Compact Muon Solenoid (CMS)
Front End Driver (FED)
Front-End FPGA

Technical Description

Written by: Bill Gannon
Supervisor: Rob Halsall
Version: 1.1
Printed: 16:15 11 March 2002

CHANGES

Version	Changes	Section(s) affected
0	Pre-release version.	
1.0	Added more configuration commands.	4.1
1.1	Re-wrote the Housekeeping section to reference the CMSdelay chips.	Fig 2-1 2.1
	Re-wrote the Autocalibration section to reflect recent design changes. (Now average every 70^{th} clock-cycle, not every 5^{th}. Coarse Calibration performed used calRAM.)	2.2
	Edited section 3 to reflect recent changes. (Output FIFO now 2048x8. Output of FPGA is 8-bit. Processed Raw data does not go through Cluster Finder.)	3
	Edited main pinout to reflect recent changes. (**command_out** and **busy** signals to/from CMSdelay chips. Output is 8-bit. **monitor_in** and **monitor_out** removed).	1.2
	Updated descriptions of old and added new commands.	4.1
	Added new section on Output Data Formats. Generalised description of Output FIFO Controller	4.3 3.4
	Spy Data now 1022 words (not 1024)	4.1.6
	Monitor Command now 157 bits long (not 156)	4.1.18

1. INTRODUCTION

The Compact Muon Solenoid (CMS) is one of two large experiments that make up the Large Hadron Collider (LHC) at CERN. The CMS detector includes 10 million channels of Silicon microstrip, which are read out through 100,000 APV (Analogue Pipeline - Voltage) chips. These in turn are readout through APVMUX chips, along 50,000 fibre optic cables, to 450 Front-End Driver (FED) cards. Inside the FED card the optical signals are digitised to 10-bits at 40MHz and passed to 8 Front-End FPGAs, each of which processes 12 fibres.

Inside the Front-End FPGA the data is conditioned and processed, and the significant information (the Hit Clusters) extracted. These are then readout via a Back-End FPGA to a Data Processing Farm which performs the Final Data Analysis. The average data-flow of the system is targeted to be 1.1 tera-words per second (70% of maximum possible) at the input, which will (hopefully) be reduced to about 40Gbytes/sec at the output.

1.1 Scope

This document describes the technical aspects of the Front-End FPGA, specifically how it operates, how it meets the requirements, and some information about design choices along the way. It is designed to be a record of the design for anyone wanting to know more about how the FPGA works, and to be read by anyone charged with modifying or augmenting the Source Code.

The FPGA Source Code is written in Verilog. This is synthesized to EDIF using Exemplar Leonardo. The Place and Route is performed using Xilinx Design Manager. The target family is the Xilinx Virtex II. The target device is the XCV2M (2000 kilo-gates equivalent).

1.2 Brief Description

A Block Diagram showing roughly how the Front-End FPGA is laid out is shown below.

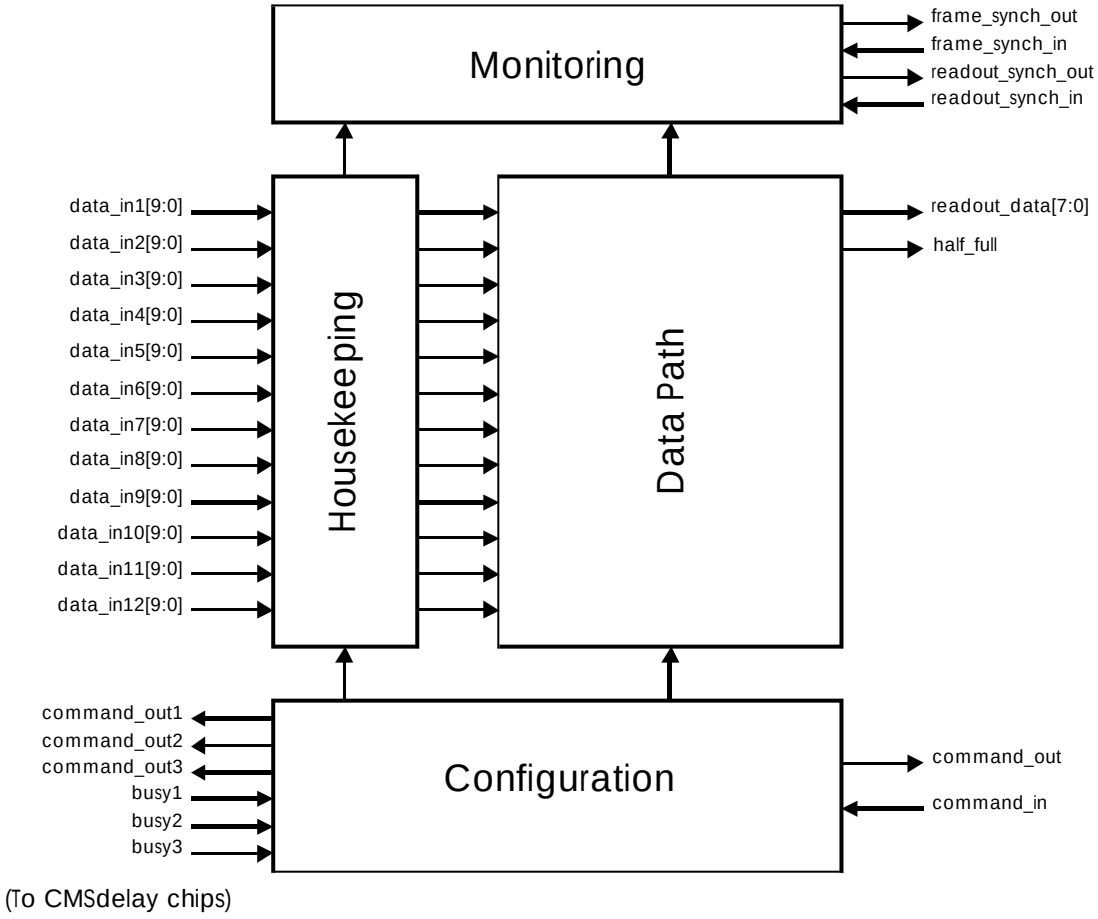

Figure 1-1 Block Diagram of the Front-End FPGA

This diagram shows how the twelve 10-bit inputs (one for each fibre) pass through a Housekeeping Block before being processed by the Data Path. The Housekeeping Block controls the synchronisation of the fibres, calculates the thresholds, and looks for the arrival of Physics Data. Once triggered, the Data Path processes the Physics Data for all 12-channels at once, extracts the relevant information and stores it, prior to readout through a single 8-bit link.

The Back-End I/O of the FPGA is fairly restricted, and in practice will be multiplexed down to about 10 wires. This is in order to reduce the amount of I/O the Back-End FPGA needs to connect to all eight Front-End FPGAs. This star-configuration also means that the loss of one Front-End FPGA does not bring down the whole card, and permits maximum data-transfer bandwidth.

The Configuration and Monitoring of the 12-channels is itself a complex task, and we hope that the design strikes the correct balance between flexibility, simplicity, tolerance (involving graceful failure and auto-recovery), ease-of-use and diagnostic ability. Each of these Main Functional Blocks shall now be discussed in more detail over the following sections.

2. HOUSEKEEPING

In order for the Data Path to operate, various tasks must be performed first to identify:

- the precise relative timing of each fibre
- the threshold to use for Tick-Mark and Frame Finding

in addition whilst the FED is receiving data it must be able to sense:

- when Frames arrive
- when faults develop

These tasks can collectively be given the title ‘Housekeeping’ since they help to keep the FED running smoothly. A Block Diagram showing the CMSdelay chips and Housekeeping components for all 12 channels is shown on the next page:

