Photo-detector DAQ and Timing Analysis VHDL Code and Simulation

Tyler Lutz
20.07.2011
University of Chicago

This document elaborates both the design and functionality of a chunk of VHDL code aimed at characterizing various aspects of incoming pulse and signal data from a photo-detector fed into a Field Programmable Gate Array (FPGA) via an onboard ASIC acting as an oscilloscope. The actual code with original comments is included in bulk in Appendix A. A testbench offering fabricated and easily manipulated triangular pulses for the purposes of simulation is given in appendix B.

Motivation

It is computationally advantageous to perform various data analysis functions on an input signal from the photo-detector in real time rather than outputting the raw signal to software which will make such calculations after the device is finished taking data. Specifically, we would like to know the time of arrival of a given pulse as well as various features of the pulse waveform such as its average height, its full width at half maximum, its rising and falling times, the height of its peak(s), and the number of electrons it represents. Additional data related to the baseline (non-pulse) signal such as its overall average, its standard deviation, and its significance is also useful for calibrating the actual pulse data.

Approach and Implementation

The timing analysis is structured around three different possible approaches, as outlined in *Signal Processing for Pico-second Resolution Timing Measurements*[^1]: the first of these is multiple threshold discriminator which records the time at which the pulse breaches various (here three) different predetermined threshold heights, the second a leading edge discriminator which performs a linear fit to the three threshold data points and identifies the time of the fit’s zero-crossing, and the final method is a constant fraction discriminator which outputs the time at which the pulse attains a predetermined fraction of its peak value. The other signal analysis procedures are carried out with straightforward computational techniques expounded on below.

As presented here, the above processes are executed in one process of a single, thus admittedly unmanageably large VHDL entity. Though we acknowledge the potential efficacy of reducing this monolith into more tenable functional units (components) at a later point, we nonetheless present the code in its current form since the exact form of such a structural model depends heavily on the desired form of output and input, which are underspecified at this time. For the purposes of this code it is assumed that the input is a 12 bit (totally arbitrary and easy to change) pulse reading in parallel, and that the outputs are similarly of 12 bit length and in parallel. At this time though we are more interested in developing and simulating the relevant functions rather than optimizing how these constituents all fit together, pending more concrete input and output specs.

Shared variables

The code utilizes a number of shared variables an understanding of which is necessary to comprehend the function of any of the sub-processes. At every clock leading edge, the 12 bit input is converted into an integer (affectionately named ‘pulsar’) in one of the code’s perambulatory clauses. This integer is then stored in the first bin of a memory array ‘pulsarsave.’ Every subsequent clock cycle first shifts the existing data into the \((n+1)\)th bin and stores new data in bin zero, effecting a memory of the pulse of storage size ‘savescope’ number of bins, specified in a constant declaration. An identical process is carried out to store the background signal in ‘pulsarbasesave,’ which also has size savescope.

The variable accsec acts as the global stopwatch, as it increases in value by one unit at each clock leading edge.

Many of the calculated results are used in later processes, and if this is the case, the calculated result is stored in a quasi-permanent shared variable (which is overwritten whenever a new result is calculated), while the original result is converted to parallel binary output, which destroys the integer data by reducing it back to zero. Parallel output is scheduled to last until the signal value changes, which using our test bench is three clock cycles—this can of course be modified to meet desired output specifications.

Elaboration and Simulation of Timing Analyses

Multi-threshold timing analysis

The functional unit for all three threshold discriminators consists of the following (condensed) snippet of code:

```plaintext
if pulsar>=multithreshold(0) and inhibit=0 and pulsarsave(1)<multithreshold(0) then
    eventprint(0):=accsec;
    eventprintread(0):=eventprint(0);

    for q in 11 downto 0 loop
        --(PARALLELIZE OUTPUT to O) --

        if q=0 then
            inhibit:=1;
        end if;
    end loop;
end if;
```

*Code snippet 1: trigger for first threshold, with the parallelization of output abbreviated for readability*

The saving of the current clock time (accsec) into eventprint(0-2) is triggered at the first time pulsar exceeds the given threshold value, which is thus also when the last stored data point is still below the threshold. This setup does not require the pulse to equal the threshold for accurate output, and it moreover ensures that the falling edge of the pulse is not counted as the arrival of a new pulse.
After the parallelization has been executed, an inhibitor effectively breaks the process to prevent continued stuttering and invalid output. This negative feedback is reset in a perambulatory clause whenever the pulse signal changes to allow for continued data processing.

A waveform simulation using triangular pulses of linearly increasing peak height, starting at 25 volts (units can be changed) and threshold values of 5, 8, and 10 respectively is presented below:

![Waveform Simulation](image1)

*Fig. 1: Multi-threshold discrimination simulation with four triangular pulses of linearly increasing height. Clock signal is at top, binary value of input pulse in middle, and multithreshold output at bottom. Time in picoseconds with 100ps per full clock cycle.*

Zooming in on the second pulse:

![Zoomed Waveform](image2)

*Fig. 2: Multi-threshold readout in ps of second pulse (see fig. 1) with trigger thresholds at 5, 8, and 10 volts, pulse height 26 volts*
It should be mentioned that there is a delay of one clock cycle between the time at which the threshold is breached and the time of the output, which then lasts until the pulse value changes (here, 3 full cycles). In all other simulations performed with varying pulse shapes and sizes, the algorithm works as anticipated.

