Unoptimized Code Generation
Big Picture

• Starting point - AST
• 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
  - Basic Block is a Sequence of Instructions with
    - 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
AST to CFG for If Then Else

Source Code
if (condition) {
  code for then
}
else {
  code for else
}

CFG

AST
if

AST for condition
AST for then
AST for else

CFG for condition
CFG for then
CFG for else
no op
AST to CFG for If Then

Source Code
if (condition) {
  code for then
}

CFG
CFG for condition
CFG for then
no op

AST
if
AST for condition
AST for then
AST to CFG for While

Source Code
while (condition) {
  code for loop body
}

CFG
CFG for condition
CFG for loop body
no op

AST
while
AST for condition
AST for loop body
AST to CFG for Statements

Source Code
- code for S1;
- code for S2

AST
- seq
  - AST for S1
  - AST 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

\[
\begin{align*}
  s &= 0; \\
  a &= 4;
\end{align*}
\]

\[
\begin{align*}
  s &= 0; \\
  a &= 4;
\end{align*}
\]
s = 0;
a = 4;
i = 0;
k == 0

b = 2;
b = 1;
i < n

s = s + a * b;
return s;
i = i + 1;

s = 0;
a = 4;
s = 0;
a = 4;
i = 0;
k == 0
b = 2;
b = 1;
i < n
s = s + a*b;
i = i + 1;
return s;

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

b = 1;
b = 2;

i < n

s = s + a*b;

return s;
i = i + 1;

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

$b = 2;$
$b = 1;$

$i < n$

$s = s + a*b;$

return $s;$

$i = i + 1;$

$b = 2;$

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

b = 2;
b = 1;
i < n
s = s + a*b;
return s;
i = i + 1;

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

b = 2;
i < n
s = 0;
a = 4;
i = 0;
k == 0

b = 2;
b = 1;
i < n

s = s + a*b;
return s;
i = i + 1;

b = 2;
i < n

s = s + a*b;

s = 0;
a = 4;
i = 0;
k == 0
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n
s = s + a*b;
return s;
i = i + 1;

s = 0;
a = 4;
i = 0;
k == 0
b = 2;
i < n
s = s + a*b;
i = i + 1;
s = 0;  
a = 4;  
i = 0;  
k == 0  

b = 2;  
b = 1;  
i < n  
s = s + a*b;  
i = i + 1;  
return s;  

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

b = 2;  
i < n  
s = s + a*b;  
i = i + 1;
```plaintext
s = 0;
a = 4;
i = 0;
k == 0
b = 2;
b = 1;
i < n
s = s + a*b;
return s;
i = i + 1;
```
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n
s = s + a*b;
i = i + 1;
return s;

