Subroutines and Control Abstraction

8.2.2 Case Studies: C on the MIPS; Pascal on the x86

To make stack management a bit more concrete, we present a pair of case studies, one for a simple language (C) on a simple RISC machine (the MIPS), the other for a language with nested subroutines (Pascal) on a CISC machine (the x86).

SGI C on the MIPS

An overview of the MIPS architecture can be found in Section 5.4.5. As noted in that section, register r31 (also known as ra) is special-cased by the hardware to receive the return address in subroutine call (jal—jump-and-link) instructions. In addition, register r29 (also known as sp) is reserved by convention for use as the stack pointer, and register r30 (also known as fp) is reserved by convention for the frame pointer, if any. The details presented here correspond to version 7.3.1.3m of the SGI MIPSpro C compiler, generating 64-bit code at optimization level -O2. The conventions for 32-bit code are different, and future versions of the compiler may be different as well.

A typical MIPSpro stack frame appears in Figure 8.10. The sp points to the last used location in the stack (note that many other compilers, including some for the MIPS, point the sp at the first unused location). Since the size of every object in the stack is known at compile time in C, a separate frame pointer is not strictly needed, and the MIPSpro compiler usually does without: it uses displacement-mode offsets from the sp for everything in the current stack frame. The principal exception occurs in subroutines whose arguments or local variables are so large that they exceed the reach of displacement addressing; for these the compiler makes use of the fp.

Argument Passing Conventions

Arguments in the process of being passed to the next routine are assembled at the top of the frame, and are always accessed via offsets from the sp. The first eight arguments are passed in integer registers r4–r11 or floating-point registers f12–f19, depending on type. Additional
arguments are passed on the stack. Record arguments (structs) are implicitly divided into 64-bit “chunks,” each of which is passed as if it were an integer. A large struct may be passed partly in registers and partly on the stack.

As noted in the main text, space is reserved in the stack for all arguments, whether passed in registers or not. In effect, each subroutine begins with some of its arguments already loaded into registers, and with “stale” values in memory. This is a normal state of affairs; optimizing compilers keep values in registers whenever possible. They “spill” values to memory when they run out of registers, or when there is a chance that the value in memory may be accessed directly (e.g., through a pointer, a reference parameter, or the actions of a nested subroutine). The fp, if present, points at the first (top-most) argument.

The argument build area at the top of the frame is designed to be large enough to hold the largest argument list that may be passed to any called routine. This convention may waste a bit of space in certain cases, but it means that arguments need not be “pushed” in the usual sense of the word: the sp does not change when they are placed into the stack.

For languages with nested subroutines (C of course is not among them), MIPS compilers generally use register r2 to pass the static link. In all languages, registers r2 and f0 (depending on type) are used to return scalar values from functions. Values of type long double are returned in the register pair (f0, f2). Record values (structs) that will fit in 128 bits are returned in (r2, r3). For larger

Copyright © 2009 by Elsevier Inc. All rights reserved.
8.2.2 Case Studies: C on the MIPS; Pascal on the x86

structs, the compiler passes a hidden first argument (in \( r4 \)) whose value is the address into which the return value should be placed. If the return value is to be assigned immediately into a variable (e.g., \( x = \text{foo}() \)), the caller can simply pass the address of the variable. If the value is to be passed in turn to another subroutine, the caller can pass the appropriate address within its own argument build area. (Writing the return value into this space will probably destroy the returning function's own arguments, but that's fine: at this point they are no longer needed.) Finally, though one doesn't see this idiom often (and most languages don't support it), C allows the caller to extract a field directly from the return value of a function (e.g., \( x = \text{foo}().a + y; \)); in this case the caller must pass the address of a temporary location within the “local variables and temporaries” part of its stack frame.

**Calling Sequence Details** The calling sequence to maintain the MIPSpro stack is as follows. The caller

1. saves (into the “local variables and temporaries” part of its frame) any caller-saves registers whose values are still needed
2. puts up to eight scalar arguments (or “chunks” of structs) into registers
3. puts the remaining arguments into the argument build area at the top of the current frame
4. performs a \( \text{jal} \) instruction, which puts the return address in register \( ra \) and jumps to the target address

The caller-saves registers consist of \( r2–r15, r24, r25, \) and \( f0–f23 \). In a language with nested subroutines, the caller would place the static link into register \( r2 \) immediately before performing the \( \text{jal} \).

