Unoptimized Code Generation
Orientation

- Source code
- Intermediate representation
- Unoptimized assembler
- Executable file
  - Data segments (initialized, zeroed, constant)
  - Code segments
Big Picture

- Starting point – Intermediate Representation
- Ending point – Generated Assembly Code

- Emphasis on UNOPTIMIZED
- Do simplest possible thing for now
- Will treat optimizations separately
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>005e</td>
<td>8B149500</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>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></td>
<td>000000</td>
<td>movl A(,%rdx,4), %edx</td>
</tr>
<tr>
<td>005e</td>
<td>8B149500</td>
<td>movl</td>
</tr>
<tr>
<td></td>
<td>000000</td>
<td>addl %eax, %edx</td>
</tr>
<tr>
<td>0065</td>
<td>01C2</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0067</td>
<td>8B45FC</td>
<td>cltq</td>
</tr>
<tr>
<td>006a</td>
<td>4898</td>
<td>movl %edx, %edi</td>
</tr>
<tr>
<td>006c</td>
<td>89D7</td>
<td>addl C(,%rax,4), %edi</td>
</tr>
<tr>
<td>006e</td>
<td>033C8500</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td></td>
<td>000000</td>
<td>movslq %eax,%rcx</td>
</tr>
<tr>
<td>0075</td>
<td>8B45FC</td>
<td>movl -8(%rbp), %eax</td>
</tr>
<tr>
<td>0078</td>
<td>4863C8</td>
<td>cltq</td>
</tr>
<tr>
<td>007b</td>
<td>8B45F8</td>
<td>movl B(,%rax,4), %edx</td>
</tr>
<tr>
<td>007e</td>
<td>4898</td>
<td></td>
</tr>
<tr>
<td>0080</td>
<td>8B148500</td>
<td></td>
</tr>
</tbody>
</table>
Assembly language

• Advantages
  – Simplifies code generation due to use of symbolic instructions and symbolic names
  – Logical abstraction layer
  – Many different architectures implement same ISA

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

Memory addresses:
- 0x0
- 0x40 0000
- 0x800 0000 0000
int a[10];
int count;

.bss
.global_count:
.zero 8
.global_a:
.zero 80
int PlusOne(int p) {
    int t;
    t = 1;
    return p+t;
}
int increment() {
    count = count + 1;
    return count;
}
```c
int sign(int p) {
    if (p < 0) {
        return -1;
    } else {
        if (p > 0) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

```assembly
.method_sign:
    PUSH_ALL_REGS
    subq $48, %rsp
    movq 128(%rsp), %rax
    movq %rax, 40(%rsp)
    .node_110:
        movq 40(%rsp), %rax
        movq %rax, 32(%rsp)
        movq 32(%rsp), %rax
        movq %rax, 24(%rsp)
        cmpq $0, 24(%rsp)
        movq $0, %rax
        setl %al
        movq %rax, 16(%rsp)
        cmpq $0, 24(%rsp)
        jl .node_111
        jmp .node_112
    .node_112:
        movq 32(%rsp), %rax
        movq %rax, 8(%rsp)
        cmpq $0, 8(%rsp)
        movq $0, %rax
        setg %al
        movq %rax, (%rsp)
        movq $0, %rax
        cmpq 8(%rsp), %rax
        jl .node_113
        jmp .node_114
```
int sign(int p) {
    if (p < 0) {
        return -1;
    } else {
        if (p > 0) {
            return 1;
        } else {
            return 0;
        }
    }
}

.node_114:
    movq $0, 160(%rsp)
    addq $48, %rsp
    POP_ALL_REGS
    ret

.node_113:
    movq $1, 160(%rsp)
    addq $48, %rsp
    POP_ALL_REGS
    ret

.node_111:
    movq $-1, 160(%rsp)
    addq $48, %rsp
    POP_ALL_REGS
    ret
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
```
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
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>_a</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>_b</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>_g</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
Addresses

Reserve Memory

\texttt{.comm\ _a,40,4}\quad\texttt{## @a}
\texttt{.comm\ _b,16,3}\quad\texttt{## @b}
\texttt{.comm\ _g,4,2}\quad\texttt{## @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

