The designs of the master and slave CCB FPGAs

[Document number: A48001N004, revision 8]

Martin Shepherd, California Institute of Technology

January 20, 2005

This page intentionally left blank.

Abstract

The aim of this document is to detail the design of the CCB FPGA firmware, and define its interfaces to the rest of the CCB hardware. The design will be presented in a hierarchical manner, starting with block diagrams of major components and their interconnections, and ending with low level generic components, such as AND gates and latches.

Contents

List of Figures

Chapter 1

Introduction

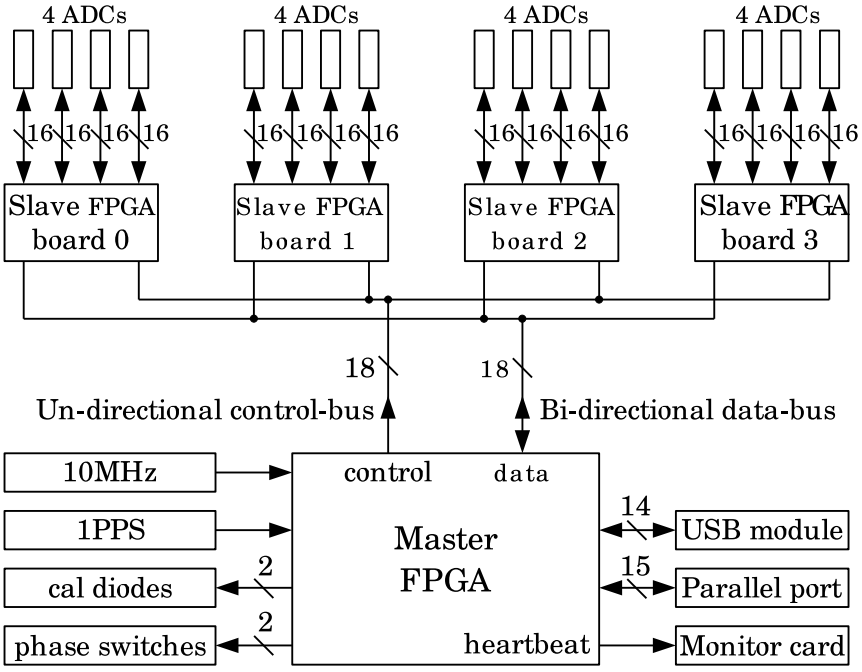

Figure 1.1: An overall summary of the FPGA connections

Figure 1.1 shows the overall architecture of the FPGAs with respect to the rest of the CCB. At the heart of the system, the Master FPGA controls 4 slave FPGAs, receives commands and sends interrupts and read-back configuration parameters, via the computer's EPP parallel port, dispatches observed data to the computer over a USB link, and controls calibration diodes and phase switches in the receiver, via opto-isolated output cables. All of its timing signals are derived from the Green Bank 10MHz and 1PPS reference signals.

Under the direction of the Master FPGA, each of the slave FPGAs continuously reads 14-bit data samples from 4 ADCs at 10MSPS, and either integrates these samples until told to deliver them to the Master FPGA, or, when in dump mode, delivers them un-integrated to the Master FPGA.

Note that although a bi-directional 18-bit data-bus is shown in the diagram, the current design only uses 16 of these bits, and only transfers data over them in one direction, directed from the slave FPGAs to the master FPGA.

The following two chapters detail the internal logic and external interconnections of the Slave and Master FPGAs, respectively.

Chapter 2

The slave FPGAs

There are 4 slave FPGAs controlled by one master FPGA. All of the slave FPGAs are identical, so this chapter documents the internal components, and external I/O connections of a single slave FPGA. Figure 2.1 shows the layout of a slave FPGA, showing the major logic components within the FPGA, the internal interconnections between these components, and all of the external I/O-pin connections to the 4 ADCs to the left, and to the master FPGA, via the backplane bus, at the bottom of the diagram.

2.1 An overview of the internals of a slave FPGA

Starting from the left hand-side of the diagram, the `adc_clock` input is a phase-shifted copy of the main FPGA clock-signal. This signal clocks the 4 external ADCs, whose outputs are then latched using the un-phase-shifted global FPGA clock, into input registers within the associated *Sampler* components. The configurable phase shift between the two clocks allows one to control at what point in each ADC sampling cycle the FPGA latches samples from the ADCs, and thus allows one not only to accommodate the relative timing requirements of the ADCs and the FPGAs, but also to move the noisy active part of the FPGA clock cycle away from critically sensitive parts of the ADC clock cycle.

Next, the *Sampler* components take either the latched ADC samples, as their input samples, or fake pseudo-random samples from the *Signal Injector* component, according to the state of the `test` control-signal. The selected input samples are then both reproduced at the `raw` outputs of the *Sampler* components, and integrated.

Within the individual *Sampler* components, each new sample is integrated by adding it to one of 4 phase-switch bins, as directed by the `phase` control-signal. When the master FPGA commands the start of a new integration period, by asserting the `start` signal, the contents of these phase-switch bins are copied into output buffers, then the bins that are selected by

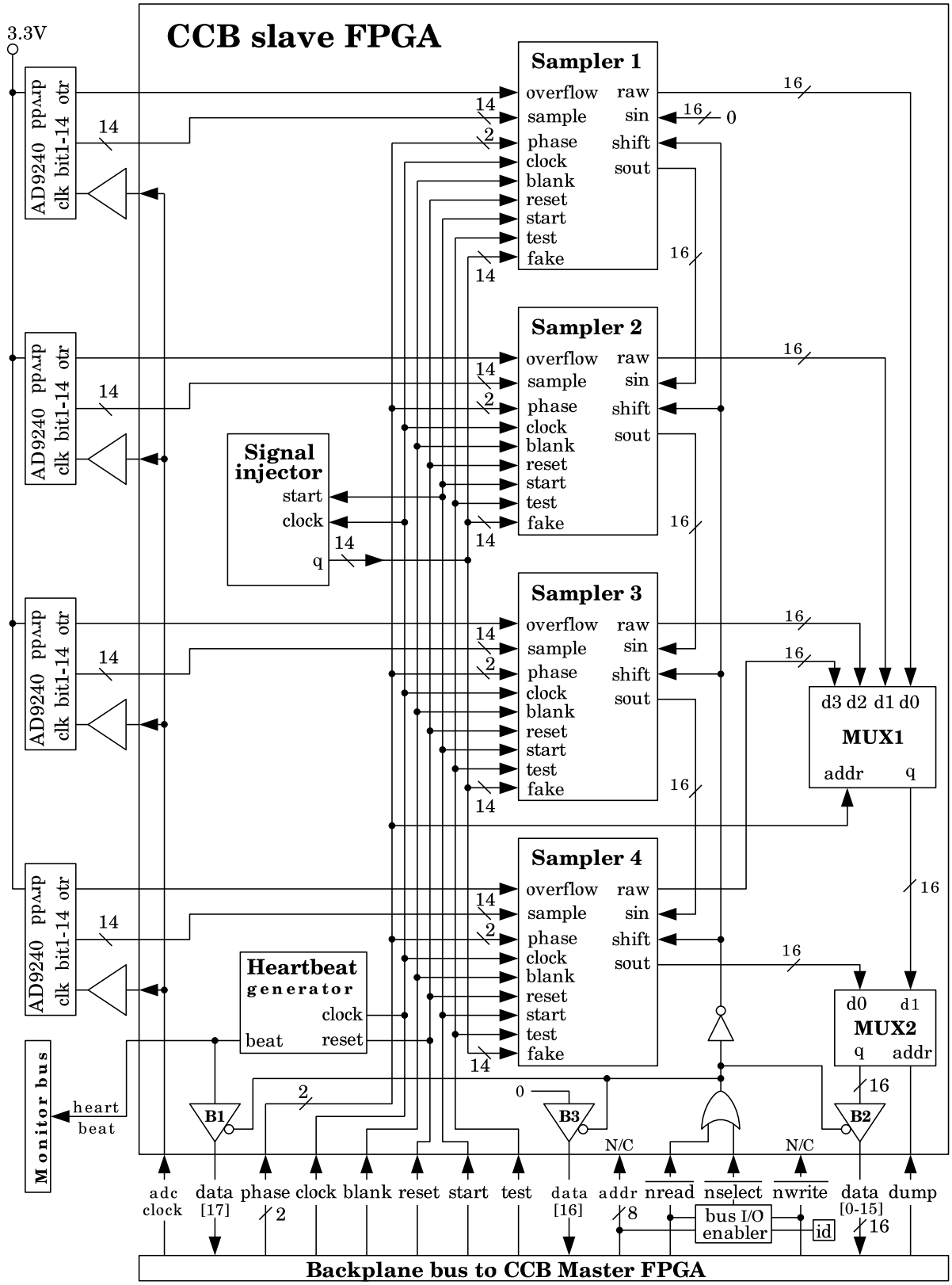

Figure 2.1: The top-level design of the slave FPGA

the `phase` signal, are initialized with the first sample of the next integration period, while the remaining bins are zeroed.

The output buffers of the *Sampler* components, take the form of PISOs (Parallel In Serial Out). The `sin` inputs and `sout` outputs of the PISOs within each `Sampler` component, are chained together to form one long PISO that contains the final integrations of all of the *Sampler* components.

The active-low $\overline{\texttt{nselect}}$ control-signal is asserted when the `addr` signal contains the board-ID of the slave, and either of the active-low $\overline{\texttt{nread}}$ or $\overline{\texttt{nwrite}}$ strobes is asserted. This tells the slave that the master wishes it to transfer data over the data-bus, in the direction that is indicated by whether the $\overline{\texttt{nread}}$ signal or the $\overline{\texttt{nwrite}}$ signal is asserted. In the current design the master never sends anything to the slaves over the data-bus, so the $\overline{\texttt{nwrite}}$ strobe is simply ignored by the slave FPGAs.

When the $\overline{\texttt{nread}}$ signal is asserted, the addressed slave responds by sending the master either integrated, or raw ADC samples, depending on whether the `dump` signal is asserted. The master asserts the $\overline{\texttt{nread}}$ strobe just after the rising edge of the clock. Until the next clock edge, all that this does is enable the tri-state output buffers of the addressed slave FPGA, to drive the first sample onto the data-bus. One clock cycle later, on the next rising edge of the clock, the data-bus lines are assumed to have settled, so the master FPGA reads the initial sample off the data-bus. At the same time, the PISOs in the *Sampler* components see the asserted $\overline{\texttt{nread}}$ strobe, and clock out the next data sample, ready to be read by the master, another clock cycle later. Subsequently, samples continue to be clocked out on the rising edges of the clock, until the $\overline{\texttt{nread}}$ strobe is deasserted again by the master.

The asserted $\overline{\texttt{nread}}$ strobe also causes the addressed slave to drive a bussed copy of its heartbeat signal, `data[17]`, as well as the currently unused `data[16]` output signal onto the data-bus.

The source of the output `data` signal of a slave FPGA is determined by *MUX2*. In normal integration mode, this selects the output of the integration PISO. In dump-mode, it selects one of the raw *Sampler* outputs.

The `phase` control-signal has different interpretations in the two acquisition modes. In normal integration mode, it identifies the phase-switch bin that the latest sample should be added to, whereas in dump mode it identifies the *Sampler* whose raw samples are to be passed to the `data` output, via `MUX2`.

Note that in normal integration mode, new integrations are ready to be read-out from the slave's output PISO on the second rising clock-edge that follows the rising edge of the `start` signal.

2.1.1 The Heartbeat Generator

The slave FPGAs generate a heartbeat signal that has two uses. On the one hand, the external PC104 based monitoring system monitors a leaky average of the `heartbeat` output signal, which should be around half of the full-scale digital high voltage if the heartbeat is toggling correctly. On the other hand, the heartbeat signal is also driven onto the bus, when the slave is selected, so that the master FPGA can determine if that slave is present and showing signs of life. The generation of this heartbeat signal is shown in figure 2.2.

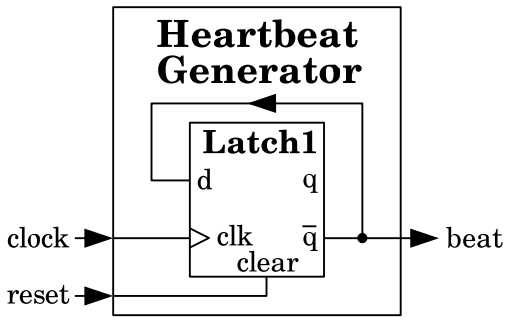

Figure 2.2: The Heartbeat Generator component

Since an FPGA that has been fried, or has failed to load its firmware, could unpredictably present any signal on its I/O pins, a dynamic heartbeat signal was chosen, with a known pattern that the master FPGA could check for. The pattern that is used, is simply a signal which toggles its state at each successive rising edge of the global clock. The master FPGA checks this at the start of each clock cycle, simply by using an XOR gate to compare a latched copy of the previous state of the heartbeat signal, to its current state. If the old and new heartbeat values of a given slave, aren't opposites, then that slave is flagged in the output data that are sent to the CCB computer.

2.1.2 The Signal Injector

The job of the *Signal Injector* is to generate repeatable pseudo-random fake ADC samples, for optional use by the *Sampler* components, in place of real ADC samples. The implementation, as shown in figure 2.3, is essentially a conventional linear-feedback shift-register, configured to generate 14-bit random positive integers. The sequence of random numbers repeats every $2^{14} - 1$ clock cycles, and within this period, each number between 1 and $2^{14} - 1$ is generated exactly once. To ensure that the results are repeatable for each integration, the sequence is re-started whenever the master FPGA asserts the `start` signal. This is done by asserting the `set` input of the shift-register, which sets all of the bits of the shift-register to 1. The

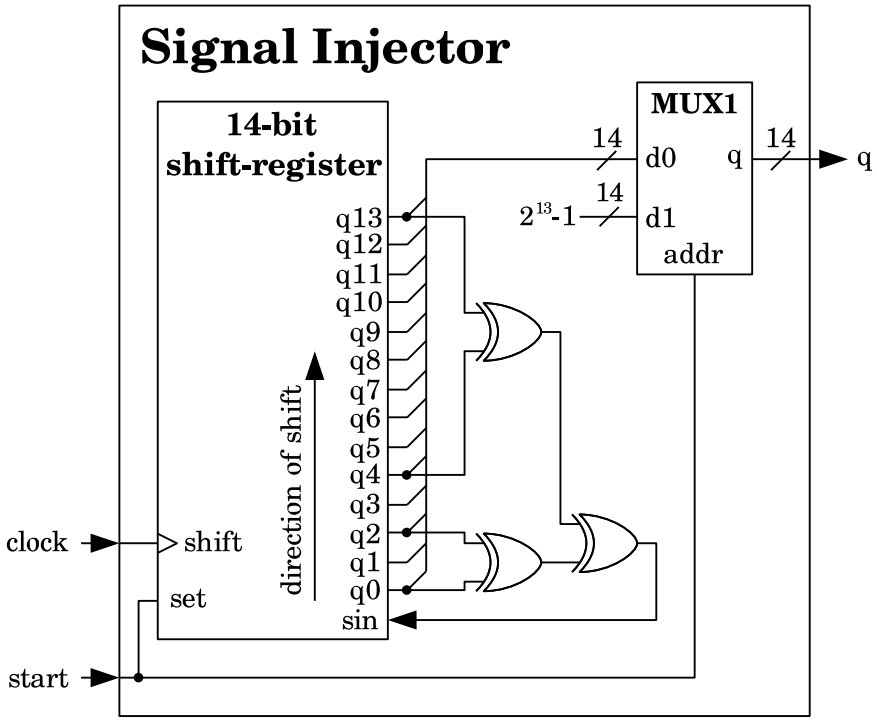

Figure 2.3: The Signal Injector component

first number of the new sequence is ready to be latched on the rising clock edge that follows the falling edge of the `start` signal. This is unfortunately one clock cycle too late for the integrators, which latch their first sample during the same rising clock edge as the *Signal Injector* is starting to reset itself. Thus while the *Signal Injector* is resetting itself, `MUX1` substitutes $2^{13} - 1$ for the otherwise unpredictable output value of the shift register. The value $2^{13} - 1$ was chosen because it is the end value of the pseudo-random sequence, and thus usually precedes the random number sequence returning to its initial value of $2^{14} - 1$. Thus, from the point of the integrators, the sequence of fake samples simply starts one number earlier in the cicrular sequence of pseudo-random numbers.

Note that if the value of the shift-register somehow becomes zero, then the generation of random numbers ceases. However, although glitches could potentially force the register into this state, the correct sequence will be started anew at the start of the next integration period, so automatic restarting hasn't been included. Automatic restarting would be of dubious utility anyway, since the operator wouldn't then see the repeatable test-sequence that they were expecting.

2.1.3 The Sampler component

The job of the *Sampler* component is to acquire raw samples from the ADC, integrate either these samples or fake ADC samples, within phase-switch bins, and present both the resulting integrations, and the real or fake samples, for collection by the master FPGA. The implementation is shown in figure 2.4.

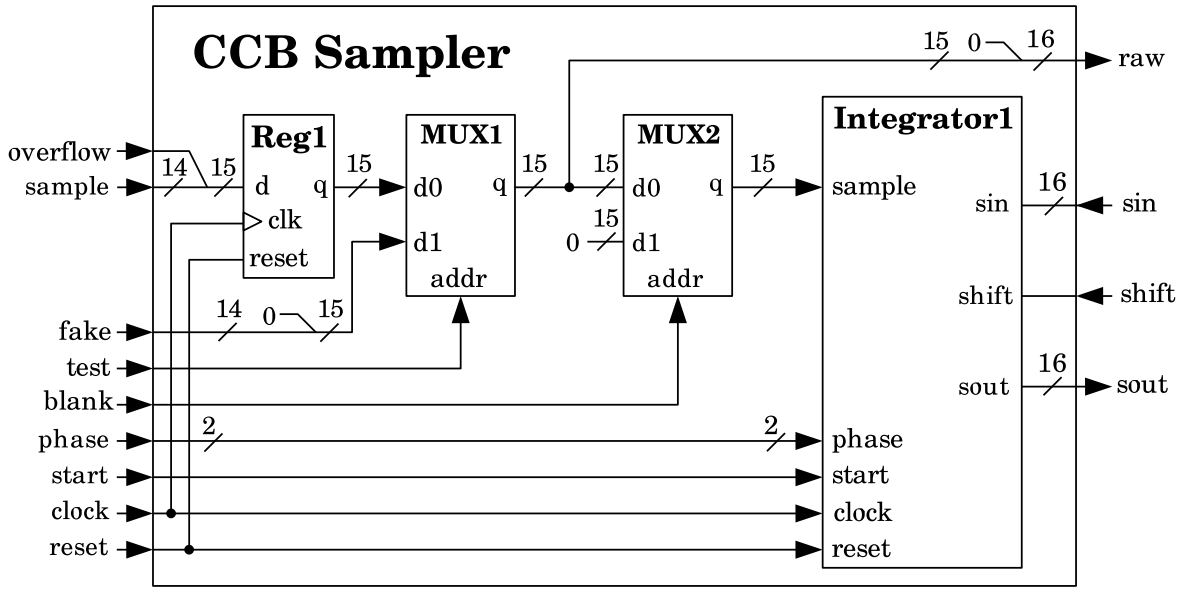

Figure 2.4: The Sampler component

Register *Reg1* uses the global FPGA clock to acquire successive sample and overflow signals from the external ADC. Multiplexer *MUX1* then takes either this sample and its overflow, or a fake sample, with no overflow, and presents these to multiplexer `MUX2`. Multiplexer `MUX2` either blanks the sample and overflow signals, by replacing them with zeroes, or presents them unchanged to integrator `Integrator1`. The integrator then routes the resulting sample and overflow signals to one of its 4 internal accumulators, according to the states of the phase switches, and the sample is added to that accumulator. The *Sampler* also taps off a copy of the sample and overflow bits, from before the blanking step, and presents these at the `raw` output, for dump-mode data-collection.

