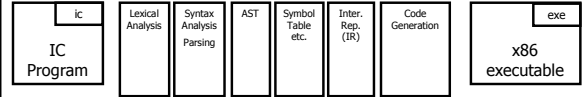


Compiler Construction

Code Generation II

Rina Zvi-Girshin and Ohad Shacham
 School of Computer Science
 Tel-Aviv University

IC compiler Compiler



We saw:

- X86 assembly
- Code generation

Today:

- Code generation
- Runtime checks

x86 assembly

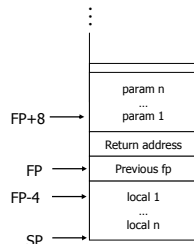
- AT&T syntax and Intel syntax
- We'll be using AT&T syntax
- Work with GNU Assembler (GAS)

Immediate and register operands

- Immediate
 - Value specified in the instruction itself
 - Preceded by \$
 - Example: `add $4, %esp`
- Register
 - Register name is used
 - Preceded by %
 - Example: `mov %esp, %ebp`

Reminder: accessing variables

- Use offset from frame pointer
- Above FP = parameters
- Below FP = locals (and spilled LIR registers)
- Examples
 - `%ebp + 4` = return address
 - `%ebp + 8` = first parameter
 - `%ebp - 4` = first local

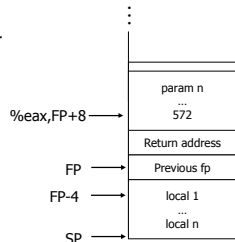


Memory and base displacement operands

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

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



7

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

8

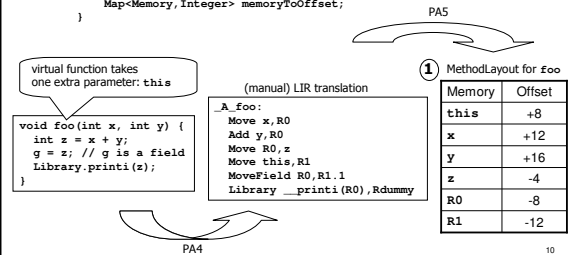
Translating LIR instructions

- 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

9

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;
}
```



10

Memory offsets example

② Translation to x86 assembly

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

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

```
_A_foo:
push %ebp # prologue
mov %esp,%ebp
sub $12,%esp
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,4(%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
```

11

Calls/returns

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

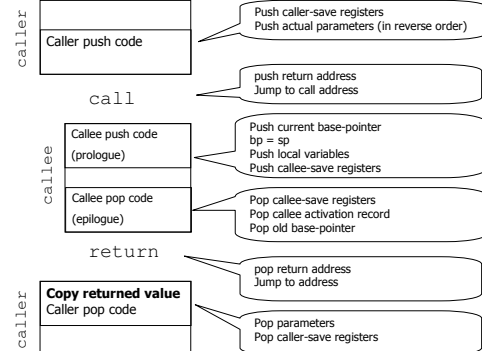
12

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

13

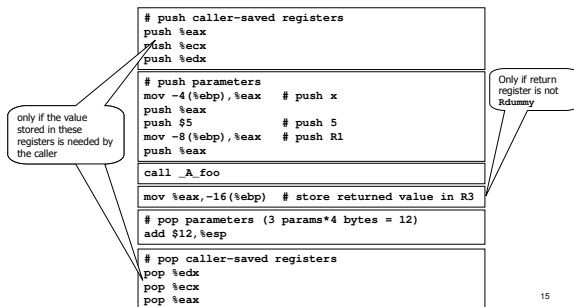
Call sequences



14

Translating static calls

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



15

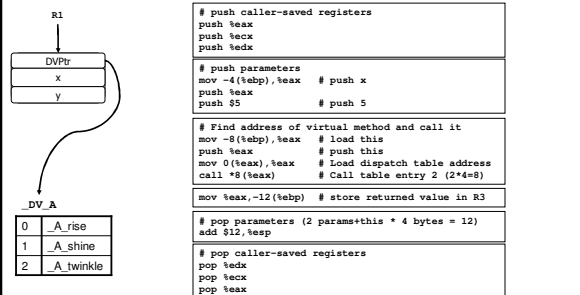
Virtual functions

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

16

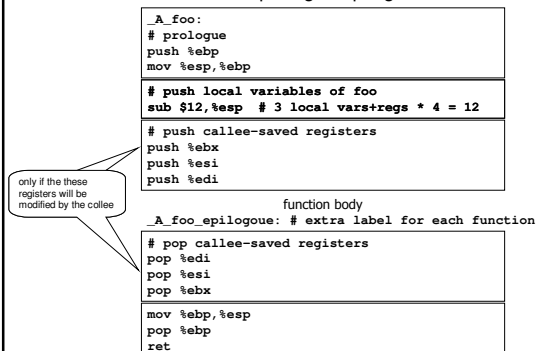
Translating virtual calls

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



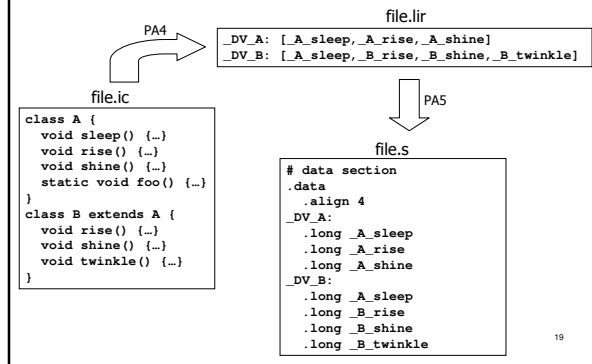
17

Function prologue/epilogue



18

Representing dispatch tables



19

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 exits program

20

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
    
```

21

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
    
```

22

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
    
```

23

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
    
```

24

Optimizations

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

25

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

26

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

27

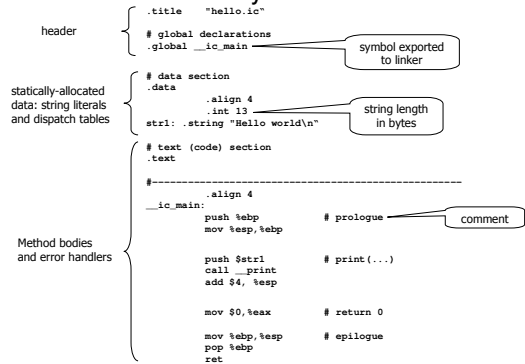
Hello world example

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

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

28

Assembly file structure



29

Assembly file structure

