

# 6.035

Fall 2006

## Lecture 8: Unoptimized Code Generation (cont.)

# Question:

- What is the difference between caller-saved (t0-t9) and callee-saved(s0-s7) registers?
- What callee saved registers should be saved and where?
- What caller saved registers should be saved and where?
- What should your codegen pass do?

Function foo

```
???  
t0 = ...  
t1 = ...  
t2 = ...  
s0 = ...  
s1 = ...  
s2 = ...  
... = t0 + t1 + t2  
... = s0 + s1 + s2  
???  
call bar  
???  
t0 = t1*2  
s0 = s1*2  
v0 = t0+s0  
???  
return
```

# Question:

- What is the difference between caller-saved (t0-t9) and callee-saved(s0-s7) registers?
- What callee saved registers should be saved and where?
- What caller saved registers should be saved and where?
- What should your codegen pass do?

Function foo

```
save(s0, s1, s2)
t0 = ...
t1 = ...
t2 = ...
s0 = ...
s1 = ...
s2 = ...
... = t0 + t1 + t2
... = s0 + s1 + s2
save(t1)
call bar
restore(t1)
t0 = t1*2
s0 = s1*2
v0 = t0+s0
restore(s0, s1, s2)
return
```

# So far we covered..

## CODE

Procedures

Control Flow

Statements

Data Access

## DATA

Global Static Variables

Global Dynamic Data

Local Variables

Temporaries

Parameter Passing

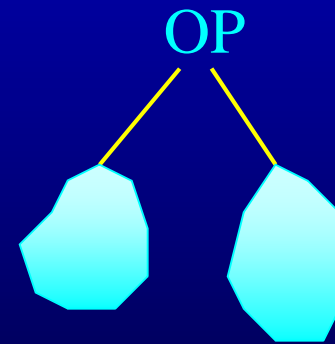
Read-only Data

# Statements

- Statements have expression trees
- Statements are either
  - An expression tree that get evaluated to set the condition codes
  - Value returning at the top of the expression tree assigned to a variable
- How do you map expression trees to the machines?
  - How to arrange the evaluation order?
  - Where to keep the intermediate values?
- Two approaches
  - Stack Model
  - Flat List Model

# Evaluating expression trees

- Stack model
  - Eval left-sub-tree  
Put the results on the stack
  - Eval right-sub-tree  
Put the results on the stack
  - Get top two values from the stack  
perform the operation OP  
put the results on the stack
- Very inefficient!



# Evaluating expression trees

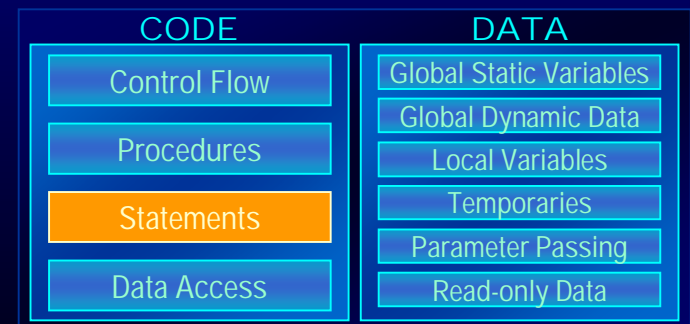
- Flat List Model

- Left to Right Depth-First Traversal of the expression tree

- Generate the expression on the return
- Allocate temporaries for intermediates (all the nodes of the tree)
  - New temporary for each intermediate
  - All the temporaries on the stack

- Each expression is a single 3-addr op

- $x = y \text{ op } z$
- Code generation for the 3-addr expression
  - Load  $y$  into register  $\%r10$
  - Load  $z$  into register  $\%r11$
  - Perform  $\text{op } \%r10, \%r11$
  - Store  $\%r11$  to  $x$



# Issues in Lowering Statements

- Map intermediates to registers?
  - registers are limited
    - when the tree is large, registers may be insufficient  $\Rightarrow$  allocate space in the stack
- No machine instruction is available
  - May need to expand the intermediate operation into multiple machine ops.
- Very inefficient
  - too many copies, add by zero etc.
  - don't worry, we'll take care of them in the optimization passes
  - keep the code generator very simple

# Two Approaches

- Template Matching Approach
  - Peephole Optimization
- Algorithmic Approach

# Generation of control flow: Template Matching Approach

- Flatten the control structure
  - use a template
- Put unique labels for control join points
- Now generate the appropriate code

