1 Topics

- Function calls
- Functions (prologues and epilogues)
- Virtual frame pointers
- Dynamic memory allocation
- Simple register allocation

One topic we haven’t talked about is how to do instruction selection for functions and function calls. These show up in the IR as either $\text{MOVE}(\text{dest, CALL}(f, e_1, \ldots, e_n))$ or as $\text{EXP}(\text{CALL}(f, e_1, \ldots, e_n))$.

2 Calling convention

A calling convention is a standardized way to invoke functions. Having a calling convention allows code generated by different compilers and languages to interoperate. Since we want to give you Xi libraries to compile against, it’s important that we follow the same calling conventions.

Unfortunately, there are multiple calling conventions for the x86-64. We will describe here the System V style calling convention used by Linux. The Microsoft calling convention is similar but differs in a few details.

Unlike in the 32-bit architecture, arguments are usually passed to functions entirely in registers. The first six word-size arguments are passed in the registers rdi, rsi, rdx, rcx, r8, and r9, in that order. For functions with more than 6 arguments, the remaining arguments are pushed onto the stack, in reverse order. Using reverse order supports functions with a variable number of arguments, though this is not a consideration in Xi. Figure[1] shows what the stack looks like as a procedure/function call proceeds. Note that stacks grow downward in this picture, so the top of the stack is at the lowest used address, which is at the bottom!

The code for doing a function call looks something like the following:

---

1 The full calling convention is more complex than described here, in order to handle struct arguments that are larger than one word. But Xi does not have these language features, simplifying matters.
Figure 2: Stack frame of a running function

```assembly
push e_n
push e_{n-1}
...
push e_7
mov e_6, r9
...
mov e_1, rdi
call f
mov rax, dest
add $(8*(n-6)), rsp
```

Note that the instruction `push x` is equivalent to `sub 8, rsp; mov x, (rsp)`. So the stack pointer register `rsp` always points to the top entry of the stack.

This code sequence shows how function results are handled, too. The result of a function, if any, comes back in the `rax` register. Once the function completes, it should clean up the stack so the stack pointer is back where it started. Thus, the final instruction adds the appropriate offset to reverse the effect on `rsp` of all the push instructions.

3 Inside the function

Once the function is entered, it sets up its stack frame to look like Figure 2. The region labeled “temporary storage” is used to store local variables and other temporaries that don’t fit into registers. Because the stack pointer can move around, it is common to use a different register, the frame pointer or base pointer register `rbp`. For example, a variable located immediately below the base pointer would be accessed with the memory operand `-4(rbp)` (or `[rbp-4]`, in Intel syntax). But if `rbp` is used, it is necessary to save the caller’s `rbp` before overwriting it. So `rbp` is saved on the stack immediately after the program counter `rip` of the caller.

Recall that we generate code for the body of a function defined as $f(x):\tau\{s\}$ as simply the statement translation $S[s]$. And the translation of `return e` is `MOVE(TEMP(RV), E[e]): return`, where `TEMP(RV)`
really a name for rax. Therefore the IR code generated for the declaration \( f(x:\text{int}, y:\text{int}) : \text{int} \ {\text{return}} \ x+y \) is something like \( \text{MOVE} (\text{TEMP}(RV), \text{ADD} (\text{TEMP}(x), \text{TEMP}(y))) ; \text{return} \). With appropriate instruction selection, we should get something like:

\[
\begin{align*}
\text{mov} & \ x, \text{rax} \\
\text{add} & \ y, \text{rax} \\
\ldots & \\
\text{return} & 
\end{align*}
\]

But this code does nothing to set up the stack frame or to tear it down, or even to move the function arguments into \( x \) and \( y \). These can be achieved by adding a function \textit{prologue} and \textit{epilogue}:

\[
\begin{align*}
f & : \text{push rbp} \\
& \text{mov} \ rbp, \ rsp \\
& \text{sub} \ 8^*\ell, \ rsp \\
& \text{mov} \ rdi, \ x \\
& \text{mov} \ rsi, \ y \\
& \text{mov} \ x, \ rax \\
& \text{add} \ y, \ rax \\
& \text{mov} \ rbp, \ rsp \\
& \text{pop} \ rbp \\
& \text{return}
\end{align*}
\]

Here we are assuming that the stack frame needs to contain \( \ell \) temporary words.

