Chapter 9

Procedures[1]

A high level language programmer uses procedures for several reasons. Code becomes modular, facilitating later modification. The writing of code within a modular program can be done by more than one programmer. All that is necessary is that the functionality and parameters of the different procedures be well defined.

A compiler needs to generate assembly langugage code for implementing procedure calls, returns, and parameter passing. There are many correct ways for doing these tasks. The high-level language specifies rules for syntax and functionality, and a compiler will implement a set of conventions. The conventions vary from computer to computer, and they vary from language to language. This chapter discusses how procedures are implemented at the assembly language level.

9.1 MAL Procedure Call and Return Mechanisms

Here are the four steps that need to be accomplished in order to call and return from a procedure.

1. save return address
2. procedure call
3. execute procedure
4. return

From Chapter 2, the SAL instructions that invoke a procedure by saving a return address and calling the procedure are the following:

```
            la    proc1_ret, ret_addr
            b     proc1
ret_addr:
```

MAL provides a single instruction that does both operations. The MAL `jal` (jump and link) instruction simplifies the procedure call sequence. The instruction

```
jal procedure_label
```

does two things. It places the address of the instruction *following* the `jal` instruction into register $31 (also called $ra), and it branches to the instruction labeled `procedure_label`. The choice of register $31 is arbitrary, but it it is fixed. Its use is implied by the `jal` instruction.

The MAL `jr` (jump register) instruction is convenient to use for procedure returns. Its execution causes the value contained in the register specified to be loaded into the Program Counter. The effect of this is an unconditional jump to the address contained in the register.

1. Copyright 1999, Oxford University Press. Use by permission only.
Contact Peter Gordon: pcg@oup-usa.org

Figure 1 contains skeleton code for a procedure that has no parameters. The procedure call and return use `jal` and `jr` instructions.

```
.text
      .
      .
      .
call: jal proc
      .
      .
      .
      done
proc: # procedure code here
      .
      .
      .
      jr $ra
```

Figure 1: Procedure implementation using `jal` and `jr` instructions.

9.2 Dynamic Storage Allocation

Placing a return address in a register works fine as long as there are no nested procedure calls. A nested call is a procedure call within the body of a procedure. If a nested procedure is invoked, the value held in register $ra will be overwritten. To avoid the loss of return addresses, a safe place for storing return addresses is needed. For each procedure that has been invoked but not completed, a return address must be saved. Note that once a procedure has returned, the space for the return address is no longer needed, and can be reused (for example, to store a return address for a newly invoked procedure). A method that can dynamically allocate space for procedures when they are invoked and deallocate the space when they return is needed.

The amount of memory required to save all the return addresses is therefore limited only by the number of levels of procedure calls permitted. In many modern programming languages this limit is very large. To provide space for storing many return addresses, a stack is employed. The space is said to be *dynamically allocated*.

Many computer systems implement a stack as part of the environment provided when a program is running. This stack is referred to as the **system stack**. It is such an important and frequently used structure that some computers provide assembly language support for accessing the stack efficiently. The MIPS RISC architecture system stack has its bottom of the stack (also known as the base) at a very large memory address. As items are pushed on the stack, it grows towards smaller memory addresses. By convention, register $29 (also called $sp) is a stack pointer for the system stack. It contains the address of the first empty location at the top of the stack. It is initialized prior to the beginning of a program's execution. Figure 2 shows a diagram of the initial

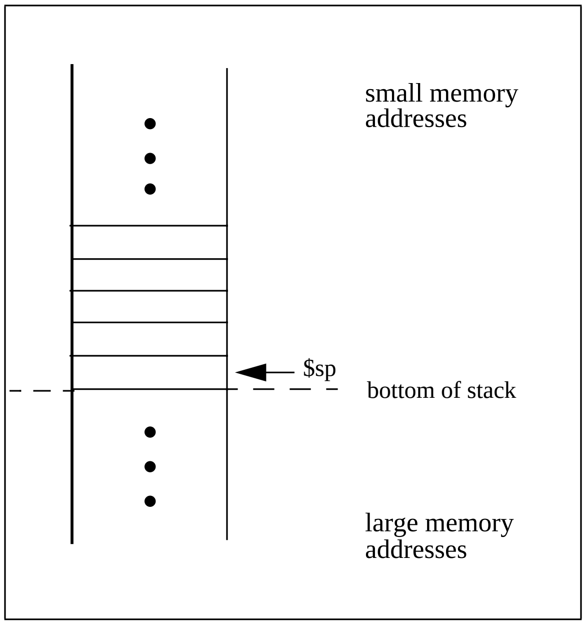

Figure 2: Initial (empty) state of the system stack.

state of the system stack.

