

6.035

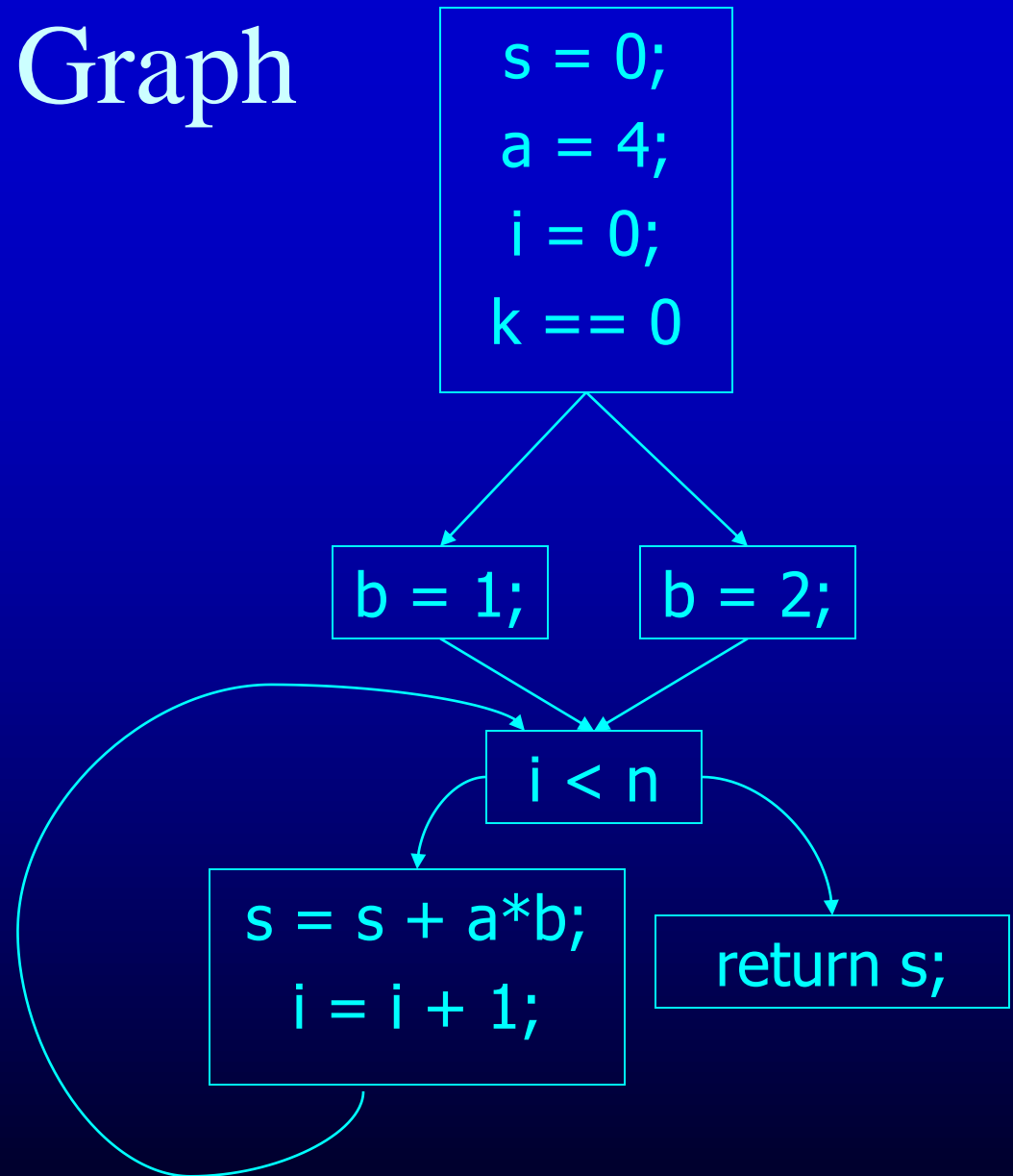
Unoptimized Code Generation

Big Picture

- Starting point – structured IR
 - Intermediate point – CFG (control flow graph)
 - Ending point – Generated Assembly Code
-
- Emphasis on UNOPTIMIZED
 - Do simplest possible thing for now
 - Will treat optimizations separately

Control Flow Graph

```
into add(n, k) {  
    s = 0; a = 4; i = 0;  
    if (k == 0)  
        b = 1;  
    else  
        b = 2;  
    while (i < n) {  
        s = s + a*b;  
        i = i + 1;  
    }  
    return s;  
}
```



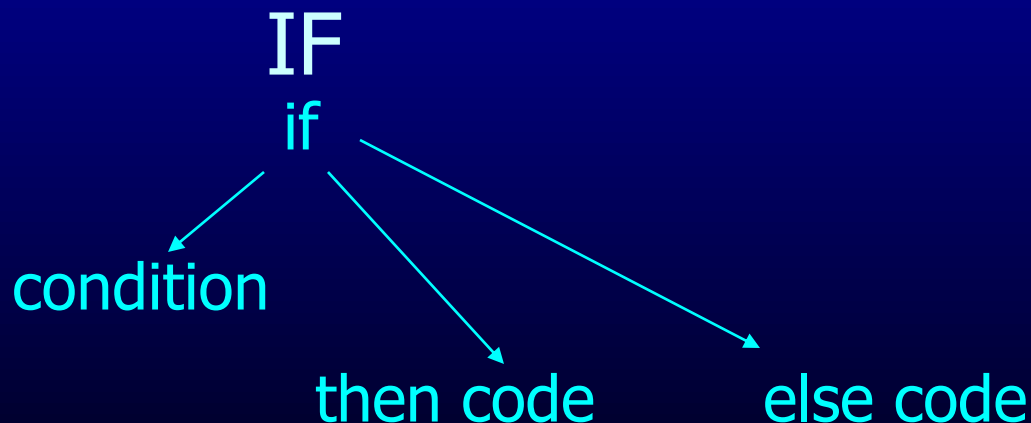
Control Flow Graph

- Nodes Represent Computation
 - Each Node is a Basic Block
 - No Branches Out Of Middle of Basic Block
 - No Branches Into Middle of Basic Block
 - Basic Blocks should be maximal
 - Execution of basic block starts with first instruction
 - Includes all instructions in basic block
- Edges Represent Control Flow

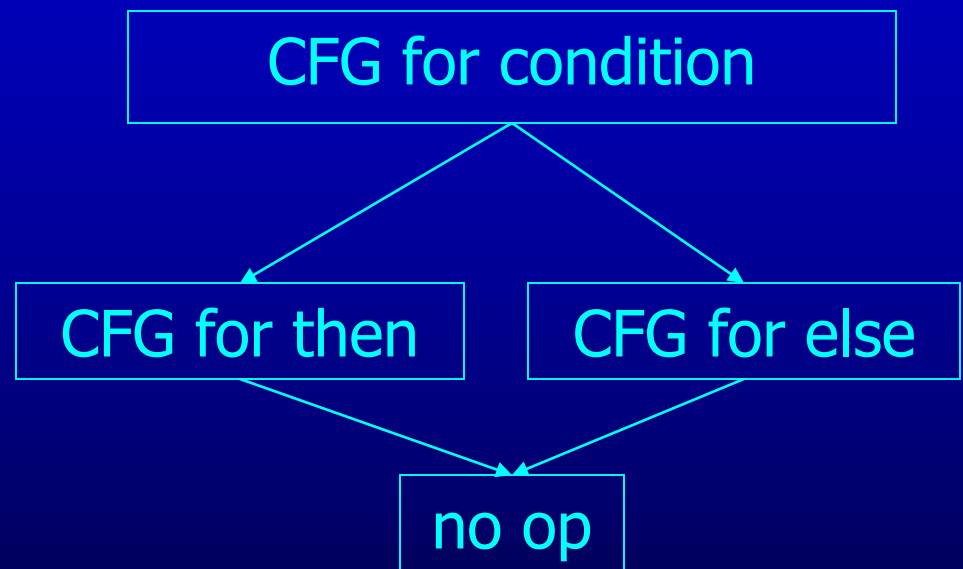
IF to CFG for If Then Else

Source Code

```
if (condition) {  
    code for then  
} else {  
    code for else  
}
```



CFG

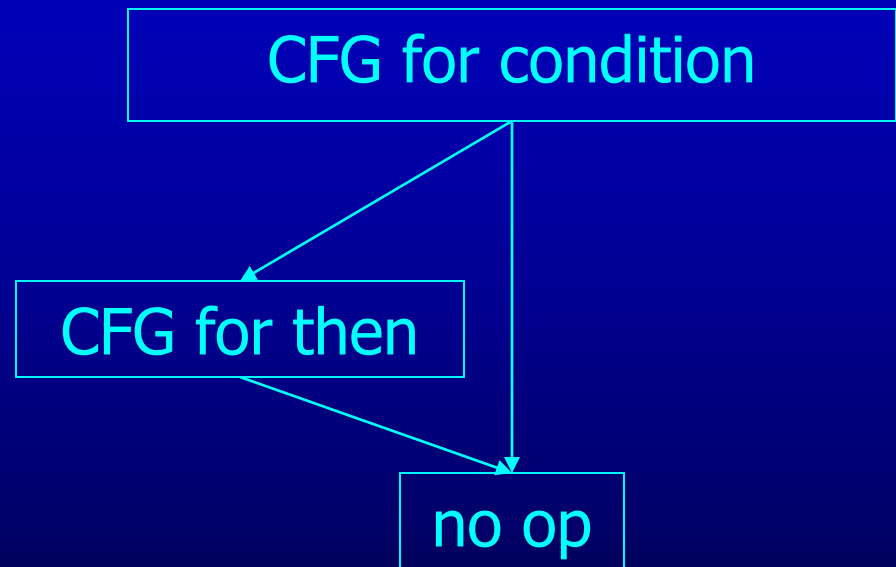


AST to CFG for If Then

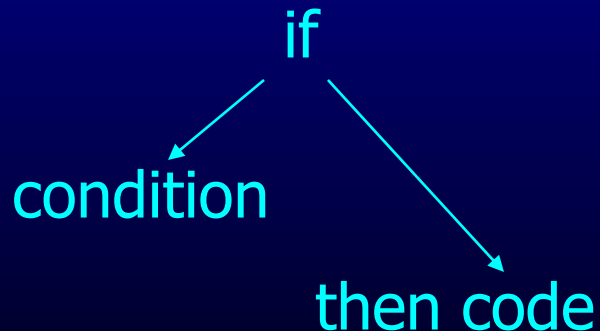
Source Code

```
if (condition) {  
    code for then  
}
```

CFG



IR

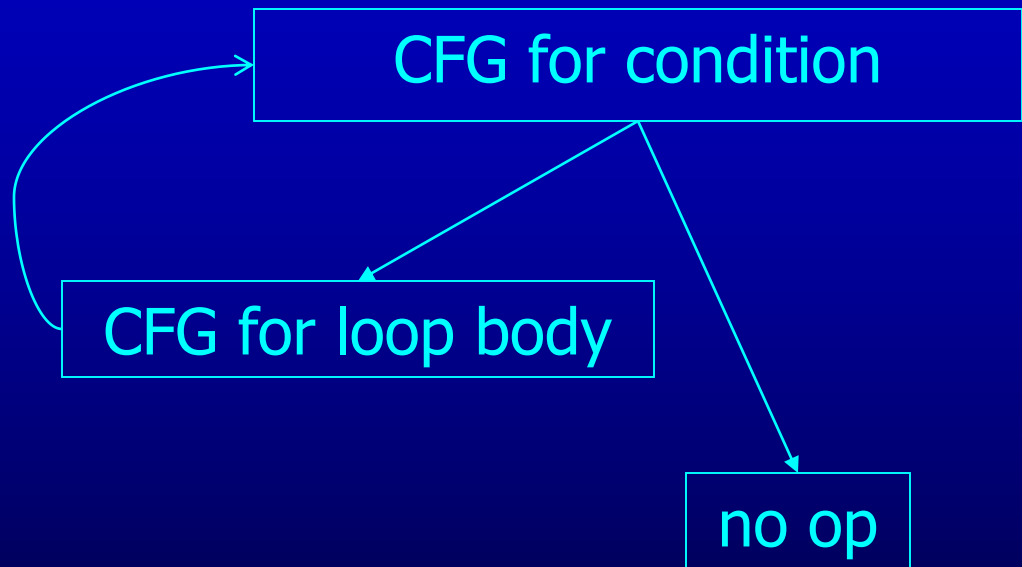


AST to CFG for While

Source Code

```
while (condition) {  
    code for loop body  
}
```

CFG



IR

while

condition

loop body code

AST to CFG for Statements

Source Code

code for S1;
code for S2

IR

seq

code for S1 code for S2



```
graph TD; seq[seq] --> S1[code for S1]; seq --> S2[code for S2];
```

CFG

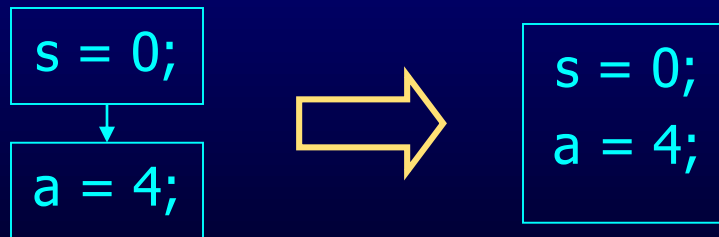
CFG for S1

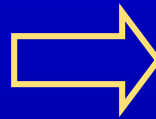
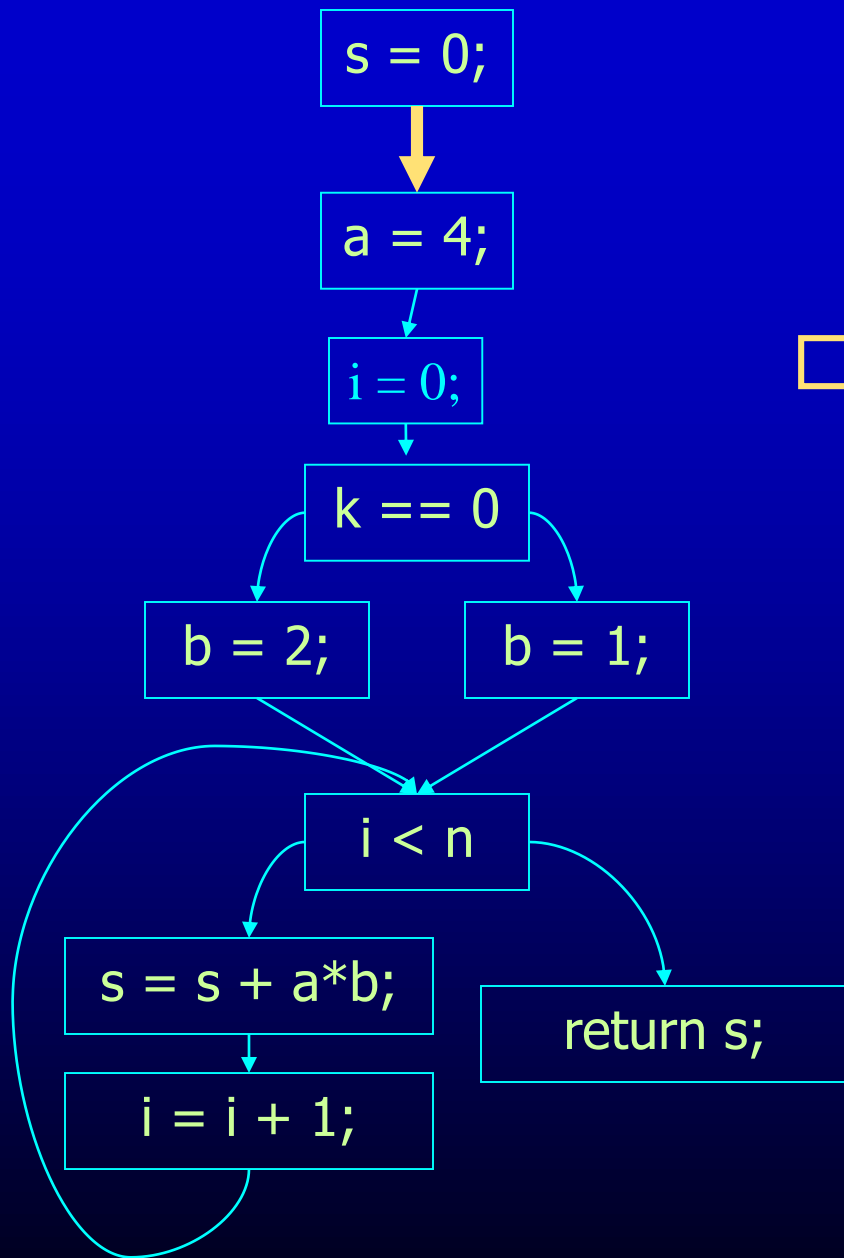


CFG for S2

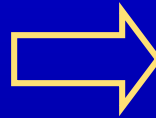
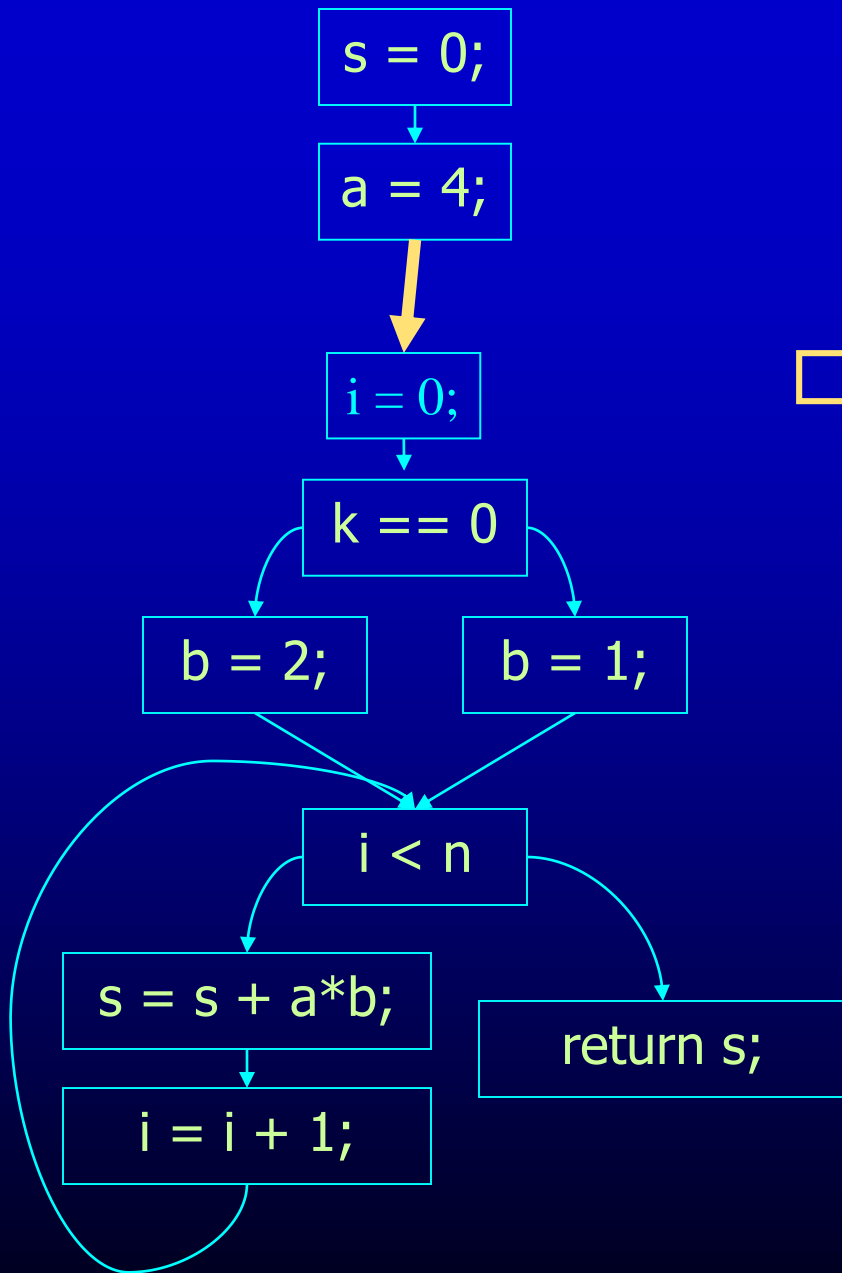
Basic Block Construction

- Start with instruction control-flow graph
- Visit all edges in graph
- Merge adjacent nodes if
 - Only one edge from first node
 - Only one edge into second node

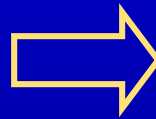
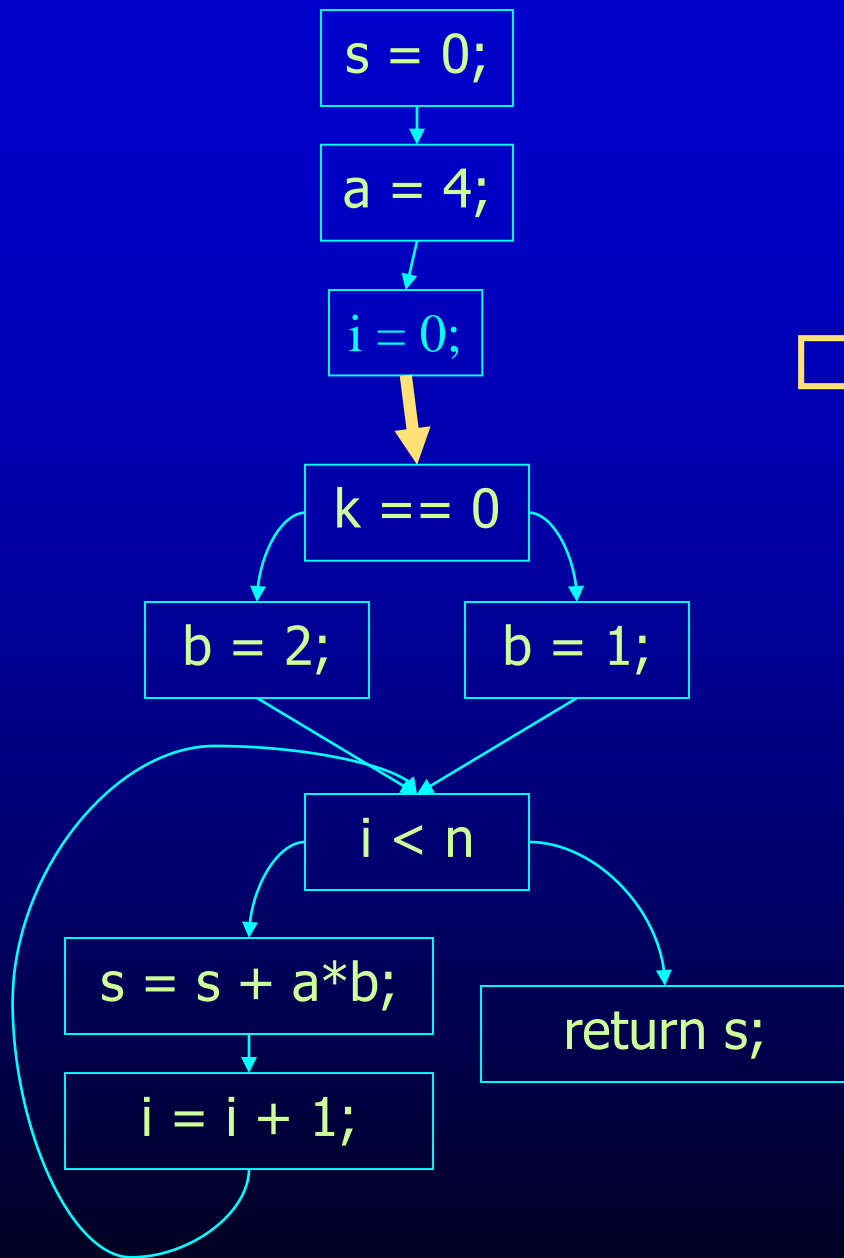




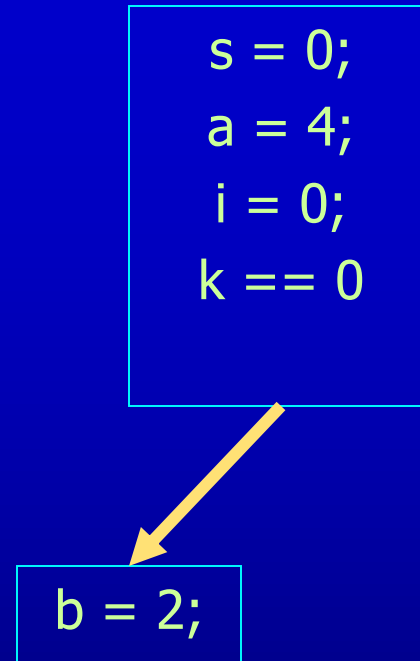
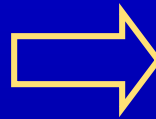
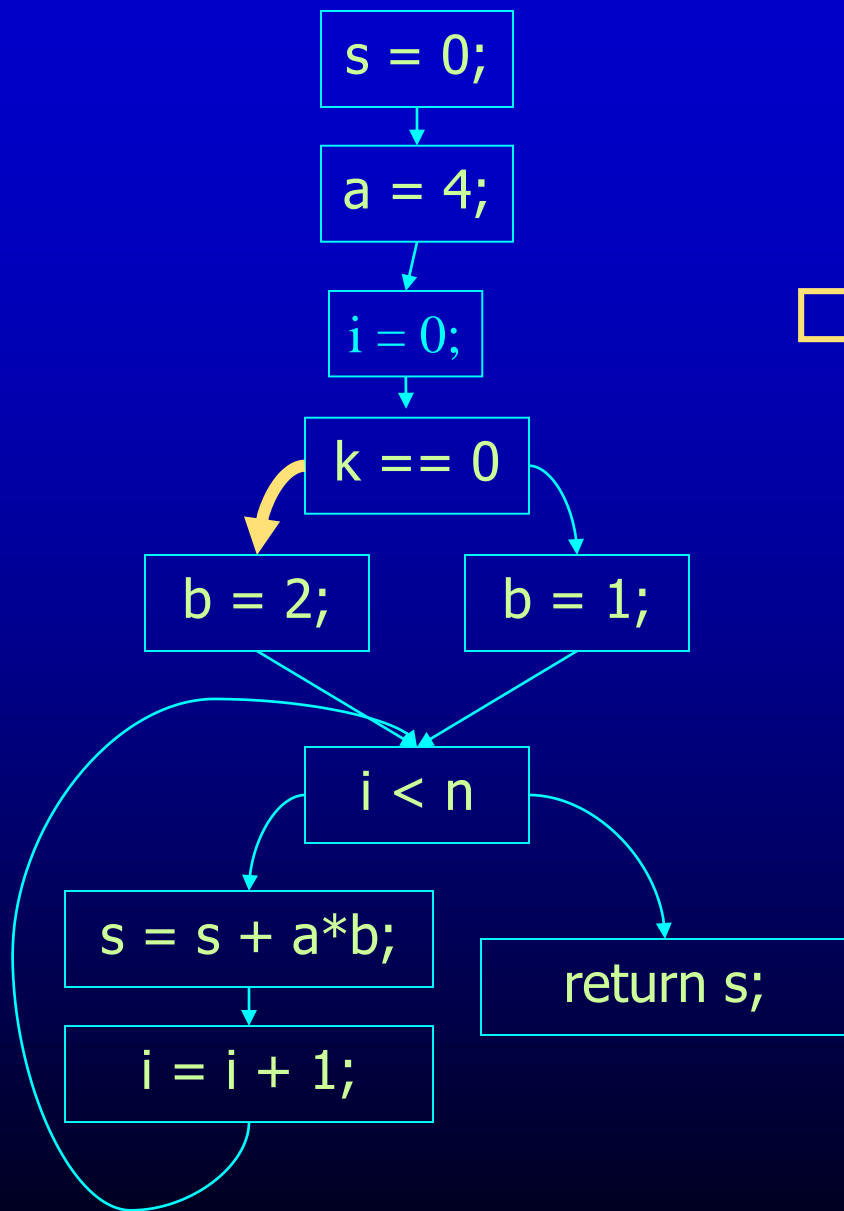
```
s = 0;  
a = 4;
```

