A Prototype Data Acquisition Chip for Large-area, Picosecond Resolution, Time-of-Flight Detectors

Jakob van Santen
The College, University of Chicago

May 16, 2006

Abstract

I describe the design and implementation of a prototype data acquisition (DAQ) and control chip for an array of ultra-high precision timing detectors. Such arrays would have applications in high-energy, nuclear, and astrophysics for particle identification by time-of-flight (TOF), and similarly in medical radiological applications for spatial location. A typical system would consist of an array of sensors, each sensor with four identical chips mounted directly on the sensor, and each chip containing a time-to-digital converter with picosecond resolution. The outputs of these four channels provide the DAQ circuit with a logic-level signal proportional to the time offset of particle arrival from a reference clock, but stretched in time by a factor of 200. The DAQ chip digitizes this signal, distributes the reference clock to the TDCs, and provides control of the readout of the data. Layout of the chip was done in Altera Quartus II software, and the design was downloaded into an Altera field-programmable gate array (FPGA) and tested. The set of schematics, Verilog HDL code, and documentation resulting from this work provide a starting point for the implementation of the same design in a custom integrated circuit able to operate at much higher frequencies. The goal of 1 picosecond resolution, corresponding to a spatial resolution of 300 microns, seems feasible. This would allow the identification of hadrons for momenta a factor of 5-10 higher than presently achieved, and would allow localization of the source of gamma ray pairs in positron-emission tomography (PET) in two dimensions rather than the current one.

Contents

List of Figures

1 Introduction

The identification of quarks and leptons by type or 'flavor' is a large part of many of the most important measurements in Elementary Particle Physics. The three flavors of charged lepton are readily distinguishable by their interactions with matter: the electron produces a distinctive shower in heavy materials, the muon is highly penetrating in matter, and the tau is unstable and decays.

Elementary particles have three directly measurable characteristics: charge, spin, and mass. Each kind of particle has a unique combination of these three quantities. Some kinds of particles have identical charge and spin, and thus can only be differentiated by mass. The pion and kaon are difficult to tell apart, as they both have unit charge and zero spin.

The identification of the component quarks of long-lived hadrons remains impossible at the momenta typical of current and future particle accelerators operating at the high energy frontier [1]. At present, it is impossible even to distinguish a pion ($m \approx 140$ MeV) from a kaon ($m \approx 494$ MeV) at momenta above a few GeV.

While current detectors have no capabilities for the identification of hadrons at high momenta, new electronics and detection techniques can extend the sensitivity of the established mass measurement techniques to a range suitable for hadron identification at the energy frontier. This could have an enormous impact on the identification of supersymmetry, extra dimensions, and other mechanisms beyond the Standard Model of particle physics.

1.1 Roadmap: A Guide to this Thesis

In Section 1.2, I explain how the mass of a particle can be measured in a collider detector, the factors that limit the accuracy of this measurement, and why time-of-flight offers the best prospects for extending the momentum range over which hadron identification is possible. In Sec. 1.3, I describe a proposed large-area time-of-flight detector for charged particles. I explain how a charged particle is detected and how this information is collected and digitized. I conclude Sec. 1 by describing the data acquisition component of the detector electronics and the need for a high-level prototype of this chip before a design is committed to silicon.

In Sec. 2, I describe the benefits of prototyping in Field Programmable Gate Arrays, the tools I used to design and test my prototype, and the requirements that influenced the final design.

In Sec. 3, I explain how the prototype design works and give a detailed description of its logical components.

In Sec. 4, I describe the programming and testing of the actual FPGA device. I discuss how the limitations of the target device affected the design. I explain the tests used to show that the design functions as expected.

Sec. 5.1 contains a brief summary, and Sec. 5.2 discusses the end product of this project and what I have gained from it.

1.2 Techniques for Mass Measurement in Collider Detectors

A particle in motion has three related characteristics: mass, velocity, and momentum. A measurement of any two will yield the remaining one[1]. If we can measure the velocity β and momentum p of a particle, we can find its mass m via $p = \gamma\beta m$.

The standard technique for measuring the momentum of collision products is to apply a strong, uniform magnetic field parallel to the opposing beams. After the collision, the transverse momentum of each collision product is given by $p_T = qBr$, where q is the charge of the particle, B is the magnitude of the magnetic field, and r is the radius of curvature of its path.

While the measurement of momentum is fairly standard, there are three widely-used measurements that lead to β: Energy loss in a dense medium, angle of Cherenkov radiation, and time-of-flight. Of these, time-of-flight presents the best prospects for extending the range of charged hadron identification without an undue increase in the physical size of the detector.

The first technique consists of measuring the particle's energy loss by ionization of a dense stopping medium. Assuming that the incident particle is much more massive than the electron and loses energy only through ionization, its energy loss is given by the Bethe-Block formula

$$-\frac{dE}{dx} = nZz^2 \frac{4\pi\alpha^2\hbar^2}{m_e\beta^2}\left[\ln\frac{2m_ec^2\beta^2}{I(1-\beta^2)} - \beta^2\right]$$

where n is the number density of the stopping medium, Z is the charge of each nucleus in the stopping medium, and z is the charge of the incident particle. I is an experimentally determined parameter that depends slightly on the chemical phase of the stopping medium. The formula breaks down at both low velocities, when positively charged particles can capture and lose many electrons before coming to rest [2]. The rate of energy loss is best determined from the distance the particle travels in the medium before coming to rest; the stopping power (dependant on density and thickness) of the medium must chosen to capture all particles in the velocity range of interest. Assuming limited spatial resolution, resolution in $\frac{dE}{dx}$ scales with the thickness of the stopping medium. Good resolution in β through $\frac{dE}{dx}$ requires significant radial space, which is likely to be in short supply in a cylindrical detector.

A second technique, the direct observation of Cherenkov radiation, is far simpler. When moving through an optically transparent medium of refractive index n, a charged particle emits radiation if its velocity v is greater than the local speed of light, $\frac{c}{n}$. The radiation is emitted at an angle $\theta = \cos^{-1}\frac{c}{nv} = \cos^{-1}\frac{1}{\beta n}$ [2]. Upon entering a refractive medium, a fast charged particle will instantaneously emit a cone of Cherenkov radiation along its flight path. Measuring β is as simple as placing a plane of photodetectors somewhere down-path of the refractive medium, resolving the resulting conic section, and calculating θ. Again assuming a fixed spatial resolution, resolution in θ scales with the distance between the edge of the refractive medium and the plane of photodetectors. This technique, like the first, requires significant radial space to achieve good resolution in β.

A third technique, time-of-flight, takes a distinctly different approach. Instead of measuring β directly, one determines the time at which a particle arrives at distance from the interaction point.

[1]The relativistic relations between mass, energy, and momentum are given by $\beta = \frac{v}{c}$, $\gamma \equiv \frac{1}{\sqrt{1-\beta^2}}$, and $E^2 = p^2 + m^2$. In the latter relation, speed of light is 1.

Instead of fixing t_0, one uses the reconstructed tracks of the particles from a single vertex to infer a time for the interaction. Each track can be used individually to determine the path length L. From these inputs, one calculates β. A time-of-flight detector can also sense Cherenkov radiation as it is generated by a charged particle traversing the detector, but rather than trying to measure the angle of emission θ, a time-of-flight detector measures the time at which the shower of Cherenkov photons arrives. This technique bears several distinct advantages over the previous two. Unlike a measurement of energy loss in a stopping medium, the resolution of this technique is not limited the radial space available, but depends on the time resolution of the sensors and readout electronics used. The same consideration allows a time-of-flight detector to be implemented in much less space than a pure Cherenkov detector of equivalent resolution.

Increased resolution in time-of-flight presents opportunities for hadron identification at much higher momenta than is currently possible. Fig. 1 shows how the 1-σ separation of pions, kaons, and protons is extended with increased time resolution.

1.3 A time-of-flight detector with sub-picosecond resolution

An effort is currently under way to develop an array of high-precision timing detectors to measure the time-of-flight of charged particles produced in high-energy collisions.

The individual detectors of the proposed array are 5 cm $\times$ 5 cm microchannel-plate photomultiplier (MCP-PM) modules with MgF ($n \sim 1.3$) faceplates. Each MCP-PM module is backed by an array of 32×32 anode pads, each 2 mm square. This array is divided into four 'pixels' by a novel equal-time anode. This anode design ensures that the path length from any two points in the 16×16 anode array to the charge collection pin differs by less than 300 microns, the distance light travels in 1 picosecond. Fig. 2 shows an expanded schematic view of one such MCP-PM module [1].

While the techniques being developed are quite general, the effort focuses on the Collider Detector at Fermilab as a concrete example of a collider experiment that could benefit from the installation of a fast time-of-flight component. Covering the 1.5 m radius solenoid of CDF will require 11,340 MCP-PM submodules, arranged in a tiling pattern as shown in Fig. 3.