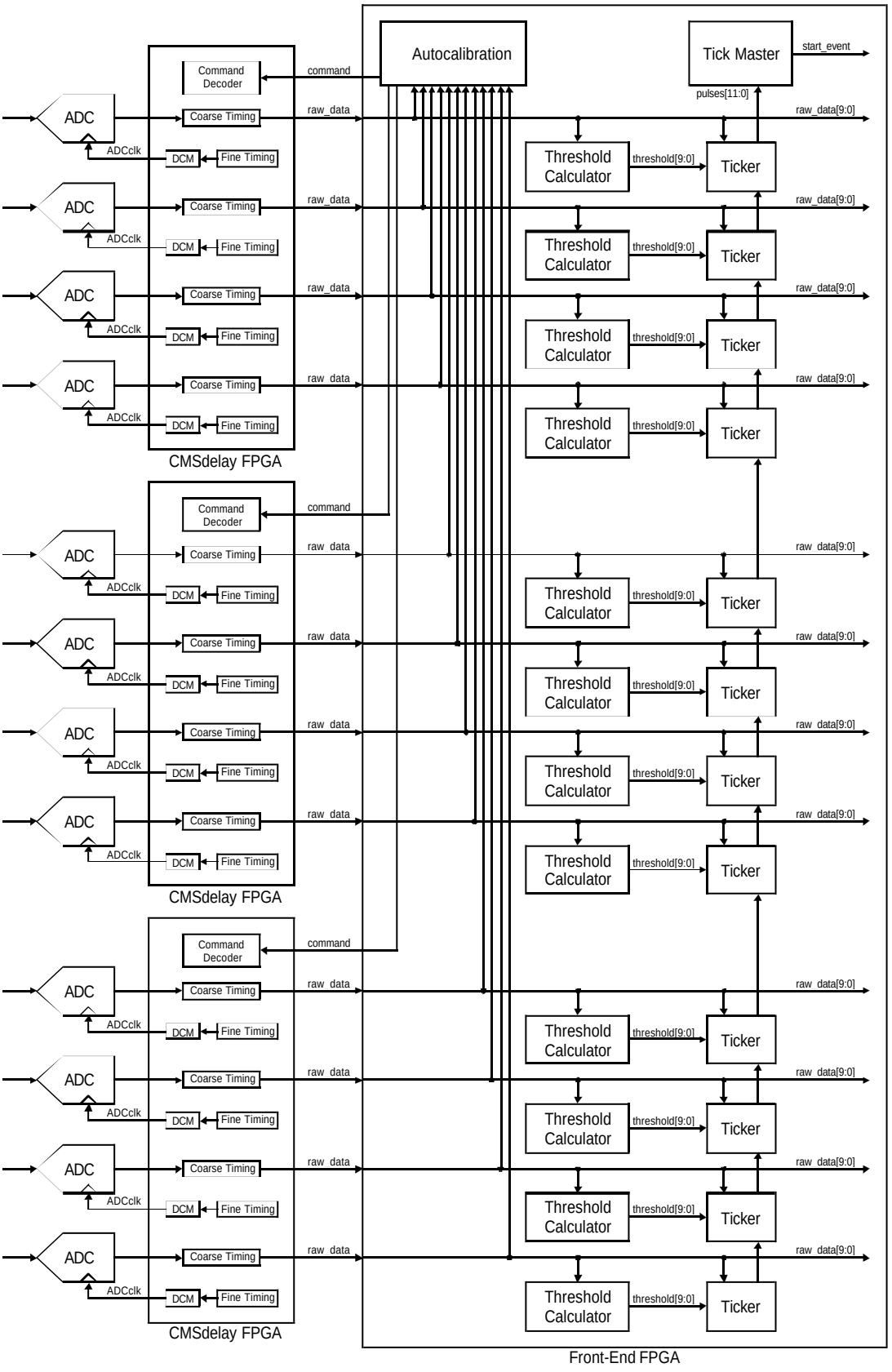

Figure 2-1 Block Diagram of the Housekeeping Components

The next section contains a Brief Description of each of these components.

2.1 Brief Description

The ADCs sample the signal coming from the fibre; one ADC per fibre. The clocks to each ADC are produced by the CMSdelay chips (small Virtex2 devices each with four DCMs) which are themselves controlled from the Front-End FPGA. This allows the sampling point of each ADC to be controlled to 1/32nd of a clock-cycle (781ps).

The output of each ADC is sampled by the CMSdelay chip and thus safely brought into the synchronous domain. In addition to the Fine Timing Adjustment of each **ADCclk**, the CMSdelay chips also contain Coarse Timing Adjustment blocks which can adjust the delay on each fibre by whole clock-cycles in order to achieve complete synchronisation.

The outputs of these blocks are then passed to the Front-End FPGA as **raw_data**, and are the basis for all further calculations on the Data Stream. The Fine and Coarse Timing Adjustments can be set manually, or the Front-End FPGA can be asked to perform an Auto-calibration wherein it attempts to determine the best settings.

The **raw_data** is monitored by the Ticker blocks which have the task of trying to lock on to the APV Tick Marks. These occur every 70 clock-cycles when the APV is not transmitting data, so the Tickers have a counter which continuously counts up to 70. When this counter coincides with the actual timing of the tick-marks the Ticker is said to be 'locked'. When it falls out of lock it repeatedly resets the counter until it locks onto the tick-marks again.

In order for the Ticker blocks to work they need a reliable **threshold**, which is half-way between the Logic 0 and the Logic 1 of the APV tick-marks and Frame Headers. This is produced by the Threshold Calculator block, which uses a combination of three IIR filters to produce a dynamic threshold that tracks the data.

When the Tickers detect a Frame Header they produce a pulse. The 12 pulses are routed in parallel to the Tick Master, which decides whether enough Tickers have fired simultaneously to constitute an 'Event'. If so it produces a pulse on **start_event** which triggers all 12 channels to start processing the Event, whether an individual Ticker agrees with this or not. In this way the 12 Data Path channels always remain in step.

A detailed description of the CMSdelay chip is beyond the scope of this document. Please refer to the relevant documentation relating to that design. The rest of the blocks shall now be described in more detail over the following pages.

2.2 Auto-Calibration

Each Front-End FPGA has 12 fibres coming into it, and each can be at a different value of skew. It is desired that each ADC shall be controlled from a programmable Clock Vernier (the Virtex2 DCM is such a device) which will allow adjustment of the sampling point in 781ps steps across the whole clock-period. This will allow each ADC to operate at its optimal point on the APVMUX waveform, thereby achieving maximum signal integrity.

The output from each ADC can be latched into the CMSdelay chip on either the negative or the positive edge of the master 40MHz clock, and then passed through a programmable-length pipeline. In this way each fibre can be adjusted across many clock-cycles in 781ps steps. In reality it is envisaged that actual skew values could certainly extend to more than 25ns, so a system of this kind shall be required.

Since there are so many fibres to examine, an auto-calibration circuit at the front-end would be very desirable. Such a circuit would have the task of tuning into the tick-marks (it is important that no frames are sent down the fibre during calibration) and scanning the ADC over its full range of skew values. The ideal sampling point for the ADC is a few nanoseconds before the trailing edge of the tick-mark. This is late enough so that the APVMUX waveform has settled to its final value, but not so late that it is starting to move towards its next value.

The current proposal is to perform this task in two stages. First a *Fine Timing Adjust* shall be performed, which will identify precisely the position of the trailing edge of the tick-mark. Second a *Coarse Timing Adjust* shall attempt to find the minimum set of delay values to bring each fibre into line. These two processes shall now be described.

2.2.1 Fine Timing Adjust

The ideal tick-mark waveform (shown below) consists of 68 clock-cycles of a low-level, followed by 2 clock-cycles of a high-level, this repeating every 70 clock-cycles. In reality the difference between 'low' and 'high' is expected to be about 300 ADC counts, so this is a pretty big signal. The optical and electrical chain will introduce time-constants for rising and falling edges, which will be much smaller than 25ns so that the value has stabilised well before the next clock-edge. The ADC will sample at regular intervals along this waveform, and by adjusting the value of skew the waveform can be mapped out.

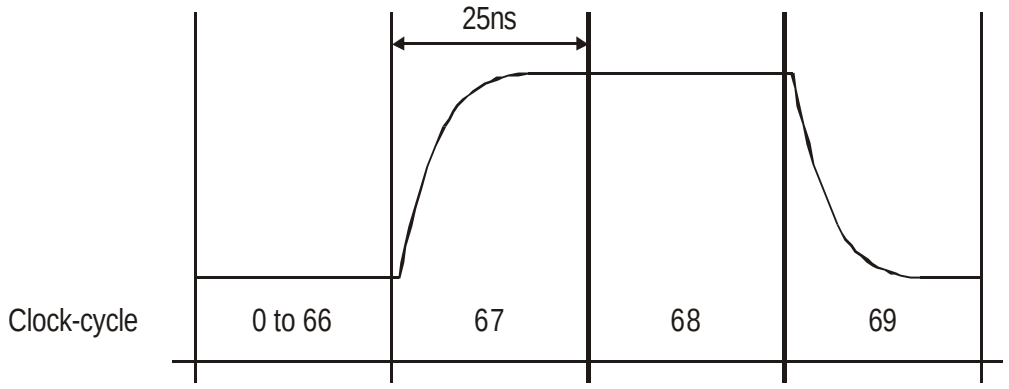

Figure 2-2 Ideal waveform of the Tick-Mark

The straightforward approach to mapping the waveform is to average every 70th ADC sample, whilst adjusting the skew over the full range (70 clock-cycles) in 781ps steps. This takes a long time (perhaps several seconds) but uses the minimum amount of memory and FPGA resource. Moreover autocalibration should only need to be run once at the beginning of an experimental session, so times of the order of one second or so are acceptable.

Noise

The APVMUX tick-mark has been through a lot of processes before it reaches the FED Front-End FPGA. First it is formed by combining the outputs from two APVs, which can introduce switching transients (an 'A' glitch and a 'B' glitch, as we shall call them). Then it is converted to an optical signal, passed through several optical couplers, then back to an electrical signal, before finally being digitised to 10-bits by the ADC.

Each of these stages can introduce imperfections into the signal, but for the moment we shall only consider the following types of imperfection:

- glitches that are the same each clock-cycle (so-called 'clock feed-through')
- glitches that are the same each alternate clock-cycle ('A' and 'B' glitches)
- normal (Gaussian) noise
- random (uncorrelated) spikes, transients and glitches
- differing rising and falling edge rates

In reality there may be further types of imperfection, such as amplitude-modulated distortion, dispersion, multiple optical or electrical reflections, jitter, and inaccuracies in the clock-vernier, but these shall not be considered for the moment. To illustrate the effect of some of these imperfections, the ideal tick-mark has been redrawn below with the addition of clock feed-through and differing edge rates.

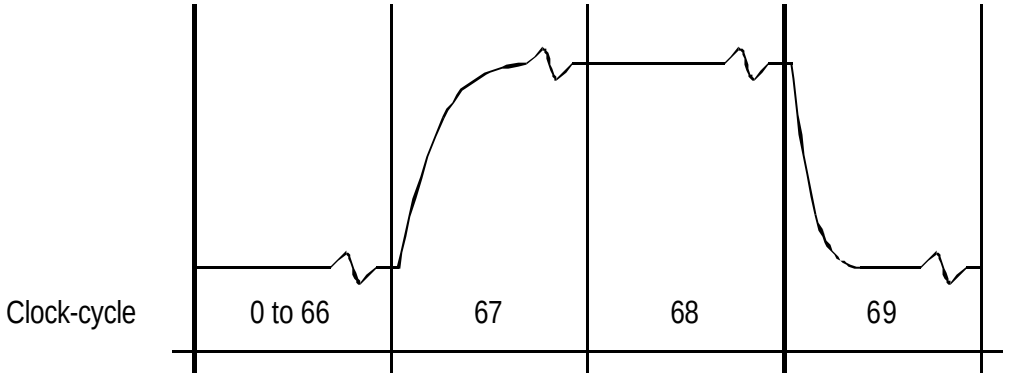

Figure 2-3 Non-ideal waveform of the tick-mark

We shall assume that averaging this waveform long enough will remove all types of uncorrelated noise, but the correlated noise will still remain. To remove this it is possible instead to average the *difference* between successive samples. Moreover, since we expect 'A' and 'B' glitches as well, it becomes prudent to average the difference between *alternate* clock-cycles. This will remove noise correlated across single clock-cycles and pairs of clock-cycles, and should be sufficient to isolate just the tick-marks.