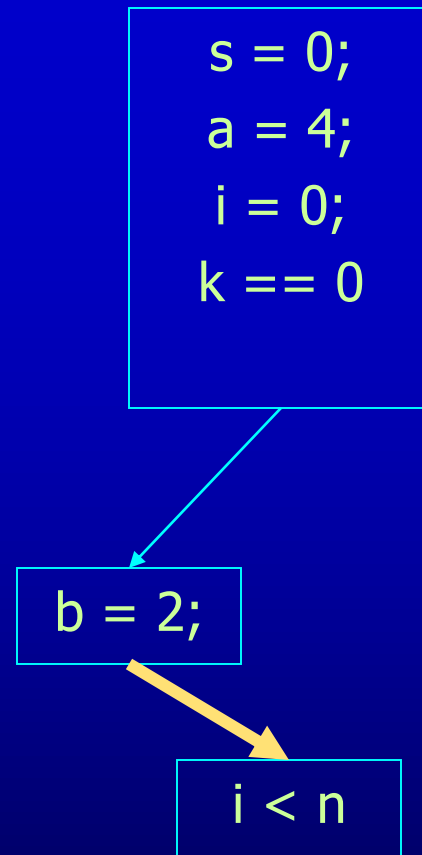
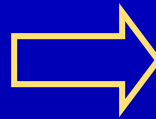
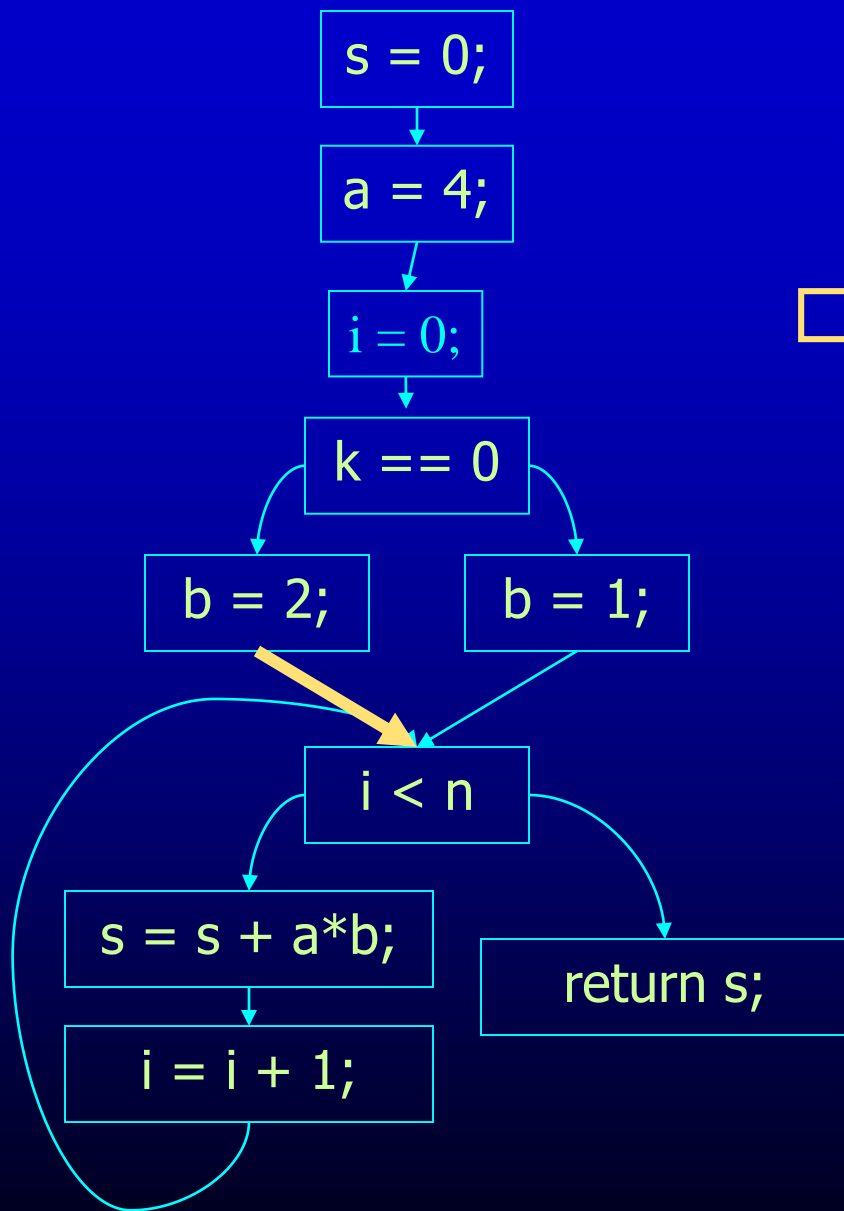


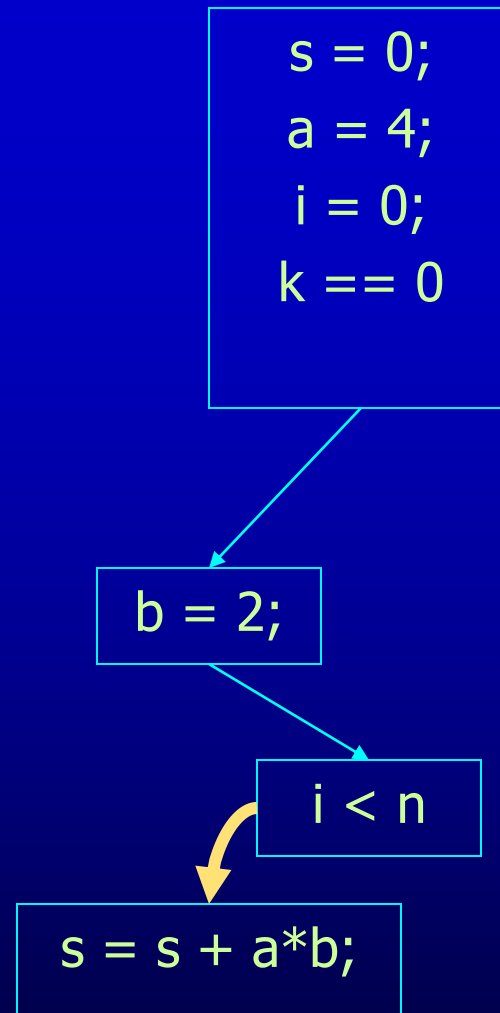
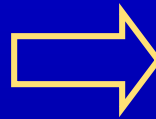
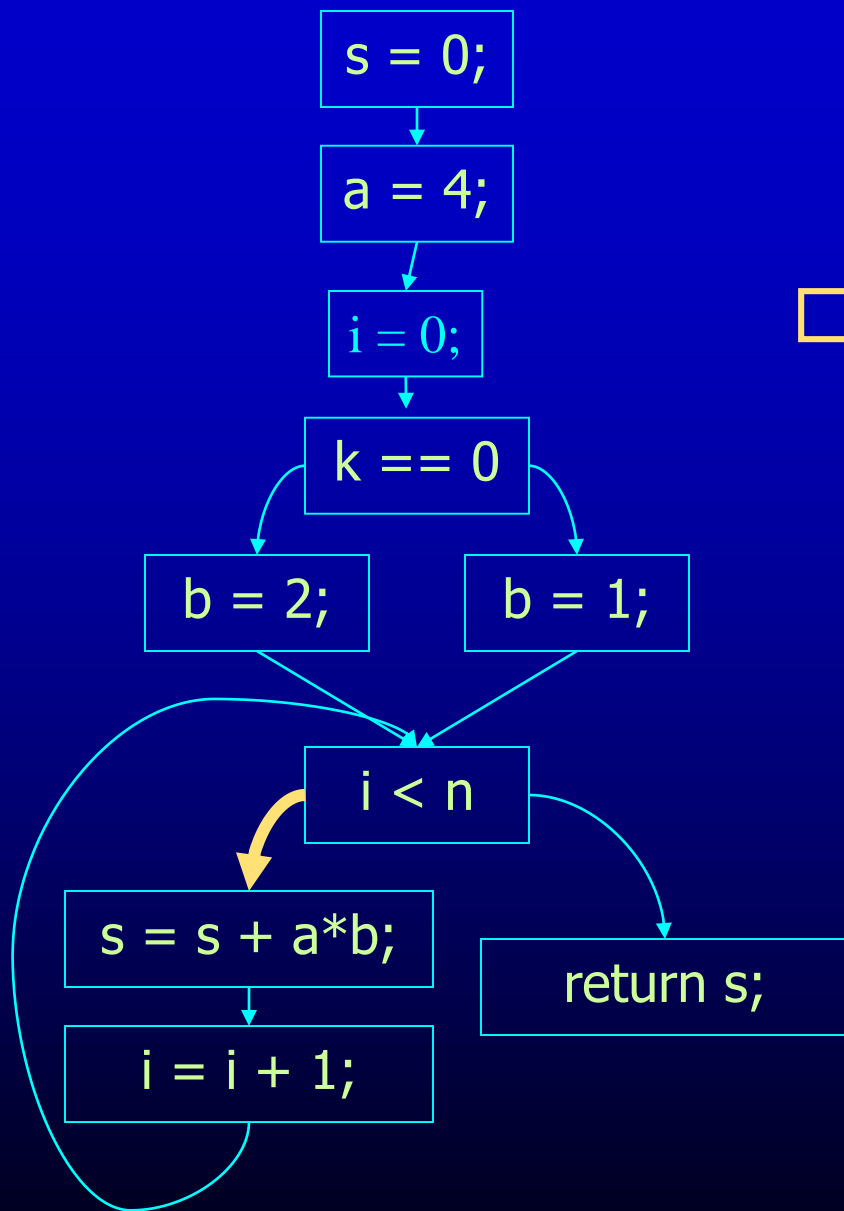
```
s = 0;  
a = 4;  
i = 0;
```

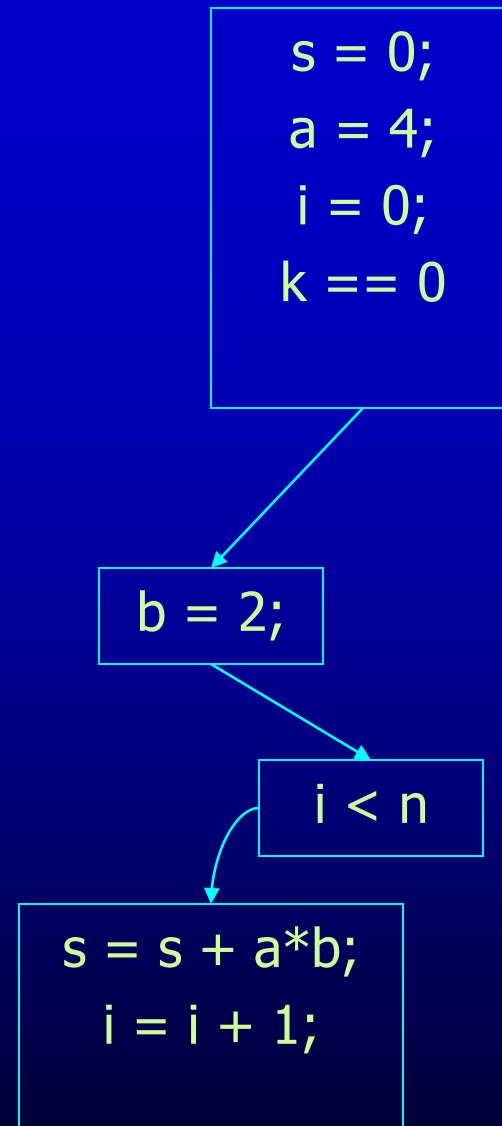
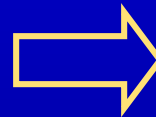
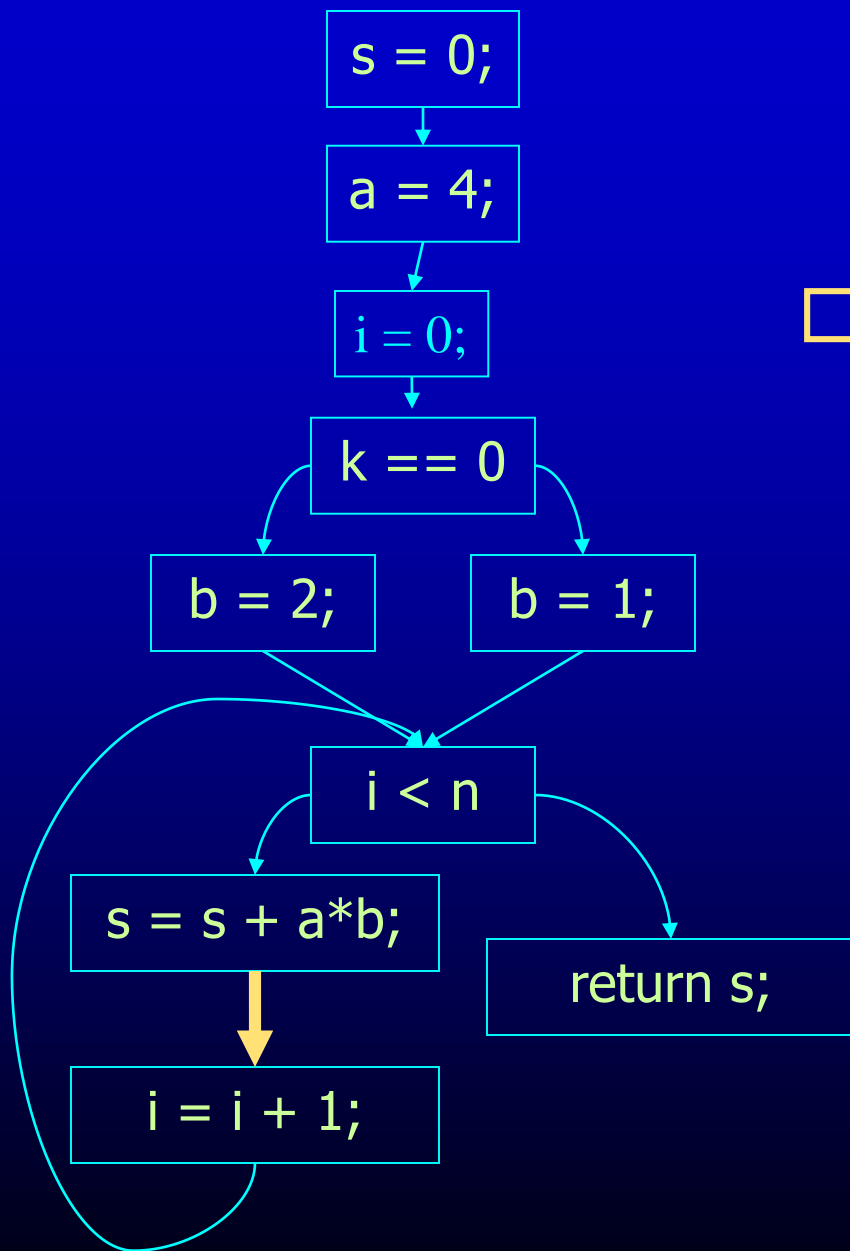


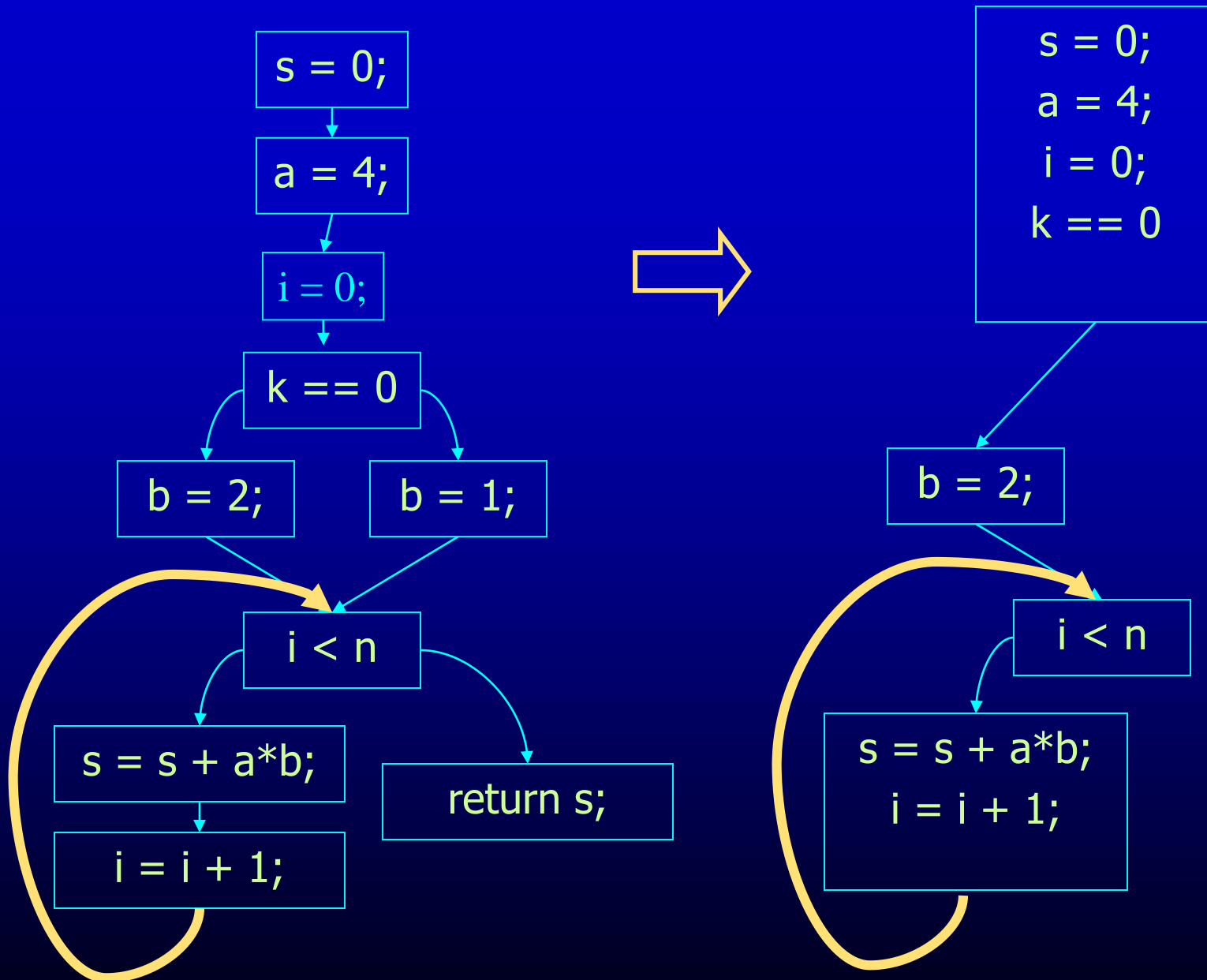
```
s = 0;  
a = 4;  
i = 0;  
k == 0
```

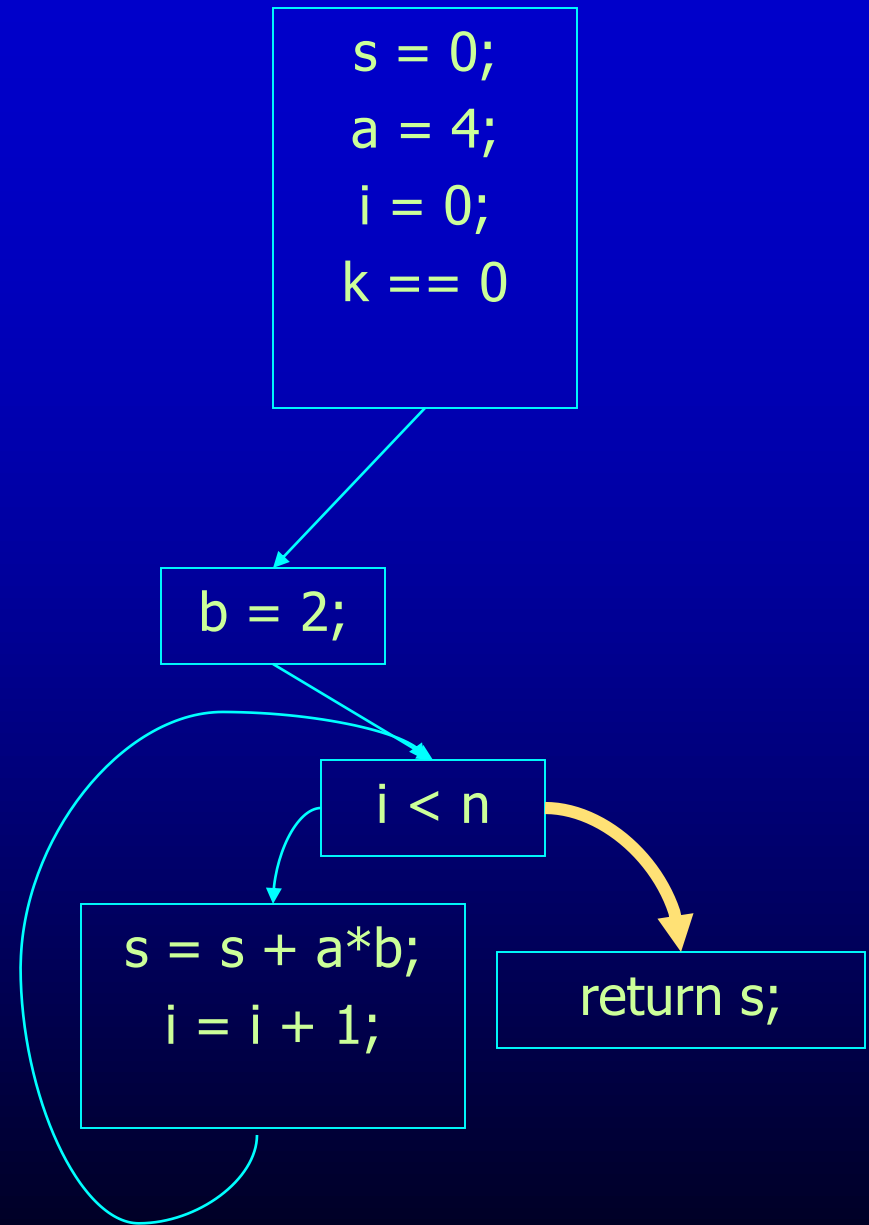
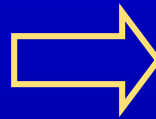
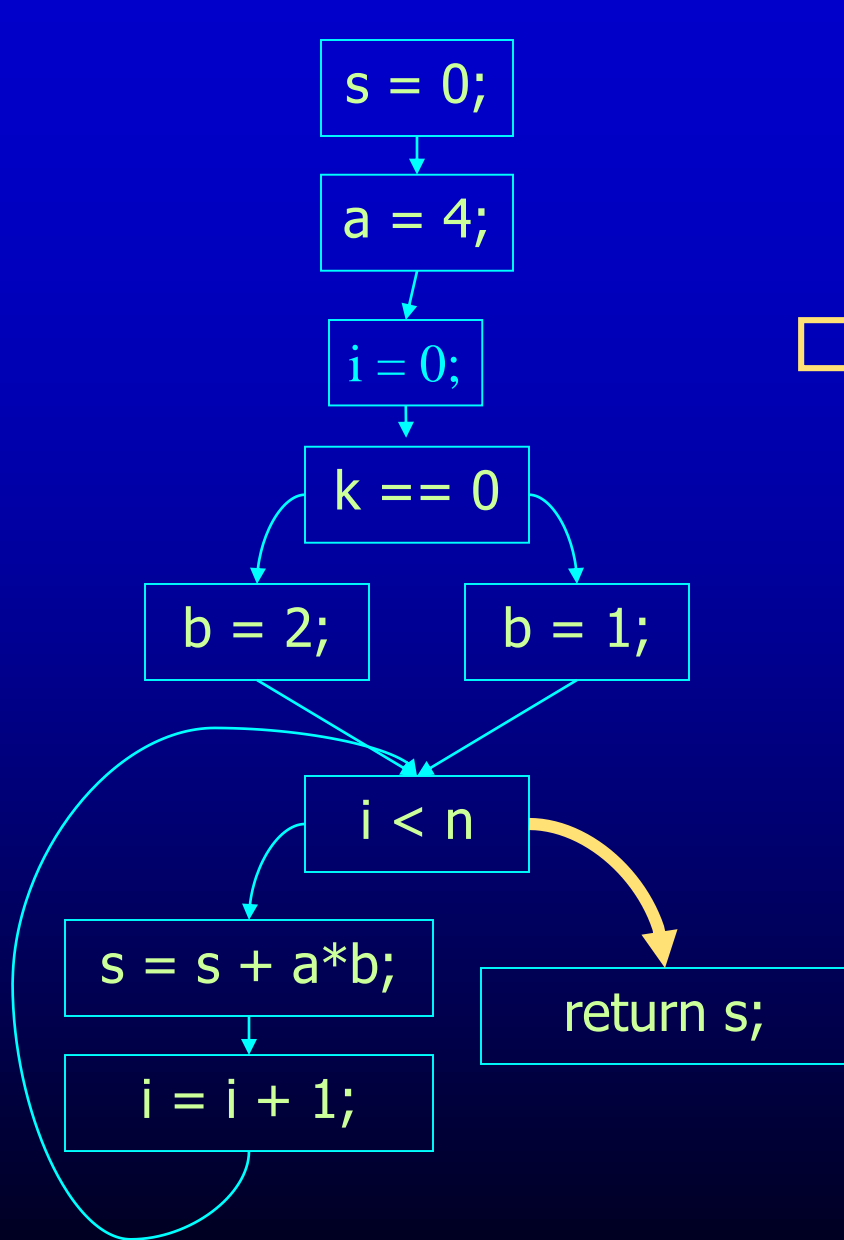


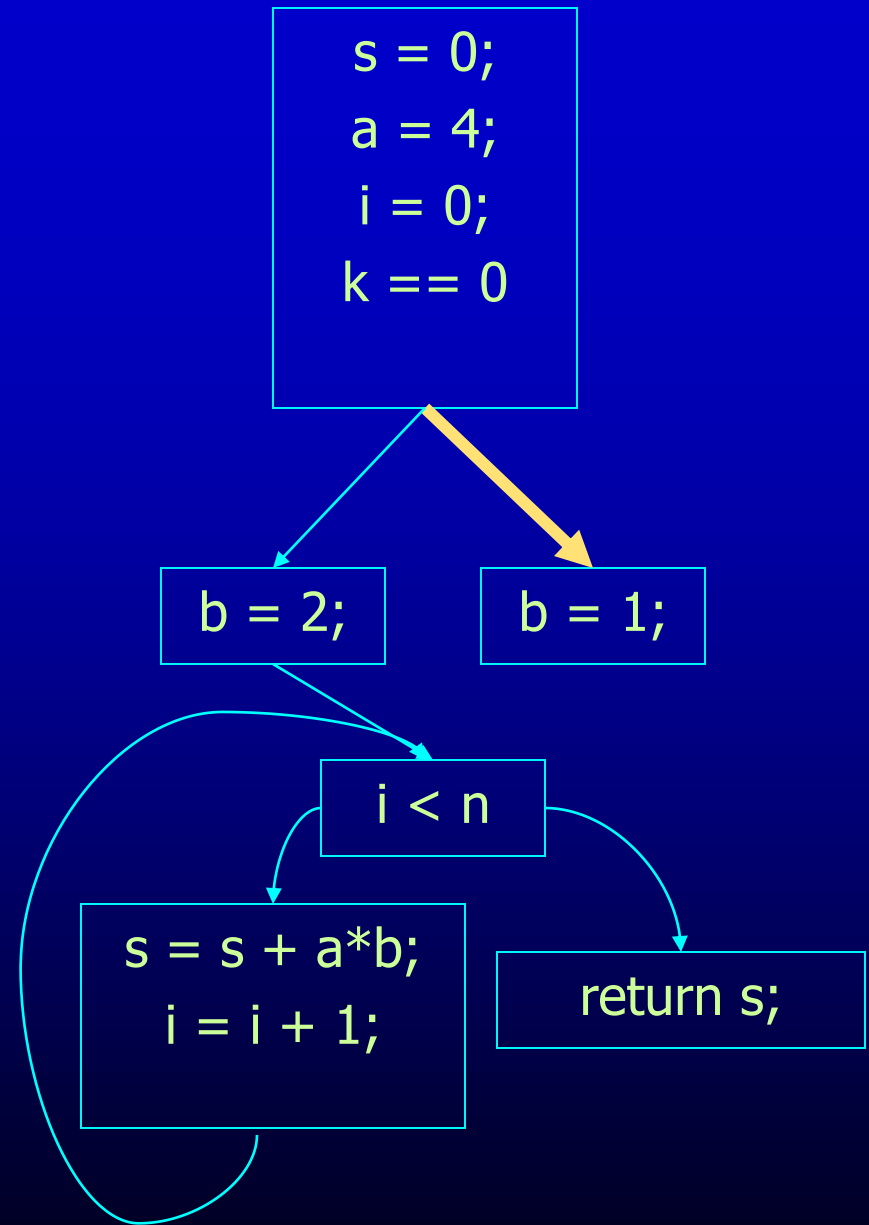
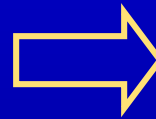
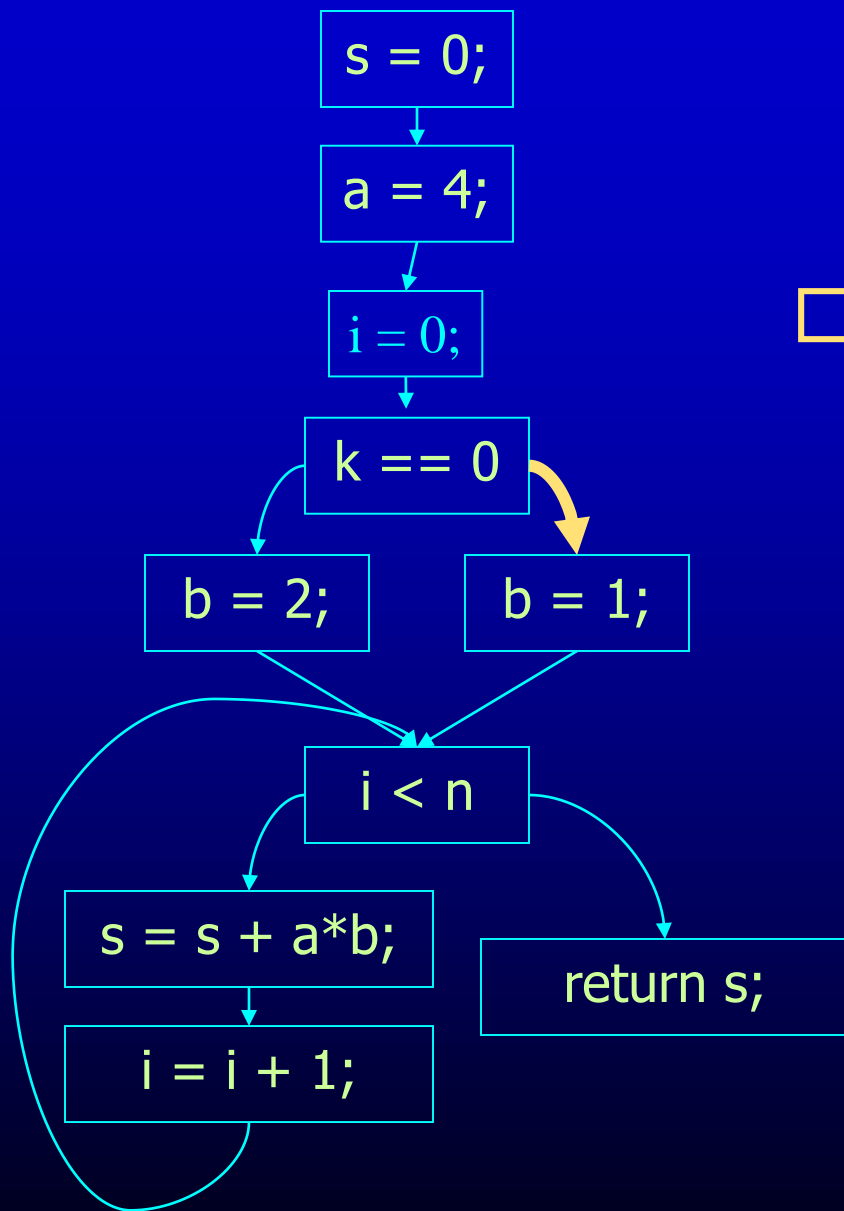


