Communiqué on VHDL Pedestal Correction Algorithm

Tyler Lutz
8.8.2011

Motivation

Inconsistencies in baseline signal values across cells introduce a major source of *systematic* error in reading incoming signal values over time. We would like to mitigate this source of error by renormalizing the baseline values across the cells. In practice, this is done by computing an average of the non-pulse signal for each cell and then subtracting this average from the any later signal readings.

Implementation

Perhaps the most straightforward way to carry out the averaging would be to compute the average of a set of sample data points taken before data acquisition begins, but this approach has the two obvious disadvantages of, firstly, failing to account for dynamic changes in the pedestal values and, secondly, requiring a dedicated period of time after start-up for the purpose of reading and averaging empty signal values. Nevertheless, because it requires practically no computation after this short period, this initialization average is our chosen approach.

The accuracy vs. storage memory trade-off

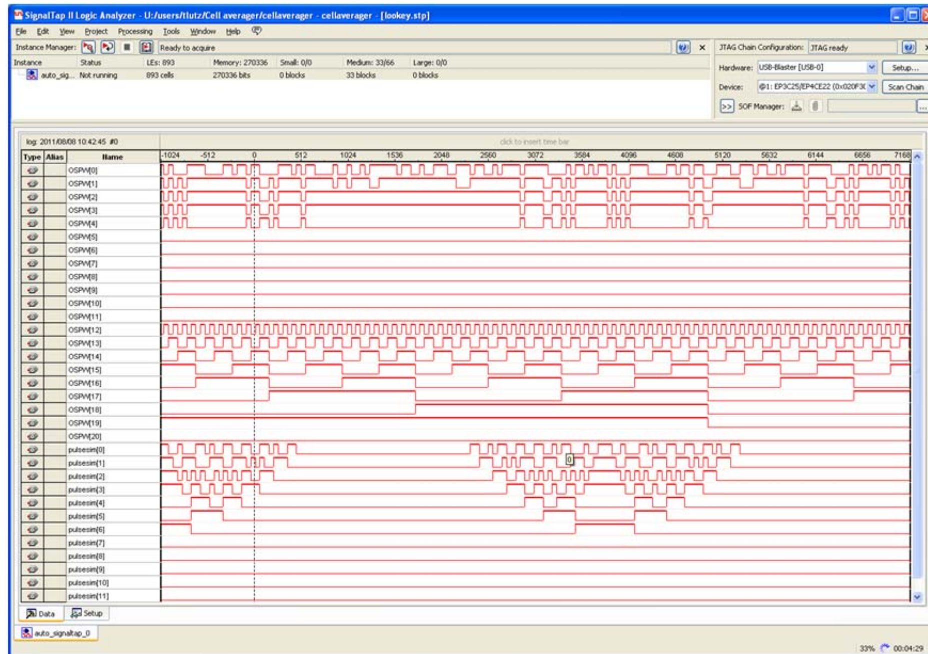
The question then is how many initial data points are necessary to form a reasonable estimate of the pedestal values? This ends up being a parameter which needs to be carefully tweaked, as it generates thorny problems on both extremes; if a large sample size is desired, it becomes too cumbersome to store a running sum of the data points (since 256 huge sums are needed) and then to divide by n afterwards, so the running *averages* must instead be stored. Malheureusement, storing the average at the end of each clock cycle necessarily leads to unavoidable truncation errors resulting from the large number of division operations being carried out. A little thought shows that, in the worst case, the average might decrease in value by one unit for every two computations (!), since on average the amount to be truncated will lie between 0 and 1, with no preference for either side. Luckily though, our specific algorithm causes the truncated quantity to be biased towards 0, leading to a much more modest decay in the average. But modest as it may be, the leakage adds up rendering the algorithm useless for large sample sizes (at least without artificially inflating the average to account for this downward trend—a possibility which is currently under investigation).

We are thus led to flirt with the idea of smaller sample sizes, allowing running sums to be saved and only one division to be computed per cell (namely the grand division at the end of the sample collection which outputs the actual average). But now how small of a sample size is too small? This, unfortunately, cannot be determined a priori but is dependent instead on the standard deviation of the signal (small deviation would allow us to accurately determine the average with a very small sample size, and a large deviation…let's not go there).

Simulation

The following figure shows an initialization average with a sample size of 25 (per cell(!)) using what end up being random background noise (the Gaussian pulses are deliberately not correlated to the 256-cell blocks, so the averages for each cell are effectively random samplings from the Gaussian pulses).