The result of averaging the difference between alternate clock-cycles is shown below.

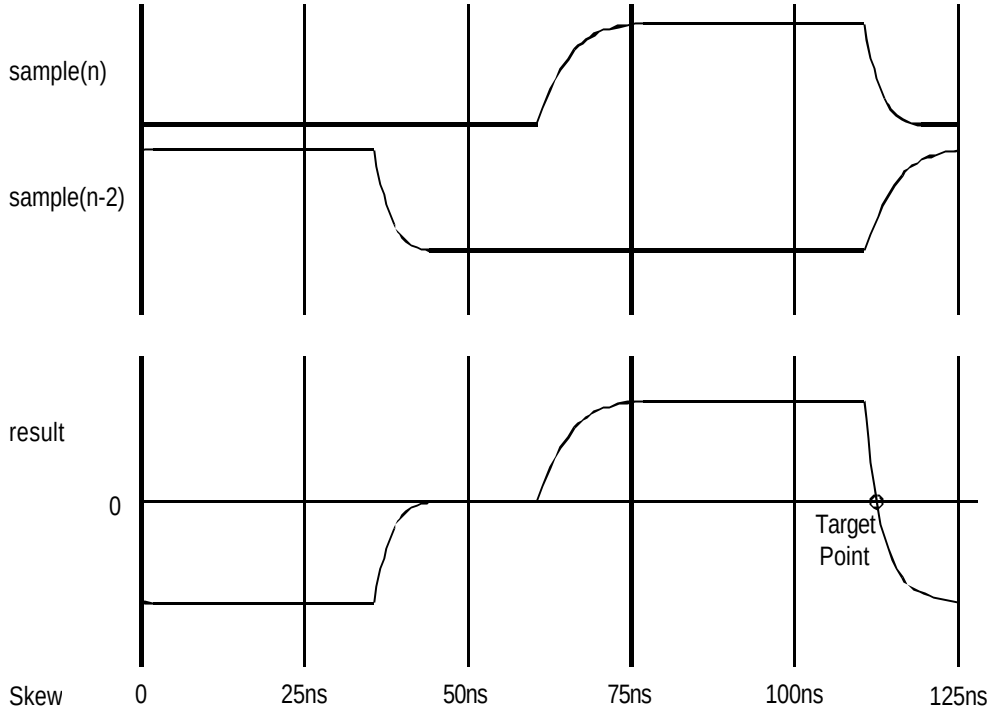

Figure 2-4 Result of averaging the difference between alternate ADC samples

Here, because we are averaging *differences* between samples, the d.c. component has been removed and the waveform is now centred around zero. The correlated noise has been removed, and all that is left is a curious shape caused by tracking the difference between two points as they are traced over the tick-mark. To illustrate how this shape is generated, I have reproduced a simplified tick-mark in the top waveform, and again shifted by two clock-cycles below it. By taking the top waveform and then subtracting the one below, it is possible to see how the bottom waveform is generated, and to prove to yourself that it has the correct general shape. [Note that the curve through the Target Point is formed by subtracting two exponentials with differing time-constants. However this should not be a problem as the faster time-constant will dominate the time to the Target Point.]

The task of the auto-calibration circuit is now to find the Target Point, which is characterised as being the only steep negative transition through zero in the waveform. This point, although it is dependent on the unpredictable time-constants of the transmission system, should be reliable enough to provide a firm foundation for determining the ADC sample point. The exact position of the sample point, which is chosen using the Load_Samplepos Command (see section 4.1.3), should be set far enough ahead to make the variation in the time constants unimportant.

The Target Point is not the only possible zero-crossing in the waveform. With the addition of noise any horizontal section around zero could also contain multiple zero-crossings. However it should be easy to spot that the slope in this area is not all that steep, and in fact the algorithm could be defined as being to find the *steepest* negative zero-crossing on the assumption that this will always be the Target Point.

Other Imperfections

Non-ideal Clock Vernier

If the Clock Vernier used to adjust the sampling point of the ADC is not accurate then problems can be caused for the Fine Timing Adjustment algorithm. In particular if the adjustment is not monotonic, then additional steps can be introduced onto the Result Waveform, as illustrated in the diagram below.

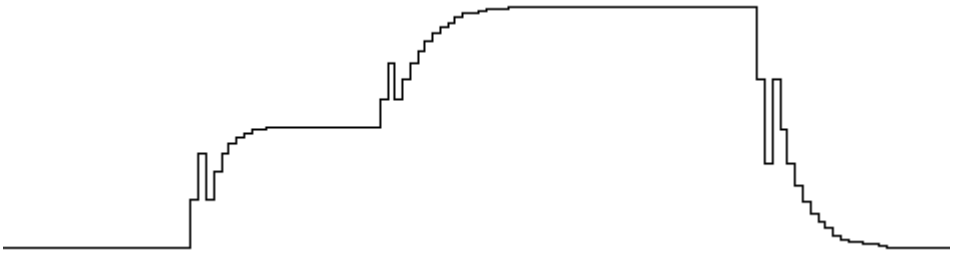

Figure 2-5 Result Waveform produced using non-ideal Clock Vernier

Evidently the Target Point has now been made much more difficult to detect due to the presence of a large wiggle. This could easily occur if the adjustment range does not mesh neatly with the whole-cycle adjustment in the Coarse Timing Blocks, and therefore it is important that the Clock Vernier does not demonstrate this failing (which we believe the Virtex 2 DCM will not).

2.2.2 Coarse Timing Adjustment

The Coarse Timing Adjustment can be performed using the information collected in the Fine Timing Pass. The positions of the twelve Target Points will in effect be integers somewhere in the range 0 to 2239 (where 2239 is 70 x 32 – 1) and pin-point the tick-marks to $1/32^{nd}$ of a clock-cycle anywhere in the 70 clock-cycle repeating period. By removing the bottom five bits we end up with numbers in the range 0 to 69.

Since all the tick-marks should all be close together, it should be possible to find an integer *n* which, when subtracted from all twelve values, reduces one of them to 0 and constrains all the values to be in the range 0 to *m*, where *m* is the adjustment range of the Coarse Timing blocks (currently 16 clock-cycles). The value of *n* will depend on when the Autocalibration command was issued, and could be anywhere in the range 0 to 69. Evidently, since the pattern repeats every 70 clock-cycles, a value of *n* of 69 is equivalent to $n = -1$, and is equivalent to *adding* one. This will occur if the Autocalibration command begins in the middle of the tick-marks, such that the distribution is split into two groups at the extremities of the range.

In practice the Autocalibration command stores all 12 bits of the tick-mark positions in a 12 location RAM called the **calRAM**. Once the Fine Timing Pass is complete and the **calRAM** values are stable, the Autocalibration block can either proceed straight onto the Auto-Coarse procedure or halt, as determined by the value of the **do_autocoarse** flag (see section 4.1.19).

If instructed to halt the values in the **calRAM** can then be read out using the **Load_calRAM** command and operated on at a higher level before being downloaded again using the same command. This permits the Coarse Calibration to be performed for a board, a crate or for all 50,000 fibres as deemed necessary. However, for this to work it is imperative that the Autocalibration command begins at the same time for all Front-End FPGAs, which can perhaps be best achieved using the TTC to trigger the Back-End FPGA to send the command.

Auto-Coarse Procedure

If the Auto-Coarse procedure begins it will do the following operations:

1. subtract all values in calRAM from 3200 (a figure chosen simply because it is a multiple of 32 and more than 2240)
2. add **samplepos**
3. xor with 31 (this complements the bottom five bits)
4. repeatedly subtract 32 from all values until one value is zero and all values are <512. All values will wrap between 0 and 2239.
5. Send lower 9-bits of all final values to the **CMSdelay** chips.

In practice the first three operations can be performed in one pass, and they have the result of changing the sign of all the delays correctly and shifting all the target points by the amount specified by **samplepos**. Once a successful conclusion is reached the resultant values are downloaded to the **CMSdelay** chips which quickly adopt the new delay values.

Problems with Auto-Coarse Procedure

One obvious problem is that if the target point values do not fit into a sixteen clock-cycle range then step 4 of the Auto-Coarse Procedure will never complete. This could result if:

1. the range of tick-marks is too large (unlikely), or
2. one of the fibres is dead, or
3. one of the fibres is noisy, or
4. one of the APVs on a fibre is dead

To cope with this the 13th bit of each **calRAM** word is a **fibre_valid** flag which, if not set, will prevent the Auto-Coarse Procedure from looking at that value, and it will attempt to complete the algorithm using the remaining values. Even if all fibres are valid, the **calRAM** should be zeroed before beginning an autocalibration run, i.e. by loading 4096 into each location.

In addition the procedure will automatically terminate if no solution is found within 8192 clock-cycles. If this occurs then the delay values will not be loaded into the **CMSdelay** chips. In practice, since the Auto-Coarse Procedure is unable to calibrate more than 12 fibres at a time, it is unlikely to be used much outside of the testing phases of the FED development.

2.2.3 Total Execution Time

The two Timing Adjustment steps (Fine and Coarse) will between them use a large number of clock cycles. The biggest factor determining how long this process takes is the amount of averaging that occurs. If it is desired to average 100 tick-marks for each of 32 settings of skew then the Fine Timing will take

3 x 70 x 100 x 70 x 32 = 47,040,000 clock-cycles (1.176 seconds)

The Coarse Timing Adjustment will only take at most 8,000 clock-cycles, so on this time-scale it is insignificant.