A push operation to the system stack can be coded as

```
      sw $8, 0($sp)
      sub $sp, $sp, 4
```
or
```
      sub $sp, $sp, 4
      sw $8, 4($sp)
```
where the content of register $8 is the data being pushed onto the stack. A pop operation from the system stack can be coded as

```
      add $sp, $sp, 4
      lw $8, 0($sp)
```
or
```
      lw $8, 4($sp)
      add $sp, $sp, 4
```

Using the Stack for Return Addresses

One common use of the system stack is saving return addresses. They can be pushed onto the stack once a procedure has been called using `jal`, and popped off just before a return instruction `jr` is executed. Figure 3 contains a MAL procedure that implements a power function. Given a

```
.text
      .
      .
      .
      li $16, 10        # base 10 calculations
      li $18, 1         # $18 will contain the result
      move $19, $17     # $19 is a counter
                        # $17 contains the power
      jal power
      .
      .
      done
      .
      .
      .
power: sub $sp, $sp, 4  # save return address by
      sw $ra, 4($sp)    # pushing it on the stack
if:   sub $19, $19, 1
      blez $19, endif

      jal power         # recursive procedure call
endif: mul $18, $18, $16 # $16 contains base
      lw $ra, 4($sp)    # restore return address by
      add $sp, $sp, 4   # popping it off the stack
return: jr $ra
```

Figure 3: MAL implementation of power function.

base and a power, the procedure calculates $\texttt{base}^{\texttt{power}}$.

The implementation given in Figure 3 uses global variables instead of parameters. This is done in order to give an example of using the system stack for saving a return address, while ignoring the issue of parameter passing.

The procedure is **recursive**. Recursive procedure calls are a special case of nested procedure calls. An example of recursion is when a procedure directly calls itself. Because of the recursion, it is necessary that return addresses be saved on the stack so that they are not overwritten and lost. After being invoked, the procedure pushes its return address (saved in register $ra by the `jal` instruction) onto the stack. Before it returns, the procedure must restore the return address by popping it off the stack into register $ra.

A procedure that does not call any other procedures is known as a **leaf procedure**. It does not need to store its return address onto the stack - the value can be held in register $ra. It *may* execute the push and pop, but the push and pop of a return address is not required. If it executes the push, it must also execute the pop.

9.3 Activation Records

The compiler's role is to correctly generate assembly language. The code generated must correctly implement its high level language rules. The rules include those of **scope**. Scoping rules generally refer to which variables, memory locations, and other procedures an active procedure has access to. For example, the scope of a local variable declared within a C function is only within that procedure. This implies that variables of the same name declared within different functions are not related, and they may be assigned different values at the same time.

When a procedure is invoked, a new **environment** is created. In this newly created environment, new local variables are defined, while the values of the previous environment must be preserved. Variables for holding intermediate values during a computation like expression evaluation may also be needed. Like return addresses, these local variables are dynamic data, whose values must be preserved over the lifetime of the procedure, but not beyond its termination. At the termination of the procedure, the current environment disappears and the previous environment must be restored.

At procedure invocation, memory space must be allocated for holding the information associated with the new environment. This space is used not only for return address, but also for preserving the values from other registers, and as a place to keep parameters. The allocated space is a single block of memory, known as an **activation record** or **stack frame**. In general, an activation record consists of all the information that corresponds to the state of a procedure. It contains enough information about a procedure for nested procedure calls to be handled correctly. An activation record is pushed onto the stack in the same way that words are pushed onto a stack, except that an activation record will be larger than a word. When returning from a procedure, the activation record associated with its invocation is popped from the stack. Figure 4 shows skeletons of C

```
main()
{
   A();
}

A()
{
   B();
   C();
}

B()
{
   D();
}

C()
{}

D()
{
   E();
}

E()
{}
```

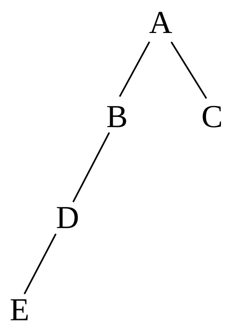

Figure 4: Example code block and call tree.

code and its associated call tree for an example. The tree shows the nesting of procedure calls, and the root of the tree is shown as procedure A.

The stack pointer must be adjusted by the size of the activation record. This size varies for different procedures. The adjustment of the stack pointer allocates space for the activation record. When the procedure returns to the calling program, the activation record is popped off the stack. The stack pointer must be adjusted by the same amount, leaving the stack the same size it was prior to the invocation of the procedure.