s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n
s = s + a*b;
i = i + 1;
return s;
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

```plaintext
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
  \(((i < n) \&\& (v[i] \neq 0)) \lor i > k)\)

• 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;
}

entry

jl xxx

jl yyy

cmp %r10, %r11

mov %r11, i
add $1, %r11
mov i, %r11

cmp %r10, %r11

exit
More Short-Circuit Conditionals

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

entry

jl xxx

cmp %r10, %r11

jne yyy

cmp %r10, %r11

mov %r11, i

add $1, %r11

mov i, %r11

exit
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 \( \text{seq} \ x \ y \)
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: \((b_x, e_x) = \text{destruct}(x);\)
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;

seq

\[\begin{array}{c}
\text{seq} \\
\text{x} \\
\text{y}
\end{array}\]

\[\begin{array}{c}
b_x \\
e_x \\
b_y \\
e_y
\end{array}\]
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: \((b_x, e_x) = \text{destruct}(x)\); 2: \((b_y, e_y) = \text{destruct}(y)\);
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

\[
\begin{align*}
\text{if} & \\
\text{c} & \text{x} \quad \text{y}
\end{align*}
\]
Destructuring If Nodes

\[ \text{destruct}(n) \]

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 \( \text{if} \ c \ x \ y \)
\[ 1: (b_x,e_x) = \text{destruct}(x); \]
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 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;
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 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);
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

\[
\text{destruct}(n)
\]

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 \( \text{while} \ c \ x \)

1: \( e = \text{new} \ \text{nop}; \)

\[
\text{while} \quad \left\langle \begin{array}{c}
c \\
x \\
e
\end{array} \right. \]

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, e\_x) = destruct(x);
3: b\_c = shortcircuit(c, b\_x, e);

while
\[ \begin{array}{c}
\text{c} \\
\text{x}
\end{array} \]
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

\begin{align*}
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;
\end{align*}

\[ \text{while} \begin{array}{c}
\text{c} \\
\text{x}
\end{array} \longrightarrow \begin{array}{c}
b_c \\
\text{b}_x \\
\text{e}
\end{array} \begin{array}{c}
e_x \\
\text{b}_x
\end{array} \]
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); 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

\[ c₁ \land c₂ \]
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
  l: b₂ = shortcircuit(c₂, t, f);

\[
c₁ \land c₂ \quad \Rightarrow \quad b₂ \quad f \\
t
\]
Shortcircuiting And Conditions

\[
\text{shortcircuit}(c, t, f)
\]
generates shortcircuit form of conditional represented by \( c \)

returns \( b - b \) is begin node of shortcircuit form

if \( c \) is of the form \( c_1 \&\& c_2 \)

1: \( b_2 = \text{shortcircuit}(c_2, t, f) \); 2: \( b_1 = \text{shortcircuit}(c_1, b_2, f) \);
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);

c₁ && c₂

\[ c₁ \land c₂ \]

\[ b₁ \]

\[ b₂ \]

\[ f \]

\[ t \]
Shortcircuiting Or Conditions

\text{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 \( c_1 \parallel c_2 \)
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 \( c_1 \parallel c_2 \)
1: \( b_2 = \text{shortcircuit}(c_2, t, f) \); 2: \( b_1 = \text{shortcircuit}(c_1, t, b_2) \);
Shortcircuiting Or Conditions

\[ \text{shortcircuit}(c, t, f) \]

generates shortcircuit form of conditional represented by \( c \)
returns \( b \) - \( b \) is begin node of shortcircuit form
if \( c \) is of the form \( c_1 \parallel c_2 \)
\begin{enumerate}
  \item \( b_2 = \text{shortcircuit}(c_2, t, f) \);
  \item \( b_1 = \text{shortcircuit}(c_1, t, b_2) \);
  \item return \( b_1 \);
\end{enumerate}
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);
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: $b = \text{new cbr}(e_1 < e_2, t, f)$;
  2: return (b);
while (i < n && v[i] != 0) {
    i = i+1;
}
Eliminating Nops Via Peephole Optimization

![Diagram showing the optimization process](image-url)
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:
  - Orchestrates register usage
  - Generates low-level code for interfacing with machine
Typical Memory Layout

- Global Variables
- Read-only constants
- Program
- Heap
- Local variables
- Temporaries
- Some parameters
- Global Variables
- Read-only constants
- Program
- Dynamic
- Stack
- Data
- Text
- Unmapped

Memory Addresses:
- 0x40 0000
- 0x800 0000 0000
- 0x0
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
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;
}
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;
}
```c
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;
}
```
movq %rdi, -24(%rbp) //t=0
movq $0, -8(%rbp)    //i=0

movq $1, -32(%rbp)   //temp2 = 1
movq -32(%rbp), %rax
movq %rax, -16(%rbp)  //t = temp2 = 0
movq $0, -32(%rbp)    //set temp2 to 0
movq -32(%rbp), %rax  //store temp2 in %rax
movq %rax, -8(%rbp)   //load %rax to t

movq %rax, -40(%rbp)   //temp3 = i
movq -24(%rbp), %rax  //temp4 = temp1
movq %rax, -48(%rbp)   //temp3 < temp4

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

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

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

movq -16(%rbp), %rax
movq %rax, -40(%rbp)   //temp3 = i
movq $20, -48(%rbp)   //temp4 = 20

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

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

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

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

movq -8(%rbp), %rax
movq %rax, -40(%rbp)   //temp3 = t
movq -16(%rbp), %rax
movq %rax, -48(%rbp)   //temp4 = i

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:
    //t = t + values[i] = temp3 + values[temp4]
    //array access
    mov     -48(%rbp), %r10
    mov     values(, %r10, 8), %rax
    movq    %rax, -48(%rbp)