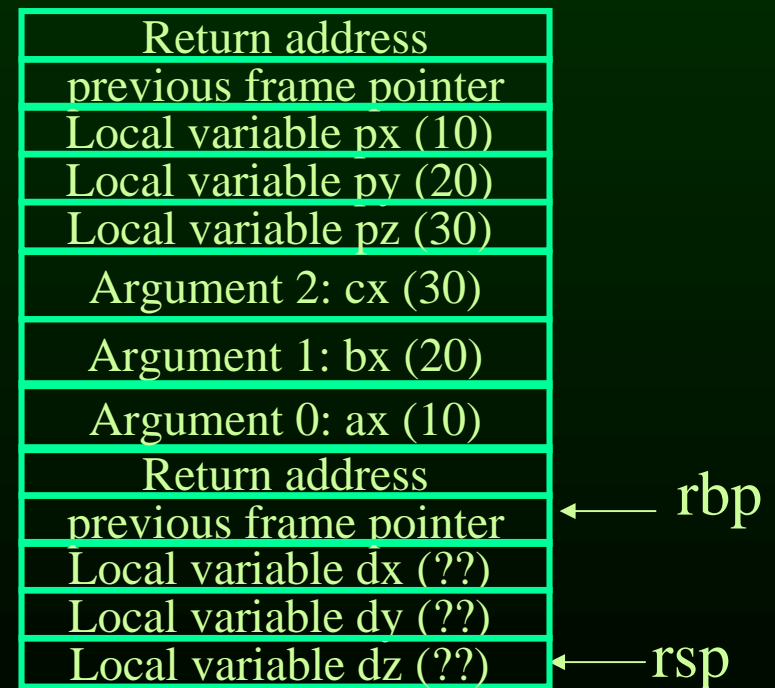
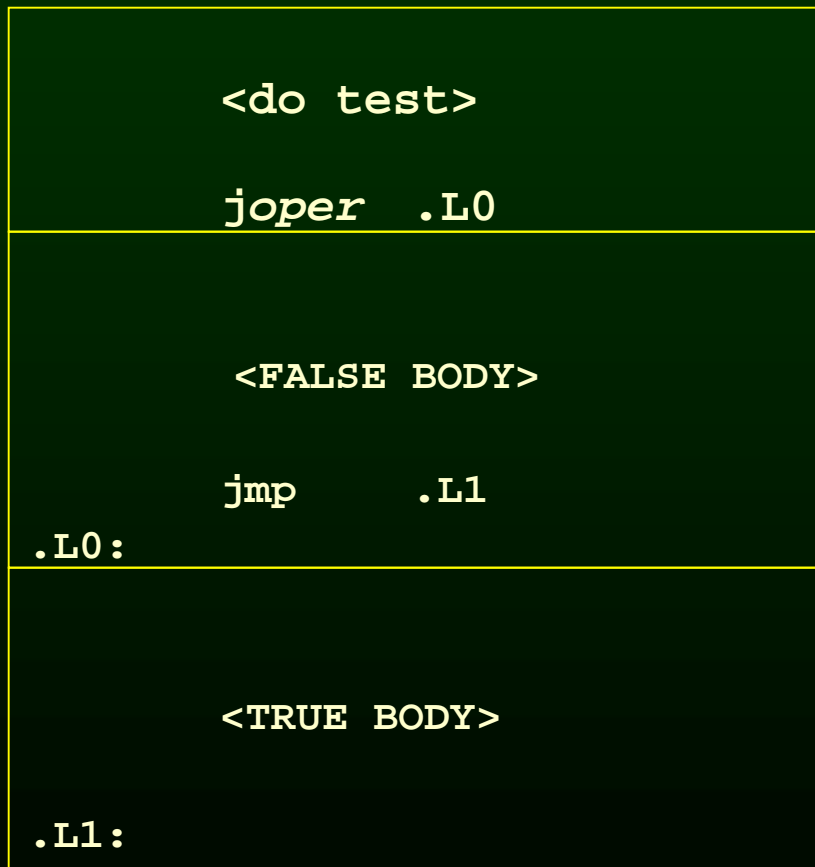
# Template for conditionals

```
if (test)
    true_body
else
    false_body
```

```
        <do the test>
        joper lab_true
        <false_body>
        jmp    lab_end
lab_true:
        <true_body>
lab_end:
```

# Example Program

```
if(ax > bx)
    dx = ax - bx;
else
    dx = bx - ax;
```



# Example Program

```
if(ax > bx)
    dx = ax - bx;
else
    dx = bx - ax;
```

```
movq    16(%rbp), %r10
movq    24(%rbp), %r11
cmpq    %r10, %r11
jg      .L0
```

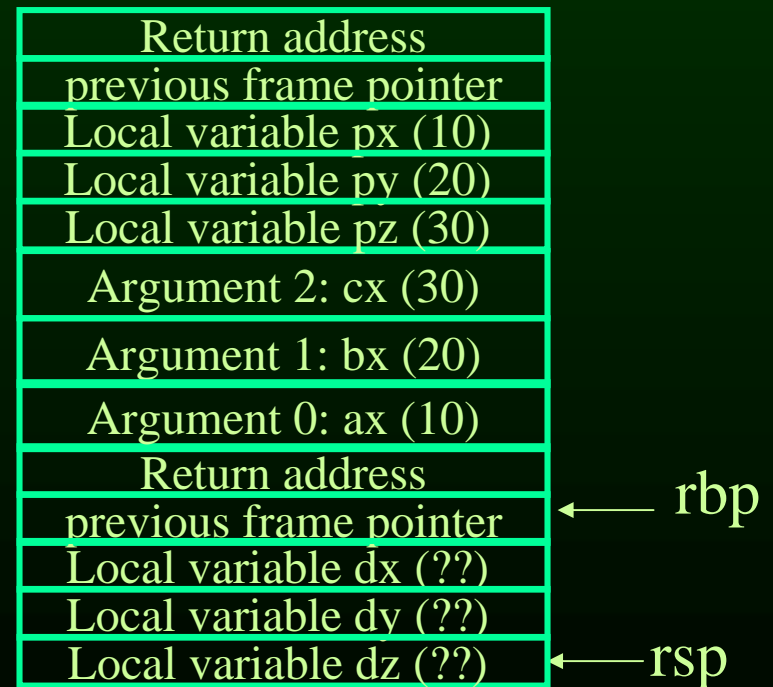
<FALSE BODY>

```
jmp     .L1
```

.L0:

<TRUE BODY>

.L1:



# Example Program

```

if(ax > bx)
    dx = ax - bx;
else
    dx = bx - ax;

```

|      |             |                |
|------|-------------|----------------|
|      | movq        | 16(%rbp), %r10 |
|      | movq        | 24(%rbp), %r11 |
|      | cmpq        | %r10, %r11     |
|      | jg          | .L0            |
|      | movq        | 24(%rbp), %r10 |
|      | movq        | 16(%rbp), %r11 |
|      | subq        | %r10, %r11     |
|      | movq        | %r11, -8(%rbp) |
|      | jmp         | .L1            |
| .L0: |             |                |
|      | <TRUE BODY> |                |
| .L1: |             |                |

|                              |
|------------------------------|
| Return address               |
| previous frame pointer       |
| Local variable px (10)       |
| Local variable py (20)       |
| Local variable pz (30)       |
| Argument 2: cx (30)          |
| Argument 1: bx (20)          |
| Argument 0: ax (10)          |
| Return address               |
| previous frame pointer ← rbp |
| Local variable dx (??)       |
| Local variable dy (??)       |
| Local variable dz (??) ← rsp |