Figure 5 follows the state of the stack for the code given in Figure 4. The state of the stack is

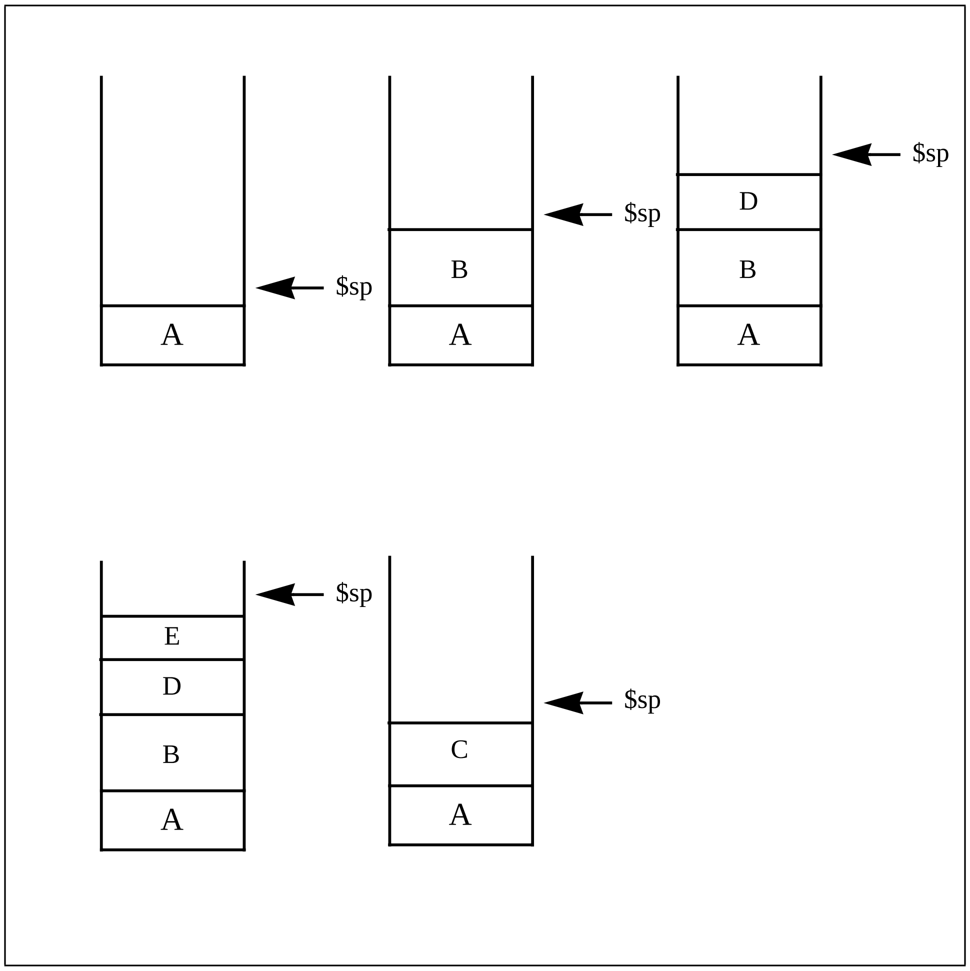

Figure 5: Stack immediately following each new procedure invocation.

shown immediately following each new procedure invocation. The letters represent the activation record for the procedures.

Skeleton MAL code for this example is given below. The size of an activation record for leaf functions is arbitrarily set to be four words. Non-leaf procedures are assigned an activation record size of six words.This code shows the allocation and deallocation of the activation records. It also shows the saving and restoring of return addresses within the stack frame. The first word within the frame is arbitrarily chosen as the position for the return address.

```
# main()
      jal A
      .
      .
      .
      done

A:    sub $sp, $sp, 24 # allocate frame for A
      sw $ra, 4($sp) # save return address in A’s frame
      jal B
      jal C
      lw $ra, 4($sp) # restore A’s return address
      add $sp, $sp, 24 # deallocate A’s frame
      jr $ra

B:    sub $sp, $sp, 24 #allocate frame for B
      sw $ra, 4($sp) # save return address in B’s frame
      jal D
      lw $ra, 4($sp) # restore B’s return address
      add $sp, $sp, 24 # deallocate B’s frame
      jr $ra

C:    sub $sp, $sp, 16 # allocate frame for C
      sw $ra, 4($sp) # unnecessary save of C’s return address
      lw $ra, 4($sp) # restore C’s return address
      add $sp, $sp, 16 # deallocate C’s frame
      jr $ra

D:    sub $sp, $sp, 24 # allocate frame for D
      sw $ra, 4($sp) # save return address in D’s frame
      jal E
      lw $ra, 4($sp) # restore D’s return address
      add $sp, $sp, 24 # deallocate D’s frame
      jr $ra

E:    sub $sp, $sp, 16 # allocate frame for E
      sw $ra, 4($sp) # unnecessary save of E’s return address
      lw $ra, 4($sp) # restore E’s return address
      add $sp, $sp, 16 # deallocate E’s frame
      jr $ra
```

One item of interest that this example does not show is a saving and restoring of the return address from `main()`. The operating system could view the execution of a program as a procedure call. Given this view, the main program would need to save its return address before calling any procedures, restore the return address, and then exit the program with `jr $ra`. The given implementation presumes that `done` is an explicit call to the operating system with information that the program is finished. Therefore, the operating system is free to execute other programs.

