

# Code Generation

## Part II

### Chapter 9

COP5621 Compiler Construction  
Copyright Robert van Engelen, Florida State University, 2005

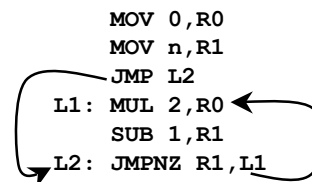
## Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges
- A flow graph can be defined at the intermediate code level or target code level

```

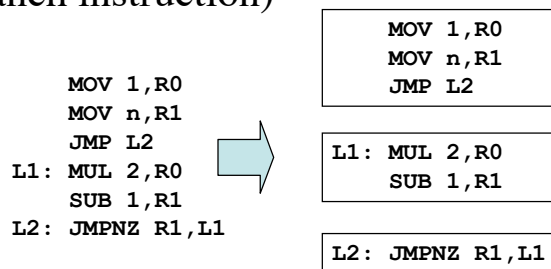
MOV 1,R0
MOV n,R1
JMP L2
L1: MUL 2,R0
SUB 1,R1
L2: JMPNZ R1,L1

```



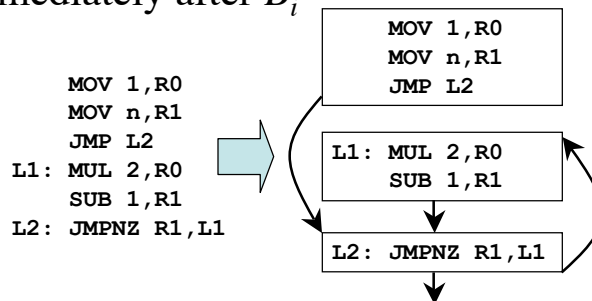
## Basic Blocks

- A *basic block* is a sequence of consecutive instructions with exactly one entry point and one exit point (with natural flow or a branch instruction)



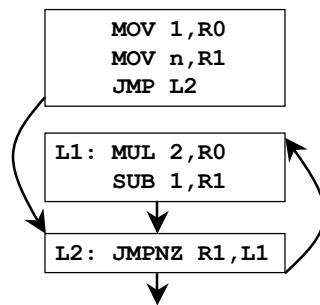
## Basic Blocks and Control Flow Graphs

- A *control flow graph* (CFG) is a directed graph with basic blocks  $B_i$  as vertices and with edges  $B_i \rightarrow B_j$  iff  $B_j$  can be executed immediately after  $B_i$



## Successor and Predecessor Blocks

- Suppose the CFG has an edge  $B_1 \rightarrow B_2$ 
  - Basic block  $B_1$  is a *predecessor* of  $B_2$
  - Basic block  $B_2$  is a *successor* of  $B_1$



## Partition Algorithm for Basic Blocks

*Input:* A sequence of three-address statements

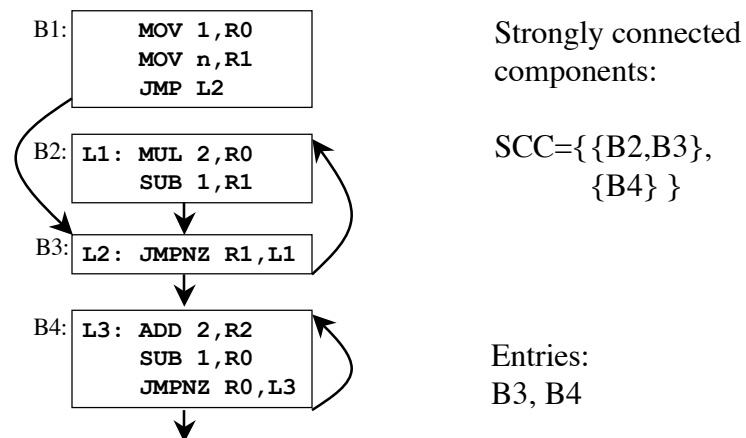
*Output:* A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements of basic blocks
  - a) The first statement is the leader
  - b) Any statement that is the target of a goto is a leader
  - c) Any statement that immediately follows a goto is a leader
2. 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

## Loops

- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry

## Loops (Example)



## Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

```

b := 0
t1 := a + b
t2 := c * t1
a := t2
  
```



```

a := c*a
b := 0
  
```

```

a := c * a
b := 0
  
```



```

a := c*a
b := 0
  
```

Blocks are equivalent, assuming **t1** and **t2** are *dead*: no longer used (no longer *live*)


## Transformations on Basic Blocks

- A *code-improving transformation* is a code optimization to improve speed or reduce code size
- *Global transformations* are performed across basic blocks
- *Local transformations* are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

## Common-Subexpression Elimination


- Remove redundant computations

a := b + c
b := a - d
c := b + c
d := a - d



a := b + c
b := a - d
c := b + c
d := b

t1 := b * c
t2 := a - t1
t3 := b * c
t4 := t2 + t3




t1 := b * c
t2 := a - t1
t4 := t2 + t1

## Dead Code Elimination

- Remove unused statements


b := a + 1
a := b + c
...



b := a + 1
...

