Assembly Code Generation

Java code → Lexical analysis → Parsing → Abstract Syntax Tree (AST) → Intermediate code generation → LLVM code → Target code generation → x86 code

Semantic analysis → Intermediate code generation
Lowering (LLVM) to Assembly

• Different instruction set
• Unbounded number of registers
  – Register allocation & spilling
• Function calls
  – Activation records
What’s in a Procedure

• A procedure needs access to
  – Its local variables
  – Its parameters
  – Return address

```c
int add(int x, int y)
{
    int inc = x;
    inc = inc + y;
    return inc;
}
```
The Deep Dive: Recursion

• Where are the arguments / local variables of each invocation stored?
• How do we know to access the correct ones?
• How do we know to which \texttt{fact} invocation to return? Or to \texttt{f}?
Activation Records / Stack Frames

• Data structure **per procedure invocation**
• Records all the necessary information
• Stored in the stack
• At **runtime**, an activation record is allocated for each invocation
  – Allocated when the procedure is called
  – Released when the procedure terminates
What if there’s no more available space in the stack?
Runtime Stack

- Stack grows downwards (towards smaller addresses)
- **BP** – base / frame pointer
  - base of current frame
- **SP** – stack pointer
  - top of current frame
  - last allocated value
How can we execute our code while...

- Finding arguments?
- Finding local variables?

Global variables access via their fixed address

Heap variables by following pointers from other variables

* 32 bit addresses
* Layout may change between architectures and operating systems
Activation Record’s Contents

How can we execute our code while...
• Finding arguments?
• Finding local variables?
How can we return to the caller’s context...
• Instruction pointer?
• Activation record?
• Registers?

* Layout may change between architectures and operating systems
Activation Record’s Contents

How can we execute our code while...

- Finding arguments?
- Finding local variables?

How can we return to the caller’s context...

- Instruction pointer?
- Activation record?
- Registers?

* Layout may change between architectures and operating systems
Application Binary Interface:
Things to Be Done (and By Whom) (and How)

Upon call:
• Storing arguments
• Storing return address
• Storing frame pointer
• Allocating stack space for registers
• Storing registers
• Allocating stack space for local variables

Upon return:
• Deallocating stack space for registers
• Deallocating stack space for local variables
• “Cleanup” arguments
• Storing return value
• Restoring base pointer
• Restoring instruction pointer
Example Application Binary Interface (ABI) in x86

Storing frame pointer and opening activation record

Allocate stack memory for vars

Restore register ebx

Restoring frame pointer

Some computation

Cleaning up args

Restoring instruction pointer (return to caller)

Store return address and jump to callee code

Saving register ecx

Saving register ebx

Restoring register ecx

Register

Register

Caller

Call

Callee

FP

SP

Enemies

Ret

Push %ecx

Push $21

Push $42

Call _foo

Adding 8, %esp

Pop %ecx

Save return address
**Caller- and Callee-Saved Registers**

**callees**

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

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

**call**

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

**saving register ecx**

**caller**

```
push %ecx
```

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

**restoring register ecx**

**Some computation**
Register Preservation

Who’s responsible to store and backup important registers?
• Caller knows which registers need to be preserved
• Callee knows which registers it overwrites

• **Callee-saved**: Caller guaranteed that they are not modified by the callee, or restored before callee returns
  – In x86: ebp, esp, ebx, edi, …
• **Caller-saved**: Can be modified by the callee, the caller needs to store them before the call if it needs them
  – In x86: eax, ecx, edx, …

• The compiler’s register allocation chooses between callee- and caller-saved
  – And generate code that respects the rules
Passing Arguments

In a register

```
return address
...
```

```
mov eax, 5
mov ebx, 37
call f
```

On the stack

```
5
push 5
37
push 37
call f
```

```
return address
...
```

```
int f(int a, int b)
{
    ...
}
```

```
void g()
{
    f(5, 37);
}
```
Passing Arguments

<table>
<thead>
<tr>
<th>In a register</th>
<th>On the stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Limited number of registers</td>
<td>• Slower access</td>
</tr>
<tr>
<td>• Register preservation</td>
<td>• Need to cleanup</td>
</tr>
</tbody>
</table>

- Most x86 (cdecl, stdcall): arguments on the stack
- x86_64: first arguments in designated (caller-saved) registers, rest on the stack
Argument Cleanup

By caller

37
ret
add $8, %esp
5
return address
...

- e.g. cdecl, ...

vararg

printf("%d",1);
printf("%d,%d",1,2);

By Callee

37
ret 8
5
return address
...

- e.g. stdcall, ...

Smaller binaries
Order of Arguments on the Stack

Left to right

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

push 5
push 37
call f

Right to left

<table>
<thead>
<tr>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

push 37
push 5
call f

○ e.g. cdecl, stdcall, ...
## Return Value

<table>
<thead>
<tr>
<th>In a register</th>
<th>On the stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Limited number of registers</td>
<td>• Slower access</td>
</tr>
<tr>
<td>• Register preservation</td>
<td>• Need to cleanup</td>
</tr>
</tbody>
</table>

- What if we want to return something that doesn’t fit in a register?
Return Address

- In a designated register or on the stack?
- Store the current instruction or the next instruction?

- In practice, this is decided by the architecture’s “call” operation
Which is Best?

- No “correct” answer
- Depends on
  - Processor capabilities,
  - Applications’ characteristics
  - Conventions
- Caller & callee must agree on the calling convention!
  - Interoperability between compilers
  - Or with explicit directives:

```c
int __cdecl system(const char *);
```
Summary

- Runtime stack
- Activation records
- Frame pointer, stack pointer
- Calling conventions