2.3 Threshold Calculation

The Threshold Calculator is implemented using an Infinite Impulse Response (IIR) filter which emulates a low-pass characteristic. The filter, the equation of which is reproduced below, operates on the digital input sequence x(n) to produce the output sequence y(n). This filter can be implemented very efficiently if the value of W (the weighting factor) is chosen to be an inverse power of two, such as 1/1024.

$$y(n) = (1-W)\,y(n-1) + Wx(n)$$

For this value of W the Verilog for the above operation can be simplified to that shown below (where '>> 10' means to shift right by 10-bits). Notice that there are no multiplication operations, so this implements very efficiently using only adders and subtractors.

```
average <= average – (average >> 10) + next_sample;
```

However, this IIR will track only the d.c. average of the input waveform which (if it consists solely of Tick-Marks) will be about 8-10 ADC counts above logic zero. Ideally the threshold should sit exactly half-way between logic zero and logic one, and one way to achieve this is to add another couple of filters. The first of these averages values below the d.c. average, and the second averages values above the d.c. average. The threshold can then be defined as the average of these two averages, and should sit exactly halfway up the tick-mark.

In practice, because of noise around the logic zero level (which might exceed 8 ADC counts), it is better to add some constant (say 50) to the d.c. level. This ensures that the first filter really does average logic zeroes, and the second filter only logic ones. The value of the constant has to be large enough to be above the noise, but small enough so that it is nowhere near the height of a tick-mark. The arbitrary choice of a constant that must be somewhere between these two values is one of the weaknesses of this technique. Note that one filter block can be used to service each of the three averages in turn, so the implementation of three filters is not much larger than for one.

If the value of W is chosen to be 1/1024, and the averages are updated every 32 clock-cycles, then the filter has a time-constant of about 1ms. Therefore, if there is noise on the input at a frequency of 1kHz or less, the filter will track it. This may be regarded as a good feature or a bad feature, depending on your point-of-view, so it is important that the user can choose to disable the filters after the auto-calibration process has completed.

If the filters are left running during normal operation, then of course they must be prevented from averaging when a Frame is being received. Fortunately the Tickers can output a signal that does this called **stop_average**. Due to the way I have implemented the design, this will have no effect on the time constant of the filters.

2.4 Tick Monitoring and Header Extraction

After a fibre has been fully synchronised and calibrated it should require no further adjustments in the short term, and possibly over the long term as well. This is because the fibre lengths, optical transmitter/receivers, amplifier characteristics and ADC characteristics should all remain pretty much constant with time. However a block is required to check that this is the case and to flag any errors should they occur.

This block (which is called the 'Ticker' block) has the job of tuning into various features of the APVMUX link and ensuring they remain consistent. The first of these features are 'tick marks' which are present every 70 clock-cycles when no frames are present. Once the counter in the Ticker is locked to the ticks, it can verify that the link is locked by checking that the input is above threshold (a logic '1') whenever **tick_counter** is 0 or 1, as shown below.

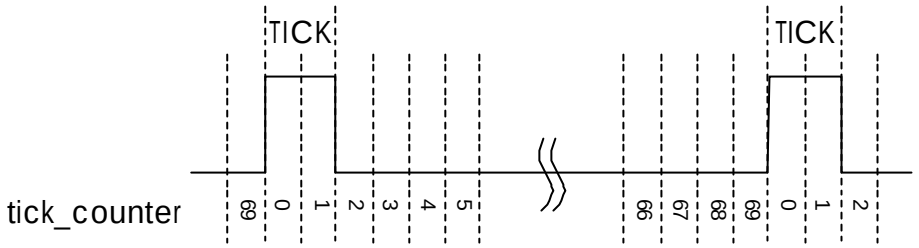

Figure 2-6 APV tick marks

Even if the Ticker fails to achieve synchronisation with its first attempt (for example if it is fooled by noise), it will eventually succeed by repeatedly resetting the **tick_counter** and trying again.

The only other data feature found on an APVMUX link is an APV Frame, as shown below. This contains two column addresses and two error bits (hopefully all identical) followed by 256 words of Physics Data. The Frame is then immediately followed by a tick-mark, the whole frame occupying the same amount of space as four ticks.

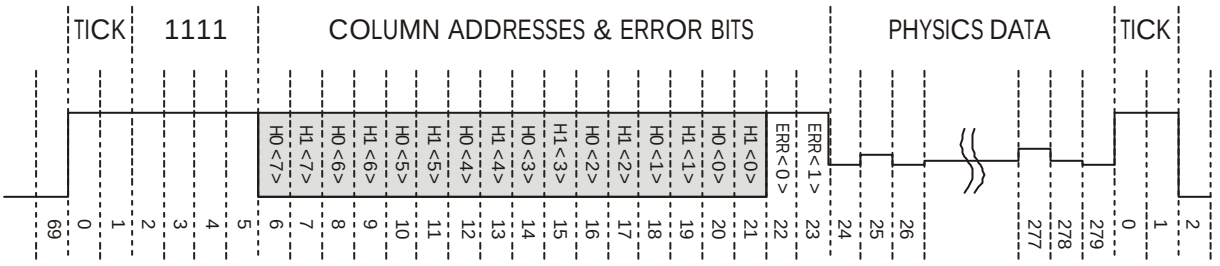

Figure 2-7 APV Frame (40MHz interleaved)

The Frame can be detected by the presence of a '1' when **tick_counter** = 2. In this instance the Ticker must check that the next three words (where 'word' means 'ADC sample') are also logic '1's. If so it can then collect the two interleaved 8-bit headers and the two error bits.

2.5 Monitoring Synchronisation

Once all the Ticker blocks on each fibre are locked, they should produce Headers and Start Pulses in synchrony. To check that this remains the case another block (called the Tick Master) is needed. The Tickers and Tick Master will be connected together as shown in the diagram below.

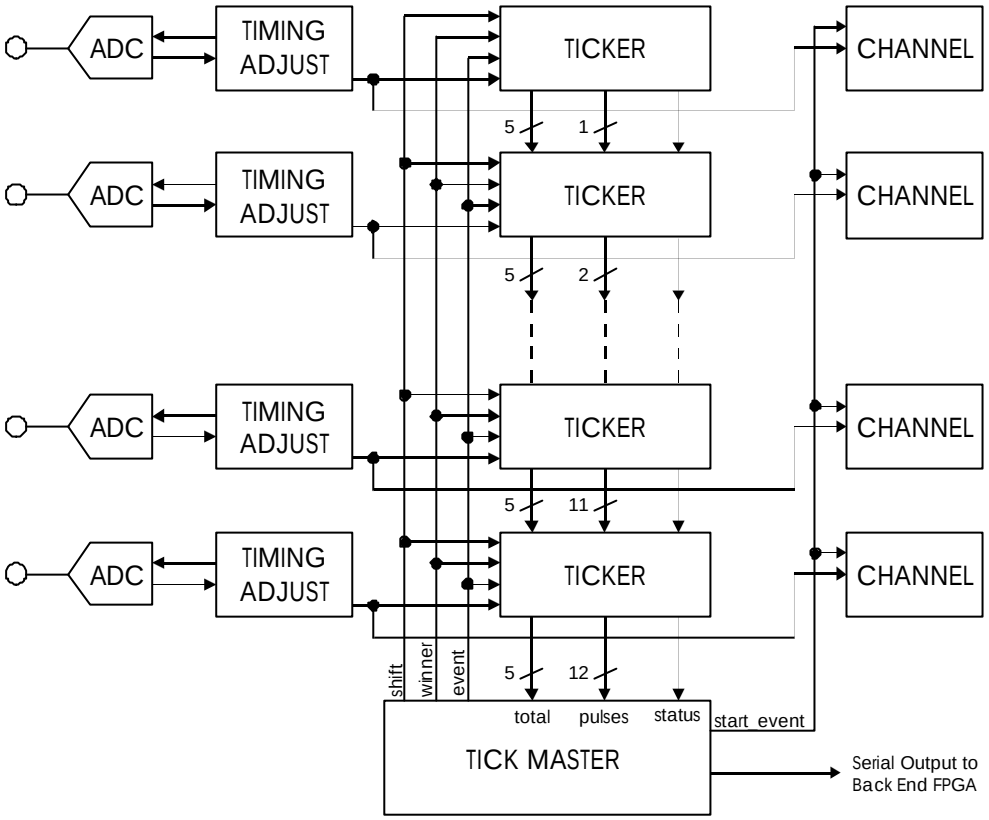

Figure 2-8 Block Diagram showing Tickers and Tick Master

A **pulse** is produced by a Ticker whenever it detects an APV Frame Header. This is defined as being four words above threshold immediately after a tick mark. If all the Tickers are in synchrony then the Tick Master will see all 12 **start_pulses** at the same time, and this will be classified as an *Event*.

In fact an *Event* shall be defined as a clock-cycle in which the clear majority of active fibres produce a **pulse**. For example if there are 12 active fibres, then an *Event* would require 7 to be high at the same time. If this never happens (for example if the fibres are split six and six) then no *Events* will be produced at all.

In response to an *Event*, the Tick Master shall output a pulse on the **pulse_event** output (not shown), which connects to all the Tickers in parallel. From this, any Tickers that are not in synch will be able to work this out and set their Status Bits accordingly.

2.5.1 Header Extraction

The next part of the process is to work out what the Header is. This is not as easy as it sounds because the Tick Master does not know which Tickers will have good Headers, and which ones bad. To understand how the Tick Master operates it is necessary to imagine the different problems that can occur. Once a Ticker falls out of synch with a link it can produce wrong headers at the right time, right headers at the wrong time, extra headers, fewer headers or no headers at all. The Tick System has to be able to cope with all of these situations, and to that end a solution has been implemented that calculates the Median Header.

The Median Header (i.e. the Header that would be in the middle if all Headers were sorted in numerical order) demonstrates good immunity to noise and is fairly quick to calculate. It also treats all the Tickers equally (so there is very much less danger of a bad fibre being regarded as valid), and is extremely good at extracting the Majority Header. In fact the Median Header is bound to be correct so long as the majority of fibres are still working.

The algorithm works by generating a total for each of the bit-slices across the 24 Headers starting with the Most Significant Bit (MSB). The Tick Master looks at the **total** and determines whether the MSB of the Median Header is a '1' or a '0' (based on a simple majority), and sends this result back to the Tickers using the signal **winner**. Any fibres that disagree with the value of **winner** will know immediately that they have the wrong header and will remove themselves from the calculation.

