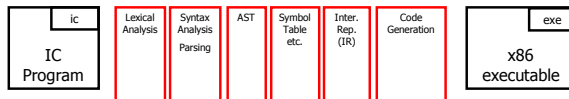


Compiler Construction

Code Generation II

Ran Shaham and Ohad Shacham
School of Computer Science
Tel-Aviv University

IC compiler Compiler



We saw:

- X86 assembly
- Code generation

Today:

- Code generation
- Runtime checks

2

Test

- 25/02/2009 at 9:00
- Recap class
 - February 22nd (Tentative)
 - Send me questions in advanced
- Look at Ran's webpage for previous tests

3

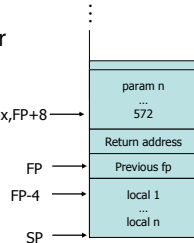
Memory and base displacement operands

- Memory operands
 - Obtain value at given address
 - Example: `mov (%eax), %eax`
- Base displacement
 - Obtain value at computed address
 - Syntax: `disp(base,index,scale)`
 - $\text{offset} = \text{base} + (\text{index} * \text{scale}) + \text{displacement}$
 - Example: `mov $42, 2(%eax)`
 - Example: `mov $42, (%eax,%ecx,4)`

7

Reminder: accessing variables

- Use offset from frame pointer
- Above FP = parameters
- Below FP = locals (and spilled LIR registers)
- Examples
 - $\%ebp + 8 =$ first parameter
 - $\%eax = \%ebp + 8$
 - $(\%eax) =$ the value 572
 - $8(\%ebp) =$ the value 572



8

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
 - Compute offsets for parameter and variables
 - Dispatch tables
 - String literals
 - Runtime checks
 - Error handlers

9

Translating LIR instructions

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

10

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
this	+8
x	+12
y	+16
z	-4
R0	-8
R1	-12

PA4

11

Memory offsets example

LIR translation

```
_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

Memor	Offset
y	+8
this	+8
x	+12
y	+16
z	-4
R0	-8
R1	-12

② Translation to x86 assembly

```
_A_foo:
    push %ebp # prologue
    mov %esp,%ebp
    mov 12(%ebp),%eax # Move x,R0
    mov %eax,-8(%ebp)
    mov 16(%ebp),%eax # Add y,R0
    add -8(%ebp),%eax
    mov %eax,-8(%ebp)
    mov -8(%ebp),%eax # Move R0,z
    mov %eax,-4(%ebp)
    mov 8(%ebp),%eax # Move this,R1
    mov %eax,-12(%ebp)
    mov -8(%ebp),%eax # MoveField R0,R1.1
    mov -12(%ebp),%ebx
    mov %eax,8(%ebx)
    mov -8(%ebp),%eax # Library.__printi(R0)
    push %eax
    call __printi
    add $4,%esp
_A_foo_epilogue:
    mov %ebp,%esp # epilogue
    pop %ebp
    ret
```

12

Calls/returns

- Direct function call syntax: call name
 - Example: call `__println`
- Return instruction: `ret`

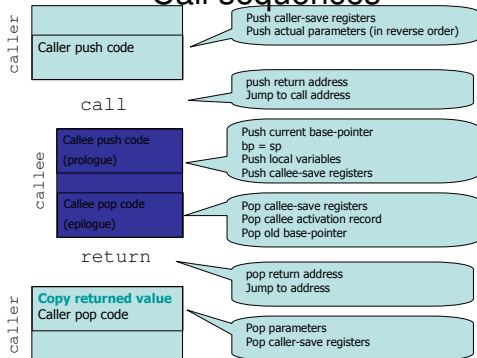
13

Handling functions

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

14

Call sequences



15

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
    
```

only if the value stored in these registers is needed by the caller

Only if return register is not Rdummy

16

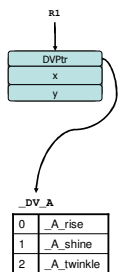
Virtual functions

- Indirect call: `call *(Reg)`
 - Example: `call *(%eax)`
 - Used for virtual function calls
- Dispatch table lookup
- Passing/receiving the `this` variable

17

Translating virtual calls

LIR code: `VirtualCall R1.2(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

