The MIPS Register Usage Conventions

NOTE: We will *NOT* follow the MIPS conventions for register usage this Fall 2007 semester! They are fairly complex, so use the simple conventions described in the material on implementing functions. These conventions are included for the advanced student who wishes to know what the MIPS architecture does.

Parameter Passing: The MIPS Way

MIPS convention -- when passing parameters in registers, the first 4 parameters are passed in registers $4-7. The aliases for $4-$7 are $a0-$a3. The first parameter to a procedure is always passed in $a0.

Then, *any* and *all* procedures use those registers for their parameters.

ALSO MIPS convention -- space for all parameters (passed in $a0-a3) is allocated in the parent's (caller's) AR !!

If there are nested calls, and registers $a0-a3 are used for parameters, the values would be lost (just like the return address would be lost for `jal` if not saved).

An example of this problem:

```
  procA:  # receives 3 parameters in $a0, $a1, and $a2

        # set up procB's parameters
        move $a0, $24  # overwrites procA's parameter in $a0
        move $a1, $9   # overwrites procA's parameter in $a1
        jal  procB     # the nested procedure call

        # procA continues after procB returns
        # procA's parameters are needed, but have been overwritten
```

Here is a solution.

current parameters are stored on the stack (by the parent) before a nested call. After the return from the nested call, the current parameters are restored.

The example re-written, to do things the MIPS way. Note that this is only a code fragment. It does not show everything (like saving $ra).

```
 procA:  # receives 3 parameters is in $a0, $a1, and $a2.
         # The caller of procA has allocated space for $a0-$a3
         # at the top of the stack.

         # assume that procA has an activation record of 5 words.
         sub  $sp, $sp, 20   # allocate space for AR

         # save procA's parameters
         sw   $a0, 24($sp)
         sw   $a1, 28($sp)
         sw   $a2, 32($sp)

         .
         .
         .

         # set up procB's parameters
         move $a0, $24
         move $a1, $9
         jal  procB     # the nested procedure call
```

```
        .
        .
        .
        # procA continues after procB returns
        # procA's parameters are needed, so restore them
        lw   $a0, 24($sp)
        lw   $a1, 28($sp)
```

In this code fragment, procA saves its 3rd parameter (from $a2) on the stack. This is necessary (even though procB only receives 2 parameters). `procB` may call `procC`, passing more than 2 parameters to `procC`.

Here is a general layout of how this second option is used on MIPS, following conventions (with 4 or fewer parameters):

```
    proc1 layout:
        allocate AR (include space for outgoing parameters)
        put return address on stack into AR of procedure

        procedure calculations

        to set up and call proc2,
           place current parameters (from $a0-a3) into previously allocated
               space
           set up parameters to proc2 in $a0-a3
           call proc2 (jal proc2)
           copy any return values out of $v0-v1, $a0-a3
           restore current parameters back to $a0-a3

        more procedure calculations (presumably using procedure's
          parameters which are now back in $a0-a3)

        get procedure's return address from AR
        deallocate AR
        return (jr $ra)
```

Frame Pointers

The stack gets used for more than just pushing/popping stack frames. During the execution of a procedure, there may be a need for temporary storage of variables. The common example of this is in expression evaluation.

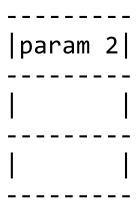
```
Example:      high level language statement
                Z = (X * Y) + (A/2) - 100
```

The intermediate values of `X*Y` and `A/2` must be stored somewhere. On older machines, register space was at a premium. There just were not enough registers to be used for this sort of thing. So, intermediate results (local variables) were stored on the stack.

