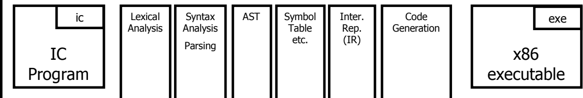
Compiler Construction

Activation records
Code Generation

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

IC compiler



We saw:
- IR
- Activation records

Today:
- Activation records
- X86 assembly
- Code generation
- Runtime checks

PA4

- PA4 is up
- Submission deadline 13/01/2010

Function calls

- Conceptually
  - Supply new environment (frame) with temporary memory for local variables
  - Pass parameters to new environment
  - Transfer flow of control (call/return)
  - Return information from new environment (ret. value)

Activation records

- New environment = activation record (a.k.a. frame)
- Activation record = data of current function / method call
  - User data
    - Local variables
    - Parameters
    - Return values
    - Register contents
  - Administration data
    - Code addresses

Runtime stack

- Stack of activation records
- Call = push new activation record
- Return = pop activation record
- Only one “active” activation record – top of stack

Runtime stack

- Stack grows downwards (towards smaller addresses)
- SP – stack pointer
  – top of current frame
- FP – frame pointer
  – base of current frame
  - Sometimes called BP (base pointer)

Previous frame

FP → Current frame

SP →

Call sequences

- The processor does not save the content of registers on procedure calls
- So who will?
  - Caller saves and restores registers
    - Caller's responsibility
  - Callee saves and restores registers
    - Callee's responsability

Call sequences

caller

Caller push code — Push caller-save registers; Push actual parameters (in reverse order)

call — push return address; Jump to call address

callee

Callee push code (prologue) — Push current base-pointer; bp = sp; Push local variables; Push callee-save registers

Callee pop code (epilogue) — Pop callee-save registers; Pop callee activation record; Pop old base-pointer

return — pop return address; Jump to address

caller

Caller pop code — Pop parameters; Pop caller-save registers

Stack: FP →, SP →, Reg 1 … Reg n, SP →, Param n … param1, SP →, Return address, SP →, Previous fp, SP →, FP →, Local 1, Local 2, … …, Local n, SP →

Call sequences – `Foo(42,21)`

caller

```
push %ecx
push $21
push $42
call _foo
```

Push caller-save registers; Push actual parameters (in reverse order)

call — push return address; Jump to call address

callee

```
push %ebp
mov %esp, %ebp
sub %8, %esp
push %ebx
```

Push current base-pointer; bp = sp; Push local variables (callee variables); Push callee-save registers

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

Pop callee-save registers; Pop callee activation record; Pop old base-pointer

return — pop return address; Jump to address

caller

```
add $8, %esp
pop %ecx
```

Pop parameters; Pop caller-save registers

"To Callee-save or to Caller-save?"

- Callee-saved registers need only be saved when callee modifies their value
- Some conventions exist (cdecl)
  - %eax, %ecx, %edx – caller save
  - %ebx, %esi, %edi – callee save
  - %esp – stack pointer
  - %ebp – frame pointer
  - Use %eax for return value

Accessing stack variables

- Use offset from EBP
- Stack grows downwards
- Above EBP = parameters
- Below EBP = locals

- Examples
  - %ebp + 4 = return address
  - %ebp + 8 = first parameter
  - %ebp – 4 = first local

Stack: Param n … FP+8 → param1; Return address; FP → Previous fp; FP-4 → Local 1, Local 2, … … … …, Local n-1; SP → Local n

`main` calling method `bar`

```
int bar(int x)
{
  int y;
  …
}
static void main(string[] args) {
  int z;
  Foo a = new Foo();
  z = a.bar(31);
  …
}
```

13

`main` calling method `bar`

```
int bar(Foo this, int x)
{
  int y;
  …
}
static void main(string[] args) {
  int z;
  Foo a = new Foo();
  z = a.bar(a,31);
  …
}
```

implicit parameter

implicit argument

	Stack	
	⋮	main's frame
EBP+12	31	main's frame
EBP+8	this ≈ a	main's frame
	Return address	main's frame
EBP	Previous fp	bar's frame
ESP, EBP-4	y	bar's frame