The example given so far presumes that the stack pointer is freely moved to accomodate the allocation and deallocation of activation records. A more traditional (and usually more efficient) implementation maintains a second pointer into the stack called a **frame pointer**. The frame pointer always points to the start of the currently invoked procedure’s activation record. The stack pointer is used to always point to the top of the stack. In the implementation of MAL, this is the empty location at the top of the stack. Then, within a procedure, the stack pointer may be used for pushing and popping temporary values used in expression evaluation, without affecting the access to values within the current activation record. All accesses within the activation record are done using offsets from the frame pointer. A register is often dedicated to be a frame pointer.

The following MAL code shows the implementation of procedure B assuming use of a frame pointer. Register $16 is chosen to be the frame pointer for this example.

```
B:    sub $sp, $sp, 24  # allocate B’s activation record
      sw $ra, 12($sp)   # save B’s return address
      sw $16, 16($sp)   # save caller’s frame pointer
      add $16, $sp, 24  # set frame pointer to B’s activation record

      # procedure B’s body here

      lw $ra, -12($16)  # restore B’s return address
      move $8, $16      # save frame pointer temporarily
      lw $16, -8($16)   # restore caller’s frame pointer
      move $sp, $8      # deallocate B’s activation record
      jr $ra
```

There are several policy issues set in this example. One is that the allocation of the activation record for procedure B is done within B itself. This is benefical to a compiler, because it means that the code of the caller does not need knowledge of the size of the activation record. Another point in this code is that the deallocation of both temporary values pushed onto the stack and the current activation record is a matter of setting the stack pointer to the current value of the frame pointer. Figure 6 contains diagrams of the stack before and after the invocation of procedure B.

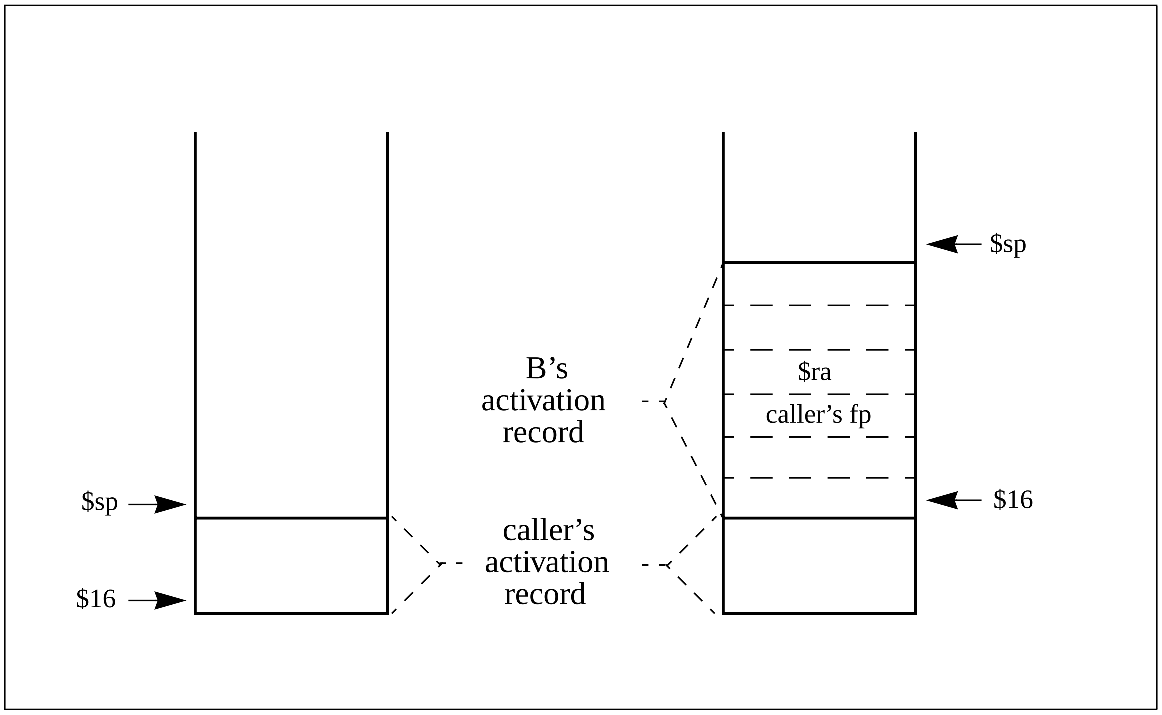

Figure 6: Stack contents before and after invocation of procedure B.

After procedure B returns, the state of the stack is the same as it was before procedure B was called.

