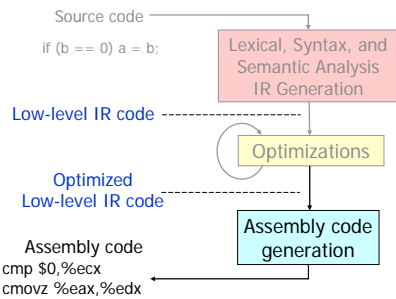


## CS412/CS413

### Introduction to Compilers Tim Teitelbaum

#### Lecture 20: Stack Frames 9 March 07

## Where We Are



## Assembly vs. Low IR

- **Assembly code:**
  - Finite set of registers
  - Variables = memory locations (no names)
  - Variables accessed differently: global, local, heap, args, etc.
  - Uses a run-time stack (with special instructions)
  - Calling sequences: special sequences of instructions for function calls and returns
  - Instruction set of target machine
- **Low IR code:**
  - Variables (and temporaries)
  - No run-time stack
  - No calling sequences
  - Some abstract set of instructions

## Low IR to Assembly Translation

- **Calling sequences:**
  - Translate function calls and returns into appropriate sequences that: pass parameters, save registers, and give back return values
  - Consists of push/pop operations on the **run-time stack**
- **Variables:**
  - Translate accesses to specific kinds of variables (globals, locals, arguments, etc)
  - **Register Allocation:** map the variables to registers
- **Instruction set:**
  - Account for differences in the instruction set
  - **Instruction selection:** map sets of low level IR instructions to instructions in the target machine

## x86 Quick Overview

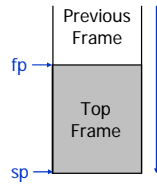
- **Few registers:**
  - General purpose 32bit: `eax, ebx, ecx, edx, esi, edi`
    - Also 16-bit: `ax, bx, etc.`, and 8-bit: `al, ah, bl, bh, etc.`
  - Stack registers: `esp, ebp`
- **Many instructions:**
  - Arithmetic: `add, sub, inc, mod, idiv, imul, etc.`
  - Logic: `and, or, not, xor`
  - Comparison: `cmp, test`
  - Control flow: `jmp, jcc, jecz`
  - Function calls: `call, ret`
  - Data movement: `mov` (many variants)
  - Stack manipulations: `push, pop`
  - Other: `lea`

## Run-Time Stack

- **A frame** (or **activation record**) for each function execution
  - Represents execution environment of the function
  - Includes: local variables, parameters, return value, etc.
  - Different frames for recursive function invocations
- **Run-time stack of frames:**
  - Push frame of `f` on stack when program calls `f`
  - Pop stack frame when `f` returns
  - Top frame = frame of currently executed function
- This mechanism is necessary to support **recursion**
  - Different activations of the same recursive function have different stack frames

## Stack Pointers

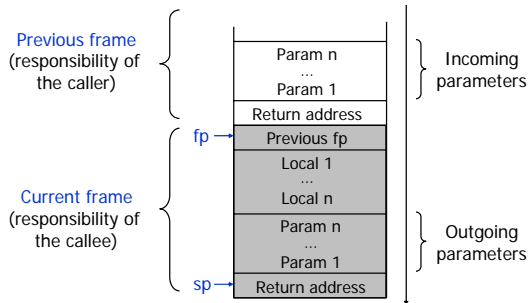
- Usually run-time stack grows downwards
  - Address of top of stack decreases
- Values on current frame (i.e., frame on top of stack) accessed using two pointers:
  - Stack pointer (sp): points to frame top
  - Frame pointer(fp): points to frame base
  - Variable access: use offset from fp (sp)
- When do we need two pointers?
  - If stack frame size not known at compile time
  - Example: `alloca` (dynamic allocation on stack)



## Hardware Support

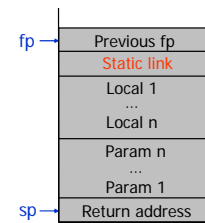
- Hardware provides:
  - Stack registers
  - Stack instructions
- X86 Registers and instructions for stack manipulation:
  - Stack pointer register: `esp`
  - Frame pointer register: `ebp`
  - Push instructions: `push`, `pusha`, etc.
  - Pop instructions: `pop`, `popa`, etc.
  - Call instruction: `call`
  - Return instruction: `ret`

## Anatomy of a Stack Frame



## Static Links

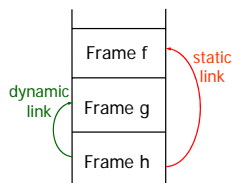
- Problem for languages with nested functions (Pascal):
  - How do we access local variables from other frames?
- Need a **static link**: a pointer to the frame of enclosing function
- Previous fp = **dynamic link**, i.e. pointer to the previous frame in the current execution



## Example Nested Procedures

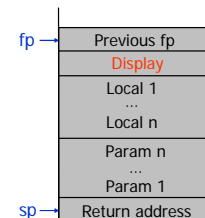
```

procedure f(i : integer)
  var a : integer;
  procedure h(j : integer)
    begin a = j end
  procedure g(k : integer)
    begin h(k*k) end
  begin g(i+2) end
    
```



## Display

- Unacceptable to have to chase down static chains to find frame containing non-local variable.
- A **display** is a linearization of the static chain copied into the local frame (or maintained globally) as an array.
- The pointer to the frame containing non-local variables at lexical level  $i$  is `display[i]`.