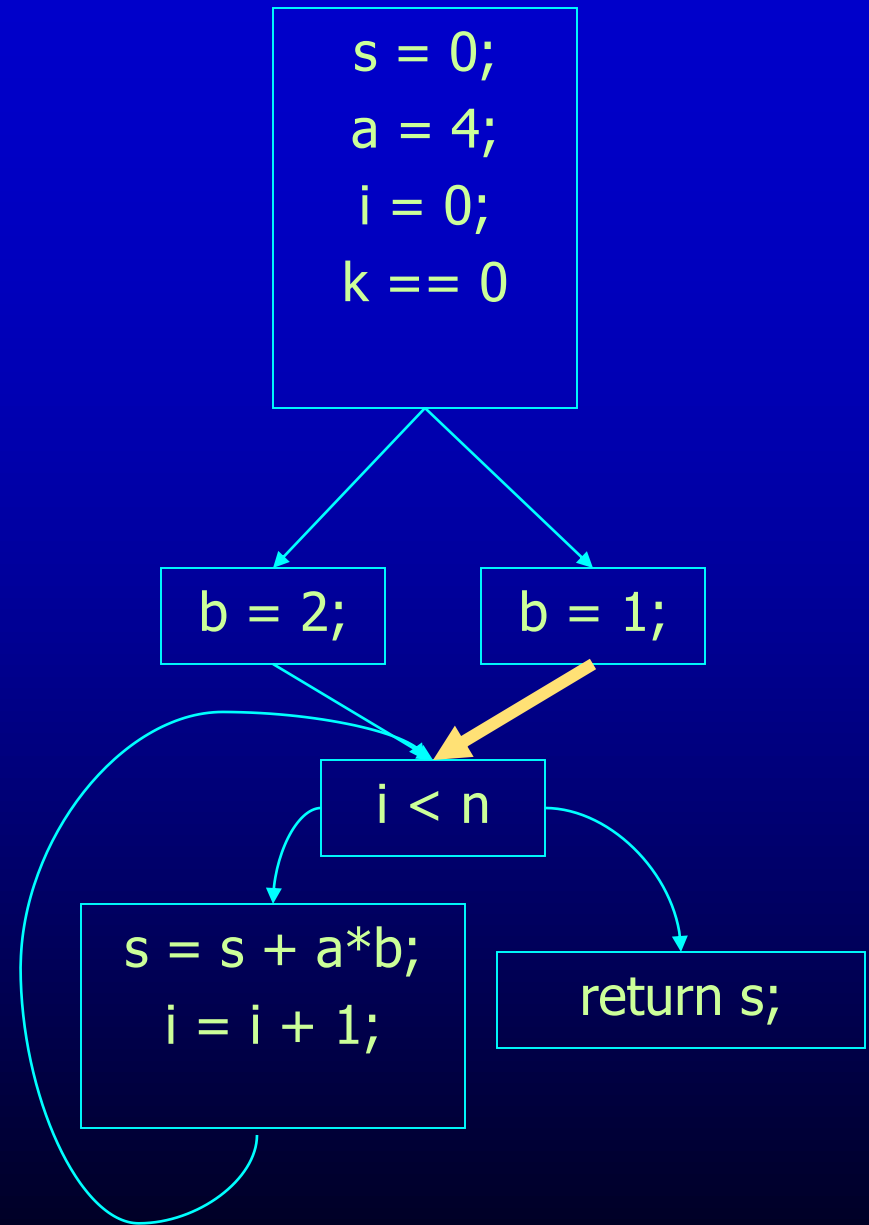
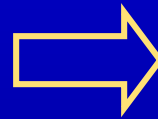
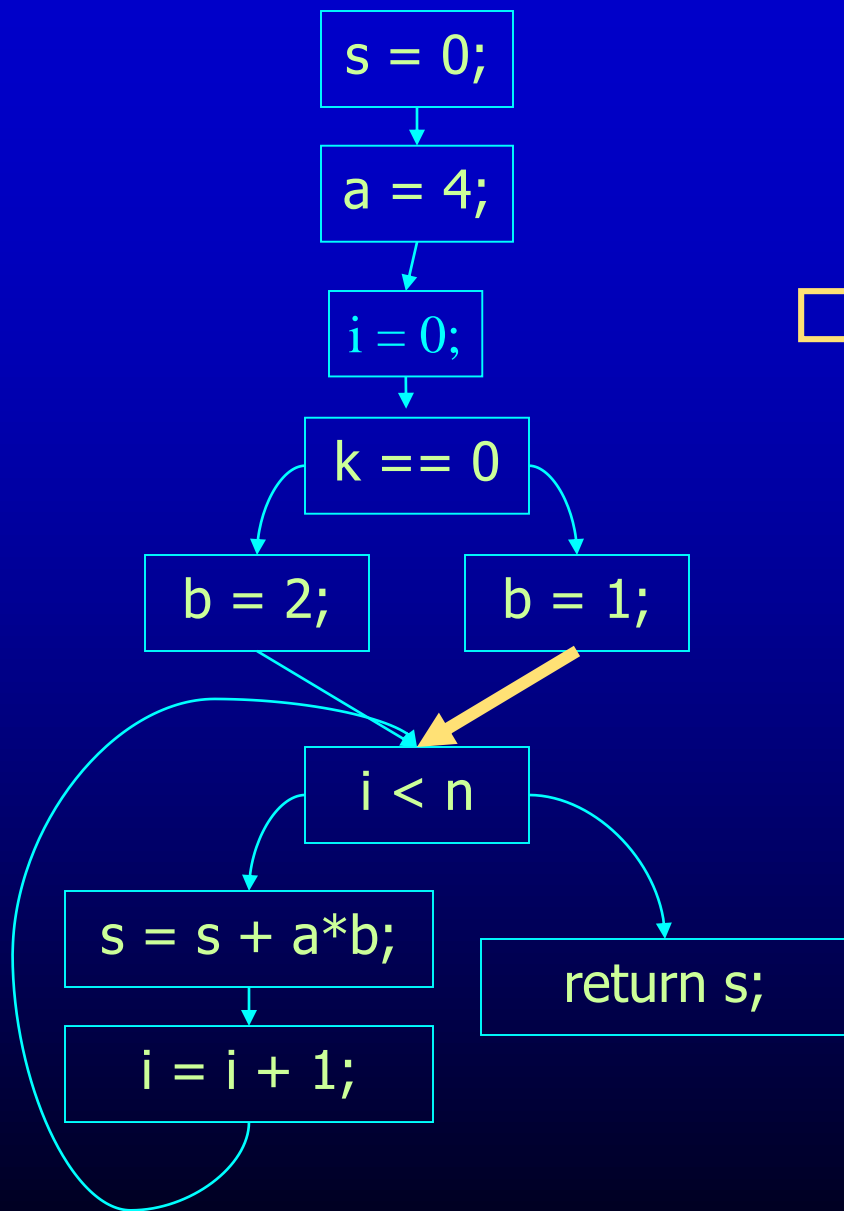


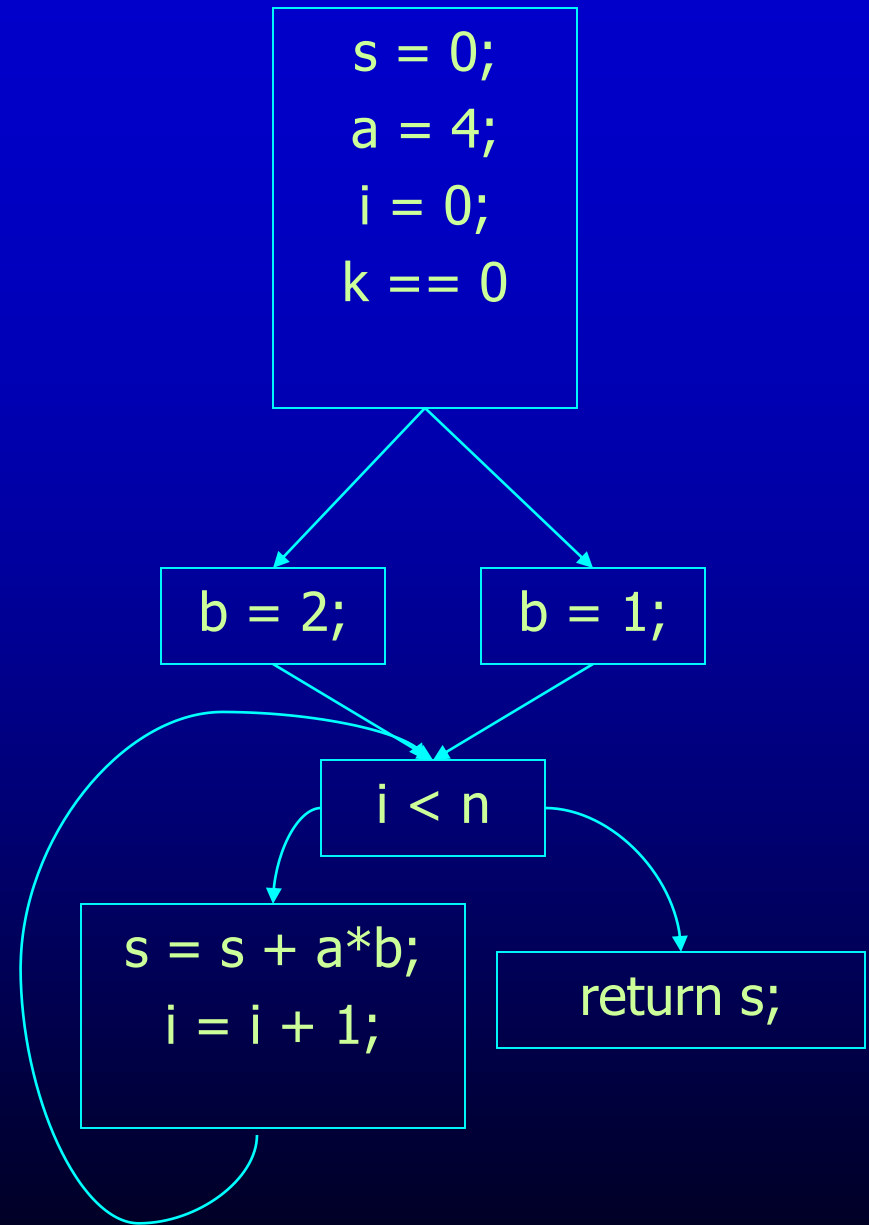
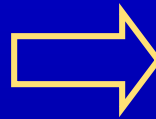
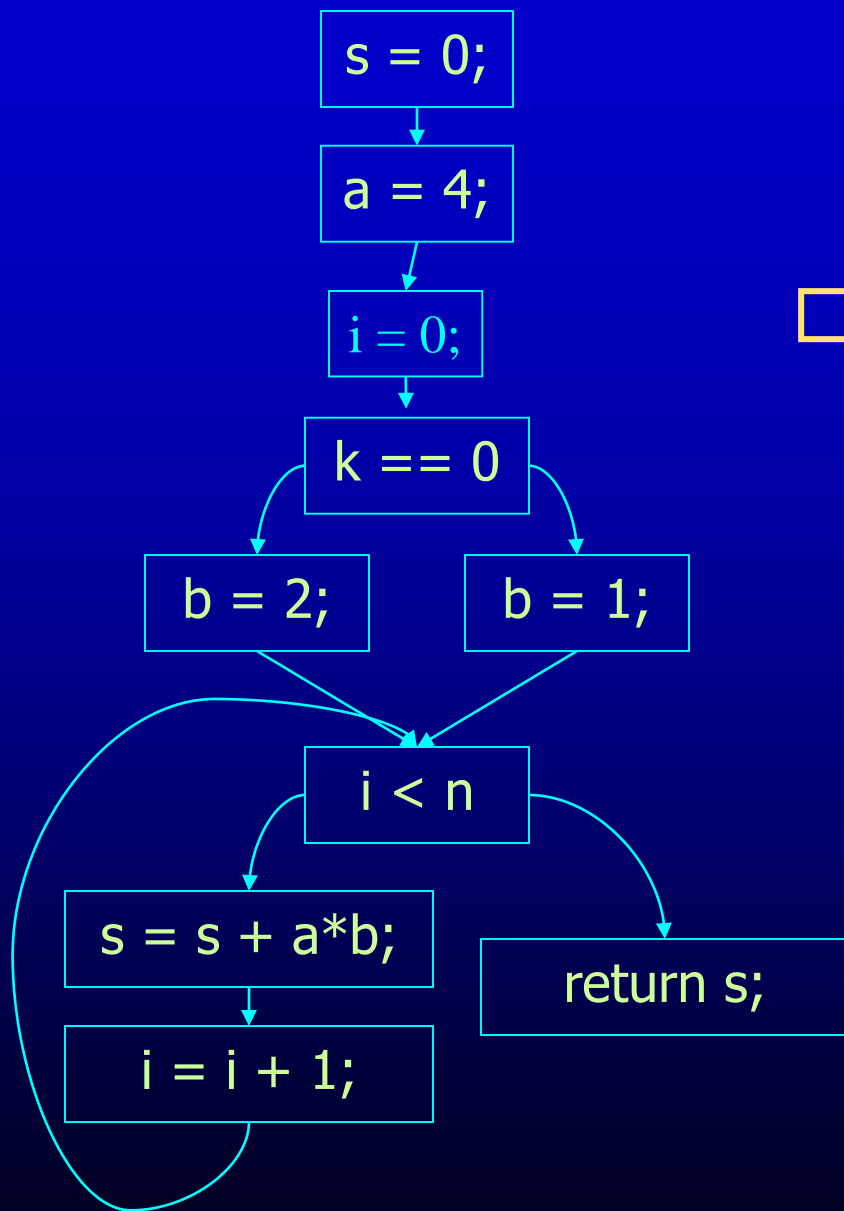












Program Points, Split and Join Points

- One program point before and after each statement in program
- Split point has multiple successors – conditional branch statements only split points
- Merge point has multiple predecessors
- Each basic block
 - Either starts with a merge point or its predecessor ends with a split point
 - Either ends with a split point or its successor starts with a merge point

Motivation For Short-Circuit Conditionals

Following program searches array for 0 element

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

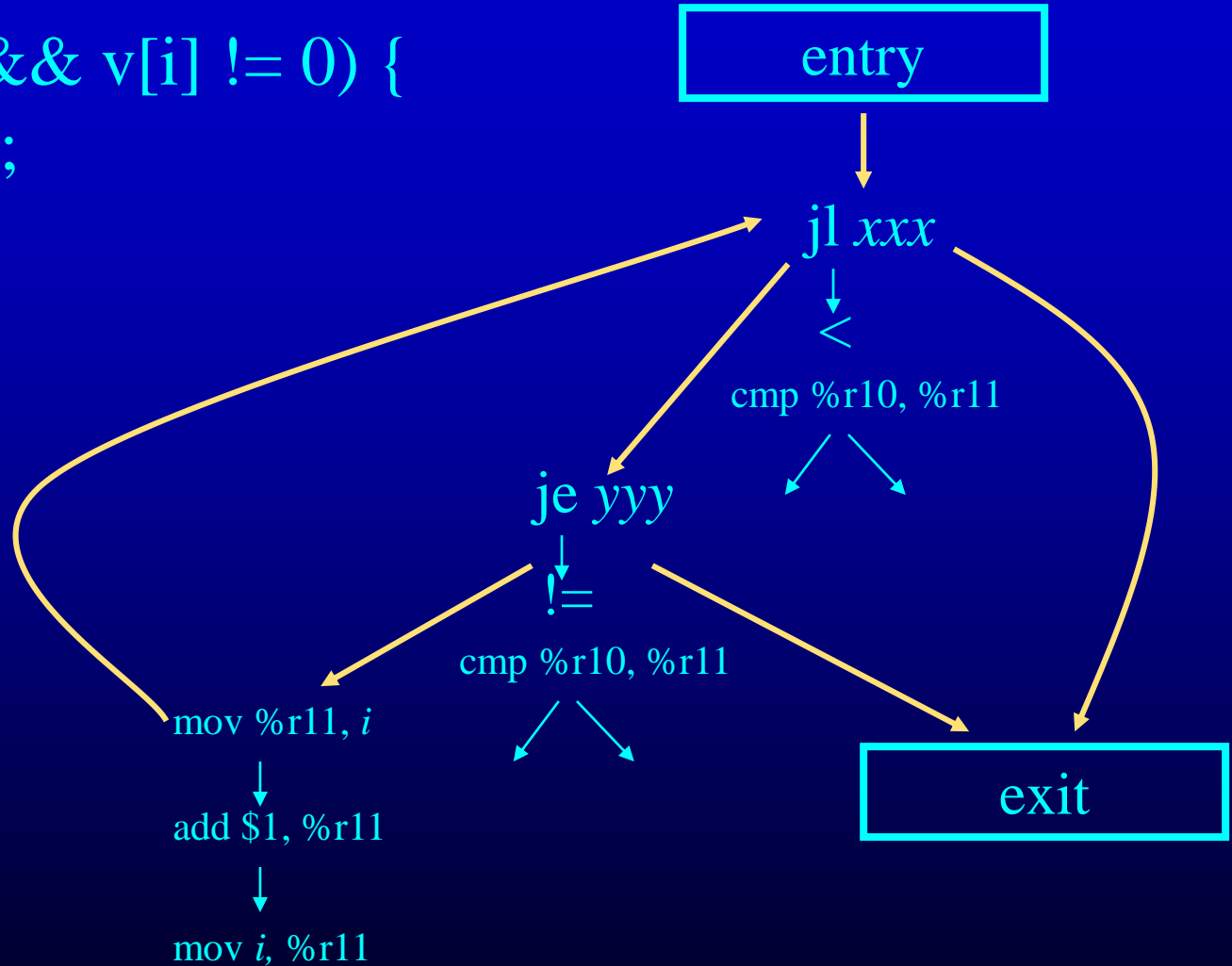
If $i < n$ is false, should you evaluate $a[i] \neq 0$?

