Lecture 34: Registers, Functions, Parameters

A Slight Change of Assumptions: Registers

- Up until now, I've been treating integers and pointers as 4-byte quantities, as they are for the ia32 architecture.
- Since our instructional machines and many (most?) laptops and home machines are 64-bit machines (most importantly, the x86-64 architecture, aka x64, AMD64, x86_64, and Intel 64), we might as well go for authenticity and change this assumption.
- This architecture extends all the 32-bit registers to 64 bits, changing names like `eax` (32 bits) to `rax` (64 bits). A reference to `eax` is now to the lower 32 bits of `rax` (likewise for other registers).
- The x86-64 also adds new registers: `r8`, `r9`, `...`, `r15`. You get the lower 32 bits of these with `r8d`, etc.

A Slight Change of Assumptions: Parameters and Frames

- Secondly, the x86-64 calling conventions use registers to pass the first six integer and pointer parameters. Only the rest get pushed on the stack. This is a matter of speed.
- Specifically, the first 6 parameters go into registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9` in that order.
- For this lecture, however, we will continue to keep all parameters on the stack. Just keep in mind that when you actually look at code generated by compilers for the x86-64, you'll find these first parameters elsewhere.
- Finally, the conventions require that the stack pointer always be evenly divisible by 16 immediately before a call, meaning that compilers should arrange for stack-frame sizes to be multiples of 16. This has to do with cache considerations and certain instructions that require that memory alignment.

Three-Address Code to x86-64

- The problem is that in reality, the x86-64 architecture has fewer physical registers than our three-address code generator from last time typically allocates as virtual registers.
- *Register allocation* is the general term for assigning virtual registers to real registers or memory locations.
- When we run out of real registers, we *spill* values into memory locations reserved for them.
- We keep a register or two around as *compiler temporaries* for cases where the instruction set doesn't let us just combine operands directly.

A Simple Strategy: Local Register Allocation

- It's convenient to handle register allocation within *basic blocks*—sequences of code with one entry point at the top and any branches at the very end.
- At the end of each such block, spill any registers needed.
- To do this efficiently, need to know when a register is *dead*—that is, when its value is no longer needed.
- We'll talk about how to compute that in a later lecture. Let's assume we know it for now.
- Let's also assume that each virtual register representing a local variable or intermediate result has a memory location reserved for it suitable for spilling.

Simple Algorithm for Local Register Allocation

- Initially, the set `availReg` contains all physical registers available for assignment. (There may also be some "very temporary" registers around to help with certain instructions).
- We execute the following for each three-address instruction in a basic block (in turn).

```
# Allocate registers to an instruction x := y op z or x := op y
# [Adopted from Aho, Sethi, Ullman]
regAlloc(x, y, z):
    if x has an assigned register already or dies here:
        return
    if y is a virtual register and dies here:
        reassign y's physical register to x
    elif availReg is not empty:
        remove a register from availReg and assign to x
    elif op requires a register:
        spill another virtual register (which could be y or z),
            and reassign its physical register to x
    else:
        # Memory-to-memory operations.
        just leave x in memory
```

Function Prologue and Epilogue for the x86-64 (Review)

- Consider a function that needs K bytes of local variables and other compiler temporary storage for expression evaluation.
- We'll consider the case where we keep a frame pointer.
- Overall, the code for a function, F, looks like this:

```
F:
        pushq %rbp              # Save dynamic link (caller's frame pointer)
        movq  %rsp,%rbp         # Set new frame pointer
        subq  K,%rsp            # Reserve space for locals
        code for body of function, leaving value in %rax
        leave                   # Sets %rbp to 0(%rbp), popping old frame pointer
        ret                     # Pop return address and return
```

Code Generation for Local Variables

- Local variables are stored on the stack (thus not at fixed location).
- One possibility: access relative to the stack pointer, but
  - Sometimes convenient for stack pointer to change during execution of of function, sometimes by unknown amounts.
  - Debuggers, unwinders, and stack tracers would like simple way to compute stack-frame boundaries.
- Solution: use frame pointer, which is constant over execution of function.
- For simple language, use fact that parameter i is pushed on the stack, at location frame pointer $+ K_1(i + K_2)$. If parameters are 64-bit integers (or pointers) on the x86-64, $K_1 = 8$ and $K_2 = 2$ [why?].
- Local variables other than parameters are at negative offsets from the frame pointer on the x86-64.