Within the currently selected accumulator, if an input sample either has its overflow bit asserted, or its addition to the integration would overflow the 32-bit accumulator, then the contents of the accumulator are replaced with a 32-bit number that has all bits set to 1. Thereafter, this state persists until the accumulator is reset for the next integration period.

The end of one integration period, and the start of the next, is signaled by the `start` input signal, which the master FPGA asserts for one clock cycle. When this is asserted, the contents of the integration bins are copied into an output PISO, within the integrator component, and

the integration bins are prepared for the new integration. Preparation for the new integration involves initializing the accumulator of the currently selected phase-switch bin with the value at the output of `MUX2`, and zeroing the accumulators of the remaining 3 phase-switch bins.

The data-bus between the slaves and the master FPGA isn't wide enough to transmit all 32 bits of a given integration bin in one clock cycle. Thus, whenever the $\overline{\texttt{nselect}}$ input of a slave FPGA indicates that the master FPGA wants to read data from that slave, the slave tells the chain of 16-bit *Sampler* PISOs, to shift one 16-bit half of an integration bin onto the bus at the start of each clock cycle. This starts with the least significant 16 bits of the first accumulator of the fourth ADC.

2.1.4 The Integrator component

The function of the *Integrator* component has already largely been described in the documentation of the *Sampler* component, so this section just describes its implementation, which is shown in figure 2.5.

Most of the work of an *Integrator* component is performed by four embedded *Accumulator* components, each of which represents one of the 4 phase-switch integration bins. Although each new sample is seen by all of the *Accumulator* components, only the *Accumulator* whose `sel` input is asserted, considers the sample for addition. The `phase` input, decoded by the `Decoder` instance, thus determines which *Accumulator* gets the latest sample, at the start of each new clock cycle.

The individual *Accumulator* components contain small PISOs that are chained by the parent *Sampler* component, to form the PISO that the parent *Sampler* clocks.

2.1.5 The Accumulator component

The *Accumulator* component accumulates the samples of a particular phase-switch integration bin, as described in the documentation of the *Sampler* component. It's implementation is shown in figure 2.6.

In the diagram, the combination of the `Adder` component and register `Reg0`, form the accumulator cell that is used to integrate samples. This updates every clock cycle, regardless of whether or not the accumulator bin is selected by the parent *Integrator* module. Thus multiplexer `MUX1` is used to add zero, instead of a new sample, at times when the input sample should be ignored.

At the start of a new integration period, as indicated by the `start` input being asserted for one clock cycle, multiplexer `MUX0`, which normally feeds back the previous value of the registered output of the adder to the `d0` input of the adder, substitutes a value of zero, to

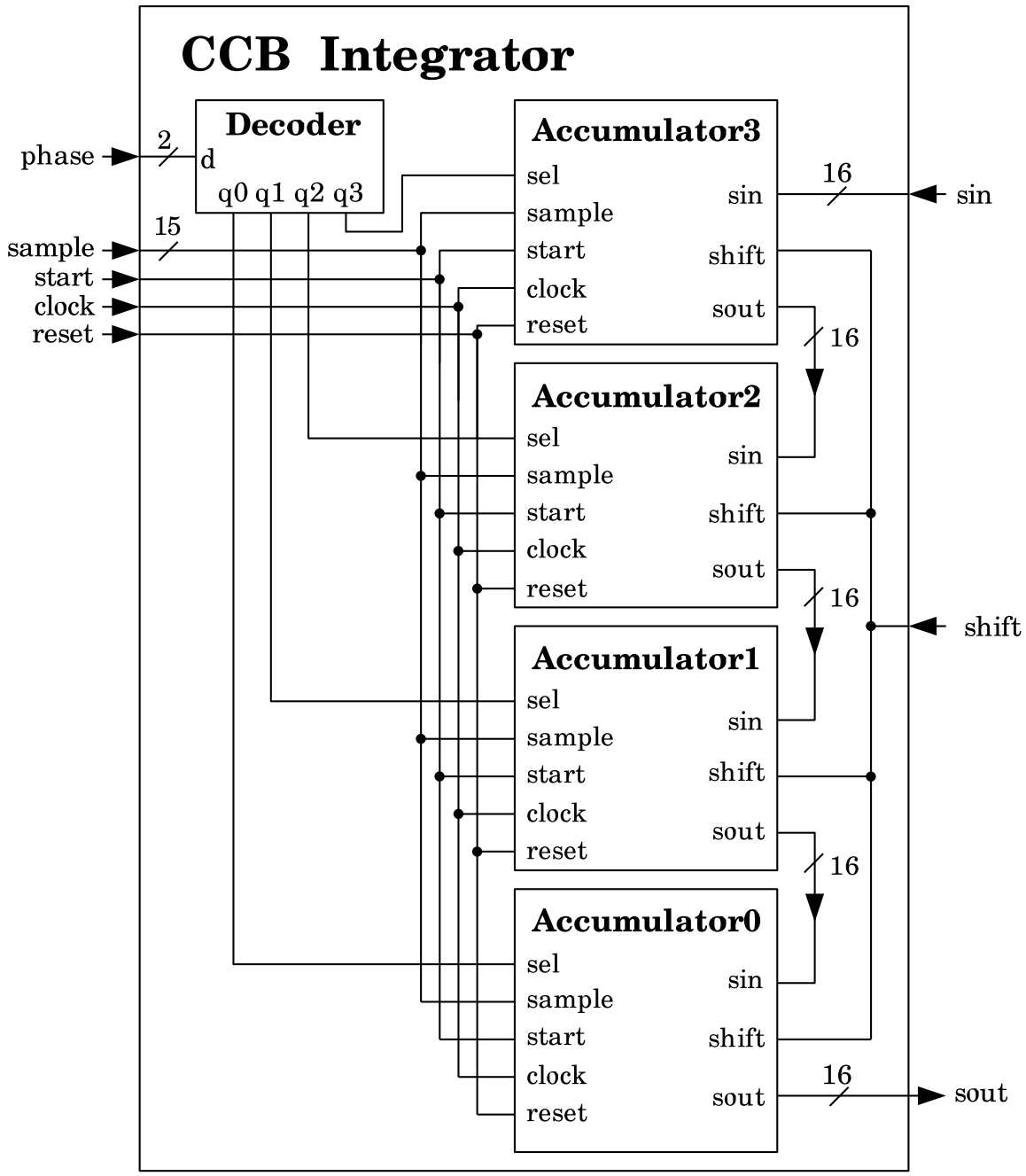

Figure 2.5: The Integrator component

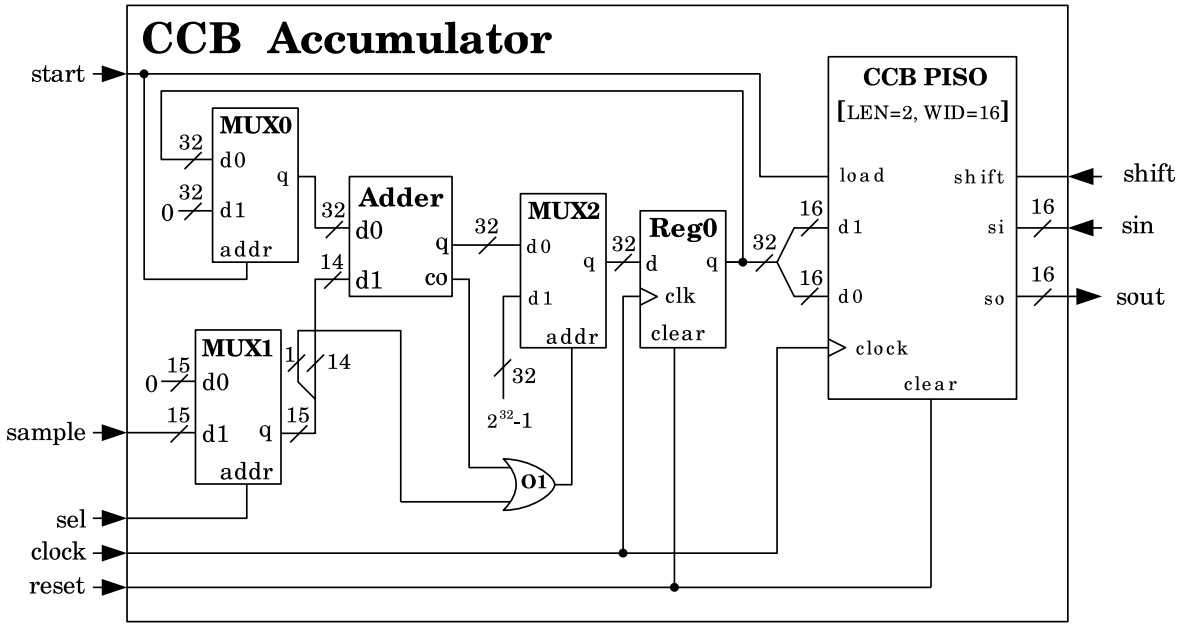

Figure 2.6: The Accumulator component

discard the previous accumulation. The initial output of the adder thus becomes equal to the value at the `d1` input of the adder, which is either equal to the 14 least-significant bits of the `sample` input, if the accumulator is selected for integration, or to zero otherwise. In the former case, whether the initial sample is then latched from the output of the adder into register `Reg0`, depends on the state of the overflow bit of the sample, which is the topmost bit of the `sample` input. If this bit is asserted, then instead of the initial sample value being latched to the accumulator output, the accumulator is initialized with the value $2^{32} - 1$, which is used to indicate an overflow condition to subsequent analysis software.

By the start of the next clock cycle, the master FPGA has deasserted the `start` input. On this and subsequent clock cycles, the accumulator continues to behave as already described for the initial clock cycle of the integration period, except that the registered output of the adder is fed back to the `d0` input of the adder, instead of zero.

If the 32-bit adder overflows, or the overflow bit of the sample is set when the *Accumulator* is selected, the registered output of the adder is set to the special value $2^{32} - 1$. This is the largest number that will fit into a 32-bit unsigned integer, so attempting to add any further non-zero samples to this, causes the `Adder` component to assert its `co` output, which causes `MUX2` to reinstate the special value. Similarly, adding a sample whose value is zero, leaves the special value unchanged. Thus once an overflow has occurred, the special value persists at the output of the registered adder, until this value gets discarded by `MUX0`, at the start of the next integration period.

The *CCB PISO* component following the accumulator, is a two-entry 16-bit-wide PISO, used

to stream the 32-bit output of the accumulator, in two 16-bit chunks, to the master FPGA, followed by those of other *Accumulator* components. This customized PISO component is documented in section 3.4.3. On the first rising edge of the clock that follows the `start` signal going high, at the start of a new integration, the accumulator register is initialized with the output of the adder, at the same time that the previous output of the accumulator register is being latched into the PISO. One clock cycle later, the output of the PISO will have settled to hold the least significant 16 bits of the accumulated integration. Thus integrated data can safely start to be read out from the accumulators two clock cycles after the `start` signal goes high.

Thereafter, whenever the `shift` input of the PISO is found to be asserted during the rising edge of the clock, the PISO is clocked to output the next 16-bit chunk. The first time that this happens, the initial output of the PISO is replaced by the 16 most significant bits of the parent accumulator. The second time it happens the least significant 16 bits of the preceding *Accumulator* in the chain of *Accumulator* PISOs, is presented, etc.

Chapter 3

The master FPGA

Figure 3.1 shows the layout of the master FPGA, showing its major internal components, along with their interconnections, and all of the external I/O-pin connections to external chips. The *State Generator* component, which can be seen as the central brain of this design, orchestrates the timing and the values of all control-signals that go to the other components within the master FPGA, as well as the control-signals that go to the slave FPGAs. The *State Generator* is in turn told what to do by the computer, via the *Control Gateway* component, which handles all interactions with the parallel port interface. The *Data Dispatcher* component is responsible for sending integrated and dump-mode data to the computer, via the USB interface. Finally, the *Heartbeat Generator*, which is identical to the heartbeat generators of the slave FPGAs, generates a signal that can be monitored by the computer, via a PC104 I/O card.

3.1 The Control Gateway

The *Control Gateway* handles all interactions with the CCB computer's EPP parallel port interface. It provides an 8-bit register-based interface for the CPU to use to send commands and configuration data to the *State Generator*, allows read-back of these same registers, and lets the *State Generator* interrupt the CPU via the parallel port interrupt line.

In addition, the reset signal of the EPP parallel port can be used at any time by the device driver in the CCB computer, to reset the firmware and the USB chip. This will automatically be done whenever the device driver is newly loaded.

The implementation of an 8-bit register-based interface, for use by the computer, is simplified by the built-in support for separate address and data cycles in standard EPP hardware. Since both of these targets have read and write cycles, there are 4 distinct I/O cycles, which are assigned to CCB operations as follows:

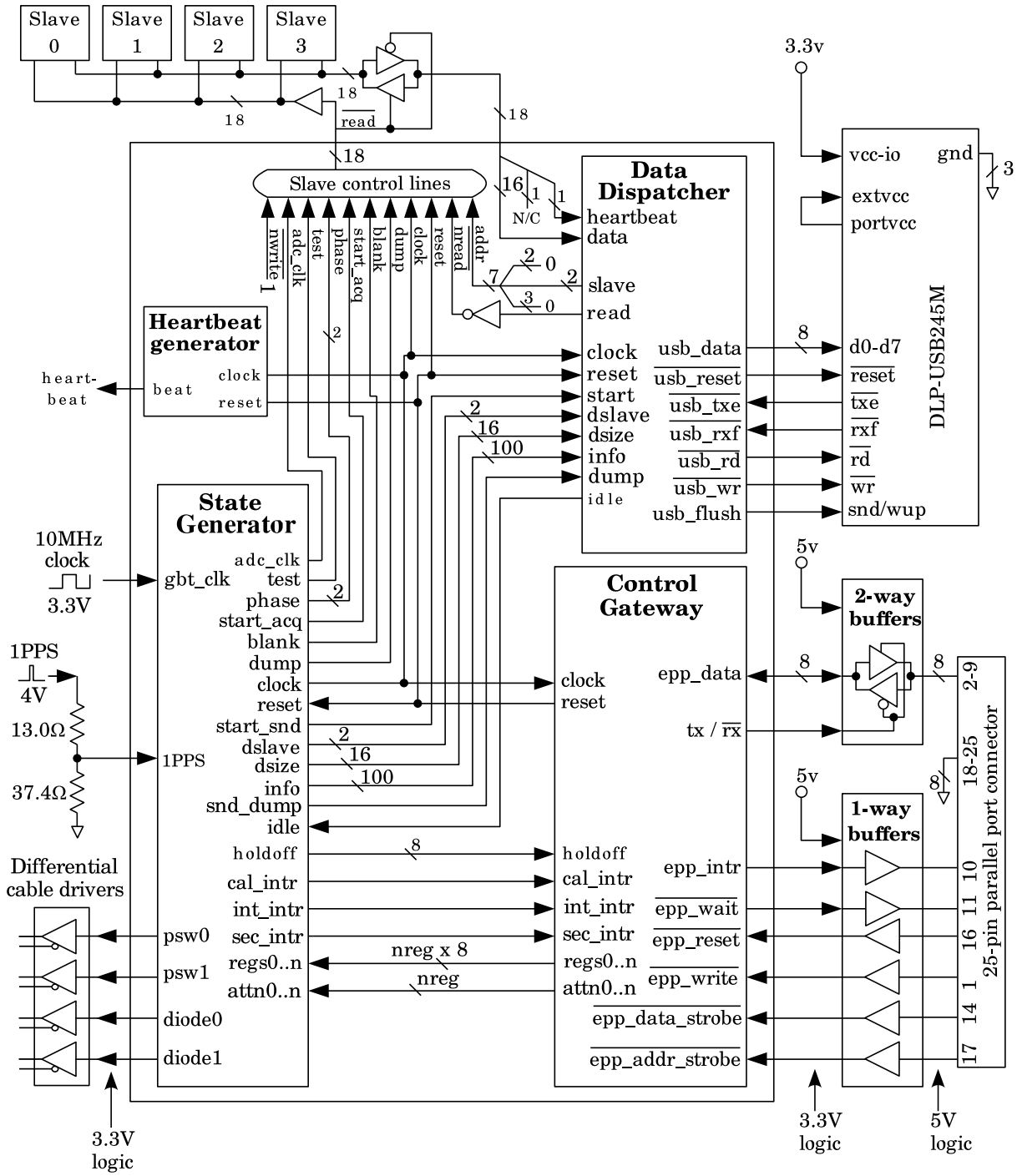

Figure 3.1: The top-level design of the master FPGA

- **The address write-cycle**

  The associated data-byte is interpreted as the address of one of the registers in the FPGA. Subsequent data-read and data-write cycles read from and write to the addressed register.

- **The data write-cycle**

  The associated data-byte is copied into the register that was indicated during the last address write.

- **The data read-cycle**

  The returned data-byte is the value of the register that was indicated during the last address write.

- **The address read-cycle**

  When the CPU initiates an address-read cycle, the FPGA responds by returning the bit-mask of all FPGA event-sources that have requested interrupts since the last time that the computer executed an address-read cycle.

There are only two periods when data are sent to the master FPGA by the computer.

1. When starting a new scan, a write to the control register is used to prepare the *State Generator* for reconfiguration. This is followed by multiple EPP write-cycles to send the configuration data of the new scan. The last such write is to the register which instructs the *State Generator* to activate the new scan.

   Note that since the FPGA does nothing with the configuration data that it is sent, until it is told to start the next scan, it is safe to send the values of multi-byte configuration registers, one byte at a time.

2. During a scan, the CPU sends the FPGA a single byte of integration-specific configuration data whenever the FPGA generates a configuration interrupt. At the start of a scan, this happens repeatedly, until the FIFO that queues these bytes fills up. Thereafter, integration-configuration interrupts are sent at the end of each integration, as the removal of one integration-configuration byte from the FIFO, makes room for another.

   Since between scans, only the integration-configuration register is written to, the device driver need not keep sending the address of the integration-configuration register before each data write. Instead it sends it once, just after the command byte that activates a new scan.

   Thus, on average, each such interrupt will cause an EPP address-read to get the interrupt mask, plus one EPP write to send the FPGA the configuration of the next un-configured integration. Once the configuration FIFO is full, this happens once per integration.

3.1.1 The internals of the Control Gateway

When configuration data and commands are received from the computer, they are recorded in a bank of 8-bit registers. The values stored in this bank of registers are included in the outputs of the *Control Gateway*, and are thus visible to the *State Generator*. The control gateway treats all of the registers alike, leaving the interpretation of their contents to the *State Generator*, where individual registers are interpreted, either as commands to be executed on receipt, or as configuration data. The registers are updated synchronously with the FPGA clock, and whenever a particular register is updated by the CPU, its `attn` (ie. attention) output is held high for one clock cycle, to indicate to the rest of the CCB that the register has been updated. The stored register values can also be read back by the CPU, via EPP data-reads.