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

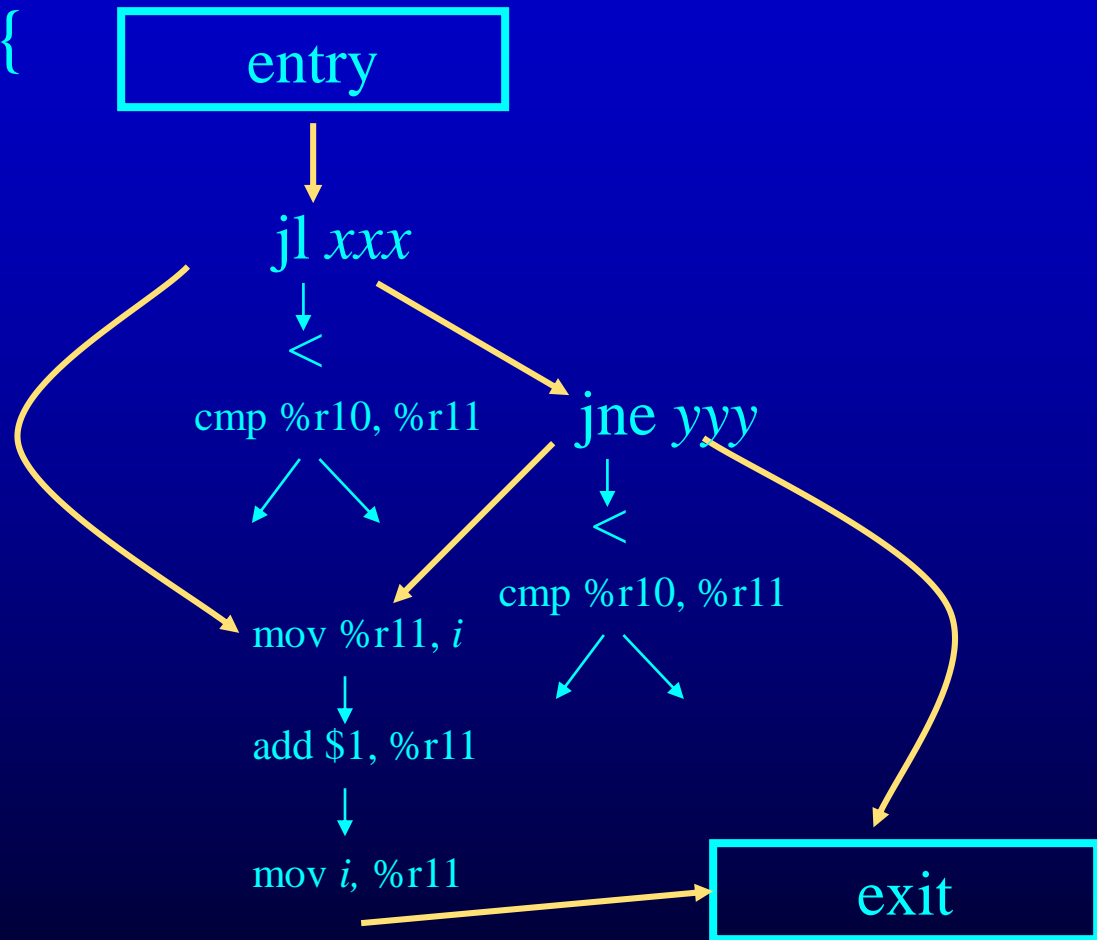
Short-Circuit Conditionals

```
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

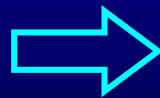
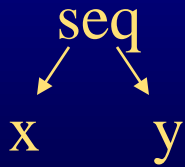
new kind of node - nop node

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



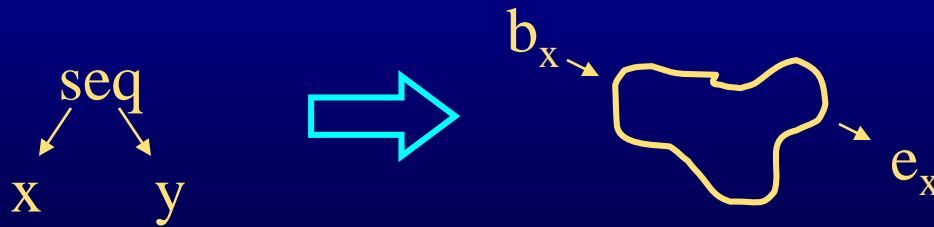
Destructuring Seq Nodes

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: (**b_x**,**e_x**) = destruct(**x**);



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



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

3: next(e_x) = b_y ;



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

3: next(e_x) = b_y ; 4: return (b_x, e_y);



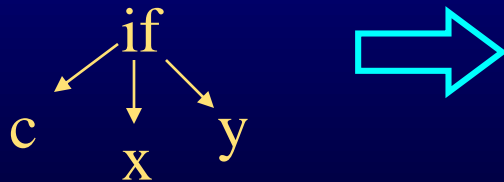
Destructuring If Nodes

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 if **c x y**



Destructuring If Nodes

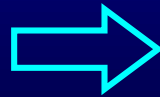
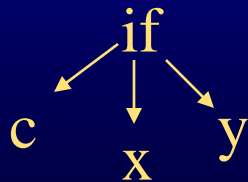
destruct(**n**)

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 if **c** **x** **y**

1: (**b_x**,**e_x**) = destruct(**x**);



Destructuring If Nodes

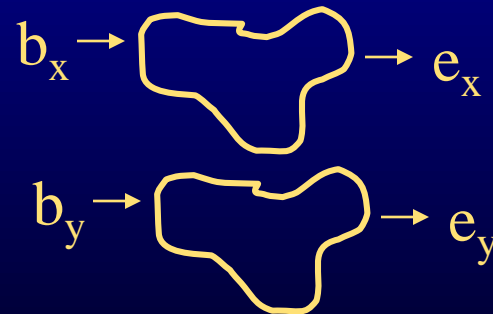
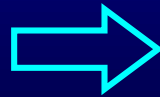
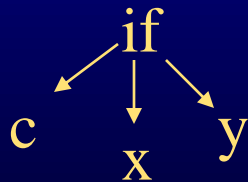
destruct(**n**)

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 if **c** **x** **y**

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



Destructuring If Nodes

destruct(**n**)

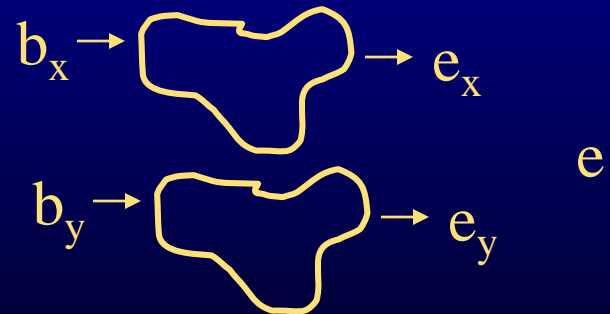
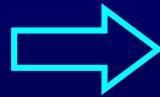
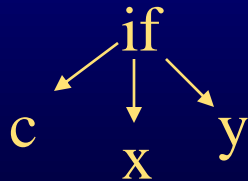
generates lowered form of structured code represented by **n**

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

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

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

3: **e** = new nop;



Destructuring If Nodes

destruct(**n**)

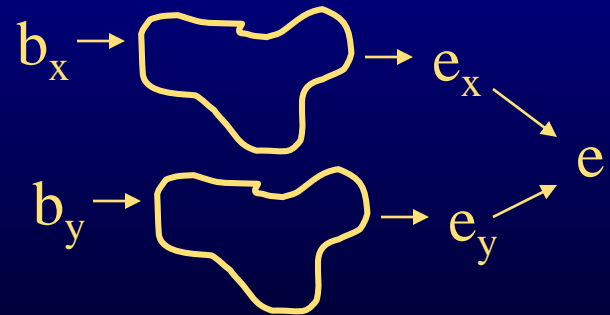
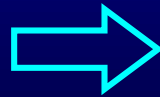
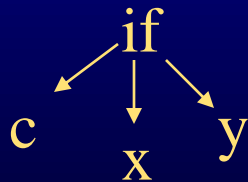
generates lowered form of structured code represented by **n**

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

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

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

3: **e** = new nop; 4: next(e_x) = **e**; 5: next(e_y) = **e**;



Destructuring If Nodes

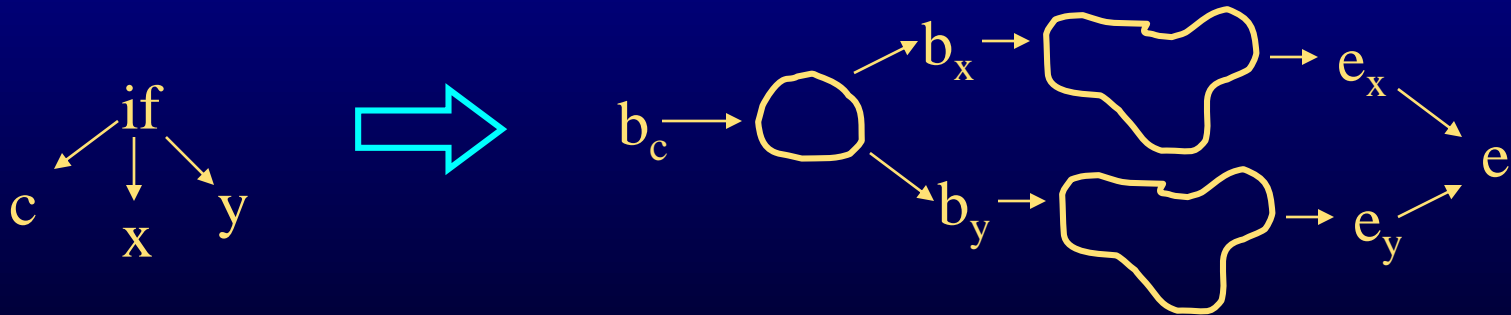
destruct(**n**)

generates lowered form of structured code represented by **n**
returns (**b**,**e**) - **b** is begin node, **e** is end node in destructured form
if **n** is of the form if **c** **x** **y**

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

3: **e** = new nop; 4: next(e_x) = **e**; 5: next(e_y) = **e**;

6: b_c = shortcircuit(**c**, b_x , b_y);

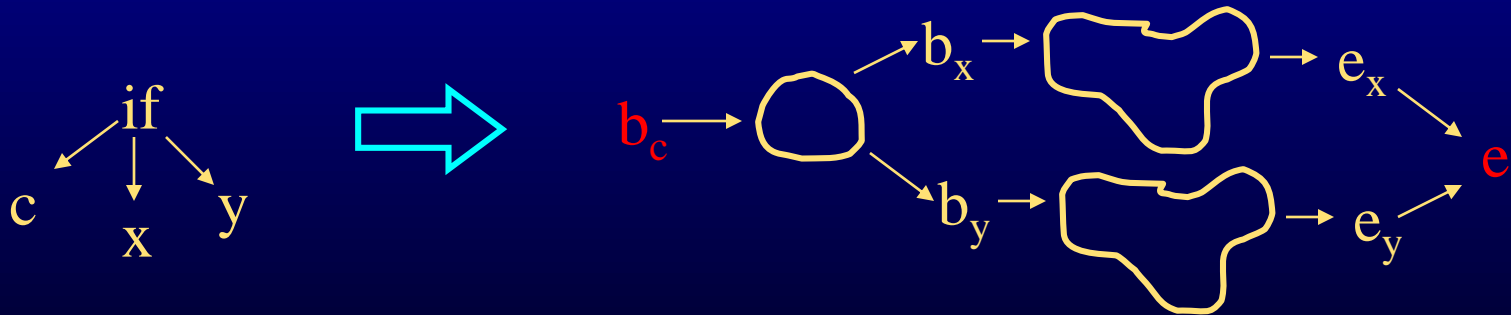


Destructuring If Nodes

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 if **c** **x** **y**

1: (b_x, e_x) = destruct(**x**); 2: (b_y, e_y) = destruct(**y**);
3: **e** = new nop; 4: next(e_x) = **e**; 5: next(e_y) = **e**;
6: b_c = shortcircuit(**c**, b_x , b_y); 7: return (b_c , **e**);



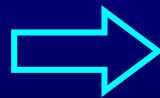
Destructuring While Nodes

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**

while
↙ ↘
c **x**



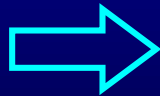
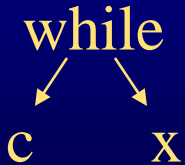
Destructuring While Nodes

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** = new nop;



e

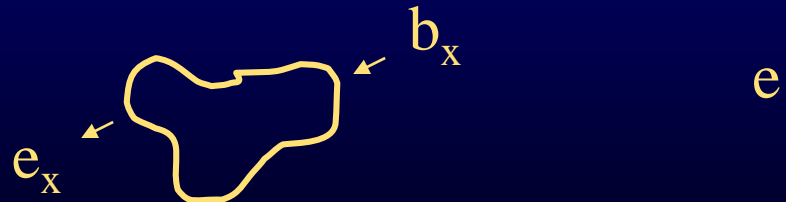
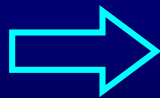
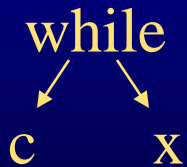
Destructuring While Nodes

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** = new nop; 2: (**b_x**,**e_x**) = destruct(**x**);



Destructuring While Nodes

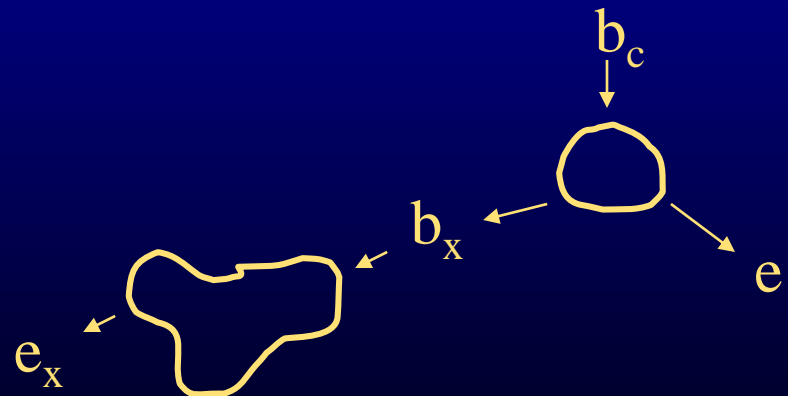
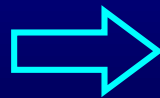
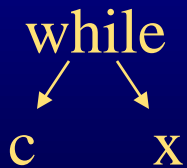
destruct(**n**)

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

returns (**b**,**e**) - **b** is begin node, **e** is end node in destructured form
if **n** is of the form while **c** **x**

1: **e** = new nop; 2: (**b_x**,**e_x**) = destruct(**x**);

3: **b_c** = shortcircuit(**c**, **b_x**, **e**);



Destructuring While Nodes

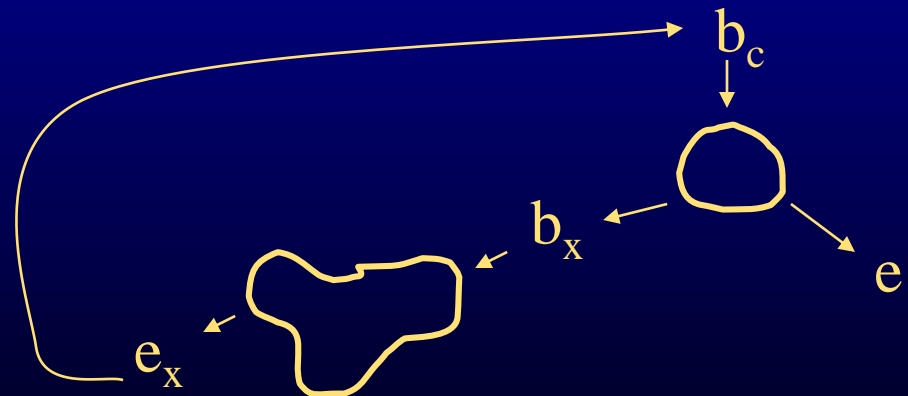
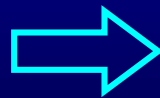
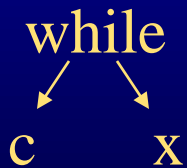
destruct(**n**)

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

returns (**b**,**e**) - **b** is begin node, **e** is end node in destructured form
if **n** is of the form while **c** **x**

1: **e** = new nop; 2: (**b_x**,**e_x**) = destruct(**x**);

3: **b_c** = shortcircuit(**c**, **b_x**, **e**); 4: next(**e_x**) = **b_c**;



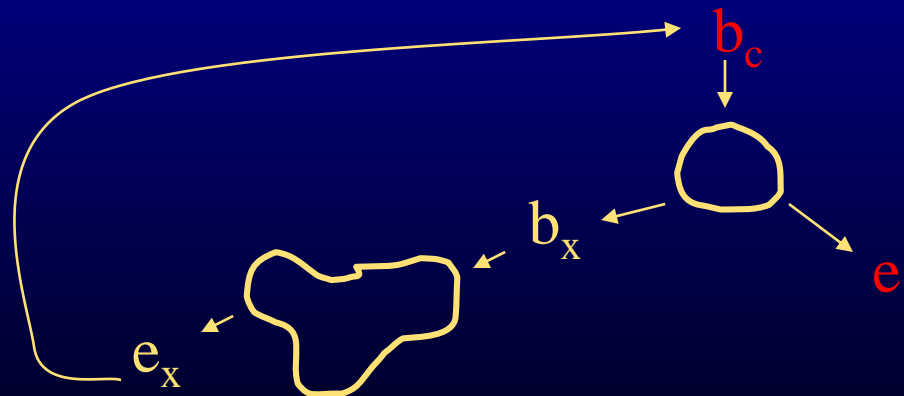
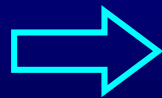
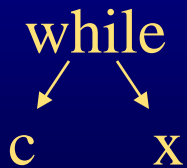
Destructuring While Nodes

destruct(**n**)

generates lowered form of structured code represented by **n**
returns (b,e) - **b** is begin node, **e** is end node in destructured form
if **n** is of the form while c x

1: e = new nop; 2: (b_x, e_x) = destruct(x);

3: b_c = shortcircuit(c, b_x, e); 4: 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`

`c1 && c2` 

Shortcircuiting And Conditions

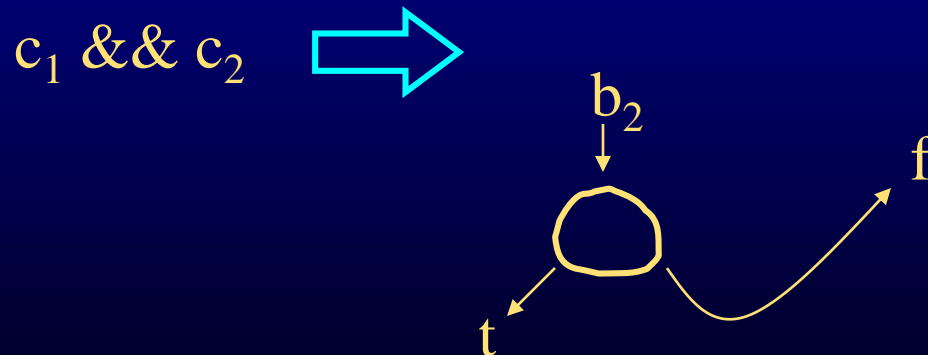
`shortcircuit(c, t, f)`

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);`



Shortcircuiting And Conditions

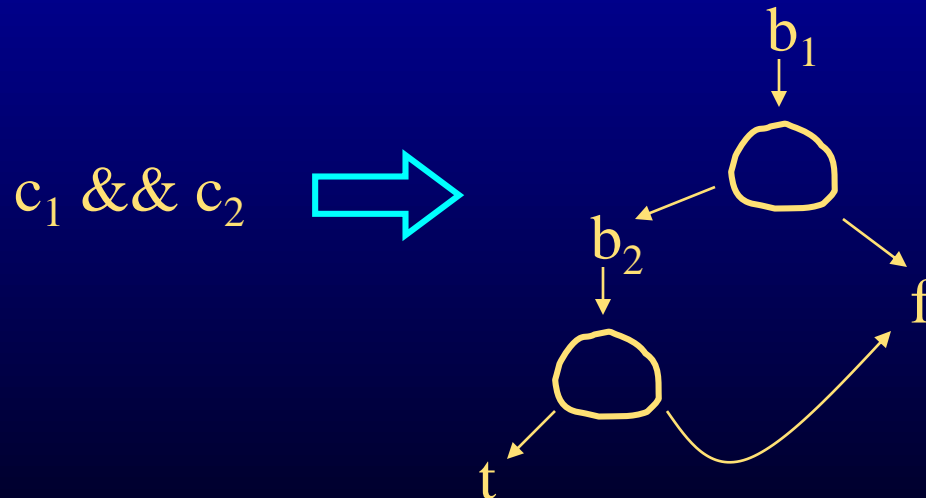
`shortcircuit(c, t, f)`

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)`;



Shortcircuiting And Conditions

`shortcircuit(c, t, f)`

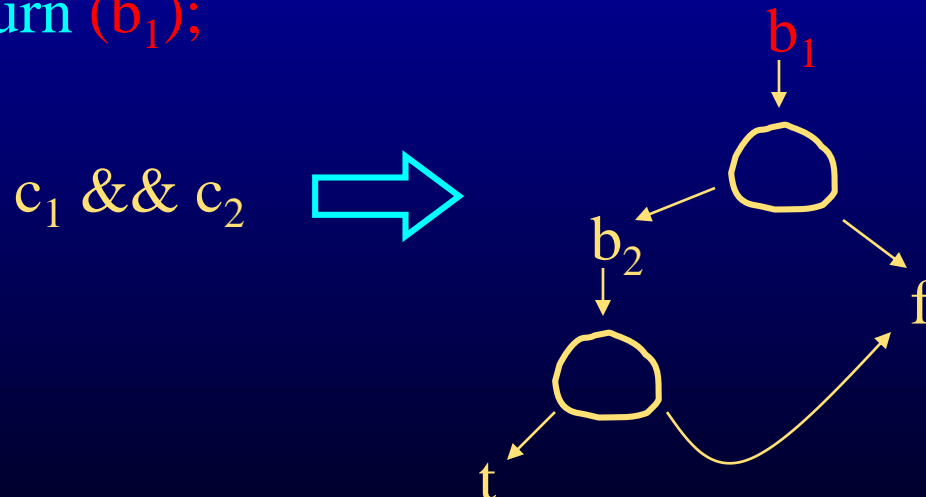
generates shortcircuit form of conditional represented by `c`

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 Or 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`

`c1 || c2` 

Shortcircuiting Or Conditions

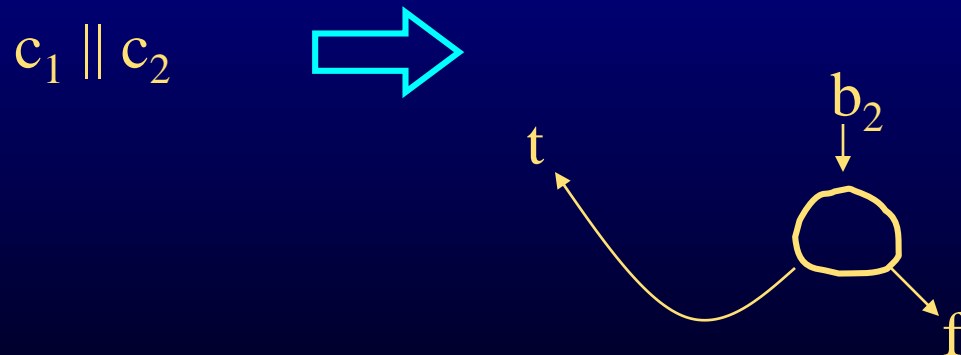
`shortcircuit(c, t, f)`

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);`



Shortcircuiting Or Conditions

`shortcircuit(c, t, f)`

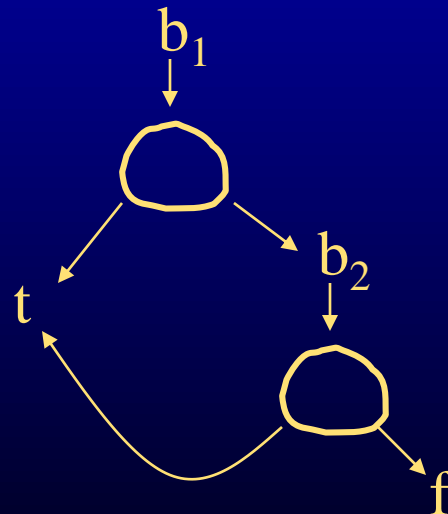
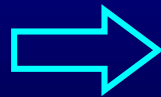
generates shortcircuit form of conditional represented by `c`

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, t, b2);`

`c1 || c2`



Shortcircuiting Or Conditions

`shortcircuit(c, t, f)`

generates shortcircuit form of conditional represented by `c`

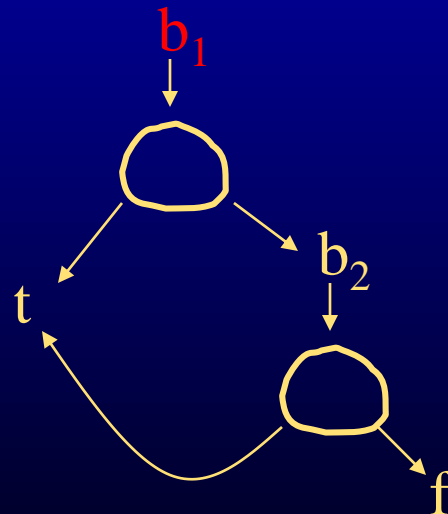
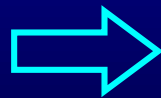
returns `b` - `b` is begin node of shortcircuit form

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

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

3: `return (b1);`

`c1 || c2`



Shortcircuiting Not Conditions

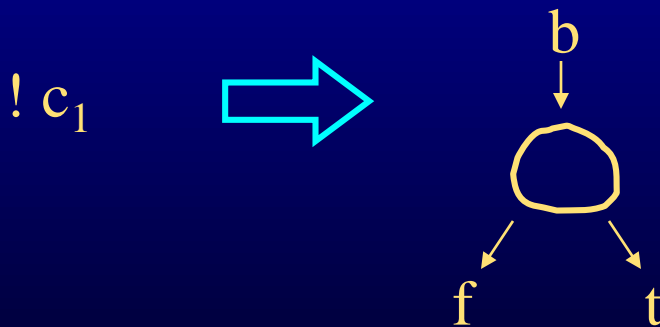
`shortcircuit(c, t, f)`

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

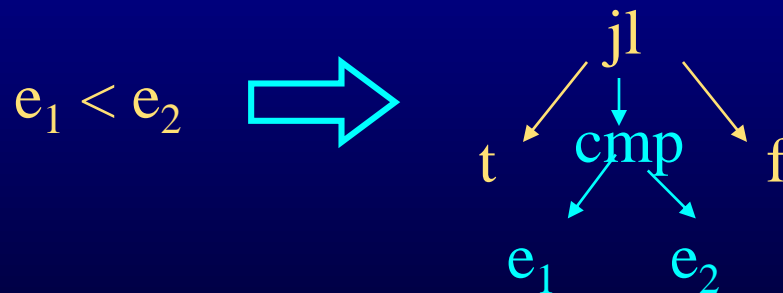
`shortcircuit(c, t, f)`

generates shortcircuit form of conditional represented by `c`

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

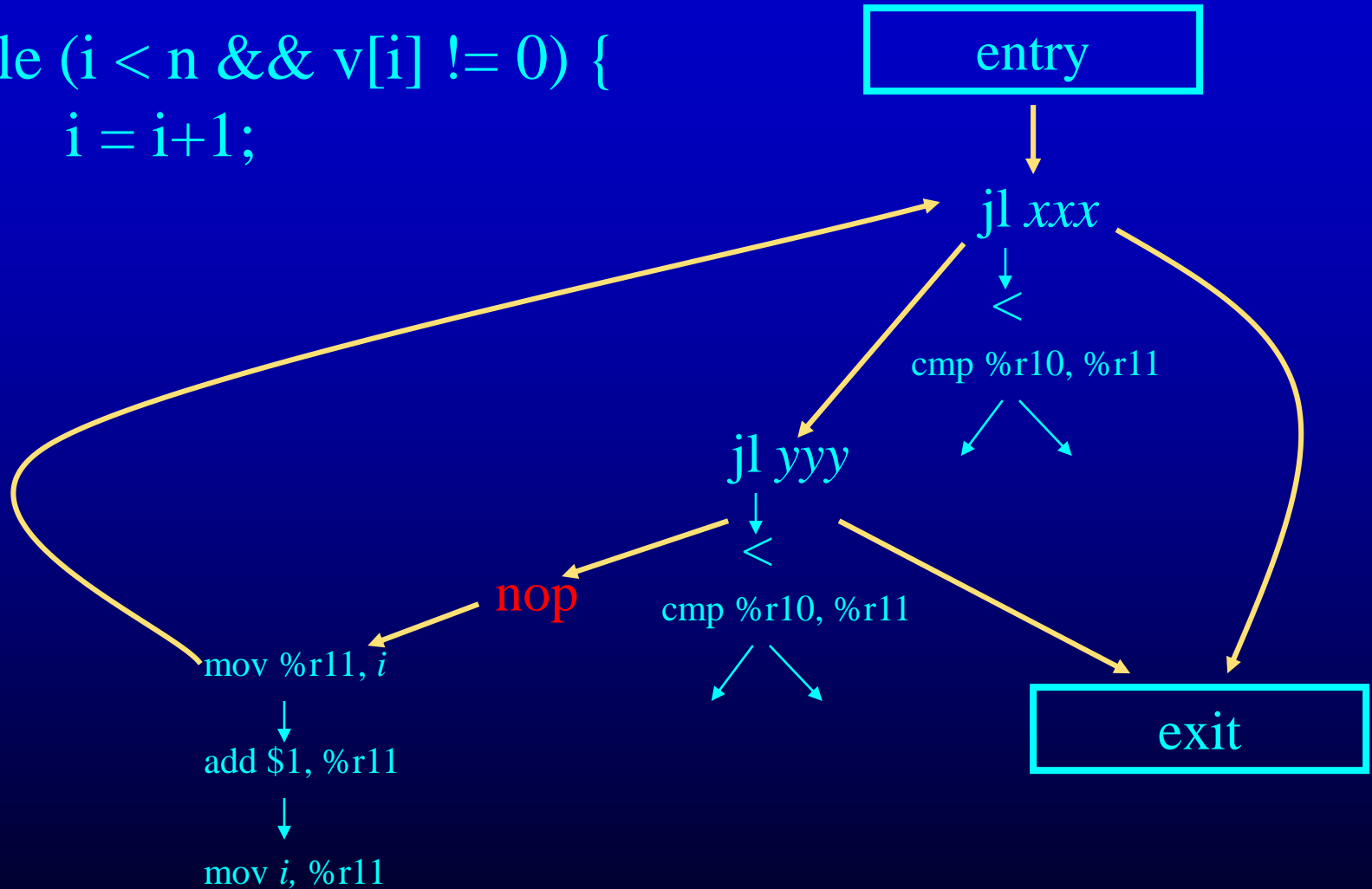
if `c` is of the form $e_1 < e_2$

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

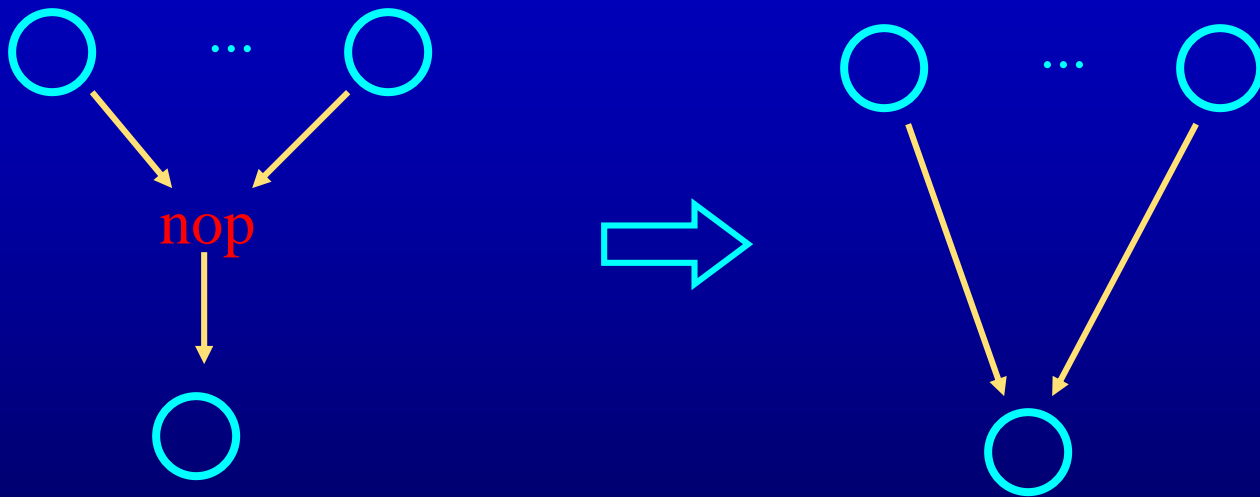


Nops In Destructured Representation

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



Eliminating Nops Via Peephole Optimization

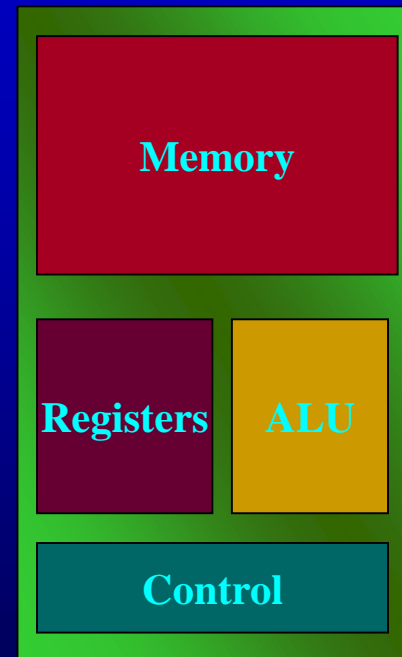


Linearizing CFG to Assembler

- Generate labels for edge targets at branches
 - Labels will correspond to branch targets
 - Can use code generation patterns for this
- Emit code for procedure entry
- Emit code for basic blocks
 - Emit code for statements/conditional expressions
 - Appropriately linearized
 - Jump/conditional jumps link basic blocks together
- Emit code for procedure exit

Overview of a modern ISA

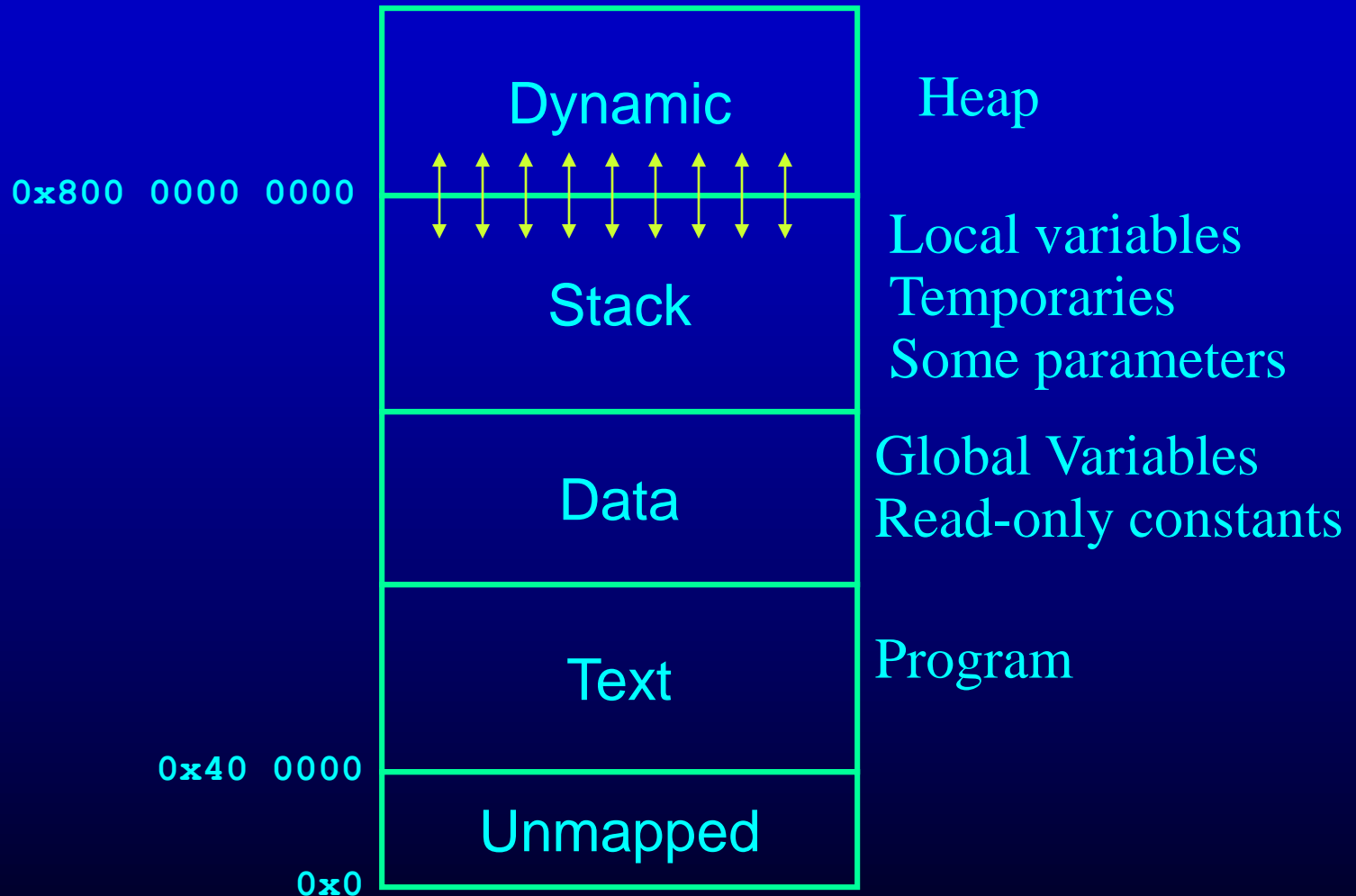
- Memory
- Registers
- ALU
- Control



Overview of Computation

- Loads data from memory into registers
- Computes on registers
- Stores new data back into memory
- Flow of control determines what happens
- Role of compiler:
 - Orchestrate register usage
 - Generate low-level code for interfacing with machine

Typical Memory Layout



Concept of An Object File

- The object file has:
 - Multiple Segments
 - Symbol Information
 - Relocation Information
- Segments
 - Global Offset Table
 - Procedure Linkage Table
 - Text (code)
 - Data
 - Read Only Data
- To run program, OS reads object file, builds executable process in memory, runs process
- We will use assembler to generate object files

Basic Compilation Tasks

- Allocate space for global variables (in data segment)
- For each procedure
 - Allocate space for parameters and locals (on stack)
 - Generate code for procedure
 - Generate procedure entry prolog
 - Generate code for procedure body
 - Generate procedure exit epilog

Generate Code For Procedure Body

- Flatten expressions

- Read program variables into temps before use

- Use temps to have all ops of form

- temp1 = temp2 op temp3

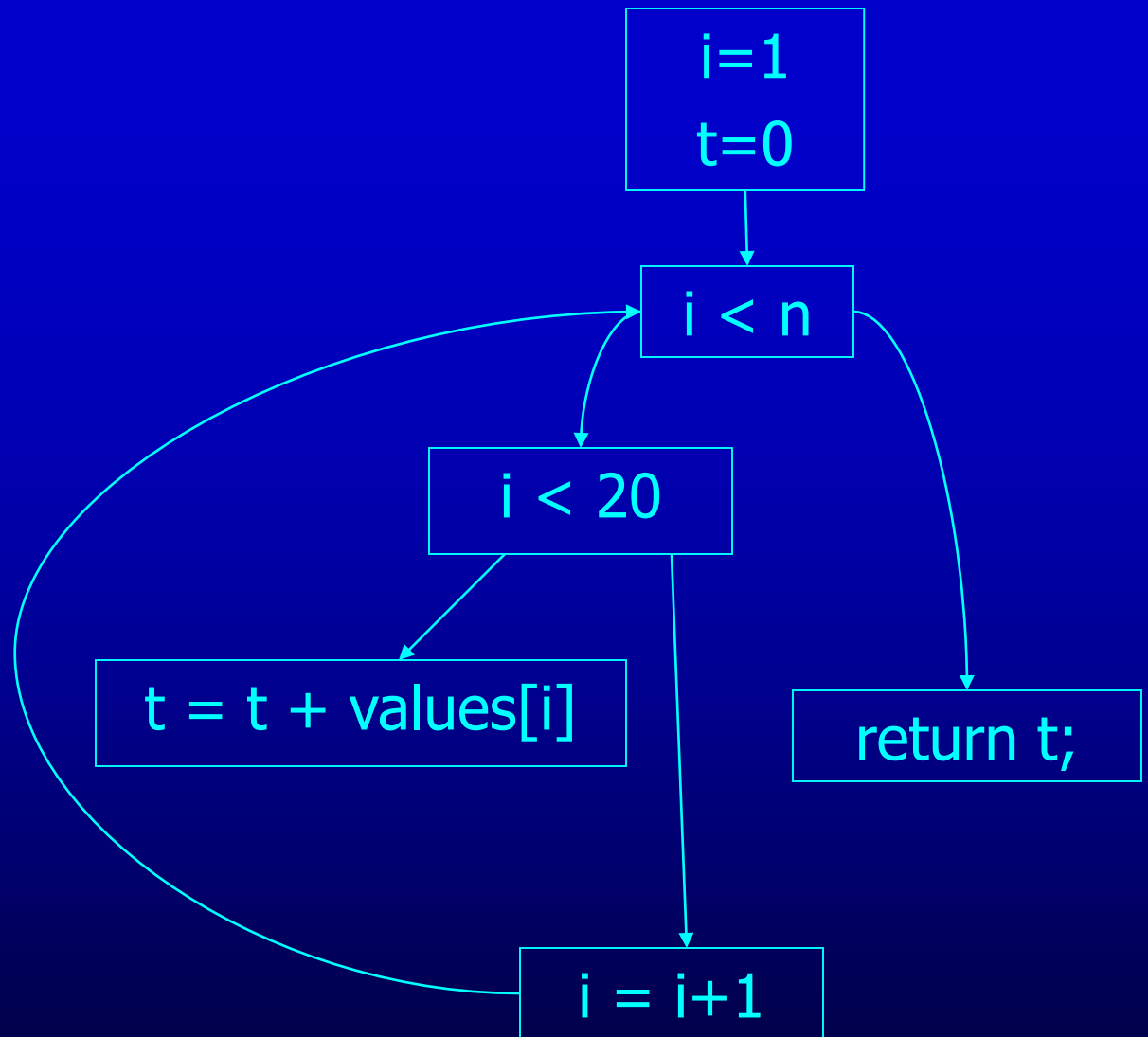
- temp1 = temp2[temp3]

- if (temp1 op temp2)

- while (temp1 op temp2)

- For unoptimized code generation, apply code generation templates/patterns to flattened expressions


