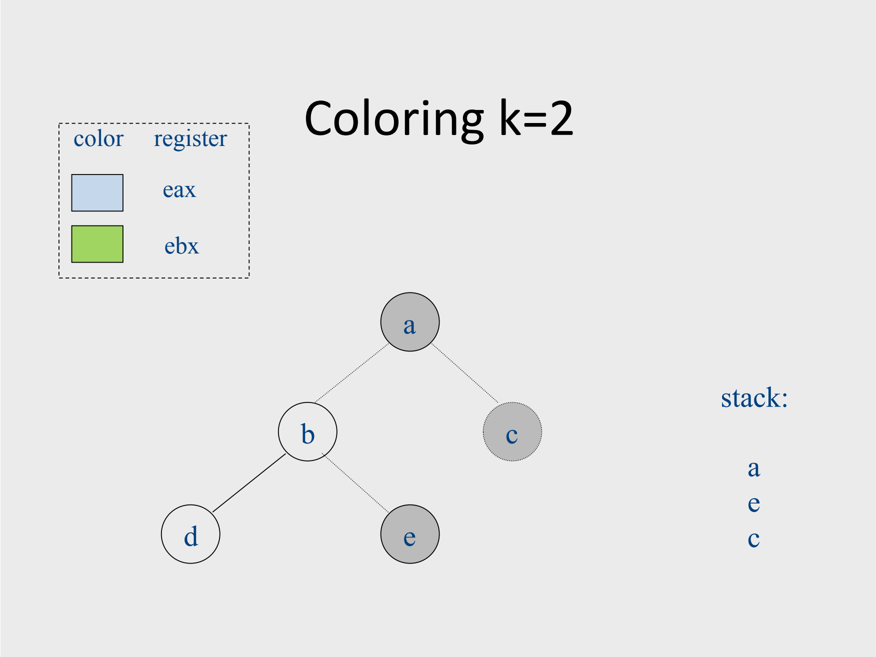


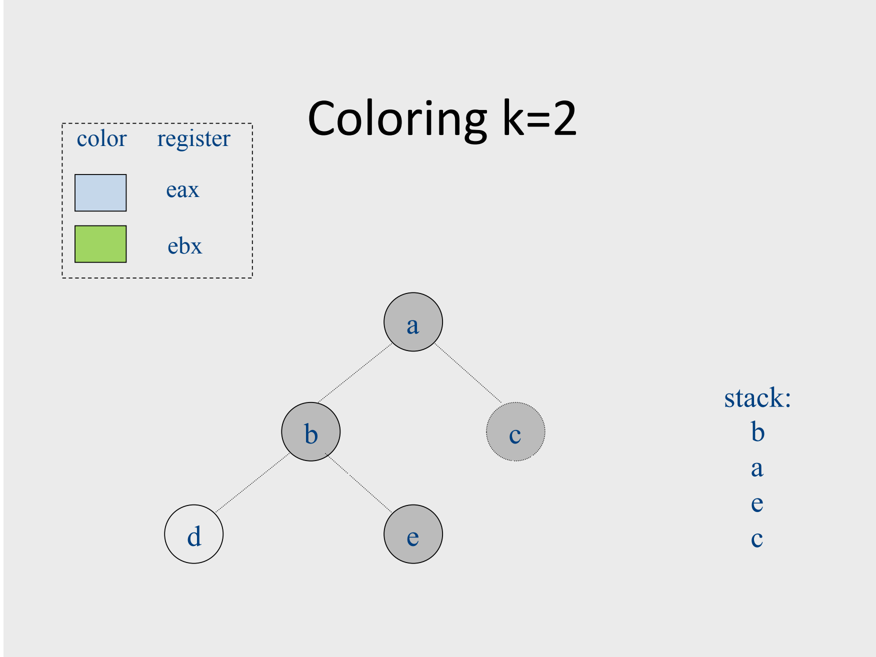
Coloring k=2

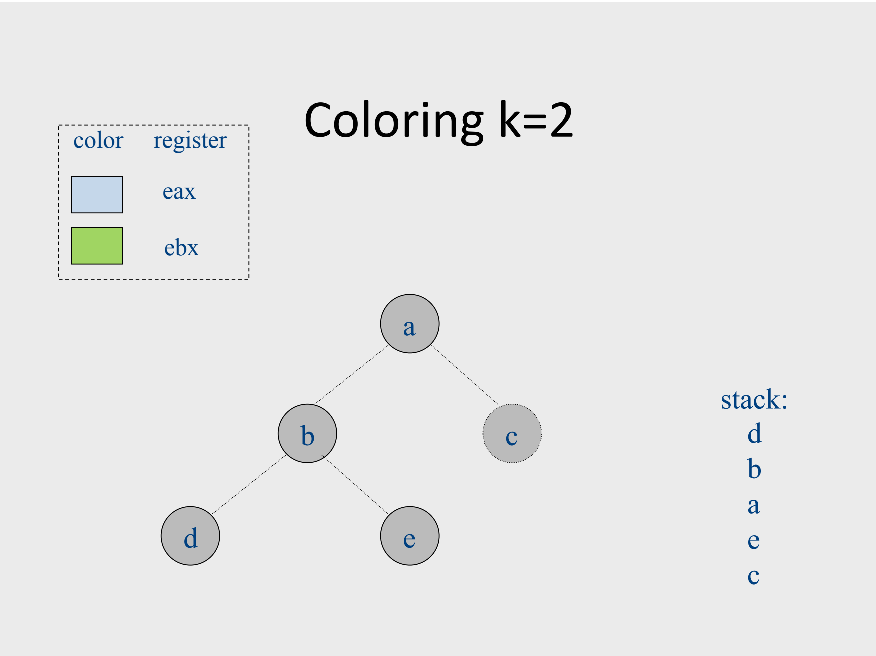


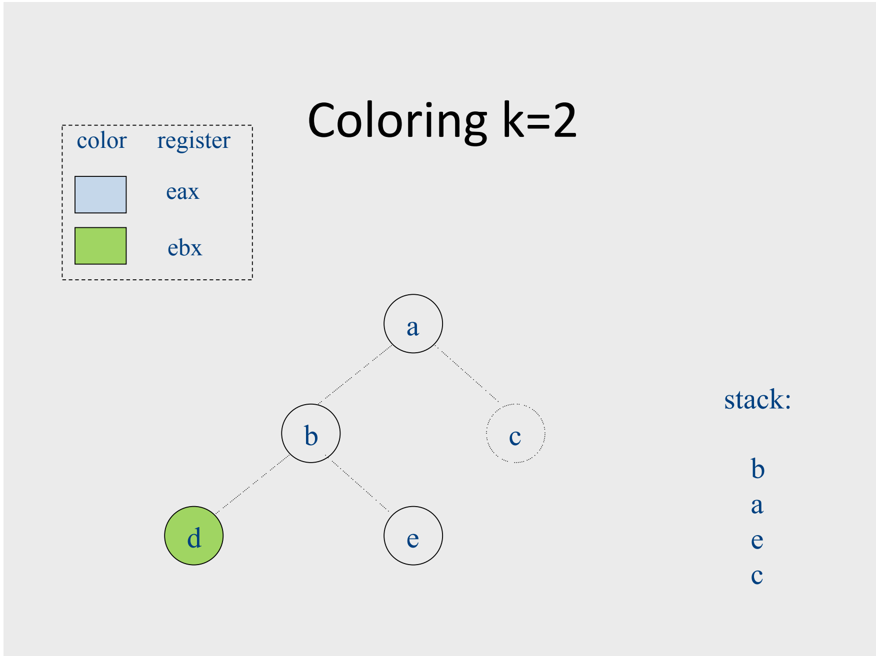
Coloring k=2



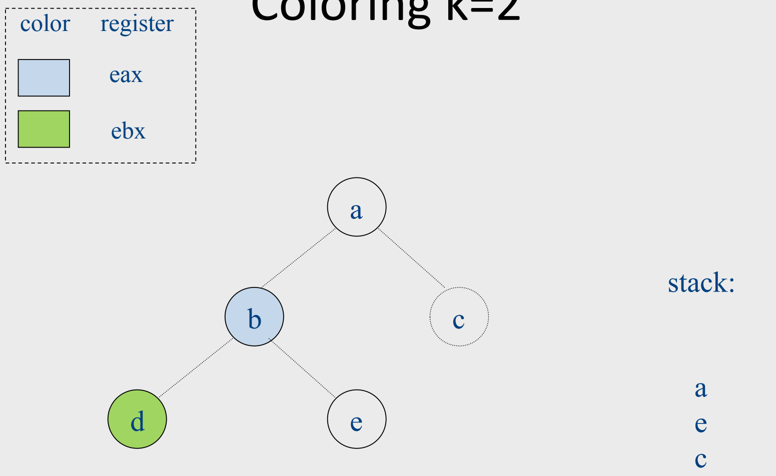

Coloring k=2
color register
eax
ebx
a
b
c
d
e
stack:
a
e
c


Coloring k=2
color register
eax
ebx
a
b
c
d
e
stack:
b
a
e
c

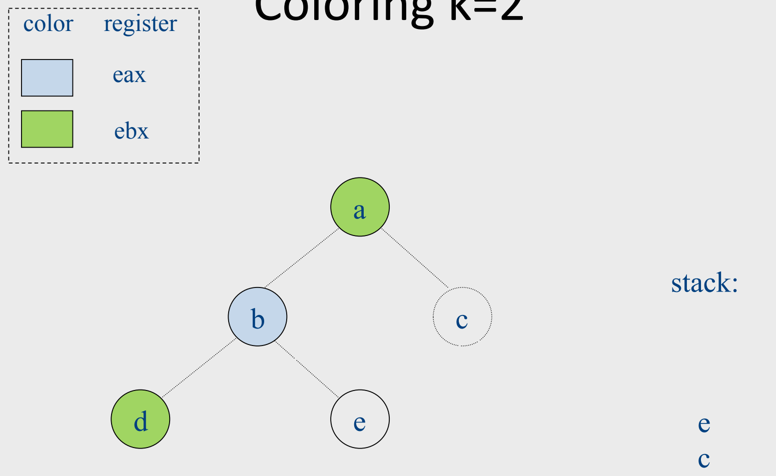

Coloring k=2
color register
eax
ebx
a
b
c
d
e
stack:
d
b
a
e
c


Coloring k=2
color register
eax
ebx
a
b
c
d
e
stack:
b
a
e
c

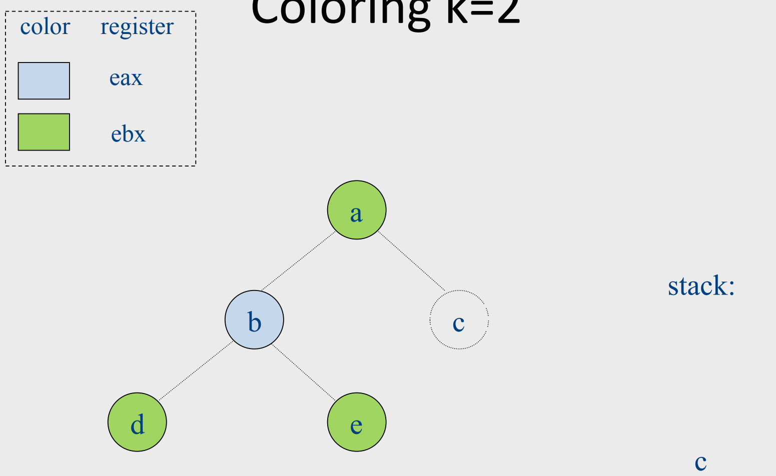
Coloring k=2



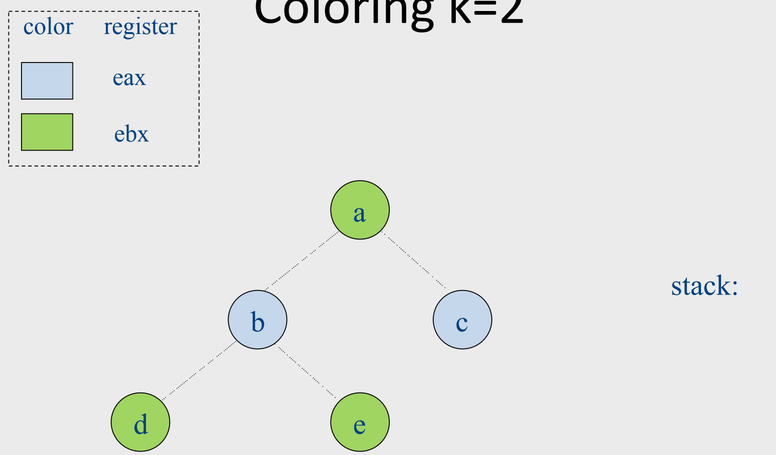
Coloring k=2



Coloring k=2



Coloring k=2

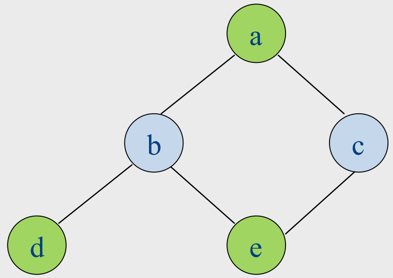


Failure of heuristic

- If the graph cannot be colored, it will eventually be simplified to graph in which every node has at least K neighbors
- Sometimes, the graph is still K-colorable!
- Finding a K-coloring in all situations is an NP-complete problem
  - We will have to approximate to make register allocators fast enough

Coloring k=2

color	register
	eax
	ebx

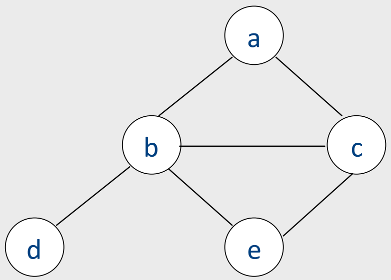


stack:

Coloring k=2

color	register
	eax
	ebx

Some graphs can't be colored in K colors:

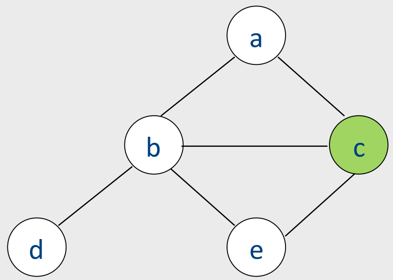


stack:
c
b
e
a
d

Coloring k=2

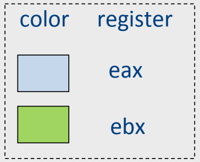