    movq -40(%rbp), %rax
    add     -48(%rbp), %rax
    movq    %rax, -32(%rbp)

    movq -32(%rbp), %rax
    movq %rax, -8(%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
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)
movq %rax, -48(%rbp)
// t = temp2 = 0
movq $0, -32(%rbp) // set temp2 to 0
movq -32(%rbp), %rax // store temp2 in %rax
movq %rax, -16(%rbp)
// i < n
// temp3 = i
movq -16(%rbp), %rax
movq %rax, -40(%rbp)
// temp4 = temp1
movq %rax, -48(%rbp)
// temp3 < temp4
movq -48(%rbp), %rax
cmp %rax, -40(%rbp)
jge .BasicBlock4
.BasicBlock3:
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
movq %rax, -40(%rbp)
// temp4 = temp1
movq -8(%rbp), %rax
movq %rax, -48(%rbp)
// temp3 < temp4
movq -48(%rbp), %rax
cmp %rax, -40(%rbp)
jge .BasicBlock5
.BasicBlock4:
jmp .BasicBlock5 // jump to condition
.BasicBlock5:
cmp $1, -32(%rbp) // if temp2 is true continue, false jump to return
jne .BasicBlock12
 boundsgood0:
// t = t + values[i] = temp3 + values[temp4]
// array access
mov -48(%rbp), %r10
movq %rax, -40(%rbp)
// temp2 = temp3 + temp4
movq %rax, -32(%rbp)
add -48(%rbp), %rax
movq %rax, -8(%rbp)
// i = temp2
movq -32(%rbp), %rax
movq %rax, -16(%rbp)
.BasicBlock6:
// i < 20
// temp3 = i
movq -16(%rbp), %rax
movq %rax, -40(%rbp)
// temp4 = 20
movq $20, -48(%rbp)
// temp3 < temp4
movq -48(%rbp), %rax
cmp %rax, -40(%rbp)
 jge .BasicBlock8
 boundsbad0:
mov q -48(%rbp), %rdx
movq $8, %rcx
call .boundserror
jge .boundsbad0
jmp .boundsgood0 // perform array access
boundsgood0:
// t = t + values[i] = temp3 + values[temp4]
// array access
mov -48(%rbp), %r10
movq %rax, -40(%rbp)
// temp2 = temp3 + temp4
movq %rax, -32(%rbp)
add -48(%rbp), %rax
movq %rax, -8(%rbp)
// i = temp2
movq -32(%rbp), %rax
movq %rax, -16(%rbp)
.BasicBlock11:
// i = i + 1
// temp3 = i
movq -16(%rbp), %rax
movq %rax, -16(%rbp)
// temp4 = 1
movq $1, -32(%rbp)
movq -32(%rbp), %rax
movq %rax, -48(%rbp)
cmp %rax, 0 // check if array index temp4 < 0
jl .boundsgood0
movq -48(%rbp), %rax
cmp $20, %rax // check if array index temp4 >= 20
jge .boundsbad0
jge .boundsbad0
jmp .boundsgood0 // jump to beginning of while loop
boundssbad0:
jmp .BasicBlock2 // jump to beginning of while loop
.BasicBlock12:
// return t
// temp2 = t
movq -16(%rbp), %rax
movq %rax, -32(%rbp)
// return temp2
movq -32(%rbp), %rax
leave
ret
Allocate space for global variables

Decaf global array declaration
int values[20];

Assembler directive (reserve space in data segment)
.comm   values,160,8

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
  
  \texttt{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
    
    \texttt{leave}

    \texttt{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

//temp4 = 1
mov $1, temp4 on stack

//temp2 = temp3 + temp4
mov temp3 from stack, %rax
add temp4 on stack, %rax
movq %rax, temp2 on stack

//i = temp2
mov temp2 on stack, %rax
movq %rax, i on stack

Temps stored on stack
%rax as working register

Apply code generation templates
temp = var
temp = temp op temp
var = temp
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)

//temp4 = 1
mov $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)

Temps stored on stack
%rax as working register

Apply code generation templates

temp = var
temp = temp op temp
var = temp
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 $\text{ op } z, %rax$
    • Store %rax to $x$

Another option

Load $y$ into register %rax
Load $z$ into register %r10
Perform $\text{ op } %r10, %rax$
Store %rax to $x$
Issues in Lowering Expressions

• Map intermediates to registers?
  – registers are limited
    • When the tree is large, registers may be insufficient ⇒ 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 to values[temp4]
mov array index in temp4, %r10
mov values[array index in %r10], %rax
movq %rax, temp for values[temp4]

%r10 as array index register
%rax as working register

Apply code generation template
Generate code for procedure body

Accessing an array element

//array access to values[temp4]
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 >= array bound
  jge .boundsbad0

  jmp .boundsgood0  //perform array access

.boundsbad0:
  first parameter is array index
  second parameter is array element size
  call .boundsserror

.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
    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 generation template for if then else (conditional branch)

// code for else case

.EndIf:

// code for after if
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
    .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

%rax as working register

Apply code generation template
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 want 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...

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0046</td>
<td>8B45FC</td>
</tr>
<tr>
<td>0049</td>
<td>4863F0</td>
</tr>
<tr>
<td>004c</td>
<td>8B45FC</td>
</tr>
<tr>
<td>004f</td>
<td>4863D0</td>
</tr>
<tr>
<td>0052</td>
<td>8B45FC</td>
</tr>
<tr>
<td>0055</td>
<td>4898</td>
</tr>
<tr>
<td>0057</td>
<td>8B048500</td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>005e</td>
<td>8B149500</td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>0065</td>
<td>01C2</td>
</tr>
<tr>
<td>0067</td>
<td>8B45FC</td>
</tr>
<tr>
<td>006a</td>
<td>4898</td>
</tr>
<tr>
<td>006c</td>
<td>89D7</td>
</tr>
<tr>
<td>006e</td>
<td>033C8500</td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>0075</td>
<td>8B45FC</td>
</tr>
<tr>
<td>0078</td>
<td>4863C8</td>
</tr>
<tr>
<td>007b</td>
<td>8B45F8</td>
</tr>
<tr>
<td>007e</td>
<td>4898</td>
</tr>
<tr>
<td>0080</td>
<td>8B148500</td>
</tr>
</tbody>
</table>
## Machines understand...

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

- **Dynamic**
  - Stack
  - Data
  - Text
  - Unmapped

- **Heap**
  - Local variables
  - Temporaries
  - Some parameters
  - Global Variables
  - Read-only constants
  - Program

Memory Addresses:
- 0x800 0000 0000
- 0x40 0000
- 0x0
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