```
int values[20];  
int sum(int n) {  
    int i, t;  
    i = 1;  
    t = 0;  
    while (i < n) {  
        if (i < 20) {  
            t = t + values[i];  
        }  
        i = i + 1;  
    }  
    return t;  
}
```



```
int values[20];  
int sum(int n) {  
    int i, t, temp1, temp2, temp3, temp4;  
    i = 0;  
    t = 0;  
    temp1 = n;  
    temp2 = 1;  
    i = temp2;  
    temp2 = 0;  
    t = temp2;  
    temp3 = i;  
    temp4 = temp1;
```

```
    while (temp3 < temp4) {  
        temp3 = i;  
        temp4 = 20;  
        if (temp3 < temp4) {  
            temp3 = t;  
            temp4 = i;  
            temp4 = values[temp4];  
            temp2 = temp3 + temp4;  
            t = temp2;  
        }  
        temp3 = i;  
        temp4 = 1;  
        temp2 = temp3 + temp4;  
        i = temp2;  
    }  
    temp2 = t;  
    return temp2;  
}
```



```

comm    values,160,8
sum:
    //allocate for t, i, temp1, temp2, temp3, temp4
    enter    $48, $0
    movq     %rdi, -24(%rbp)

    //t=0
    movq     $0, -8(%rbp)

    //i=0
    movq     $0, -16(%rbp)

    //i = temp2 = 1
    movq     $1, -32(%rbp)
    mov     -32(%rbp), %rax
    movq     %rax, -16(%rbp)

```

```

    //t = temp2 = 0
    movq     $0, -32(%rbp)    //set temp2 to 0
    mov     -32(%rbp), %rax    //store temp2 in %rax
    movq     %rax, -8(%rbp)    //load %rax to t

```

.BasicBlock2:

```

    //i < n

    //temp3 = i
    mov     -16(%rbp), %rax
    movq     %rax, -40(%rbp)

    //temp4 = temp1
    mov     -24(%rbp), %rax
    movq     %rax, -48(%rbp)

```

```

    //temp3 < temp4
    mov     -48(%rbp), %rax
    cmp     %rax, -40(%rbp)
    jge     .BasicBlock4

```

.BasicBlock3:

```

    movq     $1, -32(%rbp)    //temp2 = true
    jmp     .BasicBlock5    //jump to condition

```

.BasicBlock4:

```

    movq     $0, -32(%rbp)    //temp2 = false

```

.BasicBlock5:

```

    cmp     $1, -32(%rbp)    //if temp2 is true continue, false
    jump to return
    jne     .BasicBlock12

```

.BasicBlock6:

```

    //i < 20

    //temp3 = i
    mov     -16(%rbp), %rax
    movq     %rax, -40(%rbp)

    //temp4 = 20
    movq     $20, -48(%rbp)

    //temp3 < temp4
    mov     -48(%rbp), %rax
    cmp     %rax, -40(%rbp)
    jge     .BasicBlock8

```

.BasicBlock7:

```

    movq     $1, -32(%rbp)    //temp2 = true
    jmp     .BasicBlock9    //jump to condition

```

.BasicBlock8:

```

    movq     $0, -32(%rbp)    //temp2 = false

```

.BasicBlock9:

```

    cmp     $1, -32(%rbp)    //if temp2 is true fo in block,
    false skip
    jne     .BasicBlock11

```

.BasicBlock10:

```

    //temp3 = t
    mov     -8(%rbp), %rax
    movq     %rax, -40(%rbp)

```

```

    //temp4 = i
    mov     -16(%rbp), %rax
    movq     %rax, -48(%rbp)

```

```

    cmp     $0, -48(%rbp)    //check if array index temp4 < 0
    jl      .boundsbad0
    mov     -48(%rbp), %rax
    cmp     $20, %rax        //check if array index temp4 >=
    20

```

```

    jge     .boundsbad0
    jmp     .boundsgood0    //perform array access

```

.boundsbad0:

```

    mov     -48(%rbp), %rdx
    mov     $8, %rcx
    call    .boundserior

```

...boundsgood0:

```

    //t = t + values[i] = temp3 + values[temp4]

    //array access
    mov     -48(%rbp), %r10
    mov     values(, %r10, 8), %rax
    movq     %rax, -48(%rbp)

```

```

    //temp2 = temp3 + temp4
    mov     -40(%rbp), %rax
    add     -48(%rbp), %rax
    movq     %rax, -32(%rbp)

```

```

    //t = temp2
    mov     -32(%rbp), %rax
    movq     %rax, -8(%rbp)

```

.BasicBlock11:

```

    //i = i + 1

    //temp3 = i
    mov     -16(%rbp), %rax
    movq     %rax, -40(%rbp)

```

```

    //temp4 = 1
    movq     $1, -48(%rbp)

```

```

    //temp2 = temp3 + temp4
    mov     -40(%rbp), %rax
    add     -48(%rbp), %rax
    movq     %rax, -32(%rbp)

```

```

    //i = temp2
    mov     -32(%rbp), %rax
    movq     %rax, -16(%rbp)

```

```

    jmp     .BasicBlock2    //jump to beginning of while
    loop

```

.BasicBlock12:

```

    //return t

    //temp2 = t
    mov     -8(%rbp), %rax
    movq     %rax, -32(%rbp)

    //return temp2
    mov     -32(%rbp), %rax
    leave
    ret

```

Patterns for Unoptimized Generated Code

```
// temp3 = i
```

```
    mov    -16(%rbp), %rax
```

```
    movq   %rax, -40(%rbp)
```

```
// temp2 = temp3 + temp4
```

```
    mov    -40(%rbp), %rax
```

```
    add    -48(%rbp), %rax
```

```
    movq   %rax, -32(%rbp)
```

```
// temp4 = values[temp4]
```

```
    mov    -48(%rbp), %r10
```

```
    mov    values(, %r10, 8), %rax
```

```
    movq   %rax, -48(%rbp)
```

Code for If

```
// if (x >= 0) { then code } else { else code }
```

```
cmp    $0, -48(%rbp) // check if x < 0
```

```
jl     .elsebranch0
```

```
    ... then code
```

```
    jmp .done0
```

```
.elsebranch0:
```

```
    ... else code
```

```
.done0
```

Array Bounds Check Code

```

cmp    $0, -48(%rbp)    //check if array index temp4 < 0
jl     .boundsbad0
mov     -48(%rbp), %rax
cmp     $20, %rax       //check if array index temp4 >= 20
jge     .boundsbad0
jmp     .boundsgood0    //perform array access

.boundsbad0:
mov     -48(%rbp), %rdx
mov     $8, %rcx
call    .boundserror

.boundsgood0

```

Allocate space for global variables

Decaf global array declaration

```
int values[20];
```

Assembler directive (reserve space in data segment)

```
.comm values,160,8
```



The diagram illustrates the components of the assembler directive `.comm values,160,8`. Three yellow arrows point from labels below to the corresponding parts of the directive: an arrow from **Name** points to `values`, an arrow from **Size** points to `160`, and an arrow from **Alignment** points to `8`.

Name **Size** **Alignment**

The Call Stack

- Arguments 1 to 6 are in:
 - %rdi, %rsi, %rdx,
 - %rcx, %r8, and %r9

%rbp

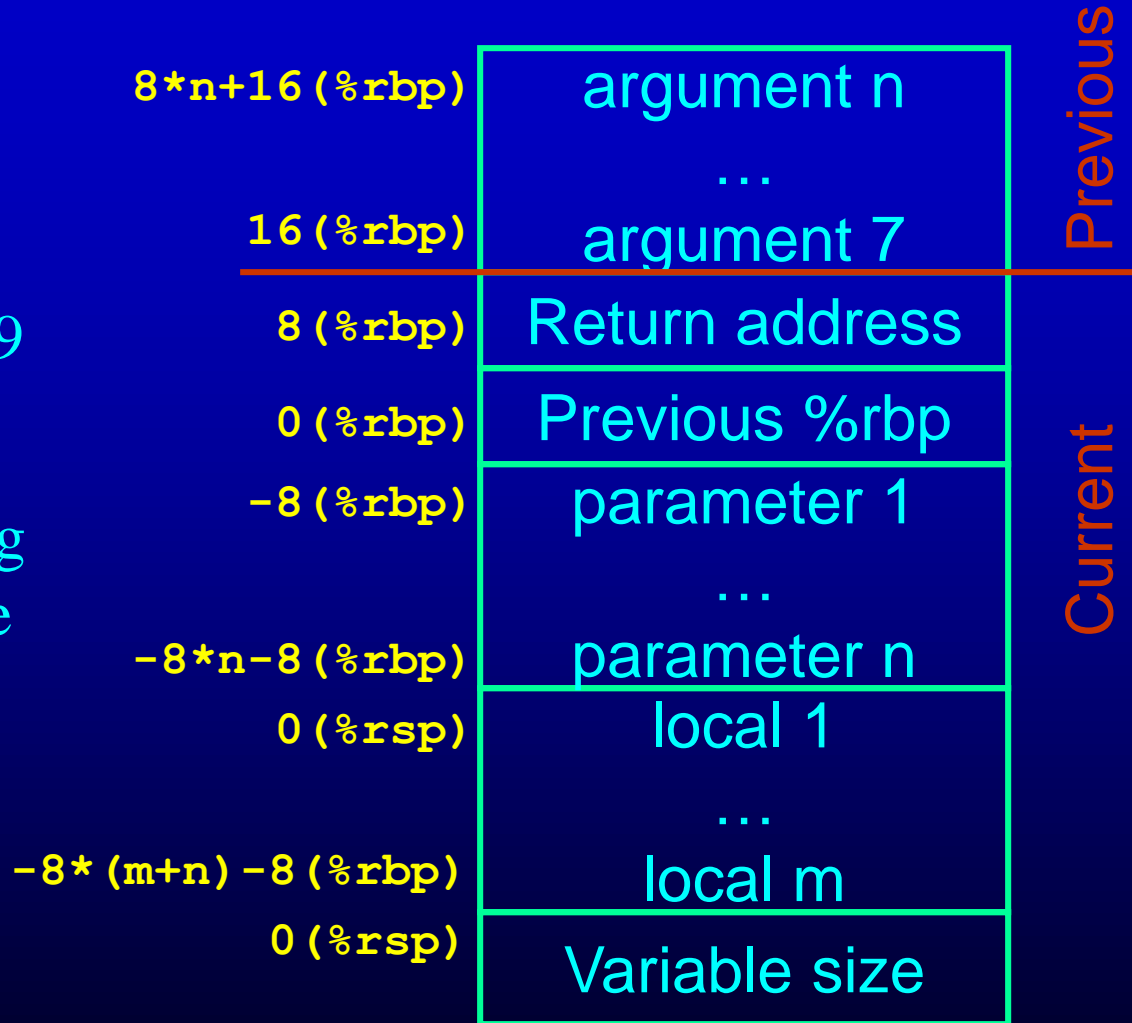
- marks the beginning of the current frame

%rsp

- marks top of stack

%rax

- return value



Questions

- Why allocate activation records on a stack?
- Why not statically preallocate activation records?
- Why not dynamically allocate activation records in the heap?

Allocate space for parameters/locals

- Each parameter/local has its own slot on stack
- Each slot accessed via %rbp negative offset
- Iterate over parameter/local descriptors
- Assign a slot to each parameter/local

Generate procedure entry prologue

- Push base pointer (%rbp) onto stack
- Copy stack pointer (%rsp) to base pointer (%rbp)
- Decrease stack pointer by activation record size
- All done by:
 - enter <stack frame size in bytes>, <lexical nesting level>
 - enter \$48, \$0
- For now (will optimize later) move parameters to slots in activation record (top of call stack)
 - movq %rdi, -24(%rbp)

x86 Register Usage

- 64 bit registers (16 of them)
 - `%rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp, %r8-%r15`
- Stack pointer `%rsp`, base pointer `%rbp`
- Parameters
 - First six integer/pointer parameters in `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
 - Rest passed on the stack
- Return value
 - 64 bits or less in `%rax`
 - Longer return values passed on the stack

Questions

- Why have %rbp if also have %rsp?
- Why not pass all parameters in registers?
- Why not pass all parameters on stack?
- Why not pass return value in register(s) regardless of size?
- Why not pass return value on stack regardless of size?

Callee vs caller save registers

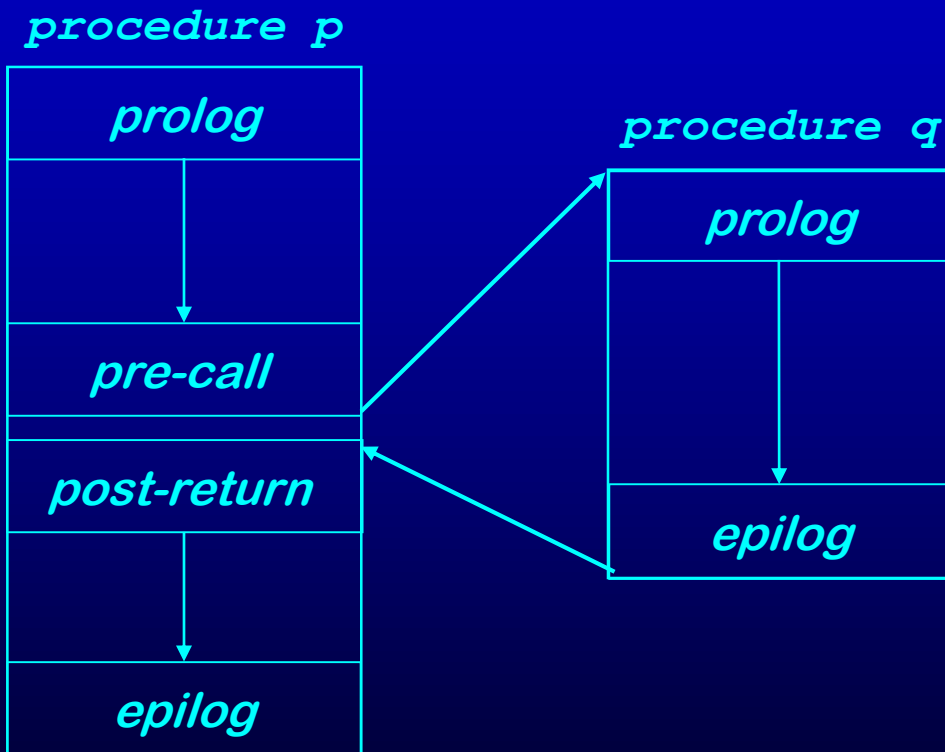
- Registers used to compute values in procedure
- Should registers have same value after procedure as before procedure?
 - Callee save registers (must have same value)
`%rsp, %rbx, %rbp, %r12-%r15`
 - Caller save registers (procedure can change value)
`%rax, %rcx, %rdx, %rsi, %rdi, %r8-%r11`
- Why have both kinds of registers?

Generate procedure call epilogue

- Put return value in %rax
 - mov -32(%rbp), %rax
- Undo procedure call
 - Move base pointer (%rbp) to stack pointer (%rsp)
 - Pop base pointer from caller off stack into %rbp
 - Return to caller (return address on stack)
 - All done by
 - leave
 - ret

Procedure Linkage

Standard procedure linkage



Pre-call:

- Save caller-saved registers
- Set up arguments
 - Registers (1-6)
 - Stack (7-N)

Prolog:

- Push old frame pointer
- Save callee-saved registers
- Make room for parameters, temporaries, and locals

Epilog:

- Restore callee-saved registers
- Pop old frame pointer
- Store return value

Post-return:

- Restore caller-saved registers
- Pop arguments

Generate code for procedure body

Evaluate expressions with a temp for each subexpression

//i = i + 1

//temp3 = i

mov i from stack, %rax

movq %rax, temp3 on stack

Temps stored on stack

//temp4 = 1

mov \$1, temp4 on stack

%rax as working register

//temp2 = temp3 + temp4

mov temp3 from stack, %rax

add temp4 on stack, %rax

movq %rax, temp2 on stack

Apply code generation templates

temp = var

temp = temp op temp

var = temp

//i = temp2

mov temp2 on stack, %rax

movq %rax, i on stack

Generate code for procedure body

Evaluate expressions with a temp for each subexpression

```
//i = i + 1
```

```
//temp3 = i
```

```
mov    -16(%rbp), %rax
```

```
movq   %rax, -40(%rbp)
```

Temps stored on stack

```
//temp4 = 1
```

```
mov    $1, -48(%rbp)
```

%rax as working register

```
//temp2 = temp3 + temp4
```

```
mov    -40(%rbp), %rax
```

```
add    -48(%rbp), %rax
```

```
movq   %rax, -32(%rbp)
```

Apply code generation templates

temp = var

temp = temp op temp

var = temp

```
//i = temp2
```

```
mov    -32(%rbp), %rax
```

```
movq   %rax, -16(%rbp)
```

Evaluating Expression Trees

Flat List Model

- The idea is to linearize the expression tree
- Left to Right Depth-First Traversal of the expression tree
 - Allocate temporaries for intermediates (all the nodes of the tree)
 - New temporary for each intermediate
 - All the temporaries on the stack (for now)
- Each expression is a single 3-addr op
 - $x = y \text{ op } z$
 - Code generation for the 3-addr expression
 - Load y into register %rax
 - Perform op z, %rax
 - Store %rax to x

Another option

Load y into register %rax
Load z into register %r10
Perform op %r10, %rax
Store %rax to x

Issues in Lowering Expressions

- Map intermediates to registers?
 - registers are limited
 - When the tree is large, registers may be insufficient \Rightarrow allocate space in the stack
- Very inefficient
 - too many copies
 - don't worry, we'll take care of them in the optimization passes
 - keep the code generator very simple

Generate code for procedure body

Basic Ideas

- Temps, locals, parameters all have a “home” on stack
- When compute, use %rax as working storage
- All subexpressions are computed into temps
- For each computation in expression
 - Fetch first operand (on stack) into %rax
 - Apply operator to second operand (on stack) and %rax
 - Result goes back into %rax
 - Store result (in %rax) back onto stack

Generate code for procedure body

Accessing an array element

```
//array access temp1 = values[temp0]
```

```
mov    array index in temp0, %r10
```

```
mov    values[array index in %r10], %rax
```

```
movq   %rax, temp1
```

%r10 as array index register

%rax as working register

Apply code generation template

Generate code for procedure body

Accessing an array element

```
//array access temp1 = values[temp0]
```

```
mov    -48(%rbp), %r10
```

```
mov    values(, %r10, 8), %rax
```

```
movq   %rax, -48(%rbp)
```

%r10 as array index register

%rax as working register

Apply code generation template

Generate code for procedure body

Array bounds checks (performed before array access)

check if array index < 0

```
jl    .boundsbad0
```

check if array index \geq array bound

```
jge    .boundsbad0
```

```
jmp     .boundsgood0    //perform array access
```

.boundsbad0:

first parameter is array index

second parameter is array element size

```
call    .boundSError
```

.boundsgood0:

perform array access

Generate code for procedure body

Array bounds checks (performed before array access)

```
cmp    $0, -48(%rbp)  //check if array index temp4 < 0
jl     .boundsbad0
mov     -48(%rbp), %rax
cmp     $20, %rax      //check if array index temp4 >= 20
jge     .boundsbad0
jmp     .boundsgood0   //perform array access
```

.boundsbad0:

```
mov     -48(%rbp), %rdx    %rax as working register
mov     $8, %rcx           Apply code generation template
call    .boundserror
```

.boundsgood0: //array access to values[temp4]

```
mov     -48(%rbp), %r10
mov     values(, %r10, 8), %rax
movq    %rax, -48(%rbp)
```

Generate code for procedure body

Control Flow via comparisons and jumps

```
//if (condition) { code } else { code }
```

```
    compute condition
```

```
    if condition not true to jump to .FalseCase
```

```
.TrueCase:
```

```
    // code for true case
```

```
    jmp .EndIf // skip else case
```

```
.FalseCase:
```

```
    // code for else case
```

```
.EndIf:
```

```
    // code for after if
```

Code generation template for
if then else (conditional branch)

Generate code for procedure body

Control Flow via comparisons and jumps

```
//if (condition) { code } else { code }  
    compute condition  
    if condition not true to jump to .ConditionFalse  
.ConditionTrue:  
    set temp=1 (true)  
    jmp    .CheckCondition //jump to check condition  
.ConditionFalse:  
    set temp = 0 (false)  
.CheckCondition:  
    check if temp is 1 (true) or 0 (false)  
    if temp is 0 (false) jump to .FalseCase  
.TrueCase:  
    // code for true case  
    jmp .EndIf // skip else case  
.FalseCase:  
    // code for else case  
.EndIf: // continuation after if
```

Code generation template for
if then else (conditional branch)
Stores condition explicitly, may
be more debuggable

Generate code for procedure body

Control Flow via comparisons and jumps

```
//if (temp3 < temp4)
```

```
    mov    -48(%rbp), %rax
```

```
    cmp    %rax, -40(%rbp)
```

```
    jge    .BasicBlock8
```

%rax as working register

Apply code generation template

```
.BasicBlock7:
```

```
    movq    $1, -32(%rbp) //temp2 = true
```

```
    jmp     .BasicBlock9 //jump to condition
```

```
.BasicBlock8:
```

```
    movq    $0, -32(%rbp) //temp2 = false
```

```
.BasicBlock9:
```

```
    cmp     $1, -32(%rbp) //if temp2 is true fall through, if false jump to false case
```

