Chapter 6: Activation Records

In this chapter, we reinforce the understanding of the following concepts from a computer architecture course:

- Memory allocation methods for different kinds of variables.
- Using registers to store local variables and temporary results.
- Using registers to pass arguments and return results (for function calls).
- Stack frames (also known as *activation records*).
- Call/return sequence.

From an introductory computer organization course or an assembly programming course, we know that

- A stack is maintained in the program's virtual address space. Variables local to a function are allocated to the $stack\ frame$, also known as the $activation\ record$, of that function.
- Variables and constants which are shared among different functions are allocated elsewhere.
  - Variables with fixed sizes known at compile time are allocated to $static$ locations.
  - Dynamic data structures are allocated at run-time on the $heap$.

A Calling Sequence

The following actions are divided between the caller and the callee:

1. Evaluates actual arguments and puts values in callee's AR.
2. Stores return address in callee's AR.
3. Stores the caller's *frame pointer* register, or called the caller's AR pointer, in callee's AR. (Current AR pointer is called the *control link* in callee's AR.)
4. Modifies the frame pointer %fp, making it point to callee's AR.

5. Modifies the stack pointer $\%\mathrm{sp}$, making it point to the to top of the stack.
6. Branches to callee's first instruction.
7. Callee begins execution.

Are there other register contents to be stored? Who stores them? We will discuss $Caller\text{-}save$ vs. $callee\text{-}save$.

A Returning Sequence

1. Caller needs to retrieve the function return value. How?
2. (In the $call\text{-}by\text{-}value$ scheme, no return values are stored in actual arguments.)
3. Restores saved stack pointer for caller (= current AR pointer).
4. Restores saved register contents for caller.
5. Return to the caller.

The memory references required to read and modify the stack contents can be time consuming. The number of such memory references can be reduced by using registers.

- Passing arguments through registers.
  - Most functions have few arguments.
  - We can use a few registers to pass the arguments.
  - The rest of the arguments, if any, can be passed in the stack frame.
- Returning function's results through registers.
  - We can use another register to return the function's result, if any, to the caller.

- Carefully considering how to save the register contents.
  - If we let the callee save the registers, then a function needs to save only those registers which it modifies. Each function will save such registers at its entry and resture them before the exit.
  - If we let the caller save the registers, then a function needs to save registers only if it calls others. Moreover, before it calls another function, the caller needs to save only those registers whose values are still needed after the call returns from the callee. When the call returns, the caller restores the saved registers.

- Caller-save and callee-save each has its upside and downside, so we divide the registers into two groups.
  - Callee-saved registers: If function f writes to a callee-saved register r, then r must be saved to the stack at the entry of f, and r must be restored before f exits.
  - Caller-saved registers: Suppose f writes to a caller-saved register r before calling another function g and suppose the written value in r will be read again in f after g returns. f must save r to the stack before calling g and restore r after the call returns.

How to Take Advantage of Caller-Saved/Callee-Saved Distinction?

- A $leaf$ function (i.e. a function that makes no function calls) should use as many caller-saved registers as possible.
- If a function f needs to save a caller-saved register $r1$ many times then it is better to use a callee-saved register $r2$ for that variable value instead of using $r1$. Then f needs to save $r2$ only once

Obviously, the return-value register must be caller-saved.

- MIPS has two v registers for return value and for expression evaluation
- 4 a registers for passing arguments.
- 10 t registers for temporary values.
- 8 s registers which are callee-saved.
- $gp, $sp, $fp, $ra, $zero
- $k0, $k1 used by the OS kernel
- $at for the assembler.
- Program Example 1: compute factorial(n)
- Program Example 2: sorting

A MIPS example to show register usage

The C source code:

```
sort(int v[ ], int n)
{    int i, j;
   for (i=0; i<n; i=i+1) {
     for (j=i-1;j>=0 &&v[j]>v[j+1]; j=j-1)
     {
     swap(v, j);
     }
 }
}
```

