2.1 When we were discussing floating point addition, we made the simplifying assumptions that each of the functional units took the same amount of time. Suppose that fetch and store each take 2 nanoseconds and the remaining operations take 1 nanosecond.

(a) How long does a floating point addition take with these assumptions?

(b) How long will an unpipelined addition of 1000 pairs of floats take with these assumptions?

(c) How long will a pipelined addition of 1000 pairs of floats take with these assumptions?

(d) The time required for fetch and store may vary considerably if the operands/results are stored in different levels of the memory hierarchy. Suppose that a fetch from level 1 cache takes 2 nanoseconds, while a fetch from level 2 cache takes 5 nanoseconds, and a fetch from main memory takes 50 nanoseconds. What happens to the pipeline when there is a level 1 cache miss on a fetch of one of the operands? What happens when there is a level 2 miss?

(a) Previously, we had 7 steps each taking 1 nanosecond for a total of 7 nanoseconds. Step 0 (fetch) now takes 2 nanoseconds, and step 6 (store) also takes 2 nanoseconds, bringing our total for a single floating point addition to 9 nanoseconds.

(b) An unpipelined addition will take 9 nanoseconds per addition $\times 1000$ floating point additions, for a total of 9000 nanoseconds.

(c) We are effectively only considered with when the last addition completes. That is operation at index 999. The 0th fetch starts at 0, and finishes at 2, the 1st fetch starts at 2 and ends at 4, 2nd at 4 and ends at 6, and so on, thus fetch i starts at $2i$ and ends at $2i + 2$. Therefore, the 999th fetch will start at time 1998 and end at 2000. It's next 5 operations will be unblocked, as each only takes 1 nanosecond, thus the previous will be done before the 999th addition needs it, thus the Round step will complete at time 2005. Again, it gets more complex when it's time to store, addition 0 starts store at time 7 and completes at 9. Thus addition 1 will not start addition until time 9 and will complete at 11, addition 2 will start at 11 and complete at 13 and so on. Generalizing this, we find that addition i will complete at $9 + 2i$. Therefore, addition 999 will complete at time 2007.

(d) Fetching from level 1 cache taking 2 nanoseconds is exactly the situation we just covered above. A level 1 cache miss of 5 nanoseconds offsets the next fetch start by 3 nanoseconds, thus the total pipeline time will be 3 nanoseconds longer than if all fetches took 2 nanoseconds.
Similarly, if a level 2 cache miss occurs of 50 nanoseconds, the total time will be offset by 48 nanoseconds longer than it would have taken if it only had taken 1 nanosecond to fetch from the cache.

2.3 Recall the example involving cache reads of a two-dimensional array (page 22). How does a larger matrix and a larger cache affect the performance of the two pairs of nested loops? What happens if MAX = 8 and the cache can store four lines? How many misses occur in the reads of A in the first pair of nested loops? How many misses occur in the second pair?

When accessing in row major order, the results will be very similar, as we have 4 cache lines, and our matrix has 8 rows, that means our cache can store up to half of the matrix at a time. Whenever the program starts a new row, a cache miss occurs, the entire row is loaded into the cache, and the program will continue without a cache miss until the next row is accessed. Thus, there will be a total of 8 cache misses when accessing data in row major order.

Things are much worse when accessing the cache in column major order. The first 4 fetches all hit cache misses, and load the first 4 rows into the cache. The program continues onto row 5 (column 1) and again hits a cache miss loading that row into the cache. This trend continues, the cache continually stores the most recent 4 rows in the cache, but we always switch to a new row when accessing in column major order, thus every single access generates a cache miss, for a total of $8 \times 8 = 64$ cache misses.