Winter 2006-2007
Compiler Construction
T11 – Activation records + Introduction to x86 assembly

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

Today

IC Language (ic) → Lexical Analysis → Syntax Analysis (Parsing) → AST → Symbol Table etc. → Inter. Rep. (IR) → Code Generation → Executable code (exe)

- Today:
  - PA4 tips
  - Activation records
  - Memory layout
  - Introduction to x86 assembly
- Next time:
  - Code generation
    - Register allocation
    - Assembling
    - Linking
  - Activation records in depth
  - PA5

Tips for PA4

- Keep **list** of LIR instructions for each translated method
- Keep `ClassLayout` information for each class
  - Field offsets
  - Method offsets
  - Don't forget to take superclass fields and methods into account
- May be useful to keep reference in each LIR instruction to AST node from which it was generated
- Two AST passes:
  - Pass 1: collect and name strings literals (`Map<ASTStringLiteral,String>`), create `ClassLayout` for each class
  - Pass 2: use literals and field/method offsets to translate method bodies
- Finally: print string literals, dispatch tables, print translation list of each method body

Roadmap

- PA4 (HIR/LIR)
  - LIR data: string literals, dispatch vectors
  - Array access and field access instructions
  - Call/return primitives
  - LIR registers
  - LIR file (file.lir) generation
- PA5 (ASM)
  - Static information: string literals, dispatch tables
  - Handlers for runtime errors (here or in PA4)
  - Call sequences
  - Dynamic binding mechanism
  - Machine registers
  - Assembly file (file.s) generation

4

LIR vs. assembly

	LIR	Assembly
#Registers	Unlimited	Limited
Function calls	Implicit	Runtime stack
Instruction set	Abstract	Concrete
Types	Basic and user defined	Limited basic types

local variables, parameters, LIR registers

5

Function calls

- LIR – simply call/return
- 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)
- Assembly – pass parameters (+`this`), save registers, call, virtual method lookup, restore registers, pass return value, return

6

Activation records

- New environment = activation record (frame)
- Activation record = data of current function / method call
  - User data
    - Local variables
    - Parameters
    - Return values
    - Register contents
  - Administration data
    - Code addresses
    - Pointers to other activation records (not needed for IC)
- In IC – a stack is sufficient!

7

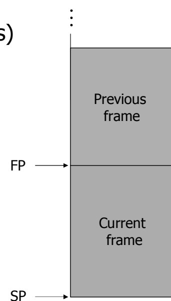
Runtime stack

- Stack of activation records
- Call = push new activation record
- Return = pop activation record
- Only one "active" activation record – top of stack
- Naturally handles recursion

8

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)



9

x86 runtime stack support

Register	Usage
ESP	Stack pointer
EBP	Base pointer

Pentium stack registers

Instruction	Usage
push, pusha,…	Push on runtime stack
pop, popa,…	Pop from runtime stack
call	Transfer control to called routine
ret	Transfer control back to caller

x86 stack and call/ret instructions

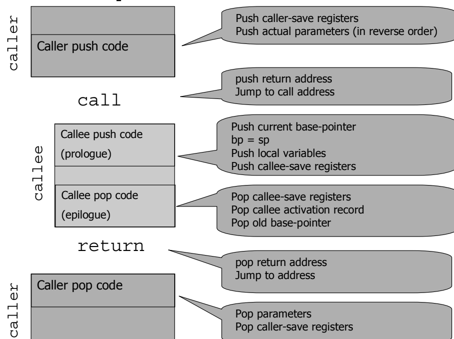
10

Call sequences

- The processor does not save the content of registers on procedure calls
- So who will?
  - Caller saves and restores registers
  - Callee saves and restores registers
  - But can also have both save/restore some registers
  - Some conventions exist (e.g., cdecl)

11

Call sequences



12

Call sequences – `foo(42,21)`

caller

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

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

push return address
Jump to call address

call

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

13

Caller-save/callee-save convention

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

14

Accessing stack variables

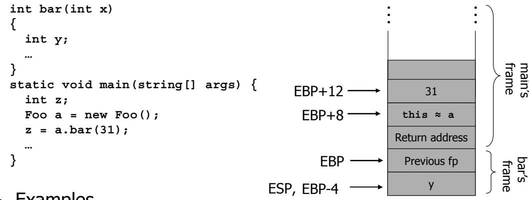
- Use offset from EBP
- Remember – stack grows downwards
- Above EBP = parameters
- Below EBP = locals
- Examples
  - %ebp + 4 = return address
  - %ebp + 8 = first parameter
  - %ebp – 4 = first local

Param n
...
param1
FP+8
Return address
FP
Previous fp
FP-4
Local 1
...
Local n
Param n
...
param1
SP
Return address