In its prologue, the callee

1. subtracts the frame size (the distance between the first argument and the \( sp \) in Figure 8.10) from the \( sp \)
2. if the frame pointer is to be used, copies its value into an available temporary register (typically \( r2 \)), then adds the frame size to the \( sp \), placing the result in the \( fp \) (this effectively moves the old \( sp \) into the \( fp \); note that an \( \text{add} \) is as fast as a simple move, so there was no harm in updating the \( sp \) first)
3. saves any necessary registers into the middle of the newly allocated frame, using the \( sp \) or, if available, the \( fp \) as the base for displacement-mode addressing

Saved registers include (a) any callee-saves temporaries (\( r16–r23 \) and \( f24–f31 \)) whose values may be changed before returning; (b) the \( ra \), if the current routine

---

1 Like all branch instructions on the MIPS, \( \text{jal} \) has an architecturally visible branch delay slot. The load delay slot was eliminated in the MIPS II version of the ISA; all recent MIPS processors are fully interlocked.
is not a leaf or if it uses the ra as an additional temporary; and (c) the temporary register containing the old fp from Step 2, if the current routine needs a frame pointer, or the fp itself if the current routine does not need a frame pointer, but uses the fp as an additional temporary.

In its epilogue, immediately before returning, the callee

1. places the function return value (if any) into r2, r3, f0, f2, or memory as appropriate
2. restores saved registers (if any), using the sp or, if available, the fp as the base for displacement-mode addressing; if the current routine needed a frame pointer, the saved fp is “restored” into a temporary register
3. deallocates the frame by moving the fp into the sp or adding the frame size to the sp
4. moves the value in the temporary register of step 2 (if any), into the fp
5. performs a jr ra instruction (jump to address in register ra)

Finally, if appropriate, the caller moves the return value to wherever it is needed. Caller-saves registers are restored lazily over time, as their values are needed.

To support the use of symbolic debuggers such as gdb and dbx, the compiler generates a variety of assembler pseudo-ops that place information into the object file symbol table. For each subroutine, this information includes the starting and ending addresses of the routine, the size of the stack frame, an indication as to which register (usually sp or fp) is the base for local objects, an indication as to which register (usually ra, if any) holds the return address, and a list of which registers were saved.

**GNU Pascal on the x86**

To illustrate the differences between CISC and RISC machines, our second case study considers the x86, still the world’s most popular instruction set architecture. (An overview of the processor appears in Section 5.4.5). To illustrate the handling of nested subroutines and closures, we consider a Pascal compiler, namely version 3.2.2 of the GNU Pascal compiler, gpc. (Ada compilers [e.g., GNU’s gnat] handle these features in similar ways, but Ada’s many extra features would make the case study much more complex.)

On modern implementations of the x86, ordinary store instructions may make better use of the pipeline than is possible with push. Most modern compilers for the x86, including gcc (on which gpc is based), therefore employ an argument build area similar to that of the previous case study. By default gpc and gcc still use a separate frame pointer, partly for the sake of uniformity with other architectures and languages (gcc is highly portable), and partly to simplify the implementation of library mechanisms that allocate space dynamically in the current stack frame (see Exercise 8.37).

The special instructions for subroutine calls vary significantly from one CISC machine to another. The ones most often used on the x86 today are relatively simple. The call instruction pushes the return address onto the stack, updating
8.2.2 Case Studies: C on the MIPS; Pascal on the x86

Figure 8.11 Layout of the subroutine call stack for the GNU Pascal compiler, gpc. The return address and saved fp are present in all frames. All other parts of the frame are optional; they are present only if required by the current subroutine. In x86 terminology, the sp is named esp; the fp is ebp (extended base pointer). SL marks the location that will be referenced by the static link of any subroutine nested immediately inside this one.

The sp, and branches to the called routine. The ret instruction pops the return address off the stack, again updating the sp, and branches back to the caller. Several additional instructions, retained for backward compatibility, are typically not generated by modern compilers, because they were designed for calling sequences with an explicit display and without an argument build area, or because they don't pipeline as well as equivalent sequences of simpler instructions.

Argument Passing Conventions Figure 8.11 shows a stack frame for the x86. As in the previous case study, the sp points to the last used location on the stack. Arguments in the process of being passed to another routine are accessed via offsets from the sp; everything else is accessed via offsets from the fp. All arguments are passed in the stack. Register ecx is used to pass the static link. That link will point at the last saved register (the saved fp if there are no others) in the frame of the lexically surrounding routine, immediately below that routine’s own static link, if any.