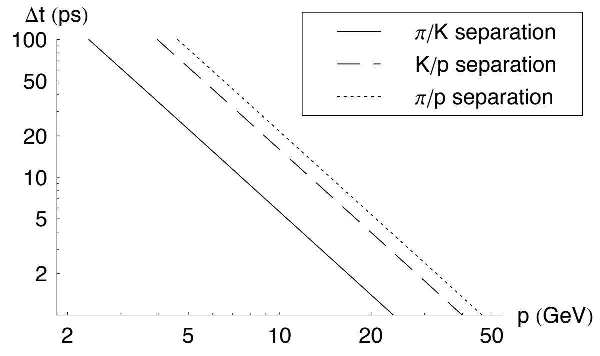

Figure 1: Contours of 1-σ separation between pions, kaons, and protons, assuming a 1.5 m flight path, at various scales of time-of-flight resolution.

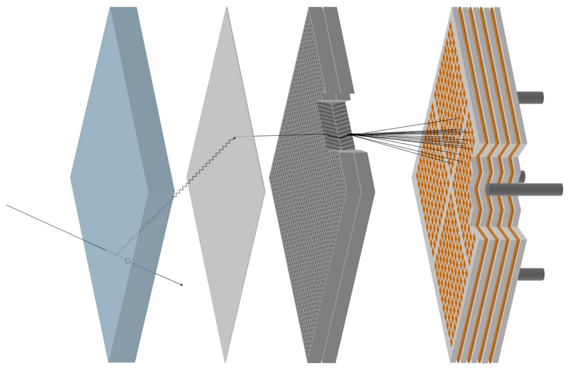

Figure 2: An expanded view of the microchannel-plate photomultiplier assembly, showing (from left to right) the MgF window with incident charged particle, photocathode with incident Cherenkov photon, photomultiplier producing a cascade of $\sim 10^6$ electrons, and custom anode assembly.

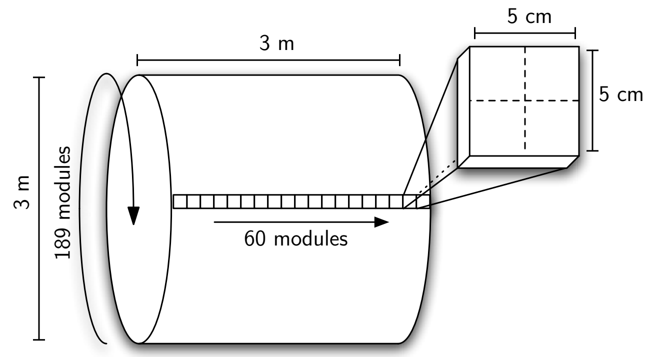

Figure 3: The arrangement of 11,340 MCP-PM submodules in the geometry of the CDF solenoid.

Because of the extremely high frequencies involved, the readout electronics cannot be mounted in a crate and connected to the sensors with cables. Instead, the MCP-PM output pulse must be digitized at the output pin. All readout electronics will be mounted on a printed circuit card and attached directly to the back of the MCP-PM module. The detector electronics system will consist of ≈ 200 logical groups, each with 60 nodes, as shown in Fig. 4.

The readout electronics for each MCP-PM module measure the time at which a charged particle traverses the window of the detector module. To that end, the readout functions are divided between two separate chips: a picosecond-resolution Wilkinson time-to-digital converter (TDC) and a data acquisition (DAQ) chip (see Fig. 5). Each of the four TDCs converts the time of arrival of an MCP-PM pulse to a logic-level (digital) signal proportional to the time offset from a reference clock. The single DAQ chip on each MCP-PM readout module is responsible for distributing the reference clock to the four TDCs, digitizing the TDC output pulses, storing the time at which the TDC output arrived, noting the pixel from which the signal originated, and uploading these data to a storage system.

In a functioning detector, both of these chips will have to function at high speed with minimal jitter[2]. The TDC being developed by Harold Sanders and Fukun Tang (see Fig. 6) requires the ability to build level discriminators and flip-flops with jitter and response times much less than 1 picosecond. For this reason, the TDC must be implemented as an application-specific integrated circuit (ASIC). The TDC is being developed in IHP Microelectronics' silicon germanium process; the final chip will be manufactured and tested by this foundry. Since the TDC operates on a very small time scale, it must be reset every nanosecond. The DAQ chip, which provides the reference clock, must therefore operate at the same high frequencies. It, too, must be implemented in an ASIC.

[2]Jitter is the general term for unpredictable (thus undesirable) variations in the start time and duration of digital signals.

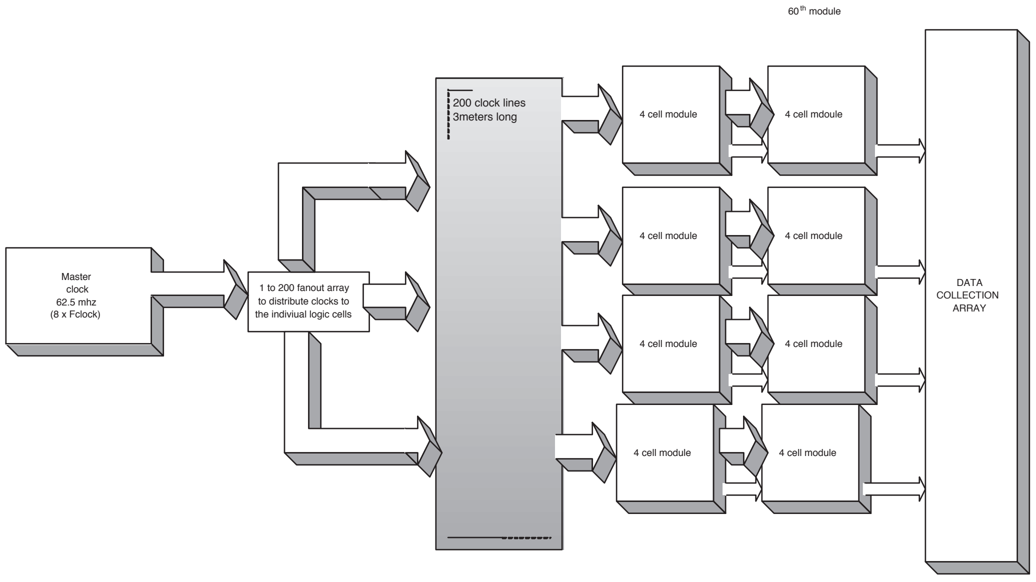

Figure 4: The top-level block diagram of the detector electronics as laid out by Harold Sanders.

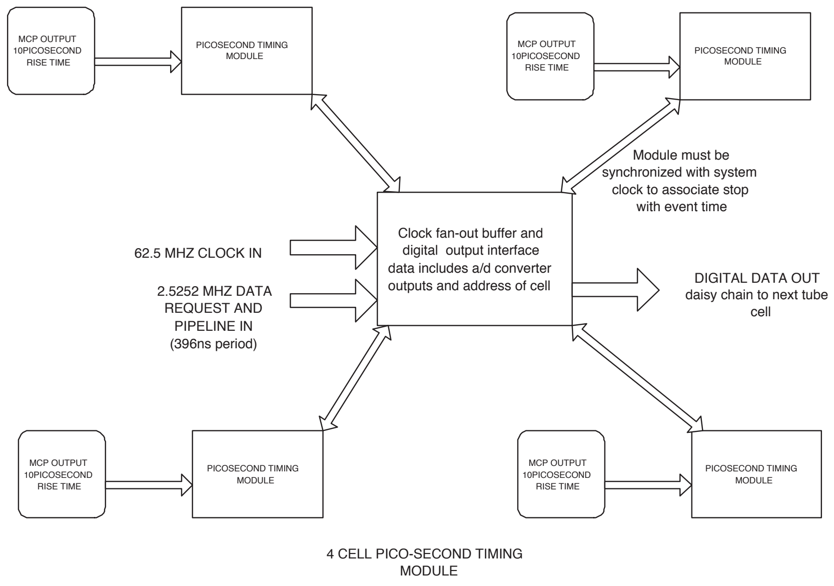
MCP OUTPUT 10PICOSECOND RISE TIME
PICOSECOND TIMING MODULE
MCP OUTPUT 10PICOSECOND RISE TIME
PICOSECOND TIMING MODULE
Module must be synchronized with system clock to associate stop with event time
62.5 MHZ CLOCK IN
Clock fan-out buffer and digital output interface data includes a/d converter outputs and address of cell
2.5252 MHZ DATA REQUEST AND PIPELINE IN (396ns period)
DIGITAL DATA OUT daisy chain to next tube cell
MCP OUTPUT 10PICOSECOND RISE TIME
PICOSECOND TIMING MODULE
MCP OUTPUT 10PICOSECOND RISE TIME
PICOSECOND TIMING MODULE
4 CELL PICO-SECOND TIMING MODULE

Figure 5: A schematic representation of the 4-cell timing module as laid out by Harold Sanders.