color	register
	eax
	ebx

Some graphs can't be colored in K colors:

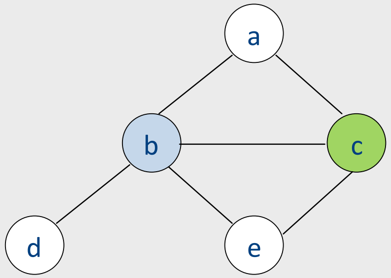


stack:
b
e
a
d

Coloring k=2

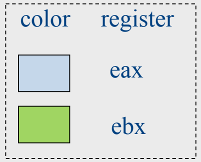


Some graphs can't be colored in K colors:

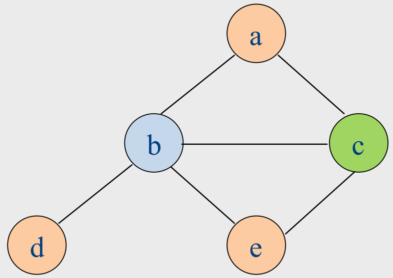


stack:
e
a
d

Coloring k=2



Some graphs can't be colored in K colors:



no colors left for e!

stack:
e
a
d

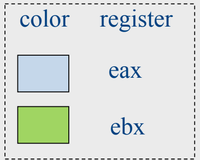
Chaitin's algorithm

- Choose and remove an arbitrary node, marking it "troublesome"
  – Use heuristics to choose which one
  – When adding node back in, it may be possible to find a valid color
  – Otherwise, we have to spill that node

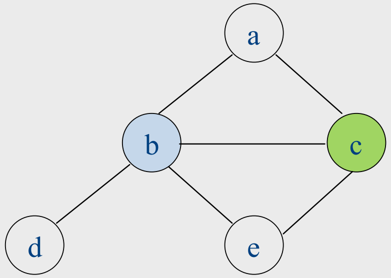
Spilling

- Phase 3: spilling
  – once all nodes have K or more neighbors, pick a node for spilling
    - There are many heuristics that can be used to pick a node
    - Try to pick node not used much, not in inner loop
    - Storage in activation record
  – Remove it from graph
- We can now repeat phases 1-2 without this node
- Better approach – rewrite code to spill variable, recompute liveness information and try to color again

Coloring k=2



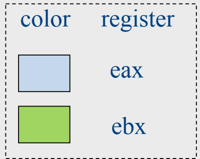
Some graphs can't be colored
in K colors:



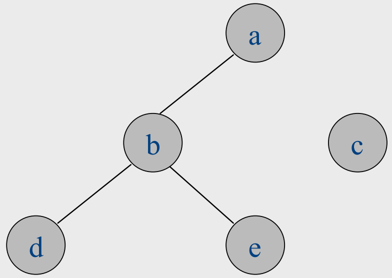
stack:
e
a
d

no colors left for e!

