Assignment 2: Safe and unsafe long long vecsum

Due: Wednesday 09/23/20 (before class)

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

```
cp ~whaley/classes/ARMF20/p2_isum/Makefile .
```

View and understand the provided Makefile and `tester.c` so that you can write your own in the future, or adapt the Makefile to work on your home system.

In this assignment, you will write 3 assembly routines that accomplish the following in three different ways:

```
unsigned long long lvecsum(unsigned int N, unsigned int *V)
{
   unsigned int i;
   unsigned long long sum=0;
   for (i=0; i < N; i++)
      sum += V[i];
   return(sum);
}
```

These three routines should provide this basic summation in 3 ways:

1. `lvecsumI.S`: This assembly will just treat `sum` as a simple 32-bit accumulator (don't forget to zero R1 so its correctly returned for `long long` though!). This code will be graded only on correctness, and makes sure you can do the basic operation correctly w/o worrying too much about long long or optimization. Build with command:
   `make xitest asm=lvecsumI`
   When running the tester, you should pass only those tests labeled 32-bit, and fail the rest.
2. `lvecsumL.S`: This assembly routine will use `adc` to return an accurate 64-bit accumulator. It will also be graded mostly on correctness, and ensures you understand how to use `adc` before going on to the fun stuff. Build with command:
   `make xitest asm=lvecsumL`
   When running the tester, you should pass all tests.
3. `lvecsum8.S`: This version is the main part of the assignment, and is described in detail below. Build with command:
   `make xitest asm=lvecsum8`
   When running the tester, you should fail only those tests labeled 64-bit (so you'll pass those labeled 8x32, in addition to the ones passed by `lvecsumI.S`).

lvecsum8.S Discussion

`lvecsum8.S` is going to be a fairly dumb way to get extra precision while using `adc` only outside the loop. We could imagine its being posed this way because a forthcoming machine will have a very slow `adc`, and so we can't use it in a loop, and we know this stupid way of achieving extra accuracy works for our application, even if we are really doing to give you a simple routine that still requires some thought to optimize for code size, while making you use a bunch of registers

The stupid method we will use to get roughly 8 times as much accuracy for randomly distributed data is to use 8 different 32-bit accumulators inside our main loop. Therefore, your assembly loop will implement something like:

```
unsigned long long sum=0;
unsigned int s0=0, s1=0, s2=0, s3=0, s4=0, s5=0, s6=0, s7=0;
for (i=0; i+8 < N; i += 8)
{
   s0 += V[i];
   s1 += V[i+1];
   s2 += V[i+2];
   s3 += V[i+3];
   s4 += V[i+4];
   s5 += V[i+5];
   s6 += V[i+6];
   s7 += V[i+7];
}
sum = s0+s1+s2+s3+s4+s5+s6+s7;
```

Note that the above code fragment is only a partial solution, since you will need to handle problems that are not multiples of 8 (therefore requiring some cleanup code, in addition to above unrolled loop). Also note that you will need to use `adc` for the summation into `long long`, but that is outside the loop. so nobody cares if its slow.

The running sums `s[0-7]` **must** be held in registers for life of loop, and each add will be an individual add instruction despite our need to make the code size small, in order to improve performance (therefore, the main loop must have at least 8 add instructions in it).

Once you get this working, you will need to optimize it for code size using ARM (not thumb!). In particular I will count off -5 for every unnecessary instruction inside your loop (my current implementation requires 12 instructions for the above unrolled loop; note that a branch is an instruction, but a label is not).

I will also count of -1.25 for every unnecessary instruction in your entire *function*. My current function requires roughly 40 instructions outside the main loop to handle all the required error checking, loop cleanup, variable allocation and initialization, etc.

If I find time to look at problem again, these instruction counts could still shrink. Use the coding style outlined in prior assignment.