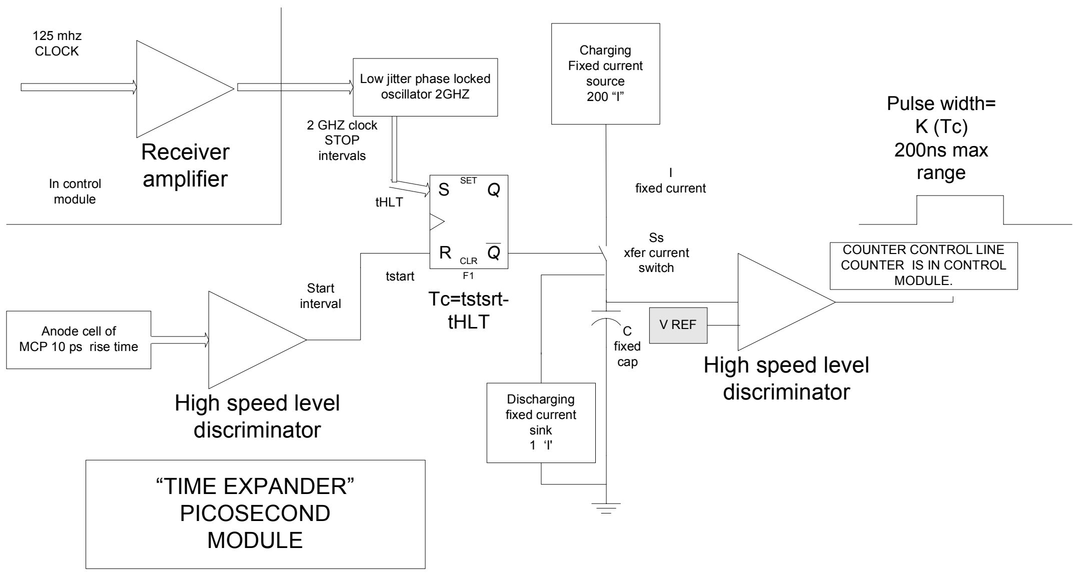
125 mhz CLOCK
Receiver amplifier
In control module
Low jitter phase locked oscillator 2GHZ
2 GHZ clock STOP intervals
tHLT
S SET Q
R CLR Q̄
F1
tstart
Start interval
Tc=tstsrt-tHLT
Charging Fixed current source 200 "I"
I fixed current
Ss xfer current switch
C fixed cap
V REF
Discharging fixed current sink 1 'I'
Pulse width= K (Tc) 200ns max range
COUNTER CONTROL LINE COUNTER IS IN CONTROL MODULE.
High speed level discriminator
Anode cell of MCP 10 ps rise time
High speed level discriminator
"TIME EXPANDER" PICOSECOND MODULE

Figure 6: A schematic representation of the Wilkinson time-to-digital converter being developed by Harold Sanders and Fukun Tang.

The TDC is a mixed analog-digital circuit with a single, well-defined function. It is reasonable to begin designing such a circuit at the level of the individual transistor and capacitor. The DAQ chip is an all-together different component. It is a purely digital device whose functionality was only loosely defined when the work described in this paper began. It is thus a good candidate for prototyping in a Field Programmable Gate Array (FPGA), a type of (re)programmable logic device. This paper describes the design, implementation in Altera FPGA, and testing of a prototype DAQ chip for the proposed array of timing detectors.

2 Development

2.1 FPGA-based design

A Field Programmable Gate Array is an integrated circuit containing many primitive logic elements whose interconnections can be programmed to assume the functionality of Boolean logic gates (AND, OR, NOT, etc.). By the same token, they can be assembled into groups that function like combinations of the basic logic gates, such as 1-bit memories, and further into more complex assemblies such as binary counters and multi-bit first-in/first-out (FIFO) memories. Since the positions of the primitive logic elements are fixed, the designer has much less control over internal delays and therefore overall device speed in an FPGA than in an application-specific integrated circuit (ASIC). They are therefore unsuitable for such high-frequency applications as the readout electronics for the final microchannel-plate photomultiplier module.

None the less, FPGA is an ideal platform for the initial development and testing of a digital integrated circuit[3]. The design tools for FPGA are extremely high-level. Rather than piecing together a binary counter out of individual transistors, one can instantiate a counter as a coherent unit, connect it to rest of the design, and expect it to work immediately. Changing an element of the design and testing the effects of that change in hardware is possible in minutes in FPGA; one could wait months for the manufacture of a revised ASIC. The reprogrammable nature of FPGAs also opens up interesting possibilities for testing and debugging. Internal signals can be routed to blocks of on-board memory, allowing the internal state of the device to be sampled while it is in operation. All of these features make prototyping in FPGA useful as a first step in ASIC development.

2.2 Design tools

The primary design tool for this work was the Quartus II software[4], published by Altera, a manufacturer of FPGAs and other assorted programmable logic devices. This software allows the user to create a design as an arrangement of "blocks," discrete units of logic with their own inputs and outputs. These blocks can then be connected with nodes (individual "wires"), buses (bundles of nodes that carry a multi-bit value), or conduits (bundles of named signals). The blocks themselves can either be schematic diagrams with their own sub-blocks, symbols representing logical primitives, or functional representations written in any one of three hardware description languages (Altera HDL, Verilog, and VHSIC HDL)[5].

The compilation process for FPGA designs is similar to that for computer code. The software synthesizes a list of logic cells and their interconnections from the design, and attempts to fit in into a selected Altera device. The placement of logic cells and connection routes is optimized according to a number of parameters set by the user, who has no direct control over cell placement. If fitting is successful, the resulting description of the compiled device can be used to program an Altera FPGA.

[3]In semiconductor industry jargon, this process is called "pre-silicon verification."

[4]Hereafter, "the software."

[5]All three have similar features (parallel execution of statements, various driving strengths, etc.); I used Verilog for some of the more complex elements of the design because of its relatively clean, C-like syntax.

Once a design has been compiled, the software offers two options for testing and debugging. The built-in simulator is based on the known delays associated with the logic cells and interconnections of a particular device. The simulator produces a set of digital waveforms for the output pins of the device, given waveforms for all the input pins[6]. The second option, the in-system logic sampler called SignalTap II, requires an Altera FPGA to run. The user specifies a sampling clock, a set of signals to observe, and a trigger condition, then recompiles the design. The software then connects the specified nodes to on-board memory cells with appropriate trigger and address-incrementation logic, and programs the FPGA with the resulting design and embedded logic sampler. When a complete sample has been stored, it is streamed to the host computer over the device's JTAG pins[7]. The internal state of the design can then be observed at full speed[8].

I started this project with a rough schematic and an extremely limited knowledge of electronics. Using the aforementioned tools, I laid out and simulated each component of the DAQ chip. I then connected various pieces together and tested their interaction in simulation. Finding that something didn't work as expected, I returned to the lowest level of the design to rework the individual components. After a sufficient number of such cycles, I had created a fully functional prototype. Before describing the prototype, I discuss the requirements and assumptions that influenced the final design.

2.3 Requirements for the DAQ chip

2.3.1 Inputs

The inputs to the DAQ chip are a system clock, a beam crossing strobe, and four TDC pulses. The electronics of CDF provide a 62.5 MHz clock signal and a strobe to signal the crossing of the proton and antiproton beams every 396 nanoseconds. Each Wilkinson time-to-digital converter provides a logic-level pulse whose length is proportional to the time offset of particle arrival from a reference clock, but stretched in time by a factor of 200 (see Fig. 6).

2.3.2 Tasks

The primary task of the DAQ chip is to measure the time at which a charged particle traverses the front window of the microchannel-plate photomultiplier (MCP-PM) assembly and upload this information to a data storage system. There are a number of sub-tasks involved in its successful and reliable completion.

- **Clock distribution** The TDC must be reset at intervals much more closely spaced than the system clock. The DAQ chip must create a subdivided version of the system clock signal and distribute it to each TDC.

[6]The simulation output is analogous to that of a many-channel oscilloscope with a high sample rate but extremely poor (1-bit digital) voltage resolution.

[7]These pins are also used for configuration of the device. JTAG refers to the boundary-scan test interface developed by the Joint Test Action Group in the mid-1980s, and later adopted as IEEE standard 1149.1.

[8]The data captured by the embedded logic sampler are analogous to a many-channel oscilloscope with resolution limited in both time and voltage, but connected to arbitrary points inside the circuit. Since the design is fully synchronous (only active on clock edges), a sample taken on the clock edge provides sufficient information.

- **Pulse width measurement** The DAQ chip must measure the length of the TDC output pulse, preferably in units of the subdivided clock signal.
- **Data storage and tracking** The width of the TDC output pulse only specifies the arrival time within a short (1 nanosecond) interval. The DAQ chip must track the "system" time on coarser scale and attach complete time stamp and pixel location information to each data point.
- **Data queueing and upload** Event data must be stored until such time as the data storage system is able to read them off.
- **Test data injection** The entire detector volume will need to be calibrated and maintained *in situ.* In order to test and troubleshoot the detector array on the fly, the DAQ chip must be able to generate signals of known length at known times and be able to inject these signals into its event processing pipeline.