Leading Edge Discriminator

Linear regression timing analysis based on the respective times at which the pulse breaches the three thresholds (above) is initiated by a positive feedback variable ‘threeyep’ which is turned on when the third threshold is crossed. A linear fit is then performed, and the fit is traced to zero (or the baseline average) and the time of the zero-crossing recorded:

```plaintext
if threeyep=1 then
    xmean:= (eventprint(0) + eventprint(1) + eventprint(2))/3;
    ymean:= (1000*multithreshold(0) + 1000*multithreshold(1) + 1000*multithreshold(2))/3;
    for i in 0 to 2 loop
        ssxx:=ssxx + (eventprint(i)-xmean)**2;
        ssxy:=ssxy + (-1)*xmean*ymean + 1000*eventprint(i)*multithreshold(i);
    end loop;
    linfitzero:=xmean — (ymean*ssxx)/ssxy;
    for u in 11 downto 0 loop
        --(PARALLELIZE OUTPUT to OL)—
    end loop;
end if;
```

*Code snippet 2: trigger for Leading Edge Discriminator, showing linear fit algorithm and with the parallelization of output abbreviated*

The linear fit algorithm[^2] computes the slope of the fit, b where y=a+bx, as b=ssxy/ssxx using ssxx=sum(xi^2-xmean^2) and ssxy=sum(xi*yi-xmean*ymean), with i from 1 to 3. Solving for the x intercept (-a/b), the desired output time is found to be xmean-ymean/b.

The mean of x (the time axis, picoseconds) and y (voltage axis) are computed in the usual way with one very important exception; it was found that the raw least squares fitting algorithm introduces a significant truncation error, and in order to mitigate this ymean is multiplied by 1000, and this is removed from the final answer by including a factor of 1000 in the ssxx calculation (they cancel in the end because the final output contains a ymean/ssxy term; also note the first term in ssxy calculation does NOT get multiplied by 1000 since it already contains a ymean term). Larger multipliers were played with, but few gains were made for numbers higher than 1000.

At least for the triangular pulses with which much of the simulation work was done, this method is (predictably) by far the most exact (since the leading edges are already lines, the linear regression is exact), deviating from the expected values only by a few clock cycles, which we attribute to further truncation errors.
A simulation of the leading edge discriminator is included below. As an illustrative example of the aforementioned accuracy of this method for triangular pulses, note that the first output offered by this discriminator gives time=1, which is exactly when our testbench says the pulse should start.

![Simulation of the leading edge discriminator](image)

**Fig.3: Leading Edge Discriminator readout (in ps) on bottom, with multi-threshold and pulse readouts above for reference; trigger thresholds 5, 8, and 10 volts, pulse height 25, 26, and 27 volts respectively**

### Constant Fraction Discriminator

A constant fraction discrimination could be carried out by literally running through a list of stored values, but here we've implemented a more computationally conservative method of attenuating the incoming pulse to 30% of its original value and comparing it to a pulse delayed by a pre-determined amount. The time at which these two are equal (or more precisely, when their difference is zero) is the recorded output. The operational code is:

```plaintext
if (pulsar>multithreshold(0)) and inhibit_cfd=0 then
    if ((-30*pulsar/10+10*pulsarsave(delay)) > 0) then
        eventprint(3):=accsec-delay;
    end if;
    for v in 11 downto 0 loop
        --(PARALLELIZE OUTPUT to OCFD) --
        if v=0 then
            inhibit_cfd:=1;
        end if;
    end loop;
end if;
```

*Code snippet 3: CFD trigger body, with the parallelization of output abbreviated for readability*
The calculation is initiated when the first threshold is breached, and we have again multiplied through by 10 to avoid truncation errors (only the relative shapes of the pulses are relevant here, so we can multiply away with impunity). The delay time is subtracted from the final answer to offset the delay used in the calculation. Also to note is our use of a negative feedback inhibitor, which our simulations have shown to eliminate anomalous output.

The parameters upon which our approach was based were roughly optimized\(^{[\text{1}]}\) for real photodetector waveforms, and thus, though this CFD still perform notably better than our above multithreshold approach, we have not taken the time to attempt to optimize the parameters for each of our simulation trials, and thus they output a time significantly later than the above leading edge discriminator. A sample simulation is exhibited below:

![Image](image.png)

*Fig.4: Constant Discriminator readout (in ps) on bottom, with multi-threshold and pulse readouts above for reference; trigger thresholds at 5, 8, and 10 volts, pulse height 25, 26, and 27 volts respectively*.

Due to the concomitant processing delay associated with the delayed signal, the output is issued substantially later than in the above two methods which each had delays of only one clock cycle between trigger and output; in the first pulse of this simulation, the readout occurs a full 11 cycles after the calculated arrival time of 9 clock cycles (i.e. the readout is on the 20\(^{\text{th}}\) cycle). Nonetheless, the pulse lasts just as long as the pulses of the other two methods. If the readout delay becomes too long and gets confused with other incoming pulses, it would be easy to add a few extra channels to the output in order to label or at the very least indicate the parity (say, mark it with a ‘0’ or ‘1’ to distinguish it from earlier/later pulse data) of the pulse in question in order to avoid such confusion.

Though the arming condition might seem to be a little too lax in that it might prematurely terminate the process before the calculation has finished, a little thought reveals that this can never be the case; the delayed pulse is taken from stored memory, and the attenuated one is simply a modification of the input pulse. The only real error this might lead to is a confusion in the association of the readout to the specific pulse as discussed above—unusually shallow pulses will not prematurely terminate the calculation since the CFD zero-crossing will occur before the end of the original pulse.
Elaboration and Simulation of Additional Data Analysis Procedures

A: Baseline Analyses

Baseline Averager

It would be nice to have a running average of the baseline signal both for purposes of comparison with the pulse heights and to check for any systematic changes in the baseline which might introduce error.

As mentioned above, we commence this process by storing all non-pulse signal values into a 1D array called ‘pulsarbasesave.’ The calculation would then be nothing more than an average of the values in this array were it not for two pesky problems; first, immediately after power-up the array is initialized to zero, and this will skew the average in an undesired way, and second, constantly having to run through the memory in order to find an average is computationally expensive. Our code thus might seem more complicated than necessary, but it eliminates these two hurdles:

```plaintext
if pulsar<multithreshold(0) then
    if sumcounter<=savescope+2 then
        pulssum:=pulsusum+pulsar;
        sumcounter:=sumcounter+1;
        baselineaverage:=pulsusum/sumcounter;
    end if;
    if sumcounter>savescope+2 then
        pulssum:=pulsusum+pulsarbasesave(0)-pulsarbasesave(savescope+1);
        baselineaverage:=pulsusum/savescope;
    end if;
    signalmean:=baselineaverage; --store for later
end if;

--(PARALLELIZE OUTPUT to OBLA)—

Code snippet 4: Abbreviated baseline average body
```

The initiation errors are thus mitigated by employing a running sum of the number of stored data points for the average as opposed to just using the total size of the array. Our method also optimized computation time in the n>savescope regime by simply keeping a running sum of the data in our storage array and adding the new data/subtracting the old data when appropriate. This reduces the need to constantly sum over the whole array into two simple addition and subtraction operations per cycle.