Coloring k=2

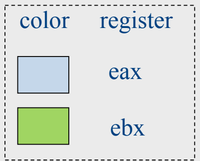


Some graphs can't be colored
in K colors:

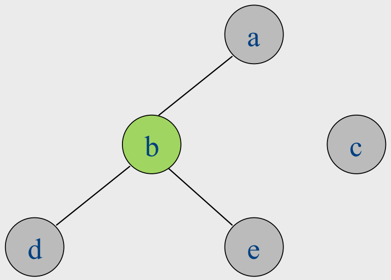


stack:
b
e
a
d

Coloring k=2

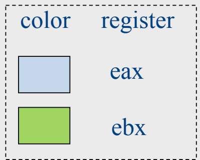


Some graphs can't be colored
in K colors:

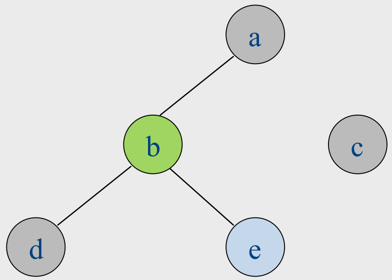


stack:
e
a
d

Coloring k=2

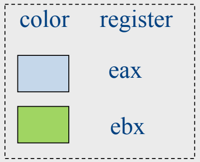


Some graphs can't be colored
in K colors:

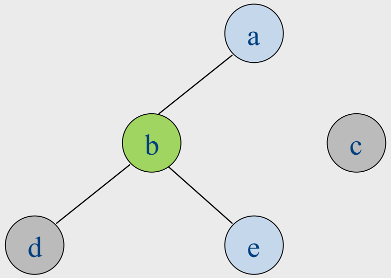


stack:
a
d

Coloring k=2

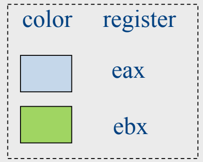


Some graphs can't be colored in K colors:

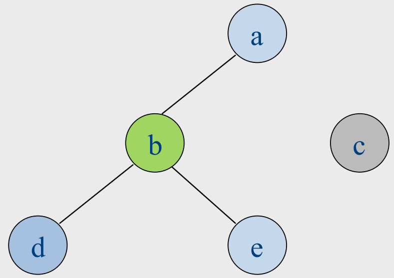


stack:
d

Coloring k=2



Some graphs can't be colored in K colors:



stack:

Handling precolored nodes

- Some variables are pre-assigned to registers
  - Eg: mul on x86/pentium
    - uses eax; defines eax, edx
  - Eg: call on x86/pentium
    - Defines (trashes) caller-save registers eax, ecx, edx
- To properly allocate registers, treat these register uses as special temporary variables and enter into interference graph as precolored nodes

Handling precolored nodes

- Simplify. Never remove a pre-colored node – it already has a color, i.e., it is a given register
- Coloring. Once simplified graph is all colored nodes, add other nodes back in and color them using precolored nodes as starting point

Optimizing move instructions

- Code generation produces a lot of extra mov instructions

  mov t5, t9
- If we can assign t5 and t9 to same register, we can get rid of the mov
  - effectively, copy elimination at the register allocation level
- Idea: if t5 and t9 are not connected in inference graph, coalesce them into a single variable; the move will be redundant
- Problem: coalescing nodes can make a graph un-colorable
  - Conservative coalescing heuristic

The End

global register allocation

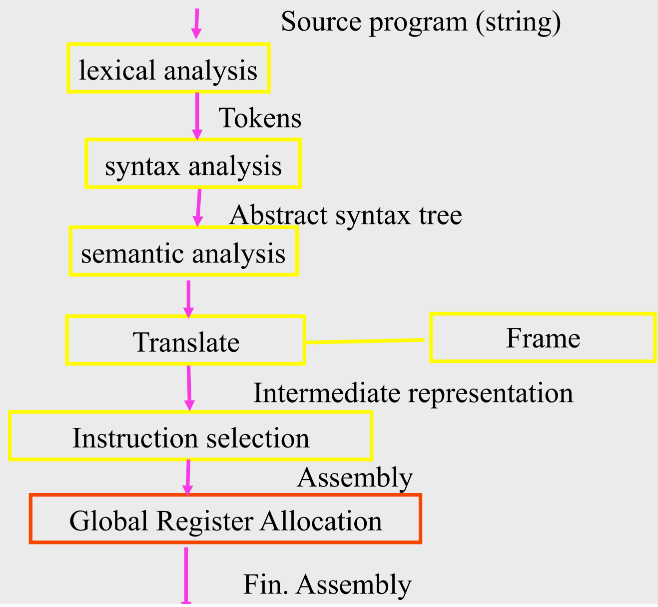
- idea: compute "weight" for each variable
  - for each use of v in B prior to any definition of v add 1 point
  - for each occurrence of v in a following block using v add 2 points, as we save the store/load between blocks
  - **cost(v) = $\Sigma_B$use(v,B) + 2*live(v,B)**
    - use(v,B) is is the number of times v is used in B prior to any definition of v
    - live(v, B) is 1 if v is live on exit from B and is assigned a value in B
  - after computing weights, allocate registers to the "heaviest" values

Two Phase Solution Dynamic Programming Sethi & Ullman

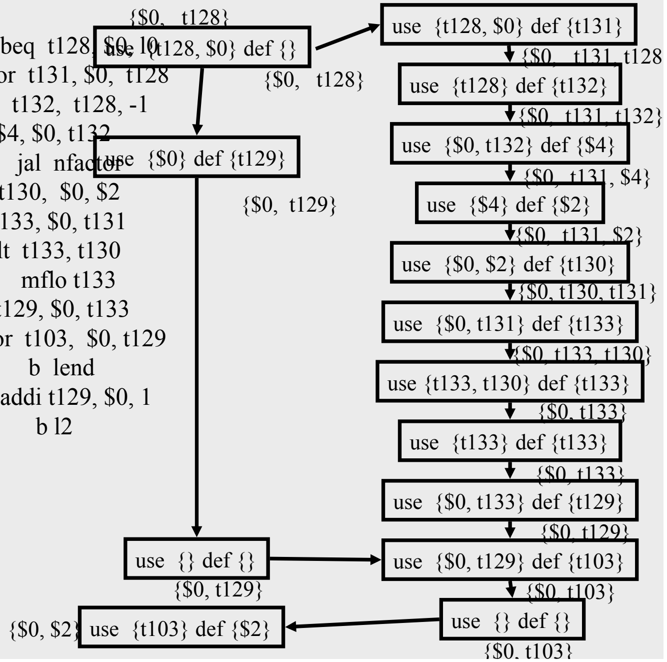
- Bottom-up (labeling)
  - Compute for every subtree
    - The minimal number of registers needed (weight)
- Top-Down
  - Generate the code using labeling by preferring "heavier" subtrees (larger labeling)