2.3.3 Special considerations

Data corruption and dead time The DAQ chip works on the assumption that it is better to drop a data point than store a corrupt one. There are two obvious corruption conditions to avoid. The chip should store event data as quickly as possible after the end of a TDC output pulse. While it is doing so, it should enter a state in which it does not accept input, thus ensuring that nothing can change the current data point. This dead time should be kept as small as possible so as not to unnecessarily add to the intrinsic dead time associated with the TDC. It should also ensure that it does not come out of this state in the middle of a new TDC output pulse, which would appear to both arrive at the "wrong" time and be artificially shortened.

Simultaneity and collisions Events may occur simultaneously within a 60-module "column" or a 4-pixel module and compete for data storage and transfer resources. While the average beam crossing at CDF produces about 40 charged particles, no strong assumptions should be made about the spatial or temporal distribution of incident particles. The DAQ chip should handle such event collisions gracefully with queueing and arbitration at both the pixel and module level.

Implicit timing requirements and portability The goal of this work was to provide a largely portable design, ready to be implemented in silicon with minimal delay. This requires a design that considers the timing characteristics of the target device to be entirely arbitrary, and doesn't rely on implicit timing requirements. Each block should be responsible for maintaining a consistent internal state; blocks should not react to each other except when instructed to do so by a dedicated strobe line. Whenever the proper function of a component depends on a minimum set-up or hold time, an identical component should be used to ensure that requirement is met. If this rule is followed, the proper operation of the design will not depend too heavily on the timing characteristics of the individual components.

These requirements determined the form of the final design described in the next section.

3 The prototype DAQ chip

3.1 Top-level block

The top-level block is the object connected directly to chip's input and output pins. It contains 6 sub-blocks: a phase-locked loop, four cell timing blocks, and a data transfer block. A simplified schematic is given in Fig. 7.

The collection of a data point begins at the individual cell timing block, proceeds through the data transfer block and then to the data storage system over the data bus. When a cell timing block receives a signal from its TDC, it stores the length of the pulse and the time at which it arrives, and alerts the data transfer block that it has data to upload. Some time later, the data transfer block reads the proffered data and acknowledges receipt. Once the data transfer block has stored the data point, it notifies the data bus that it has data to upload. Some time later, the bus controller grants the data transfer block permission to assert data on the bus, and the bus controller causes data to be read from the data transfer block.

The generation of "fake" data points is initiated from the other end. The bus controller grants bus access to the data transfer block, but holds the bus in write mode. Instructions coming over the bus are then routed to the self-test sub-block of the appropriate cell timing block. The instructions are executed by the self-test blocks, injecting known signals into the data collection pipeline. The resulting data are read off in the fashion described in the previous paragraph, and can be checked for errors once collected.

The remainder of this section describes the various elements of the top-level block in more detail.

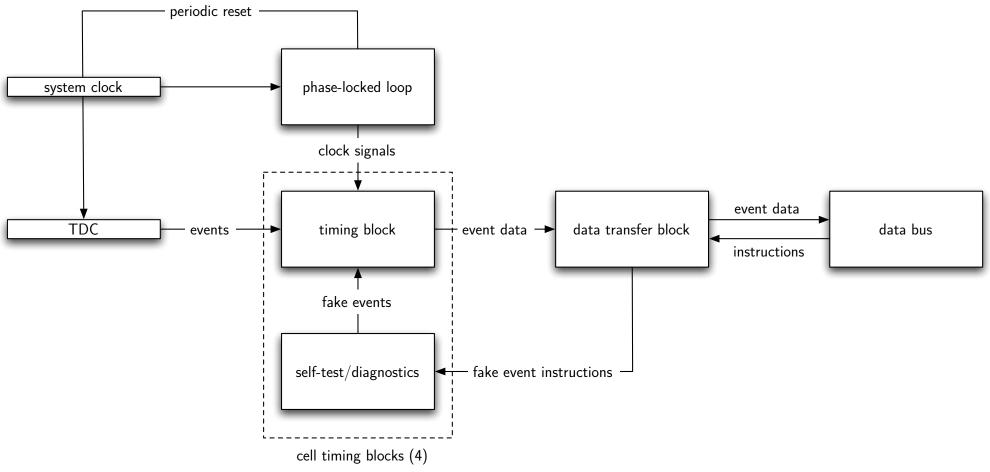

Figure 7: A simplified schematic diagram of the DAQ chip's top-level block.

3.2 Phase-locked loop

A phase-locked loop (PLL) provides the subdivided system clock used to reset the TDCs. A PLL consists of a phase detector, low-pass filter, and voltage controlled oscillator (VCO) arranged as shown in Fig. 8. The phase detector produces an "error" signal whose voltage is proportional to the phase difference between the reference and compensating signals. The voltage-controlled oscillator slowly changes the frequency of its output signal based on the controlling voltage. When the phase detector and VCO are connected as shown in Fig. 8, any difference between the input and VCO output frequency will cause the VCO frequency to drift towards the input frequency, and the PLL will produce a signal locked to the frequency of the input signal. In this form, a PLL is useful for filtering noise from a clock signal.

The PLL becomes more useful when a divide-by-n counter is inserted between the VCO and phase detector. In this configuration, the error signal only vanishes when $f_{\text{out}} = nf_{\text{in}}$. A "biased" phase detector can also be used to produce a copy of the input signal with a known phase shift. Altera's Cyclone$^{\text{TM}}$ FPGAs provide on-board PLLs with both of these features; they are used in the prototype DAQ chip to provide a subdivided and a phase-shifted copy of the system clock.

3.3 Cell timing blocks

The DAQ chip takes input from four time-to-digital converters (TDCs), each of which is serviced by an independent timing block. Whenever a pulse is received from a particular TDC, the cell timing block stores the width of the pulse and the time at which it arrived. After the pulse has passed, it alerts the data transfer block that it has data to pass on. A simplified schematic diagram of the cell timing block is given in Fig. 9, while the schematic created in Quartus[9] is given in Fig. 10.

The cell timing block is centered around three counters that count the number of system clock cycles (5 bits), number of multiplied system clock cycles (4 bits), and the width of an incoming pulse (11 bits). Whether or not they increment on each incoming clock edge is governed by the TDC output

[9]The complete schematic is given for this block only, as the schematics are rather jumbled and fail to communicate the functionality of the device.

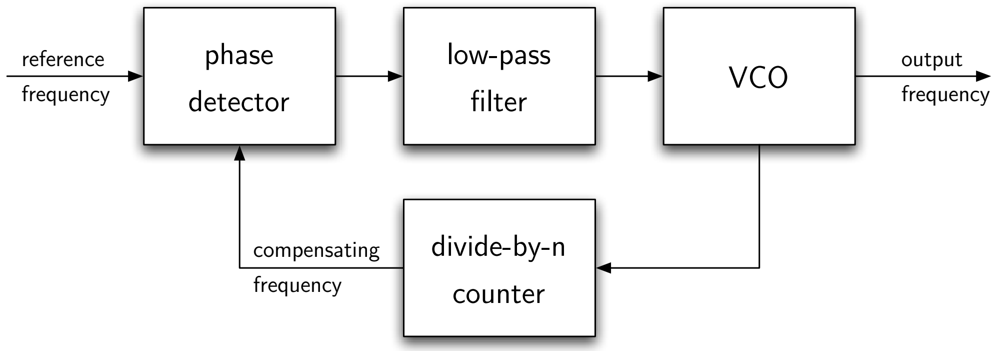

Figure 8: Schematic representation of a phase-locked loop.

`pulse`[10]. When `pulse` is LOW, the pulse width counter is stopped and the the clock counters are running. As soon as `pulse` goes HIGH, the situation is reversed. This arrangement ensures that the clock counters hold the time of arrival immediately after the leading edge of the incoming pulse and that the pulse width counter holds the duration of the pulse immediately after its trailing edge. An edge-detection circuit on `pulse` issues short asynchronous strobes to notify the rest the block of the leading and trailing edges of the pulse. The leading-edge strobe only serves to latch the values held by the clock counters, while the trailing-edge strobe signifies a complete data point, and causes the block to enter a completely new state.

Once a complete data point has been gathered, the timing block locks itself down until the data can be safely stored. Both the leading- and trailing-edge detector elements are designed to be one-shot affairs: once they have detected an edge, they will not function again unless explicitly re-enabled. As the gathered data are protected by latches gated by these strobes, the data cannot change unexpectedly as long as the edge detector remains disabled. The state bit of the trailing-edge detector is also connected to a multiplexer (mux) on `pulse` that ties the internal `pulse` LOW while the trailing-edge detector remains disabled. This ensures that the cell timing block does not respond to external input until the data have been successfully stored.

In addition to latching the value held by the pulse width counter, the trailing-edge strobe causes the clock counters (which are running again, as `pulse` is LOW) to load the values held by a pair of "backup" counters that run at all times, returning timing block to "real" time. At the same time, it causes the buffer block to read the latched data. When these have been successfully read, the buffer block issues a write acknowledgment strobe, which serves to clear the pulse width counter and re-enable the edge detector, returning the data-gathering portion of the block to its default state.