```assembly
.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 0000 - 0x810 0000 0000
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
• 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
```
• **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
sub rbp, 48
mov %rbx, -8(%rbp)
```

```
<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>previous frame pointer</td>
</tr>
<tr>
<td>calliee saved registers</td>
</tr>
<tr>
<td>local variables</td>
</tr>
<tr>
<td>stack temporaries</td>
</tr>
<tr>
<td>dynamic area</td>
</tr>
<tr>
<td>caller saved registers</td>
</tr>
<tr>
<td>argument 9</td>
</tr>
<tr>
<td>argument 8</td>
</tr>
<tr>
<td>argument 7</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous frame pointer</td>
</tr>
<tr>
<td>calliee saved registers</td>
</tr>
<tr>
<td>local variables</td>
</tr>
<tr>
<td>stack temporaries</td>
</tr>
<tr>
<td>dynamic area</td>
</tr>
</tbody>
</table>
```
- **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
    \[
    \text{sub} \quad 48, \quad \%\text{rsp} \\
    \text{or} \quad \text{enter} \quad 48, \quad 0
    \]
  – Access using \(-8-xx(\%\text{rbp})\)
    \[
    \text{mov} \quad -28(\%\text{rbp}), \quad \%\text{r10} \\
    \text{mov} \quad \%\text{r11}, \quad -20(\%\text{rbp})
    \]
• 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

```assembly
  mov  -8(%rbp), %rbx
  mov  -16(%rbp), %rax
  mov  %rbp, %rsp
  leave
  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 $\text{ op } z, %rax$
      – Store %rax to $x$
Issues in Lowering Expressions

• Map intermediates to registers?
  – registers are limited
    • when the tree is large, registers may be insufficient ⇒ 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>
jmplab_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:
Example Program

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

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

.L0:

.L1:
```