“Global” Register Allocation

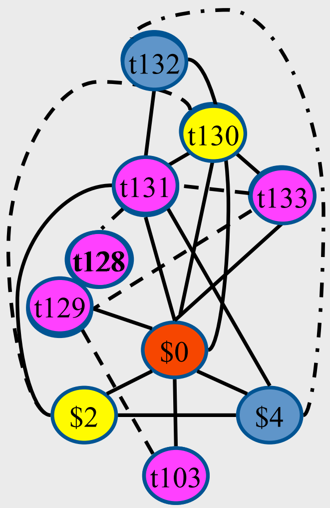
- Input:
  - Sequence of machine code instructions (assembly)
    - Unbounded number of temporary registers
- Output
  - Sequence of machine code instructions (assembly)
  - Machine registers
  - Some MOVE instructions removed
  - Missing prologue and epilogue

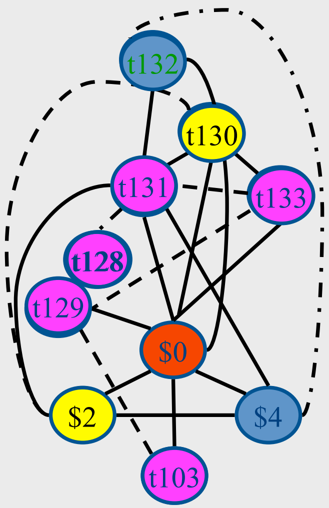



```
l3:     beq  t128, $0, l0
l1:     or  t131, $0,  t128
        addi  t132,  t128, -1
        or $4, $0, t132
        jal  nfactor
        or  t130,  $0, $2
        or t133, $0, t131
        mult  t133, t130
        mflo t133
        or  t129, $0, t133
l2:     or t103,  $0, t129
        b  lend
l0:     addi t129, $0, 1
        b l2
```




```
l3:     beq  t128, $0, l0 /* $0, t128 */
l1:     or  t131, $0,  t128 /* $0, t128, t131 */
        addi  t132,  t128, -1 /* $0, t131, t132 */
        or $4, $0, t132  /* $0, $4, t131 */
        jal  nfactor     /* $0, $2, t131 */
        or  t130,  $0, $2  /* $0, t130, t131 */
        or t133, $0, t131 /* $0, t130, t133 */
        mult  t133, t130 /* $0, t133 */
        mflo t133   /* $0, t133 */
        or  t129, $0, t133  /* $0, t129 */
l2:     or  t103,  $0, t129  /* $0, t103 */
        b  lend  /* $0, t103 */
l0:     addi t129, $0, 1    /* $0, t129 */
        b l2   /* $0, t129 */
```






```
l3:      beq  t128, $0, l0
l1:      or  t131, $0,  t128
         addi  t132,  t128, -1
         or $4, $0, t132
         jal  nfactor
         or  t130,  $0,  $2
         or t133, $0, t131
         mult  t133, t130
         mflo t133
         or  t129, $0, t133
l2:      or  t103,  $0, t129
         b  lend
l0:      addi t129, $0, 1
         b l2
```

Global Register Allocation Process

Repeat

Construct the interference graph

Color graph nodes with machine registers

Adjacent nodes are not colored by the same register

Spill a temporary into memory

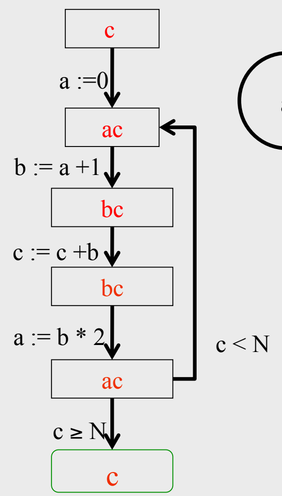
Until no more spill

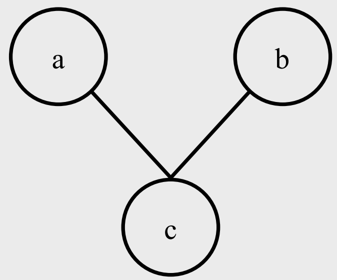
Constructing interference graphs (take 1)

- Compute liveness information at every statement
- Variables ‘a’ and ‘b’ interfere when there exists a control flow node n such that ‘a’, ‘b’ ∈ Lv[n]

A Simple Example

```
         /* c  */
L0:          a := 0
         /* ac */
L1:         b := a + 1
         /* bc */
            c := c + b
         /* bc */
            a := b * 2
         /* ac */
            if c < N goto L1
         /* c  */
            return c
```





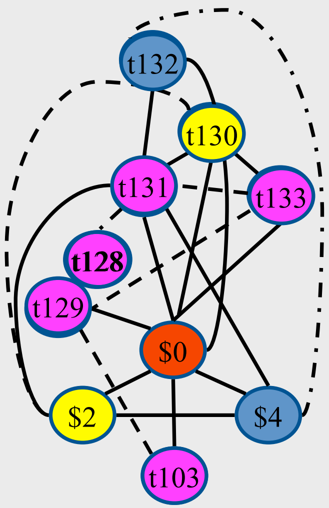
Constructing interference graphs (take 2)

- Compute liveness information at every statement
- Variables ‘a’ and ‘b’ interfere when there exists a control flow edge (m, n) with an assignment a := exp and ‘b’ ∈ Lv[n]

Constructing interference graphs (take 3)

- Compute liveness information at every statement
- Variables ‘a’ and ‘b’ interfere when there exists a control flow edge (m, n) with an assignment a := exp and ‘b’ ∈ Lv[n] and ‘b’ ≠ exp

```
l3:     beq  t128, $0, l0 /* $0, t128 */
l1:     or  t131, $0,  t128 /* $0, t128, t131 */
        addi  t132,  t128, -1 /* $0, t131, t132 */
        or $4, $0, t132  /* $0, $4, t131 */
        jal  nfactor     /* $0, $2, t131 */
        or  t130,  $0, $2  /* $0, t130, t131 */
        or t133, $0, t131 /* $0, t130, t133 */
        mult  t133, t130 /* $0, t133 */
        mflo t133   /* $0, t133 */
        or  t129, $0, t133  /* $0, t129 */
l2:     or  t103,  $0, t129  /* $0, t103 */
        b  lend  /* $0, t103 */
l0:     addi t129, $0, 1     /* $0, t129 */
        b l2    /* $0, t129 */
```



Challenges

- The Coloring problem is computationally hard
- The number of machine registers may be small
- Avoid too many MOVEs
- Handle “pre-colored” nodes

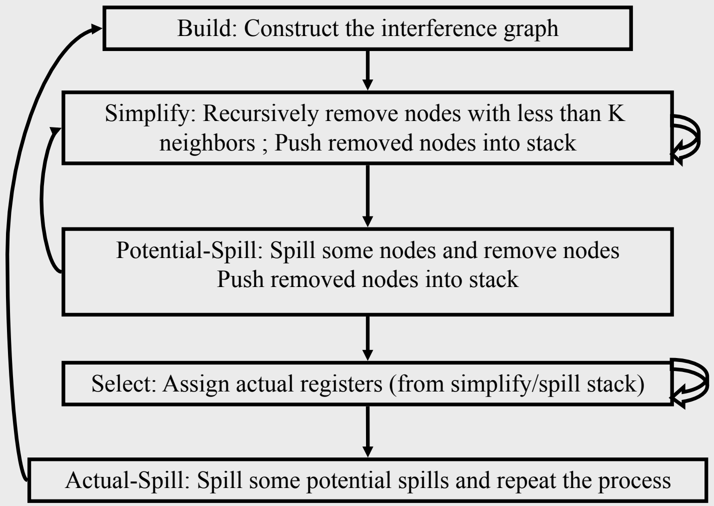
Theorem [Kempe 1879]

- Assume:
  - An undirected graph G(V, E)
  - A node v ∈V with less than K neighbors
  - G – {v} is K colorable
- Then, G is K colorable