The Tick Master will then instruct all remaining channels to move on to the next bit, produce a **total**, and from this determine the next bit of the Header, and so on. It is most convenient to perform the calculation as the data arrives, so in fact the System has 2 clock-cycles to calculate each bit of the result (see Figure 2-7).

After the calculation is complete the **total** output reports the number of APV Error Bits received. Hopefully this should always be 24 (which indicates no errors) or more specifically it should equal the number of enabled APVs (**num_enabled**) as set by the Load_Tick Command.

In fact if the Tick Master sees a number other than 0 or **num_enabled** on the total output at any time during the calculation, then it knows there is a problem and will set its **problem** Status Bit. The Median Header and all the Status Bits will be shifted out to the Back-End FPGA at the end of the process (see sections 2.5.5 and 4.2.1). If the Back-End FPGA sees that **problem** is not high then it can safely ignore the rest of the Status Bits.

When the **total** output is not being used for the Median Calculation, it reports the number of APVs enabled and in lock (maximum 24). This allows the Tick Master to determine the correct position of the Median, for instance if there are 12 working fibres, but only one APV enabled on each fibre, the Median will be at position 6 in the list.

2.5.2 Recovery

A channel will recover instantly if it produces the Right Header at the Right Time. There is a finite possibility that it might do this by chance, but the odds are very small, so this is not seen as a problem. Even then, its data can be excluded at a higher level based on that channel's case history. The idea is to make the Tickers and Tick Master as dumb and as autonomous as possible, so that they require the minimum amount of external intervention.

2.5.3 External Intervention

If one APV of a pair is non-operational then the Ticker on that fibre will only be getting half a tick-mark. In this situation the Ticker will not achieve lock, and the whole fibre will be marked as invalid. To get around this the Ticker has an extra 2-bit input called **enable** that will tell it whether to expect both APVs, one APV, or neither to be working. This **enable** signal is set externally using the Load_Tick command (see section 4.1.2).

If only one APV is working, then the Ticker must be told whether to expect a 'Left' APV or a 'Right' APV. Once it is told this, it will be able to lock on to the Tick Marks and will synchronise correctly with the other Tickers. At this point its data will be accepted as valid. If only one APV is present operating at 20MHz without an APV Mux, as will very likely be the case in the early stages of commissioning the board, then the FPGA will operate quite happily in any mode. Evidently if it is told to expect two APVs then it will interpret the data stream as two identical APVs multiplexed together, and everything will be duplicated.

2.5.4 APV Protocol Weaknesses

There is a weakness in the APV protocol in that if an input is stuck high it can be legitimately interpreted as an endless sequence of back-to-back frames, all of which have the Header 255. In this instance the Ticker will perpetually believe it is in lock, and will produce a **start_pulse** every 280 clock-cycles. If one of these **start_pulses** coincides with a proper event, and the APV Column Address happens to be FF, then the bad channel will be activated. The only solution is for this fibre to be disabled, and this is good practice in any event.

Another weakness with the APV protocol is that if the first APV is delayed by one clock-cycle with respect to the second, then the data stream still looks perfectly valid. Unfortunately the data from the two APVs will be swapped over, and this will presumably have bad implications for the Physics. If the error occurs after a calibration run, it will be spotted because the fibre will go **out_of_synch**, but if not then the error cannot be spotted except by looking at the calibration result and knowing what to expect. Fortunately this error is extremely unlikely, and will only exist until the next APV reset.

2.5.5 Diagnostics

If a channel goes down it is important to know why it has gone down. Therefore the Tickers are equipped with a number of Status Bits which can permit a certain amount of diagnosis in the event of a problem. There are six of these per Ticker, which form a 6-bit Status Word as defined below.

status = {lock, out_of_synchB, wrong_headerB<APV1>, APVerrorB<APV1>, wrong_headerB<APV0>, APVerrorB<APV0>}

Where

lock	means the Ticker has locked on to something (presumably Tick Marks)
out_of_synchB	means the Ticker is locked but out of time with the last **pulse_event**
wrong_headerB	means the Ticker got the wrong header (according to **winner**)
APVerrorB	means the Ticker received an error bit from an APV chip

The polarities are chosen so that if everything is okay the status word is 111111. The Status Words and the Tick_Master's **problem** bit will be sent to the Back-End FPGA as part of the Frame Synch Packet (see section 4.2.1), which therefore contains a total of 73 Status Bits for each Frame of data.

3. DATA PATH

The bulk of the Front-End FPGA is involved with the numerically intensive task of processing the data stream, extracting the vital information, and storing it ready for readout by the Back-End FPGA. The 12 channels together can process a peak data-rate of 480 Mwords per second, and can cope with any number of back-to-back packets within the constraints of the available memory. A Block Diagram showing the Data Path components for two channels is shown below.

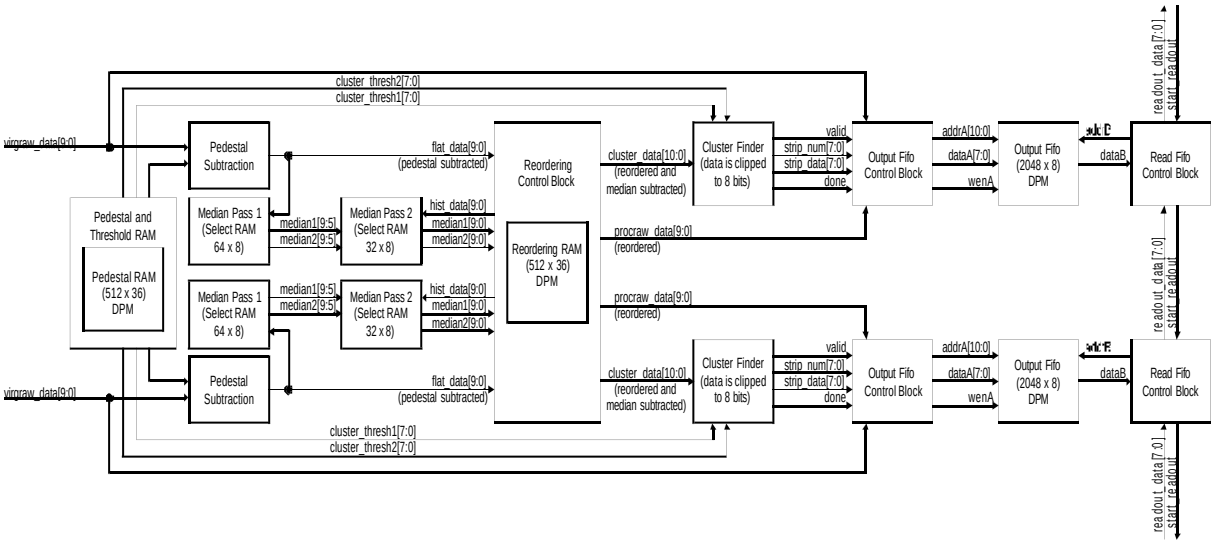

Figure 3-1 Block Diagram of the Data Path Components

The task of the Data Path is to perform the following operations in order:

- subtract a strip-dependant pedestal (256 in total)
- reorder the data into strip-order
- calculate the Common Mode for each APV and subtract it
- perform the Cluster Finding
- store the Clusters in the Output FIFO
- produce a Total Length for the Event (and send this to the Back-End FPGA as a Readout Synch Out packet)
- send the data out to the Back-End FPGA on request

Each of these processes will now be described in more detail.

3.1 Pedestal Subtraction

The *pedestal* is defined as that component of the signal that is specific to a particular strip, and can be calculated by averaging the signals from the APVs in the absence of any hits. Since the d.c. level of the pedestals is unimportant, it is expedient to normalise the data so that the smallest pedestal equals zero. Once the pedestals are subtracted, the data from each APV should form a flat line at some random Common Mode level, and for this reason the output of the Pedestal Subtraction block is called **flat_data**.

In practice there are two more complications which have to be included. Firstly, if the signal from a strip goes off-scale (ADC value = 1023) then it is impossible to say how large it really is, and it should therefore remain at 1023. Secondly, if the signal from a strip is smaller than its pedestal, it should be clipped to zero and not wrap-around to a large number. These two complications aside, the Pedestal Subtraction is a very simple process.

3.2 Common Mode Calculation

The Common Mode is defined as that component of the signal that is specific to a particular APV and a particular Frame, and is calculated by averaging the signals, minus pedestals, in the absence of any hits. It is assumed that the Common Mode component will be flat across the Frame (6.4 us).

Unfortunately, in practice there will be hits superimposed on the data, so a method must be found of either removing them or ignoring them. One such method is to calculate the Median of the signals from each APV, i.e. the value that would be in the middle if the data was sorted into numerical order.

The Median Average of 128 10-bit numbers is more difficult to calculate than the Arithmetic Average (the Mean), but has the advantage that it is considerably less affected by hit channels, and can therefore achieve in one pass what the Mean can achieve in two. In fact, assuming less than 64 channels have been hit, the Median should always be exactly right. Moreover, the median calculation is relatively unaffected by the number of valid channels (since no integer division operation is required).

There are various algorithms for calculating the median, of which one is detailed below.

3.2.1 Median Calculation Method

This method involves performing a histogram of the top 5 bits of the incoming frame. Once finished it is possible by scanning through this histogram to determine which of the 32 bins contains the 64th element, and where within that bin it will be. A 2nd Pass is then performed using the top 5-bits of the Median as a mask and histogramming the bottom 5-bits. Once finished it is possible by scanning through this histogram to determine the exact value of the Median.

The precise scheduling of these operations is as shown in the timing diagram below. All timings are given in clock-cycles at 80MHz.