# 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 *(%eax) # Call table entry 2 (2*4=8)

mov %eax, -12(%ebp) # store returned value in R3

# pop parameters (2 params+this * 4 bytes = 12)
add $12, %esp

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

18

Function prologue/epilogue

only if these registers will be modified by the callee

```

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

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

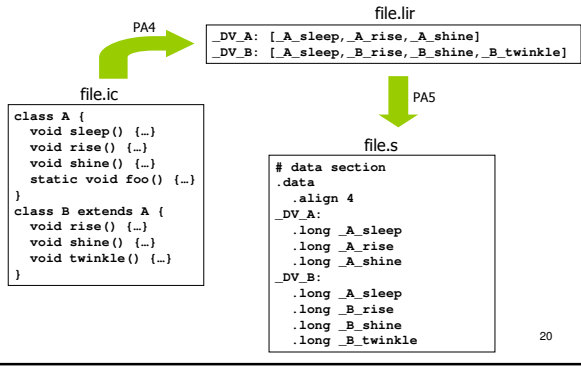
# push callee-saved registers
push %ebx
push %esi
push %edi

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
 - Array allocation size check
 - Division by zero
- If check fails jump to error handler code that prints a message and gracefully exists 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
```

22

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
```

23

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
```

24

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
```

25

Optimizations

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

26

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

27

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

28

Hello world example

```
class Library {
  void println(string s);
}

class Hello {
  static void main(string[] args) {
    Library.println("Hello world!");
  }
}
```

29

Assembly file structure

```
header {
  .title "hello.ic"
  # global declarations
  global __ic_main
}

statically-allocated
data: string literals
and dispatch tables {
  # data section
  .data
  .align 4
  .int 13
  str1: .string "Hello world\n"
}

Method bodies
and error handlers {
  # text (code) section
  .text
  -----
  .align 4
  __ic_main:
    push %ebp          # prologue
    mov %esp,%ebp
    # print(...)
    push $str1
    call __print
    add $4, %esp

    mov $0,%eax       # return 0

    mov %ebp,%esp
    pop %ebp
    ret               # epilogue
}
```

30

Assembly file structure

```

.title "hello.ic"
# global declarations
.global __ic_main

# data section
.data
.align 4
.int 13
str1: .string "Hello world\n"

# text (code) section
.text
#-----
.align 4
__ic_main:
    push %ebp          # prologue
    mov %esp,%ebp
    push $str1        # print(...)
    call __print
    add $4, %esp      # pop parameter
    mov $0,%eax      # return 0
    mov %ebp,%esp    # epilogue - restore
    pop %ebp         esp and ebp (pop)
    ret
    
```

• Immediates have \$ prefix
 • Register names have % prefix
 • Comments using #
 • Labels end with the (standard) :

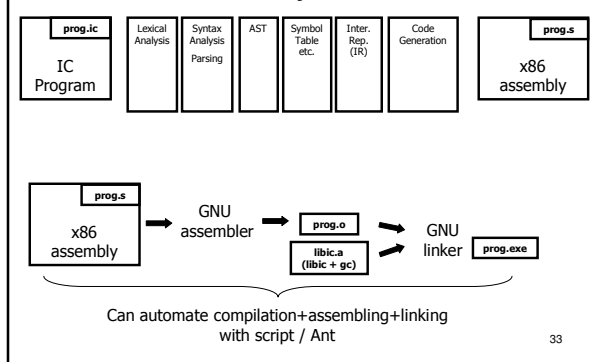
prologue - save ebp and set to be esp
 push print parameter
 call print
 store return value of main in eax
 pop parameter
 epilogue - restore esp and ebp (pop)

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

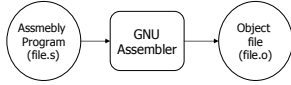
32

From assembly to executable



33

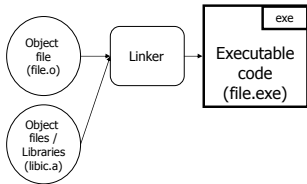
From assembly file to object file



as -o file.o file.s

34

From object file to executable File



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

IMPORTANT: don't change order of arguments

35
