15-411: Compiler Design — Fall 2019

Recitation 5: Calling Conventions — 4 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
```

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.

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 }
```

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!

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.