The initiation jitters seen in the following simulation are hence statistically relevant jitters associated with the distribution of high and low signal points, and these average out when the memory size is able to account for the periodicity of our given pulses. We are able to assert this with confidence
since the running averages in the initial regime always return to what is later seen to be the global average (here 9 volts; \(\approx 19/2\) truncated, as expected) whenever the amount of baseline data taken into account is precisely periodic (i.e. symmetric), which is unlikely to occur in the case of random memory initiation error:

![Graph showing running baseline average and pulse output.](image)

*Fig.5:* Running baseline average (volts) readout on bottom and pulse at top for reference; pulse peak height constant 19 volts, (threshold at 50, so we're in the background noise regime), saving last 1’000 data points (100’000ps)

Note that the output is continuous, and occurs only when the signal is below the first threshold, which we set artificially high here for purposes of demonstration; output shuts up when the threshold is breached and comes back online only when the pulse sinks back below the threshold.

**Baseline Standard Deviation**

Computing the standard deviation of the baseline signal requires computing the difference of the signal to the mean, squaring this, summing over all signal points in question, dividing by n-1, and finally computing the square root. All of this is very straightforward to implement in VHDL (especially given our baseline storage array instantiated in an earlier part of the program) save for the square root in the very last step. Though many algorithms exist, our challenge was to optimize the approach for the types of numbers we expect to be dealing with.

An iterative continued fraction algorithm developed by the author was found to have an optimal rate of convergence (to 8 iterations or less) for numbers between 1 and 20, and an attempt was made to map larger numbers into this range by diving by successive powers of 100 and multiplying back by as many factors of 10, but this very quickly made the code unmanageably complex. An easy modification to the algorithm was made such that any square root in the range 1 to \(2^{42}\) would converge (to within 6 decimal places) in 23 iterations or less; it was observed that the original algorithm often oscillated...
around its eventual convergent value, and thus it made sense to consider as input for the ith iteration not simply the i-1st partial sum, but rather the average of the i-1st and i-2nd partial sums!

The revamped algorithm unfortunately takes much longer to compute square roots of numbers less than 1, but since VHDL cannot handle decimal values this need not worry us, as any such square root would truncate down to 0.

The final algorithm, with (standarddevsum/(basecounter-1)) as the number the square root of which we seek, appears below:

```vhdl
for ib in 0 to 25 loop
    squirt:=(1+((standarddevsum)/(basecounter-1)-1)/(squirt+1))+squirt)/2;
    if ib=25 then
        standarddev:= squirt;
    end if;
end loop;
```

*Code snippet 5: Abbreviated baseline average body*

Standard deviation is rather more difficult to simulate that the other fitting parameters since standard deviations are difficult to compute by hand so that we might verify the code. We can however sneak up on our standard deviation from behind: for triangular pulses, the standard deviation should go roughly as the as half the average, which we’ve already computed above! We will reason for this heuristically for now—the average deviation on a triangular pulse from its mean should lie halfway between the mean and the top or bottom of the pulse, meaning that the standard deviation should be *roughly* half of the mean:

![Diagram of triangular pulse with mean, 25th percentile, and 50th percentile highlighted]

And as we see in the following simulation with constant pulse peak height 19 (threshold artificially inflated again), this is precisely what the code gives us:
Fig.6: Running baseline average (volts) readout on bottom and pulse at top for reference; pulse peak height constant 19 volts, (threshold at 50, so we’re in the background noise regime), saving last 1’000 data points (100’000ps)

Ignoring the initial jitters due to nothing more than a small sample size, the code gives us a long-term average of 9 (19/2 truncated (19 is peak height of pulses)) and a standard deviation of 5 (almost exactly 9/2).

Further simulations have provided results as expected.

**Baseline Significance**

Significance is merely the peak deviation from the mean divided by the standard deviation. The latter was verified above, and our peak deviation algorithm is identical to the pulse peak one implemented and verified below. We will thus not go over the code or simulation here, since both are sufficiently straightforward. All simulations thus far using the current code have provided results as expected

**Number of Electrons**

The algorithm for this requires no more fancy footwork than a simple integral sum of the pulse, divided by the number of electrons per volt second. That our integral calculations are sound will be demonstrated in the following section, and we will not include a full simulation here since the exact calibrations are not available at this time.
Integral calculator/average height

The integral algorithm is as simple as it should be; a running sum of the signal values above the arming threshold is stored for each pulse, and the integral is re-initialized to zero to allow us to analyze subsequent pulses. The average height computation merely records the times of the respective threshold crossings (rising and falling edges) and divides the integral by the total time. Output is issued at the falling edge threshold crossing, again in 12bit parallel.

Simulating this with triangular pulses provides us with an extremely quick way to validate the code, since averages are readily computed with mental math (average ought to be exactly halfway between the peak and the arming threshold). The following simulation increases the pulse height by one volt for each pulse, and, save for a truncation error that occurs when dividing an odd number by 2, the results are as we anticipate:

![Image of integral calculator/average height](image)

Fig.7: Average pulse height (volts) readout on bottom, with multi-threshold and pulse readouts above for reference; trigger thresholds at 7, 9, and 12 volts, peak pulse heights 26-31 increasing from left to right, saving at most last 1'000 data points (100’000ps)

As an illustrative calculation, we can consider the first pulse in the above figure; the pulse peaks at 26 volts, and it is triggered at 7 volts, so the average ought to be 33/2= 16.5. Our readout tells us an average of $2^{14}=16$, which is exactly what we were looking to within the unavoidable truncation error.

As we might expect given the testbench, the average heights are seen to increase by one for every two pulses, corresponding to a total pulse peak increase of 2volts for every two pulses.

