Laboratory 3 Notes
Conditional Execution and Procedures

MIPS Branching Instructions

beq *$rs, $rt, label* #if $rs = $rt, $pc = *label* (which means the next instruction is at the address specified by *label*)

bne *$rs,$rt, label* #if $rs != $rt, $pc = *label*

MIPS Branching Pseudo-Instructions

blt *$rs, $rt, label* #branch if $rs < $rt

bgt *$rs, $rt, label* #branch if $rs > $rt

ble *$rs, $rt, label* #branch if $rs <= $rt

bge *$rs, $rt, label* #branch if $rs >= $rt

MIPS Unconditional Branch (Jump)

j label #$pc = label (jump to address specified by *label*)

Conditional Exectution/If Statements

If $t0 is '0',
    perform *task1* and continue with the *next* section of code.
else
    perform *task2* and continue with the *next* section of code.

In **Java**:

```
        if ($t0 == 0) {
            /* code to perform task 1 */
        }
        else {
            /* code to perform task 2 */
        }
next:       /* code for the next section of the program */
```

In **MIPS**:

```
            bne  $t0,$0, task2

            # code to perform task1

            j  next

task2:      # code to perform task1

next:       #code for the next section of the program
```

Conditional Execution/Loops

$t0 contains a loop counter; repeat the code in the loop as long as $t0 is not '0'

In Java:

```
          for  ($t0 = initial_count; t0>0; t = t-1)  {
               /* code for body of loop */
          }
loopdone:     /* code following the loop */
```

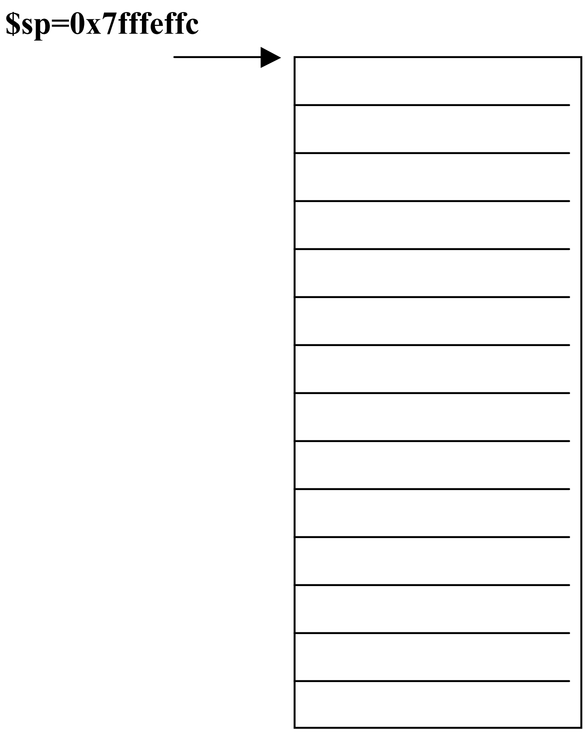
In MIPS

```
     loop:     beq  $t0,$0, loopdone     #test loop counter, exit
                                         loop if counter = 0

               #code for body  of the loop

               addi $t0, $t0,-1          #decrement loop counter
               j      loop               #repeat loop

     loopdone:                           #code following the loop
```

Stack

simply a section of main memory reserved for stack operations

Initial state of the stack

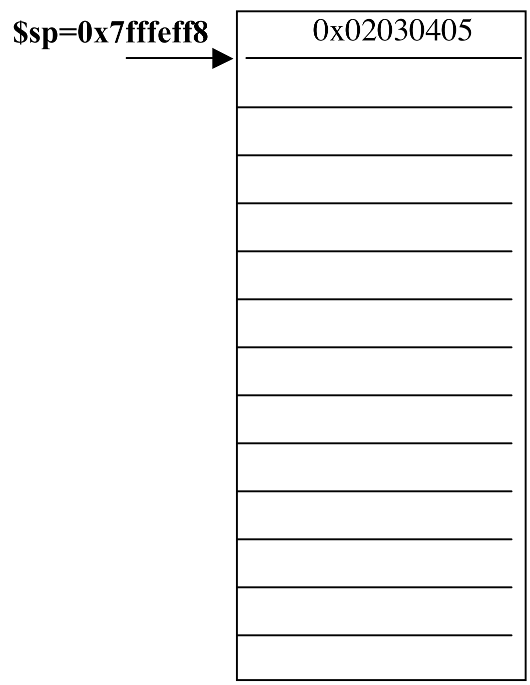


Push a word-size value in $t0 on the stack.

Assume $t0 = 0x02030405

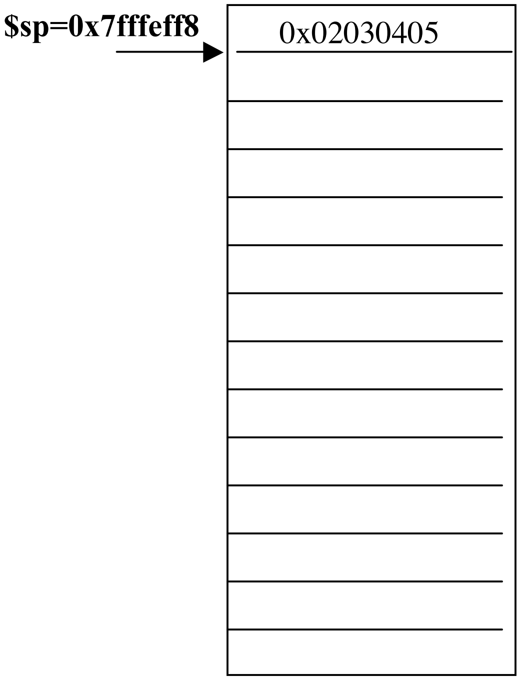
addi $sp,$sp,-4 #$sp = $sp – size in bytes of value (make room on the stack)

