

**Winter 2006-2007
Compiler Construction
T12 – Code Generation**

Mooly Sagiv and Roman Manevich
School of Computer Science
Tel-Aviv University

Notes

- Weighted register allocation in LIR
 - Can use for all expression types (not just for commutative operators)
 - Get to know LIR optimization algorithms for exam
- Must reset LIR register counter after every IC statement:
 - `x=y; y=z:` should not be translated to


```
Move y,R0
Move R0,x
Move z,R1
Move R1,y
```
 - But rather to


```
Move y,R0
Move R0,x # Finished translating statement. Set c=0
Move z,R0
Move R0,y
```

2

Weighted register allocation

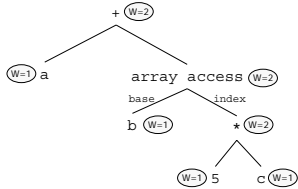
- Can save registers by re-ordering subtree computations
- Label each node with its weight
 - Weight = number of registers needed
 - Leaf weight known
 - Internal node weight
 - $w(\text{left}) > w(\text{right})$ then $w = \text{left}$
 - $w(\text{right}) > w(\text{left})$ then $w = \text{right}$
 - $w(\text{right}) = w(\text{left})$ then $w = \text{left} + 1$
- Choose heavier child as first to be translated
- WARNING: have to check that no side-effects exist before attempting to apply this optimization (pre-pass on the tree)

3

Weighted reg. alloc. example

$R0 := TR[a+b[5*c]]$

Phase 1: - check absence of side-effects in expression tree
- assign weight to each AST node

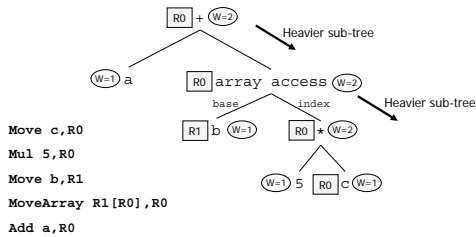


4

Weighted reg. alloc. example

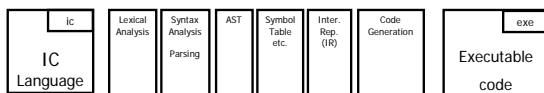
$R0 := TR[a+b[5*c]]$

Phase 2: use weights to decide on order of translation



5

Today



- Today:
 - Assembling and linking
 - Code generation
 - Runtime checks
 - Optimizations
 - Labels/Jumps
 - Register allocation for basic blocks
- Next time:
 - PA5
 - Recap
 - Final exam practice questions

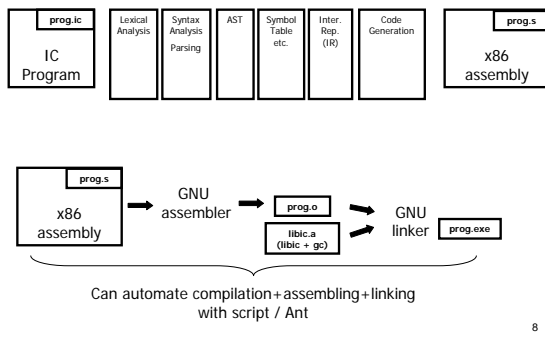
6

Intel IA-32 assembly

- Going from assembly to binary
 - Assembler tool: GNU assembler (**as**)
 - Linker tool: GNU linker (**ld**)
- Use Cygwin on Windows
 - IMPORTANT: select binutils and gcc when installing cygwin
 - Tools usually pre-exist on Linux environment
- Supporting materials for PA5 on web site

7

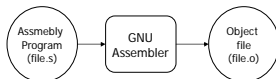
From assembly to executable



8

From assembly file to object file

- How do you get an object file from generated .s file? (Under Windows using Cygwin)

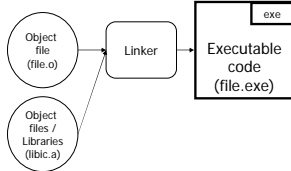


```
as -o file.o file.s
```

9

From object file to executable File

- How do you get an exe file from generated .o file? (Under Windows using Cygwin)



```
ld -o file.exe file.o /lib/crt0.o libc.a -lcygwin -lkernel32
```

IMPORTANT: don't change order of arguments

10

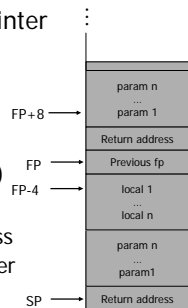
LIR to assembly

- Need to know how to translate:
 - Function bodies
 - Translation for each kind of LIR instruction
 - Calling sequences
 - Correctly access parameters and variables (and LIR register)
 - Compute offsets for parameter and variables
 - Dispatch tables
 - String literals
 - Runtime checks
 - Error handlers