The assembly code: (see next page)

```
sort:   addi   $sp, $sp, -20      # make room on stack for 5 registers
        sw     $ra, 16($sp)
        sw     $s3, 12($sp)
        sw     $s2, 8($sp)
        sw     $s1, 4($sp)
        sw     $s0, 0($sp)

        move   $s2, $a0           #copy address of v into $s2
        move   $s3, $a1           #copy n into $s3

        move   $s0, $zero         # i = 0; beginning of outer loop
for1tst: slt   $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3
        beq    $t0, $zero, exit1  # if $t0 = 0 goto exit

        addi   $s1, $s0, -1       # j = i - 1; beginning of inner loop
for2tst: slti  $t0, $s1, 0        # slti uses an immediate operand "0"
        bne    $t0, $zero, exit2
        add    $t1, $s1, $s1      # $t1 = j * 2
        add    $t1, $t1, $t1      # $t1 = j * 4
        add    $t2, $s2, $t1      # $t2 = v + j * 4
        lw     $t3, 0($t2)        # v[j]
        lw     $t4, 4($t2)        # v[j+1]
        slt    $t0, $t4, $t3
        beq    $t0, $zero, exit2

        move   $a0, $s2           # 1st parameter of swap is v
        move   $a1, $s1           # 2nd parameter of swap is j
        jal    swap
        addi   $s1, $s1, -1       # j = j - 1
        j      for2tst            # jump back to test of inner loop

exit2:  addi   $s0, $s0, 1        # i = i + 1
        j      for1tst            # jump back to test of outer loop

exit1:                            # restoring registers
        lw     $ra, 16($sp)
        lw     $s3, 12($sp)
        lw     $s2, 8($sp)
        lw     $s1, 4($sp)
        lw     $s0, 0($sp)
        addi   $sp, $sp, 20
        jr     $ra
```

GCC Compiler Structure

- Given a C program source code, the command "gcc" causes cpp and $\mathrm{cc1}$ to run, successively, before assembling, linking and producing the executable.
  - cpp – the preprocessor which processes the macros, comments, etc. A number of library routines in cpp are called from the parser to recognize tokens.
  - $\mathrm{cc1}$ – the C compiler itself.
- Our focus is on $\mathrm{cc1}$, whose $\mathrm{main}\ ()$ function is defined in $toplev.c$.

– After initialization of a bunch of data structures, $\mathrm{compile\_file(filename)}$ is called for each source file.
– Within $\mathrm{compile\_file(filename)}$, $\mathrm{yyparse()}$ is called to invoke the parser.
– All the main compilation passes are invoked from within $\mathrm{yyparse()}$ after the parsing.
– After returning from $\mathrm{yyparse()}$, the compilation is essentially done.

The Parser

- Defined in *c-lex.c*, yyparse() is a "thin wrapper" around the real parser yyparse_1() which is defined in *c_parse.c*.
- The real parser is generated automatically by BISON from the YACC source file *c-parse.y*.
  - Since BISON names the generated parser "yyparse", GCC renames that function name to yyparse_1.
  - As usual, the parser gets the tokens by calling yylex(). In GCC's case, yylex() is manually embedded in the parser.

– $\mathrm{yylex}()$ in GCC actually wraps $\_\mathrm{yylex}\ ()$, which in turn simply converts the tokens returned from $\mathrm{c\_lex}\ ()$ to tokens recognized by the parser.

– The real lexer $\mathrm{c\_lex}()$, defined in *c-lex.c*, calls a number cpp library routines to get tokens.

– The high-level IR trees are generated as a function body is parsed. The tree construction operations are defined in *tree.c*, etc.

– After parsing a whole function, $\mathrm{finish\_function}\ ()$ is called to finish the rest of the compilation of the function. (C.f. *c-parse.y*.)

The Rest of the Passes

- Defined in *c-decl.c*, finish_function () invokes tree optimization routines. The high-level trees are maintained during the compilation of the entire function, which is a major improvement from the last version of GCC.
- finish_function () then calls c_expand_body (), which generates RTL for the function body before calling rest_of_compilation () to perform optimizations on the RTL.