The buffer block, meanwhile, is left to offload the data independently. The buffer is simply a wrapper

[10]Important signals are referred to by name in a `fixed-width` typeface. HIGH denotes Boolean true and LOW denotes Boolean false.

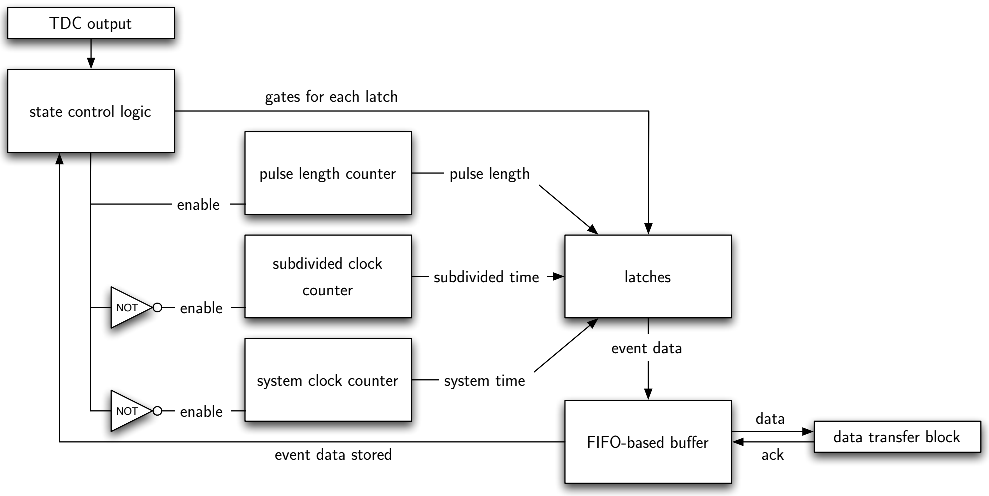

Figure 9: A simplified schematic diagram of the cell timing block.

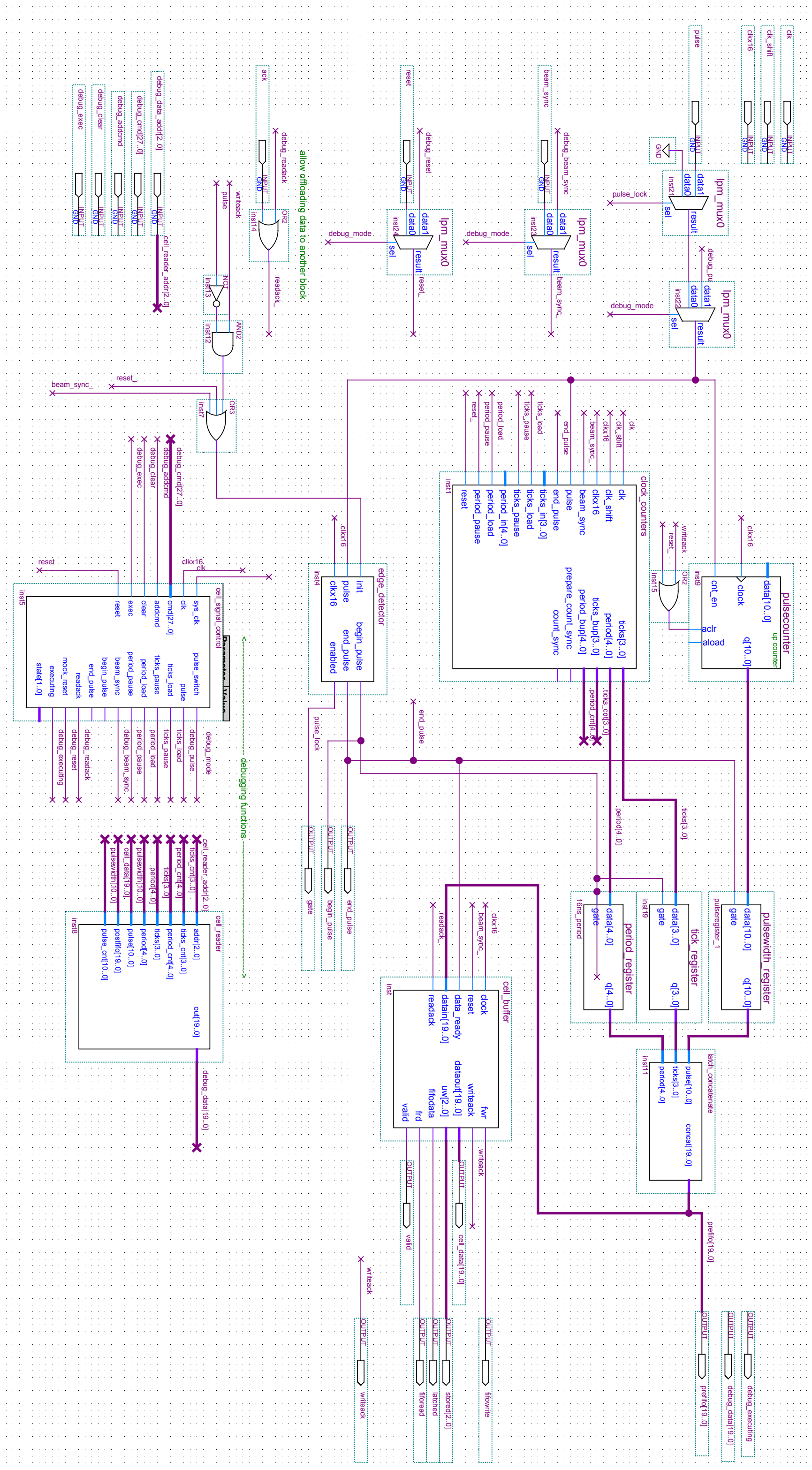
Figure 10: Block diagram for the cell timing block.

of signal-control logic around a first-in, first-out (FIFO) memory. Whenever the buffer logic receives a trailing-edge strobe, it causes the FIFO to load the data present at the data input. As soon as the FIFO contains data, the buffer logic causes a word to be read from the memory, and then brings the `valid` output HIGH. The readout element of the buffer remains in this state until `ack` is asserted, signifying that the word at output has been successfully received by the data transfer block. At this point, the buffer logic causes the next word to be read from the FIFO.

3.4 Debugging features of the cell timing blocks

Each register in the cell timing block is readable from outside the block. When a register's 3-bit address is asserted on the debug address input, its contents will be copied to the lower bits of the 20-bit debug data output.

Each cell timing block includes a programmable signal generator. The signal generator can be fed up to 16 28-bit instructions, each describing a signal identifier, a start time, and a duration. An example of such an instruction is given in Fig. 13. When its `exec` input is asserted, it switches the timing block's inputs to its control and issues the specified signals at the proper times. Event processing is done in parallel, so the state of every signal can change on a single clock edge. The signal generator operates on the multiplied system clock, and can generate signals with durations between 1 and 4095 clock cycles over a 8191-cycle window.

3.5 Data transfer block

The data outputs and `valid` lines of all four cell timing blocks are connected to a data transfer block, shown in Fig. 11. This block is again a FIFO memory wrapped in control logic, although the control logic is slightly more complex than that of the cell timing buffer. The control elements are an arbiter, a multiplexer, and a concatenation function. On the rising edge of every system clock cycle, the arbiter checks whether any of the four `valid` inputs is asserted. If so, the arbiter asserts

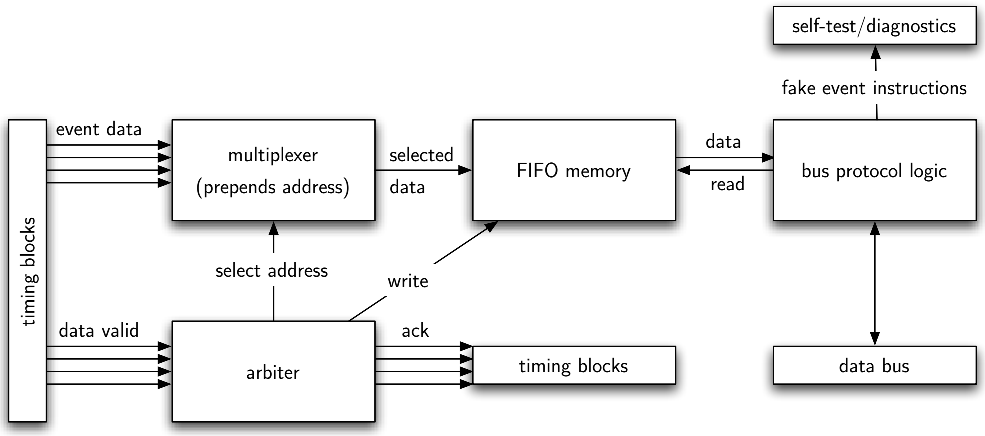

Figure 11: A simplified schematic diagram of the data transfer block.

the 2-bit address of the highest-priority `valid` input[11] at multiplexer and concatenation function. This causes the 2-bit cell address to be prepended to the data from highest-priority cell, and the resulting word to be asserted at the data input of the FIFO. An example of such a complete data point is given in Fig. 12. At the same time, the arbiter brings the write request input of the FIFO HIGH. This process is repeated on every clock edge as long as a `valid` is asserted. At the following clock edge, the arbiter asserts the proper `ack` line to allow the next word to fall through the cell buffer before moving on to read from the next-highest-priority cell. When no `valid` is asserted, `write` is brought LOW again.