Assuming **a** is *dead* (not used)

if true goto L2
-----------------

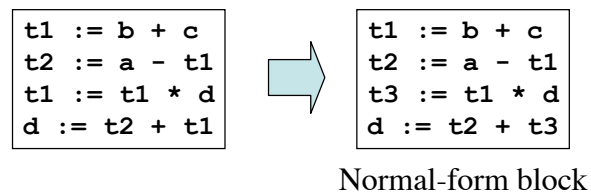


b := x + y
...

Remove unreachable code

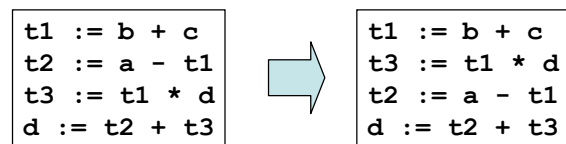
## Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed



## Interchange of Statements

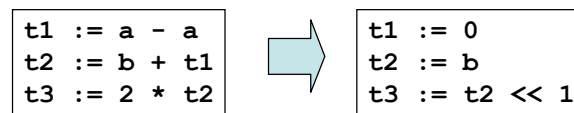
- Independent statements can be reordered



Note that normal-form blocks permit all statement interchanges that are possible

## Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms



## Next-Use

- Next-use information is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

$i: x := y \text{ op } z$

- Add liveness/next-use info on  $x$ ,  $y$ , and  $z$  to statement  $i$
- Set  $x$  to “not live” and “no next use”
- Set  $y$  and  $z$  to “live” and the next uses of  $y$  and  $z$  to  $i$

## Next-Use (Step 1)

*i*: **a := b + c**

*j*: **t := a + b** [ *live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,  
*nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none ]

Attach current live/next-use information  
 Because info is empty, assume variables are live  
 (Data flow analysis Ch.10 can provide accurate information)

## Next-Use (Step 2)

*i*: **a := b + c**

<i>live</i> ( <b>a</b> ) = true	<i>nextuse</i> ( <b>a</b> ) = <i>j</i>
<i>live</i> ( <b>b</b> ) = true	<i>nextuse</i> ( <b>b</b> ) = <i>j</i>
<i>live</i> ( <b>t</b> ) = false	<i>nextuse</i> ( <b>t</b> ) = none

*j*: **t := a + b** [ *live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,  
*nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none ]

Compute live/next-use information at *j*

## Next-Use (Step 3)

$i: \mathbf{a} := \mathbf{b} + \mathbf{c}$  [  $live(\mathbf{a}) = true, live(\mathbf{b}) = true, live(\mathbf{c}) = false,$   
 $nextuse(\mathbf{a}) = j, nextuse(\mathbf{b}) = j, nextuse(\mathbf{c}) = none$  ]

$j: \mathbf{t} := \mathbf{a} + \mathbf{b}$  [  $live(\mathbf{a}) = true, live(\mathbf{b}) = true, live(\mathbf{t}) = true,$   
 $nextuse(\mathbf{a}) = none, nextuse(\mathbf{b}) = none, nextuse(\mathbf{t}) = none$  ]

Attach current live/next-use information to  $i$

## Next-Use (Step 4)

$live(\mathbf{a}) = false$	$nextuse(\mathbf{a}) = none$
$live(\mathbf{b}) = true$	$nextuse(\mathbf{b}) = i$
$live(\mathbf{c}) = true$	$nextuse(\mathbf{c}) = i$
$live(\mathbf{t}) = false$	$nextuse(\mathbf{t}) = none$

$i: \mathbf{a} := \mathbf{b} + \mathbf{c}$  [  $live(\mathbf{a}) = true, live(\mathbf{b}) = true, live(\mathbf{c}) = false,$   
 $nextuse(\mathbf{a}) = j, nextuse(\mathbf{b}) = j, nextuse(\mathbf{c}) = none$  ]

$j: \mathbf{t} := \mathbf{a} + \mathbf{b}$  [  $live(\mathbf{a}) = false, live(\mathbf{b}) = false, live(\mathbf{t}) = false,$   
 $nextuse(\mathbf{a}) = none, nextuse(\mathbf{b}) = none, nextuse(\mathbf{t}) = none$  ]

Compute live/next-use information  $i$

## A Code Generator

- Generates target code for a sequence of three-address statements using next-use information
- Uses new function *getreg* to assign registers to variables
- Computed results are kept in registers as long as possible, which means:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Checks if operands to three-address code are available in registers

## The Code Generation Algorithm

- For each statement  $x := y \text{ op } z$ 
  1. Set location  $L = \text{getreg}(y, z)$
  2. If  $y \notin L$  then generate
    - MOV**  $y', L$
    - where  $y'$  denotes one of the locations where the value of  $y$  is available (choose register if possible)
  3. Generate
    - OP**  $z', L$
    - where  $z'$  is one of the locations of  $z$ ;
    - Update register/address descriptor of  $x$  to include  $L$
  4. If  $y$  and/or  $z$  has no next use and is stored in register, update register descriptors to remove  $y$  and/or  $z$