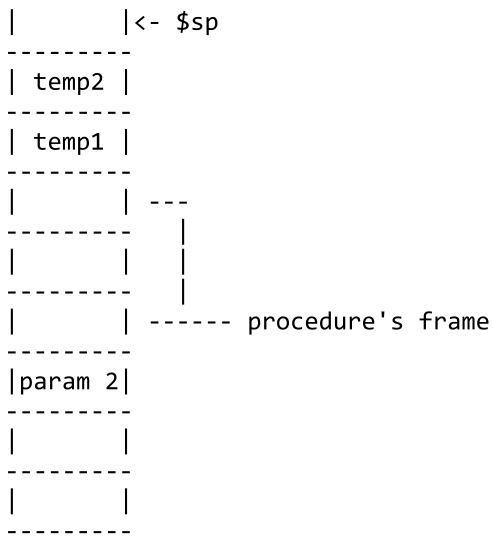
They do not go in the stack frame of the executing procedure; they are pushed/popped onto the stack as needed.

So, at one point in a procedure, parameter 2 might be at 16($sp)

```
  |       |
  ---------
  |       |<- $sp
  ---------
  |       | ---
  ---------   |
  |       |   |
  ---------   |
  |       | ------ procedure's frame
```

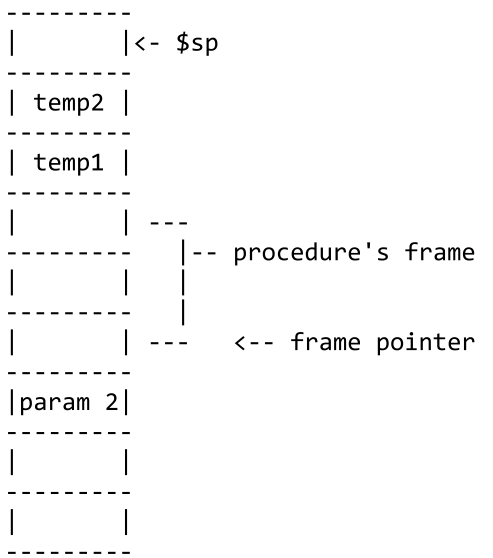


and, at another point within the same procedure, parameter 2 might be at 24($sp)



All this is motivation for keeping an extra pointer around that does not move with respect to the current stack frame.

Call it a **frame pointer**. Make it point to the base of the current frame:



Now items within the frame can be accessed with offsets from the frame pointer, *and* the offsets do not change within the procedure.

parameter 2 will be at 4(frame pointer)

A new register is needed for this frame pointer. Pick one. (The chapter arbitrarily chooses $16, but it could be any register.)

parameter 2 is at 4($16)

NOTES:

-- The frame pointer must be initialized at the start of every procedure, and restored at the end of every procedure.

-- The MIPS architecture does not really allocate a register for a frame pointer. It has something else that it calls a "virtual frame pointer," but it is not really the same as described here. On the MIPS, all data with a stack frame is accessed via the stack pointer, $sp.

The skeleton of a procedure that uses a frame pointer and has parameters:

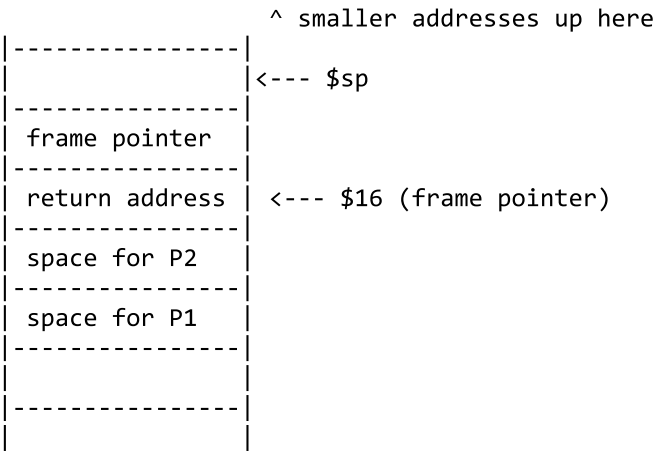
```
# the frame (AR) is 4 words, 2 words are space for 2 parameters
# passed in $a0 and $a1, 1 is for return address, and 1 is for
# the frame pointer

procedure:
  sub  $sp, $sp, 8   # allocate remainder of frame
                     # (assumes that caller allocated space for the
                     # 2 parameters)
  sw   $ra, 8($sp)   # save procedure's return address
  sw   $16, 4($sp)   # save caller's frame pointer
  add  $16, $sp, 8  # set procedure's frame pointer


  # procedure's code in here
  # Note that all accesses to procedure's AR is done with offsets from $16

  lw   $ra, ($16)    # restore return address
  move $8, $16       # save frame pointer temporarily
  lw   $16, -4($16)  # restore callers frame pointer
  move $sp, $8       # remove procedure's frame (AR)
  jr   $ra
```