11

Reminder: accessing variables

- Use offset from frame pointer
- Remember – stack grows downwards
- Above EBP = parameters
- Below EBP = locals (and spilled LIR registers)
- Examples
 - $\%ebp + 4$ = return address
 - $\%ebp + 8$ = first parameter
 - $\%ebp - 4$ = first local



12

Translating LIR instructions

- Translate function bodies:
 1. Compute offsets for:
 - Local variables (-4,-8,-12,...)
 - LIR registers (considered extra local variables)
 - Function parameters (+8,+12,+16,...)
 - Take `this` parameter into account
 2. Translate instruction list for each function
 - Local translation for each LIR instruction
 - Local (machine) register allocation

13

Memory offsets implementation

```
// MethodLayout instance per function declaration
class MethodLayout {
  // Maps variables/parameters/LIR registers to
  // offsets relative to frame pointer (BP)
  Map<Memory,Integer> memoryToOffset;
}
```

virtual function takes one extra parameter: `this`

```
void foo(int x, int y) {
  int z = x + y;
  g = z; // g is a field
  Library.printi(z);
}
```

(manual) LIR translation

```
_A.foo:
  Move x,R0
  Add y,R0
  Move R0,z
  Move this,R1
  MoveField R0,R1.1
  Library.__printi(R0),Rdummy
```

① MethodLayout for foo

Memory	Offset
<code>this</code>	+8
<code>x</code>	+12
<code>y</code>	+16
<code>z</code>	-4
<code>R0</code>	-8
<code>R1</code>	-12

PA4

14

Memory offsets example

LIR translation (optimized)

```
_A.foo:
  Move x,R0
  Add y,R0
  Move R0,z
  Move this,R1
  MoveField R0,R1.1
  Library.__printi(R0),Rdummy
```

MethodLayout for foo

Memory	Offset
<code>this</code>	+8
<code>x</code>	+12
<code>y</code>	+16
<code>z</code>	-4
<code>R1</code>	-8
<code>R2</code>	-12

② Translation to x86 assembly (non-optimized)

```
_A.foo:
  push %ebp          # prologue
  mov %esp,%ebp
  mov 12(%ebp),%eax  # Move x,R1
  mov %eax,-8(%ebp)
  mov 16(%ebp),%eax  # Add y,R1
  add -8(%ebp),%eax
  mov %eax,-8(%ebp)
  mov -8(%ebp),%eax  # Move R1,z
  mov %eax,-4(%ebp)
  mov 8(%ebp),%eax   # Move this,R2
  mov %eax,-12(%ebp)
  mov -8(%ebp),%eax  # MoveField R1,R2.1
  mov -12(%ebp),%ebx
  mov %eax,8(%ebx)
  mov -8(%ebp),%eax  # Library.__printi(R1)
  push %eax
  call __printi
  add $4,%esp
_A.foo_epilogue:
  mov %ebp,%esp
  pop %ebp
  ret
```

15

Instruction-specific register allocation

- Non-optimized translation
- Each non-call instruction has fixed number of variables/registers
 - Naïve (very inefficient) translation
 - Use direct algorithm for register allocation
- Example: **Move x, R1** translates into

```

move x_offset(%ebp), %ebx
move %ebx, R1_offset(%ebp)
    
```

Register hard-coded in translation

16

Translating instructions 1

LIR Instruction	Translation
MoveArray R1[R2], R3	<pre> mov -8(%ebp), %ebx # -8(%ebp)=R1 mov -12(%ebp), %ecx # -12(%ebp)=R2 mov (%ebx,%ecx,4), %ebx mov %ebx, -16(%ebp) # -16(%ebp)=R3 </pre>
MoveField x, R2.3	<pre> mov -12(%ebp), %ebx # -12(%ebp)=R2 mov -8(%ebp), %eax # -12(%ebp)=x mov %eax, 12(%ebx) # 12=3*4 </pre>
MoveField _DV_A, R1.0	<pre> movl \$_DV_A, (%ebx) # (%ebx)=R1.0 (movl means move 4 bytes) </pre>
ArrayLength y, R1	<pre> mov -8(%ebp), %ebx # -8(%ebp)=y mov -4(%ebx), %ebx # load size mov %ebx, -12(%ebp) # -12(%ebp)=R1 </pre>
Add R1, R2	<pre> mov -16(%ebp), %eax # -16(%ebp)=R1 add -20(%ebp), %eax # -20(%ebp)=R2 mov %eax, -20(%ebp) # store in R2 </pre>