## Register and Address Descriptors

- A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

**MOV a, R0**            “R0 contains a”

- An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

**MOV a, R0**  
**MOV R0, R1**        “a in R0 and R1”

## The *getreg* Algorithm

- To compute *getreg*(y,z)
  1. If y is stored in a register *R* and *R* only holds the value y, and y has no next use, then return *R*;  
Update address descriptor: value y no longer in *R*
  2. Else, return a new empty register if available
  3. Else, find an occupied register *R*;  
Store contents (register spill) by generating  
**MOV R, M**  
for every *M* in address descriptor of y;  
Return register *R*
  4. Return a memory location

## Code Generation Example

<i>Statements</i>	<i>Code Generated</i>	<i>Register Descriptor</i>	<i>Address Descriptor</i>
<code>t := a - b</code>	<code>MOV a,R0 SUB b,R0</code>	Registers empty R0 contains t	t in R0
<code>u := a - c</code>	<code>MOV a,R1 SUB c,R1</code>	R0 contains t R1 contains u	t in R0 u in R1
<code>v := t + u</code>	<code>ADD R1,R0</code>	R0 contains v R1 contains u	u in R1 v in R0
<code>d := v + u</code>	<code>ADD R1,R0 MOV R0,d</code>	R0 contains d	d in R0 d in R0 and memory

## Register Allocation and Assignment

- The *getreg* algorithm is simple but sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block
- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings

## Allocating Registers in Loops

- Suppose loading a variable  $x$  has a cost of 2
- Suppose storing a variable  $x$  has a cost of 2
- Benefit of allocating a register to a variable  $x$  within a loop  $L$  is
 
$$\sum_{B \in L} ( use(x, B) + 2 live(x, B) )$$
 where  $use(x, B)$  is the number of times  $x$  is used in  $B$  and  $live(x, B) = \text{true}$  if  $x$  is live on exit from  $B$

## Global Register Allocation Using Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (spilled) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*)
- Find a  $k$ -coloring for the graph, with  $k$  the number of registers

## Graph Coloring Example

## Peephole Optimization

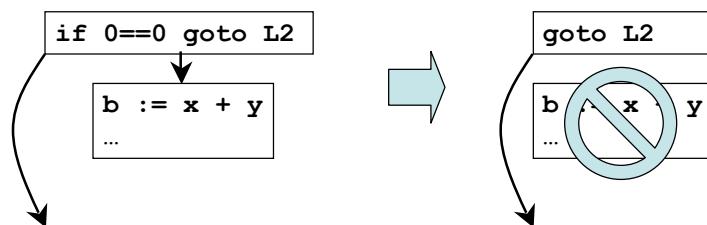
- Examines a short sequence of target instructions in a window (peephole) and replaces the instructions by a faster and/or shorter sequence when possible
- Applied to intermediate code or target code
- Typical optimizations:
  - Redundant instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

## Peephole Opt: Eliminating Redundant Loads and Stores

- Consider
  - `MOV R0, a`
  - `MOV a, R0`
- The second instruction can be deleted, but only if it is not labeled with a target label
  - Peephole represents sequence of instructions with at most one entry point
- The first instruction can also be deleted if  $live(\mathbf{a})=false$

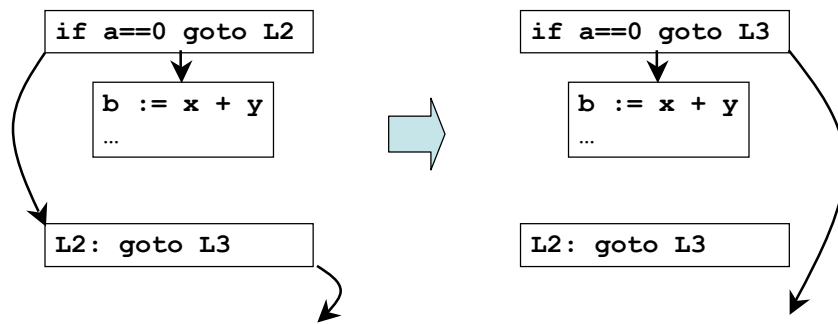
## Peephole Optimization: Deleting Unreachable Code

- Unlabeled blocks can be removed



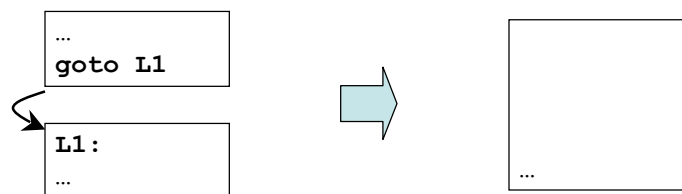
## Peephole Optimization: Branch Chaining

- Shorten chain of branches by modifying target labels



## Peephole Optimization: Other Flow-of-Control Optimizations

- Remove redundant jumps



## Other Peephole Optimizations

- *Reduction in strength*: replace expensive arithmetic operations with cheaper ones

... a := x ^ 2 b := y / 8	➔	... a := x * x b := y >> 3
---------------------------------	---	----------------------------------

- Utilize machine idioms

... a := a + 1	➔	... inc a
-------------------	---	--------------

- Algebraic simplifications

... a := a + 0 b := b * 1	➔	...
---------------------------------	---	-----