9.4 Parameter Passing

Passing parameters to a function or procedure is straightforward using space within the activation record for the procedure. A compiler will either use a fixed size activation record, or it will calculate the correct amount of memory space needed to hold all parameters to a procedure. After determining the size of the activation record, a compiler fixes the position of the parameters within the activation record. Memory space for the activation record is allocated, and then the calling program can place parameters into the activation record.

To keep the calling program from needing to know the size of the called procedure's activation record, the parameters will be placed onto the stack between the calling program and called procedure's activation record. They will be included as part of the called procedure's activation record, but the caller does not need to allocate space for the remainder of the called procedure's activation record. Figure 7 shows the state of the stack through the setup of parameters. The initial

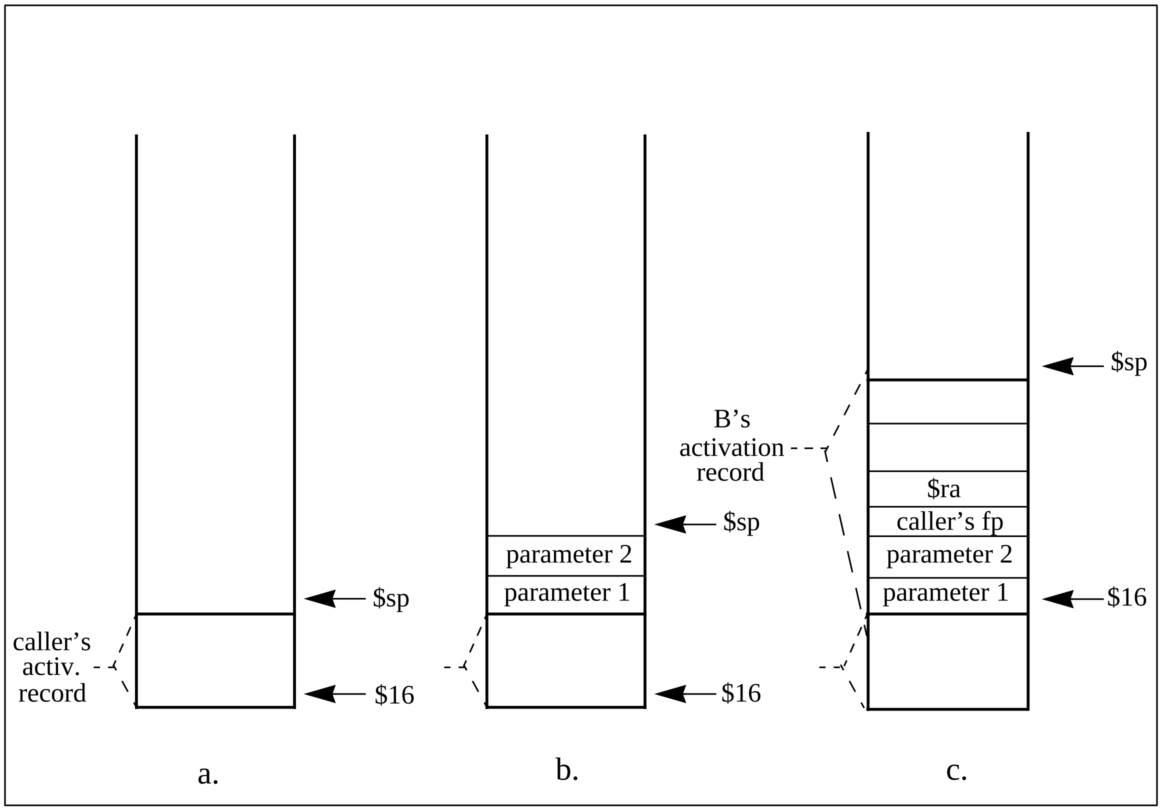

Figure 7: Stack state during parameter setup.

state of the stack has the caller's activation record and frame pointer as shown in Figure 7a. To set up the procedure call, two parameters are pushed onto the stack, to give the state of the stack in Figure 7b. Then, control is passed to procedure B (using `jal`). Procedure B allocates space for the remainder of its activation record, and then sets it up as in Figure 7c. The code to restore the state of the stack to that of the caller is the same as given above. No special consideration is needed to deal with parameters passed this way.

Within the procedure, access to the parameters is accomplished by copying the values from their location within the activation record to a register. For this same example, the first parameter would be copied to a register using the MAL instruction

```
        lw $8, 0($16)
```

where register $8 is the register designated to hold the parameter during the execution of the procedure.

An alternative method for passing parameters is to pass values directly in registers. It can be more efficient than using the stack, because there may be fewer memory accesses. Instructions are not spent pushing parameters on the stack and then copying the parameters back into registers during the procedure's execution. Like return addresses, procedure parameters are dynamic data, so they can only be passed in registers to procedures that do not themselves use those same registers for parameter passing to a nested procedure call. In general, passing parameters in registers alone will not work for programs that contain nested procedure calls.