The activation record (frame) for this procedure after everything is in it:



Caller vs. Callee saved register values

Two types:

1. CALLEE SAVED

- a procedure clears out some registers for its own use
- register values are preserved across procedure calls
- MIPS calls these **saved** registers, and designates $s0-s8 for this useage.
- $s0-$s8 are aliases for $16-$23, $30
- the called procedure saves register values in its AR, uses the registers for local variables, restores register values before it returns.

2. CALLER SAVED
   - the calling program saves the registers that it does not want a called procedure to overwrite
   - register values are NOT preserved across procedure calls
   - MIPS calls these **temporary** registers, and designates $t0-t9 for this useage.
   - $t0-$t9 are aliases for $8-15, $24-$25
   - procedures use these registers for local variables, because the values do not need to be preserved outside the scope of the procedure.

What the mechanisms should look like from the compiler's point of view:

```
THE CODE:

   call setup
   procedure call
   return cleanup
   .
   .
   .
procedure:  prologue

            calculations

            epilogue
```

- CALL SETUP
  - place current parameters into stack (space already allocated by caller of this procedure)
  - save any TEMPORARY registers that need to be preserved across the procedure call
  - place first 4 parameters to procedure into $a0-$a3
  - place remainder of parameters to procedure into allocated space within the stack frame
- PROLOGUE
  - allocate space for stack frame
  - save return address in stack frame
  - copy needed parameters from stack frame into registers
  - save any needed SAVED registers into current stack frame
- EPILOGUE
  - restore (copy) return address from stack frame into $ra
  - restore from stack frame any saved registers (saved in prologue)
  - de-allocate stack frame (move $sp so the space for the procedure's frame is gone)
- RETURN CLEANUP
  - copy needed return values and parameters from $v0-v1, $a0-a3, or stack frame to correct places
  - restore any temporary registers from stack frame (saved in call setup)

An excellent and detailed example -- written by Prof. David Wood

```
# procedure: procA
# function: demonstrate CS354 calling convention
# input parameters: $a0 and $a1
# output (return value): $v0
# saved registers: $s0, $s1
# temporary registers: $t0, $t1
```

```
# local variables: 5 integers named R, S, T, U, V
# procA calls procB with 5 parameters (R, S, T, U, V).
#
# Stack frame layout:
#
#       | in $a1  |  68($sp)
#       | in $a0  |  64($sp)
#       |---------|
#       |    V    |  60($sp)       --|
#       |    U    |  56($sp)         |
#       |    T    |  52($sp)         |
#       |    S    |  48($sp)         |
#       |    R    |  44($sp)         |
#       |   $t1   |  40($sp)         |
#       |   $t0   |  36($sp)         | --  A's activation record
#       |   $ra   |  32($sp)         |
#       |   $s1   |  28($sp)         |
#       |   $s0   |  24($sp)         |
#       | out arg4|  20($sp)         |
#       | out $a3 |  16($sp)         |
#       | out $a2 |  12($sp)         |
#       | out $a1 |   8($sp)         |
#       | out $a0 |   4($sp)       --|
#       |---------|
#       |         | <-- $sp        ---
#       |         |                  |
#       |         |                  |   --  where B's activation record
#       |         |                  |        will be
procA:
        # procedure prologue
        sub $sp, $sp, 60        #allocate activation record, includes
                                # space for maximum outgoing args
        sw $ra, 32($sp)         #save return address
        sw $s0, 24($sp)         # save 'saved' registers to stack
        sw $s1, 28($sp)         # save 'saved' registers to stack
        # end prologue

        ....    # more code

        # call setup for call to procB
        # save current (live) parameters into the space specifically
        # allocated for this purpose within caller's stack frame
        sw $a0, 64($sp)         # only needed if values are 'live'
        sw $a1, 68($sp)         # only need if values are 'live'
        # save any registers that need to be preserved across the call
        sw $t0, 36($sp)         # only need if values are 'live'
        sw $t1, 40($sp)         # only need if values are 'live'
        # put parameters into proper location
        lw $a0, 44($sp)         # load R into $a0
        lw $a1, 48($sp)         # load S into $a1
        lw $a2, 52($sp)         # load T into $a2
        lw $a3, 56($sp)         # load U into $a3
        lw $t0, 60($sp)         # load V into a temp register
        sw $t0, 20($sp)         # outgoing arg4 must go on the stack
        #end call setup

        # procedure call
        jal procB

        # return cleanup for call to procB
        # restore saved registers
        lw $a0, 64($sp)
        lw $a1, 68($sp)
        lw $t0, 36($sp)
        lw $t1, 40($sp)
```

```
        # return values are in $v0 and $v1

        ....    # more code

        # procedure epilogue
        # restore return address
        lw $ra, 32($sp)
        # restore $s registers saved in prologue
        lw $s0, 24($sp)
        lw $s1, 28($sp)
        # put return values in $v0 and $v1
        mov $v0, $t0
        # deallocate stack frame
        add $sp, $sp, 60
        # return
        jr $ra

# end of procA
```