- Examples
  - %ebp + 4 = return address
  - %ebp + 8 = first parameter
    - Always **this** in virtual function calls
  - %ebp = old %ebp (pushed by callee)
  - %ebp – 4 = first local

14

x86 assembly

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

15

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, …

16

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**

17

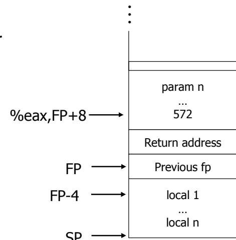
Memory and base displacement operands

- 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)**

18

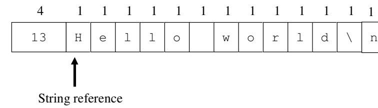
Reminder: accessing variables

- 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



19

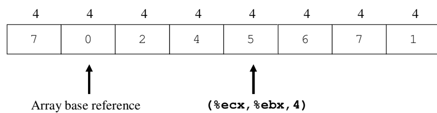
Representing strings and arrays



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

20

Base displacement addressing



`mov (%ecx,%ebx,4), %eax`

offset = base + (index * scale) + displacement

offset = %ecx + (3*4) + 0 = %ecx + 12

`%ecx = base`
`%ebx = 3`

21

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`**

22

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`

23

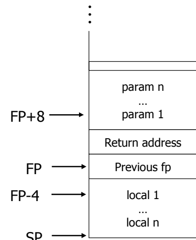
LIR to assembly

- 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

24

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



25

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

26

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;
}
```

PA5

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
```

PA4

① MethodLayout for **foo**

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

27

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

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,4(%ebx)
  mov -8(%ebp),%eax   # Library __printi(R0)
  push %eax
  call __printi
  add $4,%esp
_A_foo_epilogoue:
  mov %ebp,%esp        # epilogoue
  pop %ebp
  ret
```

28

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

29

Translating instructions 1

LIR Instruction	Translation

`MoveArray R1[R2],R3`	`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`

`MoveField x,R2.3`	`mov -12(%ebp),%ebx # -12(%ebp)=R2`
	`mov -8(%ebp),%eax  # -12(%ebp)=x`
	`mov %eax,12(%ebx)  # 12=3*4`

`MoveField _DV_A,R1.0`	`movl $_DV_A,(%ebx) # (%ebx)=R1.0` (**movl** means move 4 bytes)

`ArrayLength y,R1`	`mov -8(%ebp),%ebx  # -8(%ebp)=y`
	`mov -4(%ebx),%ebx  # load size`
	`mov %ebx,-12(%ebp) # -12(%ebp)=R1`

`Add R1,R2`	`mov -16(%ebp),%eax # -16(%ebp)=R1`
	`add -20(%ebp),%eax # -20(%ebp)=R2`
	`mov %eax,-20(%ebp) # store in R2`

30

Translating instructions 2

LIR Instruction	Translation

`Mul R1,R2`	`mov -8(%ebp),%eax # -8(%ebp)=R2`
	`imul -4(%ebp),%eax # -4(%ebp)=R1`
	`mov %eax,-8(%ebp)`

`Div R1,R2`
(`idiv` divides `EDX:EAX`
`stores quotient in EAX`
`stores remainder in EDX)`
	`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`

`Mod R1,R2`	`mov $0,%edx`
	`mov -8(%ebp),%eax # -8(%ebp)=R2`
	`mov -4(%ebp),%ebx # -4(%ebp)=R1`
	`idiv %ebx`
	`mov %edx,-8(%ebp)`

`Compare R1,x`	`mov -4(%ebp),%eax # -4(%ebp)=x` `cmp -8(%ebp),%eax # -8(%ebp)=R1`
`Return R1` (returned value stored in EAX register)	`mov -8(%ebp),%eax # -8(%ebp)=R1` `jmp _A_foo_epilogue`
`Return Rdummy`	# return; `jmp _A_foo_epilogue`

31

Calls/returns

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

32

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

33