Assignment 6: Generic structure merge sort in Thumb-1 assembly

mrgSort deadline: Monday, 11/02/20 (before class)

You will write two thumb-1 assembly functions, both of which will supply our generic merge sort ABI of

```
void *mrgSort(void *basep, unsigned long gNxt, unsigned long gI)
```

The required handin should use the filename `mrgSortFast.S`. This routine will extend the meta-data optimization we saw in p0's mrgSortFast by adding an additional pointer return which can be set during the recursion to speed up classic merge sort.

As we have seen in prior assignments, the above API will be provided by a non-recursive wrapper code that will do some checks and then call a local (possibly non-ABI compliant) function for the recursion.

As such, you can have this "function" do anything that is legal, but only with the following restrictions:

1. No hybrid algorithms: all sorting must be done by recursive merge sort;
2. Splitting of unsorted lists must be done via recursion, with the number of nodes on both sides being as close to same as possible (illegal to use nL=N/4, nR=N-nL, for instance);
3. While the recursive function signature can't be dictated, the amount of metadata *is*. You can have at most one extra pointer over p0's fast metadata, which corresponds the functional equivalent of a recursive signature of:

```
void *mrgSortRec
(
   unsigned int N,    // number of nodes in list starting at basep
   void *basep,       // address of first node in list to be sorted
   unsigned long gNxt,// gap in bytes from start of struct until next ptr
   unsigned long gI   // gap in bytes from start of struct until int to sort on
   void *ML,          // OUT: I used this to point last node in merged list
   void *UB           // OUT: like p0, base of remaining unsorted nodes
)
```

On a15, change into a directory you create for these assignments, and issue the following command to get the file you need:

```
cp ~whaley/classes/ARMF20/p6_mrgSort/* .
```

As we have seen, writing directly in assembly when still investigating algorithms is incredibly time-wasting, and often doesn't allow you to think about high level ideas clearly, and therefore the provided Makefile has special options to allow rapid prototyping in C. These targets can be invoked:

```
make x[tst,tim]SortC mrgF=[name of C file]
```

The macro `mrgF` is used so you can keep a bunch of files implementing various ideas around.

Once you've got a good-performing algorithm, you'll write your thumb-1 assembly version, which can be tested & timed:

```
make x[tst,tim]Sort mrgF=[name of .S file]
```

We will keep track of best C and assembly timings on piazza, and you should post as soon as you get things working. If you don't see an immediate use for the new metadata, you can at least port your p0 mrgSortFast as your first C file, which will get you familiar with the framework, and thinking about the recursion again.

While the assembly is the only mandatory handin, if you use C for prototyping, you should submit on canvas your best-performing C version as well, particularly if you run out of time, and don't have time to write your final algorithmic tweaks into assembly (but make sure you have at least submitted a slower .S file, since that is mainly what you will be graded on!).

We have seen that one way of getting optimization is to find a special case you can check for w/o increasing the cost of the worst-case appreciably, as when we scoped of the list was already in order before.

This type of case needs the timer to be specialized to show it, unless it happens regularly by chance. Therefore, I may extend the timer to exercise more cases, and if my cases don't show a speedup you believe you have made, you can extend the timer (w/o changing the default case) so that it shows your improved special-case performance.

The special cases ought to be something that might actually happen in real world, and not only work on one problem size, etc.

If you extend the timer, post the diff of it and the default timer, and Majedul or I will add it if it seems worthwile.

Both tester/timers take a required list of list lengths to test, for instance:

```
valgrind --leak-check=full --malloc-fill=0x88 --track-origins=yes ./xtest 0 1 3 777
```

will invoke your sort on the given 4 different lengths. My code produces no valgrind errors or warnings.

We will confine ourselves to strict Thumb-1, which means no ARMv6 or ARMv7 extensions. You should assume that this sort is a major use of the embedded system, that lists could be millions long (ultimately, bounded only by the address space of machine) and that its performance/energy use makes added **thumb-1** (not ARM) code size worthwhile.

We will be particularly concerned with performing unnecessary loads, stores (burn energy and time far worse than extra integer ops), and branches.

Note that what we are concerned with is the *dynamic* number of these instructions (i.e. the number that are encountered when running), not the *static* number (the number you get if you just count them in code without taking into account loops and other path-based decisions).

Given this sort is a main feature of our embedded system, code size is a secondary consideration. In particular, you will make the merge code as large as required to optimize it as described above.

For long lists, the energy/performance vs. code size optimization hierarchy is:

1. $O\left(Nlog_2(N)\right)$: merge code – always expanded to make faster/less energy
2. $O\left(log_2(N)\right)$: non-merge recursive code – smaller better, but don't be dumb
3. $O(1)$: non-recursive wrapper: make it small if you can

Examples:

```
>make xtstSortC mrgF=mrgSortFast
>valgrind --leak-check=full --malloc-fill=0x88 --track-origins=yes \
   ./xtstSortC 0 1 2 3 4 5 6 5 7 8 9 10 11 777
>
>make xtimSortC mrgF=mrgSortFast
>./xtimSortC 1000000
   INIT   N=1000000 took 2.153059e-01 seconds
   TIMING UNSORTED SORT N=1000000 case, word_t
   SORT UNSORTED N=1000000 took 9.358245e-01sec, sort/init=4.3
   SORT   SORTED N=1000000 took 1.120111e-01sec, sort/init=0.5, unsort/sort=8.4
>
>make xtimSortC mrgF=mrgSortSlow
>./xtimSortC 1000000
   INIT   N=1000000 took 2.124682e-01 seconds
   TIMING UNSORTED SORT N=1000000 case, word_t
   SORT UNSORTED N=1000000 took 1.949856e+00sec, sort/init=9.2
   SORT   SORTED N=1000000 took 1.119367e-01sec, sort/init=0.5, unsort/sort=17.4
```