

CS 322 Homework 2 – due 1:30 p.m., Wednesday, Feb. 2, 2005

The goal of this assignment is to re-acquaint you with assembly language programming in general and with the SPARC V8 architecture in particular. It is not directly connected with your PCAT compiler project, but should prove useful background for your code generator.

Write an assembly language subroutine `qs`, suitable for calling from C, that sorts an array of double precision numbers into ascending order using quicksort. The calling interface is given by

```
void qs(int n, double *a)
```

where `a` is the array to be sorted, and `n` is its size. The array should be sorted in place. Of course, you may choose to implement `qs` using additional, private subroutines if you wish.

For example, if your routine is linked with the following main program:

```
double a[] = {1.1,5.5,3.3,2.2,4.4};

main () {
    int i;
    qs(5,a);
    for (i = 0; i < 5; i ++)
        printf ("%f ", a[i]);
    printf ("\n");
}
```

it should produce the following output:

```
1.1 2.2 3.3 4.4 5.5
```

Of course, your code should also work when linked against more interesting main programs (like the private one we'll use as a test driver)!

In addition to the assembly code itself, you *must* turn in pseudo-code or C code corresponding to your assembly code, and you must comment *every* line of the assembly code indicating what it does in terms of the pseudo-code. For example, if the pseudo-code contained a line

```
x = t
```

the assembly code might contain the following commented lines:

```
ld [%fp-36],%i4      ! fetch t
st %i4,[%fp-40]     ! store into x
```

It is also useful to include a comment explaining where each pseudo-code variable is held (i.e., which register or stack frame slot).

Your code must be in a separate file `qs.s`; don't use `gcc`'s special mechanisms for including in-line assembly code. Your explanatory C code or psuedo-code should be included as a block comment in this file.

Strive for correctness, clarity, concision, and efficiency, in that order. For full credit, you must address *all* these criteria!

Quicksort

There are many published descriptions of quicksort; I suggest the one given in Cormen, et al., *Introduction to Algorithms*, MIT Press (Ch. 8.1 of 1990 edition or Ch. 7.1 of 2001 edition), or the one in (any edition of) Sedgewick, *Algorithms*, Addison-Wesley. *Don't* bother to compute the partition element location in some fancy way like "median of three;" just use the first or last element of the (sub)array.

Hints

Giving the `-S` option to `cc` or `gcc` together with a `.c` file produces a file with `.s` extension containing the assembly code generated by the compiler for that `.c` file. This can be very useful for seeing what instructions the compiler chooses, exactly what the function calling conventions are, etc. You may want to write `qs` in C first (a pointer-based version will compile to better code), and use the generated code as the basis for your own routine. This approach won't help you with the explanatory comments, though! And you will probably need to improve this code further by hand to avoid being penalized for obvious stupidities in the compiler-generated assembler. Incidentally, also specifying `-O2` to `gcc` is a good idea, because the resulting code will be both more efficient and clearer.

Integer register conventions on SPARC are described in Appendix D of the SPARC Architecture Manual. Note that any non-leaf subroutine (i.e., any subroutine that may call another subroutine) *must* begin by creating a stack frame of at least 96 bytes, using an instruction of the form `save %sp, -96, %sp`, and must conclude with a `restore` instruction; otherwise, don't mess with `%sp`. After the `save` instruction has been executed, you'll find the input arguments in `%i0` and `%i1`, the frame pointer in `%fp` (which is the same as `%i6`) and the call address in `%i7` (the return address is 8 more than the call address). You can use the other integer registers freely, bearing in mind that the `%o` registers aren't saved if you call another procedure.

You can assume that the caller saves any floating point registers (which are *not* windowed), so you can use them freely. Note that double-precision floats (64 bits) are stored in adjacent *pairs* of `%f` registers, where the first of the pair must be an even number (e.g. `%f0` and `%f1` or `%f6` and `%f7`). An instruction like `ldd [a], %f2` actually loads `%f2` and `%f3` with a 64-bit quantity from memory address `a`. Note also that there is no way to move floating point numbers directly to or from the integer registers; you have to go through memory(!). Don't forget that memory locations referenced in double-word load or store instructions must be aligned on 8-byte boundaries. Also, there is no `fmovd` instruction; to move a double from one pair of registers to another, you must do two `fmovs` instructions. (The assembler in `/usr/ccs/bin/as` accepts `fmovd` as a synthetic

instruction op-code, which just turns into two `fmovs` instructions. The `/pkg/gnu/bin/as` assembler doesn't, though.)

Other than delay slots on jumps, most timing issues in Sparc are hidden from programmer view by *interlocks* which prevent an instruction from executing until any preceding instruction on which it depends has been completed – even if this means stalling the processor for one or more cycles. In V8 Sparcs, one exception to this is floating point comparisons and branches: at least one non-FP instruction must occur between any FP comparison (e.g. `fcmp`) and a conditional branch based on it (e.g. `fbe`). Otherwise, the result of the branch is undefined. `gcc` often inserts a `nop` for this purpose, but you can use any other integer instruction. (Actually, V9 Sparcs apparently have an interlock to stall the processor if necessary after the compare, without the need for an explicit filler instruction, but to be properly V8-compliant you should not assume this.)

Early versions of the SPARC did not implement the integer multiply instructions (`umul` and `smul`), so most compilers don't use them, but you can on the CS dept. machines. Remember, there are especially easy ways to multiply by powers of 2.

Organization and Submission

Place your solution in a single file called `qs.s` and mail it to `cs322-01@cs.pdx.edu`. Your C code or pseudo-code should be included in a block comment at the top of this file. You must mail this file as a plain text *attachment*; the contents of the message itself don't matter. Your mail subject line should contain the word "HW2". We should then be able to assemble your program by creating a fresh directory, saving your attachment, and typing

```
gcc -c -o qs.o qs.s
```

or

```
/pkg/gnu/bin/as -o qs.o qs.s
```

Assuming that `main.c` contains a test driver (such as the one described above) we should be able to produce a linked executable `qs` containing both the driver and your code by copying `main.c` into the directory and then typing

```
gcc main.c qs.o -o qs
```

Note that we will be using automated mechanisms to read, compile, and test your programs, so adherence to this naming and mailing policy is important! You may lose points if you fail to submit your program in the correct way.