sw $t0,0($sp) #store the value on the stack



<table>
<tr>
<td>

Initial State of Stack

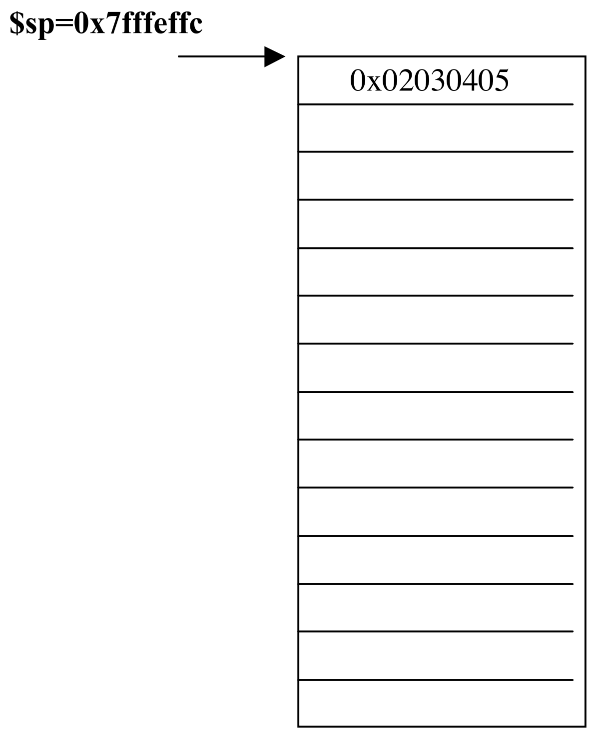


</td>
<td>

Pop a word-size value from the stack.

lw $t1,0($sp) #read the word from the stack
$t1 = 0x020304 as a result

addi $sp,$sp,4 #$sp = $sp + size in bytes of
value (deallocate stack)



</td>
</tr>
</table>

Instructions used for Procedures

jal (jump-and-link)

jal *procedure* #puts the address of the instruction following the procedure call in $ra, and puts the starting address of the procedure in $pc:

$ra ← $pc + 4
$pc ← address of procedure

jr (jump register)

jr *$ra* # return address from $ra goes into $pc #NOTE: this should always by the last statement executed in a procedure!

$pc ← $ra ($ra contains return address)

Program resumes execution of the instruction in the main program which follows the **jal** instruction (the procedure call in the main program).

MIPS registers use for procedures

$a0-$a3 argument registers to pass parameters

$v0-$v1 value registers to return values

$ra return address register to return to calling program

Stack use for Procedures

If more information or parameters than will fit in these registers is needed to call or return from a procedure, the **stack** can be used to pass the extra parameters.

Also, if the procedure modifies registers whose values are needed by the main or invoking program, the **stack** can be used to save the original values of the registers (which can then be restored before the return to the calling program).

The compiler convention is that if registers $s0 - $s7 are modified by a procedure, the procedure should save the original value of the registers on the stack, and restore them before the return from the procedure.

Storing/restoring save registers on stack for procedure calls

```
main:    la   $a0,param0     #store parameters in $a0 - $a3
         la   $a1,param1
         la   $a2,param2
         la   $a3,param3
         jal myproc           #call procedure
         …
         #rest of the main program instructions

#procedure definition
myproc:  addi $sp,$sp,-32          #allocate 8 words on stack
         sw   $s0,0($sp)     #store 8 registers on stack
         sw   $s1,4($sp)
         sw   $s2,8($sp)
         …
         sw   $s6,24($sp)
         sw   $s7,28($sp)
```

```
…
#instructions which use parameters and registers for some purpose
…

lw   $s7,28($sp)      #restore 8 registers from stack
lw   $s6,24($sp)
lw   $s5,20($sp)
…
lw   $s1,4($sp)
lw   $s0,0($sp)
addi $sp,$sp,32       #deallocate the stack

jr   $ra              #return from procedure to main program
```

Recursive and nested procedures

For recursive and nested procedures, there may be conflicts over the use of the shared registers, so the convention is more restrictive.

The calling program must also save $ra on the stack, along with $a0 - $a3 and $t0 - $t9 if their values need to be preserved across a procedure call, in addition to any of the $s0 - $s7 registers modified by the procedure.

In lab today, you will see a recursive program, which reads and sums inputs until a value of ‘0’ is entered.

Conventions for drawing stack diagrams

To record the contents of the stack to understand how the stack is used, using the following notation:

- Assume the diagram below represents the stack before execution of the main program. Each row in the stack represents a word. The initial **$sp** with a subscript of **0** is pointing to the top of the stack.

- Trace the effect on the stack of executing each instruction in the program by moving the position of the **$sp** when it changes, (incrementing the subscript for each new value), and by recording new values on the stack as they are stored there.

- When the stack starts to empty, continue with the same notation, except use the right hand side of the stack diagram to indicate the changes.

- Also record changes to relevant registers. Start a new row each time a register changes value. Only update the register with the new value in each row (don't re-write values that have not changed).

```
main:      jal getAndSumValues
           move $t0,$v0

        …

getAndSumValues:
        addi $sp,$sp,-8
        sw   $ra,4($sp)

        li $v0,4
        la $a0, prompt
        syscall

        li $v0,5
        syscall
        sw $v0,0($sp)

        bne $v0,$zero,recurse
        addi $sp,$sp,8
        jr $ra

recurse:
        jal getAndSumValues
        lw $t0,0($sp)
        lw $ra,4($sp)
        addi $sp,$sp,8

        add $v0,$t0,$v0
        jr $ra
```

$v0	**$t0**