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
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
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
Source Code
if (condition) {
  code for then
}

AST to CFG for If Then

CFG
  CFG for condition
    CFG for then
      no op
AST to CFG for While

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

CFG

AST
while

AST for condition

AST for loop body

CFG for condition

CFG for loop body

no op
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
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
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;
return s;
i = i + 1;
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
1. Initialize `s = 0`;
2. Set `a = 4`;
3. Set `i = 0`;
4. Check if `k == 0`.
   - If `k == 0`, set `b = 2`;
   - Otherwise, set `b = 1`;
5. Check if `i < n`.
6. If `i < n`, calculate `s = s + a*b`, update `s`, increment `i` by 1, and repeat from step 5.
7. If `i >= n`, return `s`.

The final result is `s`. The diagram shows the flow of the algorithm with conditional paths based on the variables and conditions.
```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 = 2;

i < n

s = s + a*b;
```


```
s = 0;
a = 4;
i = 0;
k == 0
b = 2;
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;
```
\[ s = 0; \]
\[ a = 4; \]
\[ i = 0; \]
\[ k \equiv 0 \]
\[ b = 2; \]
\[ i < n \]
\[ s = s + a \cdot b; \]
\[ i = i + 1; \]
\[ \text{return } s; \]

\[ s = 0; \]
\[ a = 4; \]
\[ i = 0; \]
\[ k \equiv 0 \]
\[ b = 2; \]
\[ i < n \]
\[ s = s + a \cdot b; \]
\[ i = i + 1; \]
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 = 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;

return s;
i = i + 1;

return s;
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 = 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;

return s;

i = i + 1;
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

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

If i < n is false, should you evaluate a[i] != 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
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

```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);
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 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

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

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: \(e = \text{new} \ \text{nop}\); 4: \(\text{next}(e_x) = e\); 5: \(\text{next}(e_y) = e\);
6: \(b_c = \text{shortcircuit}(c, b_x, b_y)\); 7: return \((b_c, e)\);
Destructuring While Nodes

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

while
    c
    x

    b_c
    b_x
    e
    e_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); 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);
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 $c_1 \land c_2$

$c_1 \land c_2$
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);

c₁ && c₂

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

\texttt{shortcircuit}(c, t, f)

generates shortcircuit form of conditional represented by \texttt{c}
returns \texttt{b} - \texttt{b} is begin node of shortcircuit form
if \texttt{c} is of the form \texttt{c}_1 \lor \texttt{c}_2
\begin{enumerate}
\item \texttt{b}_2 = \texttt{shortcircuit}(\texttt{c}_2, t, f);
\item \texttt{b}_1 = \texttt{shortcircuit}(\texttt{c}_1, t, \texttt{b}_2);
\item return (\texttt{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);
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: $b = \text{new } \text{cbr}(e_1 < e_2, t, f)$;
2: return (b);
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

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

Memory Addresses:
- 0x0
- 0x40 0000
- 0x800 0000 0000
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;
}
```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;
}
```
allocate for t, i, temp1, temp2, temp3, temp4

$t = 0$

$i = 0$

$i = temp2 = 1$

$t = temp2 = 0$

set temp2 to 0

store temp2 in %rax

load %rax to t

BasicBlock2:

$i < n$

$temp3 = i$

$temp4 = temp1$

$temp3 < temp4$

$temp2 = true$

jump to condition

$temp2 = false$

$temp2 = false$

$if temp2 is true continue, false jump to return$

$BasicBlock11:

$i < 20$

$temp3 = i$

$temp4 = 20$

$temp3 < temp4$

$temp2 = true$

jump to condition

$temp2 = false$

$if temp2 is true continue, false jump to return$

$BasicBlock10:

$temp3 = t$

$temp4 = i$

$check if array index temp4 < 0$

$if array index temp4 >= 20$

perform array access

boundsbad0:

call .boundserror

boundsgood0:

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

array access

$BasicBlock11:

$i = i + 1$

$temp3 = i$

$temp4 = 1$

$temp2 = temp3 + temp4$

$BasicBlock12:

return $t$

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

    /i = temp2 = 1
    movq $1, -32(%rbp)
    mov -32(%rbp), %rax
    movq %rax, -16(%rbp)
    /i = temp2 = 0
    movq $0, -16(%rbp)

    .BasicBlock2:
        /i < n
        //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 .BasicBlock4
        .BasicBlock3:
            //temp3 = i
            mov -16(%rbp), %rax
            movq %rax, -32(%rbp)
            //temp4 = temp1
            mov -24(%rbp), %rax
            movq %rax, -40(%rbp)
            //temp4 < temp1
            movq -40(%rbp), %rax
            cmp %rax, -48(%rbp)
            jge .BasicBlock5
            jmp .BasicBlock4
        .BasicBlock4:
            //temp2 = true
            jmp .BasicBlock9
        .BasicBlock5:
            //temp2 = false
            jmp .BasicBlock11

        .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:
                //temp2 = true
                jmp .BasicBlock9
            .BasicBlock8:
                //temp2 = false
                jmp .BasicBlock11
            .BasicBlock9:
                cmp $1, -32(%rbp)
                jge .BasicBlock10
                jne .BasicBlock11
        .BasicBlock10:
            //temp3 = i
            mov -16(%rbp), %rax
            movq %rax, -40(%rbp)
            //temp4 = temp1
            mov -24(%rbp), %rax
            movq %rax, -40(%rbp)
            //temp4 = i
            mov -16(%rbp), %rax
            movq %rax, -48(%rbp)
            cmp %rax, $0
            jl .boundsbad0
            movq -48(%rbp), %rax
            mov -48(%rbp), %rax
            cmp $20, %rax
            jge .boundsbad0
            jmp .boundsgood0
            .BasicBlock11:
                //i = i + 1
                .BasicBlock12:
                    //return t
                    movq -8(%rbp), %rax
                    jmp .BasicBlock5
            .BasicBlock5:
                //return temp2
                movq -32(%rbp), %rax
                leave
                ret
            .boundsbad0:
                /i = i + values[i] = temp3 + values[temp4]
                //array access
                mov -48(%rbp), %r10
                movq values(, %r10, 8), %rax
                movq %rax, -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, -8(%rbp)
        .boundsgood0:
            //jump to condition
            jmp .BasicBlock2
            .BasicBlock12:
                //jump to beginning of while loop
                jmp .BasicBlock2
```
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