# Example Program

```
if(ax > bx)
    dx = ax - bx;
else
    dx = bx - ax;
```

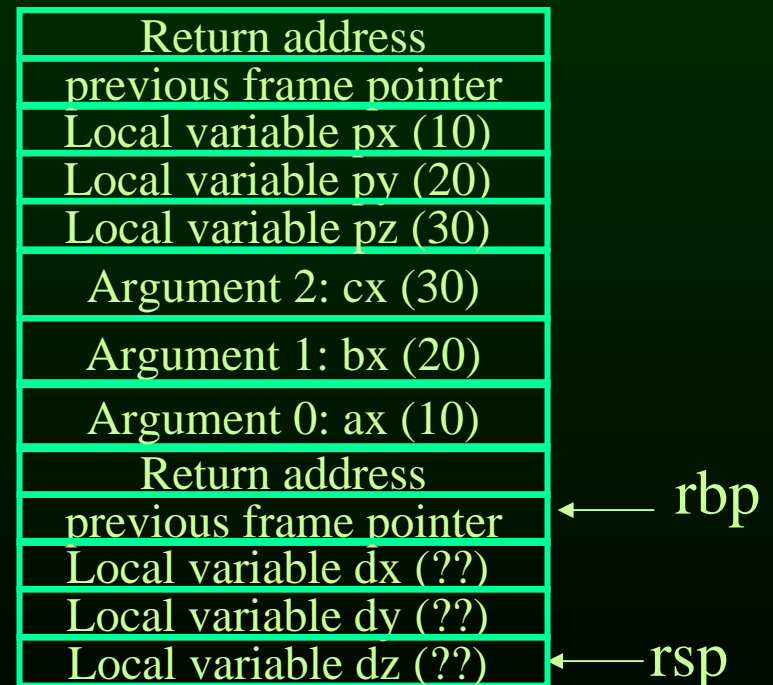
```
movq    16(%rbp), %r10
movq    24(%rbp), %r11
cmpq    %r10, %r11
jg      .L0
```

```
movq    24(%rbp), %r10
movq    16(%rbp), %r11
subq    %r10, %r11
movq    %r11, -8(%rbp)
jmp     .L1
```

.L0:

```
movq    16(%rbp), %r10
movq    24(%rbp), %r11
subq    %r10, %r11
movq    %r11, -8(%rbp)
```

.L1:



# Template for while loops

```
while (test)  
    body
```

# Template for while loops

```
while (test)
  body

lab_cont:
  <do the test>
  joper lab_body
  j      lab_end
lab_body:
  <body>
  j      lab_cont
lab_end:
```

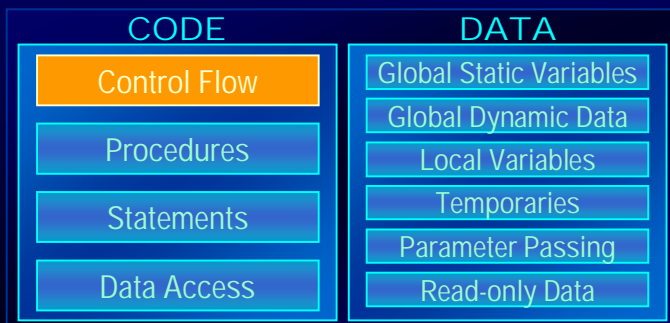
# Template for while loops

```
while (test)
    body
```

```
lab_cont:
    <do the test>
    joper lab_body
    jmp    lab_end
lab_body:
    <body>
    jmp    lab_cont
lab_end:
```

- An optimized template

```
lab_cont:
    <do the test>
    joper lab_end
    <body>
    jmp    lab_cont
lab_end:
```



# Question:

- What is the template for?