17

Translating instructions 2

LIR Instruction	Translation
Mul R1, R2	<pre> mov -8(%ebp), %eax # -8(%ebp)=R2 imul -4(%ebp), %eax # -4(%ebp)=R1 mov %eax, -8(%ebp) </pre>
Div R1, R2 (idiv divides EDX:EAX stores quotient in EAX stores remainder in EDX)	<pre> mov \$0, %edx mov -8(%ebp), %eax # -8(%ebp)=R2 mov -4(%ebp), %ebx # -4(%ebp)=R1 idiv %ebx mov %eax, -8(%ebp) # store in R2 </pre>
Mod R1, R2	<pre> mov \$0, %edx mov -8(%ebp), %eax # -8(%ebp)=R2 mov -4(%ebp), %ebx # -4(%ebp)=R1 idiv %ebx mov %edx, -8(%ebp) </pre>
Compare R1, x	<pre> mov -4(%ebp), %eax # -4(%ebp)=x cmp -8(%ebp), %eax # -8(%ebp)=R1 </pre>
Return R1 (returned value stored in EAX register)	<pre> mov -8(%ebp), %eax # -8(%ebp)=R1 jmp _A_foo_epilogue </pre>
Return Rdummy	<pre> # return; jmp _A_foo_epilogue </pre>

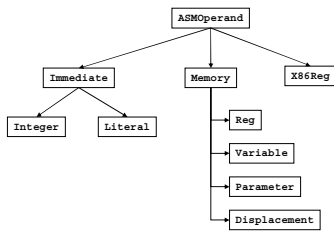
18

Implementation

- Hacker's approach: translate directly to strings
 - Prohibits any further optimizations
 - Prohibits sanity checks – expect more bugs
 - Not flexible (what if we want to support Intel syntax?)
- Create classes for
 - Assembly instruction: **ASMinstr**
 - Operand types:
 - LIR register
 - Actual register
 - Local variable
 - Parameter
 - Immediate
 - Memory displacement

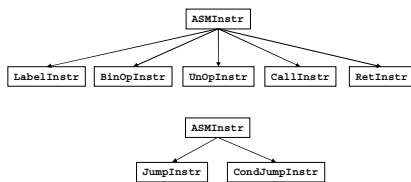
19

Possible operand hierarchy



20

Possible instruction hierarchy



```
abstract class ASMinstr {  
    public abstract List<ASMOperand> getOperands();  
    public abstract List<X86Reg> getRegisters();  
}
```

21

Implementation example

```

class BinOpInstr extends ASMinstr {
public final Operand src;
public final Operand dst;
public final String name;
public BinOpInstr(String name, Operand src, Operand dst) {
// Sanity check
// (src instanceof Memory &&
//  dst instanceof Memory) {
throw new RuntimeException("Too many memory operands for arith instruction!");
this.name = name;
this.src = src;
this.dst = dst;
}
public String toString() {
if (ASMinstr.isIntelSyntax()) {
return name + " " + src + ", " + dst;
}
else {
return name + " " + dst + ", " + src;
}
}
}

```

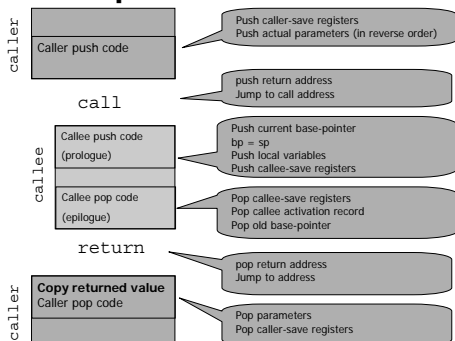
22

Handling functions

- Need to implement call sequence
 - Caller code:
 - Pre-call code:
 - Push callee-save registers
 - Push parameters
 - Call (special treatment for virtual function calls)
 - Post-call code:
 - Copy returned value (if needed)
 - Pop parameters
 - Pop callee-save registers
 - Callee code
 - Each function has prologue and epilogue

23

Call sequences



24

Translating static calls

LIR code: `StaticCall _A_foo(a=R1,b=5,c=x),R3`

```

# push caller-saved registers
push %eax
push %ecx
push %edx

# push parameters
mov -4(%ebp),%eax # push x
push %eax
push $5           # push 5
mov -8(%ebp),%eax # push R1
push %eax

call _A_foo

mov %eax,-16(%ebp) # store returned value in R3

# pop parameters (3 params*4 bytes = 12)
add $12,%esp

# pop caller-saved registers
pop %edx
pop %ecx
pop %eax
    
```

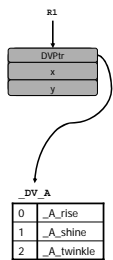
Optional: only if register allocation optimization is used (in PA5)

