CSE P 501 – Compilers

x86 Lite for Compiler Writers

Hal Perkins

Autumn 2011

Function Call and Return

- The x86 instruction set itself only provides for transfer of control (jump) and return
- Stack is used to capture return address and recover it
- Everything else – parameter passing, stack frame organization, register usage – is a matter of convention and not defined by the hardware

call and ret Instructions

call label

- Push address of next instruction and jump
- esp <- esp – 4; memory[esp] <- eip
  eip <- address of label

ret

- Pop address from top of stack and jump
- eip <- memory[esp]; esp <- esp + 4
- **WARNING!** The word on the top of the stack had better be an address, not some leftover data

enter and leave

- Complex instructions for languages with nested procedures
  - enter can be slow on current CPUs – best avoided
    - i.e., don't use it in your project
  - leave is equivalent to

```
mov esp,ebp
pop ebp
```

and is generated by many compilers. Fits in 1 byte, saves space. Not clear if it's any faster.

Win 32 C Function Call Conventions

- Wintel code obeys the following conventions for C programs
  - Note: calling conventions normally designed very early in the instruction set/ basic software design. Hard (e.g., basically impossible) to change later.
- C++ augments these conventions to include the "this" pointer
- We'll use these conventions in our code

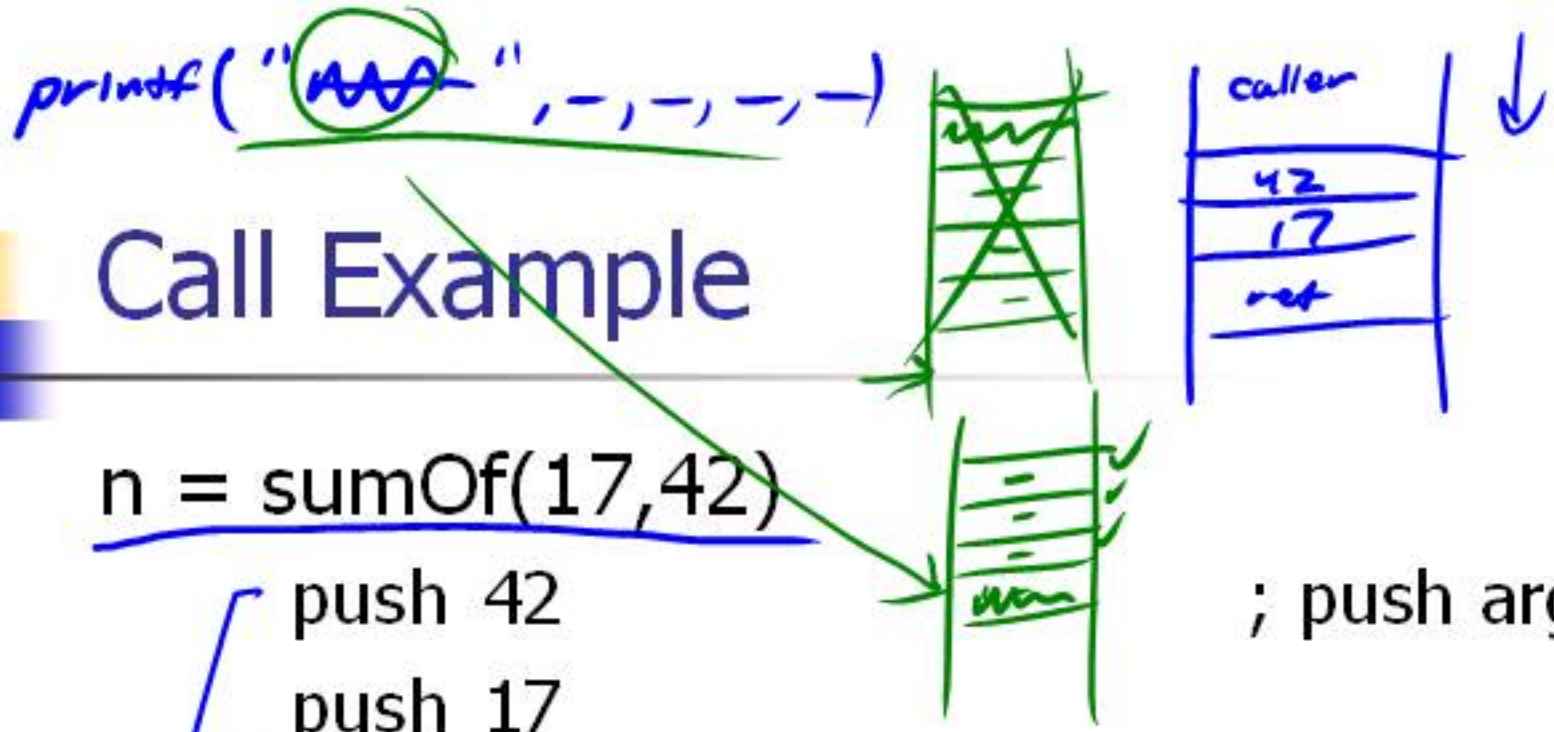
Win32 C Register Conventions

- These registers must be restored to their original values before a function returns, if they are altered during execution
  → esp, ebp, ebx, esi, edi
  - Traditional: push/pop from stack to save/restore
- A function may use the other registers (eax, ecx, edx) without having to save/restore
- A 32-bit function result is expected to be in eax when the function returns

Call Site

- Caller is responsible for
  - Pushing arguments on the stack from right to left (allows implementation of varargs)
  - Execute call instruction
  - Pop arguments from stack after return
    - For us, this means add 4*(# arguments) to esp after the return, since everything is either a 32-bit variable (int, bool), or a reference (pointer)

Call Example




```
n = sumOf(17,42)
    push 42                 ; push args