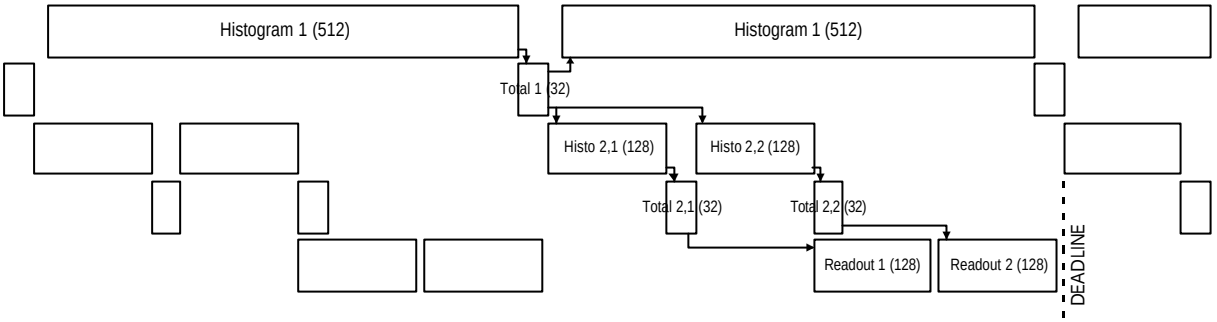

Figure 3-2 Timing Diagram for Median Calculation

Here 'Histogram 1' refers to the task of calculating the top 5-bits of the two medians, and is performed at the same time as the Frame is written into the Reordering RAM. After that the task 'Total 1' produces the **result** and **offset** for each median, which are themselves the inputs to the 2nd stage of the histogramming process. At this point the 1st stage histogramming RAMs are free to start another histogram, which (in the case of back-to-back packets) can occur after just 16 clock-cycles.

The 2nd stage histogram is performed using just one RAM to save space, so only one result can be calculated at a time. This means **median1** is available a lot earlier that **median2**, and permits the 'Readout1' task to begin as soon as the Reordering RAM read port is free (at the end of 'Histo 2,2'). This in turn permits the two Readout passes to be completed before the Deadline (shown on the diagram) when the Reordering RAM read port will be needed for the next packet.

As can be seen this sequence of events only just fits into the time available. If it did not then the real problem is that the bandwidth of the Reordering RAM read port would be exceeded. The data is written into this RAM once and read out twice for each packet, so the solution works neatly by using one port to write at 40MHz, and the other port to read at 80MHz.

3.3 Reordering

The data from each APVMux chip is a complicated (but predictable) sequence of channels from two APV chips. The sequence is created by bit-swapping, as shown below.

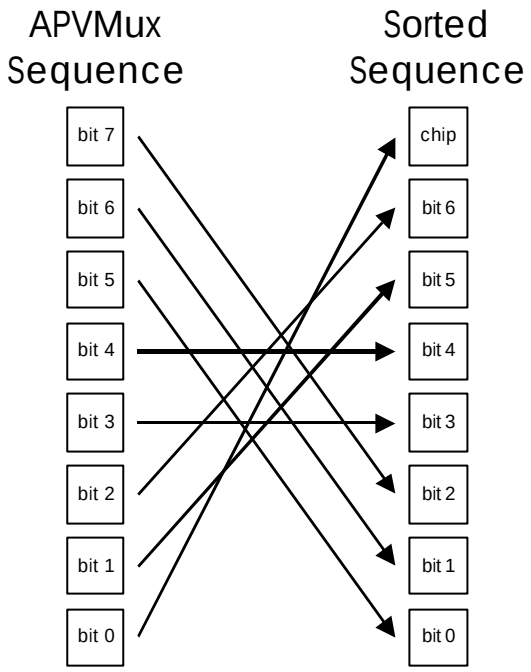

Figure 3-3 APV output bit-swapping sequence

Hence the sequence from the APVMux begins:

```
Chip0/Chan0, Chip1/Chan0, Chip0/Chan32, Chip1/Chan32, Chip0/Chan64, Chip1/Chan64,
Chip0/Chan96, Chip1/Chan96, Chip0/Chan8, Chip1/Chan8, etc.
```

This can be Reordered by writing the entire Frame into a 256 location RAM, then using a suitably bit-swapped address register to read the data back out. In practice, because Frames can arrive back-to-back, a 512 location RAM is required. The Block RAMs on the Xilinx Virtex II are each 18 kilo-bit and can be configured as 512x36, so in fact it becomes possible to Reorder 2-channels using just one Block RAM. [Note: In fact it is possible to reorder 3-channels with one RAM, but this wouldn't be quite so convenient for the rest of the design]

As noted in section 3.2 the data is written into the RAM at 40MHz, then read out twice at 80MHz. The first time it is readout for the 2nd Pass of the Median Calculation, and the second time is for the Cluster Finding. Note (from Figure 3-1) that the width of the output of the Reordering block is 11-bits. This is because the Median has been subtracted and the result could be negative. If it is negative then the Cluster Finder will treat it as zero.

3.4 Cluster Finding

The Cluster Finding algorithm is as defined below.

> *All hits above **thresh1** are output, except single-channel clusters which must be above **thresh2** (where **thresh2** > **thresh1**)*

However, in order to cope with the needs of the Output FIFO Control block (see section 3.4.2) it is necessary to slightly modify this with the following additional rule.

> *All clusters must be at least 2 strips away from every other cluster; any clusters violating this rule should be joined together.*

This guarantees that the Output FIFO Control block will have time to write the Length Field into the Output FIFO before the next cluster arrives. In practice this also makes the data stream slightly smaller. The implementation for this algorithm uses a Sliding Window approach, as illustrated below.

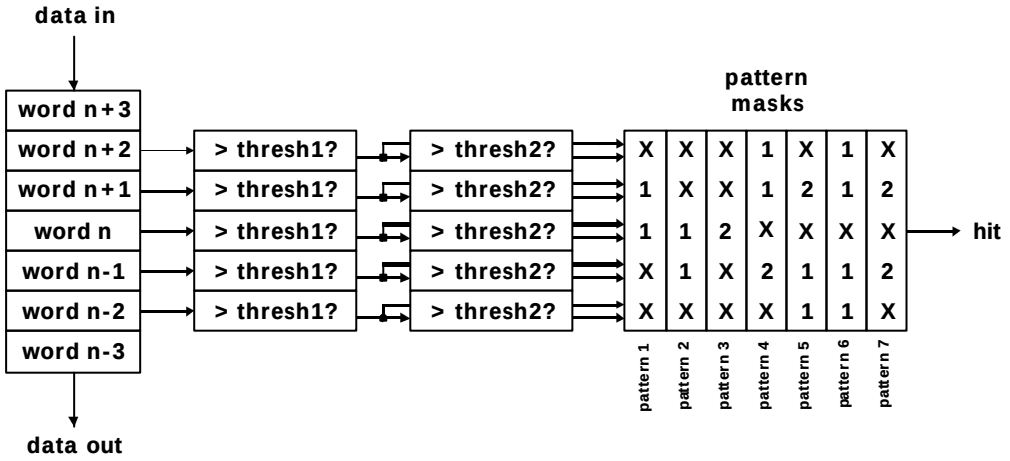

Figure 3-4 Practical Implementation of the Cluster Finding Algorithm

Here a '1' in the Pattern Mask matches anything above **thresh1**, a '2' matches anything above **thresh2**, and an 'X' means 'don't care'. A hit is then indicated whenever the data in the window matches any of the seven patterns. This 'sliding window' approach is very efficient both in terms of size and speed, especially since in practice only two comparators are necessary (since eight of the ten results are available from previous clock-cycles).

The patterns check to see whether the current strip is a hit, and whether the strips on each side of the current strip themselves constitute a hit. If it is certain that 'word n-1' and 'word n+1' are themselves both hits, then 'word n' must be a hit as well.

3.4.1 Output Data Width

The data from the Cluster Finder (**strip_data**) is reduced to 8-bits wide according to the following rules:

Input Value	Output Value
1023	255
254-1022	254
0-253	0-253
<0	0

Table 3-1 Rules for Reducing Data Width from 11-bits to 8-bits

This is needed because the standard output format for the FED is defined as 8-bit data. The number 1023 is treated as a special case because it is 'off-scale' and therefore constitutes an unknown. Normal Physics Data (up to 6 MIPs) should fit comfortably within the range 0-253. Negative values should only be possible as a result of noise and are therefore ignored.

3.4.2 Output FIFO Control

The data from the Cluster Finder must be written into the Output FIFO in a very specific order, so that when it is read out it makes sense. In fact, because the data is being written out-of-order the term 'FIFO' is clearly a misleading description. However, in the absence of anything better we shall continue to use the word FIFO.

When the data is read out from the FIFO, it obeys a format as shown below.

<Fibre1 Length><packet_code><content>
<Fibre2 Length><packet_code><content>… etc.

There will always be 12 Fibres (even if one or more Fibres are disabled using the Load_Tick command) and there can be four different types of output content (see section 4.3). The **packet_code** contains the five mode-bits (see section 4.1.14), and is followed by varying amounts of information depending on the mode. There is no Header or Trailer to the sequence.

However, the problem is that whilst the content is being written into the Output FIFO, the Length is (often) not yet known. For this reason the Output FIFO Control block has to jump back and forth whilst writing the data into the Output FIFO. In practice the Output FIFO Control block contains 3 counters: one for the Cluster Data, one for the Cluster Length, and one for the Fibre Length. By carefully sequencing these counters it is possible to write in all the different types of information in the right order.

3.5 Total Event Length

After all the Clusters have been written into the Output FIFO the Event is declared officially finished. Each Output FIFO Control Block keeps a record of how many words it has written into its Output FIFO for that Event, and now a total for the whole Event across all 12 channels is generated by adding these together.

In practice, to satisfy the timing requirements, flip-flops have to be placed across the channels to slow the addition process down. This ensures that it remains fully synchronous and eliminates the need for multi-cycle constraints.

When the Total is ready (which can be up to 12 * 2048 = 24,576 or 15-bits) it is shifted out on the **readout_synch_out** output to the Back-End FPGA. All eight Front-End FPGAs should send this packet at exactly the same time, and from this the Back-End FPGA can work out how much space is required in its RAM for this Event.

3.6 Data Readout

When the Back-End FPGA is ready to read the data out, it will simply transmit a pulse on the **readout_synch_in** line. In response the Front-End FPGA will send out the data for the next Event in the queue, in Fibre order, using a token-passing scheme between the channels. Note that this means that Fibre 12 is always more likely to overflow than Fibre 1, since it has to wait longest to be read out.