Peak finder

A simple algorithm comes to the rescue once again; once the calculation is triggered by a pulse value over the threshold, variable ‘peak’ stores the highest value thus far, which is overwritten by any higher peak values. The peak time is stored for later use.
If pulsar>=multithreshold(0) then
  if pulsar>peak then
    peak:=pulsar
    peaktime:=accsec;
  end if;
end if;

Code snippet 6: Heart of the pulse peak finder

As before, readout is issued at the threshold crossing on the falling edge of the pulse:

![Plot of a pulse wave form with annotations]

Fig.8: Peak pulse height (volts) readout on bottom, with multi-threshold and pulse readouts above for reference; trigger thresholds at 7, 9, and 12 volts, peak pulse heights 25-30 increasing from left to right, saving at most last 1’000 data points (100’000ps)

Our testbench increases the peak pulse height by one volt with each passing pulse, and indeed we can read off the heights from our peak finder – 25 to 30 – by simply converting binary to decimal.

**Rising and falling times**

This calculation cannot be reliably done using only incoming pulse data; some kind of pulse memory must be implemented. Our code references the variables time1 and time2, which were recorded in the integral function above, and then searches through the memory starting at the peak and then moves to the right and left in search of the two times (rising and falling) at which the pulse is first less than or equal to 90% of its peak value. The rising and falling algorithms are very nearly identical, so we will only show the former here:
for z in (time2-peaktime) to (time2-time1) loop
if z>savescope then     --program prone to crashing without this safety check
    risetime:=0;
    exit;
elsif pulsarsave(z)<= 9*peak/10 then
    risetime:=(time2-time1)-z;
    exit;
end if;
end loop;

Code snippet 7: Algorithm for computing rise time

The key trick here is the fact that, since the storage array ‘pulsarsave’ is updated once per clock cycle, the relative position of data within the array gives us important timing information; thus (time2-time1) gives us the total longevity of the pulse, and z provides the time when the pulse is 90% of its peak as measured by definition from time 2, so the difference between these two numbers gives us the desired rising time:

![Image](image_url)

Fig.9: Rising time (in ps) readout on bottom, with multi-threshold and pulse readouts above for reference; trigger thresholds at 7, 9, and 12 volts, peak pulse heights 25-30 increasing from left to right, saving at most last 1’000 data points (100’000ps)

The simulation nicely reflects the fact that our peak heights are increasing linearly, as a close inspection of fig.7 reveals the rising times to increase linearly as well, as expected. This pattern is sustained throughout the remainder of the simulation, while the results are almost identical to those of the falling time computations.
Full Width at Half Maximum

The algorithm here is in essence identical to that given above for the rising and falling times; the memory is probed from right and left starting at the peak value until the pulse reaches half of its maximum. The difference in the storage locations of these two data points is computed (since storage is updated once per clock cycle), giving the time between the two points and thus the full width at half maximum:

![Image](image.png)

*Fig.10:* FWHM readout (in ps) on bottom, with multi-threshold and pulse readouts above for reference; trigger thresholds at 7, 9, and 12 volts, peak pulse heights 25-30 increasing from left to right, saving at most last 1'000 data points (100'000ps)

Since our pulse heights increase by one volt per pulse, we expect the FWHM to increase by one clock cycle for every two pulses (due to truncation error); this is precisely what the above simulation illustrates.

More Exotic Simulations

Timing Analysis

Though all of the above analyses were carried out with a wide array of different variations on the repeated triangular pulse idea (constant peak height, linearly increasing peak, linearly increasing width, artificially shallow peaks, etc.), it is of course possible that the code might somehow be biased towards such pulses and thus that it might not be robust enough to handle other shapes. To have a go at excluding this possibility, we will execute a few simulations with a more ‘natural’ looking pulse, namely a single Gaussian.
As can be seen in the following figure, the multi-threshold and leading edge capabilities remain intact:

Fig.11: A Gaussian curve top, with multi-threshold and leading edge discriminators below, respectively. Pulse peak height 1200 volts, thresholds at 20, 40, and 75 volts

Though the CFD parameters of course need to be optimized to the pulse shape, the following figure confirms that we are at the very least still getting output from the CFD calculations. Shown below these in the same figure are the results from the peak finder, which read at 1199 volts – quite satisfying considering the actual peak height of 1200 volts and taking truncation error into account:

Fig.12: A Gaussian curve top, with CFD and pulse finder below, respectively. Pulse peak height 1200 volts, thresholds at 10, 20, and 30 volts. Delay time for first pulse of CFD is 4 clock cycles, with 30% pulse attenuation for the other (see CFD method explained above).
Changing the Thresholds

All threshold initializations are located on lines 104-106, in the perambulatory clauses before the body of the above algorithms:

```
multithreshold(0)<=50
multithreshold(1)<=75;
multithreshold(2)<=100;
```

Simply modify the above numbers for the first, second, and third thresholds respectively.

Changing the Timing

The timing is currently conducted by a variable (accsec) which increases by one at each clock rising edge. Picoseconds are, of course, a relatively short amount of time, and so this variable is prone to overloading. A quick fix would be something along the lines of the following, in which a certain number of seconds is defined to equal a ‘minute’:

```
begin
  if clk'event and clk='1' then --program loops at clock rising edge
    accsec<=(accsec+1) mod 10000000; --Clock
    if accsec=9999999 then
      accminute<=accminute+1; --"minute" counter
    end if;
  end if;
  --...code body here...
end if;
end process;
```

Using this time would, of course, require us to record both minutes and seconds in our calculations, but this might be necessary, since, if the variable size is capped at $2^{42}$, the code will crash after about 4 seconds ($2^{42}/10^{12}$), assuming clock pulse changes once per picosecond.