In fact, the ISA has an instruction to accomplish the first three instructions directly: \( \text{enter} \ 8^*\ell, \ 0 \) saves the frame pointer and adjusts the stack pointer. And the two instructions preceding \text{return} can be accomplished using \text{leave}.

\[
\begin{align*}
f & : \text{enter} \ 8^*\ell, \ 0 \\
& \text{mov} \ rdi, \ x \\
& \text{mov} \ rsi, \ y \\
& \text{mov} \ x, \ rax \\
& \text{add} \ y, \ rax \\
& \text{leave} \\
& \text{return}
\end{align*}
\]

In this case, the function doesn’t really need a frame pointer at all, since it doesn’t use it. Then we don’t need \text{enter} and \text{leave}. And a smart register allocator can choose \( x=\text{rdi}, \ y=\text{rsi} \), making two \text{mov} instructions superfluous:

\[
\begin{align*}
f & : \text{mov} \ rdi, \ rax \\
& \text{add} \ rsi, \ rax \\
& \text{return}
\end{align*}
\]

A couple of details to watch out for: first, the stack pointer \text{rsp} is required always to be 16-byte aligned when a \text{call} instruction is performed. (This is a requirement of various systems libraries.) Since the \text{call} instruction itself pushes an 8-byte word onto the stack (rip), the stack pointer is misaligned on entry. In the example function above, this is not a problem, because it is a leaf procedure that doesn’t call anything else.

Second, it is ordinarily extremely inadvisable to access memory below the current stack pointer, because interrupt routines are likely to overwrite anything there. However, Linux calling conventions introduce a “red zone” of two words that interrupt routines will not overwrite. This means that leaf procedures can use a small amount of local storage without bothering to set up a stack frame.
4 Caller-save vs. callee-save

In the previous example, we needed to save rbp before changing it. In general, the calling conventions define certain registers as callee-save, meaning that a called procedure must restore those registers to their original values before returning. In the current calling conventions, these registers are rbp, rsp, rbx, and r12–r15. Using these registers is not worth it if the overhead of saving them to memory is more than the speedup achieved by using them.

5 Eliminating the base pointer

The previous example shows that the frame pointer doesn’t need to be used in leaf procedures. We can also avoid using a register to keep track of the frame pointer when the offset between the stack pointer and the frame pointer is known at compile time. Suppose that the compiler knows the offset is $8 \times \ell$. Then any memory reference of the form $k(rbp)$ can be equally well written as $k'(rsp)$ where $k' = k + 8\ell$. Once all such references to rbp are removed from the code, there is no need to use rbp, and therefore no need to save it and restore it in the prologue and epilogue. If it is used, it can be used as a general register.

Although this trick is rather nice, it does mean that the distance between the two stack pointers must be statically known. This rules out using dynamic allocation on the stack, such as the nifty alloca() system call familiar to experienced C programmers.

6 Dynamic allocation

These notes are still under construction

7 Trivial register allocation

Doing a good job of register allocation is challenging. However, it is not hard to generate code if all temporaries are assigned to stack locations. For example, gcc -O0 does this.

Temporaries are assigned to distinct stack locations (e.g., -8(%rbp), -16(%rbp), etc.). Any given instruction uses at most 3 register operands, so instructions are inserted to read operand registers from the stack and to write results back to the stack.

Suppose that we have allocated temporary $t_1$ to location -8(%rbp). Then the abstract assembly instruction push $t_1$ is converted to concrete assembly that first loads the operand into a register, then performs the original instruction:

\[
\text{mov } -8(%rbp), \%rax \\
\text{push } \%rax
\]

For instructions that update temporaries, instructions are added afterward. So the instruction \text{mov } 8(t_1), \ t_2 might become:

\[
\text{mov } -8(%rbp), \%rax \\
\text{mov } 8(%rax), \%rdx \\
\text{mov } \%rdx, -16(%rbp)
\]

A couple of simple optimizations can help. On CISC instruction sets like x86-64, many instructions can read operands directly from memory or write results directly to memory, avoiding the need to add some instructions. Also, some temporaries can be allocated to registers, as long as three registers are reserved for the job of shuttling temporaries on and off the stack. However, the real solution to the problem is to do proper register allocation.