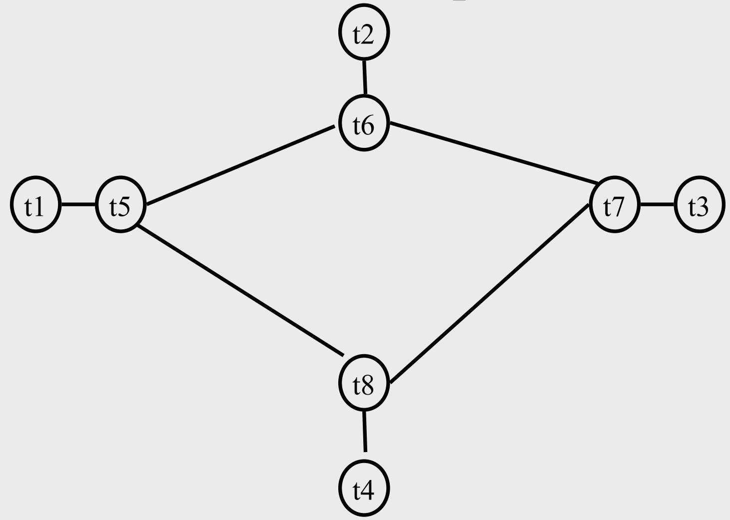
Coloring by Simplification [Kempe 1879]

- K
  - the number of machine registers
- G(V, E)
  - the interference graph
- Consider a node v ∈V with less than K neighbors:
  - Color G – v in K colors
  - Color v in a color different than its (colored) neighbors

Graph Coloring by Simplification



Artificial Example K=2



Coalescing

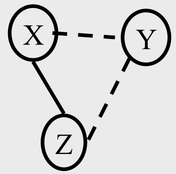
- MOVs can be removed if the source and the target share the same register
- The source and the target of the move can be merged into a single node (unifying the sets of neighbors)
- May require more registers
- Conservative Coalescing
  - Merge nodes only if the resulting node has fewer than K neighbors with degree ⩾ K (in the resulting graph)

Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

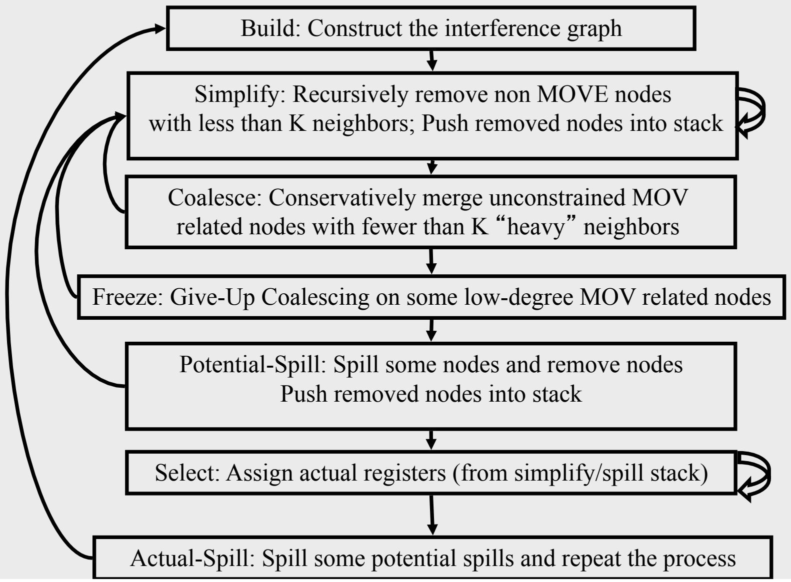
X ←Y /* X, Y, Z */

Y←Z



- Constrained MOVs are not coalesced

Graph Coloring with Coalescing



Spilling

- Many heuristics exist
  - Maximal degree
  - Live-ranges
  - Number of uses in loops
- The whole process need to be repeated after an actual spill

Pre-Colored Nodes

- Some registers in the intermediate language are pre-colored:
  - correspond to real registers (stack-pointer, frame-pointer, parameters,)
- Cannot be Simplified, Coalesced, or Spilled (infinite degree)
- Interfered with each other
- But normal temporaries can be coalesced into pre-colored registers
- Register allocation is completed when all the nodes are pre-colored

Caller-Save and Callee-Save Registers

- callee-save-registers (MIPS 16-23)
  - Saved by the callee when modified
  - Values are automatically preserved across calls
- caller-save-registers
  - Saved by the caller when needed
  - Values are not automatically preserved
- Usually the architecture defines caller-save and callee-save registers
  - Separate compilation
  - Interoperability between code produced by different compilers/languages
- But compilers can decide when to use calller/callee registers

Caller-Save vs. Callee-Save Registers

```
int foo(int a)      {
    int b=a+1;
      f1();
      g1(b);
    return(b+2);
   }
```

```
void bar (int y) {
    int x=y+1;
      f2(y);
      g2(2);
    }
```

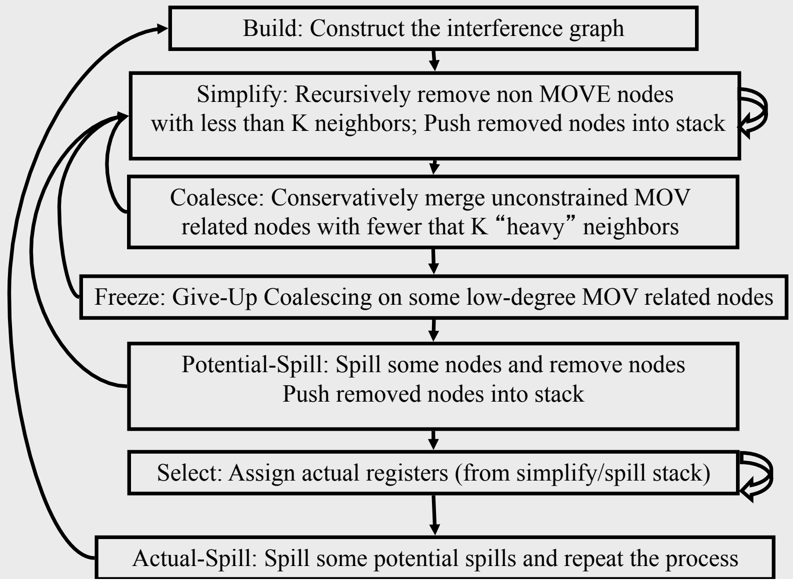
Saving Callee-Save Registers

enter: def(r_7)	enter: def(r_7)
	$t_{231} \leftarrow r_7$
…	…
	$r_7 \leftarrow t_{231}$
exit: use(r_7)	exit: use(r_7)

A Complete Example

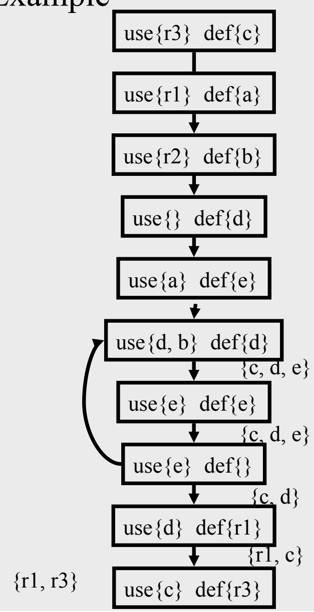
```
enter:
   c := r3        r1, r2   caller save
                  r3       callee-save
   a := r1
   b := r2
   d := 0
   e := a
loop:
   d := d+b
   e := e-1
   if e>0 goto loop
   r1 := d
   r3 := c
return  /* r1,r3 */
```

Graph Coloring with Coalescing



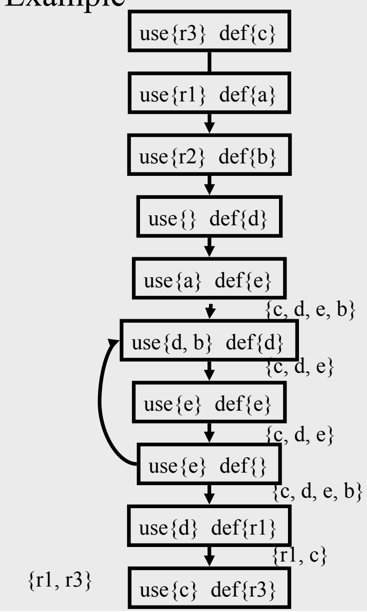
A Complete Example

```
enter:
   c := r3        r1, r2   caller save
                  r3       callee-save
   a := r1
   b := r2
   d := 0
   e := a
loop:
   d := d+b
   e := e-1
   if e>0 goto loop
   r1 := d
   r3 := c
return  /* r1,r3 */
```