Changing global constants

Constants savescope, delay (for CFD), and electroncharge are initialized in the variable declaration section. Just in case, you know, the charge of the electron spontaneously decides to change.
Bibliography


library ieee;
use ieee.std_logic_1164.all;

entity Signalprocessing_andthensome is
    port(
pulse: in std_logic_vector (0 to 11); --assumed that
ASIC readout is in parallel(?), and if not it is easy to parallelize serial input
    clk: in std_logic;
        O: out std_logic_vector (0 to 11); --Output for multithreshold mode
        OL: out std_logic_vector (0 to 11); --Output for Linear regression timing calculation
        OCFD: out std_logic_vector (0 to 11); --Output for const. fraction disc. timing calculation
        OBLA: out std_logic_vector (0 to 11); --Output baseline average
        OAVG: out std_logic_vector (0 to 11); --Output average height of pulse
        OPEAK: out std_logic_vector (0 to 11); --Dedicated output for peak of pulse
        OTEST: out std_logic_vector (0 to 11)); --for simulating miscellaneous calculations
end Signalprocessing_andthensome;

architecture Herjolfssen4 of Signalprocessing_andthensome is
    -- thresh-> single threshold,
    -- multithresh-> what it sounds like, cfract...-> continuous fraction discriminator, v-> what did we say about the man behind the curtain?
    type tok is (thresh, multithresh, cfractdiscrim, v);
    signal sitka: tok:=thresh;
    --Sitka specifies mode of calculation --uncomment if case statement used to specify mode of timing analysis
    signal acc: integer:=0;
    --our stopwatch
    signal accsec: integer :=0;
    --"seconds" or any other arbitrary time unit, to be adjusted based on the frequency of clock used.
    type gaul is array(natural range <>) of integer; --define unconstrained array
    type signal multithreshtime: gaul (1 to 3);
    --individual threshold-breaking times for the multithreshold mode
signal pulsebit: gaul (0 to 11);  
--pulse is converted later to an integer for calculation purposes; this is an intermediate step
signal multithreshold: gaul (0 to 2);
--stores our three constant threshold values for the multithresh mode

begin

process(clk)
variable baselineaverage: integer:=0;  -- outputs to OBLA (Output Base Line Average)
variable pulsesum: integer :=0;  --for baseline averager, stores sum of non-important
background data
variable ymean: integer;
--mean of thresholds (for linear regression)
variable xmean: integer;  --mean of times of threshold crossing
variable ssxx: integer:=0;
--sum of squares, for use in linear regression calculation
variable ssxy: integer:=0;  --ditto
variable pulsar: integer:=0;
-- intergized value of incoming pulse
constant savescope: integer:=1000;
--scope size of CFD delay memory and base line averager (here, last 1000 data points)
variable pulsarsave: gaul (0 to savescope+4);  
--memory of last savescope non-pulse reading heights
variable time1: integer:=0;  --in integral calculator, time of first threshold crossing
variable time2: integer:=0;  --"" time of falling below threshold
variable timeA: integer:=0;  --time of baseline average crossing
variable timeB: integer:=0;  --time when pulse returns below baseline average
constant delay: integer:=10;  --delay length for CFD (150-200 ps = 4 using our clock specs (see testbench))
constant electroncharge: integer:=16;  --no support for floating types... :'(  
variable eventprint: gaul (0 to 4);  
--array output, listing arrival time against event number
variable eventprintread: gaul (0 to 3);  --"scratchpaper" variable
variable inhibit: integer:=0;  --negative feedback
variable inhibit_cfd: integer:=0;  --ditto
variable threeyep: integer:=0;  --honk if threshold three has been breached
variable linfitzero: integer:=0;  -- time data from linear regression, to be read out to OL(0 to 11) in binary
variable peak: integer:=0;  -- height of the peak
variable basecounter: integer:=0;  --numbers the initial data for early baseline averaging
variable integralsum: integer:=0;  --the actual integral
variable intaverage: integer:=0;  --outputs to OAVG; average height of pulse
variable signalmean: integer:=0;  --for calculating number of electrons
variable Integralsumstore: integer:=0;  --stores integral using mean background crossings as indices
variable neelectrons: integer:=0;  --number of incident elections in a pulse
variable peaktime: integer:=0; --time at which peak occurs
variable risetime: integer:=0; --time pulse takes to rise from first threshold to 90% of peak
variable falltime: integer:=0; --time pulse takes to fall from 90% of first threshold down to first threshold
variable FWHM: integer:=0; --full width at half maximum
variable halffronttime: integer:=0; --for FWHM calculation; absolute time at which pulse rises to reache half its maximum
variable halfbacktime: integer:=0; --" " falls " "
variable standarddev: integer:=0; -- standard deviation of baseline
variable standarddevsum: integer:=0; -- a running sum of stored baseline data (last savescope # of points) minus baseline mean all squared
variable peakdeviation: integer:=0; --largest deviation from mean of baseline
variable significance: integer:=0; --largest deviation of baseline devided by its standard deviation
variable squirt: integer:=0; --a dummy variable whihc facilitates square root calculation using a continued fraction algorithm
variable tester: integer:=0;
variable sumcounter: integer:=0;

-- ==|==|==|==|==|==|==|==|==|==|==|

begin
if clk'event and clk='1' then --program loops at clock rising edge
  pulsar:=0; --our oscilloscope data, reinitialized to allow for sum in integerizing calculation below
  accsec<=accsec+1; --out stopwatch

  for j in 0 to 11 loop -- intigerizing the input pulse; logic vector to binary array to integer(pulsar)
    if pulse(j)='1' then
      pulsebit(j)<=1; --binary array
    elsif pulse(j)='0' then
      pulsebit(j)<=0;
    end if;
    pulsar:= pulsar+(pulsebit(j))*(2**(j)); --actual binary to integer conversion
  end loop;

  multithreshold(0)<=50; --specify desired threshold values for multithreshold calculations
  multithreshold(1)<=75;
  multithreshold(2)<=100;

  if pulsar<multithreshold(0) then --initialize output to 0
for t in 11 downto 0 loop
    O(t)<='0';
    OL(t)<='0';
    OCFD(t)<='0';
    inhibit_cfd:=0;
end loop;
end if;