`do`

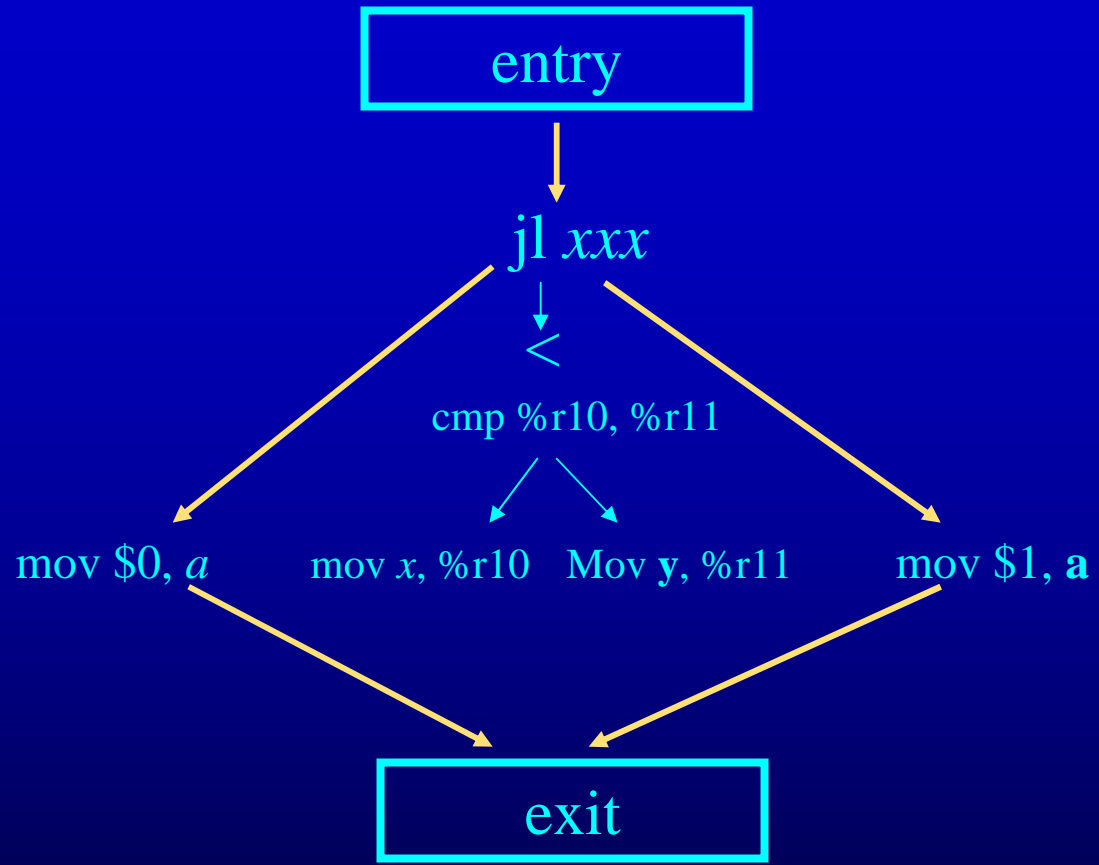
`body`

`while (test)`

# Control Flow Graph (CFG)

- Starting point: high level intermediate format, symbol tables
- Target: CFG
  - CFG Nodes are Instruction Nodes
  - CFG Edges Represent Flow of Control
  - Forks At Conditional Jump Instructions
  - Merges When Flow of Control Can Reach A Point Multiple Ways
  - Entry and Exit Nodes

```
if (x < y) {  
    a = 0;  
} else {  
    a = 1;  
}
```



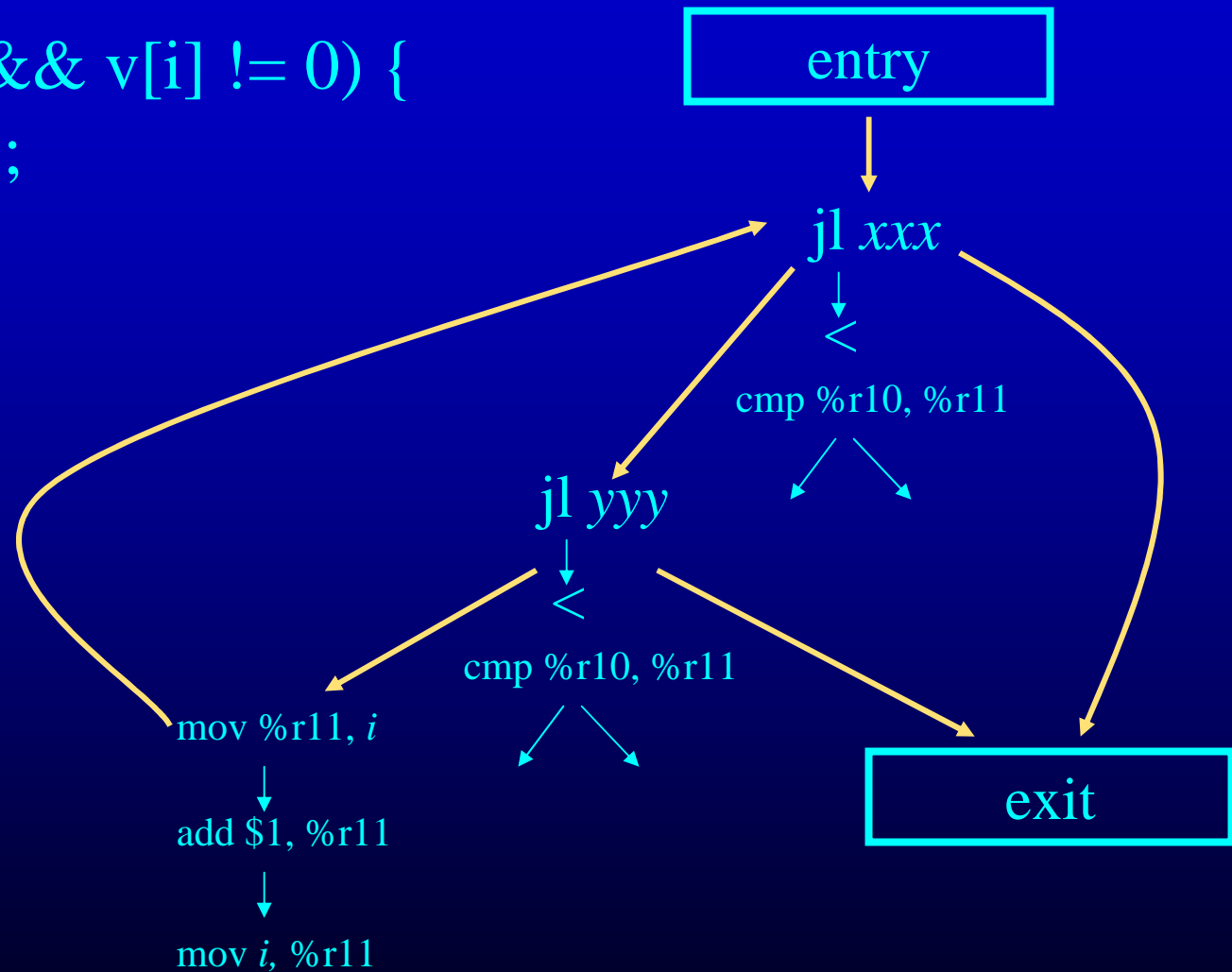
## Pattern for if then else

# Short-Circuit Conditionals

- In program, conditionals have a condition written as a boolean expression  
 $((i < n) \ \&\& \ (v[i] \neq 0)) \ || \ i > k$
- Semantics say should execute only as much as required to determine condition
  - Evaluate  $(v[i] \neq 0)$  only if  $(i < n)$  is true
  - Evaluate  $i > k$  only if  $((i < n) \ \&\& \ (v[i] \neq 0))$  is false
- Use control-flow graph to represent this short-circuit evaluation