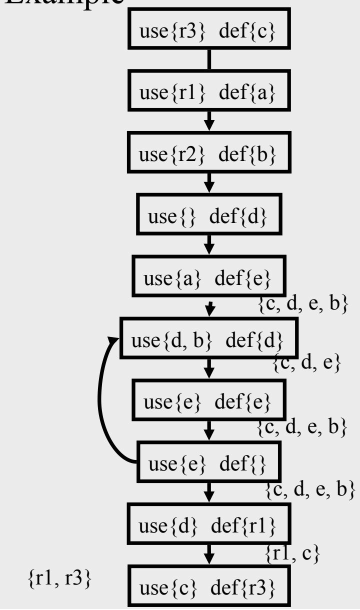
A Complete Example

```
enter:
   c := r3
   a := r1
   b := r2
   d := 0
   e := a
loop:
   d := d+b
   e := e-1
   if e>0 goto loop
   r1 := d
   r3 := c
return  /* r1,r3 */
```



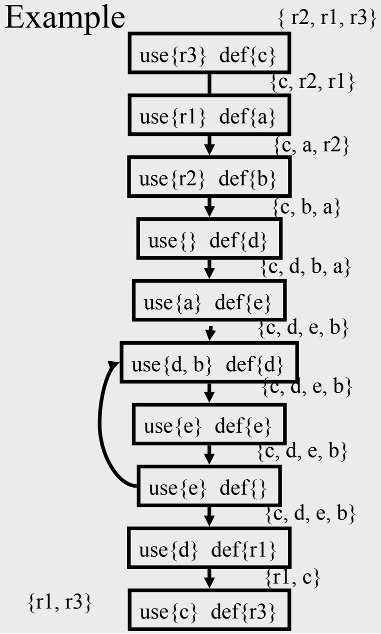
A Complete Example

```
enter:
    c := r3
    a := r1
    b := r2
    d := 0
    e := a
loop:
    d := d+b
    e := e-1
    if e>0 goto loop
    r1 := d
    r3 := c
return /* r1,r3 */
```



A Complete Example

```
enter:
    c := r3
    a := r1
    b := r2
    d := 0
    e := a
loop:
    d := d+b
    e := e-1
    if e>0 goto loop
    r1 := d
    r3 := c
return /* r1,r3 */
```

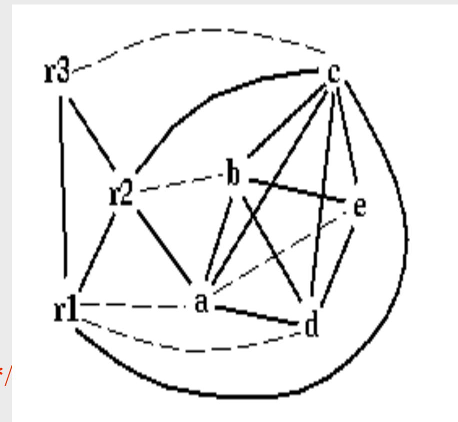


Live Variables Results

```
enter:
    c := r3
    a := r1
    b := r2
    d := 0
    e := a
loop:
    d := d+b
    e := e-1
    if e>0 goto loop
    r1 := d
    r3 := c
return /* r1,r3 */
```

```
enter:                  /* r2, r1, r3 */
    c := r3  /* c, r2, r1 */
    a := r1  /* a, c, r2 */
    b := r2  /* a, c, b */
    d := 0   /* a, c, b, d */
    e := a   /* e, c, b, d */
loop:
    d := d+b /* e, c, b, d */
    e := e-1 /* e, c, b, d */
    if e>0 goto loop /* c, d */
    r1 := d  /* r1, c */
    r3 := c  /* r1, r3 */
return /* r1, r3 */
```

```
enter:                  /* r2, r1, r3 */
    c := r3  /* c, r2, r1 */
    a := r1  /* a, c, r2 */
    b := r2  /* a, c, b */
    d := 0   /* a, c, b, d */
    e := a   /* e, c, b, d */
loop:
    d := d+b /* e, c, b, d */
    e := e-1 /* e, c, b, d */
    if e>0 goto loop /* c, d */
    r1 := d  /* r1, c */
    r3 := c  /* r1, r3 */
return /* r1,r3 */
```

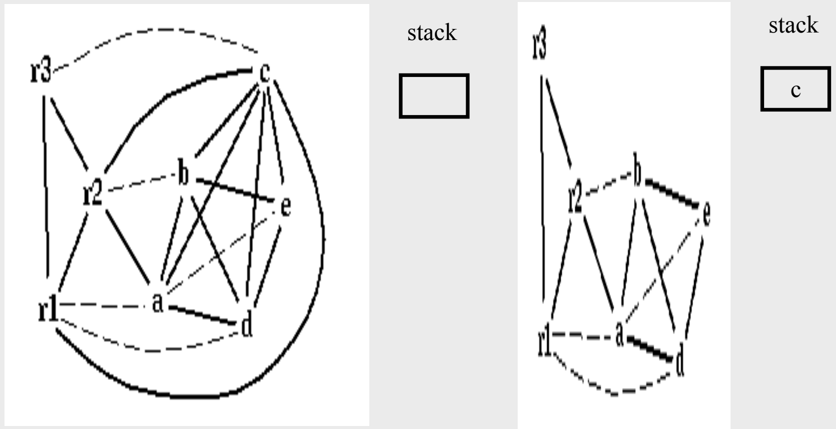


spill priority = (uo + 10 ui)/deg

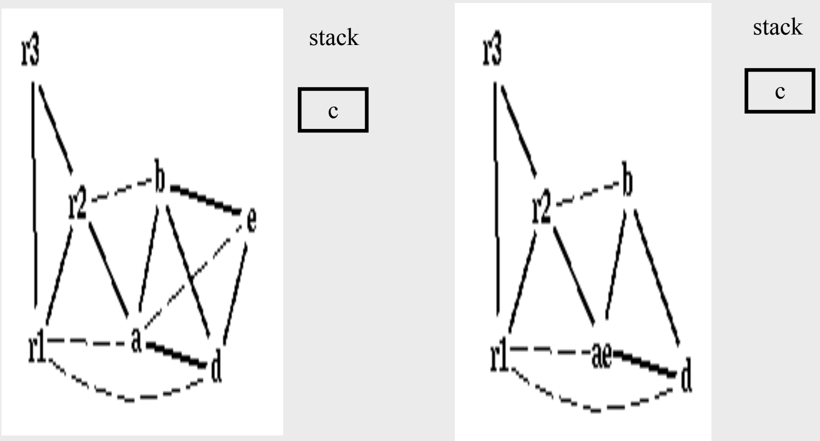
```
enter:                 /* r2, r1, r3 */
    c := r3  /* c, r2, r1 */
  a := r1  /* a, c, r2 */
    b := r2  /* a, c, b */
    d := 0   /* a, c, b, d */
    e := a   / * e, c, b, d */
        loop:
  d := d+b  /* e, c, b, d */
    e := e-1 /* e, c, b, d */
  if e>0 goto loop /* c, d */
     r1 := d  /* r1, c */
     r3 := c /* r1, r3 */
  return  /* r1,r3 */
```

	use+ def outside loop	use+ def within loop	deg	spill priority
a	2	0	4	0.5
b	1	1	4	2.75
c	2	0	6	0.33
d	2	2	4	5.5
e	1	3	3	10.3

