Calling Conventions

“For a good time...”

15-411, Fall 2011 edition
Josiah Boning
Synchronization

• Lab 2 due tonight
  • For real this time!
• Lab 3 and Homework 3 out
Synchronization: Lab 3

- Function calls
- Implementation familiar from 213?
  - Today should be a good refresher
- Still due on Tuesday
- I/O now!
Synchronization: Homework 3

- Function calls
- Design: exceptions
- Due Thursday
  - Hand in early on Tuesday to get feedback
Language Feature: Functions

- Name a programming language without functions!
  - Okay, Prolog...
- Some languages built around them
  - \((\lambda x.xx)(\lambda x.xx)\)
- Organization is good
- Recursion is powerful
Functions in C0

```c
int main() { ... }
bool foo(int bar, bool baz) { ... }
```

- Spec says:
  - $t \cdot (t_1 \ x_1, \ldots, t_n \ x_n) \{ \text{body} \}$

- Not first-class
  - So no concrete syntax for the types
Functions in C0

```c
bool foo(int bar, bool baz) { ... }
x = foo(2+3, y || z);
```

- But what does it mean?
  - `t1 = 2+3; t2 = y || z;`
  - initialize bar and baz with values of t1 and t2
  - run body of foo
  - x gets return value of foo
• Okay, so we have semantics
• Now how do we actually run these things?
Hardware – What We've Got

• State
  • Program counter
  • Registers
  • Memory

• Instructions
  • Straight-line execution (PC steps)
  • Unconditional and conditional jumps
Hardware – What We Want

• A sequence of instructions executed
  
  [instructions in main]
  
  [instructions in foo]
  
  [instructions in main]
Compilation Strategy 1

- Wherever foo appears, insert all of foo's instructions
  - Probably before register allocation
Compilation Strategy 1

- Wherever foo appears, insert all of foo's instructions
  - Probably before register allocation
- Bad
  - Much more work during register allocation
  - Huge program—lots of repeated code
  - Can't do recursion!
Hardware – What We Want

- A sequence of instructions executed
  - [instructions in main]
  - [instructions in foo]
  - [instructions in main]
Hardware – What We Want

• Insert jumps!

[instructions in main]
jmp foo

[instructions in foo]
jmp where_we_were

[instructions in main]
Hardware – What We Want

- Insert jumps!
  
  [instructions in main]
  jmp foo
  
  [instructions in foo]
  jmp where_we_were
  
  [instructions in main]

- How do we know where we were?
Compilation Strategy 2

- Self-modifying code
- Before jumping, *rewrite the last instruction in foo...*
- So that it jumps back to the next instruction!

```assembly
foo:
[instructions]

bar:
jmp some_location

main:
[instructions]
mov {jmp baz}, (bar)
```
Compilation Strategy 2

- Self-modifying code
- Before jumping, *rewrite the last instruction in foo*...
  - So that it jumps back to our next instruction!
- Yes, programs actually did this
  - Back in the good old days

```
foo:
[instructions]
bar:
jmp some_location
main:
[instructions]
mov (bar), {jmp baz}
```
Compilation Strategy 3

- Store next PC in a register
  - The “link register”
- Jump to the location in the register
- Hardware support: indirect jump

```
foo:
[instructions]
bar:
jmp %lr
main:
[instructions]
mov baz, %lr
```
Compilation Strategy 3, Improved

- Store next PC and jump all at once
- Hardware support: jump-and-link, indirect jump

foo:

[instructions]
jmp %lr

main:

[instructions]
jal foo

[instructions]
In the Real World: MIPS

- "Link Register": $31
- Instruction support:
  - jal – jump and link
  - jr – jump register

foo:
[j[instructions]]

jr $31

main:
[instructions]

jal foo
[instructions]
In the Real World: ARM

- “Link Register”: LR
- Instruction support:
  - bl – branch with link

foo:

mov pc, LR

main:

[instructions]

bl foo

[instructions]
In the Real World: x86???

• Possible!

• Instruction support:
  • No jump-and-link: need to set up a link register manually
    – lea makes it easy
  • jmp supports register argument

• Not standard.

foo:

[instructions]
jmp %ebx

main:

[instructions]
lea %ebx, bar
bl foo
Where do we stand?