if ((inhibit=1 or threeyep=1 or inhibit_cfd=1 or pulsar<multithreshold(0)) and
pulsarsave(0)=pulsar)then --shut up output (supposing there is some) if pulse changes
    for s in 0 to 11 loop
        O(s)<='0'; --reset in case new reading will be needed soon
        OL(s)<='0';
        OCFD(s)<='0';
        OBLA(s)<='0';
        OAVG(s)<='0';
        OPEAK(s)<='0';
        -- OTEST(s)<='0';
    end loop;
    threeyep:=0; --reset to allow continued data acquisition
    ssxx:=0;
    ssxy:=0;
    inhibit:=0;
    FWHM:=0;
end if;

for k in savescope downto 1 loop --pulse saver, for use in averaging and delaying pulse
    pulsarsave(k):=pulsarsave(k-1); --shift data over one bin in pulsarsave
end loop;

pulsarsave(0):=pulsar; --stick new incoming data in bin 0

if pulsar<multithreshold(0) then
    basecounter:=basecounter+1; --counts the baseline data by assigning each data point a cardinal number
for m in (savescope+2) downto 1 loop  --flesh out the storage array
  pulsarbasesave(m):=pulsarbasesave(m-1);
end loop;

pulsarbasesave(0):=pulsar;  --store baseline output in array
end if;

-- MULTITHRESHOLD
-- ensures only rising edges are counted
if pulsar>=multithreshold(0) and inhibit=0 and
  pulsarsave(1)<multithreshold(0) then  --using lowest threshold as the trigger
  eventprint(0):=accsec;
  --time stored in eventprint
  eventprintread(0):=eventprint(0);
  for q in 11 downto 0 loop
    if eventprintread(0)>=(2**q) then
      O(q):='1';
      eventprintread(0):=eventprintread(0)-2**q;
    else
      O(q):='0';
      end if;
    if q=1 and eventprintread(0)=1 then
      O(0):='1';
      end if;
    if q=0 then
      inhibit:=1;
      end if;
  end loop;
end if;

--12121212121212121212121212121212--

if pulsar>=multithreshold(1) and inhibit=0 and pulsarsave(1)<multithreshold(1) then  --threshold 2
  eventprint(1):=accsec;  --time stored in eventprint array
  eventprintread(1):=eventprint(1);
  for q in 11 downto 0 loop
    if eventprintread(1)>=(2**q) then
      O(q):='1';
      eventprintread(1):=eventprintread(1)-2**q;
    end loop;
end if;
else
    O(q)='0';
end if;
if q=1 and eventprintread(1)=1 then
    O(0)='1';
end if;
if q=0 then
    inhibit=1;
end if;
end loop;
end if;

--2323232323232323232323232323232323--

if pulsar>=multithreshold(2) and inhibit=0 and
pulsarsave(1)<multithreshold(2) then
  --threshold 3
  threeyep=1;
  eventprint(2)=accsec;
  --time stored in eventprint array
  eventprintread(2)=eventprint(2);

for q in 11 downto 0 loop
  if eventprintread(2)>=(2**q) then
    O(q)='1';
    eventprintread(2)=eventprintread(2)-2**q;
  else
    O(q)='0';
  end if;
end loop;
end if;

--VNNVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVMVN--

-- || || || || || || || || || || || || || || || || || || || --

if threeyep=1 then
  --LINEAR FIT
  xmean:=(eventprint(0) + eventprint(1) + eventprint(2))/3
  --multiply by thousand to reduce truncation error
ymean:= (1000*multithreshold(0) + 1000*multithreshold(1) + 1000*multithreshold(2))/3;
-- def of ymean

for i in 0 to 2 loop

-- LINEAR FIT: y = a + bx -> b=ssxy/ssxx, (see http://mathworld.wolfram.com/LeastSquaresFitting.html)
ssxx:=ssxx + (eventprint(i)-xmean)**2;

--ssxx=sum(xi^2-xmean^2), n=3
ssxy:=ssxy + ((-1)*xmean*ymean + 1000*eventprint(i)*multithreshold(i));
--ssxy=sum(xi*yi-xmean*ymean)
end loop;

-- linear fit b=ssxy/ssxx, a=ymean-b*xmean

linfitzero:=xmean - (ymean*ssxx)/ssxy;
--f(x)= a + bx -> x=-a/b for f(x) = 0, and -a/b = xmean-ymean/b

for u in 11 downto 0 loop
if linfitzero>(2**u) then
   OL(u)=='1';
   linfitzero:=linfitzero-2**u;
else
   OL(u)=='0';
end if;
end loop;

end if;
-- \_____________________________________________________________________/ --

\_____________________________________________________________________/ --

if (pulsar>multithreshold(0)) and inhibit_cfd=0 then
   -- first threshold "arms" the Constant Fraction Discrimination calculation
   -- multiply by ten to reduce truncation error
   if ((-30*pulsar/10+10*pulsarsave(delay)) > 0) then
      -- pulse attenuated to 30% (=3/10), inverted, and added to the delayed pulse (2000 indicates length of delay, to be adjusted based on actual clock freq)
eventprint(3):=accsec-delay;        --time of zerocrossing recorded (delay time subtracted)
end if;

for v in 11 downto 0 loop
  if eventprint(3)>=(2**v) then
    OCFD(v)<='1';
    eventprint(3):=eventprint(3)-2**v;
  else
    OCFD(v)<='0';
  end if;
  if v=0 then
    inhibit_cfd:=1;
  end if;
end loop;
end if;

--_/__/___/____/_____/______/_______/________/_________/__________/___________/
--____________/_____________/______________/_______________/
--<<(([]]))><<<(([]]))><<<(([]]))><<<(([]]))><<<(([]]))><<<(([]]))><<<(([]]))>--> 

if pulsar<multithreshold(0) then --BASELINE AVERAGER
  if sumcounter<=savescope+2 then
    pulsesum:=pulsesum+pulsar;
    sumcounter:=sumcounter+1;
    baselineaverage:=pulsesum/sumcounter;
  end if;
  if sumcounter>=savescope+2 then
    pulsesum:= pulsesum+pulsarbasesave(0)-pulsarbasesave(savescope+1);
    baselineaverage:=pulsesum/savescope;
  end if;
  signalmean:=baselineaverage; --store for later
  for w in 11 downto 0 loop --readout in parallel
    if baselineaverage>=(2**w) then
      OBLA(w)<='1';
      baselineaverage:=baselineaverage-2**w;
    else
      OBLA(w)<='0';
    end if;
  end loop;
end if;

end loop;

end if;

