15-411: Compiler Design

Spring 2023

Recitation 5: Calling Conventions and SSA Solutions

24 February

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.

To avoid having callee-saved registers occupy a very long live range during register allocation, we can handle them separately. Prioritize allocating caller-saved registers; if they are insufficient, we assign assign callee-save registers before we resort to spilling, but we make sure to save them to the stack at the beginning of a function and restore them at the end. This is more efficient than always saving and restoring all callee-saved registers.

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

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 0

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 of _main()	
Previous %rbp	
b; Arg. 8 of f()	
a; Arg. 7 of f()	
Return address of f()	
main's %rbp	← %rbp
x	← %rsp

Checkpoint 1

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 and that stack grows downward!

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:

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

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 2

Translate this program into abstract assembly, organized as basic blocks with parametrized 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 3

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

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 4

Rewrite the SSA program using Φ-functions instead of parametrized labels (except for the first basic block fib).

If the parametrized label `foo`(x_i) : can be jumped to from 2 different lines **goto** `foo`(x_j) and **goto** `foo`(x_k), then we would switch to a non-parametrized label `foo` : but add the instruction $x_i \leftarrow \Phi(x_j, x_k)$ to the start of the basic block under `foo` :.

Solution:

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

```
  return 0
pre_loop:
  n1 <- Phi(n0)
  a0 <- 0
  b0 <- 1
  i0 <- 1
  goto loop
loop:
  n2 <- Phi(n1, n3)
  a1 <- Phi(a0, a3)
  b1 <- Phi(b0, b3)
  i1 <- Phi(i0, i3)
  if (i1 < n2)
    then body
    else done2
body:
  n3 <- Phi(n2)
  a2 <- Phi(a1)
  b2 <- Phi(b1)
  i2 <- Phi(i1)
  c0 <- b2
  b3 <- a2 + b2
  a3 <- c0
  i3 <- i2 + 1
  goto loop
done2:
  b4 <- Phi(b1)
  return b4
```

Checkpoint 5

Now minimize the Φ-function SSA program. Recall that we do this by repeatedly removing instructions of the form

$$t_i = \Phi(t_{x_1}, ..., t_{x_k})$$

whenever there exists a j such that all the x_n are either i or j, then replacing all instances of t_i with t_j.

Solution:

```
fib(n0):
  if (n0 == 0)
    then done1
    else pre_loop
done1:
  return 0
pre_loop:
  a0 <- 0
  b0 <- 1
  i0 <- 1
```

```
  goto loop
loop:
  a1 <- Phi(a0, a3)
  b1 <- Phi(b0, b3)
  i1 <- Phi(i0, i3)
  if (i1 < n0)
    then body
    else done2
body:
  c0 <- b1
  b3 <- a1 + b1
  a3 <- c0
  i3 <- i1 + 1
  goto loop
done2:
  return b1
```

Checkpoint 6

A very useful optimization that is made easy to implement by transforming programs into SSA form is copy and constant propagation. Since by definition each variable is only defined once in the program, whenever we see

- $x \leftarrow c$, we can replace all instances of x with c
- $x \leftarrow y$, we can replace all instances of x with y

Additionally, although it will not come up on the specific example we're working on for this checkpoint, whenever we see

- $\Phi(c, c)$, we can replace it with c
- $\Phi(x, x)$, we can replace it with x

Now apply this optimization to the minimized Φ-function SSA program from above.

Solution:

```
fib(n0):
  if (n0 == 0)
    then done1
    else pre_loop
done1:
  return 0
pre_loop:
  goto loop
loop:
  a1 <- Phi(0, b1)
  b1 <- Phi(1, b3)
  i1 <- Phi(1, i3)
  if (i1 < n0)
    then body
    else done2
body:
  b3 <- a1 + b1
  i3 <- i1 + 1
  goto loop
done2:
  return b1
```

Checkpoint 7

Convert the optimized Φ-function SSA program back into abstract assembly without Φ-functions, aka the de-SSA transformation.

For every occurence of $x \leftarrow \Phi(y, z)$ at the start of some basic block b, delete it, and instead insert $x \leftarrow y$ and $x \leftarrow z$ respectively at the end of each of b's predecessor blocks. Note that this will result

in x having multiple assignment sites, hence why the program is no longer in Static Single Assignment (SSA) form.

Solution:

```
fib(n0):
  if (n0 == 0)
    then done1
    else pre_loop
done1:
  return 0
pre_loop:
  a1 <- 0
  b1 <- 1
  i1 <- 1
  goto loop
loop:
  if (i1 < n0)
    then body
    else done2
body:
  b3 <- a1 + b1
  i3 <- i1 + 1
  a1 <- b1
  b1 <- b3
  i1 <- i3
  goto loop
done2:
  return b1
```

Tips and Hints for Lab3

- **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.
- **RBP:** You are not required to use %rbp as a base pointer, so you are allowed to treat it like a normal callee-saved register in your compiler.
- **Code Review:** Code Review happens one week after Lab3 is due. So if you haven't polished the style of your compiler and added a README describing the design of various passes of your compiler, now would be a good time to start. We are looking for good coding style and comments, modular design, and that both of you are familiar with all components of the implementation.