## Saving Registers

- **Problem:** execution of invoked function may overwrite useful values in registers
- Generated code must:
  - Save registers when function is invoked
  - Restore registers when function returns
- **Possibilities:**
  - Callee saves and restores registers
  - Caller saves and restores registers
  - ... or both

CS 412/413 Spring 2007

Introduction to Compilers

13

## Calling Sequences

- How to generate the code that builds the frames?
- Generate code that pushes values on stack:
  1. Before call instructions (caller responsibilities)
  2. At function entry (callee responsibilities)
- Generate code that pops values from stack:
  3. After call instructions (caller responsibilities)
  4. At return instructions (callee responsibilities)
- **Calling sequences** = sequences of instructions performed in each of the above 4 cases

CS 412/413 Spring 2007

Introduction to Compilers

14

## Push Values on Stack

- **Code before call instruction:**
  - Push caller-saved registers
  - Push each actual parameter (in reverse order)
  - Push static link (or display) (if necessary)
  - Push return address (current program counter) and jump to caller code
- **Prologue = code at function entry**
  - Push dynamic link (i.e., current fp)
  - Old stack pointer becomes new frame pointer
  - Push local variables
  - Push callee-saved registers

CS 412/413 Spring 2007

Introduction to Compilers

15

## Pop Values from Stack

- **Epilogue = code at return instruction**
  - Pop (restore) callee-saved registers
  - Restore old stack pointer (pop callee frame!)
  - Pop old frame pointer
  - Pop return address and jump to that address
- **Code after call**
  - Pop (restore) caller-saved registers
  - Pop parameters from the stack
  - Pop static link (or display) (if necessary)
  - Use return value

CS 412/413 Spring 2007

Introduction to Compilers

16

## Example: Pentium

- Consider call foo(3, 5), %ecx caller-saved, %ebx callee-saved, no static links, result passed back in %eax
- Code before call instruction:

```
push %ecx    // push caller saved registers
push $5     // push second parameter
push $3     // push first parameter
call _foo   // push return address and jump to callee
```
- Prologue:

```
push %ebp   // push old fp
mov %esp, %ebp // compute new fp
sub $12, %esp // push 3 integer local variables
push %ebx  // push callee saved registers
```

CS 412/413 Spring 2007

Introduction to Compilers

17

## Example: Pentium

- Epilogue:

```
pop %ebx    // restore callee-saved registers
mov %ebp, %esp // pop callee frame, including locals
pop %ebp    // restore old fp
ret        // pop return address and jump
```
- Code after call instruction:

```
add $8, %esp // pop parameters
pop %ecx    // restore caller-saved registers
```

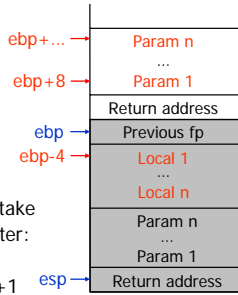
CS 412/413 Spring 2007

Introduction to Compilers

18

## Accessing Stack Variables

- To access stack variables:  
use offsets from fp
- Example:  
 $8(\%ebp)$  = parameter 1  
 $12(\%ebp)$  = parameter 2  
 $-4(\%ebp)$  = local 1
- Translate low-level code to take into account the frame pointer:  
 $a = p+1$   
 $\Rightarrow -4(\%ebp) = 16(\%ebp)+1$



CS 412/413 Spring 2007

Introduction to Compilers

19

## Accessing Other Variables

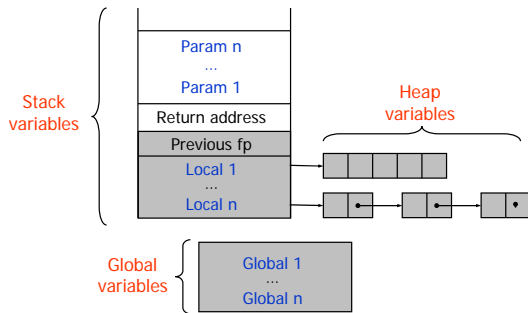
- Global variables
  - Are statically allocated
  - Their addresses can be statically computed
  - Don't need to translate low IR
- Heap variables
  - Are unnamed locations
  - Can be accessed only by dereferencing variables that hold their addresses
  - Therefore, they don't explicitly occur in low-level code

CS 412/413 Spring 2007

Introduction to Compilers

20

## Big Picture: Memory Layout



CS 412/413 Spring 2007

Introduction to Compilers

21