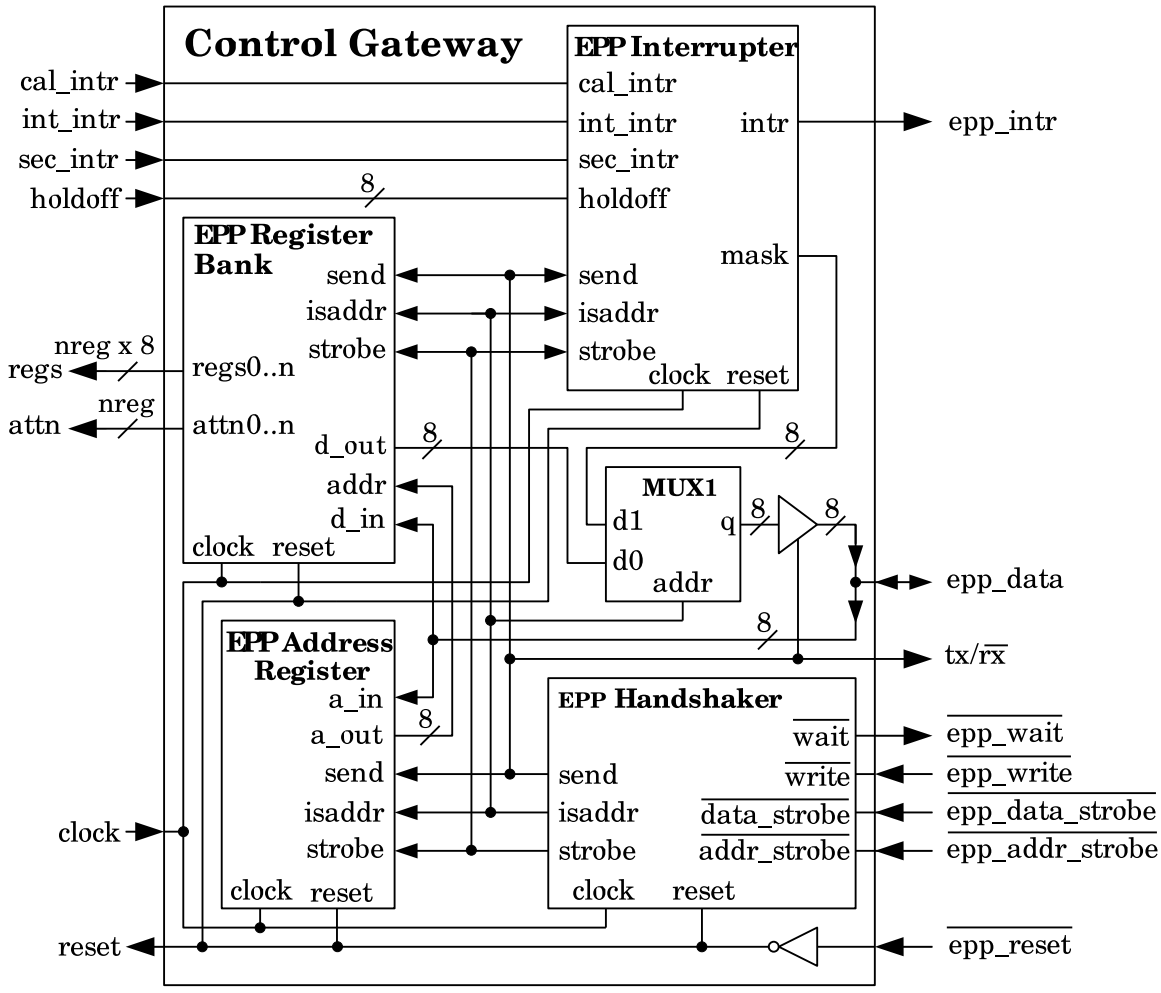

Figure 3.2: The Control Gateway

Since only one register can be read from or written to by the CPU in a single EPP transaction,

a way is needed for the CPU to specify which register is to be the current I/O target. As previously mentioned, to do this, the CPU uses an EPP address-write transaction to send the 8-bit address of the register of interest. On receiving such an address, the *Control Gateway* stores it in the *EPP Address Register*. Thereafter the output of the *EPP Address Register* is used by the *EPP Register Bank*, to route any subsequent EPP data transactions to the specified register.

The *EPP Interrupter* allows multiple interrupt sources in the FPGA to share the single parallel-port interrupt line. When the CPU receives a parallel-port interrupt, it responds by performing an EPP address-read, which both acknowledges the interrupt, and asks the FPGA which FPGA event-sources requested the interrupt. The *EPP Interrupter*, which is told about the address-read by the *EPP Handshaker*, responds by sending the CPU an 8-bit interrupt mask, whose individual bits indicate which event-sources have requested interrupts since the last time that the mask was read by the CPU.

The *EPP Interrupter* has a `holdoff` input, whose value is the minimum number of clock cycles to wait after sending one interrupt, before sending another. This both prevents interrupts from being sent too frequently, and sets the rate at which unacknowledged interrupts are to be re-sent. Note that there is no danger that a re-sent interrupt will be interpreted by the CPU as indicating two events in the FPGA, since it is the contents of the interrupt mask, rather than the number of interrupts received, that matters, and the mask is automatically cleared as part of the read operation.

To avoid a tug-of-war with the CPU, the FPGA only drives the `data` lines when explicitly requested, as indicated by the `send` output of the *EPP Handshaker* being asserted. Thus the tri-state output buffers in the I/O blocks of the data-line pins, and the external `data` line transceivers are configured to passively receive data from the computer, except when the `send` signal is asserted.

The EPP Handshaker

The *EPP Handshaker* module, as depicted in figure 3.4, is responsible for responding to the standard EPP handshaking signals for all single-byte EPP transfers.

The timings of the two standard EPP I/O cycles are shown in figure 3.3. Note that the $\overline{\texttt{strobe}}$ signal represents either the $\overline{\texttt{addr\_strobe}}$ or $\overline{\texttt{data\_strobe}}$ signals, depending on whether an address-write or data-write cycle is in progress, and that the $\overline{\texttt{write}}$, $\overline{\texttt{data\_strobe}}$, $\overline{\texttt{addr\_strobe}}$, and $\overline{\texttt{wait}}$ EPP signals are all active-low. The $\overline{\texttt{write}}$ and $\overline{\texttt{strobe}}$ signals are generated by the computer, while the $\overline{\texttt{wait}}$ signal is generated by the FPGA. The 8-bit `data` signal is generated by the computer when performing an EPP write-cycle, and by the FPGA when performing an EPP read-cycle.

At the start of each FPGA clock-cycle, the value of the $\overline{\texttt{wait}}$ signal is derived from the previous value of this signal, using the value of the $\overline{\texttt{write}}$ signal, and the value of the

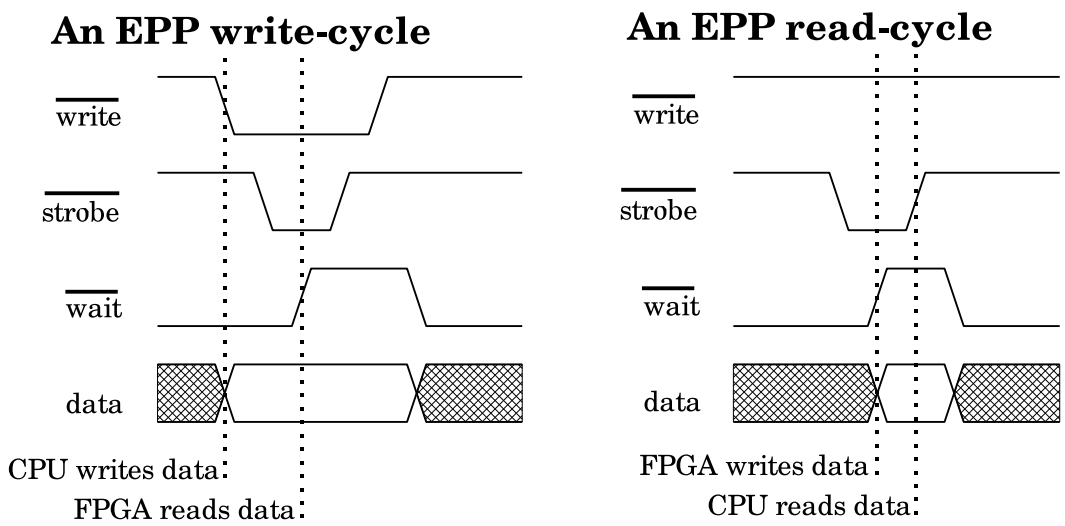

Figure 3.3: The standard EPP I/O cycles

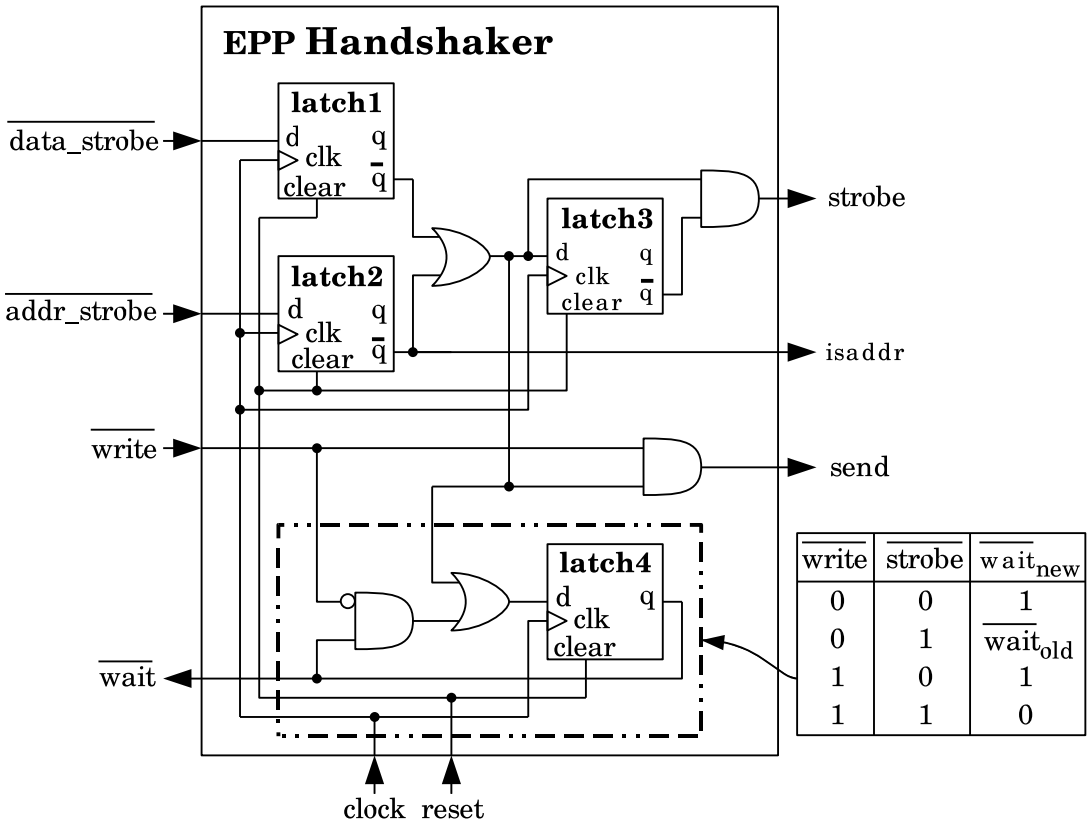

Figure 3.4: The EPP Handshaker

appropriate strobe signal, according to the truth table shown to the right of figure 3.4. The circuit within the dashed box implements this truth-table.

The $\overline{\texttt{data\_strobe}}$ and $\overline{\texttt{addr\_strobe}}$ inputs of the circuit in the dashed-box, are pre-conditioned by latches 1 and 2, which both re-time them to rise and fall in sync with the FPGA clock, and also invert them. The asynchronous signals at the inputs of these latches will periodically violate the latch setup and hold times, resulting in the latch output signals being metastable for an indefinite duration. Thus it is important that the outputs of latches 1 and 2 be given one clock cycle to settle, before they are used. Other *Control Gateway* components are thus required to use latches which only respond to the outputs of the *EPP Handshaker* when the `strobe` output is asserted at the start of a clock cycle. Note that `latch3` ensures that the `strobe` output is asserted for at most one clock cycle after either of the input strobes becomes asserted. Thus the other components in the `Control Gateway` will see an asserted `strobe` signal just once per EPP strobe, one clock cycle after latches 1 or 2 see the EPP strobe. Latch 4 ensures that the $\overline{\texttt{wait}}$ signal also goes high one clock cycle after each EPP strobe, in sync with the FPGA transferring data to and from the EPP data lines.

Note that a side-effect of synchronizing the EPP signal with the local clock is that it potentially adds either 1 or 2 FPGA clock cycles to the handshaking delay, and thus reduces the possible throughput. However, since the CCB won't be streaming large amounts of data through the parallel port, this shouldn't be important. The bottom line is that the rising edge of the output $\overline{\texttt{wait}}$ signal follows the falling edge of the pertinent $\overline{\texttt{strobe}}$ signal by between 1 and 2 FPGA 10MHz clock cycles, and this corresponds to between 0.8 and 1.6 EPP 8MHz clock cycles. Thus each EPP I/O transaction will be lengthened from the standard 4-cycle minimum duration, to either 5 or 6 8MHz cycles, and thus last either 0.625μs or 0.75μs, instead of 0.5μs.

Note that, unlike the $\overline{\texttt{strobe}}$ signals, the $\overline{\texttt{wait}}$ and $\overline{\texttt{write}}$ signals aren't pre-latched, since the EPP protocol assures that they will have stabilized before the pertinent $\overline{\texttt{strobe}}$ signal is driven low, and remain stable until after the $\overline{\texttt{wait}}$ line is next driven high.

The `isaddr` output signal tells the other components of the *Control Gateway* that the `strobe` signal represents an address transaction. Similarly the `send` output indicates whether this is an EPP read (`send=1`) or EPP write (`send=0`) transaction. Since these signals have one clock cycle to settle, before other components see an asserted `strobe` output signal, they can be used to drive routing logic, such as multiplexers, that also need time to settle, before the strobe causes data to be latched through them, to or from the data lines.

The EPP address register

The *EPP Address Register*, as shown in figure 3.5, holds the address of the target data-register of subsequent EPP data-write and data-read cycles. It is implemented using an 8-bit register with a synchronous enable-input, `ien` (see section 3.4.2). At the start of most clock cycles, the existing value of the enable input is not asserted, so the register retains its

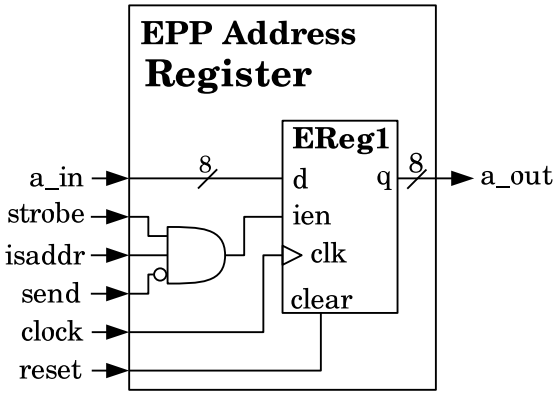

Figure 3.5: The EPP Address Register

current value. However, when the `strobe`, `isaddr` and `send` inputs indicate that an EPP address-write transaction is in progress, the asserted `ien` input of `EReg1`, causes the signals on the EPP data lines (at the `a_in` input) to be loaded into the register.

The `a_out` output is permanently connected to the `addr` input of the *EPP Register Bank* module, and thus specifies which register in the bank of registers, is to be addressed in subsequent data-register I/O transactions.

The EPP Register Bank

The *EPP Register Bank*, as shown in figure 3.6, contains the registers that are used to record and provide read-back of configuration parameters and command opcodes sent by the CPU. The `addr` input, which comes from the *EPP Address Register* module, selects which register should present its contents at the `d_out` output, and which register should latch a new value from the `d_in` input, when an EPP data-write transaction is in progress. The current values of all of the registers are also made available to the *State Generator*, at the `regs0..N` outputs, and whenever any register is updated, the corresponding `attn0..N` output is asserted for one clock cycle to inform the *State Generator*.

When an EPP data-write transaction is initiated, the *EPP Handshaker* deasserts the `isaddr` and `send` inputs of the *EPP Register Bank*, then asserts the `strobe` signal for up to one clock cycle, until just after the clock cycle at which the value on the EPP data lines, presented at the `d_in` input, should be latched into the currently addressed register. To this end `DMUX1` routes the output of AND gate `A1` to the `load` input of the currently selected register, which latches the data at its `d` input at the start of the clock cycle at which it sees that `load` has become asserted.

Note that although the 8-bit width of the `addr` input would allow up to 256 registers to be

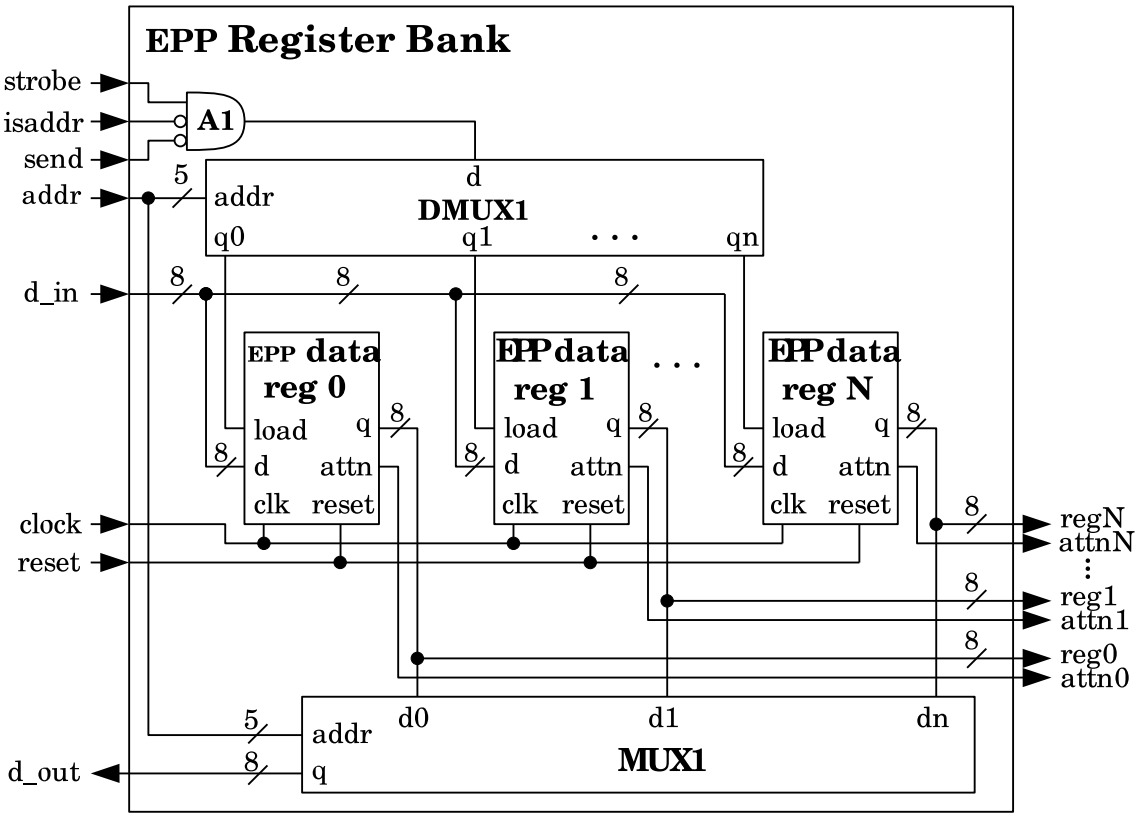

Figure 3.6: The EPP Register Bank

addressed, much fewer registers are actually needed, so the smaller number of bits shown going to `MUX1` and `DMUX1`, has been chosen to accommodate the preliminary list of registers given in section A.

The individual data registers are implemented as *EPP Data Register* modules, as shown in figure 3.7. During clock cycles when an asserted `load` input indicates that an EPP data-write transaction to this register is in progress, the asserted input-enable input (`ien`) input of latch `EReg1` (see section 3.4.2), causes the signals on the EPP data lines, at the `d` input, to be loaded into the register. Simultaneously, `Latch1` latches the asserted `load` input to the `attn` output, and thus indicates to the *State Generator* whenever the register is updated. During all other clock cycles, the contents of the register remain unchanged, and the `attn` output is held low.