Readout from the FIFO is controlled entirely by the bus protocol logic. The protocol for the data bus is extremely simple. All modules on the bus[12] share 32 `data` lines, 6 `address` lines, an `rw` line that determines the direction of data flow on the bus, and an `activity` line whose meaning is dependent on the value of `rw`. In addition, each module controls a dedicated `interrupt` line that indicates to the bus controller that it has data to upload. The block's connections to `data` are driven by tri-state buffers[13] controlled by `address` and `rw`. The bus controller grants access to a module by asserting its 6-bit address on `address`. If the bus is in read mode (`rw` is LOW), the FIFO output is connected to `data`, and `activity` is connected to the FIFO read request input. The bus controller can then read words from the module by asserting `activity`. If the bus is in write mode, `data` is connected to the cell signal generators through a routing function, and `activity` instructs the routing function to write the instruction on `data` to the appropriate signal generator, as shown in Fig. 13.

$$\overbrace{10}^{\text{addr}}\ \overbrace{01010}^{\text{period}}\ \overbrace{0011}^{\text{tick}}\overbrace{00000001010}^{\text{pulse}}\overbrace{0000000000}^{\text{padding}}$$

Figure 12: An example of data point. This indicates that timing block #2 received a pulse starting at 10 system clock cycles and 3 subdivided cycles, and that the pulse lasted for 30 subdivided cycles. The bottom 10 bits are unused.

$$\overbrace{00}^{\text{cmd}}\ \overbrace{10}^{\text{addr}}\ \overbrace{0001}^{\text{sigid}}\overbrace{000000110000}^{\text{start}}\overbrace{000000011110}^{\text{duration}}$$

Figure 13: An example of a testing command. The first 4 bits instruct the routing function to write the remaining 28 bits into the signal generator #2. The 28-bit command instructs signal generator to bring `pulse` HIGH at $t = 48$ and hold it there for 30 subdivided clock cycles.

[11]Priority is determined by address: if `valid` #2 (10) is asserted, it takes priority over `valid` #1 (01)

[12]Each column of modules from Fig. 3 is on an independent bus.

[13]Tri-state logic can drive HIGH, LOW, or Z (high impedance). With tri-state logic, a node can be driven by more than one source. The gates that drive Z are effectively detached from the node until they are enabled.

4 Programming and Testing

4.1 Programming

Once the design had been laid out and had passed *ad hoc* functional simulations, it was necessary to program an FPGA with the finished design in order to test its proper operation in actual hardware. Since a bare FPGA is rather useless without a power supply, programming interface, and breakouts for its pins, it was necessary to use pre-assembled development board. The target hardware was an Altera Nios Development Kit[14] (see Fig. 14) fitted with an Altera EP1C20F400C7 FPGA[15].

The limitations of this particular FPGA forced some changes to the design. While the internal switches would not function reliably at clock frequencies above 100 MHz, the on-board PLL could not operate at frequencies much below 25 MHz. It was thus impossible to maintain the target 16:1 ratio between the multiplied clock frequency and system clock frequency. The 50 MHz crystal oscillator on the development board was replaced with a 25 MHz crystal, and the clock ratio reduced to 4:1. Since the multiplied system clock was intended to subdivide the system clock, it was necessary to ensure that the clock counters rolled over at 3 (2-bit count) rather than 15 (4-bit count). This was accomplished by simply masking the upper two bits of the multiplied clock counter. Once these kinks were worked out, I programmed the FPGA with the final design.

[14]For the curious, Nios is a 16- or 32-bit RISC microprocessor design implemented in Altera FPGA. Altera intended this board to be used for the system-on-a-programmable-chip development. None of these features were used in the work described here.

[15]For those who remain curious, "EP1C20F" denotes an FPGA of the Cyclone™ family, "400" denotes 400 I/O pins, and "C7" means Speed Class 7, or "really slow."

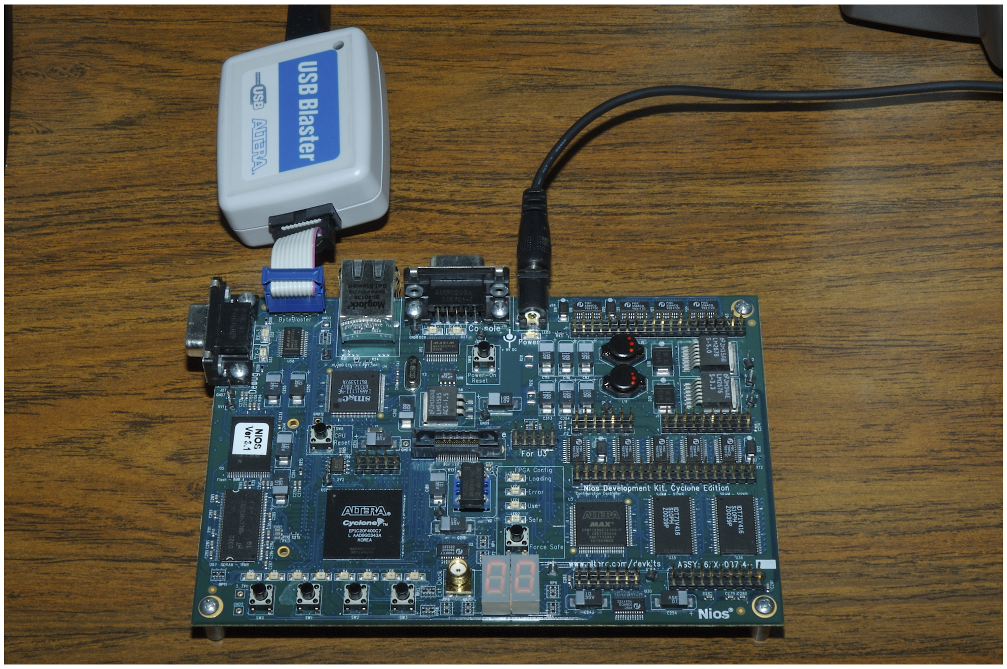

Figure 14: Altera development board with USB JTAG interface.

4.2 On-board diagnostics

Testing was done with the Verilog HDL program given in Appendix A.2. The program loads five instructions into each cell timing block's signal generator and observes the buffer input to ensure that the block behaves correctly. The first two instructions issue a `reset` and a `beam sync` in order to prepare the cell timing block for input. The next three instructions create a 10-cycle `pulse` at $t = 30$, a 30-cycle `pulse` at $t = 48$, and a 5-cycle `pulse` at $t = 90$. The value on the buffer input is sampled at time 41, 79, and 96. The results of each comparison to the expected value are written to a 12-bit `tests_passed` register. If this register reads 111111111111, then the cell timing blocks work as expected. Since the same instructions are executed by all four cell signal generators, this is also a "collision" test for the arbiter in the data transfer block.

4.3 Runtime observation

The on-board test program is a sort of black box; it can't prove that the design functions correctly unless it can be shown that the generator issues the proper signals at the proper times. To that end, Altera's SignalTap II embedded logic sampler was used to observe the internal state of the design during the test.

In the test of the test program, one of the momentary-contact push-button switches on the test board was connected to the `start` input of the test program, while another was tied to the global `reset`. All 12 bits of `tests_passed` were ANDed together and the result connected to one of the LEDs for easy reference. The SignalTap instance was set to trigger on rising edge of the `executing` output of the test program, so that the time scale of the sample would match that of the signal generators. Tests performed in this manner showed that the design works as expected.

In Fig. 15, all four `pulse` lines are driven HIGH and LOW at the proper times. Immediately above the `pulse` waveforms, the `valid` lines are asserted 6 cycles after the end of `pulse`, and are `ack`ed off in the proper order. The waveform immediately below the `pulse` waveforms shows the data transfer FIFO hold 12 words, as expected. Finally, lower right-hand corner shows `tests_passed` storing 111111111111 at the end of the test, showing that the prototype works as expected.