Accessing Non-Local Variables

- In program on left, how does f3 access x1?
- Let's suppose that functions pass static links as the first parameter of their callees (that is: `16(ebp)`).
- The static link passed to f3 will be f2's frame pointer.

```
def f1 (x1):
   def f2 (x2):
       def f3 (x3):
           ... x1 ...
       ...
       f3 (12)
   ...
   f2 (9)
```

```
# To access x1 in f3:
  movq 16(%rbp),%rbx  # Fetch FP for f2
  movq 16(%rbx),%rbx  # Fetch FP for f1
  movq 24(%rbx),%rax  # Fetch x1 (static link is 1st)

# When f2 calls f3:
  pushq $12
  pushq %rbp          # Pass f2's frame to f3
  call f3
```

- We'll say a function is at nesting level 0 if it is at the outer level, and at level $k+1$ if it is most immediately enclosed inside a level-k function. Likewise, the variables, parameters, and code in a level-k function are themselves at level $k+1$ (enclosed in a level-k function).
- In general, for code at nesting level n to access a variable at nesting level $m \leq n$, perform $n-m$ loads of static links.

Accessing Non-Local Variables (II)

- Gcc on the instructional machines passes the static link in register `r10` and the callee saves it in a fixed local variable (`-8(%ebp)`), making it easy to ignore if not needed. We'll use that in what follows.

```
def f1 (x1):
   def f2 (x2):
       def f3 (x3):
          ... x1 ...
       ...
       f3 (12)
   ...
   f2 (9)
```

```
# Immediately after prologue:
 pushq %r10          # Save static link at -8 off %rbp.

# To access x1 in f3:
 movq -8(%rbp),%rbx  # Fetch FP for f2
 movq -8(%rbx),%rbx  # Fetch FP for f1
 movq 16(%rbx),%rax  # Fetch x1

# When f2 calls f3:
 compute parameters
 movq %rbp, %r10     # Pass f2's frame to f3
 call f3
```

Calling Function-Valued Variables and Parameters

- As we've seen, a function value consists of a code address and a static link (let's assume code address comes first).
- So, in project 3, when we need the value of a function itself:

```
def caller(f):
    f(42)
```

  we create an object containing the code pointer and static link for `f`, and pass a pointer to this object.

- Then the call `f(42)` might get translated to

```
pushq $42
movq  16(%rbp), %rax   # Get parameter f
movq  8(%rax), %r10    # Fetch static link from f
movq  0(%rax), %rax    # Get code address for f
call  *%rax            # GNU assembler for call to address in eax
```

Static Links for Calling Known Functions

- For a call $F(\ldots)$ to a fixed, known function F, we could use the same strategy as for function-valued variables:
  - Create a closure for F containing address of F's code and value of its static link.
  - Call F using the same code sequence as on previous slide.
- But can do better. Functions and their nesting levels are known.
- In code that is at nesting level n, to call a function at known nesting level $m \leq n$, get correct static link in register r10 with:
  - 'movq %rbp,%r10' (start at my frame pointer)
  - Do 'movq -8(%r10),%r10' $n - m + 1$ times.

  (assuming again we save static links at -8 off our frame pointer).
- When calling outer-level functions, it doesn't matter what you use as the static link.

Passing Static Links to Known Functions: Example

```
def f1 (x1):
   def f2 (x2):
       def f3 (x3):
          ... f2 (9) ...
       ...
       f3 (12)
       f2 (10) # (recursively)
   ...
```

```
# To call f2(9) (in f3):
  pushq $9
  movq  -8(%rbp),%r10  # Fetch FP for f2
  movq  -8(%r10),%r10  # Fetch FP for f1, and pass it
  call  f2
  addq  $8,%rsp    # Pop parameter from stack

# To call f3(12) (in f2):
  push1 $12
  movq %rbp, %r10     # f2's FP is static link
  call  f3
  addq  $8,%rbp

# To call f2(10) (in f2):
  pushq $10
  movq  -8(%rbp),%r10  # Pass down same static link
  call  f2
  addq  $8,%rbp
```

A Note on Pushing

- Don't really need to push and pop the stack as I've been doing. Not needed when using registers to pass parameters.
- Or, when allocating local variables, etc., on the stack, leave sufficient extra space on top of the stack to hold any parameter list in the function. Eg., to translate

```
def f(x):
   g(g(x+2))
```