```
    jne     .BasicBlock11
```

```
.BasicBlock10:
```

```
    // code for true (then) case
```

```
    jmp     .BasicBlock12 // skip else case
```

```
.BasicBlock11:
```

```
    // code for false (else) case
```

```
.BasicBlock12: // continuation after if
```

Code For Conditional Branch in CFG

- Each basic block has a label
- Each conditional branch in CFG has
 - True edge (goes to basic block with label LT)
 - False edge (goes to basic block with label LF)
- Emitted code for CFG tests condition
 - If true, jump to LT
 - If false, jump to LF
- Emit all basic blocks (in some order), jumps link everything together

Quick Peephole Optimization

- Emitted code can look something like:
 jmp .BasicBlock0
 .BasicBlock0:
• In this case can remove jmp instruction

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.
- Write sanity checks, consistency checks, use often

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 , delta debugging

Machine Code Generator Should...

- Translate all the instructions in the intermediate representation to assembly language
- Allocate space for the variables, arrays etc.
- Adhere to calling conventions
- Create the necessary symbolic information

Machines understand...

LOCATION	DATA
0046	8B45FC
0049	4863F0
004c	8B45FC
004f	4863D0
0052	8B45FC
0055	4898
0057	8B048500
	000000
005e	8B149500
	000000
0065	01C2
0067	8B45FC
006a	4898
006c	89D7
006e	033C8500
	000000
0075	8B45FC
0078	4863C8
007b	8B45F8
007e	4898
0080	8B148500

Machines understand...

LOCATION	DATA	ASSEMBLY INSTRUCTION
0046	8B45FC	movl -4(%rbp), %eax
0049	4863F0	movslq %eax,%rsi
004c	8B45FC	movl -4(%rbp), %eax
004f	4863D0	movslq %eax,%rdx
0052	8B45FC	movl -4(%rbp), %eax
0055	4898	cltq
0057	8B048500	movl B(,%rax,4), %eax
	000000	
005e	8B149500	movl A(,%rdx,4), %edx
	000000	
0065	01C2	addl %eax, %edx
0067	8B45FC	movl -4(%rbp), %eax
006a	4898	cltq
006c	89D7	movl %edx, %edi
006e	033C8500	addl C(,%rax,4), %edi
	000000	
0075	8B45FC	movl -4(%rbp), %eax
0078	4863C8	movslq %eax,%rcx
007b	8B45F8	movl -8(%rbp), %eax
007e	4898	cltq
0080	8B148500	movl B(,%rax,4), %edx

Assembly language

- Advantages

- Simplifies code generation due to use of symbolic instructions and symbolic names
- Logical abstraction layer
- Multiple Architectures can describe by a single assembly language
 - ⇒ can modify the implementation
 - macro assembly instructions

- Disadvantages

- Additional process of assembling and linking
- Assembler adds overhead

Assembly language

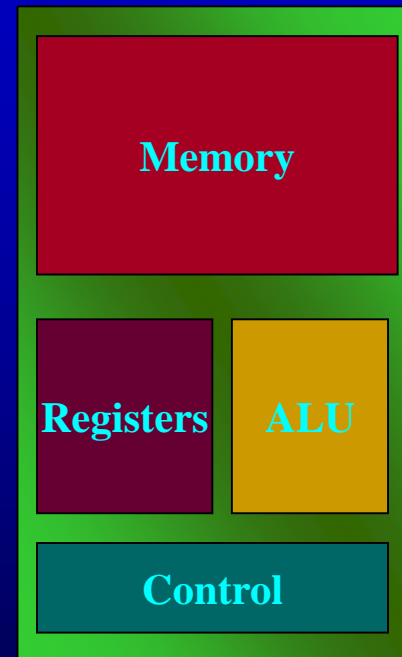
- Relocatable machine language (object modules)
 - all locations(addresses) represented by symbols
 - Mapped to memory addresses at link and load time
 - Flexibility of separate compilation
- Absolute machine language
 - addresses are hard-coded
 - simple and straightforward implementation
 - inflexible -- hard to reload generated code
 - Used in interrupt handlers and device drivers

Concept of An Object File

- The object file has:
 - Multiple Segments
 - Symbol Information
 - Relocation Information
- Segments
 - Global Offset Table
 - Procedure Linkage Table
 - Text (code)
 - Data
 - Read Only Data
- To run program, OS reads object file, builds executable process in memory, runs process
- We will use assembler to generate object files

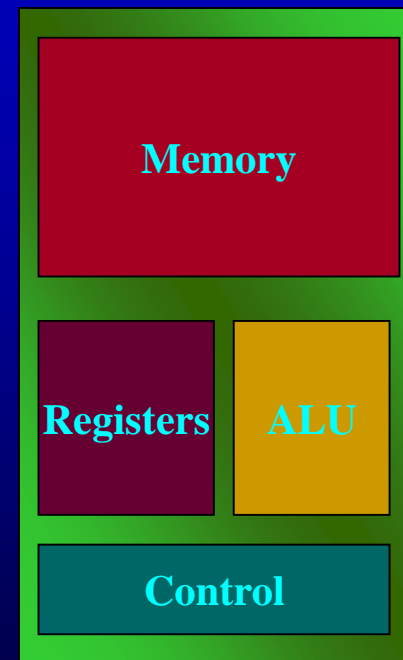
Overview of a modern ISA

- Memory
- Registers
- ALU
- Control

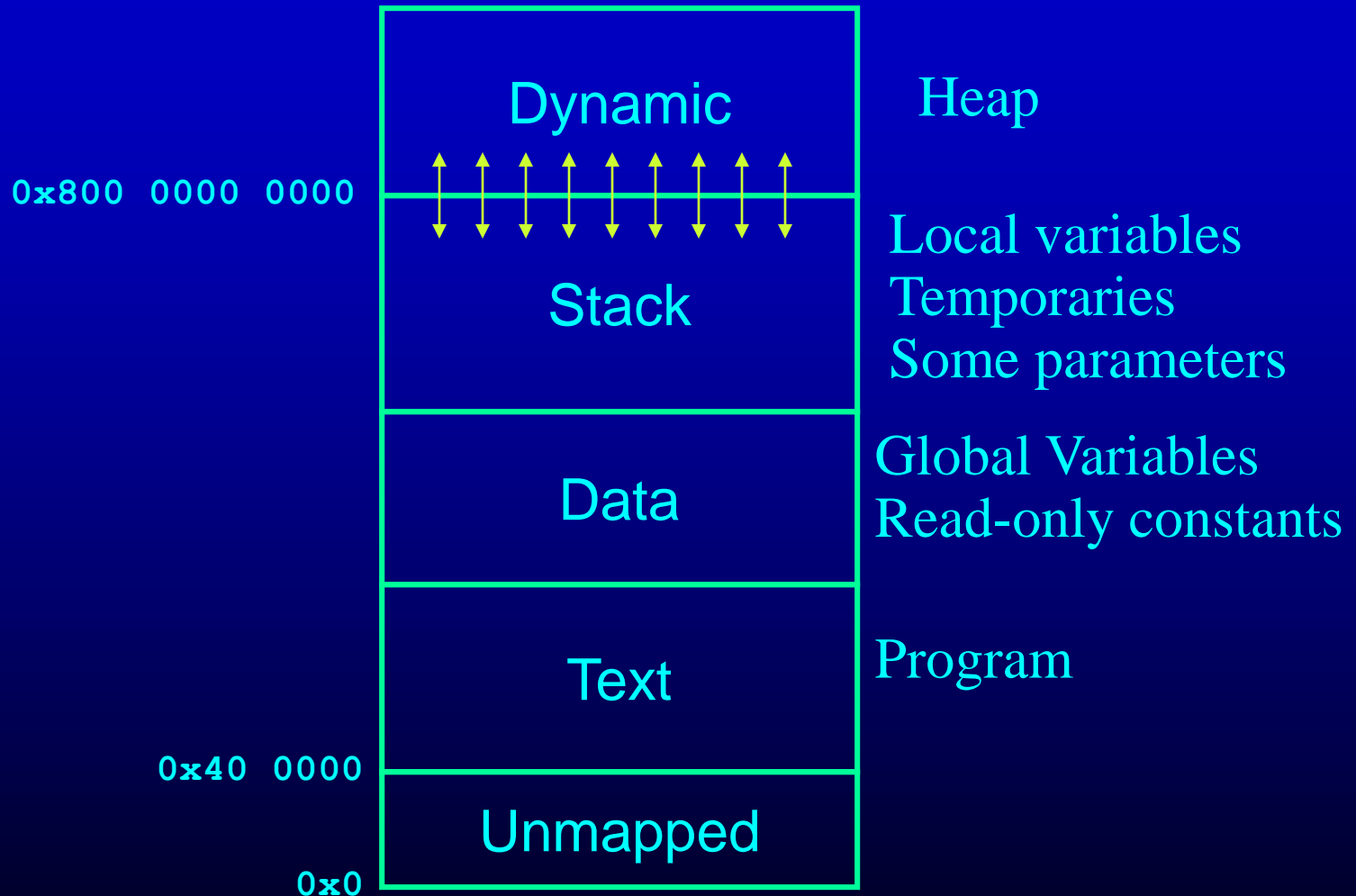


From IR to Assembly

- Data Placement and Layout
 - Global variables
 - Constants (strings, numbers)
 - Object fields
 - Parameters, local variables
 - Temporaries
- Code
 - Read and write data
 - Compute
 - Flow of control



Typical Memory Layout



Global Variables

C

```
struct { int x, y; double z; } b;
```

```
int g;
```

```
int a[10];
```

Assembler directives (reserve space in data segment)

```
.comm _a,40,4          ## @a
```

```
.comm _b,16,3          ## @b
```

```
.comm _g,4,2           ## @g
```

Name **Size** **Alignment**

Addresses

Reserve Memory

```
.comm _a,40,4          ## @a  
.comm _b,16,3          ## @b  
.comm _g,4,2           ## @g
```

Define 3 constants

```
_a – address of a in data segment  
_b – address of b in data segment  
_g – address of g in data segment
```

Struct and Array Layout

- `struct { int x, y; double z; } b;`
 - Bytes 0-1: x
 - Bytes 2-3: y
 - Bytes 4-7: z
- `int a[10]`
 - Bytes 0-1: a[0]
 - Bytes 2-3: a[1]
 - ...
 - Bytes 18-19: a[9]

Dynamic Memory Allocation

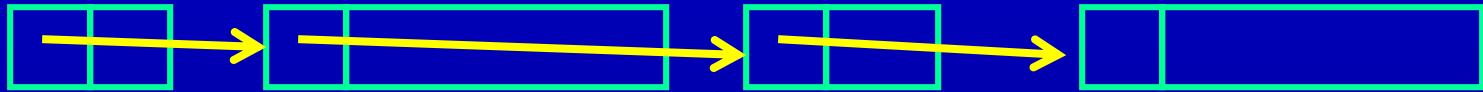
```
typedef struct { int x, y; } PointStruct, *Point;  
Point p = malloc(sizeof(PointStruct));
```

What does allocator do?

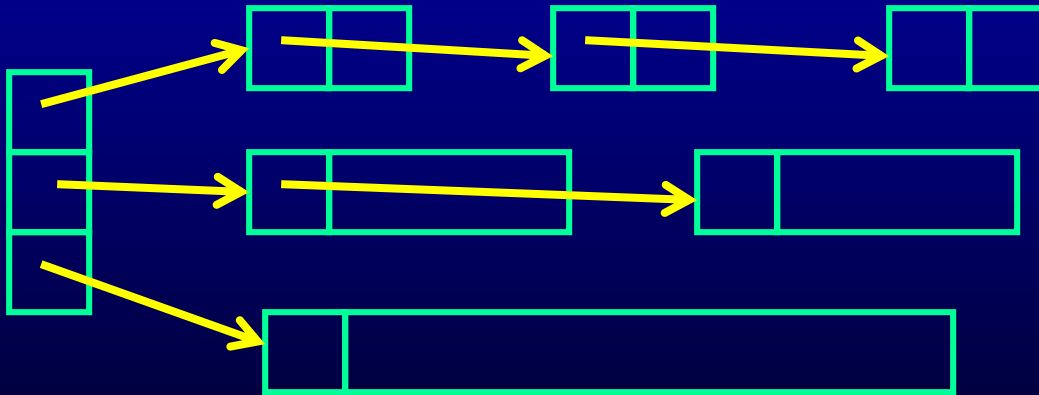
- returns next free big enough data block in heap
- appropriately adjusts heap data structures

Some Heap Data Structures

- Free List (arrows are addresses)

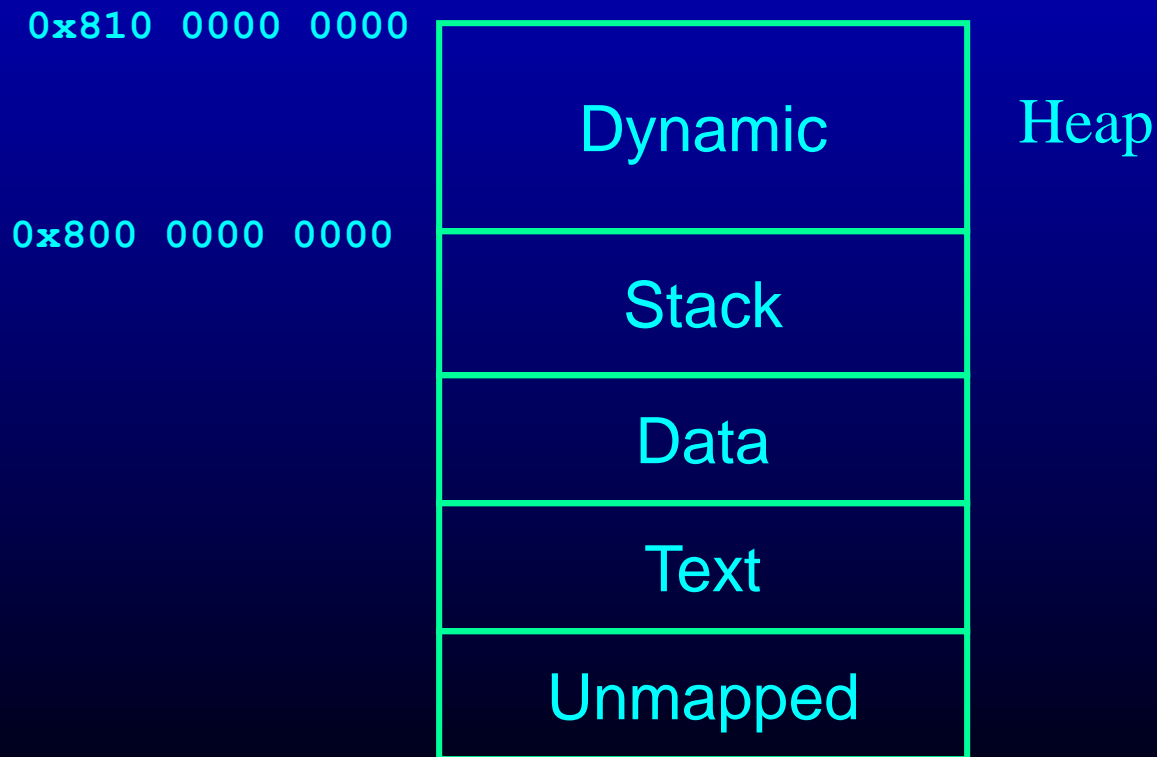


- Powers of Two Lists



Getting More Heap Memory

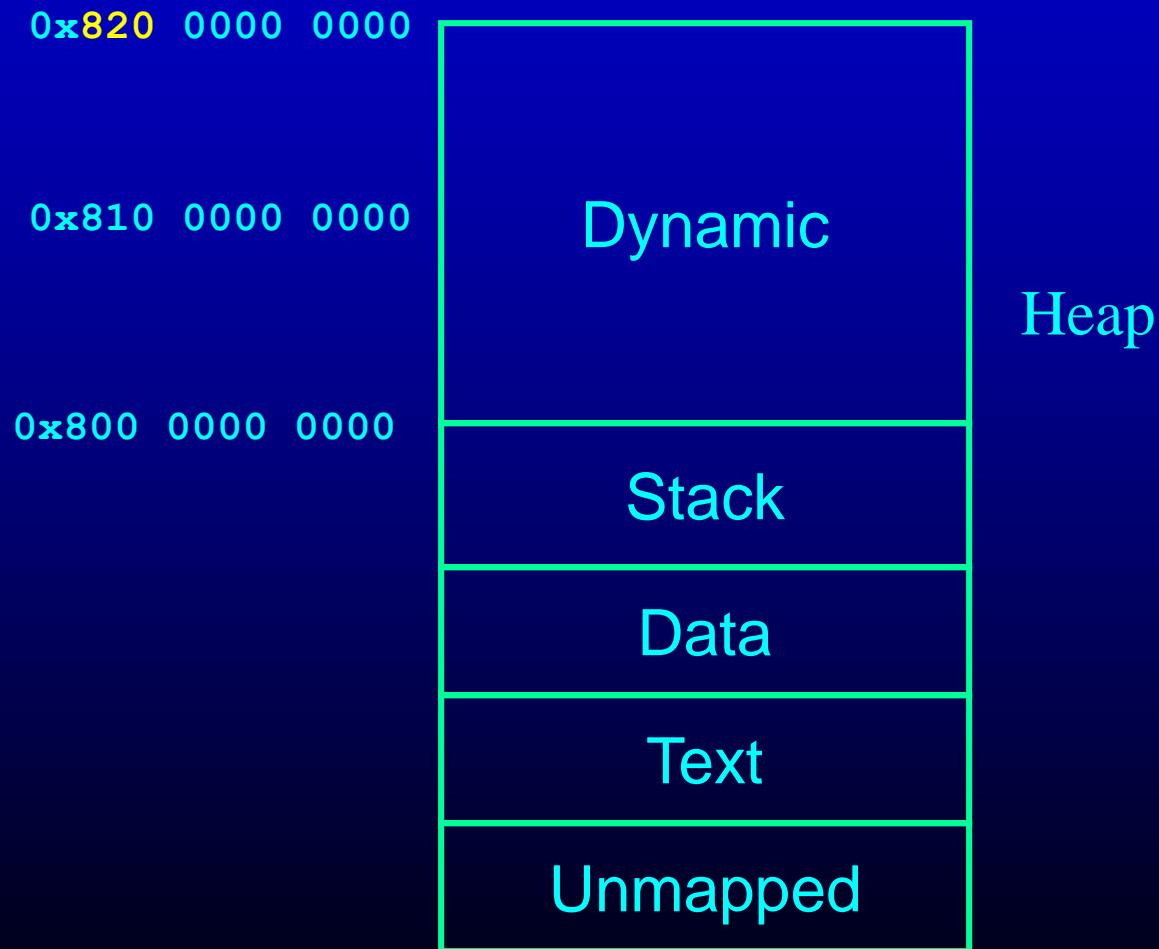
Scenario: Current heap goes from 0x800 0000 000- 0x810 0000 000
Need to allocate large block of memory
No block that large available



Getting More Heap Memory

Solution: Talk to OS, increase size of heap (sbrk)

Allocate block in new heap



The Stack

- Arguments 0 to 6 are in:

- %rdi, %rsi, %rdx,
- %rcx, %r8 and %r9

%rbp

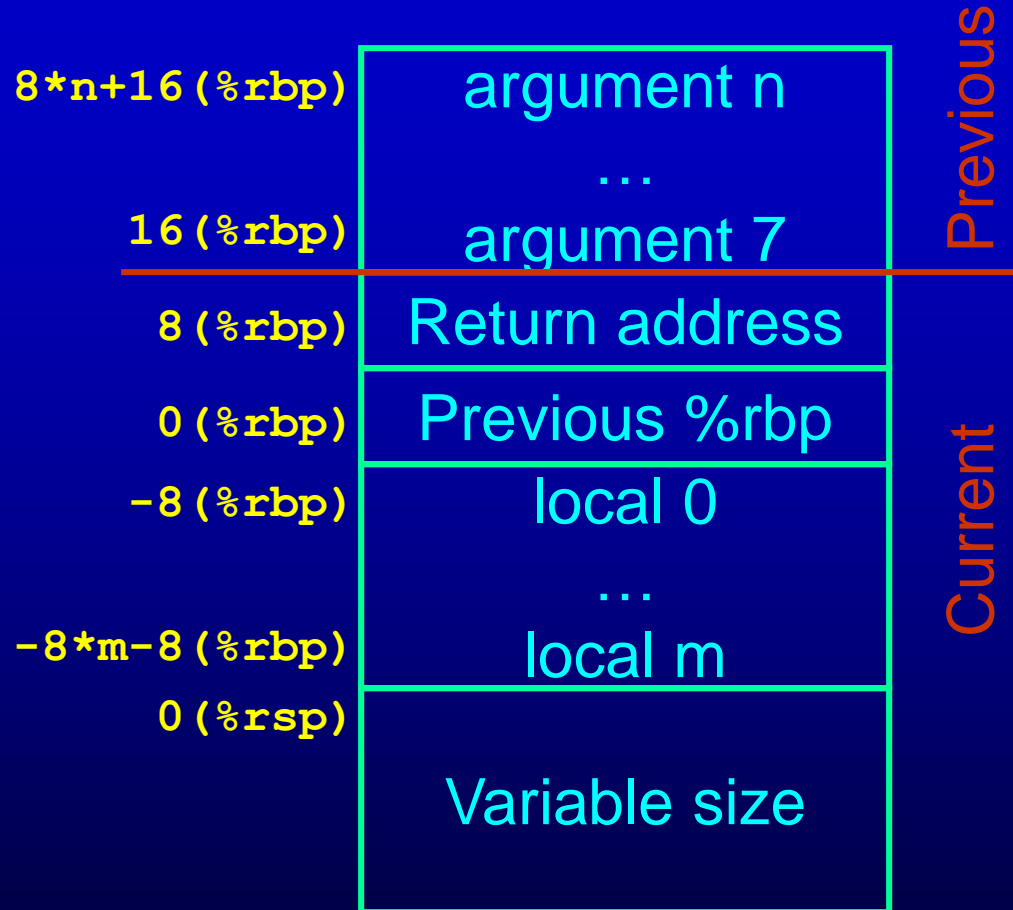
- marks the beginning of the current frame

%rsp

- marks the end

%rax

- return value

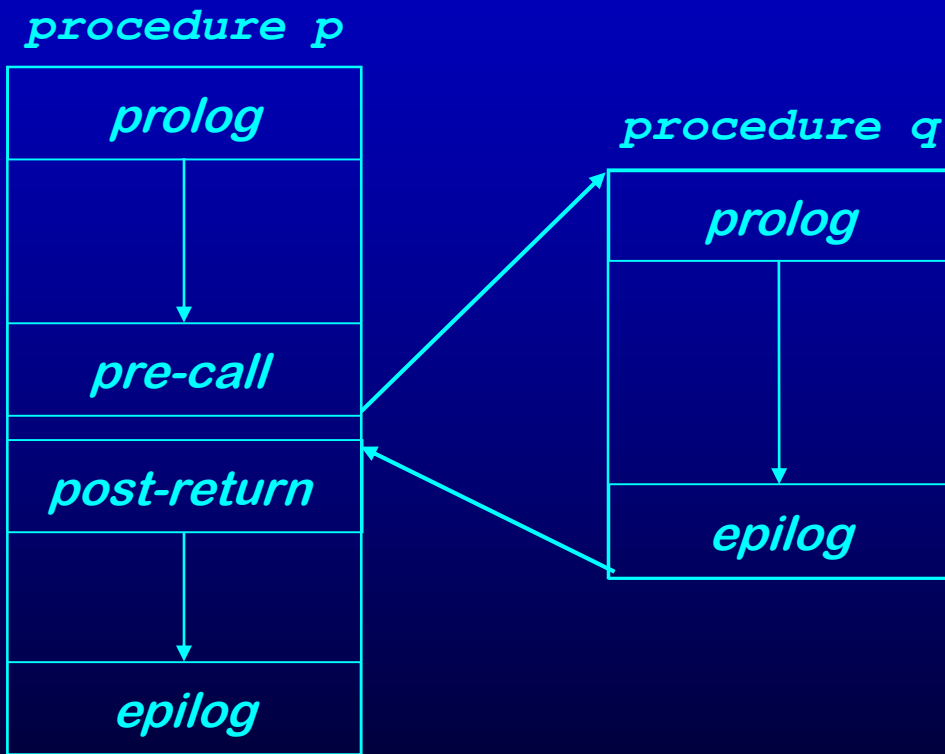


Question:

- Why use a stack? Why not use the heap or pre-allocated in the data segment?

Procedure Linkages

Standard procedure linkage



Pre-call:

- Save caller-saved registers
- Push arguments

Prolog:

- Push old frame pointer
- Save callee-saved registers
- Make room for temporaries

Epilog:

- Restore callee-saved
- Pop old frame pointer
- Store return value

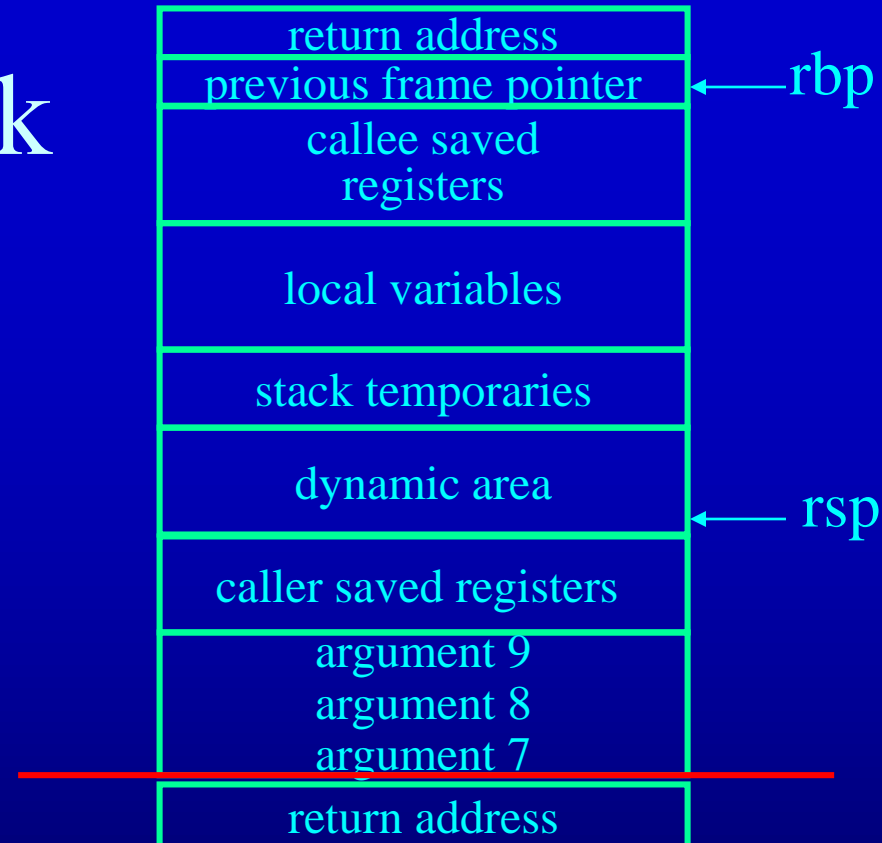
Post-return:

- Restore caller-saved
- Pop arguments

Stack

- Calling: Caller
 - Assume %rcx is live and is caller save
 - Call foo(A, B, C, D, E, F, G, H, I)
 - A to I are at -8(%rbp) to -72(%rbp)

```
push    %rcx
push    -72(%rbp)
push    -64(%rbp)
push    -56(%rbp)
mov     -48(%rbp), %r9
mov     -40(%rbp), %r8
mov     -32(%rbp), %rcx
mov     -24(%rbp), %rdx
mov     -16(%rbp), %rsi
mov     -8(%rbp), %rdi
call    foo
```

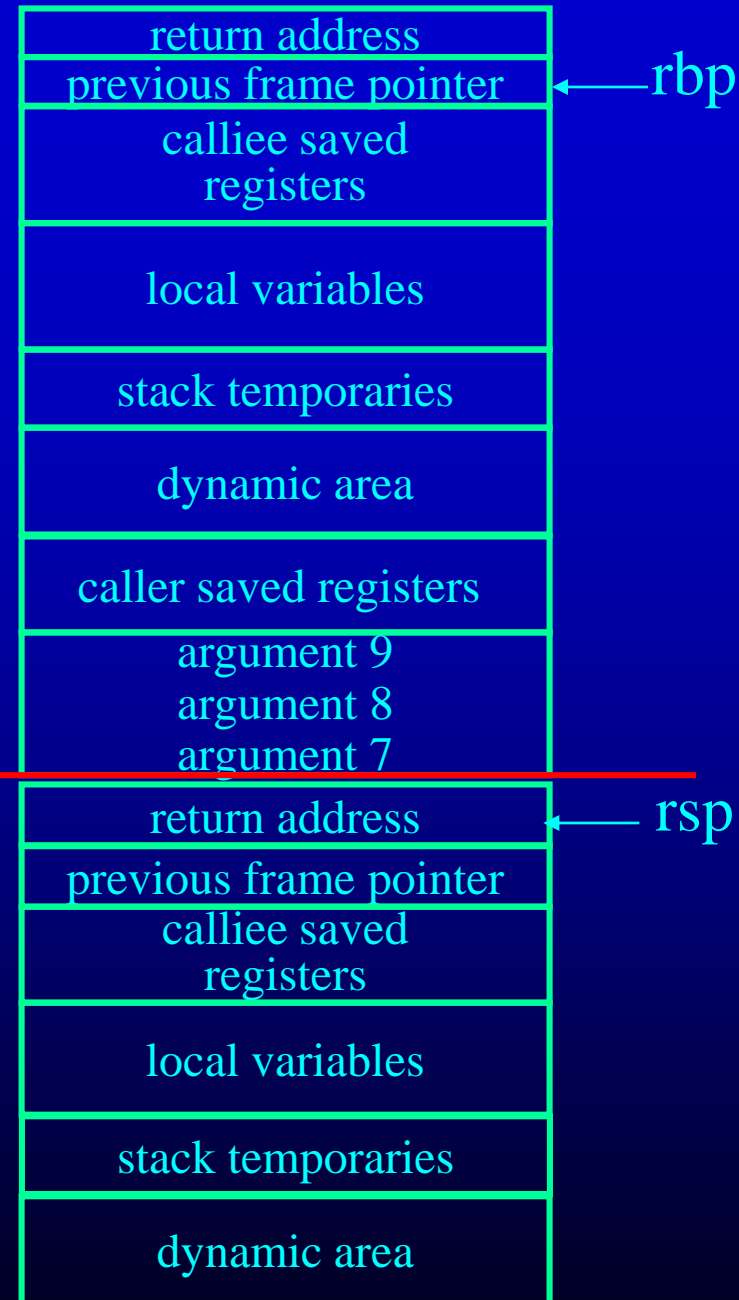


Stack

- Calling: Callee
 - Assume %rbx is used in the function and is callee save
 - Assume 40 bytes are required for locals

foo:

```
push    %rbp
mov     %rsp, %rbp
enter  $48, $0
sub     $48, %rsp
mov     %rbx, -8(%rbp)
```



Stack

- Arguments
- Call foo(A, B, C, D, E, F, G, H, I)
 - Passed in by pushing before the call

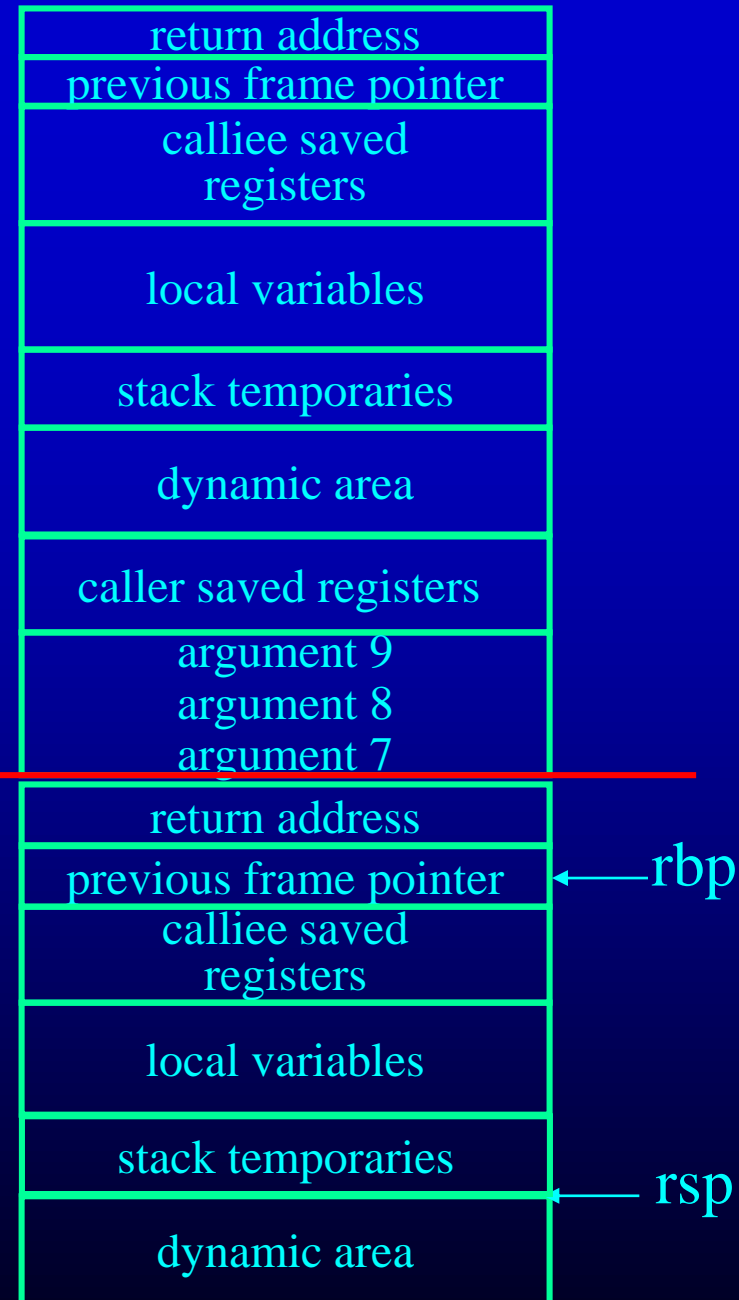
```

push    -72(%rbp)
push    -64(%rbp)
push    -56(%rbp)
mov     -48(%rbp), %r9
mov     -40(%rbp), %r8
mov     -32(%rbp), %rcx
mov     -24(%rbp), %rdx
mov     -16(%rbp), %rsi
mov     -8(%rbp), %rdi
call    foo
    
```

- Access A to F via registers
 - or put them in local memory
- Access rest using 16+xx(%rbp)

```

mov     16(%rbp), %rax
mov     24(%rbp), %r10
    
```



Stack

- Locals and Temporaries

- Calculate the size and allocate space on the stack

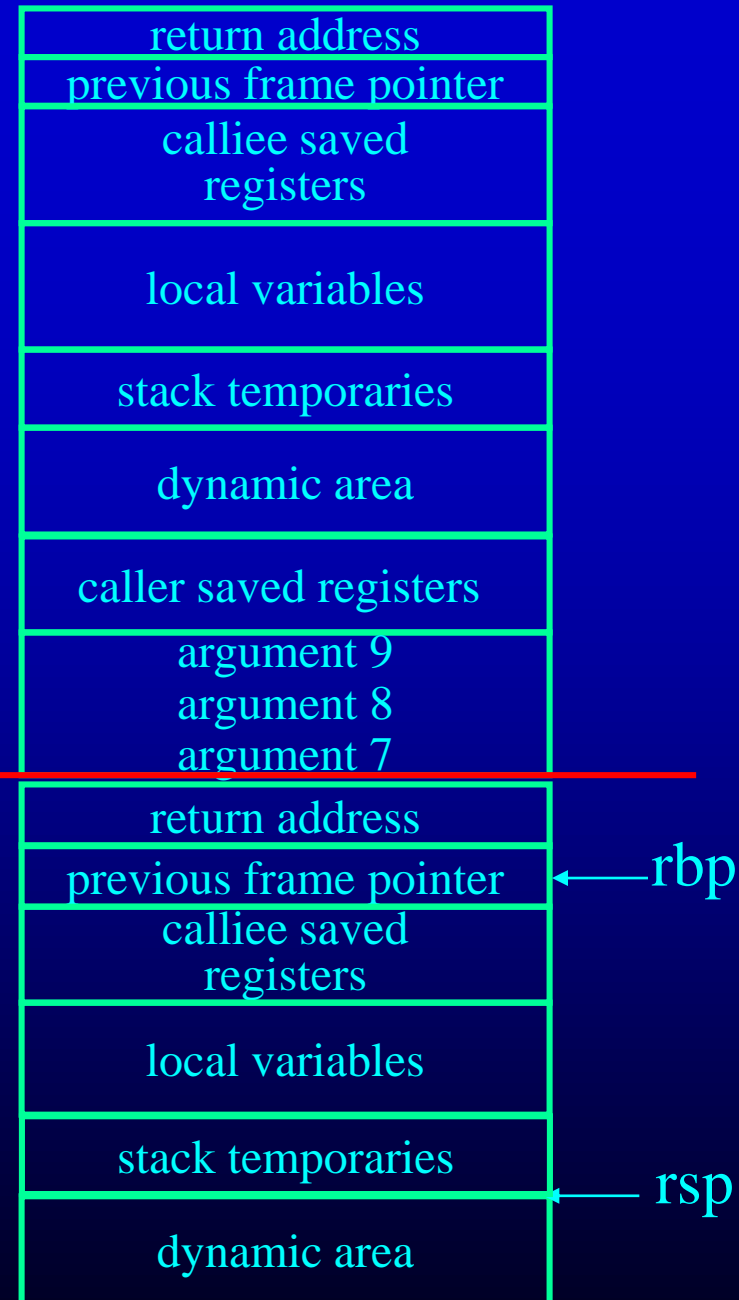
```
sub    $48, %rsp
```

```
or     enter    $48, 0
```

- Access using -8-xx(%rbp)

```
mov    -28(%rbp), %r10
```

```
mov    %r11, -20(%rbp)
```

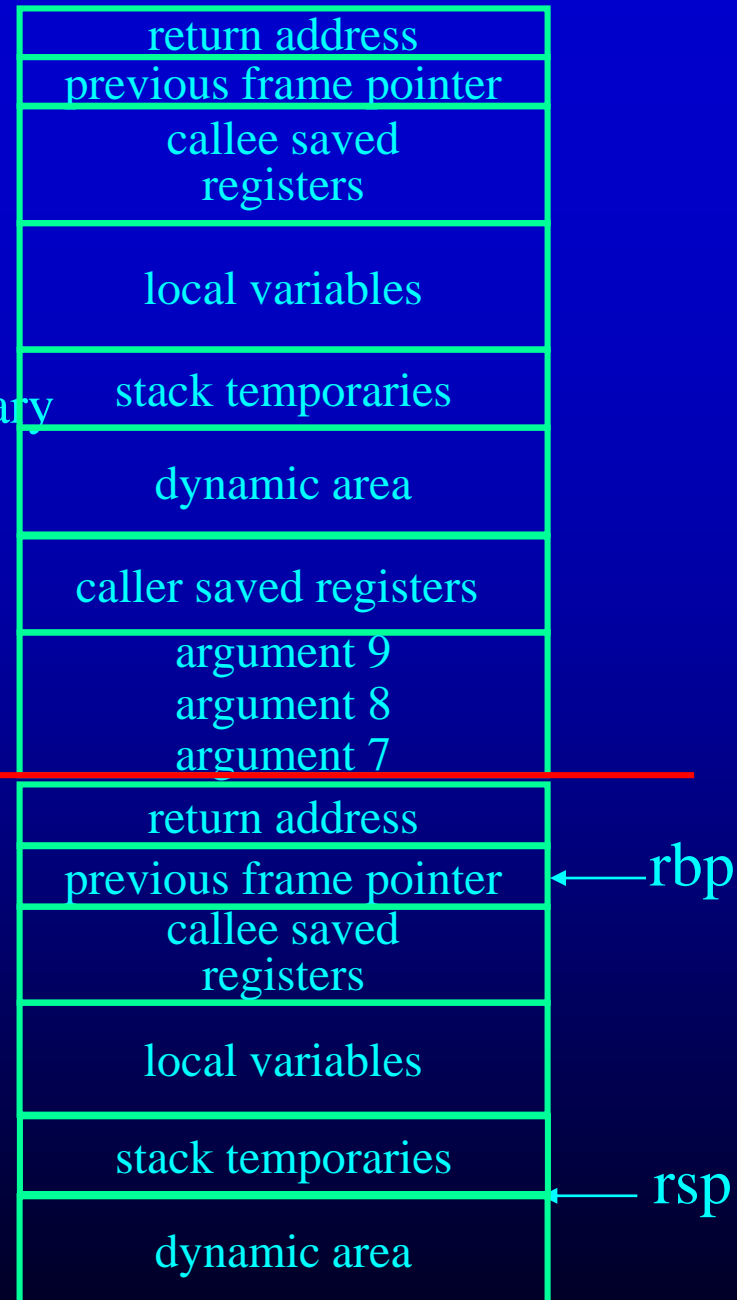


Stack

- Returning Callee

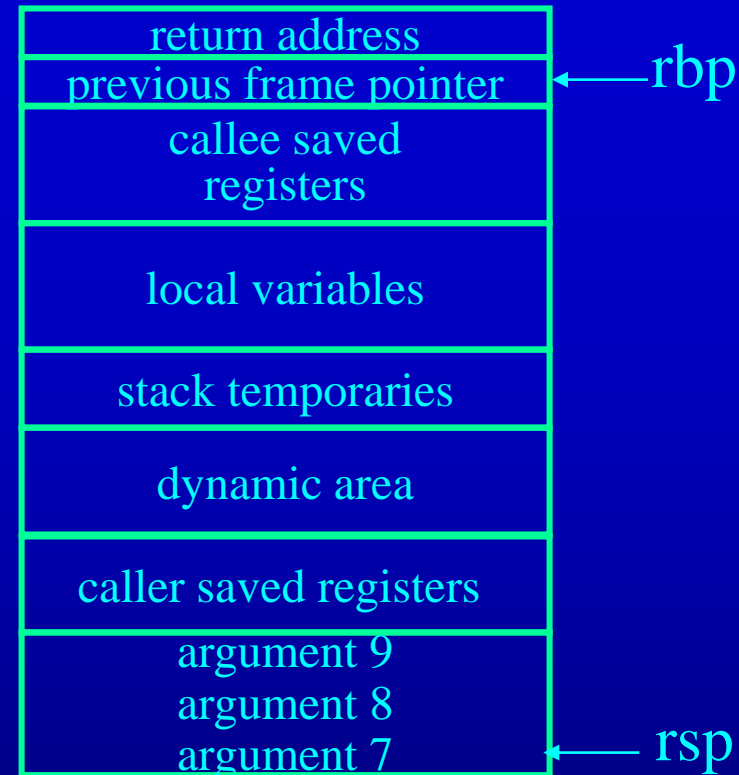
- Assume the return value is the first temporary
- Restore the caller saved register
- Put the return value in %rax
- Tear-down the call stack

```
mov     -8(%rbp), %rbx
mov     -16(%rbp), %rax
mov     leave    %rbp, %rsp
pop     %rbp
ret
```



Stack

- Returning Caller
- Assume the return value goes to the first temporary
 - Restore the stack to reclaim the argument space
 - Restore the caller save registers
 - Save the return value



```
call    foo
add     $24, %rsp
pop     %rcx
mov     %rax, 8(%rbp)
...
```

Question:

- Do you need the \$rbp?
- What are the advantages and disadvantages of having \$rbp?

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

Outline

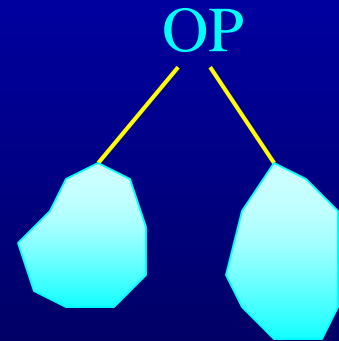
- Generation of expressions and statements
- Generation of control flow
- x86-64 Processor
- Guidelines in writing a code generator

Expressions

- Expressions are represented as trees
 - Expression may produce a value
 - Or, it may set the condition codes (boolean exprs)
- 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
 - The idea is to linearize the expression tree
 - Left to Right Depth-First Traversal of the expression tree
 - Allocate temporaries for intermediates (all the nodes of the tree)
 - New temporary for each intermediate
 - All the temporaries on the stack (for now)
 - Each expression is a single 3-addr op
 - $x = y \text{ op } z$
 - Code generation for the 3-addr expression
 - Load y into register `%rax`
 - Perform `op z, %rax`
 - Store `%rax` to x

Issues in Lowering Expressions

- 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
 - don't worry, we'll take care of them in the optimization passes
 - keep the code generator very simple

What about statements?

- Assignment statements are simple
 - Generate code for RHS expression
 - Store the resulting value to the LHS address
- But what about conditionals and loops?

Outline

- Generation of statements
- Generation of control flow
- Guidelines in writing a code generator

Two Techniques

- Template Matching
- Short-circuit Conditionals
- Both are based on structural induction
 - Generate a representation for the sub-parts
 - Combine them into a representation for the whole

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;
```

<do test>

joper .L0

<FALSE BODY>

jmp .L1

.L0:

<TRUE BODY>

.L1:

Return address
previous frame pointer
Local variable px (10)
Local variable py (20)
Local variable pz (30)
Argument 9: cx (30)
Argument 8: bx (20)
Argument 7: ax (10)
Return address
previous frame pointer
Local variable dx (??)
Local variable dy (??)
Local variable dz (??)

← rbp

← 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
```

<FALSE BODY>

```
jmp      .L1
```

.L0:

<TRUE BODY>

.L1:

Return address
previous frame pointer
Local variable px (10)
Local variable py (20)
Local variable pz (30)
Argument 9: cx (30)
Argument 8: bx (20)
Argument 7: ax (10)
Return address
previous frame pointer
Local variable dx (??)
Local variable dy (??)
Local variable dz (??)

← rbp

← 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:

<TRUE BODY>

.L1:

Return address
previous frame pointer
Local variable px (10)
Local variable py (20)
Local variable pz (30)
Argument 9: cx (30)
Argument 8: bx (20)
Argument 7: ax (10)
Return address
previous frame pointer
Local variable dx (??)
Local variable dy (??)
Local variable dz (??)

← rbp

← 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:

Return address
previous frame pointer
Local variable px (10)
Local variable py (20)
Local variable pz (30)
Argument 9: cx (30)
Argument 8: bx (20)
Argument 7: ax (10)
Return address
previous frame pointer
Local variable dx (??)
Local variable dy (??)
Local variable dz (??)

← rbp

← rsp

Template for while loops

```
while (test)  
    body
```

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:
```

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:

```

CODE	DATA
Control Flow	Global Static Variables
Procedures	Global Dynamic Data
Statements	Local Variables
Data Access	Temporaries
	Parameter Passing
	Read-only Data

Question:

- What is the template for?

`do`

`body`

`while (test)`

Question:

- What is the template for?

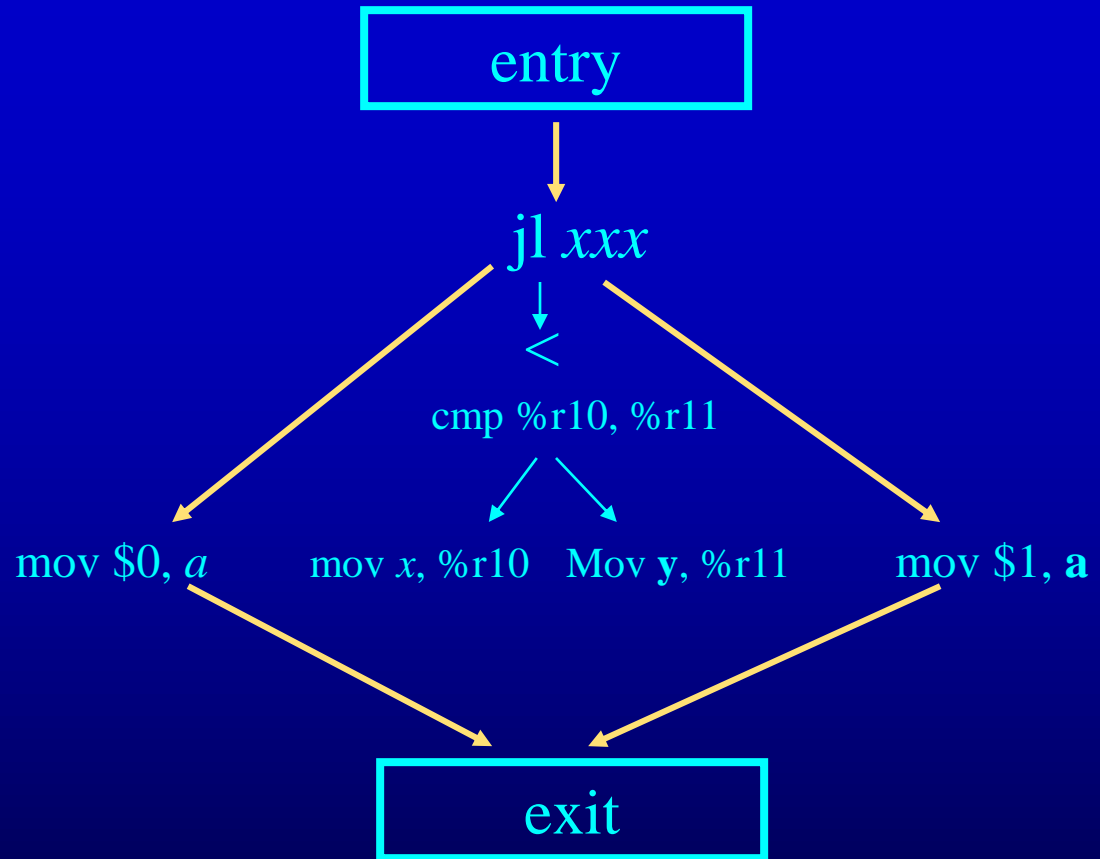
```
do  
    body  
while (test)
```

```
lab_begin:  
    <body>  
    <do test>  
joper lab_begin
```

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;  
}
```