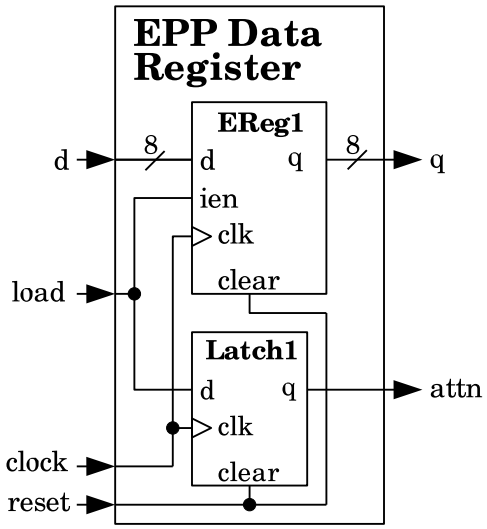

Figure 3.7: An EPP Data Register

The EPP Interrupter

The implementation of the *EPP Interrupter* module is shown in figure 3.8.

As explained shortly, the CCB FPGA has three sources of interrupt-worthy events, all of which share the single parallel-port interrupt line (`intr`), under the auspices of the *EPP Interrupter* module. As such, the receipt of a parallel-port interrupt by the computer does not necessarily imply the occurrence of any particular new event in the FPGA. What it does tell the computer is that it should perform an EPP address-read to find out which events have occurred since the last time that it performed such a read. The resulting loose association between individual events and parallel-port interrupts, reduces the number of interrupts that the CPU has to handle, and allows a repeat interrupt to be sent if the computer appears

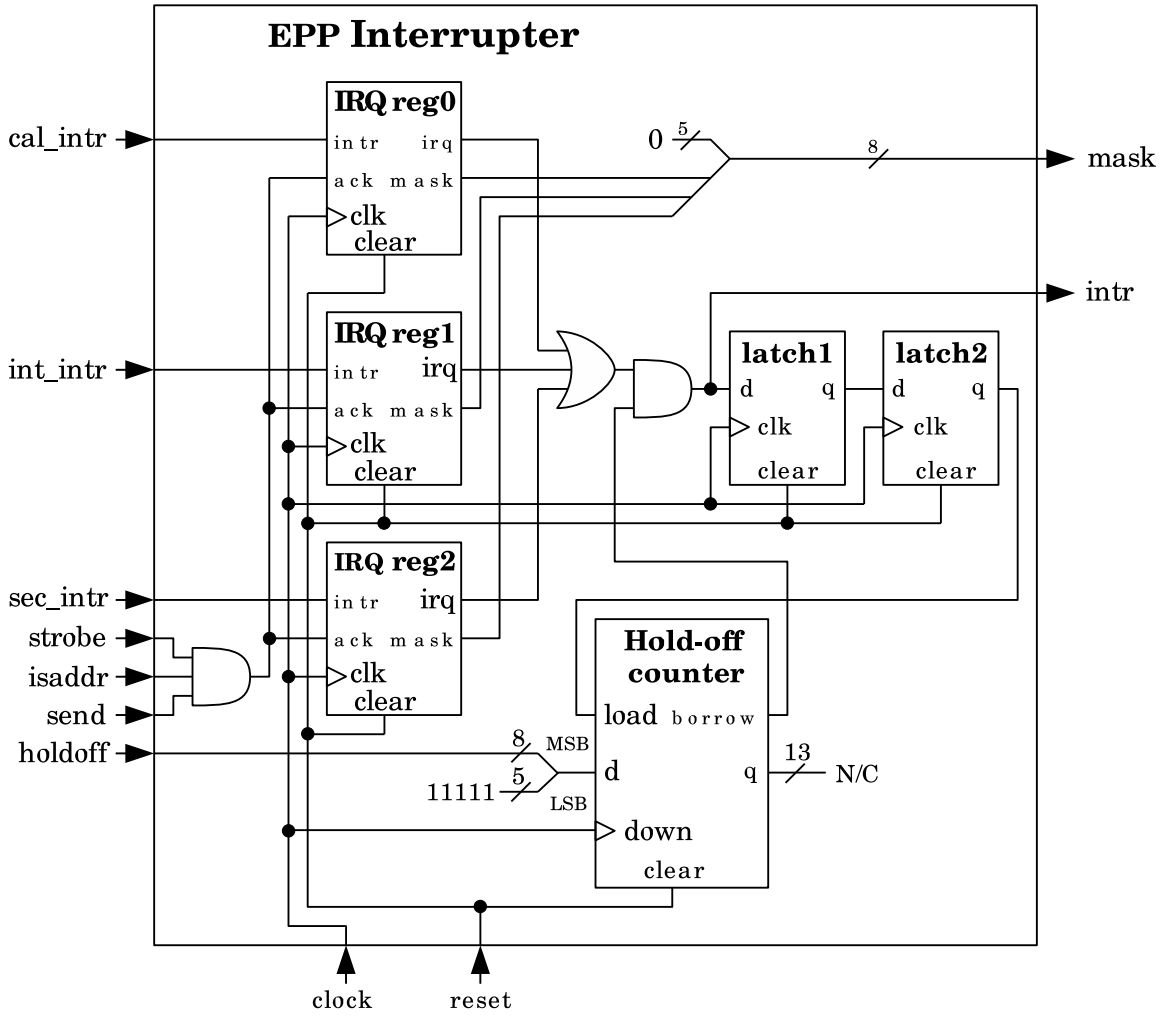

Figure 3.8: The EPP Interrupter module

to have missed the previous one, without any danger of the computer incorrectly believing that a repeated interrupt represents a new event. Similarly, the only harm that spurious interrupts can do is steal a bit of CPU time, since the bit-mask of events returned by the subsequent EPP address-read, after a bogus interrupt, will indicate that nothing has really happened.

Interrupts are sent to the CPU at most once every `holdoff` clock cycles. In particular, once any interrupt source has requested an interrupt, a new CPU interrupt is sent every `holdoff` clock cycles, until the computer performs an EPP address-read to get the bit-mask of previously unreported events.

When a particular event-source in the FPGA wishes to notify the computer of a new event, it synchronously asserts the associated one of the `cal_intr`, `int_intr` or `sec_intr` interrupt-request inputs of the *EPP Interrupter* for one clock cycle. Just after the end of this clock cycle, the corresponding IRQ (interrupt-request) register becomes asserted, and remains asserted until the computer next performs an EPP address-read to query which event-sources have requested interrupts.

The *EPP Interrupter* examines the `irq` outputs of the IRQ registers at the start of each clock cycle, and if any of them are asserted, and the hold-off counter isn't still counting down from the previously sent interrupt, it raises the parallel-port `intr` signal to interrupt the CPU, and holds this signal high for two FPGA clock cycles (ie. 1.6 EPP 8MHz clock cycles). Simultaneously, it reloads the hold-off down-counter with the number of clock cycles that it should hold-off the generation of the next interrupt.

When the computer responds to the receipt of an interrupt, by performing an EPP address-read, the *EPP Handshaker* asserts the `isaddr` and `send` inputs, then asserts the `strobe` input for up to one clock cycle, until just after the clock cycle at which the bit-mask of unreported events should be latched onto the EPP data lines. At the end of this clock cycle, all of the IRQ registers respond to this by moving their current `irq` output signals to their `mask` outputs, while resetting their `irq` outputs (unless a new interrupt is simultaneously being requested).

Note that if an event-source requests a new interrupt while its IRQ register is still asserted from a previous unacknowledged request, the new request is lost. A way to prevent such losses would be to implement the IRQ registers using up/down counters. New interrupt requests would increment these counters, and acknowledgements would decrement them. The first iteration of this design did just that. However, interrupts generally represent events that require a response while the interrupting event is still relevant, so queuing out-dated interrupts is pointless. Furthermore, anytime that EPP interrupts were disabled, the CCB would quickly queue hundreds of unacknowledged events, which when interrupts were subsequently re-enabled, would then keep the CPU busy for a while acknowledging stale events. For these reasons, the idea of using up/down counters was abandoned, and it was decided that it made more sense to simply design the event-sources and the device driver around a limitation of one queued event per interrupt source, per EPP address-read.

Figure 3.9, shows the internals of a single IRQ register.

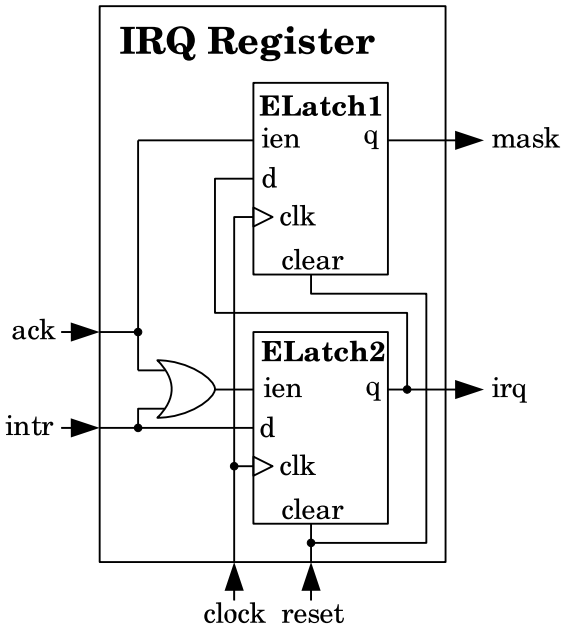

Figure 3.9: An Interrupt Request (IRQ) Register

When the event source associated with this register wishes to send an interrupt, it asserts the `intr` input for one clock cycle. At the end of this cycle (ie. the start of the next clock cycle), `ELatch2` (see section 3.4.1) becomes asserted. It stays asserted until the `ack` (acknowledge) input is subsequently asserted for one clock cycle, at which point, the value of `ELatch2` is transfered to `ELatch1`, while `ELatch2` adopts the current value of the `intr` input. Thus, whereas normally an asserted `ack` signal causes `ELatch2` to be cleared; if a new interrupt is requested at the same time as the `ack` input is asserted, `ELatch2` will record the new interrupt, instead of the new interrupt request being lost.

The end result is that the `irq` output reliably indicates whether the parent event-source has requested one or more interrupts since the last time that the `mask` output was updated by a pulse on the `ack` input.

The three interrupt sources that are envisaged at this point, are the following:

- `cal_intr` - Calibration-diode configuration interrupts.

  Before the start of each new integration, the *State Generator* needs to know the desired on/off states of the cal-diodes. In principle this could be sent one integration in advance, from an end-of-integration interrupt handler. That was the original plan. However, to soften the real-time requirements placed on the device driver in the CCB

embedded computer, and thereby make the CCB insensitive to occasional transient anomalies in Linux's interrupt latency, the current plan is to instead implement a FIFO containing the configurations of many integrations in advance, instead of just one. Keeping this FIFO filled is the job of the `cal_intr` interrupt. At the start of a scan, to fill the FIFO, multiple `cal_intr` interrupts are generated, each one telling the computer to send one new calibration-diode configuration, for one or more consecutive integrations that are to have the same configuration. Thereafter whenever the oldest configuration in the FIFO is exhausted, that configuration is discarded from the FIFO, and a new entry is requested by sending another `cal_intr` interrupt.

The rapid-fire `cal_intr` interrupts at the start of a scan are rate-limited in two ways. First, a new `cal_intr` input-signal is never raised by the *State Generator* until the CPU responds to the previous one by sending a new cal-diode configuration entry. Secondly, the `holdoff` timer of the *EPP Interrupter* sets a hard limit on the parallel-port interrupt rate, regardless of how quickly the CPU responds.

- `int_intr` - Integration-done interrupts.

  Integration-done interrupts are generated when one integration ends and another starts. If a new integration starts before the interrupt from the start of the previous integration has been acknowledged by the computer, the new interrupt request is simply discarded, but the previous integration request continues to generate retry interrupts at intervals controlled by the `holdoff` timer. Thus the CCB device driver should not count integration interrupts to determine how many integrations have been completed at a given time, and nor should it use this interrupt for anything that absolutely has to be performed within a small time frame following the boundary between 2 integrations.

  As mentioned in the discussion of the `cal_intr` input, originally integration-done interrupts were needed for sending cal-diode configurations one integration at a time. It isn't clear yet whether this event will be useful for anything else in the device driver, so for the moment, it is included here mostly as a placeholder, and may end up being removed.

- `sec_intr` - 1 second interrupts.

  A `sec_intr` interrupt is requested once per second, at the rising edge of the second FPGA clock cycle that follows the rising edge of the pulse of the external 1PPS signal (to avoid metastable latch states). Like the integration interrupt, if a previous 1-second interrupt hasn't been acknowledged by the time that a new one is to be generated, the new one is simply ignored, while the *EPP interrupter* continues to retry sending the original. Given the length of time between these interrupts, this should only happen when the CCB device driver isn't loaded, or if either the parallel cable or the computer are damaged.

By default, at boot time, EPP interrupts are disabled, and a write to the parallel-port configuration register is needed to enable them. While they are disabled, signals on the `intr`

interrupt line are simply ignored by the computer. Thus the FPGA doesn't redundantly provide its own way to enable and disable the generation of interrupt signals on the `intr` line. Note that the resending of unacknowledged interrupts every `holdoff` clock-cycles, ensures that interrupts that are missed while the parallel-port has interrupts disabled, get re-sent and acknowledged as soon as interrupts become enabled.

3.2 The Data Dispatcher

At the end of each integration period, and at the start of dump mode, the *Data Dispatcher* component reads integrated or dump-mode data from the slave FPGAs into a large FIFO, then streams the contents of this FIFO, preceded by a header, to the computer, via the USB bus. All communications over the USB bus are directed from the FPGA to the computer. Thus, although the read (`rd`) and read-enable (`rxf`) pins of the USB interface are shown as inputs to the *Data Dispatcher*, there are no plans to use them at the moment.

Note the use of the DLP-USB245M module. This is a tiny PCB module containing a 6MHz crystal, a surface-mount FT245BM USB1.1 chip, a USB connector and all the interconnections needed between these parts. The PCB is just 1.5×0.7 inches in size, and the USB connector sticks out a further third of an inch from one end. The module can be soldered onto the CCB PCB, via 24 dual in-line pins. Its data-sheet can be downloaded from:

`http://www.dlpdesign.com/usb/dlp-usb245m12.pdf`

The two of these modules that I bought for testing the FT245BM, I got from a company called Saelig (`www.saelig.com`), which is an official US distributor for the FT245BM. The modules arrived overnight. Since then, I have noticed that Mouser Electronics carries them as well. Their catalog number at Mouser is 626-DLP-USB245M, and they cost $25.

3.2.1 The internals of the Data Dispatcher

Figure 3.10 shows the building blocks of the *Data Dispatcher*, and how they are interconnected.

One clock cycle after the `start` input-signal is asserted, to tell the *Data Dispatcher* to collect and dispatch a new frame of integrated or dump-mode data to the computer, the *Slave Reader* asserts its `read` output, and keeps it asserted until all available data-samples have been transfered from the slave FPGAs into a FIFO within the *Frame Buffer*. At the rising edges of the clock, this signal is examined both by the *Frame Buffer* and by the currently addressed slave FPGA, and when it is found to be asserted, it causes the transfer of one 16-bit data-sample from the addressed slave to the *Frame Buffer*.

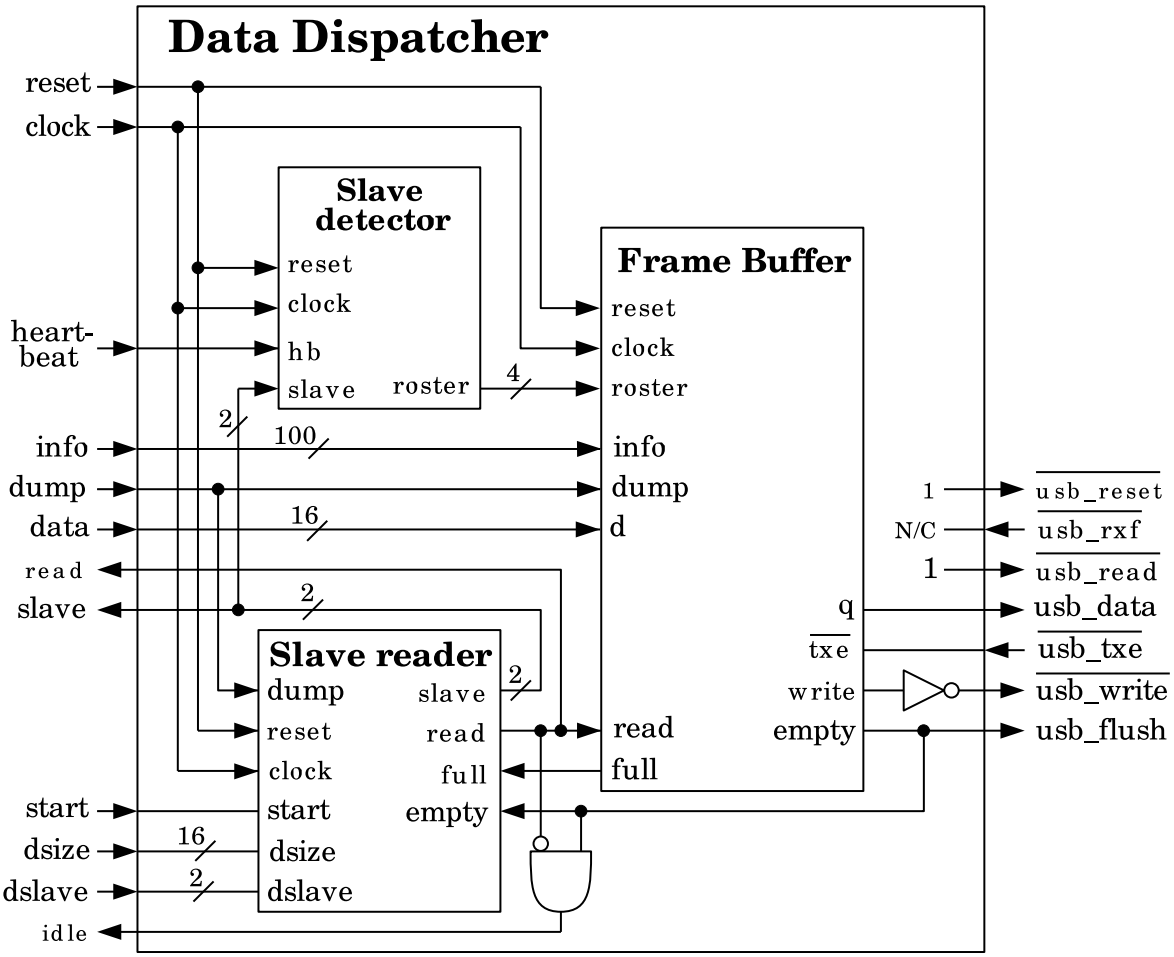

Figure 3.10: The Data Dispatcher

The currently addressed slave FPGA is the one identified by the `slave` output of the *Slave Reader*. In normal integration mode, this slave-address is first set to that of the highest numbered slave FPGA, and then, after all of that slave's samples have been transfered, to the next lower numbered FPGA, and so on, until the samples of all of the FPGAs have been transfered to the *Frame Buffer*. In dump mode, the `slave` output is simply assigned the value of the `dslave` input, which identifies the slave whose raw ADC samples are to be collected.

Both during and after the period when samples are being read from the data-bus into the *Frame Buffer*'s FIFO, the *Frame Buffer* streams the frame-header, followed by the contents of the FIFO, to the USB interface chip, 8 bits at a time.

Once all data in the *Frame Buffer* FIFO have been delivered to the USB chip, the `empty` output of the *Frame Buffer* is asserted, to tell the *Slave Reader* that it is okay for it to start collecting a new frame. At the same time, the USB chip's `flush` input is asserted, to tell it not to await any further data, before flushing the data that it has received so far, to the computer.

The *Slave Reader* de-asserts its `read` output, to terminate collection of the current data-frame, either when all data have been read from the slaves, or when the *Frame Buffer* indicates that its FIFO is full, and thus can't accept any more samples. The latter should only occur if the computer has asked for a dump frame that is too big to be accommodated by the *Frame Buffer*.