- Can transfer control to and from blobs of code
- “Subroutine call”
- No arguments or return value
  - Can emulate using global state
    - Yuck
- Both blobs of code want to use registers
  - Who has to remember the original values?
Introducing: The Stack (x86)

- Area in memory
  - `%esp (stack pointer) tracks the front of the stack
  - push and pop instructions
  - Arguments go there
  - Local variables go there
  - Return addresses go there
- I hope this is all review
In the Real Real World – x86

- Store the return address on the stack
- The standard in x86
- Instructions:
  - call pushes next PC
  - ret pops into PC

foo:

[instructions]

ret

main:

[instructions]

call foo

[instructions]
Arguments (x86)

- Pushed onto the stack before a call
- Right-to-left!

Directly after a call:

arg3
arg2
arg1
return address
Stack Frames (x86)

- Set up a new “stack frame”
  - push %ebp
  - mov %ebp, %esp
  - sub %esp, size
- The stack is available to store local variables
- Clean up before ret
  - mov %esp, %ebp

During function execution:
- arg3
- arg2
- arg1
- return address
- old %ebp
- <local storage>
Return Values (x86)

- In %eax
Across Architectures

- As with return address, other ways to do it
- Arguments in registers
- More than one return value
## Across Architectures

<table>
<thead>
<tr>
<th></th>
<th>MIPS (32-bit)</th>
<th>ARM</th>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments</strong></td>
<td>$a0-$a4, then stack</td>
<td>r0-r3, then stack</td>
<td>on stack</td>
<td>%rdi, %rsi, %rdx, %r8, %r9, then stack</td>
</tr>
<tr>
<td>Return Address</td>
<td>$31</td>
<td>LR</td>
<td>on stack</td>
<td>on stack</td>
</tr>
<tr>
<td>Return Value</td>
<td>$v0, $v1</td>
<td>r0-r3</td>
<td>%eax</td>
<td>%eax</td>
</tr>
</tbody>
</table>
Across Architectures

<table>
<thead>
<tr>
<th></th>
<th>MIPS (32-bit)</th>
<th>ARM</th>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments</strong></td>
<td>$a0-$a4, then stack</td>
<td>r0-r3, then stack</td>
<td>on stack</td>
<td>%rdi, %rsi, %rdx, %r8, %r9, then stack</td>
</tr>
<tr>
<td><strong>Return Address</strong></td>
<td>$31</td>
<td>LR</td>
<td>on stack</td>
<td>on stack</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>$v0, $v1</td>
<td>r0-r3</td>
<td>%eax</td>
<td>%eax</td>
</tr>
</tbody>
</table>

- Secretly, it's worse than this
  - Floating point?
  - x86-64: Microsoft x64 or System V AMD64?
  - x86: stdcall, fastcall, safecall, thiscall
  - Your compiler must use the System V AMD64 convention
Where Are We?

- Have control flow transfer
- Have argument passing
- Have local variable storage
- Have return values
- Missing: register coordination
Register Saving

• Called function uses registers
• Caller's data was there
• Someone's got to save it somewhere
• *Caller save*: callee may overwrite values
  • Caller must store on stack before the call
• *Callee save*: must be unchanged across call
  • Callee's job to ensure this
<table>
<thead>
<tr>
<th></th>
<th>MIPS (32-bit)</th>
<th>ARM</th>
<th>x86</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Callee Save Registers</strong></td>
<td>$16-$23, $28, $29, $30, $31</td>
<td>r4-r8, r10, r11, SP</td>
<td>(others)</td>
<td>%rbx, %rbp, %r12, %r13, %r14, %r15</td>
</tr>
<tr>
<td><strong>Caller Save</strong></td>
<td>(others)</td>
<td>(others)</td>
<td>%eax, %ecx, %edx</td>
<td>%rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11</td>
</tr>
</tbody>
</table>
Registers & Function Calls

• x86-64: arguments in registers
  • Move temps into argument registers
  • Call function
  • Minimizes live ranges of pre-colored nodes in register allocation

• Caller-save registers
  • Add a rule: if \( l \) is a function call instruction, \( \forall r \in \) the caller-save registers, \( \text{def}(l, r) \)
  • If a temp is alive after the call, add edges between it and the caller-save registers
Handling Callee Save Registers

- One approach:
  - Save at the beginning of the function
  - Restore at the end
- Bad
  - Saves registers that aren't overwritten
Handling Callee Save Registers

• Better:
  • Add moves from callee save registers into temps at the beginning, and moves back at the end
  • Let register allocation deal with it

• See also Frank Pfenning's notes (on the course website)
So now...

- You're ready to write a compiler, right?

- Questions?