The following code uses the same example of procedure B, modified only in that the two parameters are passed in registers $4 and $5 instead of within the activation record. Space for the parameters is still allocated within the activation record for reasons explained later within this chapter.

```
        sub $sp, $sp, 8   # allocate space for 2 parameters
        add $4, $18, 1    # first parameter placed into register $4
        move $5, $21      # second parameter placed into register $5
        jal B
        .
        .
        .
B:      sub $sp, $sp, 16  # allocate B's activation record
        sw $ra, 12($sp)   # save B's return address
        sw $16, 16($sp)   # save caller's frame pointer
        add $16, $sp, 24  # set frame pointer to B's activation record

        # procedure B's body here

        lw $ra, -12($16)  # restore B's return address
        move $sp, $16     # deallocate B's activation record
        lw $16, -8($16)   # restore caller's frame pointer
        jr $ra
```

Returning a result from a function is similar in many ways to passing a parameter. The difference is that the information is being passed in the reverse direction -- from the called procedure to the calling program. In addition, the time during which the value must be preserved is very short. The value must be preserved only over the interval from when the result is generated until control is returned to the calling program. The system stack could be used for this purpose. The function return value could be placed at the top of the stack just before the function returns. Then, the calling program retrieves the value from the top of the stack. It would also be simple and expedient to return the value within a register. This is common practice.

Programming languages give restrictions on the use of parameters. As an example, Pascal requires parameters to be passed either by value or by reference. Parameters of type `var` are passed by reference. This means that the value of the parameter may be changed within the procedure, and the changed value will be reflected outside the bounds of the procedure. A value type parameter in Pascal will not have its value changed by a procedure. C allows only value type parameters. A simple way of enforcing these rules on assembly language programming is to pass either addresses or copies of values as parameters. By passing an address, a procedure can access the variable, thereby changing its value if necessary. If a copy of a variable is passed as a parameter, the caller's variable is never touched, and will not be changed. In this case, the copy can be modified within the procedure, but will not change the value of the original variable.

9.5 Saving Registers

A procedure often needs many registers for local variables, copies of parameters, and temporary calculations. In a load/store architecture, the procedure needs registers in order to access values stored on the stack. A newly invoked procedure therefore must immediately have some registers made available to it. The state of the registers is part of the current environment, however, so the values in the registers set by the calling program should be left intact. Therefore any register that is part of the environment of the calling program cannot be used by a newly called procedure unless its value can first be preserved and then be restored upon return to the calling program. Once again, use of activation records provides a convenient solution. Register values can be saved by the called procedure within its activation record. Then, the registers can be reused by the procedure. Before returning, the registers are restored to their original values. The activation record is therefore defined to include space for any registers that might need to be saved. This method of clearing out registers for the procedure's use is called **callee save**. It is said that register values are preserved across procedure calls, because the register values of the calling program are not changed by a called procedure.

An alternative to callee saved registers is **caller saved** register values. These register values are said to *not* be preserved across procedure calls. Before a procedure is called, any registers whose values should not be modified by the procedure to be called are first saved in the caller's activation record. After the procedure returns, the register values can be restored by loading them from the activation record.

Both caller saved and callee saved schemes are used in practice. It is up to the programmer to choose one, although many systems adopt a fixed convention to improve interoperability of procedures written in different environments. The calling program and the called procedure must have the same understanding about what registers are being saved, and when.

Note for advanced readers:

The VAX architecture had a single procedure call instruction that set up a complete activation record on the stack.The call instruction pushes a return address, parameters, and register variables onto the stack, and then branches to the first instruction within the procedure. The programmer has control over which registers to save on the stack (as part of the procedure call) by providing a mask. Often only a subset of the registers need to be saved on the stack. One bit of the mask corresponds to each register in the machine. The procedure call instruction checks each bit in the mask. If the bit is set, then its value is saved on the stack. The instruction that implements a procedure return is similarly complex. It must restore registers to their previous values, deal with parameters, and use the return value (from the stack) to return from the procedure.

9.6 MIPS RISC Procedure Conventions

A set of conventions is encouraged for use of registers in the MIPS RISC architecture. A register is not dedicated for use as a frame pointer in the architecture. Therefore, access to data within the activation record is accomplished using offsets from the stack pointer ($sp or $29). A compiler could implement a frame pointer by designating one of the registers to be a frame pointer, and then using it for that purpose.The programmer or compiler is free to choose whether a frame pointer is implemented or not.

To increase efficiency, the MIPS RISC architecture does not specify all registers as callee saved or all registers as caller saved. Rather, some registers are preserved across procedure calls, others are not. Those that are preserved across procedure calls are the callee saved registers. They are called **saved registers**, and are designated as $s0-$s8 or $16-$23, $30.