- <FALSE BODY>

```
jmp .L1
```

- <TRUE BODY>

```
```

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

- rbp
- rsp
Example Program

```c
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:

.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 (??)
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

jmp lab_end

lab_body:

<body>

jmp lab_cont

lab_end:
Template for while loops

while (test)
    body

• An optimized template

lab_cont:
    <do the test>
    joper lab_body
    jmp lab_end

lab_body:
    <body>
    jmp lab_cont

lab_end:

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

lab_end:
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

```
entry
  jl xxx
  <
  cmp %r10, %r11
  mov $0, a
  mov x, %r10
  Mov y, %r11
  mov $1, a
exit
```
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

```c
while (i < n && v[i] != 0) {
    i = i+1;
}
```
More Short-Circuit Conditionals

```c
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);

seq
  x
  y

⇒

b_x

⇒
e_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;

seq
  x  y

\[ \rightarrow \]

\[ b_x \rightarrow \]
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);`

```
seq
\x/ \y
  x   y
```

```
seq
  b_x
/    /
\    /
  e_x  b_y
  \    /    
   \  /      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 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;
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 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);
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 \( \text{if } c \ x \ y \)

1: \((b_x,e_x) = \text{destruct}(x);\)
2: \((b_y,e_y) = \text{destruct}(y);\)
3: \(e = \text{new} \ \text{nop};\)
4: \(\text{next}(e_x) = e;\)
5: \(\text{next}(e_y) = e;\)
6: \(b_c = \text{shortcircu}t(c, b_x, b_y);\)
7: \(\text{return } (b_c, e);\)
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 \(\text{while } c \ x\)

1: \(e = \text{new } \text{nop};\)

\[
\begin{array}{c}
\text{while} \\
\downarrow \\
c \\
\uparrow \\
x
\end{array}
\quad \\
\rightarrow \\
\quad \\
e
\]

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 \( \text{while} \ c \ x \)

1: \( e = \text{new nop} \); 2: \((b_x, e_x) = \text{destruct}(x)\);
3: \( b_c = \text{shortcircuit}(c, b_x, e)\);

\[
\text{while} \quad c \quad x \quad \rightarrow \quad b_c \quad b_x \quad e_x \quad e
\]
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);
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 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); 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 $c_1$ && $c_2$

$c_1$ && $c_2$
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 $c_1 \&\& c_2$
1: $b_2 = \text{shortcircuit}(c_2, t, f)$;
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);

\[ c₁ && c₂ \]
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 $c_1 \parallel c_2$
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 $c_1 \ || \ c_2$

1: $b_2 = \text{shortcircuit}(c_2, t, f)$; 2: $b_1 = \text{shortcircuit}(c_1, t, b_2)$;
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: \( b_2 = \text{shortcircuit}(c_2, t, f) \);
2: \( b_1 = \text{shortcircuit}(c_1, t, b_2) \);
3: return \((b_1)\);

\[ c_1 \parallel c_2 \]
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);
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: $b = \text{new cbr}(e_1 < e_2, t, f)$; 2: return (b);
while (i < n && v[i] != 0) {
    i = i+1;
}

entry

jl xxx

jl yyy

cmp %r10, %r11

[jl]

cmp %r10, %r11

mov %r11, i
add $1, %r11
mov i, %r11

exit

nop
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

```c
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