The input data is on the bottom, the middle shows the cell number (the relative position in the 256 cell block) and the top shows the raw amount of the average:

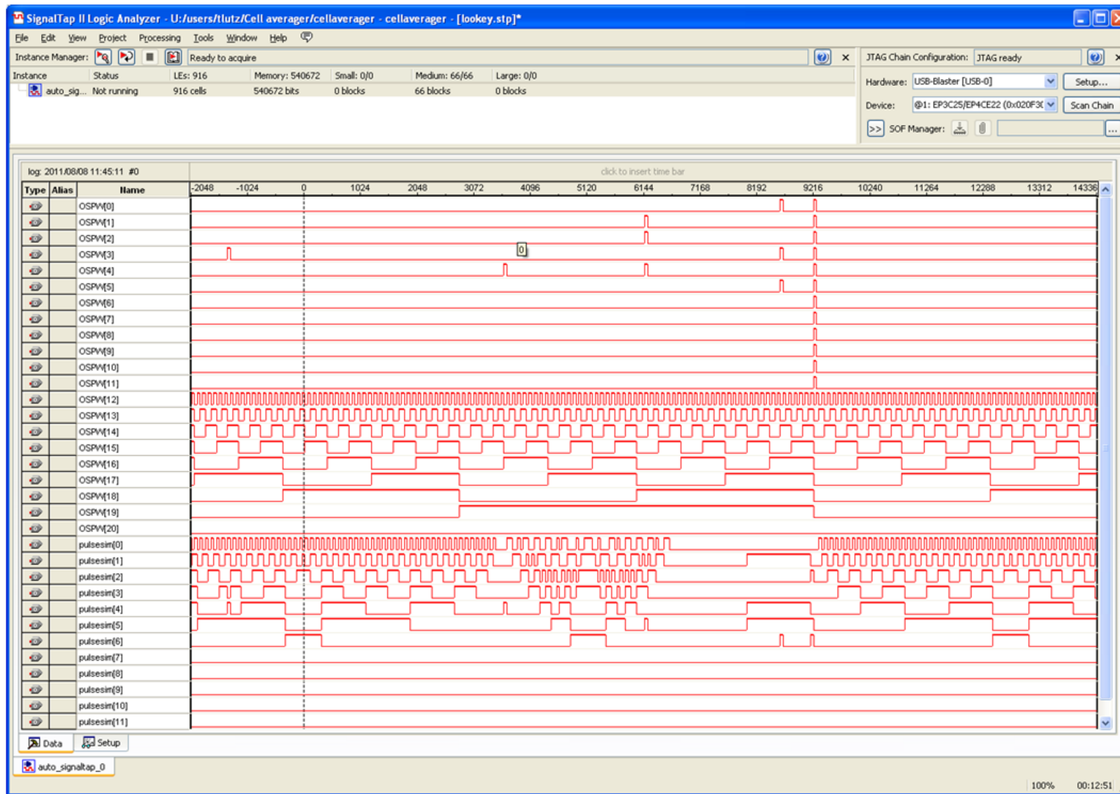


Since, as we have said, the sample points are effectively random, we expect the average to hover around the overall average of the pulse input value, which, including all of the 0's in between, comes out to 20.23 volts. A look at the above figure shows the algorithm's computed average to be around 16volts for any given cell—satisfactory, taking the aforementioned truncation error into account.

Another way to test the algorithm is to actually line the input up with the 256 cell blocks—in which case our algorithm should always spit out an average identical to the input. To be sure it is actually identical (hard to determine from just looking at wiggly red lines), we subtract the input value from the computed average, meaning that we would expect a uniform output of zero for every cell.

But not so fast—how can we be sure that the zeros we are seeing are actually what we want and not just the output mechanism malfunctioning altogether, or some other devious trick our code might be playing on us? To convince you that this is not the case, we doubled the input pulse size (for 256 to 512) by copying and pasting the first set of 256 points but introducing a few impurities in the second set which should show up as non-zero values in our output reading of the second set of data! (think about it; if cell 100 is fed 16 the first time around and 20 the second, the average will be 18, and then later when 20 comes in again, the difference of the pulse from the average would be 2, not zero. (This will not occur with the input of 16, since the difference is negative, and thus readout to zero). This wouldn't (shouldn't) happen in an actual physical device, but it is a clever way to test the algorithm).

The results speak for themselves:



Input pulse bottom, address middle, and difference between stored average and pulse top. Four impurities introduced, with a marker embedded in the code to show where the block repeats.