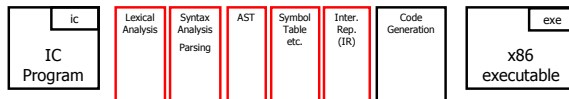


Compiler Construction

Code Generation I

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

IC compiler Compiler



We saw:

- Activation records

Today:

- X86 assembly
- Code generation
- Runtime checks

2

x86 assembly

- AT&T syntax and Intel syntax
- We'll be using AT&T syntax
- Work with [GNU Assembler \(GAS\)](#)
Summary of differences

	AT&T	Intel
Order of operands	op a, b means b = a op b (second operand is destination)	op a, b means a = a op b (first operand is destination)
Memory addressing	disp(base, offset, scale)	[base + offset * scale + disp]
Size of memory operands	instruction suffixes (b, w, l) (e.g., movb, movw, movl)	operand prefixes (e.g., byte ptr, word ptr, dword ptr)
Registers	%eax, %ebx, etc.	eax, ebx, etc.
Constants	\$4, \$foo, etc	4, foo, etc

3

IA-32

- Eight 32-bit general-purpose registers
 - EAX, EBX, ECX, EDX, ESI, EDI
 - EBP – stack frame (base) pointer
 - ESP – stack pointer
- EFLAGS register
 - info on results of arithmetic operations
- EIP (instruction pointer) register
- Machine-instructions
 - add, sub, inc, dec, neg, mul, ...

4

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`

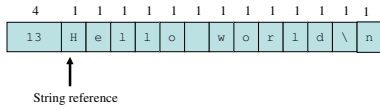
5

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)`

6

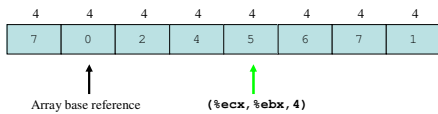
Representing strings and arrays



- Array preceded by a word indicating the length of the array
- Project-wise
 - String literals allocated statically, concatenation using `__stringCat`
 - `__allocateArray` allocates arrays

7

Base displacement addressing



```
mov (%ecx, %ebx, 4), %eax    %ecx = base
                             %ebx = 3
offset = base + (index * scale) + displacement
offset = %ecx + (3*4) + 0 = %ecx + 12
```

8

Instruction examples

- Translate `a=p+q` into
 - `mov 16(%ebp), %ecx` (load p)
 - `add 8(%ebp), %ecx` (arithmetic p + q)
 - `mov %ecx, -8(%ebp)` (store a)
- Accessing strings:
 - `str: .string "Hello world!"`
 - `push $str`

9

Instruction examples

- Array access: `a[i]=1`
 - `mov -4(%ebp), %ebx` (load a)
 - `mov -8(%ebp), %ecx` (load i)
 - `mov $1, (%ebx, %ecx, 4)` (store into the heap)
- Jumps:
 - Unconditional: `jmp label2`
 - Conditional: `cmp $0, %ecx`
`jnz cmpFailLabel`

10

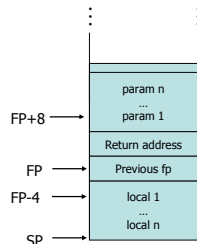
LIR to assembly

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

11

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



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
```

MethodLayout for foo

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



PA4

14

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

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

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) # Move R0,z
  mov -8(%ebp),%eax
  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
```

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

Calls/returns

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

19

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

20