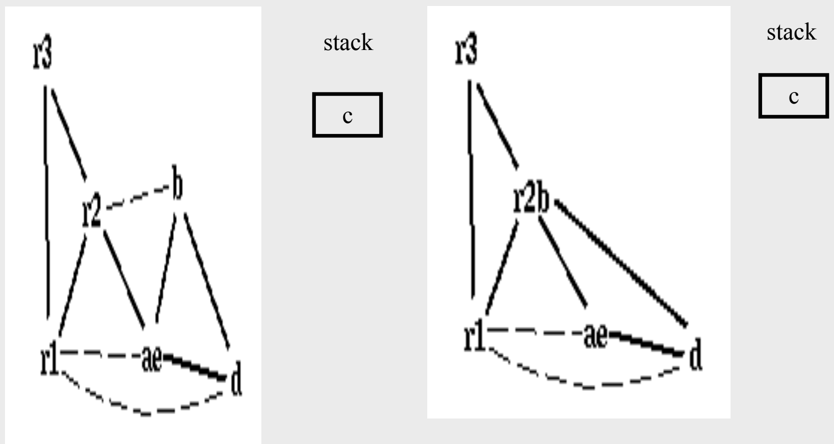
Spill C



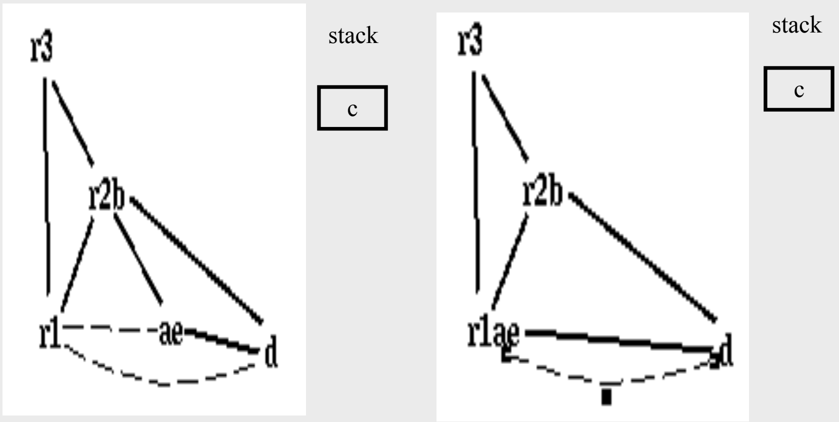
Coalescing *a+e*



Coalescing *b+r2*

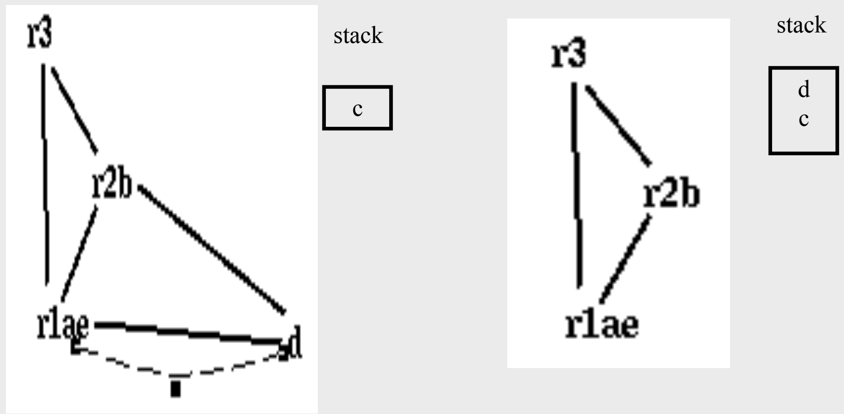


Coalescing *ae+r1*

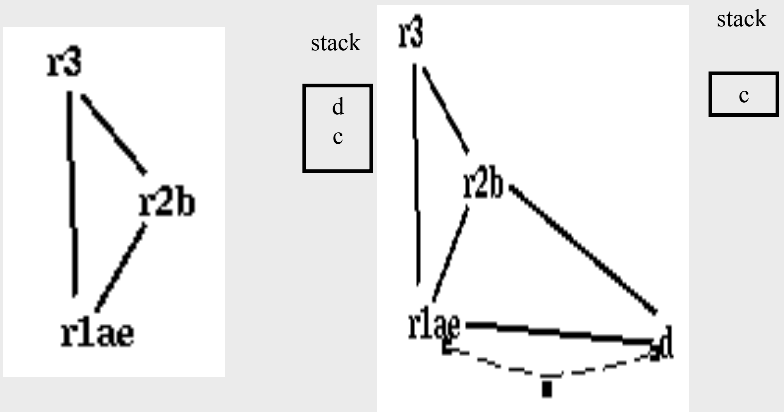


r1ae and d are constrained

Simplifying *d*

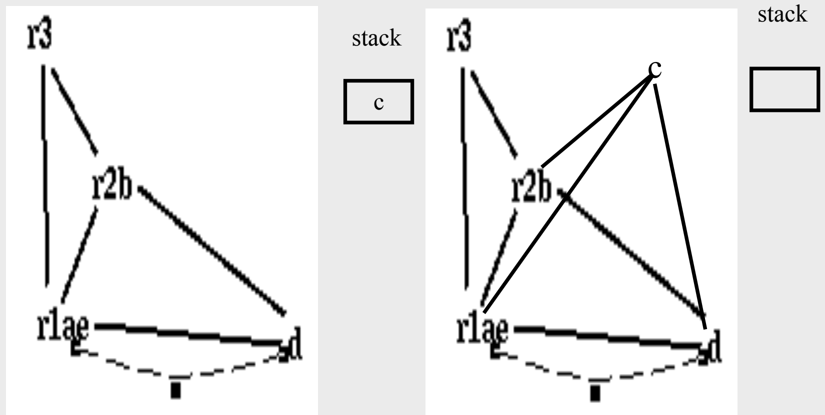


Pop *d*



d is assigned to r3

Pop *c*

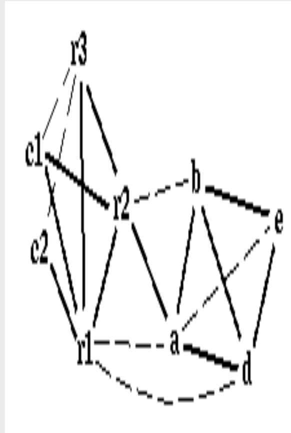


actual spill!

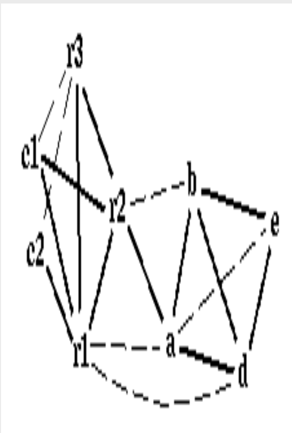
```
enter:            /* r2, r1, r3 */
    c := r3   /* c, r2, r1 */
  a := r1   /* a, c, r2 */
    b := r2   /* a, c, b */
   d := 0    /* a, c, b, d */
   e := a    / * e, c, b, d */
        loop:
  d := d+b   /* e, c, b, d */
   e := e-1  /* e, c, b, d */
  if e>0 goto loop /* c, d */
     r1 := d   /* r1, c */
     r3 := c  /* r1, r3 */
   return   /* r1,r3 */
```

```
enter:            /* r2, r1, r3 */
     c1 := r3   /* c1, r2, r1 */
     M[c_loc] := c1  /* r2 */
       a := r1   /* a, r2 */
         b := r2   /* a, b */
        d := 0    /* a, b, d */
        e := a    / * e, b, d */
            loop:
      d := d+b   /* e, b, d */
        e := e-1  /* e, b, d */
     if e>0 goto loop /* d */
          r1 := d   /* r1 */
  c2 := M[c_loc] /* r1, c2 */
        r3 := c2 /* r1, r3 */
     return   /* r1,r3 */
```

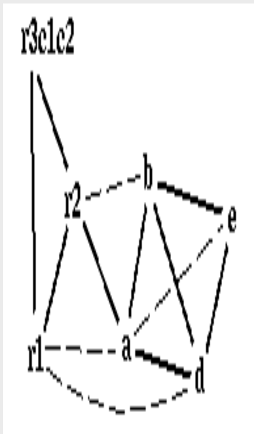
```
enter:            /* r2, r1, r3 */
     c1 := r3   /* c1, r2, r1 */
     M[c_loc] := c1  /* r2 */
       a := r1   /* a, r2 */
         b := r2   /* a, b */
        d := 0    /* a, b, d */
        e := a    / * e, b, d */
            loop:
      d := d+b   /* e, b, d */
        e := e-1  /* e, b, d */
     if e>0 goto loop /* d */
          r1 := d   /* r1 */
  c2 := M[c_loc] /* r1, c2 */
        r3 := c2 /* r1, r3 */
     return   /* r1,r3 */
```



