Filling the Delay Slot

- The SPARC call and branch instructions are *delayed control transfer* instructions
  - A delayed control transfer instruction changes the Program Counter (recall that the PC contains the address of the next instruction to be executed) ***after*** the next instruction has already been fetched
    - This is an effect of pipelining, which we will study in a few weeks
  - The instruction after the delayed control transfer instruction is called the *delayed instruction*, and is said to *fill the delay slot*
  - Since the delayed instruction has already been fetched before the PC is changed, that means that any instruction following a call or branch instruction is ***always executed***:
    - ***before*** going to the called subroutine, or
    - ***before*** going to the target of the branch

Filling the Delay Slot (cont.)

- Filling the delay slot with a "nop" avoids confusion, but wastes an instruction:

```
call   _printf
nop
mov  1, %g1
```

- Since the delayed instruction is actually executed ***before*** the call or branch occurs, the delay slot can be filled more effectively by *moving another instruction into the delay slot*

```
call   _printf          ! call printf to print out
mov  1, %g1             ! exit request
```

  - Think carefully about which instruction to move, though!!
  - Do not put an instruction that sets the condition codes in the delay slot if the branch needs those condition codes
  - Do not put a branch, call, or set instruction in a delay slot

Finding Largest Integer in an Array (From Last Time)

```
        .data
arr:    .word   1,45,-16,23,38,17      ! int arr[6] = {...}
msg:    .ascii  "Value is %d\n\0"

        .text
        .global  _main                 ! main must be global
        .global  _printf               ! linker will find printf
_main:
        save %sp, -64, %sp             ! space to save registers

        mov  0, %l0                    ! %l0 (counter) = 0
        set  arr, %l1                  ! %l1 is base of arr
        mov  0, %l2                    ! %l2 (index) = 0
        ld   [%l2+%l1],%l3             ! %l3 (maxnum) = arr[0]

for:    cmp  %l0,6                     ! if (counter < 6) enter loop
        bge  end                       ! otherwise print answer
        nop
        ld   [%l1+%l2],%l4             ! %l4 (temp) = arr[index]
        cmp  %l4,%l3                   ! if (arr[index} > maxnum)
        ble  ok
        nop
        mov  %l4,%l3                   ! max num = arr[index]
ok:     inc  %l0                       ! counter++
        add  %l2,4,%l2                 ! index = index + 4
        ba   for
        nop
```


Assembling a SPARC Assembler Program

- We will use the C compiler "gcc" (which calls the SPARC assembler "as") to assemble our programs as follows:

  nimitz> gcc –g –o *file file.s*

  - where *file.s* is the source file
  - The "–g" switch includes debugging information for the "gdb" debugger

- When "gcc" is used to compile a C program:
  - It first translates the C program into assembly language, placing the code in a file named *file.s*
    - To see that file, type "gcc -S *file.c*"
  - It then calls "as" to assemble that file and produce a .o file

- If "gcc" is given a file with a ".s" extension, it calls "as" directly

Debugging a SPARC Assembler Program

- We will use the debugger "gdb" to debug our programs as follows:

  nimitz> gdb *file*

  - where *file* is the object file
  - Note — the program must have been assembled using the "–g" switch, which adds debugging information to the file

- Useful "gdb" commands:
  - b *func* sets breakpoint at beginning of function *func*
  - b **addr* sets breakpoint at *addr*
  - d *bnum* deletes breakpoint *bnum*
  - r run the program
  - c continue (after break)

Debugging a SPARC Assembler Program (cont.)

- Useful "gdb" commands (cont.):
  - x/w addr examine word at addr
  - x/i $pc examine contents of program counter as an instruction
  - p $*reg* print contents of register *reg*
  - display/i $pc continuous display of pc
  - display $*reg* continuous display of *reg*
  - undisplay *n* stop display of item *n* in display list
  - si step through next instruction, without going inside subroutines
  - ni step through next instruction or subrouine call

Homework #5 — Due 11/9/98 (Part 3)

3. Write a program in SPARC assembly language to determine if an integer is odd or even as follows:
   - Prompt the user to enter a positive integer, using printf. Use a loop to read in the integer using scanf, stopping when a zero or negative value is read in.
   - Determine whether the integer is odd or even, and print out that result, using printf
     - Hint: use "and" or "or" as appropriate to mask out all but the lsb, then branch based on the result
   - Use good programming style & comments.
   - This program must be your own work.
   - Email your program to "wmiao1@kent.edu" by 11am on 11/9/98.

This program counts 3/5 of this hw grade. (This is the last question on Homework #5)