Note that since the *Slave Reader* doesn't allow the collection of a new data frame to be initiated until the *Frame Buffer* indicates that the previous one has been completely sent, and because the collection of a given frame of data is terminated if the FIFO becomes full, there is no danger of gaps in the collected data, caused by temporary overflow conditions in the *Frame Buffer*'s FIFO, or of a new frame trampling on the contents of a frame that hasn't been fully sent yet.

The `idle` output signal of the *Data Dispatcher* is asserted when the *Data Dispatcher* is not in the process of either collecting or sending a data-frame to the computer. This is used by the *State Generator* to determine when it is safe to terminate a scan.

The internals of the Slave Reader

The implementation of the *Slave Reader* is shown in figure 3.11.

As previously described, the *Slave Reader* selects one slave at a time to write to the data-bus, by way of its `slave` output, while, at the same time, asserting its `read` output, to tell both that slave, and the *Frame Buffer*, to transfer one sample over the data-bus, at each rising edge of the clock.

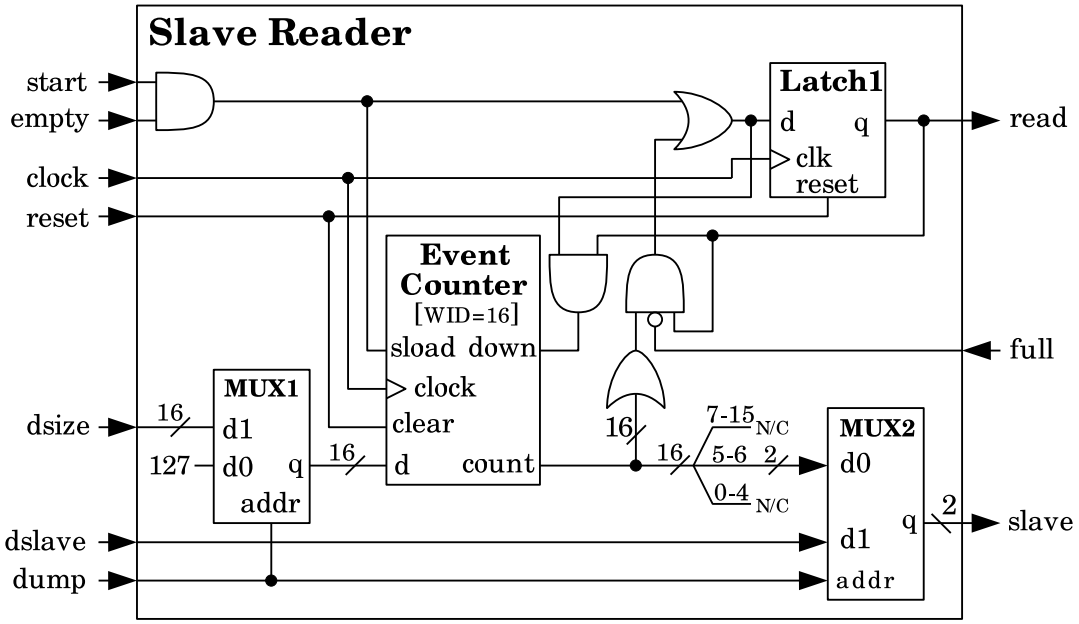

Figure 3.11: The Slave Reader

When the `empty` input signal is asserted, `Latch1` and the combinational logic around it, arrange for the `read` signal to go high, one clock cycle after the `start` input pulses high for one clock cycle. In normal integration mode, this initiates the collection of integrated samples from the preceding integration period. In dump-mode it initiates the collection of raw ADC samples for the subsequent dump-frame period.

Alternatively, if the `empty` input signal is still low, when the `start` pulse arrives, this means that the *Frame Buffer* is still busy sending the previous data-frame, and is not ready to start collecting a new frame. When this happens, the low `empty` input-signal prevents the *Slave Reader* from seeing the `start` pulse. As a result, a new data-collection period is not initiated, and the data that would have been collected, are simply discarded.

So, when a `start` pulse arrives when the `empty` signal is asserted, although the `start` pulse only lasts for one clock cycle, `Latch1` and its surrounding logic thereafter hold the `read` signal high until either the `full` input is asserted by the *Frame Buffer*, or the countdown of samples remaining to be collected, reaches zero. The `read` input is then pulled low, to terminate the collection of samples, and thereafter held low until a new `start` pulse is received.

The `start` pulse, when enabled by the `empty` input, also loads a down-counter with the number of samples that are to be read. The counter thereafter counts down by one at the start of each clock cycle, until one clock cycle before the `read` input is due to go low again, which happens either when the *Frame Buffer* asserts the `full` input, or the output count of the counter reaches zero.

In normal integration mode, `MUX1` initializes the down-counter to 127, which is 32samples × 4slaves. The 2 most significant bits of this number, in the output count, are used to select which slave is to be read, such that 32 16-bit samples are read from one slave at a time, starting with the the 4th slave, and working down to the 1st slave.

In dump mode, `MUX1` initializes the counter with the value presented by the *State Generator*, at the `dsize` input. This specifies how many dump-mode samples to attempt to collect. Although this can be any 16-bit number, if it exceeds the capacity of the *Frame Buffer*'s FIFO, the actual number of samples collected and sent to the computer will be truncated to fit in the available space. Unlike in normal integration mode, where the output of the counter dynamically selects which slave is to be read from, in dump mode `MUX2` arranges that all of the dump-mode samples be loaded from the single slave that is specified by the `dslave` input.

To better illustrate the operation of this circuit, a timing diagram of it, derived by hand, is shown in figure 3.12.

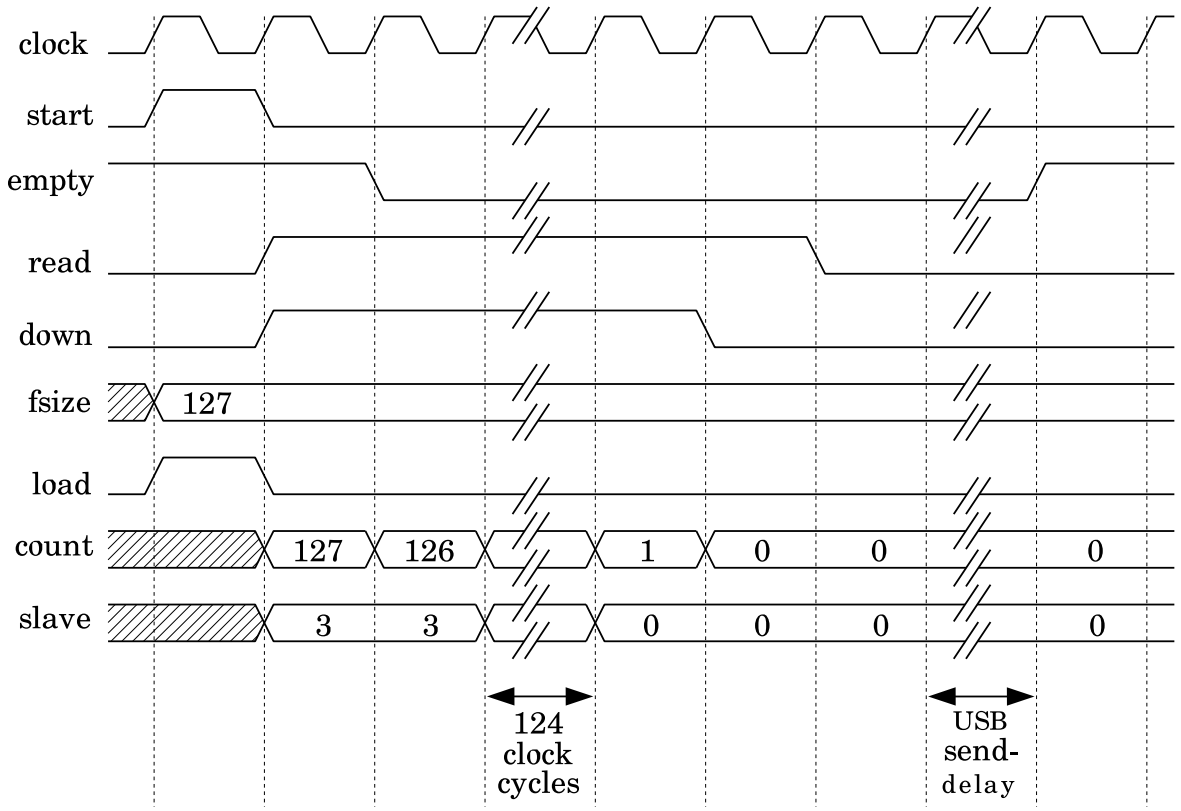

Figure 3.12: A timing diagram of the Slave Reader

The internals of the Frame Buffer

As shown in figure 3.13, the *Frame Buffer* has three major parts.

1. A *Frame Header*. This initially contains an 8×16-bit header describing the sample data.
2. A large FIFO, in which sample data are collected from the slave FPGAs at a faster rate than they can be sent to the computer.
3. A *Byte Streamer* component which transfers data, first from the header, then from the FIFO, to the USB interface chip.

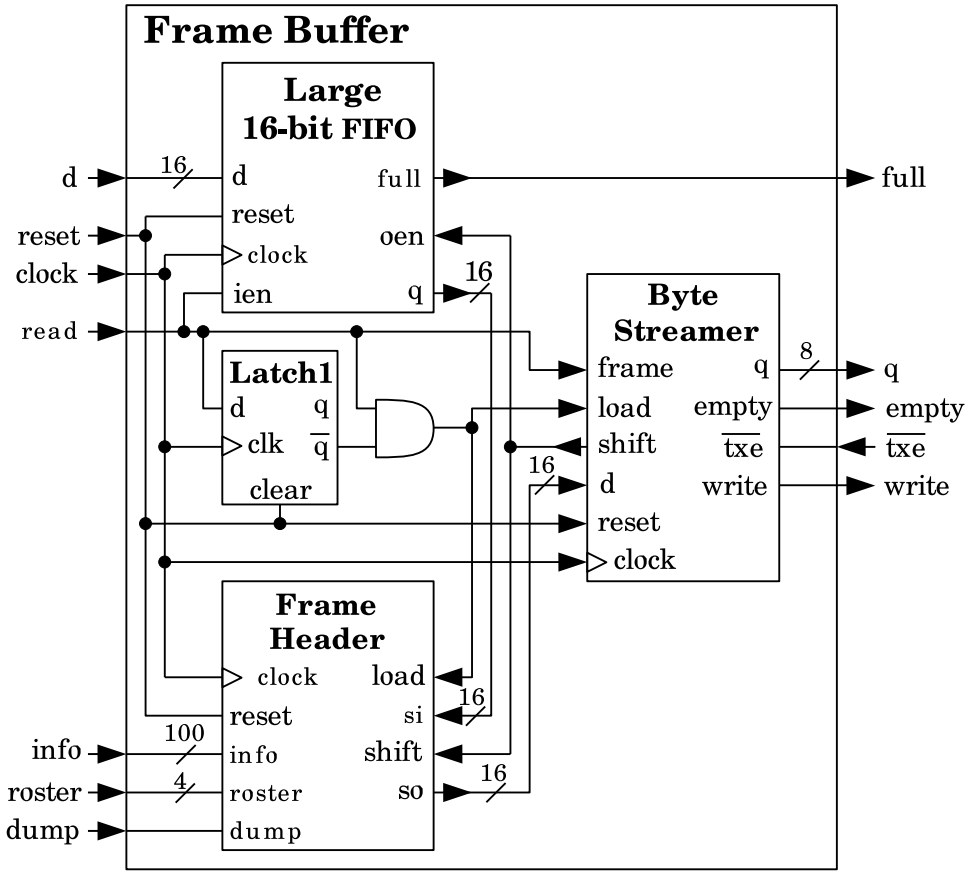

Figure 3.13: The Frame Buffer

Note that the contents of the FIFO are streamed through the *Frame Header*'s internal PISO. Thus, to ensure that as each sample gets shifted out of the *Frame Header*'s PISO, a new sample is shifted into it from the FIFO, the `shift` output of the *Byte Streamer* is connected

to both the output-enable, `oen`, input of the FIFO and the `shift` input of the *Frame Header*. This is how the two data sources are combined into one stream.

The purpose of `Latch1` is to form a pulse that lasts for one clock cycle, starting from the moment when the `read` input signal first goes high. This is used to initialize the *Frame Header* and the *Byte Streamer* at the start of each new frame.

The internals of the Frame Header

As shown in figure 3.14, the *Frame Header* is basically an 8-entry, 16-bit synchronous PISO.

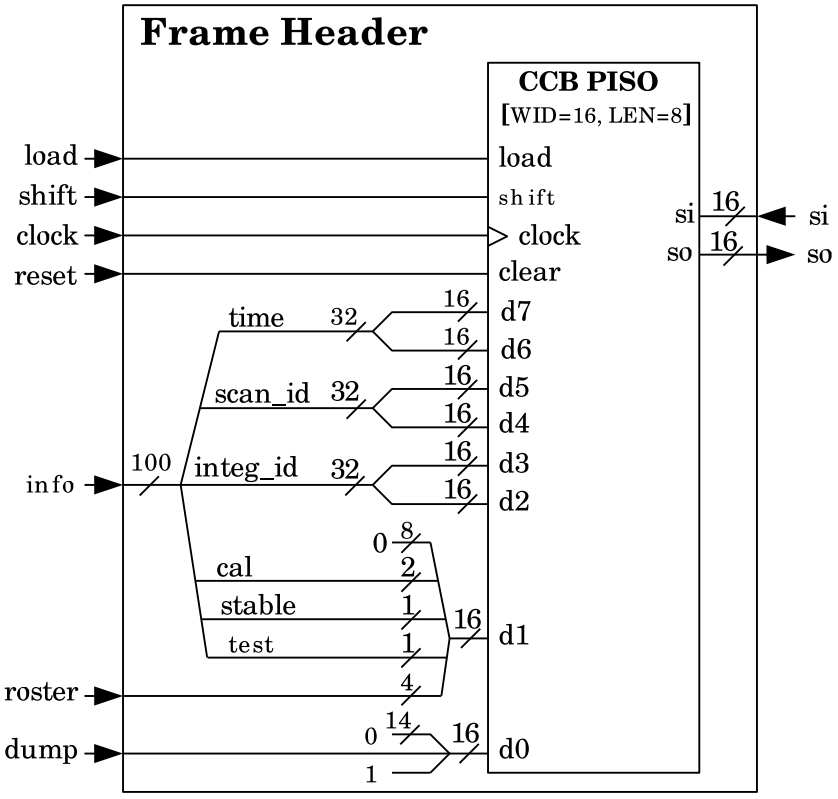

Figure 3.14: The Frame Header

Note that instead of using a conventional PISO, a copy of the customized PISO described in section 3.4.3, is used. This has separate `load`-enable and `shift`-enable inputs, which are acted upon at the rising edge of the clock. Unlike a conventional PISO, which either shifts serial data or loads parallel data on each clock cycle, the contents of the customized PISO remain unchanged during clock cycles when neither the `load` nor the `shift` signals are asserted. This is important, since the *Byte Streamer* doesn't want to be force-fed a

new sample from the *Frame Header* every clock cycle, due to the handshaking overhead and flow-control delays imposed by the USB chip.

A new frame-header is loaded into the PISO by arranging for the `load` input of the `Frame Header` component to be asserted at the next rising edge of the clock. This fills the eight 16-bit entries in the PISO with the following information.

- The first of the 16-bit header words (ie. `d0`) identifies the type of frame that is being packaged, and since it has a value that doesn't look like a data value, the CPU can use it as the indication of the start of a new frame, in case other frame separation measures don't work.

  Note that a normal data value will either be zero, in the case of a missing ADC board, or be a significantly non-zero number, in the presence of sampled noise. So a small non-zero 16-bit number, is a good choice for something that should not look like a data sample.

  Thus to ensure that the first header-word not look like a data sample, its 16-bit value is always a small non-zero number, having either the value 1 or the value 3. A value of 1 signifies that the frame is a normal integration frame, whereas a value of 3 means that it is a dump-mode frame.

- The second of the header words is a 16-bit word indicating various conditions that pertained while the data were being taken. Bits 0 through 3 form a boolean list of the slave FPGAs whose heartbeat signals indicate that they are present and functioning. Bit 4 reports whether the samples that were integrated, or dumped were fake test samples or real ADC samples. Bit 5 tells the CCB manager that the cal-diode switches were stable throughout the integration. Bits 6 and 7 report the commanded states of the cal-diode switches during the integration. The remaining 8 bits are currently unused.

- The 3rd and 4th header words are the least and most significant 16 bits of a 32-bit number, which specifies the sequential number of the integration within its parent scan, starting from zero for the first integration of a new scan, and incrementing by one each time that a new integration starts.

- The 5th and 6th of the header words are the least and most significant 16-bits of the 32-bit number which identifies the parent scan, according to the number of new scans (and intra-scans) that had been requested when the parent scan was commanded. Whenever the CCB firmware is reset, the scan-counter is reset to zero.

- Finally, the 7th and 8th header words are the least and most significant 16 bits of a 32-bit time-stamp. This is the value of a counter in the *State Generator* which is reset to zero at the start of each new scan, and incremented by 1 every clock cycle thereafter. Thus the time-stamp measures the time elapsed since the start of the second on which the last scan started, has a resolution of 100ns, and wraps around every 430 seconds.

On the real-time computer, the sum of the absolute time of the 1PPS edge on which the scan was started, and the above relative time-stamp (after accounting for wraparounds), will form the high-resolution time-stamp that is sent with the data, to the manager.

Once the PISO has been initialized, the `shift` input, when asserted during the rising edge of the clock, causes the contents of the PISO to be shifted by one towards the `so` output. This presents the next previously unseen value, at the `so` output, while shifting in a new value from the `si` input.

The internals of the Byte Streamer

Figure 3.15 shows the contents of the *Byte Streamer* component.

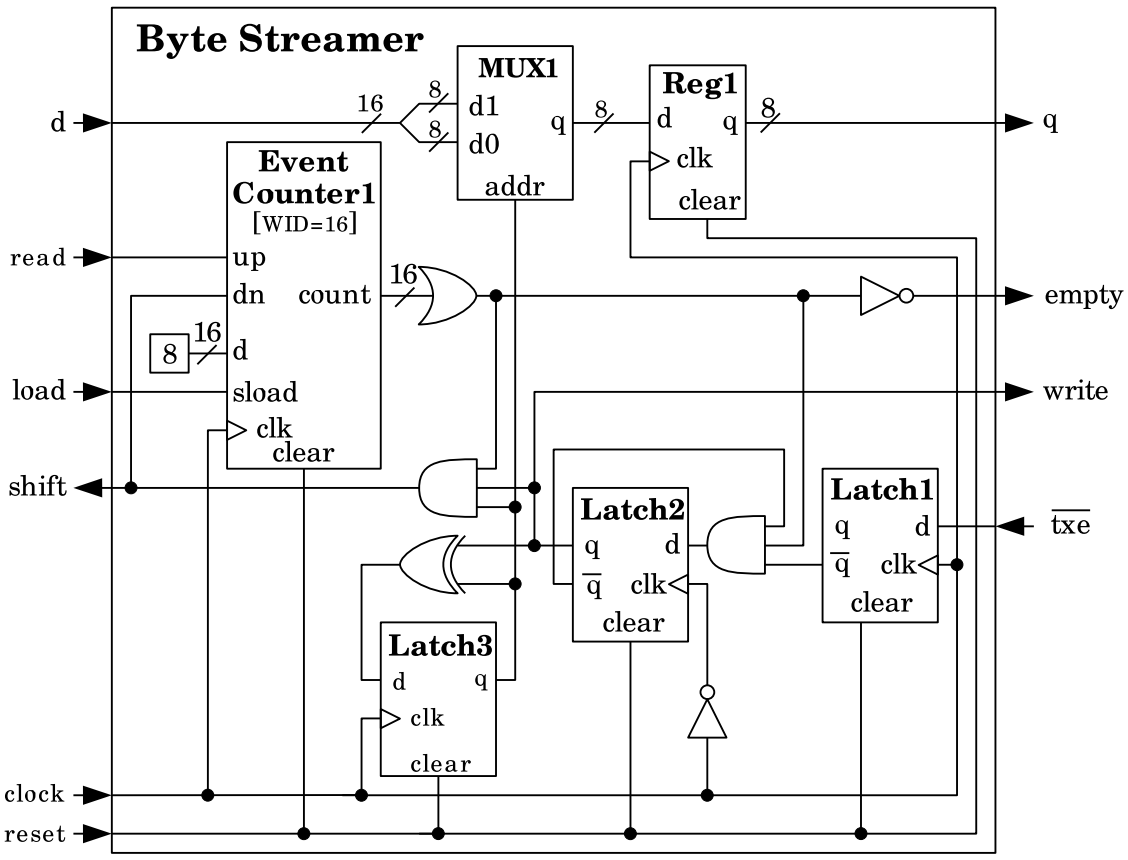

Figure 3.15: The Byte Streamer

The *Byte Streamer* takes one 16-bit sample at a time from its `d` input, and sends this to the USB chip, via the `q` output, starting with the least significant 8-bits, followed by the most

significant 8-bits. Since the *Frame Buffer* fills up much faster than the USB chip accepts bytes for transmission, there is never any need for the *Byte Streamer* to wait for more data to arrive in the FIFO, once a new frame has been started. Thus the flow of data is solely regulated by the transmit-enable, $\overline{\texttt{txe}}$, output of the USB chip. From the point of view of the FPGA firmware, the $\overline{\texttt{txe}}$ signal is asynchronous, and thus needs to be passed through a pair of latches to stabilize and synchronize it with the FPGA clock. This is satisfied by latches 1 and 2. Note that `Latch2` is negative edge-triggered, while all other latches are positive edge-triggered. There are two reasons for this.