The radix of the readout data is 8-bit, but the method of transfer to the Back-End FPGA is more likely to be 4-bit at 160MHz to make the best use of the Back-End FPGA's limited I/O resources.

3.7 Overflow

The FPGA is carefully designed so that only one thing can overflow – the Output FIFO. The FPGA could have been designed with multiple FIFOs all over the place, and each one would pose the question: what should we do when it overflows? By reducing the design down to just one FIFO (per channel) we only have the one question to answer.

The Output FIFO Control block is not designed to fail gracefully. When the Output FIFO is full, it just stops writing, even if it is mid-packet. The problem is that even if only packet headers are written to the Output FIFO, the FIFO is still bound to fill up eventually, so there is very little the Output FIFO Control block can do to avoid overflow. The lead has to come from the Back-End FPGA.

4. CONFIGURATION AND MONITORING

4.1 Configuration

Although the Front-End FPGA can be configured at any time, some commands may cause temporary loss of operation, and in general it is better to wait until the FPGA is off-line before reprogramming it. All set-up commands are performed via the **command_in** input and obey the following general structure.

<1><r_wB><5-bit Designator><1><16-bit Length><Arguments>

The 5-bit Designator permits 32 commands in total, and the 16-bit length permits up to 65535 bits of Arguments to be used for each command. All fields are transmitted MSB first.

The selected registers are always connected in a loop (of the specified length), so whilst writing the arguments to **command_in**, the previous contents of the registers will be shifted out on **command_out**. If a read-only operation is required, then the **r_wB** bit can be set high. In this case the contents of the registers will be re-circulated, and any arguments on **command_in** will be ignored. Note that in most cases (to save space) there is no double-buffering of the registers. A brief description of all the available commands is given in the table below.

Designator	Command	Length	Description	readback?
00001	Load_calRAM	180	Set up the delay values on each fibre	y
00010	Load_Tick	24	Enable/Disable APVs and Fibres	y
00011	Load_SamplePos	5	Set the ADC Sample Position Offset	y
00100	Load_Thresh	120	Set the Threshold Over-ride Registers	y
00101	Thresh_Enable	1	Enable/Disable Threshold Over-ride	y
00110				
00111	Spy_Arm	1	Starts recording Spy Data	n/a
01000	Spy_Fire	40880	Read out Spy Data for four fibres	n/a
01001				
01010	Load_Median	240	Set the Median Over-ride Registers	y
01011	Median_Enable	1	Enable/Disable Median Over-ride	y
01100	Soft_Reset	n	Apply n clock-cycles of Soft Reset	n/a
01101	Load_Ped_Data	36	Load data into the Pedestal RAM	y
01110	Set_Ped_Address	12	Set the Pedestal Address	y
01111	Set_Number_Valid	192	Set the no. of Valid Strips for each APV	y
10000	Load_Mode_Reg	5	Determine which mode the FPGA is in	y
10001	Load_Scope_Length	10	No. of words captured in Scope Mode	y
10010	Start_Autocal	1	Start an Auto-calibration run	n/a
10011	Load_Numsamples	12	No. of samples for Autocalibration	y
10100	Monitor	145	Read the Output FIFO levels	read only
10101	Do_Autocoarse	1	Enable/Disable Autocoarse procedure	y

Table 4-1 Available Configuration Commands

4.1.1 Load_calRAM Command

<1><r_wB><00001><1><00000000 10110100><00><delay1[12:0]>…<00><delay12[12:0]>

Sets the values of delay for the 12 fibres. The delay consists of three fields as specified below:

delay = <fibre_valid><7-bit coarse delay><5-bit fine delay>

of which only the least significant 9 bits will be transmitted to the CMSdelay chip. This permits the delay on each fibre to be adjusted in 1/32nd clock-cycle increments over a 16 clock-cycle range. The upper 3-bits of the coarse delay field are to maintain compatibility with the output of the Autocalibration function. The **fibre_valid** flag is to permit the Autocoarse function to operate in the presence of dead or faulty fibres.

Once the Load_calRAM command is finished, the values will be transmitted serially to the three CMSdelay chips, a process which will take a further 160 clock-cycles. After this the CMSdelay chips will take an unspecified time to lock on to the new settings, this delay being dependent on the precise function of the Xilinx Virtex2 DCMs.

4.1.2 Load_Tick Command

<1><r_wB><00010><1><00000000 00011000><enable1[1:0]>…<enable12[1:0]>

Used to enable/disable individual APVs and Fibres, as defined in the table below.

enable[1:0]	Meaning
11	enable all (try to interpret the link as two working APVs)
10	enable left (try to interpret the link as a working APV0)
01	enable right (try to interpret the link as a working APV1)
00	enable neither (ignore the link completely)

Table 4-2 Possible values of Load_Tick Command

4.1.3 Load_SamplePos Command

<1><r_wB><00011><1><00000000 00000101><samplepos[4:0]>

Used by the Autocoarse function to adjust the ADC sampling point to a position **samplepos**/32 clock-cycles ahead of the Target Point. This parameter is included because (by definition) the Target Point is the worst possible place to sample the incoming data. This parameter allows the sampling point to be advanced by a programmable amount, this amount chosen depending on the precise rise and fall time-constants of the fibre-link.

If the **Do_Autocoarse** flag is low then **samplepos** will not be used. Changing the value of **samplepos** after an Autocalibration run has finished will have no effect. The default value of **samplepos** after a Hard Reset is 5. It is not affected by a Soft Reset.

4.1.4 Load_Thresh Command

<1><r_wB><00100><1><00000000 01111000><threshold1[9:0]>…<threshold12[9:0]>

Sets the values of the Threshold Over-ride Registers.

4.1.5 Thresh_Enable Command

<1><r_wB><00101><1><00000000 00000001><thresh_enable>

Used to manually over-ride the value of **threshold** to the Tickers.

4.1.6 Spy_Arm Command

<1><0><00111><1><00000000 00011001><1><chip_number[1:0]>→
<1><0><00111><1><00000000 00000001><spy_arm>

Instructs one or all of the CMSdelay chips to collect 1022 words of Spy Data on each fibre. Once instructed the CMSdelay chips will collect data continuously until told otherwise. The **chip_number** field addresses the CMSdelay chips as specified in the table below.

chip_number[1:0]	CMSdelay chip(s) addressed
0	chip 1 (fibres 1 to 4)
1	chip 2 (fibres 5 to 8)
2	chip 3 (fibres 9 to 12)
3	all (fibres 1 to 12)

Table 4-3 Addressing of CMSdelay chips

4.1.7 Spy_Fire Command

<1><0><00111><1><10100000 00011001><1><chip_number[1:0]>→
<1><0><00111><1><10100000 00000000><anything[40879:0]>

Reads back 1022 words of Spy Data for each fibre from the specified CMSdelay chip. Note that it is <u>not</u> possible to read back the data from all three chips at once. Note also that this command will tie up the **command_in** line for a long time, so no other commands can be transmitted until it is finished. The 10-bit Spy Data words are interleaved thus:

<fibre1, word0><fibre2, word0><fibre3, word0><fibre4, word0>→
<fibre1, word1><fibre2, word1><fibre3, word1><fibre4, word1>… etc

4.1.8 Load_Median Command

<1><r_wB><01010><1><00000000 11110000><median1[9:0]>…<median24[9:0]>

Sets the values of the Median Over-ride Registers.

4.1.9 Median_Enable Command

<1><r_wB><01011><1><00000000 00000001><median_enable>

Used to manually over-ride the value of Common Mode used by the Data Path. This (might) be useful for Pb-Pb collisions, where even the Median Average may be affected by the large number of hits (i.e. more than 64 per APV). This technique will only be possible if the value of Common Mode is found to be invariant to within a few ADC counts.

4.1.10 Soft_Reset Command

<1><01100><1><00000000 00000001><1>

Applies a 1 clock-cycle Soft Reset to the FPGA. The Soft Reset resets everything except the Configuration Registers (i.e. any registers set by the commands in this section). If more than one clock-cycle is required (just to make sure) then the Length Field can be adjusted accordingly, e.g.

<1><01100><1><00000000 00000101><11111>

will apply a 5 clock-cycle Soft Reset.

4.1.11 Load_Pedestal_Data Command

<1><r_wB><01101><1><00000000 00100100><ped_data[35:0]>

Used to write/read data to/from the Pedestal RAMs. Each RAM contains the pedestals and cluster thresholds for the four APVs connected to the relevant pair of fibres. Each time this command finishes, the Pedestal Address auto-increments.

The configuration of the **ped_data** field changes depending on the Pedestal Address, as described below.

Pedestal Address	**Configuration of ped_data[35:0]**
<xxx0><n[7:0]>	<xxxxxxx><validstrip1><APV1_ped_n[9:0]>→ <xxxxxxx><validstrip0><APV0_ped_n[9:0]>
<xxx1><n[7:0]>	<x><APV1_cluster_thresh2_n[7:0]><x><APV1_cluster_thresh1_n[7:0]>→ <x><APV0_cluster_thresh2_n[7:0]><x><APV0_cluster_thresh1_n[7:0]>

Setting **validstrip** low excludes that strip from the Common-mode calculation and the Cluster Finding. Setting **cluster_thresh** to 255 just excludes it from the Cluster Finding. The FPGA must be told the number of valid strips using the Load_Number_Valid command.

4.1.12 Set_Pedestal_Address Command

<1><r_wB><01110><1><00000000 00001100><ped_addr[11:0]>

The Pedestal Address determines which location and which RAM is written to when the Load_Pedestal_Data command is executed. The top 3-bits select the RAM, the next bit determines which half of that RAM (pedestals or cluster thresholds) and the bottom 8-bits determine the applicable strip, as described in the table below.

Part of Address	**Description**
ped_addr[11:9]	Pedestal RAM number: <000> selects the RAM connected to fibres 0 and 1 <101> selects the RAM connected to fibres 11 and 12 <111> selects all six RAMs
ped_addr[8]	Pedestal RAM section <0> selects the lower (pedestal) half of the RAM <1> selects the upper (cluster threshold) half of the RAM
ped_addr[7:0]	Selects the strip number. [This may be disordered for the pedestals, and ordered for the cluster thresholds. I'm not sure.]