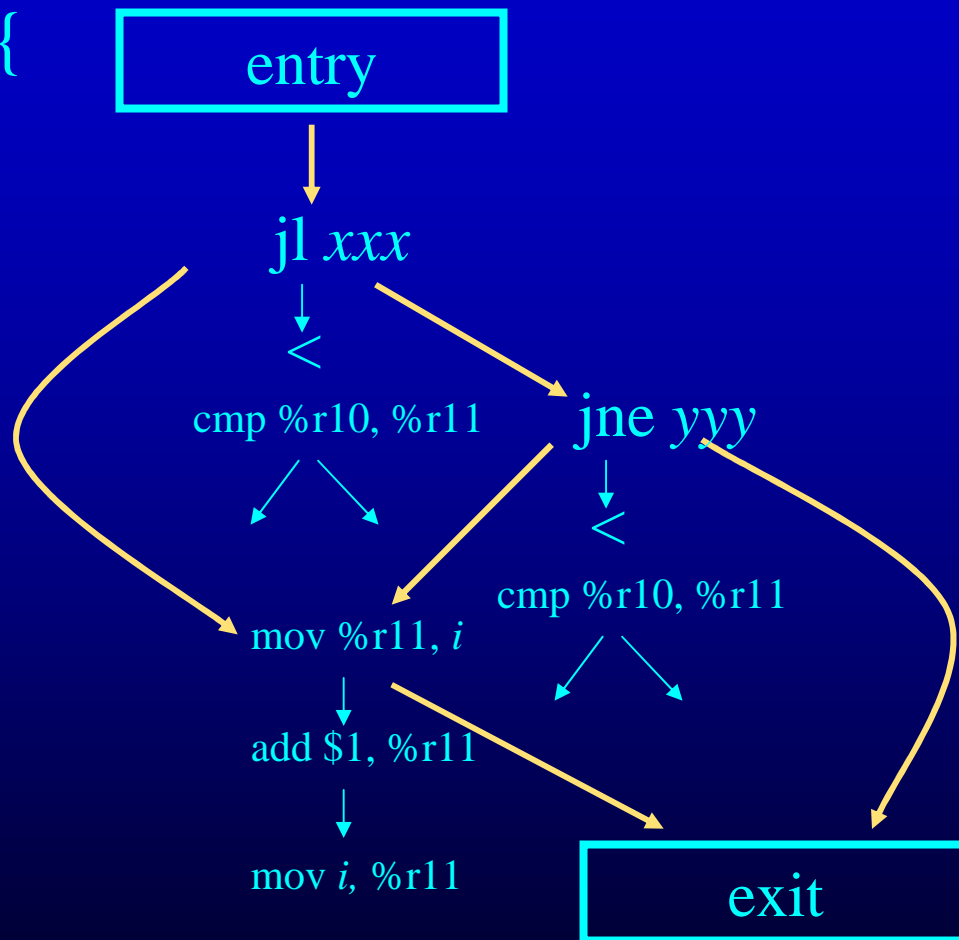
# Short-Circuit Conditionals

```
while (i < n && v[i] != 0) {  
    i = i+1;  
}
```



# More Short-Circuit Conditionals

```
if (a < b || c != 0) {  
    i = i+1;  
}
```



# Routines for Destructuring Program Representation

destruct(**n**)

generates lowered form of structured code represented by **n**

returns (**b,e**) - **b** is begin node, **e** is end node in destructed form

shortcircuit(**c, t, f**)

generates short-circuit form of conditional represented by **c**

if **c** is true, control flows to **t** node

if **c** is false, control flows to **f** node

returns **b** - **b** is begin node for condition evaluation

new kind of node - nop node

# Destructuring Seq Nodes

`destruct(n)`

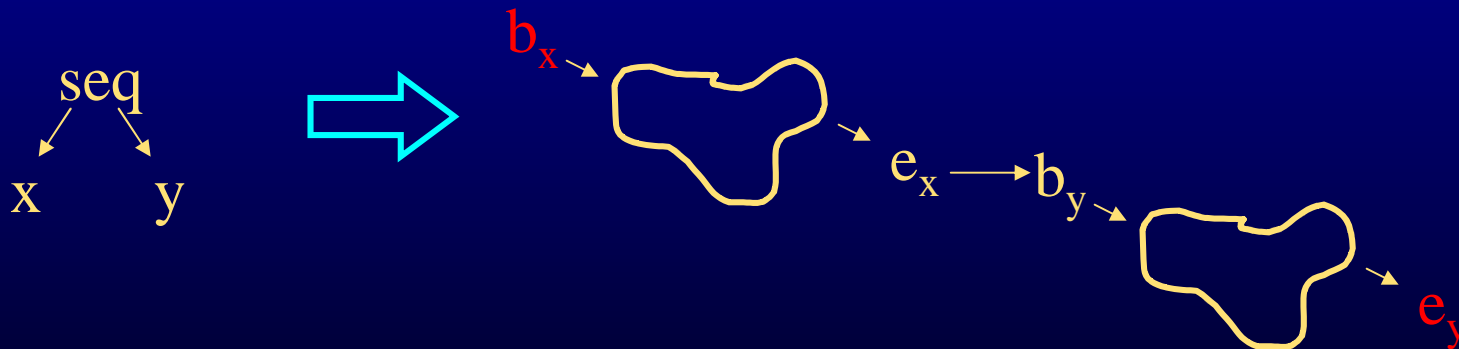
generates lowered form of structured code represented by `n`

returns `(b,e)` - `b` is begin node, `e` is end node in destructed form

if `n` is of the form `seq x y`

1: `(bx,ex) = destruct(x)`; 2: `(by,ey) = destruct(y)`;