- We could either get the code on the left (pushing and popping) or that on the right (ignoring static links):

```
f:   movq  16(%rbp),%rax           f:   subq  $16,%rsp
     addq  $2,%rax                      movq  16(%rbp),%rax
     pushq %rax                         addq  $2,%rax
     call  g                            movq  %rax,0(%rsp)
     addq  $8,%rsp                      call  g
     pushq %rax                         movq  %rax,0(%rsp)
     call  g                            call  g
     addq  $8,%rsp
```

...and you can continue to use the depressed stack pointer for arguments on the right.

Real Life Is More Complicated

Excerpt from a recent version of GCC:

```
long
f(long y, long x, long* p,
  long z, long a, long b, long c) {
    printf("%ld %ld %ld %ld\n", y, x, *p, z);
    printf("%ld %ld %ld\n", a, b, c);
    return *p;
}
```

```
pushq   %rbp
movq    %rsp, %rbp
subq    $48, %rsp
movq    %rdi, -8(%rbp)
movq    %rsi, -16(%rbp)
...
movq    %r9, -48(%rbp)
movq    -24(%rbp), %rax
movq    (%rax), %rcx
...
movq    -8(%rbp), %rax
movq    %rsi, %r8
movq    %rax, %rsi
movl    $.LC0, %edi
movl    $0, %eax
call    printf
...
movq    -24(%rbp), %rax
movq    (%rax), %rax
leave
ret
```

Parameter Passing Semantics: Value vs. Reference

- So far, our examples have dealt only with *value parameters*, which are the only kind found in C, Java, and Python

  Ignorant comments from numerous textbook authors, bloggers, and slovenly hackers notwithstanding [End Rant].

- Pushing a parameter's value on the stack creates a copy that essentially acts as a local variable of the called function.
- C++ (and Pascal) have *reference parameters*, where assignments to the formal are assignments to the actual.
- Implementation of reference parameters is simple:
  - Push the address of the argument, not its value, and
  - To fetch from or store to the parameter, do an extra indirection.

Copy-in, Copy-out Parameters

- Some languages, such as Fortran and Ada, have a variation on this: *copy-in, copy-out*. Like call by value, but the final value of the parameter is copied back to the original location of the actual parameter after function returns.
  - "Original location" because of cases like `f(A[k])`, where `k` might change during execution of `f`. In that case, we want the final value of the parameter copied back to `A[k0]`, where `k0` is the original value of `k` before the call.
  - Question: can you give an example where call by reference and copy-in, copy-out give different results?

Parameter Passing Semantics: Call by Name

- Algol 60's definition says that the effect of a call $P(E)$ is as if the body of P were substituted for the call (dynamically, so that recursion works) and E were substituted for the corresponding formal parameter in the body (changing names to avoid clashes).
- It's a simple description that, for simple cases, is just like call by reference:

```
procedure F(x)                      F(aVar);
   integer x;                   becomes
begin                               aVar := 42;
   x := 42;
end F;
```

- But the (unintended?) consequences were "interesting".

Call By Name: Jensen's Device

- Consider:

```
procedure DoIt (i, L, U, x, x0, E)
    integer i, L, U; real x, x0, E;
begin
    x := x0;
    for i := L step 1 until U do
        x := E;
end DoIt;
```

- To set y to the sum of the values in array A[1:N],

```
integer k;
DoIt(k, 1, N, y, 0.0, y+A[k]);
```

- To set z to the Nth harmonic number:

```
DoIt(k, 1, N, z, 0.0, z+1.0/k);
```

- Now how are we going to make this work?

Call By Name: Implementation

- Basic idea: Convert call-by-name parameters into parameterless functions (traditionally called *thunks*.)
- To allow assignment, these functions can return the addresses of their results.
- So the call

  ```
  DoIt(k, 1, N, y, 0.0, y+A[k]);
  ```

  becomes something like (please pardon highly illegal notation):

  ```
  integer t1;  real t2, t3, t4;
  t2 := 1.0; t3 := 0.0;
  DoIt(lambda: &k, lambda: &t2, lambda: &N, lambda: &y,
       lambda: &t3, lambda: (t4 := y+A[k], &t4));
  ```

- Later languages have abandoned this particular parameter-passing mode.