0x800 0000 0000
0x810 0000 0000
0x820 0000 0000

Dynamic

Heap

Stack

Data

Text

Unmapped
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
Question:

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

Standard procedure linkage

Procedure \( p \)
- **prolog**
- **pre-call**
- **post-return**
- **epilog**

Procedure \( q \)
- **prolog**
- **epilog**

**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

```assembly
foo:
push %rbp
mov %rsp, %rbp
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
    ```

<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>previous frame pointer</td>
</tr>
<tr>
<td>calliee saved registers</td>
</tr>
</tbody>
</table>

| local variables |

| stack temporaries |

| dynamic area |

<table>
<thead>
<tr>
<th>caller saved registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument 9</td>
</tr>
<tr>
<td>argument 8</td>
</tr>
<tr>
<td>argument 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>previous frame pointer</td>
</tr>
<tr>
<td>calliee saved registers</td>
</tr>
</tbody>
</table>

| local variables |

| stack temporaries |

| dynamic area |

`rbp` — previous frame pointer

`rsp` — return address
• 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 %rbp, %rsp
pop %rbp
leave
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

```assembly
  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 `%r10`
      - Load $z$ into register `%r11`
      - Perform $\text{op } %r10, %r11$
      - Store `%r11` 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>
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:
Example Program

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

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

<jalse> BODY
jmp    .L1

.L0:

<true> BODY
```

- **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)
- Previous frame pointer
- 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 (??)
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
```

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

```
.L0:
```
```
.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

lab_cont:
    <do the test>
    joper lab_body
    jmp lab_end

lab_body:
    <body>
    jmp lab_cont

lab_end:

• An optimized template

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

lab_end:
Question:

• What is the template for?

do
    body
while (test)
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

  \(((i < n) \&\& (v[i] \neq 0)) \; \text{||} \; i > k)\)

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

---

**Short-Circuit Conditionals**

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

```
seq
  x
  y
```

```
  arrow
b_x

b_x
  arrow

next
  ex
  arrow

b_y

next
  ey
  arrow
```
Destructuring Seq Nodes

defstruct(n)

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

1: (bx,ex) = destruct(x); 2: (by,ey) = destruct(y);
3: next(ex) = by; 4: return (bx, ey);
Destructuring If Nodes

destruct(n)

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

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

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: \((b_x, e_x) = \text{destruct}(x)\); 2: \((b_y, e_y) = \text{destruct}(y)\); 3: \( e = \text{new nop}; \)
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: (bₜₚₚ₂, eₜₚₚ₂) = destruct(x); 2: (b₁ₜₚₚ₂, e₁ₜₚₚ₂) = destruct(y);
3: e = new nop; 4: next(e₁ₜₚₚ₂) = e; 5: next(e₁ₜₚₚ₂) = e;
6: bᵡ = shortcircuit(c, bₜₚₚ₂, bₜₚₚ₂);
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;

while c 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 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 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);

while

c

x

b_c

b_x

e

b_c

b_x

e

e_x
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 \( c_1 \land c_2 \)

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

\[ c₁ || c₂ \]
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 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{align*}
1: b_2 &= \text{shortcircuit}(c_2, t, f); \\
2: b_1 &= \text{shortcircuit}(c_1, t, b_2); \\
3: \text{return } (b_1);
\end{align*}

\[c_1 \parallel c_2\]
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);
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 e₁ < e₂
1: b = new cbr(e₁ < e₂, t, f); 2: return (b);
Nops In Destructured Representation

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

entry

jl xxx

jl yyy

<

cmp %r10, %r11

<

cmp %r10, %r11

nop

mov %r11, i

add $1, %r11

mov i, %r11

exit
Eliminating Nops Via Peephole Optimization

[Diagram showing the process of eliminating nops]
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
Outline

• Generation of statements
• Generation of control flow
• x86-64 Processor
• Guidelines in writing a code generator
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 $0 + 1\times x + 0\times y$
  – 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