15-411: Compiler Design — Fall 2018

Recitation 5: Calling Conventions Solutions — 5 October

The L3 language adds support for function calls, type definitions, and header files with C interoperability. In this recitation, we'll discuss some of the implications of adding these features and how your compiler should deal with them.

Caller- and Callee-Saved Registers

In Lab 3, your compiler's code-generation and register allocation phases will need to distinguish between *callee-saved* and *caller-saved* registers:

- The values stored in **callee-saved registers** must be preserved across function calls. This means that your function must save and restore any callee-saved registers that it modifies.
- The values stored in **caller-saved registers** may be modified by any function call, so your compiler cannot assume that they will retain their values after calling a function. If you need those values to be preserved, you must save and restore them before and after the function call.

In your register allocation, you will probably want to consider the differences between these two types of registers in order to reduce the number of save and restore instructions you must add. In lecture on Tuesday, you'll see a relatively simple way of dealing with most of these issues.

Function	64-bit	32-bit	16-bit	8-bit
Return Value	**%rax**	**%eax**	%ax	%al
Callee saved	**%rbx**	**%ebx**	%bx	%bl
4th Argument	**%rcx**	**%ecx**	%cx	%cl
3rd Argument	**%rdx**	**%edx**	%dx	%dl
2nd Argument	**%rsi**	**%esi**	%si	%sil
1st Argument	**%rdi**	**%edi**	%di	%dil
Callee saved	**%rbp**	**%ebp**	%bp	%bpl
Stack Pointer	**%rsp**	**%esp**	%sp	%spl
5th Argument	**%r8**	**%r8d**	%r8w	%r8b
6th Argument	**%r9**	**%r9d**	%r9w	%r9b
Caller saved	**%r10**	**%r10d**	%r10w	%r10b
Caller saved	**%r11**	**%r11d**	%r11w	%r11b
Callee saved	**%r12**	**%r12d**	%r12w	%r12b
Callee saved	**%r13**	**%r13d**	%r13w	%r13b
Callee saved	**%r14**	**%r14d**	%r14w	%r14b
Callee saved	**%r15**	**%r15d**	%r15w	%r15b

Checkpoint 0

One team's compiler made some bad decisions about where to store values, and also forgot to save and restore registers! Add the necessary save and restore instructions to the following assembly function.

```
_c0_foo:
  mov $15, %ebx
  mov $411, %r12d
  mul $100, %ebx
  add %r12d, %ebx
  mov %ebx, %edi
  mov $2, %esi
  call _c0_bar
  mov %edi, %eax
  div %esi, %eax
  ret
```

Solution: NOTE: The solution below uses `push` and `pop` instructions, but it might be easier to implement the save/restore operations by moving `%rsp` in practice.

```
_c0_foo:
```

```
  push %rbx
  push %r12
  mov $15, %ebx
  mov $411, %r12d
  mul $100, %ebx
  add %r12d, %ebx
  mov %ebx, %edi
  mov $2, %esi
  push %rdi
  push %rsi
  call _c0_bar
  pop %rsi
  pop %rdi
  mov %edi, %eax
  div %esi, %eax
  pop %r12
  pop %rbx
  ret
```

Checkpoint 1

If different choices were made during register allocation, some of the save and restore operations that you just added would not have been necessary. Modify the above function so that it has the same behavior, but uses less save and restore operations.

Solution:

```
_c0_foo:
  push %rbx
  push %r12
  mov $15, %ebx
  mov $411, %esi  # we don't need 411 after the call, so store it in a caller-saved reg
  mul $100, %ebx
  add %esi, %ebx
  mov %ebx, %edi
  mov $2, %r12d   # we need 2 after the call, so store it in a callee-saved reg
  mov %r12d, %esi
  call _c0_bar
  mov %ebx, %eax
  div %r12d, %eax
  pop %r12
  pop %rbx
  ret
```

Tracing Function Calls in x86-64

In Lab 3, your compiler must conform to the standard C calling conventions for x86-64. As a reminder, this means that:

- The first six arguments to a function should be stored in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` (respectively).
- The remaining arguments should be placed on the stack. The seventh argument should be stored at the address `%rsp`, the eighth at `%rsp + 8`, etc.
- The return value of a function should be stored in `%rax`.
- The use of `%rbp` as a base pointer is not required (but you may find that using it simplifies your compiler's logic significantly). LLVM uses the base pointer, but GCC does not.

Another interesting observation: unlike in C, every function in C0 (and thus in L3) has a fixed stack size that can be computed at compile time. This observation allows you to make your compiler's stack-handling much simpler than if you were unable to determine the stack size beforehand.