--->

if sumcounter<=savescope then
  -- initialization error mitigation
  for zz in basecounter-1 downto 0 loop
    standarddevsum:= standarddevsum+(abs(pulsarbasesave(zz)-signalmean)**2; -- the usual summing-difference-from-mean-and-squaring process
    if zz=0 then
      squirt:= standarddevsum/2; -- just a rough estimate to speed convergence of continued fraction
      if basecounter = 1 then -- avoid catastrophic division by zero
        standarddev:=0;
      else
        for ib in 0 to 100 loop
          squirt:= ((1+(standarddevsum)/(basecounter-1)-1)/(squirt+1)+squirt)/2;
        end if;
        end if;
      end if;
    end if;
  end loop;
end if;

if sumcounter>savescope then
  for zz in savescope downto 0 loop
    standarddevsum:= standarddevsum+(abs(pulsarbasesave(zz)-signalmean)**2; -- the usual summing-difference-from-mean-and-squaring process
    if zz=0 then
      squirt:= standarddevsum/2; -- just a rough estimate to speed convergence of continued fraction
      for ib in 0 to 100 loop
        squirt:= ((1+(standarddevsum)/(savescope-1)-1)/(squirt+1)+squirt)/2;
      end if;
      end if;
    end if;
  end loop;
end if;
end if;
end loop;
end if;

--SIGNIFICANCE CALCULATION
peakdeviation:=signalmean;  -- peakdeviation reinitialized to mean to eliminate layover from previous calculation(s)
for z in sumcounter-1 downto 0 loop
  -- significance is only to within the savescope, but this can be changed according to specs
  if (abs(pulsarbasesave(z)-signalmean)>peakdeviation)then
    peakdeviation:=pulsarbasesave(z);  -- HOLEY MOLEY! we have a winner!
  end if;
  if z=0 then
    if standarddev=0 then
      significance:=0;
    else
      significance:=peakdeviation/standarddev;  -- when done running through the memory, calculate significance
    end if;
  end if;
end loop;

if pulsar>=multithreshold(0) then
  -- PEAK FINDER
  if pulsar>peak then  -- find largest pulsar
    peak:=pulsar;  -- and store it in "peak"
    peaktime:=accsec;
  end if;
end if;

if pulsar<multithreshold(0) and pulsarsave(1)>=multithreshold(0) then
  -- when pulse goes below 1st threshold
  for y in 11 downto 0 loop
    -- readout in parallel
    if peak>=(2**y) then
      OPEAK(y)<'1';
      peak:=peak-2**y;
      -- automatically reinitializes 'peak' to zero!!!
    else
      OPEAK(y)='0';
    end if;
  end loop;
end if;
if pulsar >= multithreshold(0) then

--INTEGRAL CALCULATOR-OUTPUTS AVERAGE HEIGHT

if pulsarsave(1) < multithreshold(0) then
  --store time of first threshold-crossing
  time1 := accsec;
end if;

integralsum := integralsum + pulsar;  --sum all data points
end if;

if pulsar <= multithreshold(0) and pulsarsave(1) > multithreshold(0) then
  time2 := accsec;
  intaverage := (integralsum / (time2 - time1));
  integralsum := 0;  --prepare for next calculation
end if;

for x in 11 downto 0 loop  --readout in parallel
  if intaverage >= (2**x) then
    OAVG(x) := '1';
    intaverage := intaverage - 2**x;
  else
    OAVG(x) := '0';
  end if;
end loop;
end if;

if pulsar >= signalmean then  --NELECTRONS

if pulsarsave(1) < signalmean then  --store time of deviation from mean
  timeA := accsec;
end if;

integralsumstore := integralsumstore + pulsar;  --sum all data points
end if;
if pulsar <= multithreshold (0) and pulsarsave(1)>multithreshold(0) then --if signal dips below mean...
    timeB:=accsec; --store that time
    nelectrons:=(integralsumstore*(time2-time1))/(50*(electroncharge/10**(20)));
    --take integral and divide by electron charge
    integralsumstore:=0;
end if;

if pulsar <= multithreshold(0) and pulsarsave(1)>multithreshold(0) then --GET RISING AND FALLING TIMES
    time2:=accsec;
    for z in (time2-peaktime) downto 0 loop
        if z>savescope then --program prone to crashing without this safety check
            falltime:=0;
            exit;
        elsif pulsarsave(z)<=9*peak/10 then
            falltime:=z;
            exit;
        end if;
    end loop;

    for z in (time2-peaktime) to (time2-time1) loop
        if z>savescope then --program prone to crashing without this safety check
            risetime:=0;
            exit;
        elsif pulsarsave(z)<=9*peak/10 then
            risetime:=(time2-time1)-z;
            exit;
        end if;
    end loop;
end if;

if pulsar <= multithreshold(0) and pulsarsave(1)>multithreshold(0) then --GET FWHM TIME
    for z in (accsec-peak) downto 0 loop
        if z>savescope then --program prone to crashing without this safety check
            halfbacktime:=0;
            exit;
        end if;
    end loop;
end if;
elsif pulsarsave(z)= 5*peak/10 then
    halfbacktime:=time2-z;
    exit;
end if;
end loop;

for z in (accsec-peaktime) to (time2-time1) loop
    if z>savescope then --program prone to crashing without this safety check
        halffronttime:=0;
        exit;
    elsif pulsarsave(z)= 5*peak/10 then
        halffronttime:=time2-z;
        exit;
    end if;
end loop;

FWHM:=halfbacktime-halffronttime;
end if;

tester:=standarddev;
    for x in 11 downto 0 loop --readout in parallel
        if tester>=(2**x) then
            OTEST(x)<'1';
            tester:=tester -2**x;
        else
            OTEST(x)<'0';
        end if;
    end loop;

end if;
end process;
end Herjolfssea4;
Appendix B: A simple test bench: triangular pulses with linearly increasing peak height