- `0(%rsp)` return value
- `16(%rbp)` argument 7
- `8(%rbp)` return address
- `8*(%rbp)` argument n
- `-8*(%rbp)` previous %rbp
- `-8*(%rbp)` previous %rbp
- `0(%rbp)` parameter 1
- `0(%rbp)` parameter n
- `-8*(%rbp)` local 1
- `-8*(%rbp)` local m
- `0(%rbp)` local m
- `0(%rbp)` local m
- `0(%rbp)` local m
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

procedure p

prolog

pre-call

post-return

epilog

to

procedure q

prolog

epilog

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 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 >= 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                  Apply code generation template
call   .boundsgood0

.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

```c
//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
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 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>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>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

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

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

Address:

- $0x40 0000$
- $0x800 0000$
- $0x0$

Note: The diagram illustrates the typical memory layout, including sections for global and local variables, temporaries, and mapped and unmapped regions of memory.
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
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]
  - ...
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 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

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

Diagram:
- **Heap**
  - **Dynamic**
  - **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
- %rax
  - return value

<table>
<thead>
<tr>
<th>Previous</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Previous %rbp</td>
<td>local 0</td>
</tr>
<tr>
<td>8*n+16(%rbp)</td>
<td>argument n</td>
</tr>
<tr>
<td>16(%rbp)</td>
<td>Return address</td>
</tr>
<tr>
<td>8(%rbp)</td>
<td></td>
</tr>
<tr>
<td>0(%rbp)</td>
<td></td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td></td>
</tr>
<tr>
<td>-8*m-8(%rbp)</td>
<td></td>
</tr>
<tr>
<td>0(%rsp)</td>
<td></td>
</tr>
<tr>
<td>Variable size</td>
<td></td>
</tr>
</tbody>
</table>
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

### foo:
```assembly
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), %rdx
    ```
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)
• 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
  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

```c
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

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

<do test>

joper .L0

<FALSE BODY>

jmp .L1

.L0:

<TURE BODY>

.L1:
Example Program

```plaintext
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
```

```
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)
```
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:

<TRUE BODY>

.L1:
```
**Example Program**

