CS 201 Section 1 Lab 3 – due 6pm, Monday, February 28, 2000

Matrix Multiplication

Write an assembly language subroutine `square`, suitable for calling from C, that multiplies a double precision matrix by itself and returns the result matrix. The calling interface is given by

```
double *square(int n,double *a)
```

where `a` is the input matrix, of dimension `n` $\times$ `n`. The input matrix is stored as a one-dimensional array of `doubles` in row-major order, e.g.,

$$a_{11}, a_{12}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{m1}, \ldots, a_{mn}$$

The subroutine should allocate a fresh array on the heap, in the same format, to hold the result matrix (also of dimension `n` $\times$ `n`), and return a pointer to this array after filling it in.

You can assume that $\texttt{n}^2$ does not exceed 2^{31}. If there is no room available in the heap, your routine should return the null pointer 0.

In case you don't know, the formula for multiplying matrices is as follows: if A is an $m \times n$ matrix and B is a $n \times p$ matrix, then the product $C = AB$ is a $n \times p$ matrix whose elements are given by:

$$C_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

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

```
double a[] = {1.1,2.2,3.3,
              4.4,5.5,6.6,
              7.7,8.8,9.9};

main () {
   double *c = square(3,a);
   int i,j;
   for (i = 0; i < 3; i ++) {
        for (j = 0; j < 3; j++)
           printf ("%f ", *c++);
        printf("\n");
   }
}
```

it should produce the following output:

```
36.300000 43.560000 50.820000
79.860000 98.010000 116.160000
123.420000 152.460000 181.500000
```

In addition to the assembly code itself, you *must* turn in pseudo-code or C code for your subroutine, in the form of a comment at the top of your assembler file. Also, 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).

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

Hints

You can assume that the caller saves any floating point registers, so you can use them freely.

To allocate space for the result array, use the `malloc` function from the C library. It takes a single integer argument (in `%o0`) indicating the number of bytes to be allocated, and returns a pointer (in `%o0`) to the allocated space, or a 0 if no further space is available.

Don't forget that memory locations referenced in double-word moving instructions must be aligned on 8-byte boundaries.

You may choose to generate your assembly code by writing C code first and then using the `-S` flag on `cc` to produce a `.s` file. But be warned: if you do this, you'll still need to understand and document (by a comment), every line of assembler, and you'll need to make sure that there are no obvious improvements you can make over what the compiler produces—otherwise, you'll lose points on the "clarity" and "concision" criteria.

To submit this lab, you need to send just one file, `square.s`, containing all your assembly code. We'll test it using our own driver, which will be similar to, but more demanding than, the driver above. Mail it as the body of an email message to `cs201-1acc`, e.g.,

```
mail cs201-1acc < square.s
```

Your submission will be built by using the command

```
cc -o squaretest driver.c square.s
```

and then tested by running `squaretest`