3: `next(ex) = by`; 4: `return (bx, ey)`;



# Destructuring If Nodes

destruct(**n**)

generates lowered form of structured code represented by **n**

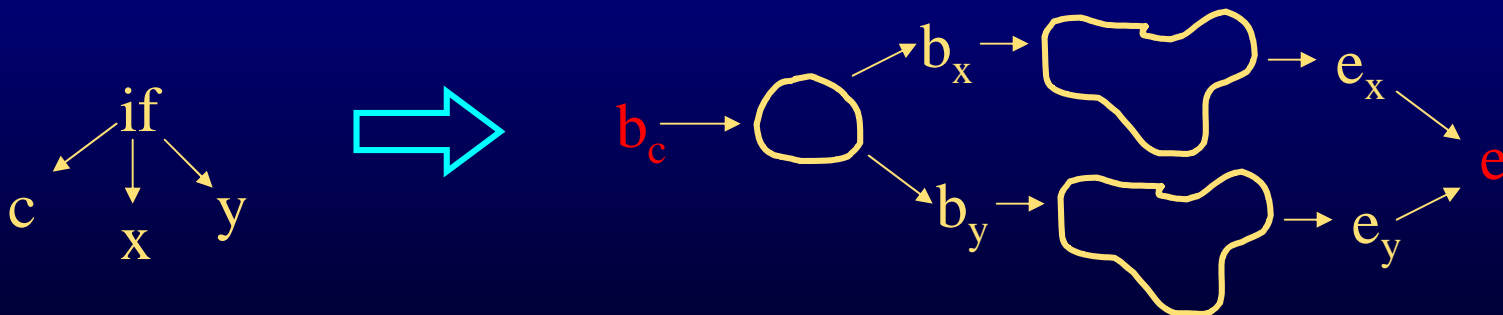
returns (**b,e**) - **b** is begin node, **e** is end node in destructed form

if **n** is of the form **c x y**

1: ( $b_x, e_x$ ) = destruct(**x**); 2: ( $b_y, e_y$ ) = destruct(**y**);

3:  $e = \text{new nop}$ ; 4:  $\text{next}(e_x) = e$ ; 5:  $\text{next}(e_y) = e$ ;

6:  $b_c = \text{shortcircuit}(c, b_x, b_y)$ ; 7: return ( $b_c, e$ );



# Destructuring While Nodes

destruct(**n**)

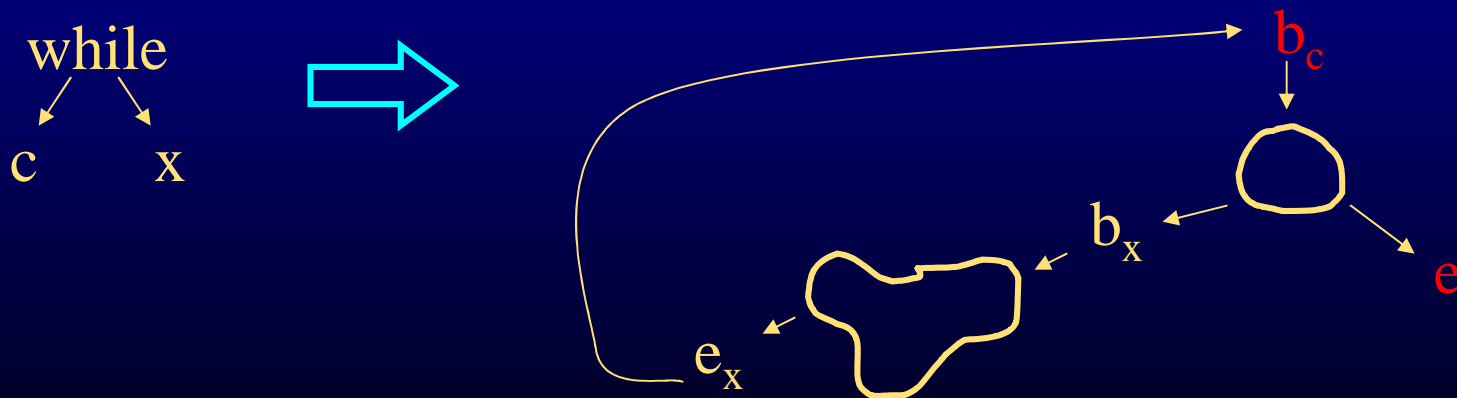
generates lowered form of structured code represented by **n**

returns (**b,e**) - **b** is begin node, **e** is end node in destructed form

if **n** is of the form while **c x**

1:  $e = \text{new nop}$ ; 2:  $(b_x, e_x) = \text{destruct}(x)$ ;

3:  $b_c = \text{shortcircuit}(c, b_x, e)$ ; 4:  $\text{next}(e_x) = b_c$ ; 5: return  $(b_c, e)$ ;