Those that are not preserved across procedure calls are **temporary registers**, and are designated as $t0-t9 or $8-$15, $24-$25. These are caller saved registers. When allocating a register, a choice must be made. If a procedure does not contain any nested calls, it prefers to use the temporary registers. The body of a procedure can use the temporary registers without having to preserve their previous values. If the procedure does contain procedure calls, there is a choice between using a temporary register or a saved register. There are many strategies for optimizing the use of registers. The goal behind having some of each kind of register is to reduce the number of stack (memory) accesses.

Parameter passing on the MIPS RISC architecture uses a combination of registers and space in the activation record. The first four (non-floating point) parameters are passed in registers $a0-$a3, or $4-$7. Additional parameters are passed within the activation record. Studies determined that a majority of procedures receive four or fewer parameters. Therefore, the possibility exists for reducing the number of stack (memory) accesses to store and retrieve parameters if register are designated as a place to pass parameters. Note that if a procedure is called with parameters passed in registers $a0 and $a1, the called procedure must be careful to save these two registers before it invokes another (nested) procedure. For this reason, space for the parameters is always allocated within the activation record of the (non-leaf) called procedure. It may not be used, but is made available for those cases when it will be needed.

Figure 8 specifies the conventions for register usage for the MIPS RISC architecture. Regis-

Register Name	Alias	Use
$0		the value 0
$1	$at	reserved by the assembler
$2-$3	$v0-$v1	expression evaluation and function results
$4-$7	$a0-$a3	the first four parameters -- *not* preserved across procedure calls
$8-$15	$t0-$t7	temporaries-- *not* preserved across procedure calls
$16-$23	$s0-$s7	saved values-- preserved across procedure calls
$24-$25	$t8-$t9	temporaries-- *not* preserved across procedure calls
$26-$27	$k0-$k1	reserved for use by the operating system
$28	$gp	global pointer
$29	$sp	stack pointer
$30	$s8	saved values-- preserved across procedure calls
$31	$ra	return address
$f0-$f2		floating point function results
$f4-$f10		temporaries-- *not* preserved across procedure calls
$f12-$f14		the first two floating point parameters-- not preserved across procedure calls
$f16-$f18		temporaries-- *not* preserved across procedure calls
$f20-$f30		saved values-- preserved across procedure calls

Figure 8: MIPS RISC register usage conventions.

ters $v0 and $v1, or $2 and $3 are used for function return values. Register $1 is used by the assembler when an extra register is needed for temporary calculations. Floating point registers $f12 and $f14 may also be used to pass some of the first four parameters.

The MAL input and output instructions are implemented similar to procedures. They take parameters in registers $a0-$a3, and they return values in registers $v0 and $v1.