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16(％rbp)</td>
<td>movq 16(％rbp), ％r10</td>
<td>Load register ％r10</td>
</tr>
<tr>
<td>24(％rbp)</td>
<td>movq 24(％rbp), ％r11</td>
<td>Load register ％r11</td>
</tr>
<tr>
<td></td>
<td>cmpq ％r10, ％r11</td>
<td>Compare registers ％r10 and ％r11</td>
</tr>
<tr>
<td></td>
<td>jg .L0</td>
<td>Jump if greater than ％r10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24(％rbp)</td>
<td>movq 24(％rbp), ％r10</td>
<td>Load register ％r10</td>
</tr>
<tr>
<td>16(％rbp)</td>
<td>movq 16(％rbp), ％r11</td>
<td>Load register ％r11</td>
</tr>
<tr>
<td></td>
<td>subq ％r10, ％r11</td>
<td>Subtract ％r10 from ％r11</td>
</tr>
<tr>
<td></td>
<td>movq ％r11, -8(％rbp)</td>
<td>Store register ％r11, offset 8</td>
</tr>
<tr>
<td></td>
<td>jmp .L1</td>
<td>Jump to label .L1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16(％rbp)</td>
<td>movq 16(％rbp), ％r10</td>
<td>Load register ％r10</td>
</tr>
<tr>
<td>24(％rbp)</td>
<td>movq 24(％rbp), ％r11</td>
<td>Load register ％r11</td>
</tr>
<tr>
<td></td>
<td>subq ％r10, ％r11</td>
<td>Subtract ％r10 from ％r11</td>
</tr>
<tr>
<td></td>
<td>movq ％r11, -8(％rbp)</td>
<td>Store register ％r11, offset 8</td>
</tr>
</tbody>
</table>

- **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)**
- **Local variable dx (??)**
- **Local variable dy (??)**
- **Local variable dz (??)**
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?

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

• 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 $((i < n) \land (v[i] \neq 0))$ is false

• Use control-flow graph to represent this short-circuit evaluation
Short-Circuit Conditionals

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

More Short-Circuit Conditionals

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

```assembly
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 \( \text{seq } x \; y \)
1: \((b_x,e_x) = \text{destruct}(x)\);
2: \((b_y,e_y) = \text{destruct}(y)\);
3: next(e_x) = b_y;

\[
\begin{array}{c}
\text{seq} \\
\downarrow \quad \downarrow \\
x & y
\end{array}
\quad \rightarrow 
\begin{array}{c}
b_x \\
\quad \rightarrow \\\ne_x & b_y
\end{array}
\quad \rightarrow 
\begin{array}{c}
e_x \\
\quad \rightarrow \\
e_y
\end{array}
\]
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) = 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 \(\text{if } c \ x \ y\)

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

\[
\begin{align*}
&\text{if } c \xrightarrow{x} x \xrightarrow{y} y \\
&\text{destruct(n)} \quad \text{generates lowered form of structured code represented by n} \\
&\text{return } (b,e) - b \text{ is begin node, } e \text{ is end node in destructed form} \\
&\text{if } n \text{ is of the form if } c \xrightarrow{x} x \xrightarrow{y} y \\
&\quad 1: (b_x,e_x) = \text{destruct}(x); 2: (b_y,e_y) = \text{destruct}(y); \\
&\quad 3: e = \text{new nop}; 4: \text{next}(e_x) = e; 5: \text{next}(e_y) = e; \\
&\quad 6: b_c = \text{shortcircuit}(c, b_x, b_y);
\end{align*}
\]
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;

\[
\text{while} \quad c \quad x \quad \rightarrow \\
\text{e}
\]
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: $e = \text{new nop}$; 2: $(b_x,e_x) = \text{destruct}(x)$;
3: $b_c = \text{shortcircuit}(c, b_x, e)$;

```latex
\textbf{while} \\
c \quad x
```

```latex
\textbf{while} \\
c \quad x
```

\[ b_c \]

\[ b_x \]

\[ e \]

\[ e_x \]
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); 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);
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 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: \( b₂ = \text{shortcircuit}(c₂, t, f) \); 2: \( b₁ = \text{shortcircuit}(c₁, b₂, f) \);
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 $c_1 \&\& c_2$

1: $b_2 = \text{shortcircuit}(c_2, t, f)$; 2: $b_1 = \text{shortcircuit}(c_1, b_2, f)$;
3: return $(b_1)$;
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 \( c_1 \mid\mid c_2 \)
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: \( b_2 = \) shortcircuit(c₂, t, f); 2: \( b_1 = \) shortcircuit(c₁, t, b₂);

\[ c_1 \| c_2 \]
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: $b_2 = \text{shortcircuit}(c_2, t, f)$; 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 ! c_1

1: b = shortcircuit(c_1, 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;
}

Nops In Destructured Representation
Eliminating Nops Via Peephole Optimization

![Diagram of eliminating Nops via peephole optimization.]

The diagram shows a series of operations before and after the reduction of a `nop` instruction, illustrating how peephole optimization simplifies the code.
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