15

`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);
  …
}
```



- 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

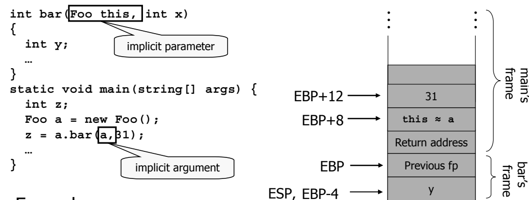
16

`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

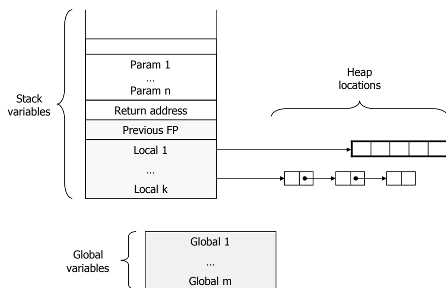
implicit argument



- 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

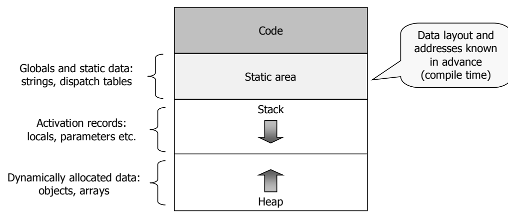
17

Conceptual memory layout



18

Memory segments layout



19

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

20

IA-32 registers

- 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
- Six 16-bit segment registers
- We ignore the rest

21

Register roles

- EAX
  - Accumulator for operands and result data
  - Required operand of MUL, IMUL, DIV and IDIV instructions
  - Contains the result of these operations
  - Used to store procedure return value
- EBX
  - Pointer to data, often used as array-base address
- ECX
  - Counter for string and loop operations
- EDX
  - Stores remainder of a DIV or IDIV instruction (EAX stores quotient)
- ESI, EDI
  - ESI – required source pointer for string instructions
  - EDI – required destination pointer for string instructions
- Destination registers for arithmetic operations
  - EAX, EBX, ECX, EDX
- EBP – stack frame (base) pointer
- ESP – stack pointer

22

Sample of x86 instructions

Instruction	Operands	Meaning
`add`	Reg,Reg imd,Reg mem,Reg	Addition
`sub`	Reg,Reg imd,Reg mem,Reg	
`inc`	Reg	Increment value in register
`dec`	Reg	Decrement value in register
`neg`	Reg	Negate
`mul`	Reg, %eax imd, %eax mem, %eax	Unsigned multiplication
`imul`	Reg Reg, %eax imd, %eax mem, %eax	Signed multiplication eax=eax*Reg
`idiv`	Reg	Signed division eax=quo, edx=rem eax / Reg

23

IA-32 addressing modes

- Machine-instructions take zero or more operands (usually allow at most one memory operand)
- Source operand
  - Immediate
  - Register
  - Memory location
  - (I/O port)
- Destination operand
  - Register
  - Memory location
  - (I/O port)

24

Immediate and register operands

- Immediate
  - Value specified in the instruction itself
  - GAS syntax – immediate values preceded by **`$`**
  - Example: **`add $4,%esp`**
- Register
  - Register name is used
  - GAS syntax – register names preceded with **`%`**
  - Example: **`mov %esp,%ebp`**

25

Memory and base displacement operands

- Memory operands
  - Obtain value at given address
  - GAS syntax – parentheses
  - Example: **`mov (%eax), %eax`**
- Base displacement
  - Obtain value at computed address
  - Address computed out of
    - base register, index register, scale factor, displacement
  - offset = base + (index * scale) + displacement
  - Syntax: disp(base,index,scale)
  - Example: **`mov $42,2(%eax)`**
  - Example: **`mov $42,1(%eax,%ecx,4)`**

26

Representing strings and arrays

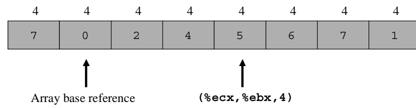
4	1	1	1	1	1	1	1	1	1	1	1	1	1
13	H	e	l	l	o		w	o	r	l	d	\	n

String reference

- Each string preceded by a word indicating the length of the string
- Array length stored in memory word preceding base address (i.e., the location at offset **`-4`**)
- Project-wise
  - String literals allocated statically, concatenation using **`__stringCat`**
  - Use **`__allocateArray`** to allocate arrays
    - NOTICE: accepts number of bytes not including the length word

27

Base displacement addressing




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

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

offset = base + (index * scale) + displacement

offset = base + (3*4) + 0 = base + 12

Instruction examples

- Translate `a=p+q` into
  - `mov 16(%ebp),%ecx` (load)
    `add 8(%ebp),%ecx` (arithmetic)
    `mov %ecx,-8(%ebp)` (store)
- Translate `a=12` into
  - `mov $12,-8(%ebp)`
- Accessing strings:
  - `str: .string "Hello world!"` access as constant:
    `push $str`
- 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 %ecx,0`
    `jnz L`

Runtime check example

- Suppose we have an array access statement `…=v[i]` or `v[i]=…`
  - In LIR: `MoveArray R1[R2],…`
    or `MoveArray …,R1[R2]`
  - We have to check that $0 \leq i < v.length$
  - `cmp $0,-12(%ebp)` (compare `i` to `0`)
    `jl ArrayBoundsError` (test lower bound)
    `mov -8(%ebp),%ecx` (load `v` into `%ecx`)
    `mov -4(%ecx),%ecx` (load array length into `%ecx`)
    `cmp -12(%ebp),%ecx` (compare `i` to array length)
    `jle ArrayBoundsError` (test upper bound)

Hello world example

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

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

31

Assembly file structure

header

```
.title    "hello.ic"

# global declarations
.global __ic_main
```

symbol exported to linker

statically-allocated data: string literals and dispatch tables

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

string length in bytes

Method bodies and error handlers

```
# text (code) section
.text

#----------------------------------------------------
        .align 4
__ic_main:
        push %ebp          # prologue
        mov %esp,%ebp

        push $str1         # print(...)
        call __print
        add $4, %esp

        mov $0,%eax        # return 0

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

comment

32

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

        mov $0,%eax        # return 0

        mov %ebp,%esp      # epilogue
        pop %ebp
        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

pop parameter

store return value of main in eax

epilogue – restore esp and ebp (pop)

33

Calls/returns

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

34

Virtual functions

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

35

See you next week

36