Only if return register is not %edx

25

Translating virtual calls

LIR code: `VirtualCall R1.2(a=R1,b=5,c=x),R3`



```

# push caller-saved registers
push %eax
push %ecx
push %edx

# push parameters
mov -4(%ebp),%eax # push x
push %eax
push $5           # push 5
mov -8(%ebp),%eax # push R1
push %eax

# Find address of virtual method and call it
mov -8(%ebp),%eax # load this
push %eax        # push this
mov 0(%eax),%eax # Load dispatch table address
call *8(%eax)    # Call table entry 2 (2*4=8)

mov %eax,-16(%ebp) # store returned value in R3

# pop parameters (3 params+this * 4 bytes = 16)
add $16,%esp

# pop caller-saved registers
pop %edx
pop %ecx
pop %eax
    
```

26

Library function calls

- Same as static function calls
- Reference type arguments require non-null runtime check (explained soon)

27

Function prologue/epilogue

Optional: only if register allocation optimizations is used (in PAs)

```

_A_foo:
# prologue
push %ebp
mov %esp,%ebp

# push local variables of foo
sub $12,%esp # 3 local vars+regs * 4 = 12

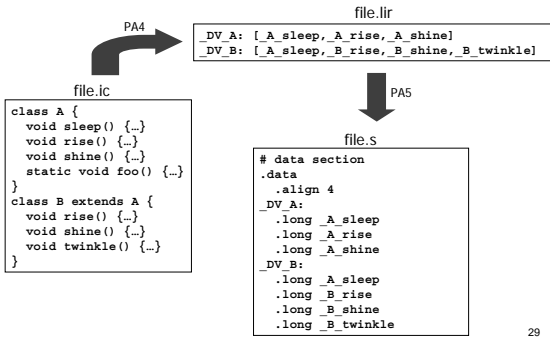
# push callee-saved registers
push %ebx
push %esi
push %edi

function body

_A_foo_epilogue: # extra label for each function
# pop callee-saved registers
pop %edi
pop %esi
pop %ebx

mov %ebp,%esp
pop %ebp
ret
    
```

Representing dispatch tables



Runtime checks

- Insert code to check attempt to perform illegal operations
 - Null pointer check
 - MoveField, MoveArray, ArrayLength, VirtualCall
 - Reference arguments to library functions should not be null
 - Array bounds check
 - MoveArray
 - Array allocation size check
 - `__allocateArray(size)` # size needs to be >0
 - Do we need to check argument of `__allocateObject?`
 - Division by zero
 - Mul, Mod
- If check fails jump to error handler code that prints a message and gracefully exits program

Null pointer check

```
# null pointer check
cmp $0,%eax
je labelNPE
```

Single generated handler for entire program

```
labelNPE:
push $strNPE # error message
call __println
push $1 # error code
call __exit
```

31

Array bounds check

```
# array bounds check
mov -4(%eax),%ebx # ebx = length
mov $0,%ecx # ecx = index
cmp %ecx,%ebx
jle labelABE # ebx <= ecx ?
cmp $0,%ecx
jl labelABE # ecx < 0 ?
```

Single generated handler for entire program

```
labelABE:
push $strABE # error message
call __println
push $1 # error code
call __exit
```

32

Array allocation size check

```
# array size check
cmp $0,%eax # eax == array size
jle labelASE # eax <= 0 ?
```

Single generated handler for entire program

```
labelASE:
push $strASE # error message
call __println
push $1 # error code
call __exit
```

33

Division by zero check

```
# division by zero check
cmp $0,%eax    # eax is divisor
je labelDBE   # eax == 0 ?
```

Single generated handler for entire program

```
labelDBE:
  push $strDBE # error message
  call __println
  push $1      # error code
  call __exit
```

34

Error handlers code

```
.data
.int 40
strDBE: .string "Runtime error: Null pointer dereference."
.int 41
strAIBE: .string "Runtime error: Array index out of bounds!"
.int 50
strAABE: .string "Runtime error: Array allocation with negative array size!"
.int 51
strDBE: .string "Runtime error: Division by zero!"
.text
#----- Runtime error handlers -----
.align 4
labelDBE:
  push $strDBE
  call __println
  push $1
  call __exit
  .align 4
labelAIBE:
  push $strAIBE
  call __println
  call __println
  push $1
  call __exit
  .align 4
labelAABE:
  push $strAABE
  call __println
  push $1
  call __exit
  .align 4
labelDBE:
  push $strDBE
  call __println
  push $1
  call __exit
```

35

Optimizations

- More efficient register allocation for statements
 - Allocate machine registers during translation
- Eliminate unnecessary labels and jumps
 - Post-translation pass

36

Optimizing labels/jumps

- If we have subsequent labels:
_label1:
_label2:
- We can merge labels and redirect jumps to the merged label
- After translation (easier)
 - Map old labels to new labels
- If we have
jump label1
_label1:
Can eliminate jump
- Eliminate labels not mentioned by any instruction

37

Optimizing register allocation

- Goal: associate machine registers with LIR registers as much as possible
- Optimization done only for sequence of instructions translated from single statement
- See more details on [web site](#)

38

See you next week

39