1. This reduces the time taken for a change in the state of the $\overline{\texttt{txe}}$ signal to reach the output of `Latch3`, by a full FPGA clock cycle. This correspondingly speeds up the handshake with the USB chip.

2. It is needed to arrange for the active edge of the `write` strobe to occur more than half a clock cycle after new data have been presented on the `q` output that goes to the USB chip, and half a clock cycle before this data is replaced with the next byte. This safely exceeds the 20ns setup and 10ns hold-time requirements of the USB chip.

`Latch3` and the exclusive-OR gate which drives its input, effectively form a latched 1-bit counter. This counts the number of $\overline{\texttt{txe}}$ pulses, modulo 2, and its output is used by multiplexer `MUX1`, to select which of the two halves of the 16-bit number at the `d` input, is latched into register, `Reg1`, and subsequently read by the USB interface chip. After both bytes of the `d` input have been transfered in this way, the `shift` output is asserted for one clock cycle, to tell the *Frame Buffer* FIFO and the *Frame Header* components to present a new 16-bit number at the `d` input.

The 16-bit counter, `Event Counter1` (see section 3.4.4), keeps a record of how many 16-bit samples remain to be transfered from the *Frame Buffer*'s FIFO and *Frame Header* components. When the `load` input is pulsed, for one clock cycle, at the start of a new frame, the counter is pre-loaded with the number of samples in the frame header. Thereafter, it counts up by one whenever the `read` strobe is found to be high at the rising edge of the clock, indicating that a new sample is being read into the FIFO, from the slave FPGAs. Similarly, it counts down by one when the `shift` strobe is pulsed, indicating that a new 16-bit sample has been transfered to the USB interface chip. When both the `shift` and `read` strobes are asserted, the count remains unchanged, since the number of samples remaining in the FIFO and header, remains unchanged. When the number of samples remaining to be transfered from the FIFO and header, reaches zero, the output of the 16-input OR gate at the output of the counter, goes low, and this asserts the `empty` output signal, indicating to the *Slave Reader*, that all data have been transfered to the USB chip.

A timing diagram illustrating the operation of the *Byte Streamer*, is given in figure 3.16. In this diagram, the example frame has only one header entry, instead of 8, and only one data sample. Note that the pulses of the `shift` signal that are marked with an `x`, are unintentional side-effects that don't affect the intended operation of the circuit.

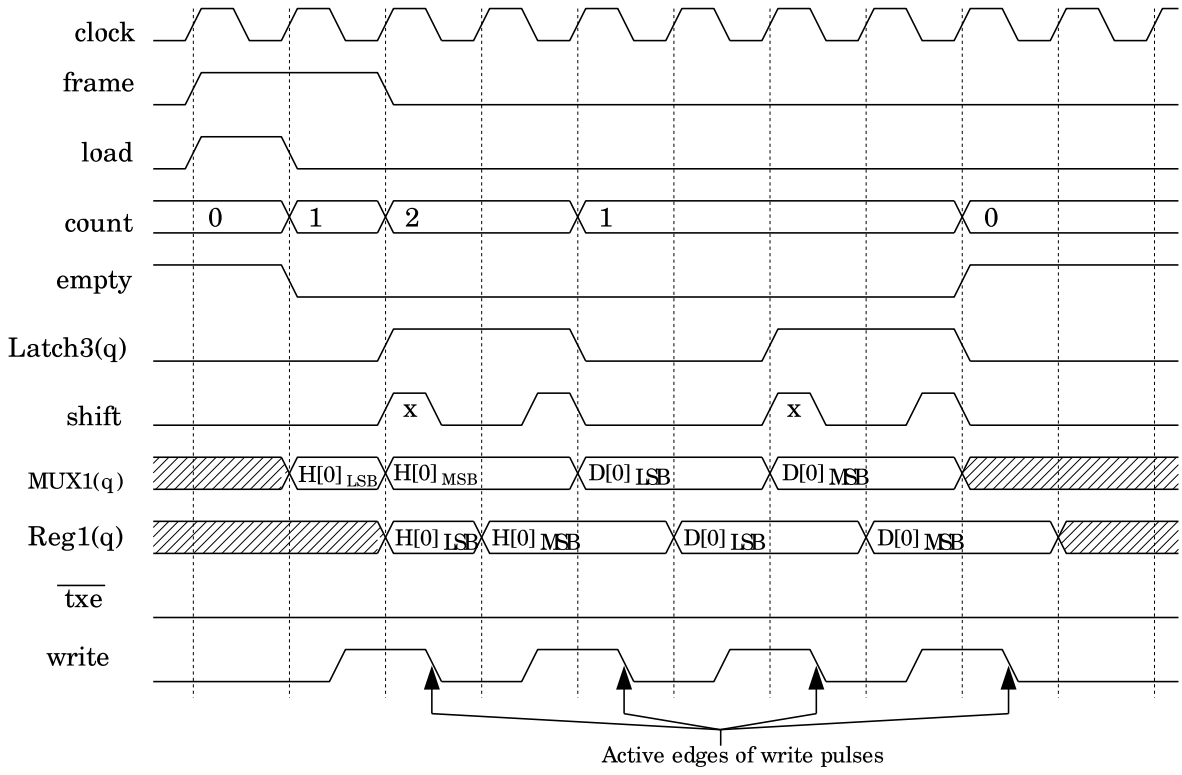

Figure 3.16: A timing diagram of the Byte Streamer

The internals of the Slave Detector

The *Slave Detector* module attempts to determine whether each of the slave FPGAs are present and functional, by monitoring their heartbeat signals. Each slave emits a single-bit heartbeat signal which changes state at the start of each new clock cycle. The job of the *Slave Detector* is thus to verify that each of the heartbeat signals switches state from one clock cycle to the next. The implementation is shown in figure 3.17.

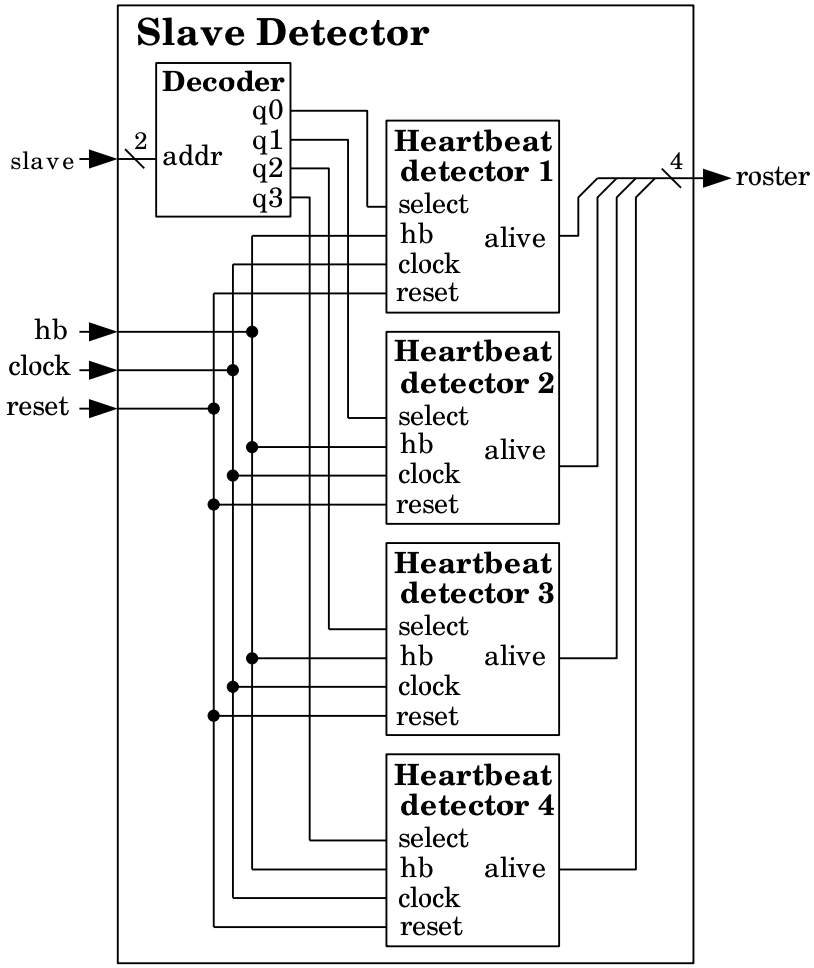

Figure 3.17: The Slave Detector

As can be seen, each slave has its own *Heartbeat Detector* module, which is enabled when the slave is selected for readout by the *Slave Reader*. At the end of each integration the 4 `alive` outputs of the *Heartbeat Detector*s, combined into the 4-bit `roster` output, provide an indication of which slaves were alive during the integration period.

The internals of the Heartbeat Detector modules

The implementation of the individual *Heartbeat Detector* modules is shown in figure 3.18.

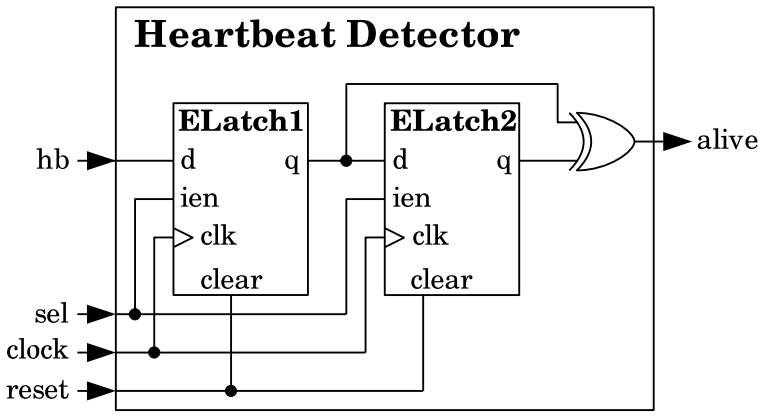

Figure 3.18: The Heartbeat Detector

When the `sel` input of a *Heartbeat Detector* module is asserted, a rising edge at the clock input causes `ELatch1` (see section 3.4.1) to acquire the state of the heartbeat signal, at the `hb` input. At the same time, the previous state of the heartbeat signal is transfered from `ELatch1` to `ELatch2`. Since a functional heartbeat signal changes state at the same point in each new clock cycle, the outputs of latches 1 and 2 should be complements of each other. If so, the exclusive OR of these outputs will be true. Thus the `alive` output indicates whether the heartbeat signal was present, and behaving correctly, during the last two clock cycles.

When the `sel` input of a *Heartbeat Detector* module is not asserted, this means that the heartbeat signal of a different slave is being sampled by a different *Heartbeat Detector* module. Thus, the de-asserted input-enable (`ien`) inputs of `ELatch`es 1 and 2 prevent their parent latches from responding to the other module's heartbeat signal.

Individual slaves are always selected for readout for many consecutive clock cycles, so although it takes a couple of clock cycles after a slave has been newly selected, for the `alive` output to reliably indicate the presence or absence of a slave; by the time that the readout of that slave has finished, the `alive` output will have settled into the appropriate state. This state is then preserved, while the `sel` input is not asserted, until just after all of the `alive` outputs are sampled for inclusion in the header of the latest data frame.

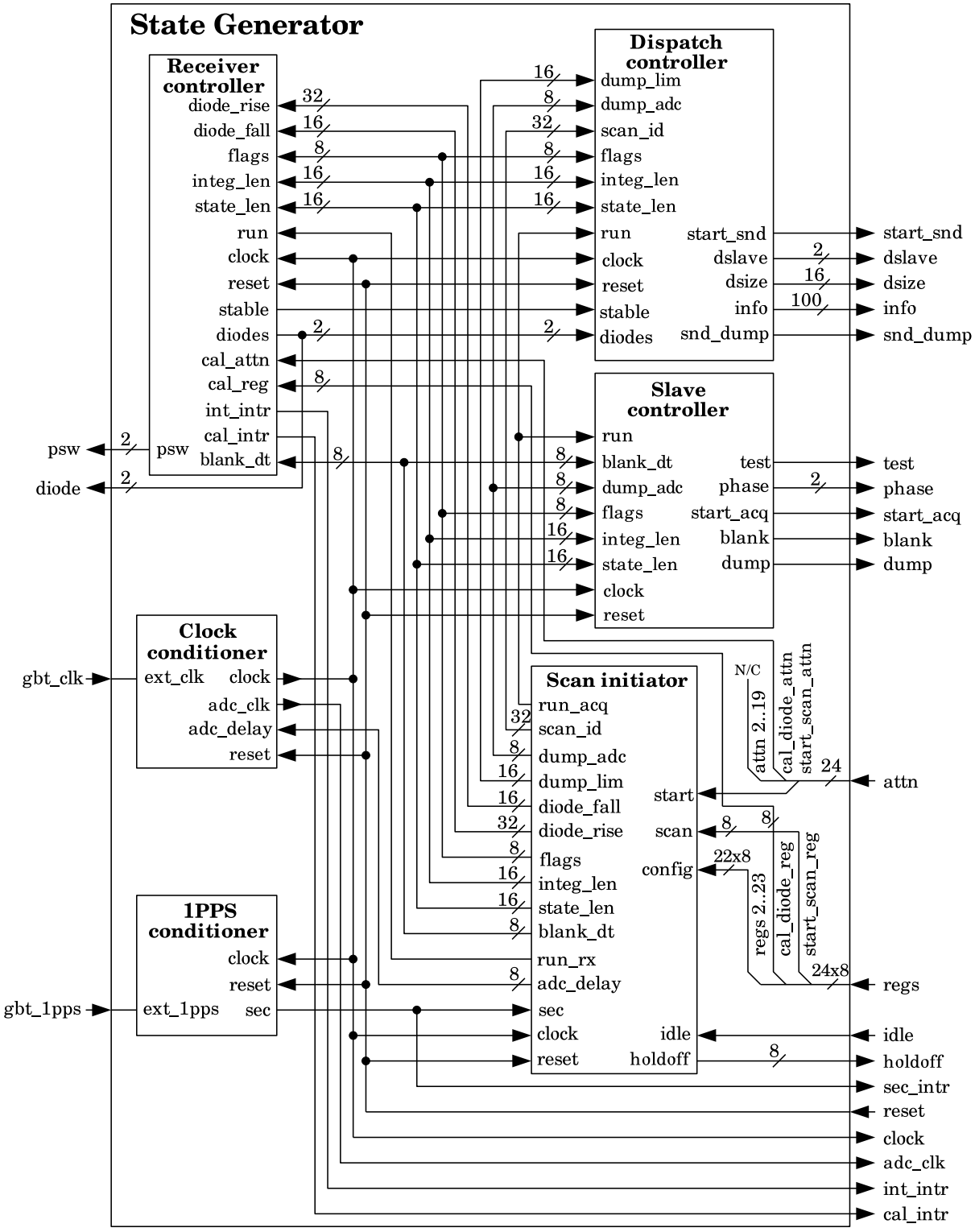

Figure 3.19: The State Generator

3.3 The State Generator

The *State Generator* is the central hub that responds to commands and configuration information from the CCB real-time computer, sequences the activities of the other top-level modules in the master and slave FPGAs, commands CPU interrupts, controls the cal-diode and phase switches in the receiver, and receives and conditions the external 1PPS and FPGA clock signals. The internals of the *State Generator* are depicted in figure 3.19.

Along the lower half of the left edge of figure 3.19, the external 1PPS and FPGA clock signals enter the FPGAs, to be conditioned for use elsewhere within the FPGAs. Along the upper half of the left edge, the cal-diode and phase-switch control signals exit on their way to the receiver.

At the top of the right edge of the diagram are the signals that go to the slave FPGAs and the *Data Dispatcher*. In the middle of the right edge are the read-only register values and notification signals that the *Control Gateway* distills from its communications with the CCB computer. At the bottom of the right edge, are the interrupt signals that go to the CCB computer via the EPP interface in the *Control Gateway*. The remaining signals are the conditioned clock signal that goes to the other modules in the FPGAs, the EPP reset signal from the *Control Gateway*, the `idle` signal which tells the *State Generator* when the *Data Dispatcher* has finished sending the data of the last integration period to the computer, and the holdoff interval that tells the *Control Gateway* the minimum interval to wait between generating interrupts.

Everything in the *State Generator* occurs according to configuration information and commands received from the CCB computer, via the register interface that the *Control Gateway* presents to it. The 8-bit registers are presented as a group to the *State Generator*, via the `regs` input signal, which is thus an integer multiple of 8-bits in width. The same number of single-bit `attn` (ie. attention) input signals tell the *State Generator* whenever the individual registers are updated, by being asserted for one clock cycle after the corresponding register has been updated. The official list of CCB registers, together with their contents and effects can be found in appendix A.

Within the *State Generator* the *Scan Initiator* module starts and stops scans, according to commands received from the computer, via the *Control Gateway*. It also presents a frozen snapshot of the scan-configuration registers to the other modules.

The *Receiver Controller* controls the receiver phase-switch and calibration-diode control lines. The *Slave Controller* generates the signals that control the acquisition and integration of data by the slave FPGAs. The *Dispatch Controller* controls the collection of data from the slave FPGAs, at the boundaries between integration periods, and the communication of these data to the computer.

3.3.1 The Scan Initiator

An overview of the Scan Initiator functionality

Within the *State Generator*, the *Scan Initiator* is the module which responds to the real-time computer writing to the `start_scan_reg` register. When this happens, the *Scan Initiator* performs the following actions.