Functions return integer or pointer values in register eax. Floating-point values are returned in the first of the floating-point registers, st(0). For functions that return values of constructed types (records, arrays, or sets), the compiler passes a hidden first argument (on the stack) whose value is the address into which the return value should be placed.

Copyright © 2009 by Elsevier Inc. All rights reserved.
**Calling Sequence Details**  The calling sequence to maintain the \texttt{gpc} stack is as follows. The caller

1. saves (into the “local variables and temporaries” part of its frame) any caller-saves registers whose values are still needed
2. puts arguments into the build area at the top of the current frame
3. places the static link in register \texttt{ecx}
4. executes a \texttt{call} instruction

The caller-saves registers consist of \texttt{eax}, \texttt{edx}, and \texttt{ecx}. Step 1 is skipped if none of these contain a value that will be needed later. Step 2 is skipped if the subroutine has no parameters. Step 3 is skipped if the subroutine is declared at the outermost level of lexical nesting. The \texttt{call} instruction pushes the return address and jumps to the subroutine.

In its prologue, the callee

1. pushes the \texttt{fp} onto the stack, implicitly decrementing the \texttt{sp} by 4 (one word)
2. copies the \texttt{sp} into the \texttt{fp}, establishing the frame pointer for the current routine
3. pushes any callee-saves registers whose values may be overwritten by the current routine
4. pushes the static link (\texttt{ecx}) if this is not a leaf
5. subtracts the remainder of the frame size from the \texttt{sp}

The callee-saves registers are \texttt{ebx}, \texttt{esi}, and \texttt{edi}. Registers \texttt{esp} and \texttt{ebp} (the \texttt{sp} and \texttt{fp}, respectively) are saved by Steps 1 and 2. The instructions for some of these steps may be replaced with equivalent sequences by the compiler's code improver, and mixed into the rest of the subroutine by the instruction scheduler. In particular, if the value subtracted from the \texttt{sp} in Step 5 is made large enough to accommodate the callee-saves registers, then the pushes in Steps 3 and 4 may be moved after Step 5 and replaced with \texttt{fp}-relative stores.

In its epilogue, the callee

1. sets the return value
2. restores any callee-saved registers
3. copies the \texttt{fp} into the \texttt{sp}, deallocating the frame
4. pops the \texttt{fp} off the stack
5. returns

Finally, as in the previous case study, the caller moves the return value, if it is in a register, to wherever it is needed. It restores any caller-saves registers lazily over time.

Because Pascal allows subroutines to nest, a subroutine \(S\) that is passed as a parameter from \(P\) to \(Q\) must be represented by a closure, as described in Section 3.6.1. In many compilers the closure is a data structure containing the...
8.2.2 Case Studies: C on the MIPS; Pascal on the x86

address of $S$ and the static link that should be used when $S$ is called. In gpc, however, the closure contains an x86 code sequence known as a trampoline: typically a pair of instructions to load ecx with the appropriate static link and then jump to the beginning of $S$. The trampoline resides in the “local variables and temporaries” section of $P$’s activation record. Its address is passed to $Q$. Rather than “interpret” the closure at run time, $Q$ actually calls it. One advantage of this mechanism is its interoperability with gcc, in which C functions passed as parameters are simply code addresses. In fact, if $S$ is declared at the outermost level of lexical nesting, then gpc too can pass an ordinary code address; no trampoline is required.

CHECK YOUR UNDERSTANDING

55. For one or both of our case studies, explain which aspects of the calling sequence and stack layout are dictated by the hardware, and which are a matter of software convention.

56. On the MIPS some compilers make the $sp$ point at the last used word on the stack, while others make it point at the first unused word. On the x86 all compilers make it point at the last used word. Why the difference?

57. Why don’t the MIPSpro compiler and gpc restore caller-saves registers immediately after a call?

58. What is a subroutine closure trampoline? How does it differ from the usual implementation of a closure described in Section 3.6.1? What are the comparative advantages of the two alternatives?

DESIGN & IMPLEMENTATION

Executing code in the stack

A disadvantage of trampoline-based closures is the need to execute code in the stack. Many machines and operating systems disallow such execution, for at least two important reasons. First, as noted in Section 5.1, modern microprocessors typically have separate instruction and data caches, for fast concurrent access. Allowing a process to write and execute the same region of memory means that these caches must be kept mutually consistent (coherent), a task that introduces significant hardware complexity. Second, many computer security breaches involve so-called buffer overflow attacks, in which an intruder exploits the lack of array bounds checking to write code into the stack, where it will be executed when the current subroutine returns. Such attacks are only possible on machines in which writable data are also executable.