# Shortcircuiting And Conditions

`shortcircuit(c, t, f)`

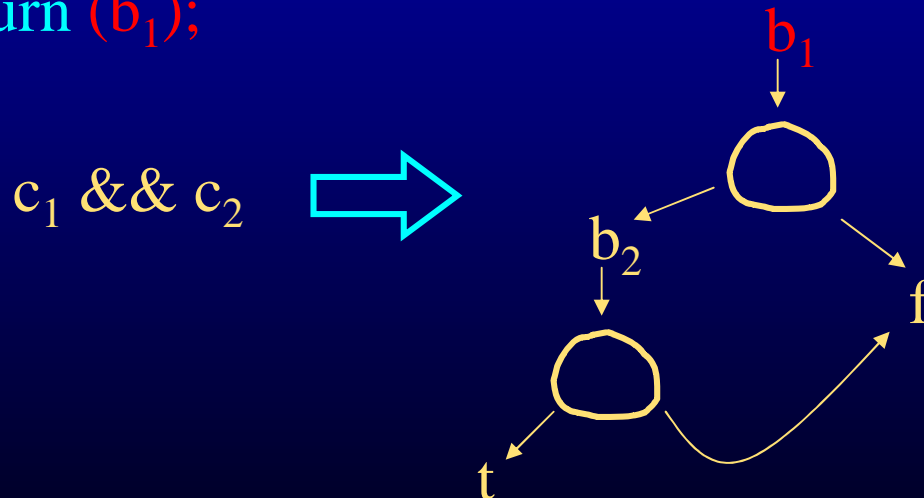
generates shortcircuit form of conditional represented by `c`

returns `b` - `b` is begin node of shortcircuit form

if `c` is of the form `c1 && c2`

1: `b2 = shortcircuit(c2, t, f)`; 2: `b1 = shortcircuit(c1, b2, f)`;

3: `return (b1)`;



