

Assignment 5: Basic x86 Assembly Programming

Due: Monday 02/21/05, Multiplication Factor: 1.5

This assignment is just to get you comfortable with writing x86 assembly. This assignment will not be graded on measured performance, and we will not be applying a lot of complex optimizations (eg., software pipelining). However, I will be counting off for inefficient use of resources in the normal code (for instance, performing unnecessary memory loads in loops if you have available registers). You can get credit back for mistakes of this sort by correcting the code.

I will also be grading on readability, as assembly needs to be made as clear as possible if it is to be maintained. Use the cpp tricks we discussed in class, and provide good (though not effusive) comments, particularly when you are doing something non-straightforward. Always show your current x87 stack. You may assume all dimensions are at least 1.

You will need to write three assembly routines, which I recommend tackling in the following order:

1. Double precision dot product (`ddot`), with prototype:

```
double ATL_UDOT(const int N, const double *X, const int incX,
                const double *Y, const int incY)
```

You will ignore the `incX` and `incY` parameters. This routine should not be unrolled, or optimized beyond making sure you efficiently use the available resources. Name the file `ddot.S`.

2. Single precision dot product (`sdot`), with prototype:

```
float ATL_UDOT(const int N, const float *X, const int incX,
               const float *Y, const int incY)
```

You will ignore the `incX` and `incY` parameters. This routine should unroll the loop to a factor of 6, but use only 3 accumulator registers for the dot product (i.e. scalar expansion using 3 regs instead of 1). The kernel must produce correct answers regardless of `N`, so be sure to write cleanup code. Name the file `sdot.S`.

3. Double precision matrix multiply (`dgemm`), of prototype:

```
void dgemm_asg(int ta, int tb, int M, int N, int K, double alpha,
               const double *A, int lda, const double *B, int ldb,
               double beta, double *C, int ldc)
```

You will ignore the `ta` and `tb` parameters, always doing the 'NoTrans', 'Trans' case, as before. You do not need to unroll any loops, but you do need to correctly handle all α and β cases. In particular, $\beta == 0.0$ should not cause a read of `C`. You do not have to worry about efficiency, so you may apply α and β in the loops (rather than specialize many different cases). However, make sure you apply them in an most outer loop possible. Your loop order (from outer to inner) should be : NMK. Why is that? Name the file `dgemm_asg.S`.

To get the files you need, perform:

```
cp ~whaley/teach/cis5930/ASG5/* .
make archdirs
```

I have provided testers for these routines. Advice on their use is as follows:

1. **ddot**: This one just gets your feet wet (my routine was 27 lines with comments, and took me approximately 1 minute to write). Test with: `make xddottst ; ./xddottst`.
2. **sdot**: Here we are going to see the complexities of unrolling, as well as wrestling with the dreaded division instruction. You may want to first adapt your ddot.S to single precision, and make sure that works:

```
make xsdottst ; ./xsdottst
```

Then, I recommend getting the unrolled code working, where you test only multiples of 6:

```
./xsdottst -n 12
```

Once that gives the correct answer, add the cleanup code, and test all cleanup cases.

```
./xsdottst -N 6 18 19 20 21 22 23
```

I'd be careful to test that problem sizes less than the unrolling factor work as well.

3. **dgemm**: `matmul` requires three nested loops, and you must get the ptr arithmetic and updates correct for each. I recommend first writing the general the $\beta = X$, $\alpha = X$ case, and getting that to work for no looping at all, eg:

```
./xdgemmtst -a 1 2.1 -b 1 0.9 -m 1 -n 1 -k 1
```

Once that passes, make sure your K-loop indexing is correct:

```
./xdgemmtst -a 1 2.1 -b 1 0.9 -m 1 -n 1 -k 8
```

When that passes, put in the special cases for alpha and beta, and test them all:

```
./xdgemmtst -a 3 1.0 0.0 2.1 -b 3 1.0 0.0 0.9 -m 1 -n 1 -k 8
```

Now, test the first inner loop:

```
./xdgemmtst -m 3 -n 1 -k 8
```

And finally the outer loop:

```
./xdgemmtst -m 3 -n 2 -k 8
```

When all of these pass, test larger problems and all combinations