LIBRARY IEEE;
LIBRARY MAXII;
USE IEEE.STD_LOGIC_1164.ALL;
USE MAXII.MAXII_COMPONENTS.ALL;

-- Set up this testbench as an entity
entity test_signalprocessing_andthensome is
describe
end entity test_signalprocessing_andthensome;

-- Create an implementation of the entity
-- (May have several per entity)
architecture testbench5 of test_signalprocessing_andthensome is

-- Set up the signals on the 3bit_counter
signal clk : std_logic;
signal pulse: std_logic_vector (0 to 11);
signal O: std_logic_vector (0 to 11);
signal OL: std_logic_vector (0 to 11);
signal OCFD: std_logic_vector (0 to 11);
signal OBL: std_logic_vector (0 to 11);
signal OAVG: std_logic_vector (0 to 11);
signal OPEAK: std_logic_vector (0 to 11);
signal OTEST: std_logic_vector (0 to 11);

begin

-- dut = device under test (same name as top project from Quartus)
dut : entity work.Signalprocessing_andthensome

-- Map the ports from the dut to this testbench
port map ( clk => clk,
            O(0)=>O(0),
            O(1)=>O(1),
            O(2)=>O(2),
            O(3)=>O(3),
            O(4)=>O(4),
            O(5)=>O(5),
            O(6)=>O(6),
            O(7)=>O(7),
            O(8)=>O(8),
            O(9)=>O(9),
            O(10)=>O(10),
            O(11)=>O(11),
            OL(0)=>OL(0),
            ...)
OL(1) => OL(1),
OL(2) => OL(2),
OL(3) => OL(3),
OL(4) => OL(4),
OL(5) => OL(5),
OL(6) => OL(6),
OL(7) => OL(7),
OL(8) => OL(8),
OL(9) => OL(9),
OL(10) => OL(10),
OL(11) => OL(11),
OCFD(0) => OCFD(0),
OCFD(1) => OCFD(1),
OCFD(2) => OCFD(2),
OCFD(3) => OCFD(3),
OCFD(4) => OCFD(4),
OCFD(5) => OCFD(5),
OCFD(6) => OCFD(6),
OCFD(7) => OCFD(7),
OCFD(8) => OCFD(8),
OCFD(9) => OCFD(9),
OCFD(10) => OCFD(10),
OCFD(11) => OCFD(11),
OBLA(0) => OBLA(0),
OBLA(1) => OBLA(1),
OBLA(2) => OBLA(2),
OBLA(3) => OBLA(3),
OBLA(4) => OBLA(4),
OBLA(5) => OBLA(5),
OBLA(6) => OBLA(6),
OBLA(7) => OBLA(7),
OBLA(8) => OBLA(8),
OBLA(9) => OBLA(9),
OBLA(10) => OBLA(10),
OBLA(11) => OBLA(11),
OAVG(0) => OAVG(0),
OAVG(1) => OAVG(1),
OAVG(2) => OAVG(2),
OAVG(3) => OAVG(3),
OAVG(4) => OAVG(4),
OAVG(5) => OAVG(5),
OAVG(6) => OAVG(6),
OAVG(7) => OAVG(7),
OAVG(8) => OAVG(8),
OAVG(9) => OAVG(9),
OAVG(10) => OAVG(10),
OAVG(11) => OAVG(11),
OPEAK(0) => OPEAK(0),
OPEAK(1) => OPEAK(1),
OPEAK(2) => OPEAK(2),
OPEAK(3) => OPEAK(3),
OPEAK(4) => OPEAK(4),
OPEAK(5) => OPEAK(5),
OPEAK(6) => OPEAK(6),
OPEAK(7) => OPEAK(7),
OPEAK(8) => OPEAK(8),
OPEAK(9) => OPEAK(9),
OPEAK(10) => OPEAK(10),
OPEAK(11) => OPEAK(11),
OTEST(0) => OTEST(0),
OTEST(1) => OTEST(1),
OTEST(2) => OTEST(2),
OTEST(3) => OTEST(3),
OTEST(4) => OTEST(4),
OTEST(5) => OTEST(5),
OTEST(6) => OTEST(6),
OTEST(7) => OTEST(7),
OTEST(8) => OTEST(8),
OTEST(9) => OTEST(9),
OTEST(10) => OTEST(10),
OTEST(11) => OTEST(11),
pulse(0) => pulse(0),
pulse(1) => pulse(1),
pulse(2) => pulse(2),
pulse(3) => pulse(3),
pulse(4) => pulse(4),
pulse(5) => pulse(5),
pulse(6) => pulse(6),
pulse(7) => pulse(7),
pulse(8) => pulse(8),
pulse(9) => pulse(9),
pulse(10) => pulse(10),
pulse(11) => pulse(11));

-- Set up the signals
stimulus : process is
  variable gosh: integer := 0;
  variable pulser: integer := 0;
  variable pulserarith: integer := 0;
type direction is (up, down);
  variable move: direction := up;
  variable kjuin: integer := 25;
begin
  -- Just make a clock

  loop
    if pulser = kjuin then
      move := down;
    end if;
  end loop;
elsif pulser=1 then
  move:=up;
  kjuin:=kjuin+1;
end if;

case move is
  when up=>
    pulser:=pulser+1;
  when down=>
    pulser:=pulser-1;
end case;

  pulserarith:=pulser->kjuin;
  for q in 11 downto 0 loop
    if pulserarith > 2**q then
      pulse(q)='1';
      pulserarith:=pulserarith-2**q;
    else
      pulse(q)='0';
    end if;
  end loop;
  clk <= '0'; wait for 50 ps;
  clk <= '1'; wait for 50 ps;
  clk <= '0'; wait for 50 ps;
  clk <= '1'; wait for 50 ps;
  clk <= '0'; wait for 50 ps;
  clk <= '1'; wait for 50 ps;
end loop;

end process stimulus;
end architecture testbench5;