- Simultaneously it first performs the following actions.
  - The `run_rx` signal is deasserted, to tell the *Receiver Controller* to stop toggling the receiver's cal-diode and phase-switch control lines, and to tell it to clear its queue of integration configurations. The latter queue must be cleared within one clock cycle of the `run_rx` signal becoming deasserted, and thereafter allowed to receive new entries from the computer. Thus, in the time between one scan being halted, and a new one being started, the computer can start to send the configurations of the initial integrations of the new scan.
  - Similarly, the `run_acq` signal is deasserted to tell the *Dispatch Controller* to stop the *Data Dispatcher* from collecting any further integration periods of data. If an integration period just ended, and its data are in the process of being collected and dispatched to the computer, this is allowed to continue. But until the `run_acq` signal is asserted again, the data of subsequent integration periods must be discarded.
  - A snapshot of the values of the configuration registers at the `config` input is copied into an internal bank of registers. At the output of this bank the 8-bit registers are collected into multi-byte logical registers and presented as output signals to configure the other modules within the *State Generator*. Taking a snapshot of the configuration registers, in this way, provides a guarantee that configuration updates from the computer only take affect at the start of the following scan.
  - If the `sync` flag in the `start_scan_reg` register is set, the *Scan Initiator* arms a latch to assert a time-synchronization flag at the next rising edge of the 1PPS signal. Otherwise it asserts this flag without waiting, since synchronization with the 1-second tick has not been requested.
- A few clock cycles after doing the above operations, the *Scan Initiator* simultaneously waits both for the aforementioned time-synchronization flag to be asserted, and for the `Data Dispatcher` to assert the `idle` signal, indicating that it is ready to start collecting and dispatching data from the new scan. The delay of a few clock cycles before checking for these conditions reflects the fact that it takes a few clock cycles before the *Data Dispatcher* reports that it is not idle, after it has been told to start collecting a new frame of data.
- Once both the `idle` signal and the time-synchronization flag are found to be asserted, the *Scan Initiator* asserts the `run_rx` output signal. This causes the *Receiver Controller*

to adopt the configuration parameters presented by the `scan_regs` output of the *Scan Initiator*, and then start the *Receiver Controller*'s control loop.

At the same time, the *Scan Initiator* starts a countdown of the number of clock cycles that it takes for the initial effects of toggling any of the receiver's control signals, to become apparent in the data that reach the slave FPGAs. This is referred to as the round-trip delay countdown.

- When the round-trip delay countdown reaches zero, the *Scan Initiator* asserts the `run_acq` signal. This causes both the *Slave Controller* and the *Dispatch Controller* to adopt the scan configuration parameter outputs of the *Scan Initiator*, and then start controlling their respective charges.

At the cores of the three controller modules, are identical copies of the `Scan Sequencer` module, which generates the timing signals that control transitions between the various states within a scan. Because of the delay between the `run_rx` and `run_acq` signals being asserted, the `Scan Sequencer` in the *Receiver Controller* generates timing ticks that precede their equivalents in the `Scan Sequencer`s of the *Slave Controller* and *Dispatch Controller* modules. This deliberate timing offset compensates for the round-trip delay between receiver-control signals being generated by the `Receiver Controller`, and the resulting effects reaching the data-acquisition modules within the slave FPGAs.

For example, after the *Receiver Controller* toggles the states of the phase switches, the *Slave Controller* doesn't immediately stop the slave FPGAs from integrating samples into the previous phase-switch accumulation bin. Instead, the change in phase-switch bins is delayed by the round-trip control delay, so that residual samples from the previous phase-switch state don't get incorrectly added into the accumulation bins of the new phase-switch state.

The implementation of the Scan Initiator

The internals of the *Scan Initiator* are depicted in figure 3.20.

Whenever the `start_scan_reg` register is written to by the computer, the *Control Gateway* asserts the corresponding register-attention signal for one clock cycle. This signal is routed to the `start` input of the *Scan Initiator*, and initiates the start of a new scan or intra-scan. At the first rising edge of the clock that follows the `start` signal being asserted, all of the components on the left side of figure 3.20 respond to it, as follows.

- Register `EReg1` latches a snapshot of the configuration register values from its `config` input, and makes them available to the other components of the *State Generator* as single byte, or aggregated multi-byte configuration signals.
- The remaining components control the `run_acq` and `run_rx` output signals, whose requirements have already been discussed in the overview of the *Scan Initiator*. Initially,

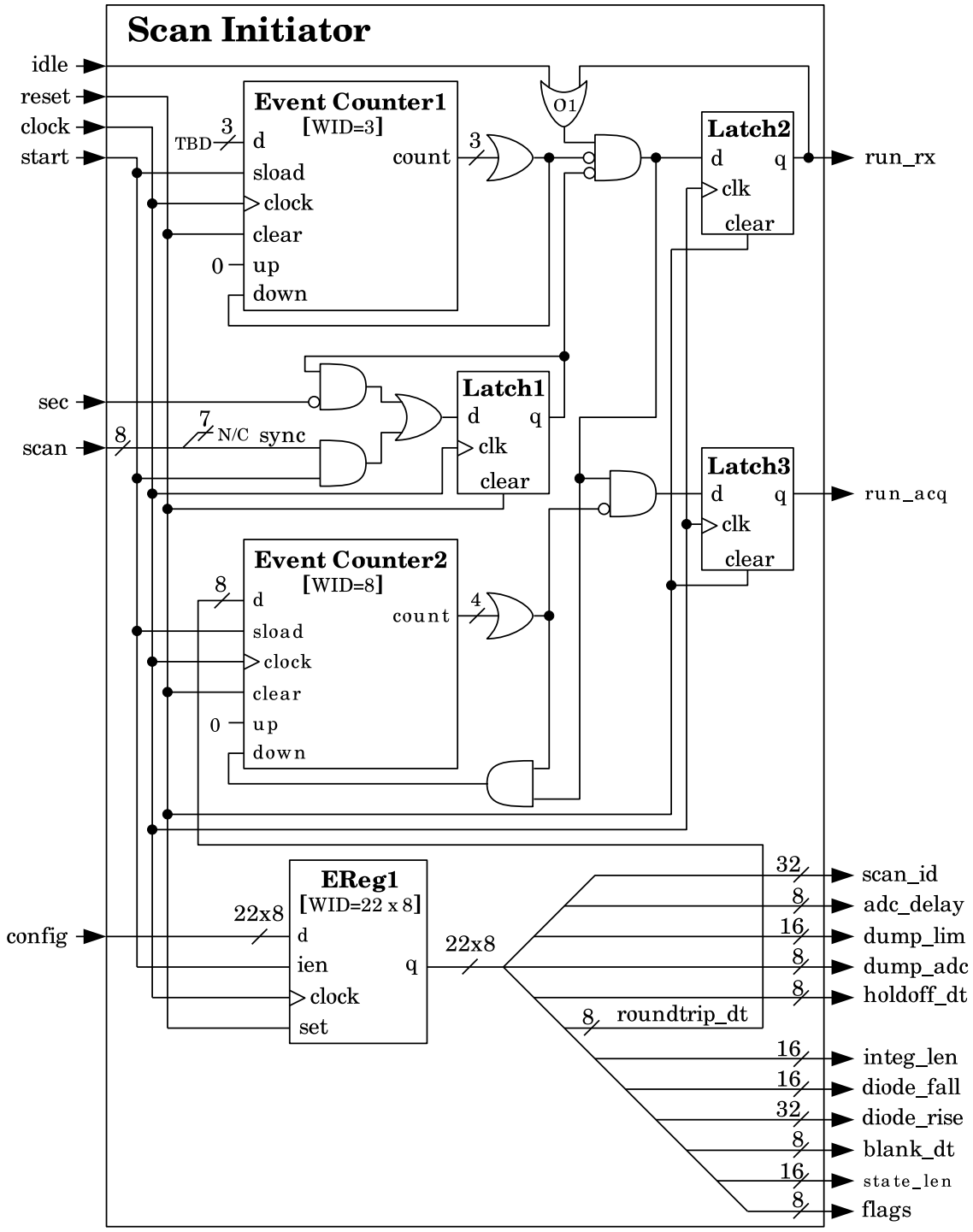

Figure 3.20: The Scan Initiator

the start signal loads countdown delays into EventCounter1 and EventCounter2, and starts Latch1 waiting for the next 1-second tick (if the sync flag of the start_scan_reg register is asserted). Loading a finite value into EventCounter1, causes both the run_rx and run_acq output signals to be deasserted on the next rising edge of the clock. EventCounter1 thereafter counts down by 1, every clock cycle, and thus maintains the run_rx and run_acq signals deasserted, at least until it's count reaches zero. Once it does reach zero, the states of the idle signal and the output of Latch1 determine when the run_rx signal becomes asserted again. In particular, it becomes asserted when the idle input indicates that the *Data Dispatcher* has finished sending any data that was in the process of being sent, and when Latch1 indicates that either no 1-second synchronization was needed, or that the requested 1-second tick has been seen. Once both of these conditions have been met, the asserted run_rx output, fed back to the rightmost input of OR gate O1, stops subsequent changes to the state of the idle input from affecting the run_rx output (or the run_acq output), until the next start pulse forces the output of Latch2 low again. Thus the run_rx output remains asserted until the next write to the start_scan_reg register.

Once the run_rx signal has been asserted, EventCounter2 starts counting down the round-trip control delay. When this counter reaches zero, the run_acq output becomes asserted. Thus the run_acq output signal becomes asserted one round-trip delay after the run_rx signal, and stays asserted until the next write to the start_scan_reg register.

3.3.2 The Receiver Controller

The *Receiver Controller* controls the phase-switch and calibration-diode control lines that go to the receiver. It also handles the requesting and receipt of multi-integration calibration-diode configurations from the computer, and the assignment of these configurations to successive groups of integrations. In addition, it generates the cal-diode stability flag that is placed in the data-header that the *Data Dispatcher* subsequently sends to the computer, plus the interrupt-signal that signals the end of each integration period. The implementation of the *Receiver Controller* is shown in figure 3.21.

The behaviors of most of the input signals to this module have already been described, in the documentation of the *Scan Initiator*. Not mentioned so far, are the cal_reg and cal_attn signals. The cal_reg signal brings in the contents of the cal_diode_reg register (see appendix A), and the cal_attn signal informs the *Receiver Controller* whenever the computer writes to this register. The run signal, whose properties have already been described in general, is inverted to form a preparation and hold signal, which is asserted during the time between a new scan request being received by the *Scan Initiator* from the computer, and the actual start of the scan. On the first rising clock edge that follows this period, the frozen *Scan Sequencer*, *Phase Sequencer* and *Cal Controller* components all simultaneously proceed with the new scan.

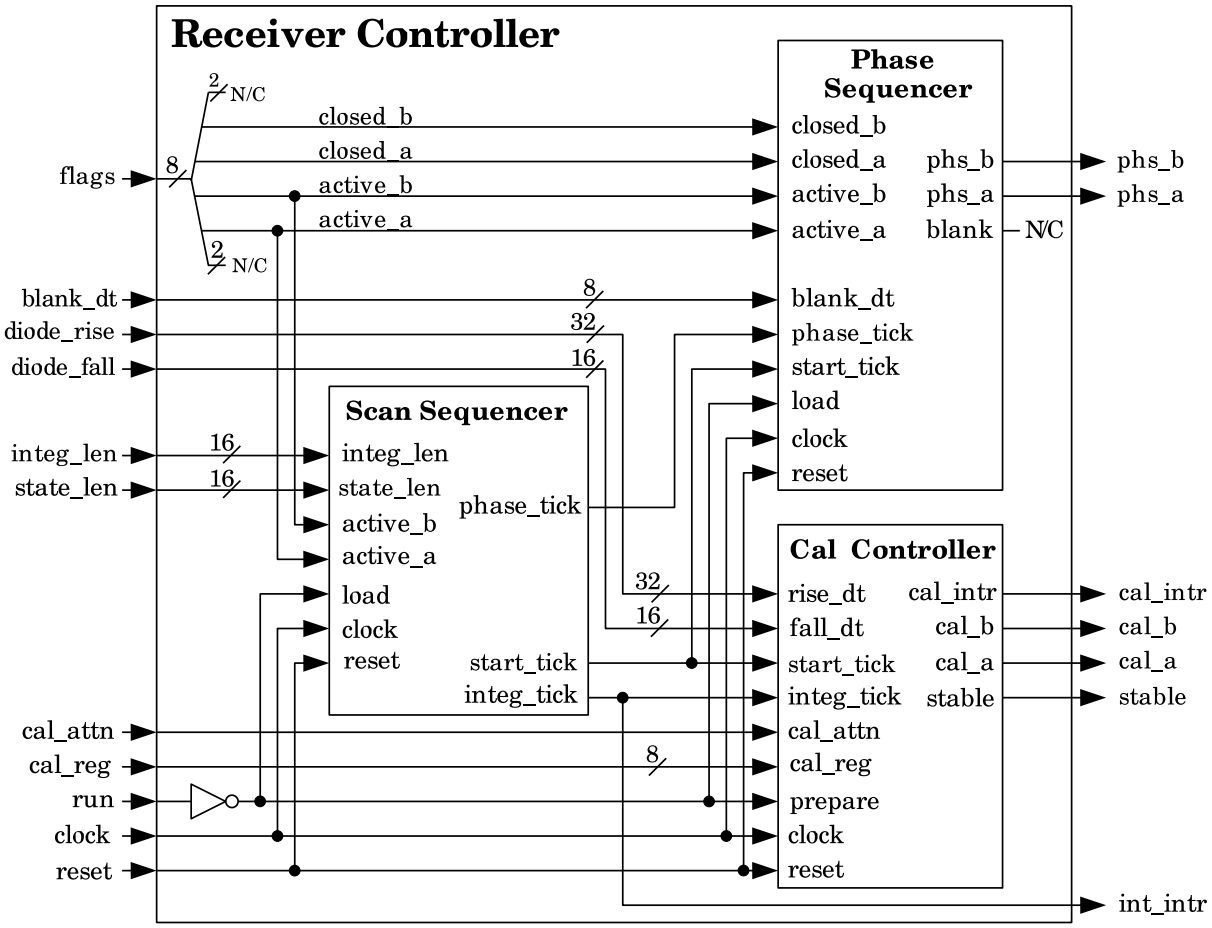

Figure 3.21: The Receiver Controller

Within the *Receiver Controller*, the *Scan Sequencer* generates ticks that announce the end of each integration period, phase-switch cycle, and phase-switch state. The *Cal Controller* uses the integration tick that this generates, to sequence the calibration diode control signals, according to the incoming stream of cal-diode configurations, received via the `cal_reg` register. Similarly, the *Phase Sequencer* uses the phase-switching tick to cycle the phase-switching control signals through the sequence of states specified in the `scan_flags_reg` configuration register.

The *Cal Controller* also generates a stability flag, `stable`, that is used by the *Dispatch Controller* to signal to the computer when the data that it is sending, is from an integration period during which the calibration diodes were in stable states.

Similarly, the *Phase Sequencer* generates a blanking signal, after each switch transition. However this signal is not used by the *Receiver Controller*, since it is designed for the copy of this module in the *Slave Controller*.

The majority of the implementation of the *Receiver Controller* lies within the *Scan Sequencer*, *Cal Controller* and *Phase Sequencer* components. These are described next.

The Scan Sequencer

The *Scan Sequencer* generates single-cycle ticks at the ends of each of the various types of cycles within a scan, including a tick that marks the end of each phase-switch state, a tick which marks the end of each phase-switch cycle, and a tick that marks the end of each integration period. In addition, there is a similar tick which marks the start of all cycles at the start of a new scan, and a time-stamp output which reports the length of time that has passed since the start of the scan. This is all implemented as shown in figure 3.22.

Since the integration period is configured as a multiple of the phase-switch-cycle period, and the latter period is always a multiple of the phase-switch-state period, the corresponding ticks are generated by a cascaded chain of *Metronome* components (see section 3.4.5). `Metronome1` generates one `state_tick` every `state_len` clock cycles. `Metronome2` generates one `phase_tick` every `state_len` $\times$ *nactive* clock cycles, where *nactive* is the number of active phase switches, as computed by `MUX1`. `Metronome3` generates one `integ_tick` every `state_len` $\times$ *nactive* $\times$ `integ_len` clock cycles.

Since *Metronome* components introduce a pipeline delay of one clock cycle between their `step` inputs and their `tick` outputs, latches 4,5 and 6 are needed to equalize the delays that accumulate along the chain.

On the first rising clock edge that follows the `load` input going low, the counter in `metronome1` counts down by one, indicating that one clock cycle has passed since the start of the scan. Thus, relative to the input signals of `metronome1`, the start of the scan lies one clock cycle before the first rising clock-edge at which the `load` input is seen to be low. However, the

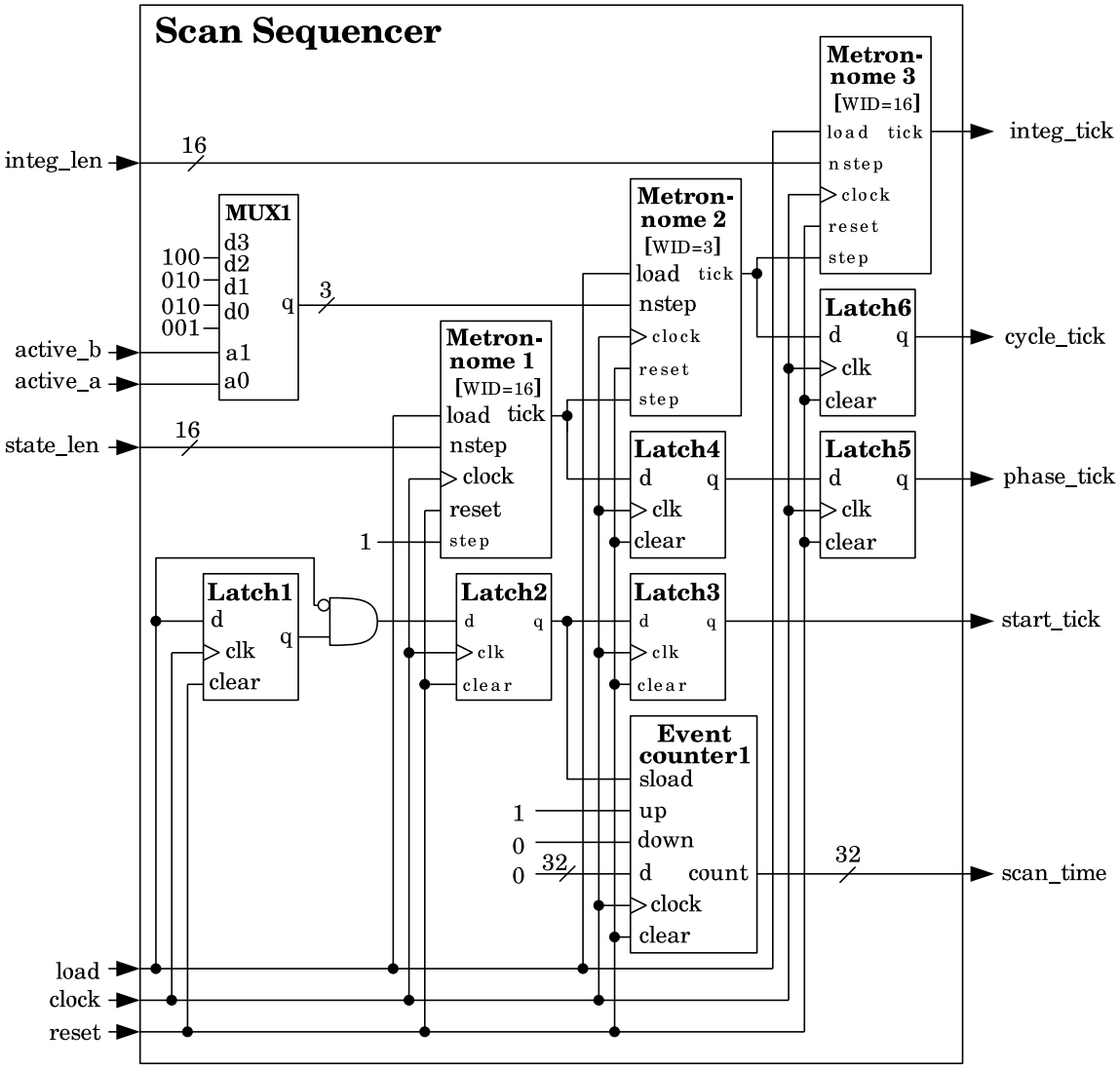

Figure 3.22: The Scan Sequencer

pipeline delays of 1 clock cycle introduced by `metronome1` and 2 clock cycles introduced by latches 4 and 5, move the start of the scan, relative to the output ticks at the `phase_tick` output, to 2 clock cycles after the first rising clock-edge at which the `load` input is seen to be low. Thus the `start_tick` output, which marks the actual start of the scan, must been seen to be high on the second rising clock-edge that follows the first rising clock-edge at which the `load` input is seen to be low. This is the function of latches 1, 2 and 3.