9.7 A MAL Program that uses Procedures

The following MAL code contains a modular program that calculates the greatest common divisor of two positive integers. The user is prompted for two integers. The program then calculates the greatest common divisor, and prints out the result. This MAL program follows the MIPS RISC register usage conventions. It uses a recursive function to calculate the greatest common divisor.

```
# This MAL program computes the greatest common divisor of 2 positive
#   integers.  The function that does the main calculation is recursive.

.data
err_msg:  .asciiz  “\nbad integer entered\n”
.text
__start:   jal  intro
           jal  getint
           beqz $v0, input_err
           move $s0, $v1
           jal  getint
           beqz $v0, input_err
           move $a1, $v1
           move $a0, $s0
           jal  gcd
           move $a0, $v0
            jal  print_result
           done
input_err:
           la   $t0, err_msg
           puts $t0
           done


# intro
# A simple procedure to print a little introduction message.
# There are no parameters.
.data
msg1:  .asciiz  “This program computes the greatest common divisor of\n”
msg2:  .asciiz  “two user entered integers.\n\n”
.text
intro:     la   $t0, msg1
           puts $t0
           la   $t0, msg2
           puts $t0
           jr $ra
```

```
#getint
# A function to get a single positive integer from the user.
# return values:
#  $v0 -- flag indicating success of the function
#         value is 1 if integer entered is ok.
#         value is 0 if not ok.
#  $v1 -- the integer entered
#
# register assignments:
#  $t0 --  user entered character
#  $t1 --  digit of integer
#  $t2 --  newline character (the constant)
#  $t3 --  the integer being entered
#  $t4 --  the constant 10, the base
#  $t5 --  temp
#  $t6 --  the constant 9
.data
prompt:  .asciiz  “Enter positive integer:  “
.text
getint:
           li   $t2, 10   # $t2 <- newline character
           li   $t4, 10   # constant, 10
           li   $t3, 0
           li   $t6, 9
           la   $t5, prompt
           puts $t5
getchar:
           getc $t0
           beq  $t0, $t2, getint_rtn
           sub  $t1, $t0, 48
           bgt  $t1, $t6, int_err
           bltz $t1, int_err
           mul  $t3, $t3, $t4
           add  $t3, $t3, $t1
           b    getchar

getint_rtn:
           li   $v0, 1
           move $v1, $t3
           jr $ra

int_err:   li  $v0, 0
           jr $ra
```

```
# gcd
# A recursive function to calculate the greatest common divisor.
#
#    gcd(m,n) =   m,                if n = 0
#                 gcd(n, m mod n),  if n > 0
# return values:
#  $v0 -- the greatest common divisor
# parameters:
#  $a0 -- m
#  $a1 -- n
.data
gcd_err_msg:  .asciiz “error in calculating gcd -- quitting.\n”
.text
gcd:
           sub  $sp, $sp, 12    # allocate activation record
           sw   $ra, 12($sp)    # save return address
           bgtz $a1, n_greater
           bnez $a1, gcd_err
           move $v0, $a0        # return m
           b    gcd_rtn
n_greater: sw   $a0, 8($sp)     # save current parameters
           sw   $a1, 4($sp)
           rem  $t0, $a0, $a1
           move $a0, $a1        # set up parameters for call
           move $a1, $t0
           jal  gcd             # recursive call to gcd
           lw   $a0, 8($sp)     # restore current parameters
           lw   $a1, 4($sp)
                  # return value already in $v0
gcd_rtn:
           lw   $ra, 12($sp)    # restore return address
            add  $sp, $sp, 12    # remove activation record
           jr   $ra             # return
gcd_err:   la   $t0, gcd_err_msg
           puts $t0
           done


# print_result
# Procedure to print out the base ten integer passed to the procedure
# as a parameter.
#
# return values:
#  none
# parameters:
#  $a0 -- the integer to be printed out
#
# register assignments:
#  $t0 -- address of output string
#  $t1 -- stack pointer before pushing characters onto stack
#  $t2 -- copy of integer to be printed
#  $t3 -- ASCII character code of digit to be printed as pushed
#  $t4 -- ASCII character code of digit to be printed as popped
#  $t5 -- newline character for nice output
```

```
.data
result_msg:  .asciiz  “The greatest common divisor is “
.text
print_result:
           move $t2, $a0
           la   $t0, result_msg
           puts $t0
           move $t1, $sp
           bnez  $t2, push_loop
           li   $t3, 0           # special case for printing 0
           sw   $t3, 0($sp)
           sub  $sp, $sp, 4
           b    print_loop

push_loop:                       # push characters onto stack
           rem  $t3, $t2, 10
           add  $t3, $t3, 48
           sw   $t3, 0($sp)
           sub  $sp, $sp, 4
           div  $t2, $t2, 10
           bnez $t2, push_loop

print_loop:                      # pop characters off stack and print
           add  $sp, $sp, 4
           lw   $t4, 0($sp)
           putc $t4
           bne  $t1, $sp, print_loop

           li   $t5, 10   # print newline
           putc $t5
           jr   $ra
```

9.8 Summary

Procedures have both advantages and disadvantages when it comes to assembly language programming. The disadvantage is that much extra code is needed to deal with parameter passing, register saving and restoring, and stack accesses. Yet procedures are also an advantage. They facilitate modular code, and they are easy to use in high-level languages. Fortunately, compilers provide the benefit of generating detailed assembly code automatically.

PROBLEMS

1. What is the point of having procedures in a high-level language? Is there a point to having a procedure mechanism in assembly language?
2. When is the use of a stack necessary in implementing procedure call and return mechanisms?
3. Write a MAL code fragment that implements the same operations as the `jal` instruction.
4. Write MAL code for a procedure called `switch` that takes two (integer) parameters and swaps them. Write MAL code that implements a procedure call and the procedure. Pass the parameters in registers.
5. Write MAL code for a procedure called `switch` that takes two (integer) parameters and switches them. Write MAL code that implements a procedure call and the procedure. Pass the parameters on the stack.
6. Write MAL code for a procedure called `switch` that takes two (integer) parameters and switches them. Write MAL code that implements a procedure call and the procedure. Use MIPS RISC procedure conventions.
7. One method of parameter passing might be to permanently assign all variables to their own registers. Then there won't be any shuffling of data into and out of registers before and after the procedure call. When will this method fail?
8. Write a MAL procedure that implements the factorial function recursively. Factorial is recursively defined by

```
factorial(0) = 1
factorial(x) = x * factorial(x-1)
```

9. Can Pascal `var` type parameters be passed in registers? Can Pascal value type parameters be passed in registers? Why or why not?
10. Write a modular MAL program to calculate the integer X, where

$$X = \sum_{i=1}^{n} i$$

The user enters an integer value for n.

11. Write a modular MAL program that prints out a user-entered integer in base 3, base 5, and base 8 (octal).