Checkpoint 2

Draw a stack diagram for the following L3 program at the point when execution reaches line 4. Assume that `%rbp` is being used as a base pointer.

```
1 int f(int we, int dont, int care, int about, int these, int args, int a, int b) {
2   // assume that x is spilled on the stack
3   int x = a + b;
4   return 2 * x;
5 }
6
7 int main() {
8   return f(0,0,0,0,0,0,3,5);
9 }
```

Solution:

Value	Pointers
Return address in `_main()`	
Previous `%rbp`	
b; Arg. 8 of `f()`	
a; Arg. 7 of `f()`	
Return address in `_c0_main()`	
Previous `%rbp`	← `%rbp`
x	← `%rsp`

Checkpoint 3

Using your stack diagram, convert the program to x86-64 assembly following the standard calling conventions. Remember to use the 64-bit and 32-bit versions of the registers appropriately!

Solution:

```
_c0_f:
  push %rbp
  movq %rsp, %rbp
  subq $8, %rsp
  movl 24(%rbp), %eax
  addl 16(%rbp), %eax
  movl %eax, (%rsp)
```

```
  movl (%rsp), %eax
  imull $2, %eax
  addq $8, %rsp
  pop %rbp
  ret
_c0_main:
  push %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movl $0, %edi
  movl $0, %esi
  movl $0, %edx
  movl $0, %ecx
  movl $0, %r8d
  movl $0, %r9d
  movl $3, (%rsp)
  movl $5, 8(%rsp)
  call _c0_f
  addq $16, %rsp
  pop %rbp
  ret
```

Static Single Assignment Form

Recall the Fibonacci sequence:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad\qquad n > 1 \end{aligned}$$

Check out this lil program that computes the nth Fibonacci number:

```
int fib(int n) {
  if (n == 0) return 0;
  int a = 0;
  int b = 1;
  int i = 1;
  while (i < n) {
    int c = b;
    b = a + b;
    a = c;
    i++;
  }
  return b;
}
```

Checkpoint 4

Translate this program into abstract assembly, organized as basic blocks with parameterized labels.

Solution:

```
fib(n):
  if (n == 0)
    then done1()
    else pre_loop(n)

done1():
  return 0

pre_loop(n):
  a <- 0
  b <- 1
  i <- 1
  goto loop(n, a, b, i)

loop(n, a, b, i):
  if (i < n)
    then body(n, a, b, i)
    else done2(b)

body(n, a, b, i):
```

```
  c <- b
  b <- a + b
  a <- c
  i <- i + 1
  goto loop(n, a, b, i)

done2(b):
  return b
```

Checkpoint 5

Use generation counters to convert this basic block assembly into SSA.

Solution:

```
fib(n0):
  if (n0 == 0)
    then done1()
    else pre_loop(n0)

done1():
  return 0

pre_loop(n1):
  a0 <- 0
  b0 <- 1
  i0 <- 1
  goto loop(n1, a0, b0, i0)

loop(n2, a1, b1, i1):
  if (i1 < n2)
    then body(n2, a1, b1, i1)
    else done2(b1)

body(n3, a2, b2, i2):
  c0 <- b2
  b3 <- a2 + b2
  a3 <- c0
  i3 <- i2 + 1
  goto loop(n3, a3, b3, i3)

done2(b4):
  return b4
```

Checkpoint 6

Minimize the SSA program.

Solution:

```
fib(n0):
  if (n0 == 0)
    then done1()
    else pre_loop()

done1():
  return 0

pre_loop():
  a0 <- 0
  b0 <- 1
  i0 <- 1
  goto loop(a0, b0, i0)

loop(a1, b1, i1):
  if (i1 < n0)
    then body(a1, b1, i1)
    else done2()

body(a2, b2, i2):
  c0 <- b2
  b3 <- a2 + b2
  a3 <- c0
  i3 <- i2 + 1
  goto loop(a3, b3, i3)

done2():
  return b1
```

Checkpoint 7

Convert the minimized SSA program into assembly without label parameters.

Solution:

```
fib(n0):
  if (n0 == 0)
    then done1
    else pre_loop

done1:
  return 0

pre_loop:
  a0 <- 0
  b0 <- 1
  i0 <- 1
  a1 <- a0
  b1 <- b0
```

```
  i1 <- i0
  goto loop

loop:
  if (i1 < n0)
    then pre_body
    else done2

pre_body:
  a2 <- a1
  b2 <- b1
  i2 <- i1
  goto body

body:
  c0 <- b2
  b3 <- a2 + b2
  a3 <- c0
  i3 <- i2 + 1
  a1 <- a3
  b1 <- b3
  i1 <- i3
  goto loop

done2:
  return b1
```

Checkpoint 8

AAAA Yikes! What's a compiler optimization we could've applied to the program to make this less sad?

Solution: Copy propagation

Header Files in L3

Unlike in C, header files in L3 (and above) are only used to declare types and external functions. If a function is declared in a header file, then it may not be defined in the program – it is declared as *external*. External functions are defined in C source files, which are linked together with the assembly produced by your compiler.