Assembly Language Programming

Example programs and program segments illustrate the use of the MIPS instructions and the assembler conventions.

Programming a(b+c)

- Assume a, b and c are declared variables and that the result is saved in $v0

```
lw    $t0,a       # Get value of a
lw    $t1,b       # Get value of b
lw    $t2,c       # Get value of c
add   $t1,$t1,$t2 # Add b and c
mult  $v0,$t0,$t1 # Multiply result times a
```

This 3-operand multiply pseudoinstruction might be generated as ...

```
mult  $t0,$t1     # Do multiply
mflo  $v0         # Get result assuming < 2x10^9
```

How can one test to see if the number was small enough?

Make a(b+c) Into A Procedure

- This distributive law procedure will receive a, b and c via the argument registers
- No other procedures are called, so nothing has to be saved

```
Dist: # A procedure to compute $a0($a1+$a2)
      add   $t1,$a1,$a2 # Add b and c
      mult  $v0,$a0,$t1 # Multiply result times a
      jr    $ra         # Return to caller
```

The procedure **Dist** is called by ...

```
      jal   Dist
```

> Procedures that do not call other procedures are sometimes called “leaf procedures”

Compute N factorial

- N! = N * (N-1)*(N-2)*...*2*1; 0! = 1 and 1!=1
- Return result in $v0

```
      addi  $v0,$0,1    # Initialize
      beq   $a0,$0,Done # 0! = 1
      add   $s0,$a0,$0  # Move argument
Loop: addi  $s1,$s0,-1  # Reduce arg and move
      beq   $s1,$0,Done # Exit if we're finished
      mult  $v0,$v0,$s0 # Multiply next term
      addi  $s0,$s0,-1  # Find the next term
      j     Loop        # Continue until done
Done:
```

Compute 3!

```
        addi    $v0,$0,1        # Initialize
        beq     $a0,$0,Done     # 0! = 1
        add     $s0,$a0,$0      # Move argument
Loop:   addi    $s1,$s0,-1      # Reduce arg and move
        beq     $s1,$0,Done     # Exit if we're finished
        mult    $v0,$v0,$s0     # Multiply next term
        addi    $s0,$s0,-1      # Find the next term
        j       Loop            # Continue until done
Done:
```

$v0	$a0	$s0	$s1
–	3	–	–
1	3	–	–
1	3	3	–
1	3	3	2
3	3	3	2
3	3	2	2
3	3	2	1

$v0	$a0	$s0	$s1
6	3	2	1
6	3	1	1
6	3	1	0

Calling the Dist Procedure

- The factorial can be written as
  - {[N(N-1)](N-2)}(N-3) ...
  - Dist(Dist(Dist(N,N,-1),N,-2),N,-3)

```
      addi  $v0,$0,1    # Initialize
      beq   $a0,$0,Done # 0! = 1
      add   $s0,$a0,$0  # Move argument
      addi  $s1,$0,1    # Get 1 constant
      add   $a1,$s0,$0  # Move N
Loop: add   $a0,$v0,$0  # Move running product
      sub   $a2,$0,$s1  # Negate and move
      jal   Dist        # Go to subroutine
      addi  $s1,$s1,1   # Bump count
      bne   $s1,$a1,Loop# Continue until done
Done: add   $a0,$s0,$0  # Put arguement back
```

Compute 3!

```
        addi    $v0,$0,1        # Initialize
        beq     $a0,$0,Done     # 0! = 1
        add     $s0,$a0,$0      # Move argument
        addi    $s1,$0,1        # Get -1 constant
        add     $a1,$s0,$0      # Move N
Loop:   add     $a0,$v0,$0      # Move running product
        sub     $a2,$0,$s1      # Move in decrement
        jal     Dist            # Go to subroutine
        addi    $s1,$s1,1       # Bump count
        bne     $s1,$a1,Loop    # Continue until done
Done:
```

\$v0	\$a0	\$a1	\$a2	\$s0	\$s1
–	3	–	–	–	–
1	3	–	–	–	–
1	3	–	–	3	–
1	3	–	–	3	1
1	3	3	–	3	1
1	1	3	–1	3	1

\$v0	\$a0	\$a1	\$a2	\$s0	\$s1
6	1	3	–1	3	1
6	1	3	–1	3	2
6	6	3	–2	3	2
6	6	3	–2	3	2
6	6	3	–2	3	3

Saving Registers

- At the start of the procedure, save everthing that must be preserved ... at the end, put it back
- Since this factorial is not recursive ...

```
      Start of Procedure           End of Procedure
Fact: addi  $sp, $sp,-24           lw    $a0, 20($sp)
      sw    $a0, 20($sp)           lw    $a1, 16($sp)
      sw    $a1, 16($sp)           lw    $a2, 12($sp)
      sw    $a2, 12($sp)           lw    $ra, 8($sp)
      sw    $ra, 8($sp)            lw    $s0, 4($sp)
      sw    $s0, 4($sp)            lw    $s1, 0($sp)
      sw    $s1, 0($sp)            addi  $sp, $sp,24
                                   jr    $ra
```