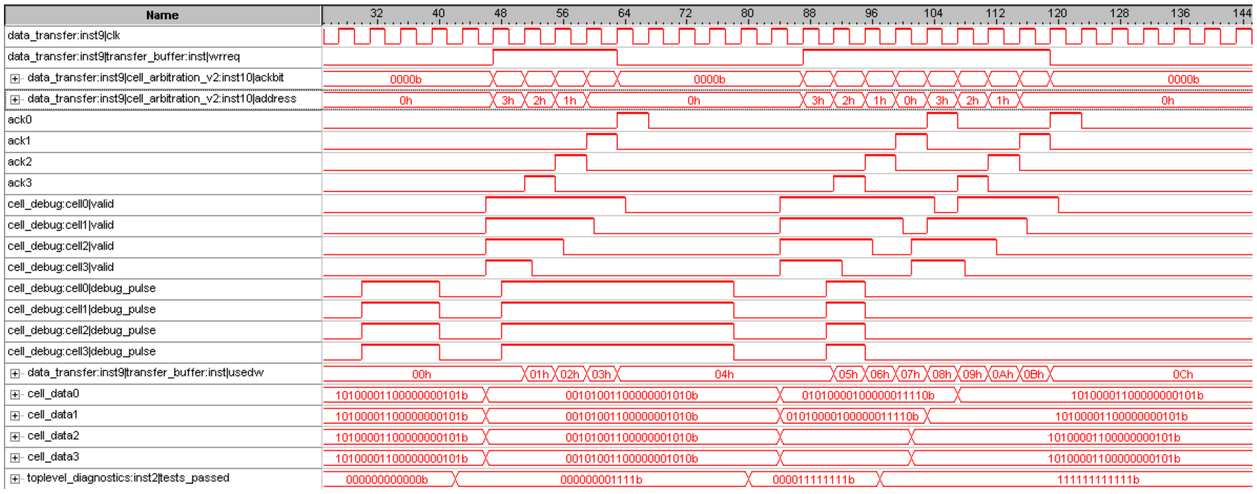

Figure 15: Captured waveforms showing that the DAQ chip prototype works as expected.

5 Conclusion

5.1 Summary

In Section 1, I explained how elementary particles are identified and how some can only be differentiated by mass. I then described various techniques for mass measurement and the factors that limit their resolution. I concluded my general introduction by describing how a time-of-flight measurement can yield the mass of a particle, and how the resolution of this technique depends on the readout electronics used rather than spatial resolution. I then described the effort to develop a time-of-flight detector with sub-picosecond resolution, and explained the requirements for the readout electronics. I concluded this section explaining the need for a high-level prototype of the data acquisition component of the detector electronics.

In Sec. 2, I described the benefits of prototyping in Field Programmable Gate Arrays, the tools I used to design and test my prototype, and the requirements that influenced the final design.

In Sec. 3, I explained how the prototype design works and give a detailed description of its logical components.

In Sec. 4, I described the programming and testing of the actual FPGA device. I discussed how the limitations of the target device affected the design, and explained the tests used to show that the design functions as expected.

5.2 Results

With the completion of this sub-project, the overall effort is a little closer to the goal of picosecond resolution. The set of documentation, schematics, and Verilog source code resulting from the development of this working FPGA prototype form a starting point for the implementation of the DAQ chip in a high-frequency, application-specific integrated circuit.

Although the results of this work will be quite useful, it could have been completed by a skilled electrical engineer in much less time. Never the less, it was beneficial to do this work myself. More than anything else, this was a learning experience. Knowledge of the underlying physical processes is only part of experimental physics; experimentalists also need to know how to design a decent experiment and a solid grasp of electronics is big part of this. At the start of this project, my knowledge of electronics was essentially limited to Maxwell's Laws. This project taught me how to design digital logic, what problems plague various digital circuits, and clever ways to avoid them. These new skills will be quite useful throughout my career.

Acknowledgements

I would like to thank Henry Frisch, my advisor, for his guidance and patience throughout the course of the project.

The schematic layout of the top-level detector electronics system (Fig. 4) and 4-cell timing module (Fig. 5) was done by Harold Sanders. Harold also provided me with a rough schematic of the DAQ chip as a starting point for the project. Harold and Fukun Tang are developing the Wilkinson TDC shown in Fig. 6. Tim Credo designed the equal-time anode shown on the right of Fig. 2. I would also like to thank Mary Heintz for helping me to beat Windows into submission.

References

[1] Timothy Credo, Henry Frisch, Harold Sanders, Fukun Tang, Karen Byrum, and Gary Drake. The development of large-area, picosecond resolution, time-of-flight detectors. `http://hep.uchicago.edu/~frisch/adr.pdf`, 2005.

[2] W. S. C. Williams. *Nuclear and Particle Physics.* Oxford University Press, 1991.

A Verilog HDL source code

A.1 Cell signal generator

```
// Generated by Quartus II Version 4.2 (Build Build 178 01/19/2005)
// Created on Mon Apr 17 14:01:20 2006

//  Module Declaration
module cell_signal_control
(
  sys_clk, clk, cmd, addcmd, clear, exec, reset, pulse_switch, pulse, ticks_load, ticks_pause,
  period_load, period_pause, beam_sync, begin_pulse, end_pulse, readack,
  mock_reset, executing, state
);
// Port Declaration
  parameter maxcmd = 15;
  parameter gather = 2'b00;
  parameter parse = 2'b01;
  parameter execute = 2'b10;
  parameter flush = 2'b11;
  input sys_clk;
  input clk;
  input [27:0] cmd;
  input addcmd;
  input clear;
  input exec;
  input reset;
  output pulse_switch;
  output pulse;
  output ticks_load;
  output ticks_pause;
  output period_load;
  output period_pause;
  output beam_sync;
  output begin_pulse;
  output end_pulse;
  output readack;
  output mock_reset;
  output executing;
  output [1:0] state;

// output signal registers
reg pulse;
reg ticks_load;
reg ticks_pause;
reg period_load;
reg period_pause;
reg beam_sync;
reg begin_pulse;
reg end_pulse;
reg readack;
reg mock_reset;

reg [27:0] cmdreg [maxcmd:0]; // input stack
reg [27:0] cur_cmd; // temp register to allow bit-selection for parsing purposes

// events queue
reg [12:0] events [(maxcmd*2)+1:0];
reg [3:0] events_sigid [(maxcmd*2)+1:0];
reg [(maxcmd*2)+1:0] events_bit;

// pointers for writing in commands and parsing them
reg [maxcmd:0] writepos;
reg [(maxcmd*2)+1:0] parsepos;

reg [12:0] int_time; // internal timer
integer i; // needed for loop construct (not synthesized)
reg [1:0] state; // hooray, we're going to build a finite state machine! (kinda)
reg executing; // extra bit to sync execution to unmultiplied system clock

// ensure that generated signals are synced to the system clock
always @ (posedge sys_clk) begin
// this bit prevents the internal timer from running (see always block below)
executing = (state == execute);
end

assign pulse_switch = executing; // switch cell inputs to this block's control

always @ (posedge clk) begin
// increment internal timer while executing
if (executing && !clear) int_time <= int_time + 1;
else if (state == flush) int_time <= 0; // reset clock here to avoid multiple drivers
end

always @ (posedge clk) begin
if (reset || clear) state <= flush;
case (state)
  gather : begin
    if (addcmd) begin // when addcmd is asserted, push word into input stack
      cmdreg[writepos]  <= cmd;
      writepos      <= writepos + 1;
    end
    if (exec) begin // rest input stack pointer and move to next state
      writepos      <= 0;
      state        <= parse;
    end
  end
  parse : begin
    // fill each event register with a signal id, on/off bit, and a timestamp
    cur_cmd          <= cmdreg[writepos]; // pull word into temp register to allow bit-selection
    writepos        <= writepos + 1; // increment input stack pointer
    events[parsepos]    <= cur_cmd[23:12]; // begin-time timestamp
    events_sigid[parsepos]  <= cur_cmd[27:24]; // signal identifier
    events_bit[parsepos]  <= 1; // bring signal HIGH
    events[parsepos+1]    <= cur_cmd[23:12]+cur_cmd[11:0]; // end-signal timestamp
    events_sigid[parsepos+1]<= cur_cmd[27:24]; // signal identifier
    events_bit[parsepos+1]  <= 0; // bring signal LOW
    parsepos        <= parsepos + 2; // increment parsing stack pointer
    if (writepos == maxcmd) begin // at end of input stack, begin execution
      state      <= execute;
    end
  end
  execute : begin
    if (int_time == 0) begin // at time=0, bring all signal outputs LOW
      pulse      <= 0;
      ticks_load   <= 0;
      ticks_pause  <= 0;
      period_load  <= 0;
      period_pause <= 0;
      beam_sync    <= 0;
      begin_pulse  <= 0;
      end_pulse    <= 0;
      readack      <= 0;
      mock_reset   <= 0;
    end
    /* Hard-coding the memory addresses is a dirty, dirty way to do this
     * unfortunately, it seems neccessary to allow parallel signals
     * without time-intensive magic sorting foo
     */
    for (i=0;i<=(maxcmd*2)+1;i=i+1) begin
      if (events[i] == int_time) begin
        case (events_sigid[i])
          1  : pulse     <= events_bit[i];
          2  : ticks_load    <= events_bit[i];
          3  : ticks_pause   <= events_bit[i];
          4  : period_load   <= events_bit[i];
          5  : period_pause  <= events_bit[i];
          6  : beam_sync     <= events_bit[i];
          7  : begin_pulse   <= events_bit[i];
          8  : end_pulse     <= events_bit[i];
          9  : readack       <= events_bit[i];
          10  : mock_reset    <= events_bit[i];
          //default : state <= flush;
        endcase
      end
    end
    //if (int_time == 13'b1111111111111) begin // drop out of execution at end of reachable timescale
    if (int_time == 13'd1200) begin //finish early
      state  <= flush;
    end
  end
  flush : begin
    for (i=0;i<=maxcmd;i=i+1) begin // clear input stack
      cmdreg[i]      <= 0;
    end
    for (i=0;i<=(maxcmd*2)+1;i=i+1) begin // clear events queue
        events[i]    <= 0;
        events_sigid[i]  <= 0;
        events_bit[i]  <= 0;
    end
    cur_cmd          <= 0; // clear temp parsing register
    writepos        <= 0; // reset input stack pointer
    parsepos        <= 0; // reset parsing stack pointer
    state          <= gather;
    // it is also necessary to reset the internal timer, but this must be done in another always block (above)
  end
endcase
end

endmodule
```