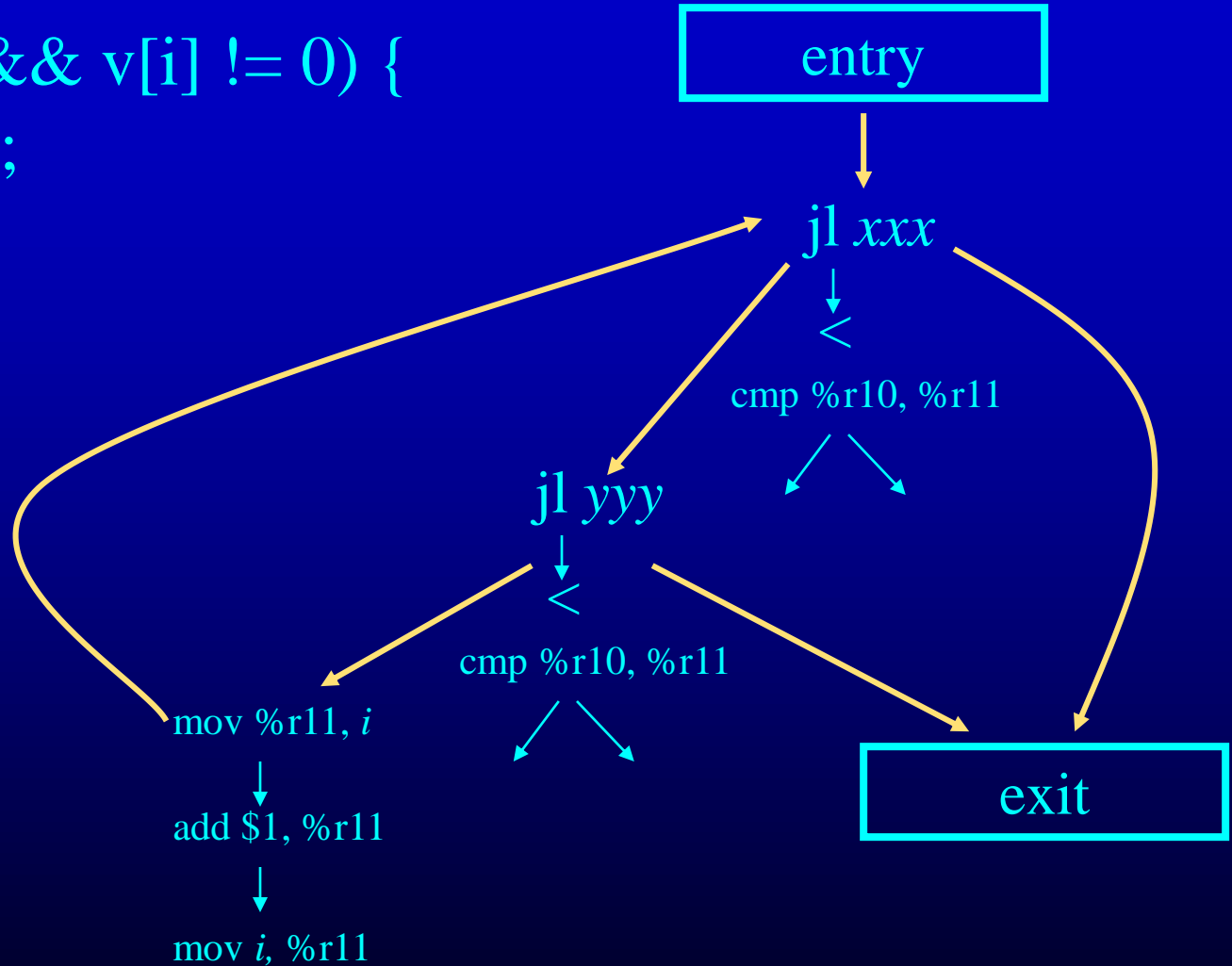
Pattern for if then else

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

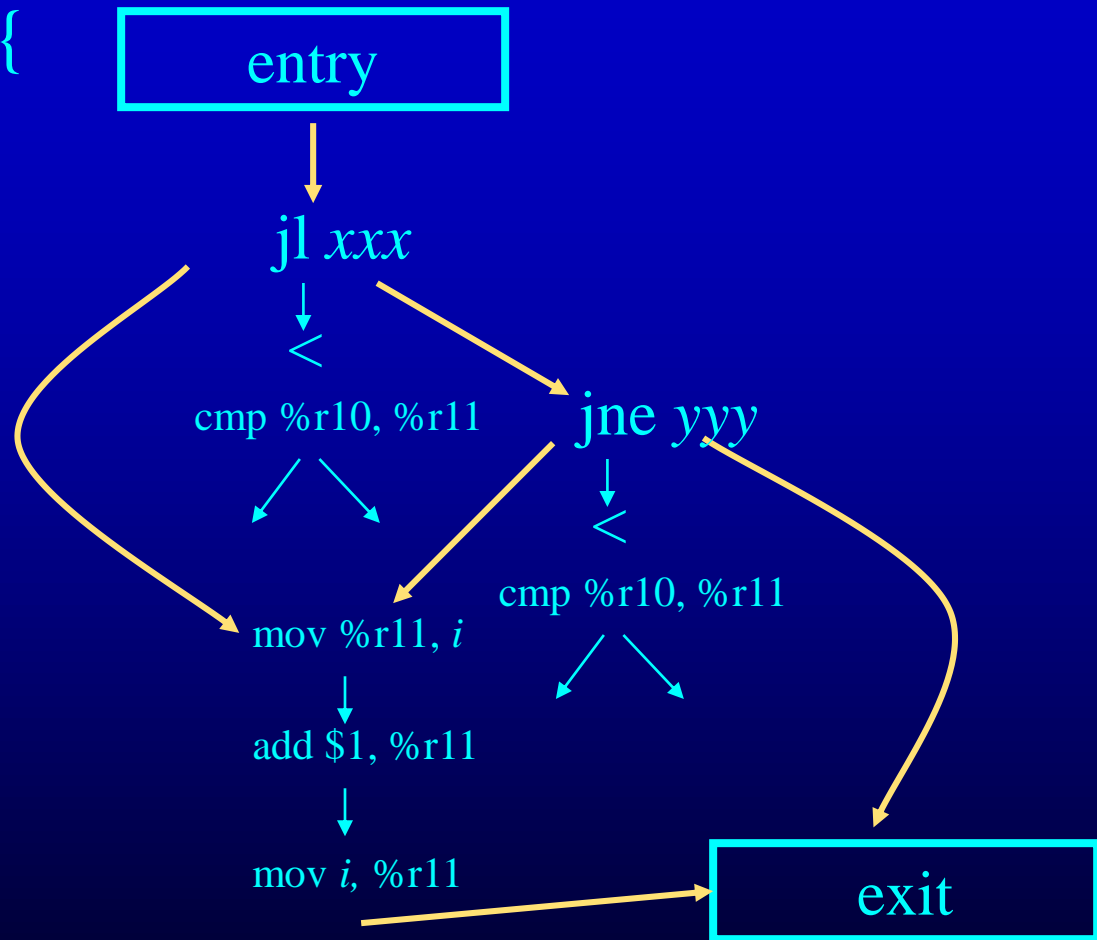
Short-Circuit Conditionals

```
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

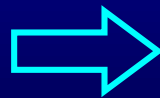
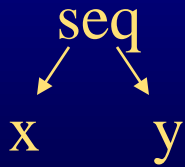
new kind of node - nop node

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



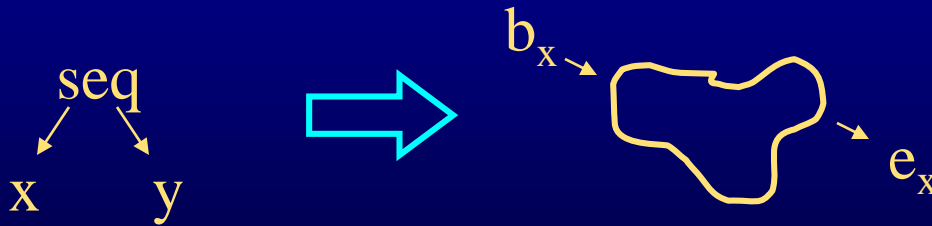
Destructuring Seq Nodes

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: (b_x, e_x) = destruct(x);



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



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

3: next(e_x) = b_y ;



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

3: next(e_x) = b_y ; 4: return (b_x, e_y);



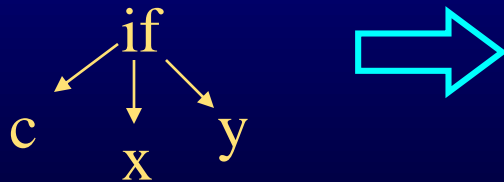
Destructuring If Nodes

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 if **c x y**



Destructuring If Nodes

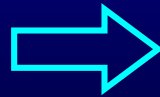
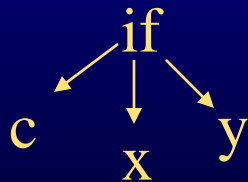
destruct(**n**)

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 if **c** **x** **y**

1: (**b_x**,**e_x**) = destruct(**x**);



Destructuring If Nodes

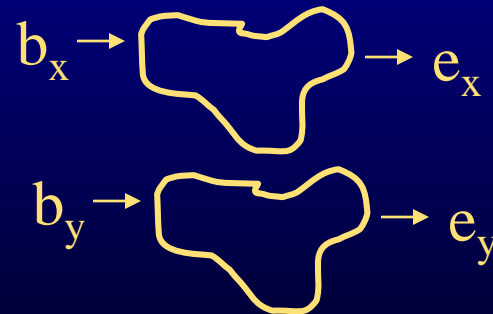
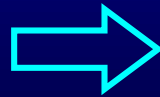
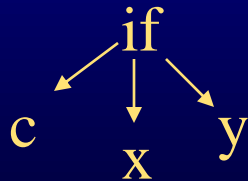
destruct(**n**)

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

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

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

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



Destructuring If Nodes

destruct(**n**)

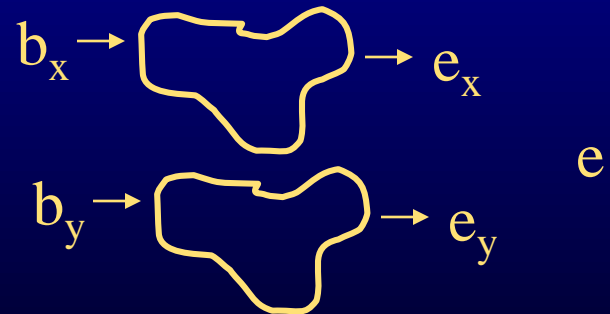
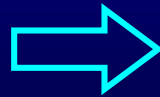
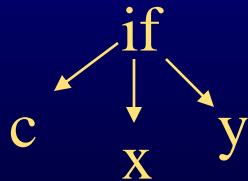
generates lowered form of structured code represented by **n**

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

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

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

3: **e** = new nop;



Destructuring If Nodes

destruct(**n**)

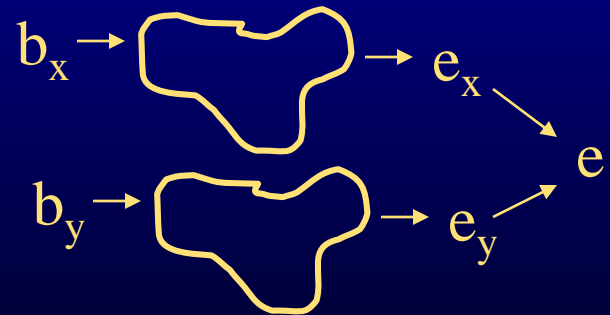
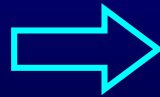
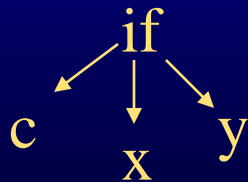
generates lowered form of structured code represented by **n**

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

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

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

3: **e** = new nop; 4: next(e_x) = **e**; 5: next(e_y) = **e**;



Destructuring If Nodes

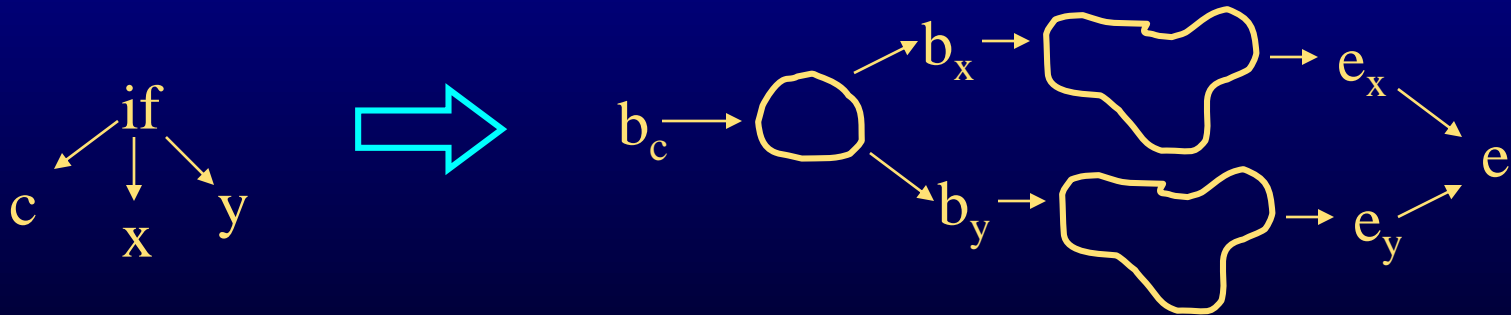
destruct(**n**)

generates lowered form of structured code represented by **n**
returns (**b**,**e**) - **b** is begin node, **e** is end node in destructured form
if **n** is of the form if **c** **x** **y**

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

3: **e** = new nop; 4: next(e_x) = **e**; 5: next(e_y) = **e**;

6: b_c = shortcircuit(**c**, b_x , b_y);



Destructuring If Nodes

destruct(**n**)

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

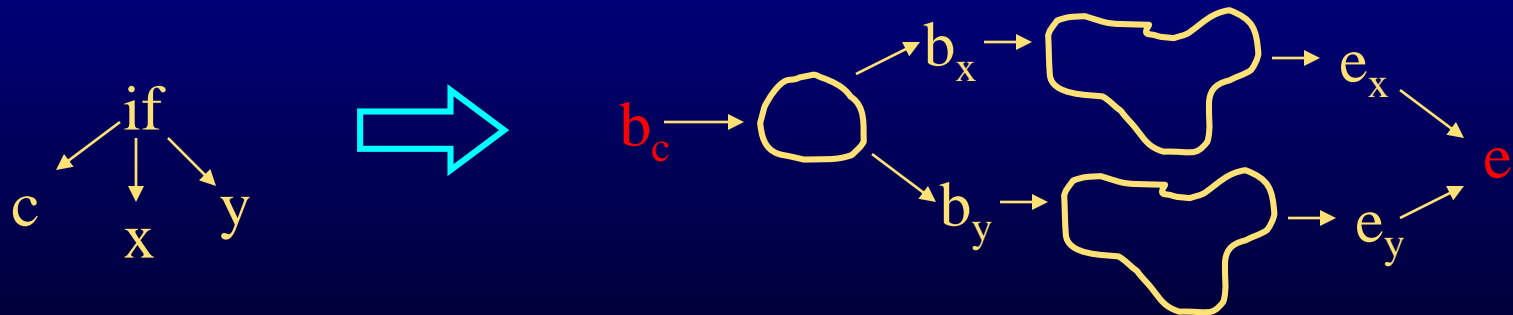
returns (**b**,**e**) - **b** is begin node, **e** is end node in destructed form

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

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

3: **e** = new nop; 4: next(e_x) = **e**; 5: next(e_y) = **e**;

6: b_c = shortcircuit(**c**, b_x , b_y); 7: return (b_c , **e**);

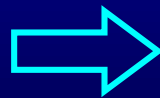
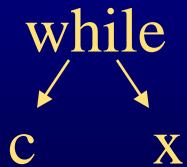


Destructuring While Nodes

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**



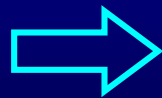
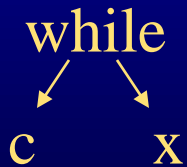
Destructuring While Nodes

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** = new nop;



e

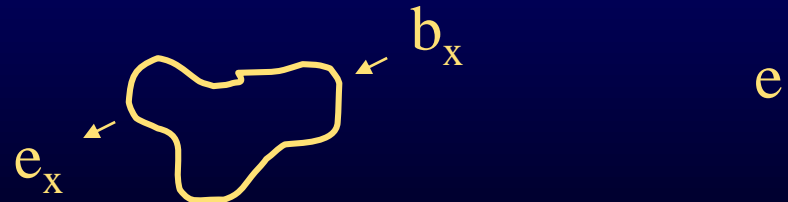
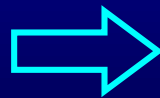
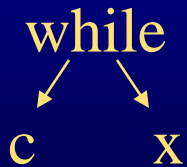
Destructuring While Nodes

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** = new nop; 2: (**b_x**,**e_x**) = destruct(**x**);



Destructuring While Nodes

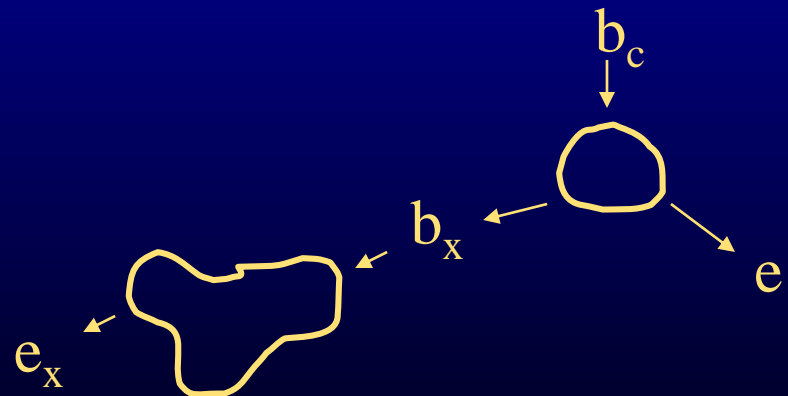
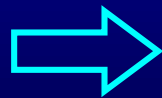
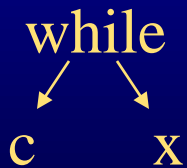
destruct(**n**)

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** = new nop; 2: (**b_x**,**e_x**) = destruct(**x**);

3: **b_c** = shortcircuit(**c**, **b_x**, **e**);



Destructuring While Nodes

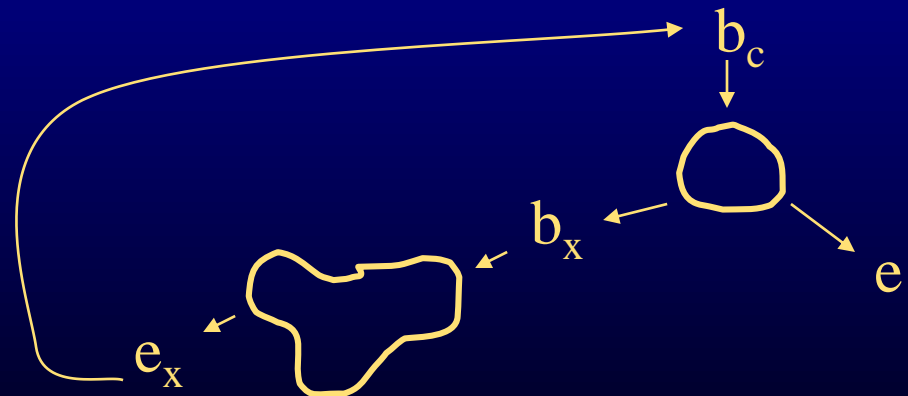
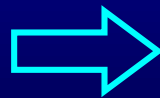
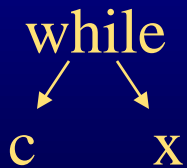
destruct(**n**)

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

returns (**b**,**e**) - **b** is begin node, **e** is end node in destructured form
if **n** is of the form while **c** **x**

1: **e** = new nop; 2: (**b_x**,**e_x**) = destruct(**x**);

3: **b_c** = shortcircuit(**c**, **b_x**, **e**); 4: next(**e_x**) = **b_c**;



Destructuring While Nodes

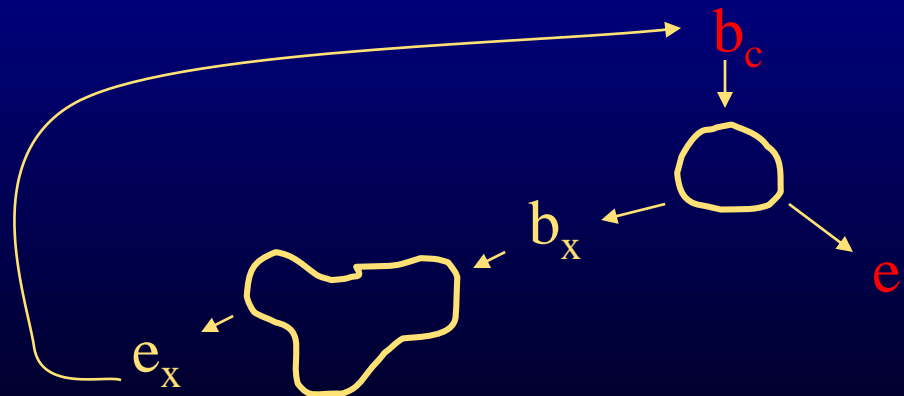
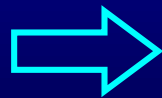
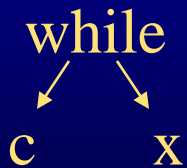
destruct(**n**)

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

returns (**b**,**e**) - **b** is begin node, **e** is end node in destructured form
if **n** is of the form while **c** **x**

1: **e** = new nop; 2: (**b_x**,**e_x**) = destruct(**x**);

3: **b_c** = shortcircuit(**c**, **b_x**, **e**); 4: 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`

`c1 && c2` 

Shortcircuiting And Conditions

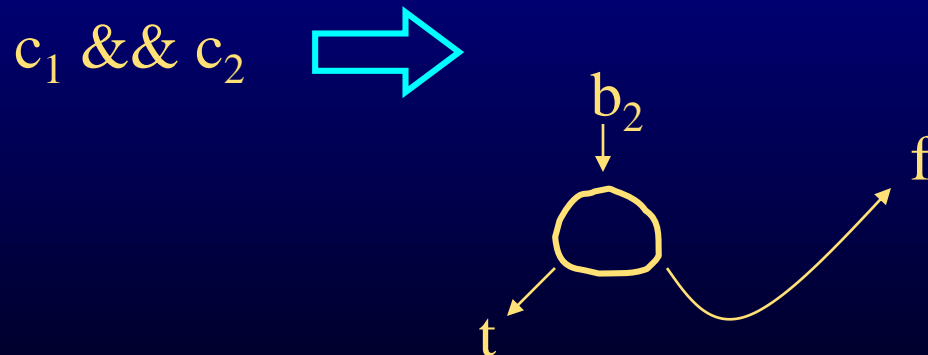
`shortcircuit(c, t, f)`

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);`



Shortcircuiting And Conditions

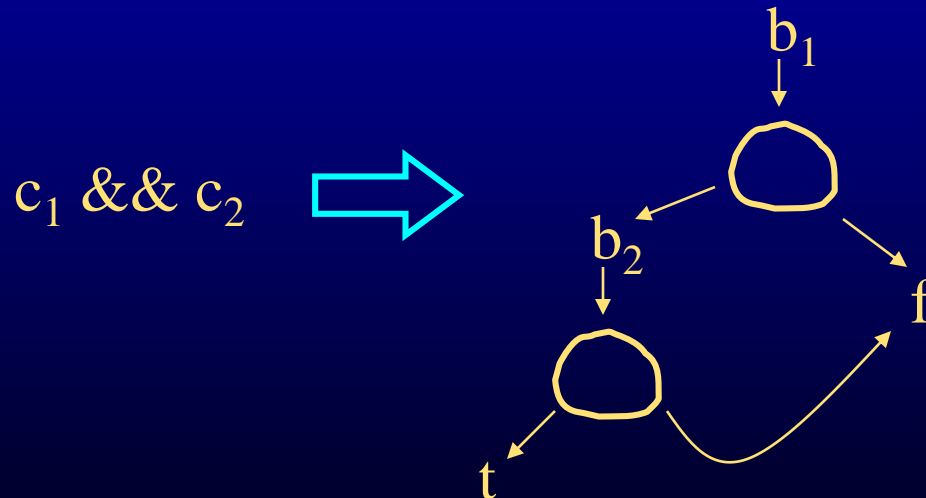
`shortcircuit(c, t, f)`

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)`;



Shortcircuiting And Conditions

`shortcircuit(c, t, f)`

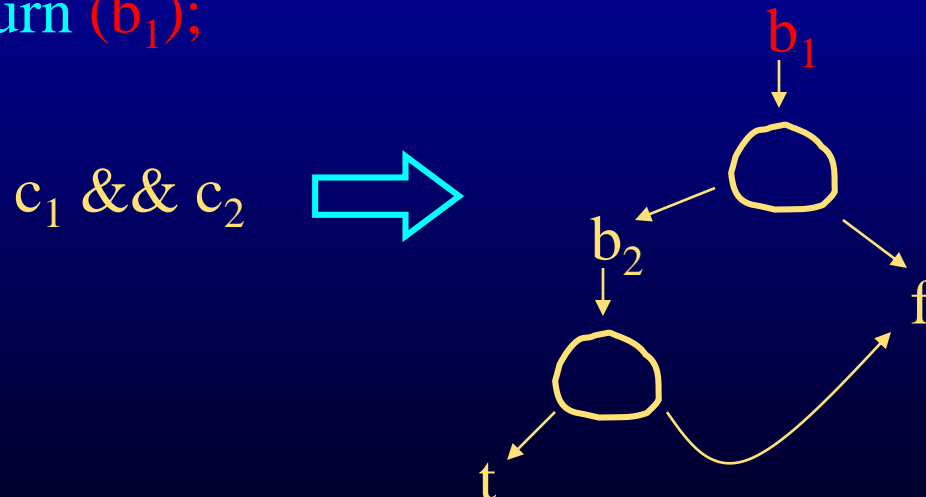
generates shortcircuit form of conditional represented by `c`

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 Or 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`

`c1 || c2` 

Shortcircuiting Or Conditions

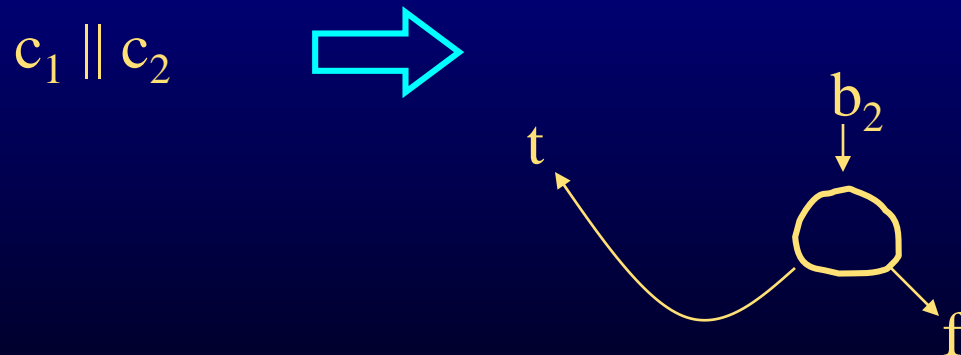
shortcircuit(c , t , f)

generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

if c is of the form $c_1 \parallel c_2$

1: $b_2 = \text{shortcircuit}(c_2, t, f);$



Shortcircuiting Or Conditions

`shortcircuit(c, t, f)`

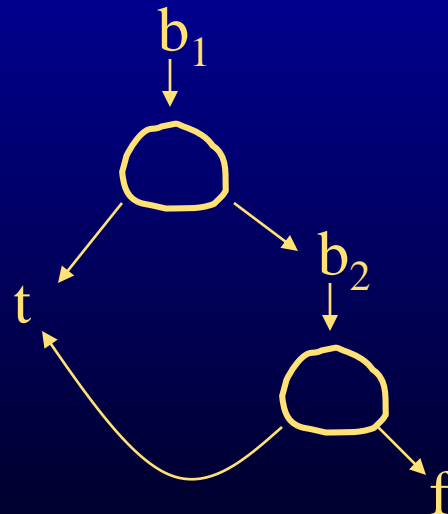
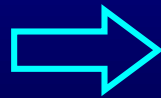
generates shortcircuit form of conditional represented by `c`

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, t, b2);`

`c1 || c2`



Shortcircuiting Or Conditions

`shortcircuit(c, t, f)`

generates shortcircuit form of conditional represented by `c`

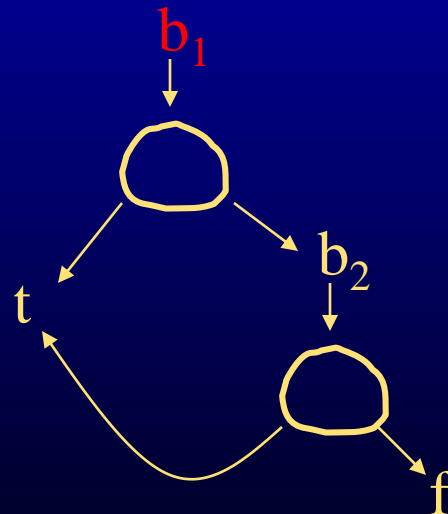
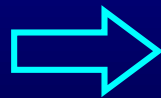
returns `b` - `b` is begin node of shortcircuit form

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

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

3: `return (b1);`

`c1 || c2`



Shortcircuiting Not Conditions

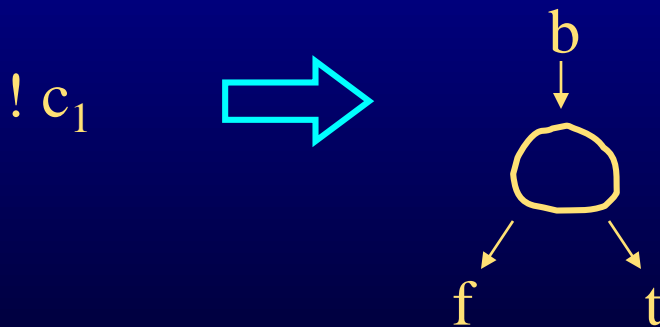
`shortcircuit(c, t, f)`

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

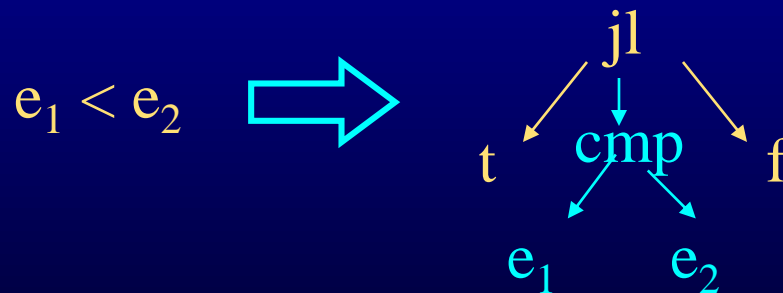
`shortcircuit(c, t, f)`

generates shortcircuit form of conditional represented by `c`

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

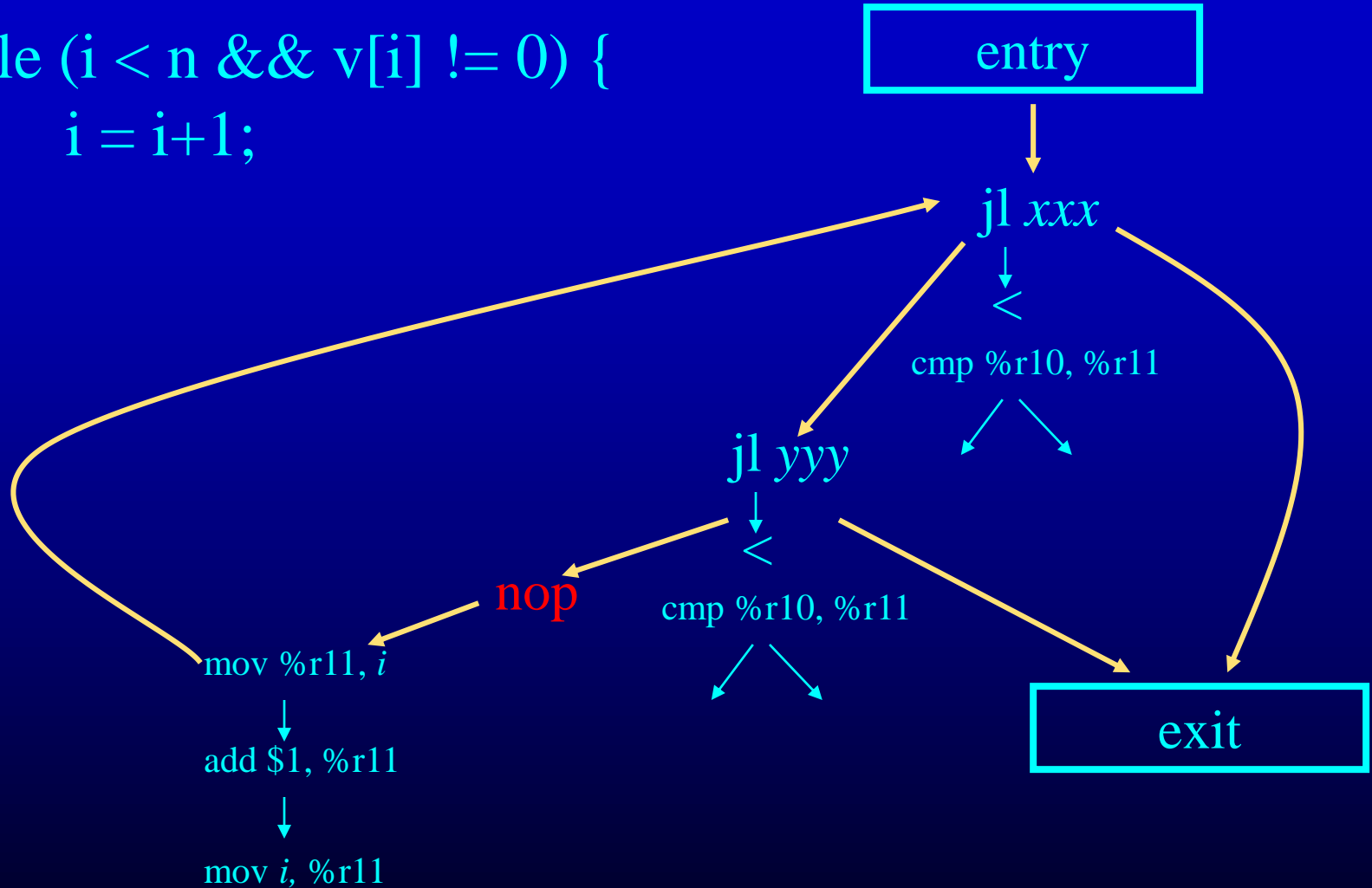
if `c` is of the form $e_1 < e_2$

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

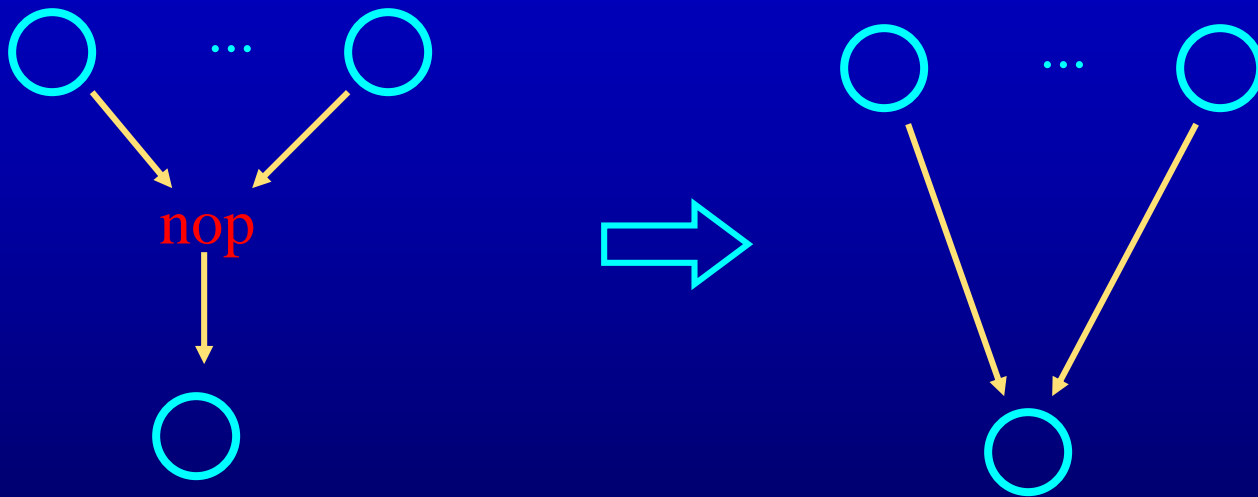


Nops In Destructured Representation

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



Eliminating Nops Via Peephole Optimization



Linearizing CFG to Assembler

- Generate labels for edge targets at branches
 - Labels will correspond to branch targets
 - Can use patterns for this
- Generate code for statements/conditional expressions
- Generate code for procedure entry/exit

Exploring Assembly Patterns

```
struct { int x, y; double z; } b;
```

```
int g;
```

```
int a[10];
```

```
char *s = "Test String";
```

```
int f(int p) {
```

```
    int i;
```

```
    int s;
```

```
    s = 0.0;
```

```
    for (i = 0; i < 10; i++) {
```

```
        s = s + a[i];
```

```
    }
```

```
    return s;
```

```
}
```

- gcc -g -S t.c
- vi t.s

Outline

- Generation of statements
- Generation of control flow
- x86-64 Processor
- Guidelines in writing a code generator