# Shortcircuiting Not Conditions

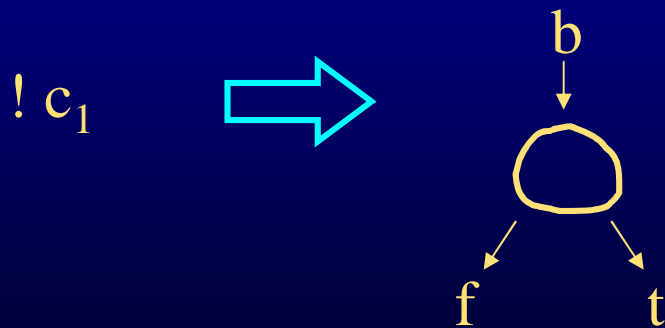
`shortcircuit(c, t, f)`

generates shortcircuit form of conditional represented by `c`

returns `b` - `b` is begin node of shortcircuit form

if `c` is of the form `! c1`

1: `b = shortcircuit(c1, f, t); return(b);`



# Computed Conditions

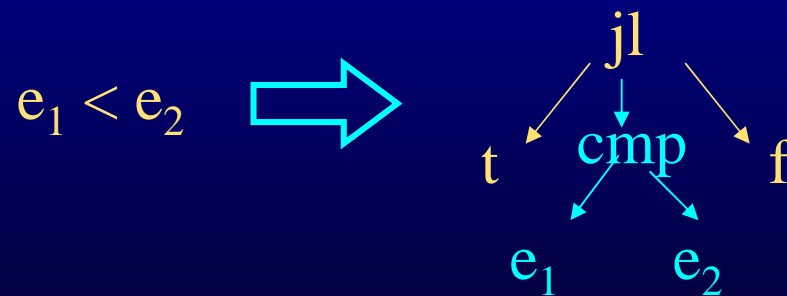
`shortcircuit(c, t, f)`

generates shortcircuit form of conditional represented by `c`

returns `b` - `b` is begin node of shortcircuit form

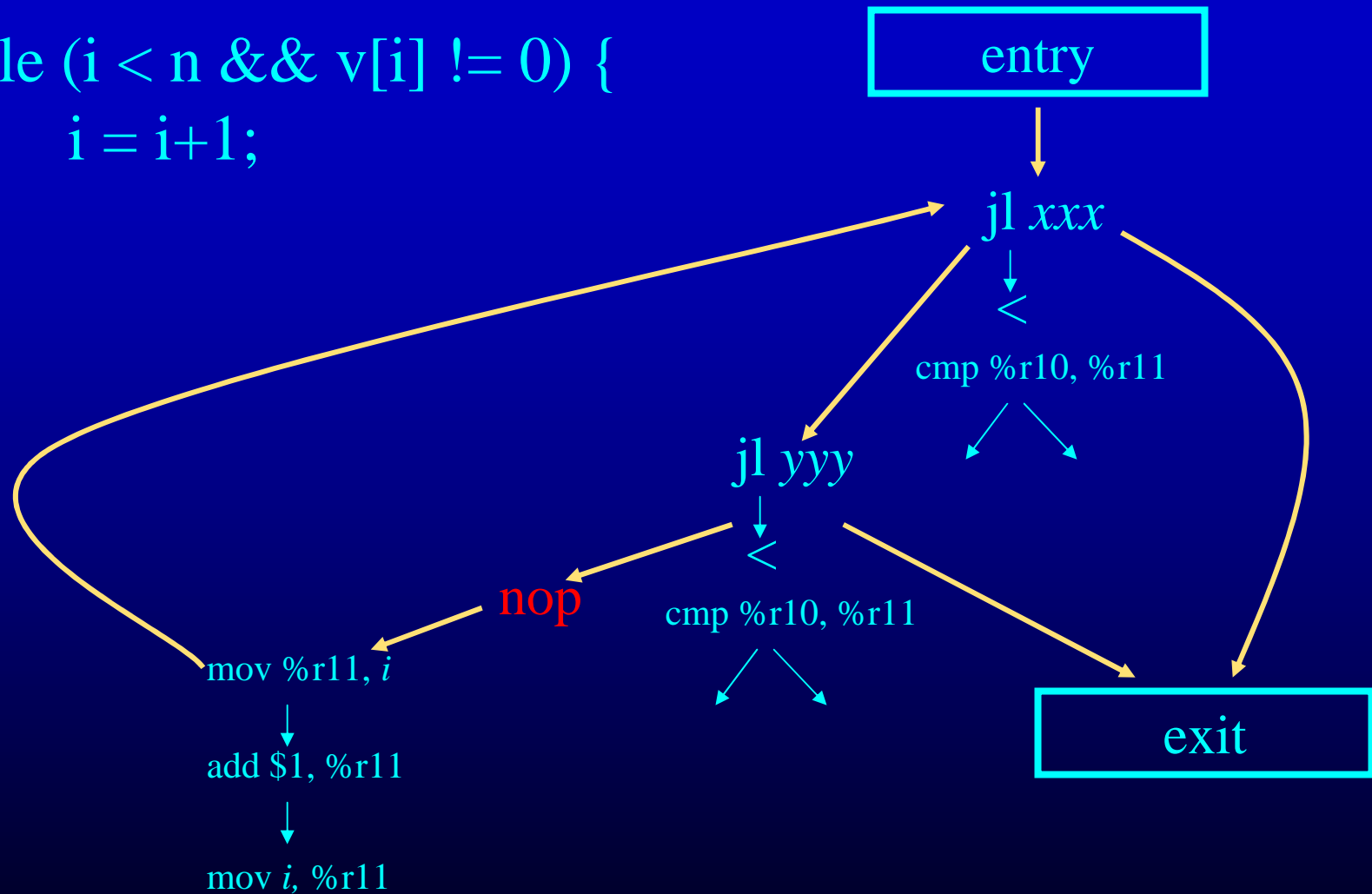
if `c` is of the form  $e_1 < e_2$

1: `b = new cbr( $e_1 < e_2$ , t, f);` 2: `return (b);`

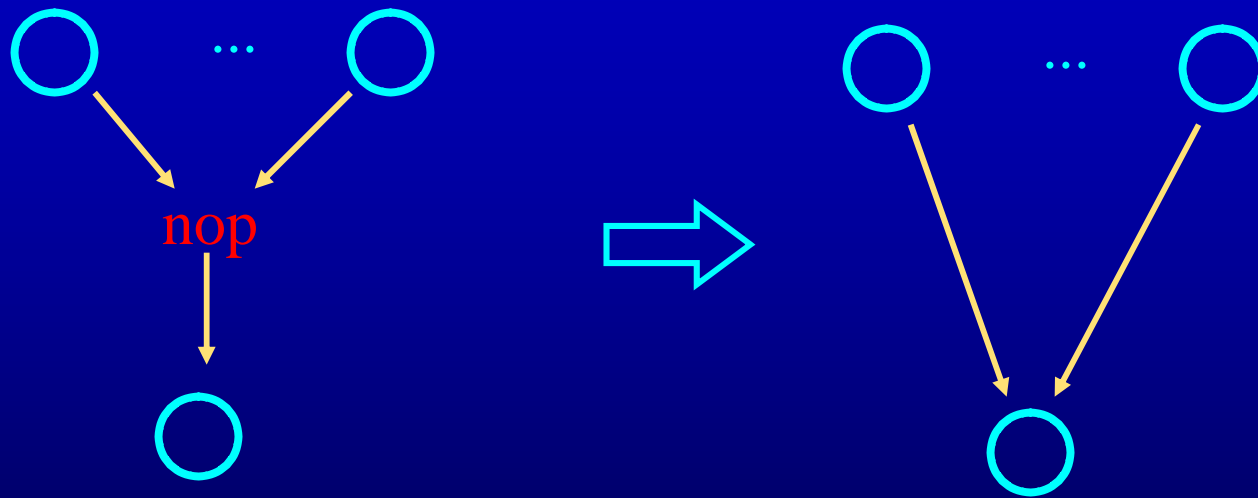


# Nops In Destructured Representation

```
while (i < n && v[i] != 0) {  
    i = i+1;  
}
```



# Eliminating Nops Via Peephole Optimization



# Question:

- What are the pros and cons of template matching vs. algorithmic approach?

# The x86-64 Processor

- A CISC Architecture (vs. RISC architecture)
- Very long lineage (i.e. a lot of baggage)
  - 8080
  - 8088
  - 80286
  - 80386
  - 80486
  - Pentium
  - Pentium II
  - ...
  - x86-64
- First AMD led design
  - x86 is 32 bit, Intel wanted a different architecture for 64 bit (Itanium or IA-64)
  - AMD extended x86 to handle 64 bit

# Diversity of Processors

- General Purpose Processors
  - x86, PowerPC, MIPS R4000, HP PA-RISC, Alpha
- Digital Signal Processors (DSP)
  - TI 56000
- Supercomputing Processors
  - Cray
- Embedded Processors
  - StrongARM
- Network Processors

# Diversity of Processors

- Diversity in execution
  - VLIW, Superscalar, Vector, Systolic Arrays
- Diversity in the memory system
  - Multiple memories in DSPs
  - register windows in SPARC
- Different/unique ISAs
- Different goals/markets
  - All out performance in supercomputers
  - Maximum energy savings in embedded processors

# What We Covered Today..

## CODE

Procedures

Control Flow

Statements

Data Access

## DATA

Global Static Variables

Global Dynamic Data

Local Variables

Temporaries

Parameter Passing

Read-only Data

# Guidelines for the code generator

- Lower the abstraction level slowly
  - Do many passes, that do few things (or one thing)
    - Easier to break the project down, generate and debug
- Keep the abstraction level consistent
  - IR should have ‘correct’ semantics at all time
    - At least you should know the semantics
  - You may want to run some of the optimizations between the passes.
- Use assertions liberally
  - Use an assertion to check your assumption

# Guidelines for the code generator

- Do the simplest but dumb thing
  - it is ok to generate  $0 + 1*x + 0*y$
  - Code is painful to look at; let optimizations improve it
- Make sure you know what can be done at...
  - Compile time in the compiler
  - Runtime using generated code

# Guidelines for the code generator

- Remember that optimizations will come later
  - Let the optimizer do the optimizations
  - Think about what optimizer will need and structure your code accordingly
  - Example: Register allocation, algebraic simplification, constant propagation
- Setup a good testing infrastructure
  - regression tests
    - If a input program creates a bug, use it as a regression test
  - Learn good bug hunting procedures
    - Example: binary search