An important detail for 354 students writing MAL code and using the simulator:

The I/O instructions putc, puts, and getc are implemented as functions within the operating system. They are not actual instructions on a MIPS R2000 processor. (In general, NO modern architecture has explicit input/output instructions.)

Parameters get passed to the operating system, and return values get set by the functions implementing `putc`, `puts`, and `getc`. Therefore, in general, you must assume that $a0-$a3 and $v0-$v1 will be overwritten during the execution of any `putc`, `puts`, or `getc` instruction.

In practice, I believe the simulator is implemented to only change values in $a0 and $v0.

On choosing $s versus $t registers

Since either method (**caller saved** or **callee saved**) potentially wastes time saving/restoring values that may not be overwritten, the MIPS solution utilizes some registers in its register file to be used as **caller saved**, and others are used as **callee saved**.

- $8-$15, $24-$25 are **caller saved**
- aliases are $t0 - $t9
- It is presumed that any child can overwrite these registers in any way it wants to. The 't' in $t stands for temporary.

- $16-$23, $30 are **callee saved**
- aliases are $s0 - $s8
- A parent (caller) presumes that no child (callee) will modify values in these registers. But, any child (callee) that uses one of these registers MUST save/restore it value for its parent (caller).

This leaves the question for the programmer writing code: which register should be chosen for a variable, $s or $t?

Here is my advice:

- In a leaf procedure (which is a child, never a parent), use $t registers for everything possible. The $t registers do *not* need to be saved/restored by a child.
- In a non-leaf procedure (which is both a parent and a child!) the choice between $s and $t registers is not as clear. For a variable that is live across a call, it is often best to use a $s register. For those variables that

are not live, it is often best to use a $t register. Note that the choice of a $s register implies that there *will* be a save/restore pair.

The choice for using a $s or $t register may still not be obvious. Consider the following abstract code choices.

```
within A:
     (showing correct usage of the $t register)

   loop:  use  $t
          sw   $t, __($sp)  # save $t because it is live across a call
          jal  B
          lw   $t, __($sp)  # restore the $t
          use  $t
          b    loop

          jr   $ra
```

Alternatively, here is this same example, with the value in an $s register, instead of in a $t register.

```
within A:
     (showing correct usage of the $s register)
          sw   $s, __($sp)  # save $s once by convention

   loop:  use  $s
          jal  B            # B is guaranteed (by convention) not to change $s
          use  $s
          b    loop

          lw   $s, __($sp)  # restore the $s
          jr   $ra
```

In this case, using an $s register is more efficient. The save restore pairs using the $t register are inside the loop, so there are memory access instructions (2 of them) for every iteration of the loop. Using the $s register still requires a save/restore pair, but it happens just once.