Coalescing c1+r3; c2+c1r3

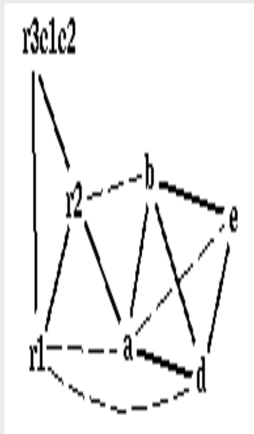


stack

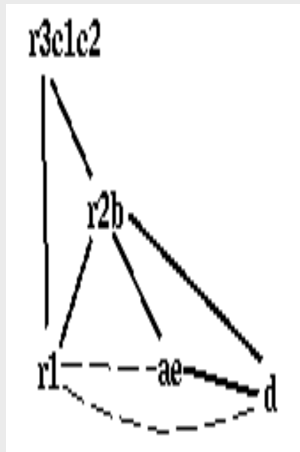


stack

Coalescing a+e; b+r2

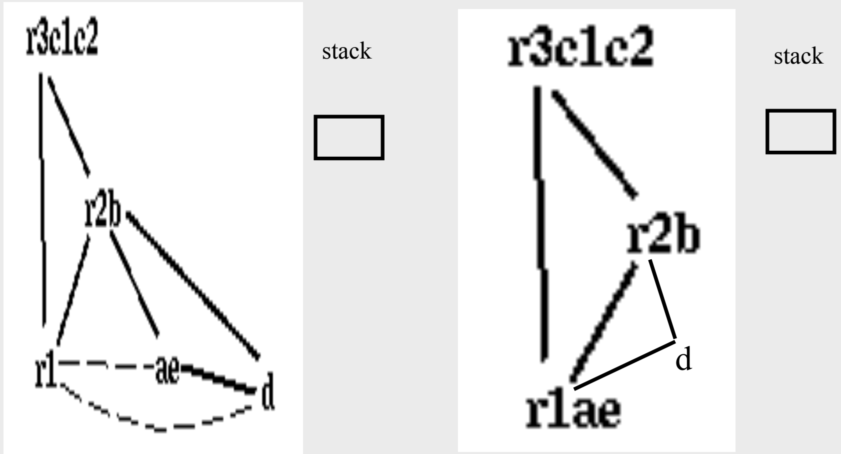


stack



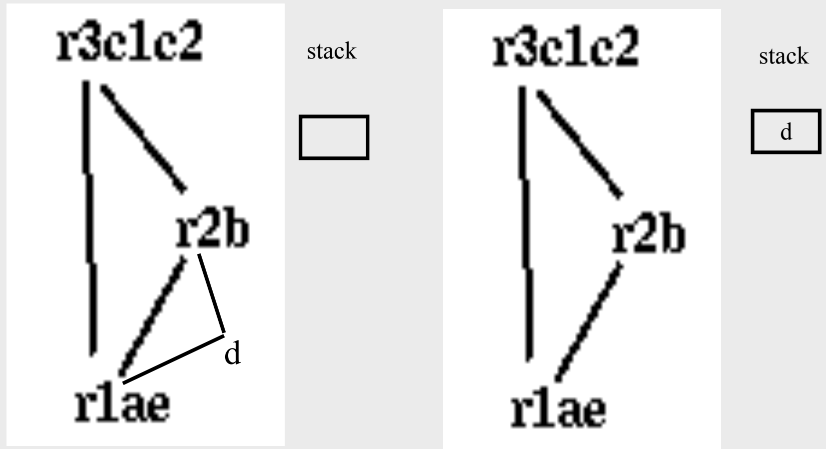
stack

Coalescing ae+r1

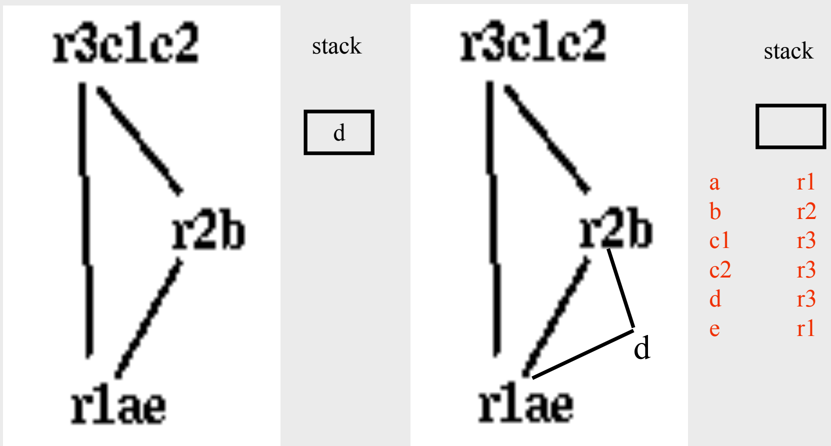


r1ae and d are constrained

Simplify d



Pop d




```
enter:
    c1 := r3
M[c_loc] := c1
    a := r1
    b := r2
    d := 0
    e := a
loop:
    d := d+b
    e := e-1
if e>0 goto loop
    r1 := d
c2 := M[c_loc]
    r3 := c2
return  /* r1,r3 */
```

a	r1
b	r2
c1	r3
c2	r3
d	r3
e	r1

```
enter:
    r3 := r3
M[c_loc] := r3
    r1 := r1
    r2 := r2
    r3 := 0
    r1 := r1
loop:
    r3 := r3+r2
    r1 := r1-1
if r1>0 goto loop
    r1 := r3
r3 := M[c_loc]
    r3 := r3
return  /* r1,r3 */
```

```
enter:
    r3 := r3
    M[c_loc] := r3
    r1 := r1
    r2 := r2
    r3 := 0
    r1 := r1
loop:
    r3 := r3+r2
    r1 := r1-1
    if r1>0 goto loop
    r1 := r3
    r3 := M[c_loc]
    r3 := r3
    return /* r1,r3 */
```

```
enter:
    M[c_loc] := r3
    r3 := 0
loop:
    r3 := r3+r2
    r1 := r1-1
    if r1>0 goto loop
    r1 := r3
    r3 := M[c_loc]
    return /* r1,r3 */
```

```
main:   addiu  $sp,$sp, -K1
L4:     sw     $2,0+K1($sp)
        or     $25,$0,$31
        sw     $25,-4+K1($sp)
        addiu  $25,$sp,0+K1
        or     $2,$0,$25
        addi   $25,$0,10
        or     $4,$0,$25
        jal    nfactor
        lw     $25,-4+K1
        or     $31,$0,$25
        b      L3
L3:     addiu  $sp,$sp,K1
        j      $31
```

```
nfactor: addiu $sp,$sp,-K2
L6:     sw     $2,0+K2($sp)
        or     $25,$0,$4
        or     $24,$0,$31
        sw     $24,-4+K2($sp)
        sw     $30,-8+K2($sp)
        beq    $25,$0,L0
L1:     or     $30,$0,$25
        lw     $24,0+K2
        or     $2,$0,$24
        addi   $25,$25,-1
        or     $4,$0,$25
        jal    nfactor
```

```
        or     $25,$0,$2
        mult   $30,$25
        mflo   $30
L2:     or     $2,$0,$30
        lw     $30,-4+K2($sp
        or     $31,$0,$30
        lw     $30,-8+K2($sp
        b      L5
L0:     addi   $30,$0,1
        b      L2
L5:     addiu  $sp,$sp,K
        j      $31
```

Interprocedural Allocation

- Allocate registers to multiple procedures
- Potential saving
  - caller/callee save registers
  - Parameter passing
  - Return values
- But may increase compilation cost
- Function inline can help

Summary

- Two Register Allocation Methods
  - Local of every IR tree
    - Simultaneous instruction selection and register allocation
    - Optimal (under certain conditions)
  - Global of every function
    - Applied after instruction selection
    - Performs well for machines with many registers
    - Can handle instruction level parallelism
- Missing
  - Interprocedural allocation

Challenges in register allocation

- Registers are scarce
  - Often substantially more IR variables than registers
  - Need to find a way to reuse registers whenever possible
- Registers are complicated
  - x86: Each register made of several smaller registers; can't use a register and its constituent registers at the same time
  - x86: Certain instructions must store their results in specific registers; can't store values there if you want to use those instructions
  - MIPS: Some registers reserved for the assembler or operating system
  - Most architectures: Some registers must be preserved across function calls

The End

The End