A.2 Test program

```
// Generated by Quartus II Version 4.2 (Build Build 178 01/19/2005)
// Created on Thu Apr 20 14:36:21 2006

//  Module Declaration
module toplevel_diagnostics
(
  clk, start, reset,
  cell_reg_data0, cell_reg_data1, cell_reg_data2, cell_reg_data3,
  sig_executing0, sig_executing1, sig_executing2, sig_executing3,
  cell_reg_addr0, cell_reg_addr1, cell_reg_addr2, cell_reg_addr3,
  sig_cmd0, sig_cmd1, sig_cmd2, sig_cmd3,
  sig_addcmd0, sig_addcmd1, sig_addcmd2, sig_addcmd3,
  sig_clear, sig_exec,
  active, tests_done, tests_allpassed, tests_passed
);
// Port Declaration

  input clk;
  input start;
  input reset;
  input [19:0] cell_reg_data0;
  input [19:0] cell_reg_data1;
  input [19:0] cell_reg_data2;
  input [19:0] cell_reg_data3;
  input sig_executing0;
  input sig_executing1;
  input sig_executing2;
  input sig_executing3;
  output [2:0] cell_reg_addr0;
  output [2:0] cell_reg_addr1;
  output [2:0] cell_reg_addr2;
  output [2:0] cell_reg_addr3;
  output [27:0] sig_cmd0;
  output [27:0] sig_cmd1;
  output [27:0] sig_cmd2;
  output [27:0] sig_cmd3;
  output sig_addcmd0;
  output sig_addcmd1;
  output sig_addcmd2;
  output sig_addcmd3;
  output sig_clear;
  output sig_exec;
  output active;
  output tests_done;
  output tests_allpassed;
  output [11:0] tests_passed;

reg [3:0] state; // state machine declaration
parameter idle  = 4'd0;
parameter done  = 4'd1;
parameter test1  = 4'd2;
parameter test2  = 4'd3;
parameter test3  = 4'd4;
parameter test4  = 4'd5;
parameter test5  = 4'd6;
parameter test6  = 4'd7;
parameter test7  = 4'd8;

reg [12:0] int_time; // internal timer
reg int_time_enable; // flag to run internal timer
reg int_time_clear; // flag to clear internal timer
reg tests_done; // test programs have finished
reg [11:0] tests_passed; // flags for tests completed successfully
reg [2:0] reg_addr; // address of cell register to read from
reg [27:0] sig_cmd; // command to load into signal generator {4'b(sigid),12'b(start time),12'b(duration)}
reg sig_addcmd; // load sig_cmd into signal generator
reg sig_exec; // instruct signal generator to begin execution of commands in its input stack
reg sig_clear; // clear the signal generator
reg waiting; // test done, waiting for signal generator to time out

//simple flag for all passed
assign tests_allpassed = (tests_passed == 12'b111111111111);

//tie all blocks together
wire sig_executing;
assign sig_executing = (sig_executing0 && sig_executing1 && sig_executing2 && sig_executing3);
//echo to commands to all blocks
assign {cell_reg_addr0,cell_reg_addr1,cell_reg_addr2,cell_reg_addr3} = {4{reg_addr}};
assign {sig_cmd0,sig_cmd1,sig_cmd2,sig_cmd3} = {4{sig_cmd}};
assign {sig_addcmd0,sig_addcmd1,sig_addcmd2,sig_addcmd3} = {4{sig_addcmd}};

assign active = (state != idle); // give test program control of signal generators when running

always @ (posedge clk) begin // timer control
  if (int_time_clear) begin
    int_time    <= 0;
  end else if (int_time_enable) begin
    int_time  <= int_time + 1;
  end
end

always @ (posedge clk) begin
if (int_time_clear) int_time_clear <= 0; // kill any clock clears from the previous cycle
if (reset) state <= idle; // allow reset at any time
case (state)
idle: begin
  // keep the timer stopped and cleared
  int_time_enable    <= 0;
  int_time_clear    <= 1;
  // don't block observation phase
  waiting        <= 0;
  // make sure we're not lying about our status
  tests_done      <= 0;
  tests_passed    <= 0;
  if (start) begin
  //state  <= test1;
  state  <= test3; // jump straight into triple-pulse stress test
  end
end
done: begin
tests_done      <= 1;
if (reset) state <= idle; // return to idle state on reset
end

test3: begin // triple-pulse stress test
case (sig_executing)
0: begin
  waiting <= 0; // clear waiting bit so observation phase can continue
  case (int_time)
    0: begin // feeding vectors to signal control block
      sig_clear <= 1; // for safety's sake, clear out the sig control block
      int_time_enable <= 1; // start the timer
    end
    1: begin
      sig_clear <= 0;
      sig_cmd <= {4'd10,12'd0,12'd2}; // at time=5, assert mock_reset for 2 ticks
      sig_addcmd <= 1;
    end
    2: begin
      sig_cmd <= {4'd6,12'd8,12'd2}; // at time=8, assert beam_sync for 2 ticks
    end
    3: begin
      sig_cmd <= {4'd1,12'd30,12'd10}; // at time=30, assert pulse for 10 ticks
    end
    4: begin
      sig_cmd <= {4'd1,12'd48,12'd30}; // at time=48, assert pulse for 30 ticks
    end
    5: begin
      sig_cmd <= {4'd1,12'd90,12'd5}; // at time=90, assert pulse for 5 ticks
    end
    6: begin
      sig_addcmd <= 0; // stop pushing commands into input stack
      sig_exec <= 1; // start executing
    end
    7: begin
      sig_exec <= 0;
      int_time_enable <= 0; // important! shut off the timer until sigcontrol block starts!
    end
  endcase
end
1: begin // signal control block running, observing results
  if (!waiting && !int_time_enable) begin
    int_time_clear <= 1;
    int_time_enable <= 1;
  end
  case (int_time)
    5: begin
      reg_addr <= 0; //glom on to the pre-FIFO output
    end
    40: begin // pulse ends at t=40, allowing for edge-detector delay
      /*
      period = 5
      ticks = 3
      pulsewidth = 10
      */
      tests_passed[0] <= (cell_reg_data0 == {5'd5,4'd3,11'd10});
      tests_passed[1] <= (cell_reg_data1 == {5'd5,4'd3,11'd10});
      tests_passed[2] <= (cell_reg_data2 == {5'd5,4'd3,11'd10});
      tests_passed[3] <= (cell_reg_data3 == {5'd5,4'd3,11'd10});
    end
    78: begin // pulse ends at t=78
      /*
      period = 10
      ticks = 1
      pulsewidth = 30
      */
      tests_passed[4] <= (cell_reg_data0 == {5'd10,4'd1,11'd30});
      tests_passed[5] <= (cell_reg_data1 == {5'd10,4'd1,11'd30});
      tests_passed[6] <= (cell_reg_data2 == {5'd10,4'd1,11'd30});
      tests_passed[7] <= (cell_reg_data3 == {5'd10,4'd1,11'd30});
    end
    95: begin // pulse ends at t=95
      /*
      period = 20
      ticks = 3
      pulsewidth = 5
      */
      tests_passed[8] <= (cell_reg_data0 == {5'd20,4'd3,11'd5});
      tests_passed[9] <= (cell_reg_data1 == {5'd20,4'd3,11'd5});
      tests_passed[10] <= (cell_reg_data2 == {5'd20,4'd3,11'd5});
      tests_passed[11] <= (cell_reg_data3 == {5'd20,4'd3,11'd5});
    end
    100: begin
      int_time_clear <= 1; // set clock to zero next cycle
      int_time_enable <= 0; // disable timer
      waiting <= 1; // wait for signal execution to finish before starting next test
      state <= done; // remember to exit this state (else, you loop every time the internal timer rolls over)
    end
  endcase
end
endcase
end

endcase
end

endmodule
```