4.1.13 Load_Number_Valid Command

<1><r_wB><01111><1><00000000 11000000>→
<fibre1_APV0_number_valid[7:0]><fibre1_APV1_ number_valid [7:0]>…
<fibre12_APV0_ number_valid [7:0]><fibre12_APV1_ number_valid [7:0]>

Informs the Median Calculation block how many strips are valid for each APV. This is equal to the number of strips with **validstrip** set high in the Pedestal RAM. If **number_valid** does not correctly mirror the contents of the Pedestal RAM, then the common-mode value will be wrong, and in the worst case the algorithm will not complete.

By choosing different values of **number_valid**, different common-mode algorithms can be selected. For instance (if all 128 strips are valid), **number_valid** = 128 returns the median, 0 returns the smallest, 255 returns the largest, and 86 returns the 43rd from bottom (i.e. the 'one-third-of-the-way-up' median) which may be useful for Pb-Pb collisions.

4.1.14 Load_Mode_Reg Command

<1><r_wB><10000><1><00000000 00000101><mode_reg[4:0]>

The FPGA has various modes of operation which are controlled using this command, as described below.

Mode Reg Bit	Name	Description
mode_reg[4]	procraw_mode	Output will be Processed Raw Data
mode_reg[3]	zerosupp_mode	Output will be Zero Suppressed Data
mode_reg[2]	virgraw_mode	Output will be Virgin Raw Data
mode_reg[1]	frame_finding	The FPGA will respond to incoming frames
mode_reg[0]	scope_mode	The FPGA will respond to incoming triggers

Restrictions: Only one bit of **mode_reg[4:2]** should be set, and only one bit of **mode_reg[1:0]** should be set. In **scope_mode**, only **virgraw_mode** is supported. For any other combinations the action of the FPGA is undefined.

'Virgin Raw Data' are the samples straight from the ADC (via the CMSdelay chip)
'Processed Raw Data' has been through pedestal subtraction and reordering.
'Zero Suppressed Data' has been through pedestal subtraction, reordering and cluster finding.

In **scope_mode** a pulse on the **frame_synch_in** input will cause Virgin Raw Data to be written into the Output FIFO of each channel. The number of words written is determined using the Load_Scope_Length command.

4.1.15 Load_Scope_Length Command

<1><r_wB><10001><1><00000000 00001010><scope_length[9:0]>

Sets the number of words of Virgin Raw Data written into the Output FIFOs after a Scope Mode Event.

4.1.16 Start Autocal Command

<1><r_wB><10010><1><00000000 00000001><1>

Instructs the Housekeeping block to begin an Autocalibration run. Once started there is no means of stopping the run, short of a soft-reset. A read operation on Start_Autocal will return the value of **autocalbusy**, which can be used to determine when the run is finished.

4.1.17 Load_Numsamples Command

<1><r_wB><10011><1><00000000 00010000><ln2_numsamples[3:0]><numsamples[11:0]>

Sets the number of samples taken for each step of the Autocalibration procedure. Values must be a power of 2 between 1 and 2048, and must be preceded by their base 2 logarithm, e.g.

<1><r_wB><10011><1><00000000 00010000><0000><000000000001> numsamples = 1
<1><r_wB><10011><1><00000000 00010000><0001><000000000010> numsamples = 2
<1><r_wB><10011><1><00000000 00010000><0010><000000000100>… numsamples = 4
<1><r_wB><10011><1><00000000 00010000><1101><100000000000> numsamples = 2048

4.1.18 Monitor Command

<1><1><10100><1><00000000 10011101><anything[156:0]>

The Monitor Command will cause the Front-End FPGA to output on **command_out** a serial packet showing the current occupancy of the Output FIFOs. This can be used by the FED system to monitor the buffer levels during experimental runs. The serial packet contains information in the following format:

<1><total1[11:0]><1><total2[11:0]>…<1><total12[11:0]>

4.1.19 Do_AutoCoarse Command

<1><r_wB><10101><1><00000000 00000001><do_autocoarse>

Sets a flag which controls whether the Autocalibration procedure proceeds from the AutoFine procedure to the AutoCoarse procedure.

4.2 Monitoring

The Monitoring block handles most of the communication between the Front-End and Back-End FPGAs on the FED card. This communication is conducted along three pairs of wires:

Signal	Function
frame_synch_out	A packet is produced every time that an APV Frame arrives when in frame-finding mode. This tells the Back-End FPGA when the frame arrived, and contains the header and the Ticker status bits.
frame_synch_in	Only used in Scope Mode to trigger an event.
readout_synch_out	A packet is produced when an event has been processed. This tells the Back-End FPGA that there is data ready to be read out, and contains the chip length field.
readout_synch_in	Used to read out the data for an event.
command_in	Used to configure the FPGA and request monitoring information.
command_out	Outputs configuration and monitoring information.

Table 4-4 Communication signals between Front- and Back-End FPGAs

4.2.1 Frame Synch Out

The Frame Synch Out packet is produced in response to an event, and occurs 28 clock-cycles (at 40MHz) after the 1st word of the APV Frame Header i.e. the first '1' of the six '1's that signify an APV Header (as measured relative to Virgin Raw Data). No packet is produced in response to a Scope Mode event (as triggered by a pulse on **frame_synch_in**). The packet is 84-bits long (sent at 40 MHz) and consists of the following information:

<1><median header[9:0]><problem><fibre 0 status word[5:0]>…<fibre 11 status word[5:0]>

where

median_header[9:0]	is the APV Header that would be in the middle if they were all sorted into ascending order. In practice all 24 headers should be the same.
problem	is low if all fibres report no errors
fibre status word[5:0]	contains the Ticker status bits for that fibre (see section 2.5.5)

At all other times the **frame_synch_out** line is held low.

4.2.2 Readout Synch Out

The *Readout Synch Out* packet is produced when the Datapath completes the processing of an event. In frame-finding mode this occurs 559 clock-cycles (at 40MHz) after the first bit of the Frame Synch Out packet. In Scope Mode this occurs (**scope_length** + 10) clock-cycles (at 40MHz) after the **frame_synch_in** pulse. The packet is 15-bits long (sent at 40 MHz) and consists of the following information:

<1><chip_length[14:0]>

where

chip_length[14:0] is the total length of data held on the chip for this event

At all other times the **readout_synch_out** line is held low.

4.3 Output Data Formats

The main data output of the FPGA is **readout_data[7:0]** which produces data in response to a pulse on **readout_synch_in**. The format of the data will depend on which mode the FPGA was in when the event was written. Up to 255 events can be stored on the FPGA at any one time, after which point further events will be ignored. The FPGA does not require all events in memory to be in the same format. The format of each event is specified by an 8-bit **packet_code** as shown below:

packet_code = <111><procraw_mode><zerosupp_mode><virgraw_mode>→
<frame_finding><scope_mode>

The details of each of these formats shall be described over the next sections. Each of the fields surrounded by <..> are 8-bits, and are right-justified (i.e. any extra padding zeroes occupy the higher order bits).

4.3.1 Zero Suppression Mode

When the FPGA is placed into Zero Suppression Mode, any Frames that come in will be fully processed by the Datapath to produce the smallest possible output format.

When read out, the data packet will have the following format:

<fibre1_length[7:0]><fibre1_length[11:8]><packet_code>→
<median1[7:0]><median1[9:8]><median2[7:0]><median2[9:8]>→
<cluster address><cluster length><cluster data 0><cluster data 1>… etc.
<cluster address><cluster length><cluster data 0><cluster data 1>… etc.
<cluster address><cluster length><cluster data 0><cluster data 1>… etc.
<fibre2_length[7:0]><fibre2_length[11:8]><packet_code>…

The maximum possible length for a cluster is 128 strips (one whole APV).

4.3.2 Virgin Raw Data Mode

When the FPGA is placed into Virgin Raw Data Mode, any Frames that come in will be treated as Raw Data. This means that no pedestal subtraction, reordering or cluster finding will be performed. Instead the 256 words of Raw Data immediately following the Frame Header will be written into the Output FIFO.

When read out, the data packet will have the following format:

<fibre1_length[7:0]><fibre1_length[11:8]><packet_code><raw_word0[7:0]><raw_word0[9:8]>…
<fibre2_length[7:0]><fibre2_length[11:8]><packet_code><raw_word0[7:0]><raw_word0[9:8]>…

This mode will efficiently facilitate the calculation of pedestals and Cluster Finding thresholds.

4.3.3 Processed Raw Data Mode

When the FPGA is placed into Processed Raw Data Mode, any Frames that come in will have a pedestal subtracted, and be reordered. This means that no median will be calculated, and no cluster finding will be performed.

When read out, the data packet will have the following format:

<fibre1_length[7:0]><fibre1_length[11:8]><packet_code><raw_word0[7:0]><raw_word0[10:8]>…
<fibre2_length[7:0]><fibre2_length[11:8]><packet_code><raw_word0[7:0]><raw_word0[10:8]>…

Note that the processed raw words are 11-bit because the median has been subtracted, so the result could be negative. This mode could be useful for Pb-Pb collisions.

4.3.4 Scope Mode

When the FPGA is placed into Scope Mode, all incoming Frame Headers will be ignored. Instead the FPGA will respond to a pulse on the **frame_synch_in** input by writing **scope_length** words of Raw Data into the Output FIFO for each channel. The maximum value of **scope_length** is 1020.

When read out, the data packet will have the following format:

<fibre1_length[7:0]><fibre1_length[11:8]><packet_code><raw_word0[7:0]><raw_word0[9:8]>…
<fibre2_length[7:0]><fibre2_length[11:8]><packet_code><raw_word0[7:0]><raw_word0[9:8]>…

This mode will permit the all aspects of the data stream to be checked and verified, e.g. ADC sampling point, synchronisation, Ticker threshold, Frame Headers, Tick-Marks and Common Mode.