    push 17
    call   sumOf            ; jump &
                            ;   push addr
    add    esp,8            ; pop args
    mov    [ebp+offset_n],eax   ; store result
```

n =

Callee

- Called function must do the following
  - Save registers if necessary
  - Allocate stack frame for local variables
  - Execute function body
  - Ensure result of non-void function is in eax
  - Restore any required registers if necessary
  - Pop the stack frame
  - Return to caller

prologue

*

epilogue

Win32 Function Prologue

- The code that needs to be executed before the statements in the body of the function are executed is referred to as the *prologue*
- For a Win32 function *f*, it looks like this:


```
f:  push   ebp              ; save old frame pointer
    mov    ebp,esp          ; new frame ptr is top of
                            ;  stack after arguments and
                            ;  return address are pushed
    sub    esp,"# bytes needed"
                            ; allocate stack frame (size
                            ; must be multiple of 4)
```

Win32 Function Epilogue

- The *epilogue* is the code that is executed for a return statement (or if execution "falls off" the bottom of a void function)
- For a Win32 function, it looks like this:

```
mov     eax,"function result"
                        ; put result in eax if not already
                        ;     there (if non-void function)
mov     esp,ebp         ; restore esp to old value
                        ;     before stack frame allocated
pop     ebp             ; restore ebp to caller's value
ret                     ; return to caller
```

Example Function

- Source code

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

Stack Frame for sumOf

```
int sumOf(int x, int y) {
   int a, int b;
   a = x;
   b = a + y;
   return b;
}
```

→ n = sumOf(17,42) call sumOf
add esp, 8

Assembly Language Version

```
;; int sumOf(int x, int y) {
;; int a, int b;
sumOf:
    push  ebp      ; prologue
    mov   ebp,esp
    sub   esp, 8

;; a = x;
    mov  eax,[ebp+8]
    mov  [ebp-4],eax

;; b = a + y;
    mov  eax,[ebp-4]
    add  eax,[ebp+12]
    mov  [ebp-8],eax

;; return b;
    mov  eax,[ebp-8]
    mov  esp,ebp
    pop  ebp
    ret
;; }
```

leave