In particular, `latch1` generates a pulse that lasts from when the `load` input first goes low, until just after the next rising edge of the clock. `Latch2` converts this into a pulse that lasts precisely one clock cycle, lasting from just after the first rising clock-edge that follows the `load` input going low, to just after the following rising clock-edge. `Latch3` delays this pulse for a further clock cycle, such that 2 clock cycles after the rising-edge at which the `load` input is first seen to have gone low, the `start_tick` output is high.

The intervals between the ticks generated by the metronomes are loaded just before the start of a new scan, by asserting the `load` input for at least one clock cycle. While this input remains asserted, the metronomes load new values at each rising clock edge. They don't start counting down until the first rising clock edge after the `load` input is deasserted. Thus the `load` signal is intended to be used not only to load a new scan configuration, but also to prevent the metronomes from running until the `load` input is deasserted at the actual start of the scan. As explained above, the actual start of the scan occurs 2 clock cycles after the first rising clock edge that follows the `load` input going low. Assuming that the `load` input is generated synchronously by a latch, this actually means that there is a delay of 3 clock cycles between said latch starting to pull its output low, and the start of the new scan.

`Event Counter1` counts the number of clock cycles that have elapsed since the start of the scan, for use by the `Data Dispatcher` for time-stamping integration periods. This counter is synchronously cleared for the start of the scan, by latching zero from its `d` input at the same time as `latch3` is latching the `start_tick` output. Thus it is zero when the `start_tick` output is asserted.

The Cal Controller

The *Cal Controller* component is responsible for maintaining a queue of per-integration calibration-diode states, queued in the order that the computer writes them to the 8-bit `cal_diode_reg` register. At the first integration of a new scan, the oldest value in this queue is popped from the queue, and split into its functional parts, which are then held in latches within other components. The 2 latched lsbs are immediately used to command the states of the calibration diodes, for as many integration periods as are specified in the remaining 6 bits of the latched value. After that many integrations have passed, the next oldest value is popped from the queue, and the cycle continues. Whenever space becomes available in the queue, the *Cal Controller* prompts the computer to send another value, by sending it a `cal_intr` interrupt.

While preparing to start a new scan, the queue is cleared, and a `cal_intr` interrupt is sent to the computer, to tell it to send the first queue entry of the new scan. In the case of a normal observing scan, start-scan commands are sent by the computer from the 1PPS interrupt handler, one second before the 1 second tick at which the scan should start. Thus the computer has up to a second to receive and respond to at least one `cal_intr` interrupt before the *Cal Controller* tries to pop the oldest entry from the queue. This should be plenty of time. On the other hand, intra-scan commands aren't synchronized with the 1 second tick, and are required to start as soon as possible after receiving the start-scan command. In this case it is likely that the queue will still be empty by the time that the first integration starts. As a result, the calibration diodes will usually remain in their pre-scan states for the first integration of an intra-scan. The informational header that is subsequently sent to the computer, along with the integrated or dump-mode data, will indicate this via its calibration-diode status bits, so this shouldn't be a problem. Similarly, if the computer unexpectedly fails to keep the queue from emptying at any time, the cal diodes will be held in their existing states until a new configuration value is written to the queue, and the data headers will indicate this to the CCB manager.

At the start of each integration, the *Cal Controller* is also responsible for generating a flag that indicates whether the calibration diodes had had time to settle by the start of that integration.

The implementation of the *Cal Controller* is shown in figure 3.23.

The queue of values written to the `cal_diode_reg` register, are held in `FIFO1`. This receives one new value at a time, whenever the *Control Gateway* asserts the `cal_attn` input, for one clock cycle. AND gate `A3` is a precaution to prevent any attempt to push a new value into the FIFO, when it is full. In practice this shouldn't happen, since the computer is only supposed to write a new value into the `cal_diode_reg` register when it receives a `cal_intr` interrupt; and latches 1 and 2 ensure that this interrupt is only generated when the FIFO has room for at least one new value.

Latch 1 is asserted to request that a single `cal_intr` interrupt be sent to the computer. It stays high until, in response, the computer writes a new value into the `cal_diode_reg` register. It then goes low for the following clock cycle. If the `full` output of `FIFO1` indicates that there is still room for another value, the output of latch 1 again becomes asserted, and remains asserted until the corresponding value is received. Whereas the output of latch 1 remains high between requesting a `cal_intr` interrupt and receiving a new value from the computer, latch 2 generates a pulse lasting precisely one clock cycle, each time that the output of latch 1 goes high for at least one clock cycle. This accommodates the interface requirements of the *Control Gateway* for requesting a single interrupt.

Whenever a new value is popped from the output of `FIFO1`, at the start of a new integration, different bits of this value are latched by `Event Counter1`, and `Cal Switcher`s 1 and 2. In particular, the event counter reinitializes its count with the top 6 bits, which contain an integer specifying for how many integrations the new calibration-diode states should

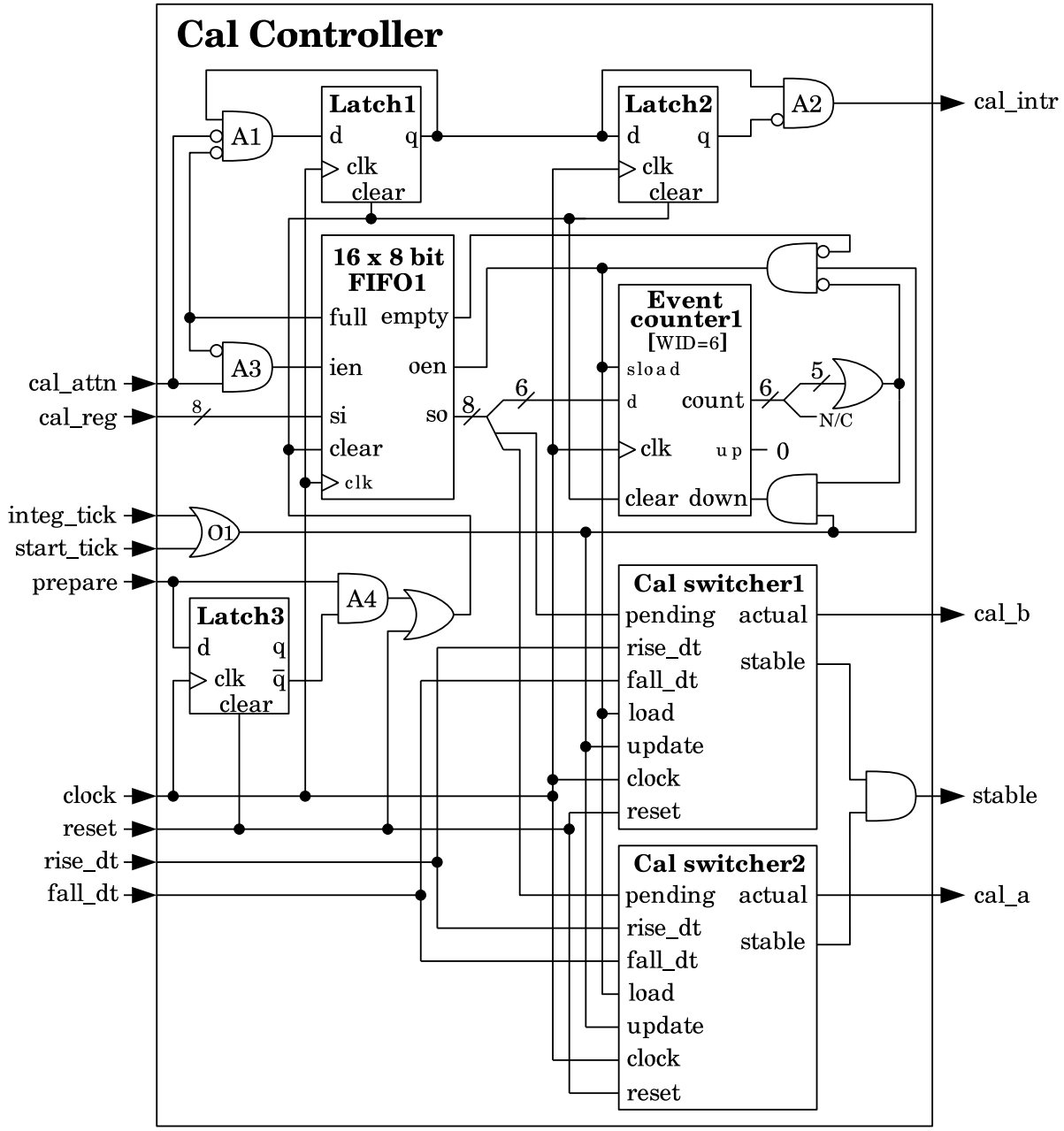

Figure 3.23: The Cal Controller

be commanded, while the two *Cal Switcher* components latch the corresponding single-bit target states of the calibration diodes for these integrations. Subsequently, at the start of each integration, `Event Counter1` counts down by 1, unless the count has fallen to 1, or been reset to zero. In the latter two cases, a new value is popped from the output of `FIFO1`, and the cycle continues. Note that the countdown normally goes down to 1, rather than zero, because when the initial count is loaded into the counter, it is not decremented to account for the integration that starts at that point.

If `FIFO1` is found to be empty when a new value is needed, neither the counter nor the *Cal Switcher* components latch a new value from the output of the FIFO, and the FIFO is not clocked. Thus the counter remains at 1 or zero, and the *Cal Switcher* components continue to drive the previous cal-diode states, until the next integration at which `FIFO1` is found to be no longer empty.

Just before the start of a new scan, the `prepare` input of the *Cal Controller* is asserted by its parent module for at least one clock cycle, until the new scan actually starts. `Latch3` and AND gate `A4` generate a pulse of one clock-cycle when this signal is first asserted. This clears the FIFO, the interrupt-request latches, and `Event Counter1`, for the new scan, then on the next clock cycle, allows them to start accepting new cal-diode information from the computer, in preparation for the new scan. The *Cal Switcher* components aren't reset by this signal, since they need to retain knowledge of the current calibration-diode states in order to compute settling times, and in order not to switch the diodes to arbitrary states during the setup preceding the new scan.

Note that *Cal Switcher* components 1 and 2 see the target cal-diode states of the next entry in the FIFO at least one clock cycle in advance of them being latched by the signal that pops them from the FIFO. This allows the *Cal Switcher* components to determine in advance, whether there will be a switch transition which will make the diodes unstable when the next value is popped from the FIFO. This is necessary to allow the stability flags to be latched to the `stable` outputs of the *Cal Switcher* components on the same clock edge that the new cal-diode states are popped from the FIFO, and latched to the corresponding `actual` outputs of these components. As a result, both the `stable` and `actual` outputs are latched by the rising clock-edge that occurs during the one-clock-cycle pulse that comes from OR gate `O1`, at the start of each integration period.

The implementation of the *Cal Switcher* components, is shown in figure 3.24.

Each *Cal Switcher* component is responsible for driving one calibration diode. At the beginning of every integration, the `update` input of the *Cal Switcher* is asserted for one clock cycle. This tells the *Cal Switcher* to update the `stable` output to indicate whether the latest settling-time countdown has reached zero or not. Similarly, the `load` output is also asserted for one clock cycle at the start of some integrations, but unlike the `update` input, this only happens at the start of integrations where a new cal-diode state has been popped from the FIFO of the parent *Cal Controller* module. This signal thus tells the *Cal Switcher* module to load the new cal-diode state, and update the settling-time countdown to reflect any resulting

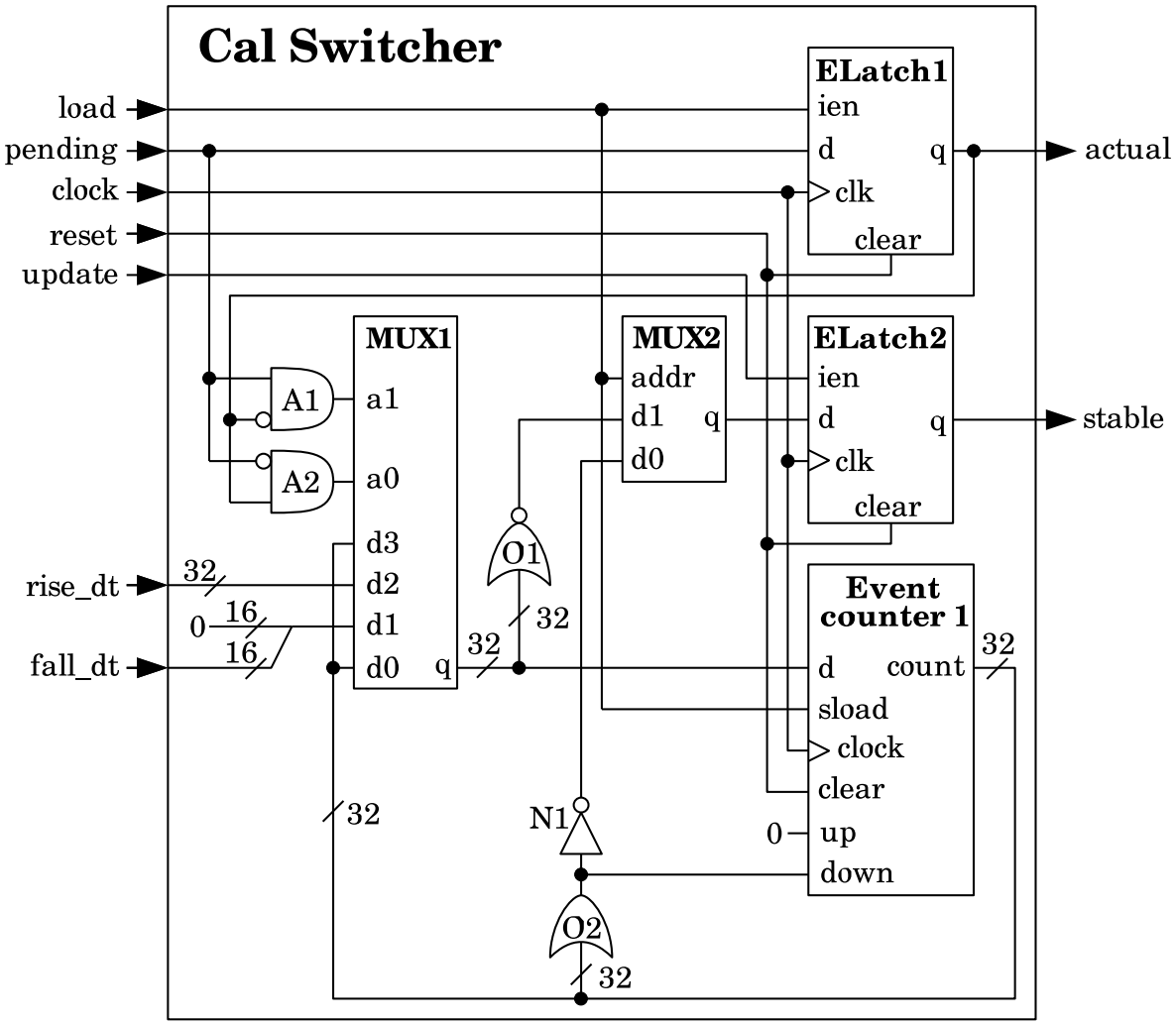

Figure 3.24: The Cal Switcher

change in the cal-diode state.

On the left of the diagram, AND gates `A1` and `A2` compare the currently commanded state of the calibration diode, as presented at the `actual` output, to the next state in the queue of the *Cal Controller*, which is taken from the `pending` input. According to the result of this comparison, multiplexer `MUX1` outputs the settling time that would correspond to the change in the state of the calibration diode, if this change were to happen at the start of the next clock-cycle. Specifically, if the diode would be switched from off to on, then the rise-time specified by the `rise_dt` input is output by `MUX1`. Alternatively, if the diode would be switched from on to off, then `MUX1` outputs the fall-time specified by the `fall_dt` input. Finally, if the diode's state would remain unchanged, then `MUX1` outputs any residual settling-time countdown from the last change in state. In other words the output of `MUX1` is a continually updated estimate of the settling time, ready a clock cycle in advance of whenever it is actually needed.

A new settling time is loaded into `Event Counter1` whenever the `load` input indicates that a new calibration diode-state has been loaded from the FIFO in the parent *Cal Controller* module. Thereafter, it counts down by one at the start of each clock cycle. If it gets to zero before a new value is next popped from the *Cal COntroller*'s FIFO, then it stops counting at zero, and the zero count indicates that the diode has settled.

The stability flag is updated at the start of each integration, by looking at the residual settling-time count. If the residual settling-time is zero, then the diode has settled, and the `stable` output becomes asserted. Otherwise, it is set to zero, to indicate that the diode is still turning on or off. The appropriate source of the residual settling time, to use for this purpose, depends on whether a new cal-diode value is currently being loaded. When no new value is being loaded, the residual count output of `Event Counter1` is used directly. When a new value is being loaded, the output of the counter still shows the residual count from the previous cal-diode transition, so instead the settling down countdown of the transition that is just starting, is taken from the output of `MUX1`. Selection between these two sources is the purpose of `MUX2`. Note that the reason that we can't always make `MUX1` the source of the residual count, is because the `pending` input that determines `MUX1`'s output, represents a future switch transition, which may not occur for a few more integrations, depending on how many integrations the current states were configured to last. Thus the output of `MUX1` is only valid when the `load` input is asserted.

The Phase Sequencer

The *Phase Sequencer* generates the control signals that command both the receiver phase switches and the routing of ADC samples to the corresponding integration bins. It also generates the flag that is used to blank ADC samples that are taken while the phase switches are changing state.

During a given scan, the phase switches cycle through a fixed set of states. The number and

choice of states is determined by the number of active phase switches, as follows.

- **When both phase switches are active**

  When both phase switches are configured to be active, a 4 state cyclic gray-code sequence is followed, since this means that only one switch changes state at a time, and thus minimizes the maximum switching rate required of the individual switches.

  At the start of the first integration, and thereafter, at the start of each new phase-switch cycle, the switches are commanded to initial states that are specified by a scan configuration parameter. Accommodating this initial state, while retaining the cyclic gray-code sequence, simply involves starting the gray-code cycle at the point in its cycle where these states are found.

- **When only one phase switch is active**

  When one switch is configured to be inactive, that switch is commanded to retain a single state throughout the scan. This state is determined by the scan-configuration parameter which sets the initial state of that phase switch at the start of each phase-switching cycle.

  The other switch is toggled on and off on successive states within the phase-switching cycle.

- **When neither phase switch is active**

  When both phase switches are configured to be inactive, both switches are perpetually commanded to maintain the initial states specified in the configuration of the scan.

The implementation of the *Phase Sequencer* is shown in figure 3.25.

In this diagram, multiplexer `MUX1` outputs the sequence of phase-switch states that corresponds to one of 16 phase-switching cycles; selected according to which phase-switches are active, and what their initial states within this cycle should be. The lower 4 bits of this sequence denote the sequence of states for phase switch `A`, while the remaining top 4 bits specify the corresponding parallel sequence of states for phase-switch `B`.

The address input of multiplexer `MUX1` is formed from the `active_a` and `active_b` inputs of the *Phase Sequencer*, which signal whether switches A and B are to be active or not, and the `closed_a` and `closed_b` inputs, which signal whether these switches should start off closed at the beginning of each phase-switch cycle, or on.

Before the first integration of a new scan, the `load` input-signal is asserted for one or more clock cycles. This loads the two parts of the phase-switching sequence output by `MUX1`, into two 4-bit PISOs. The serial outputs of these PISOs are connected back to their serial inputs, such that clocking them continuously rotates the sequence of states within the PISO, while presenting each successive state at the serial outputs. The PISOs are clocked once each time that the `phase_tick` input of the *Phase Sequencer* is asserted for one clock cycle. The

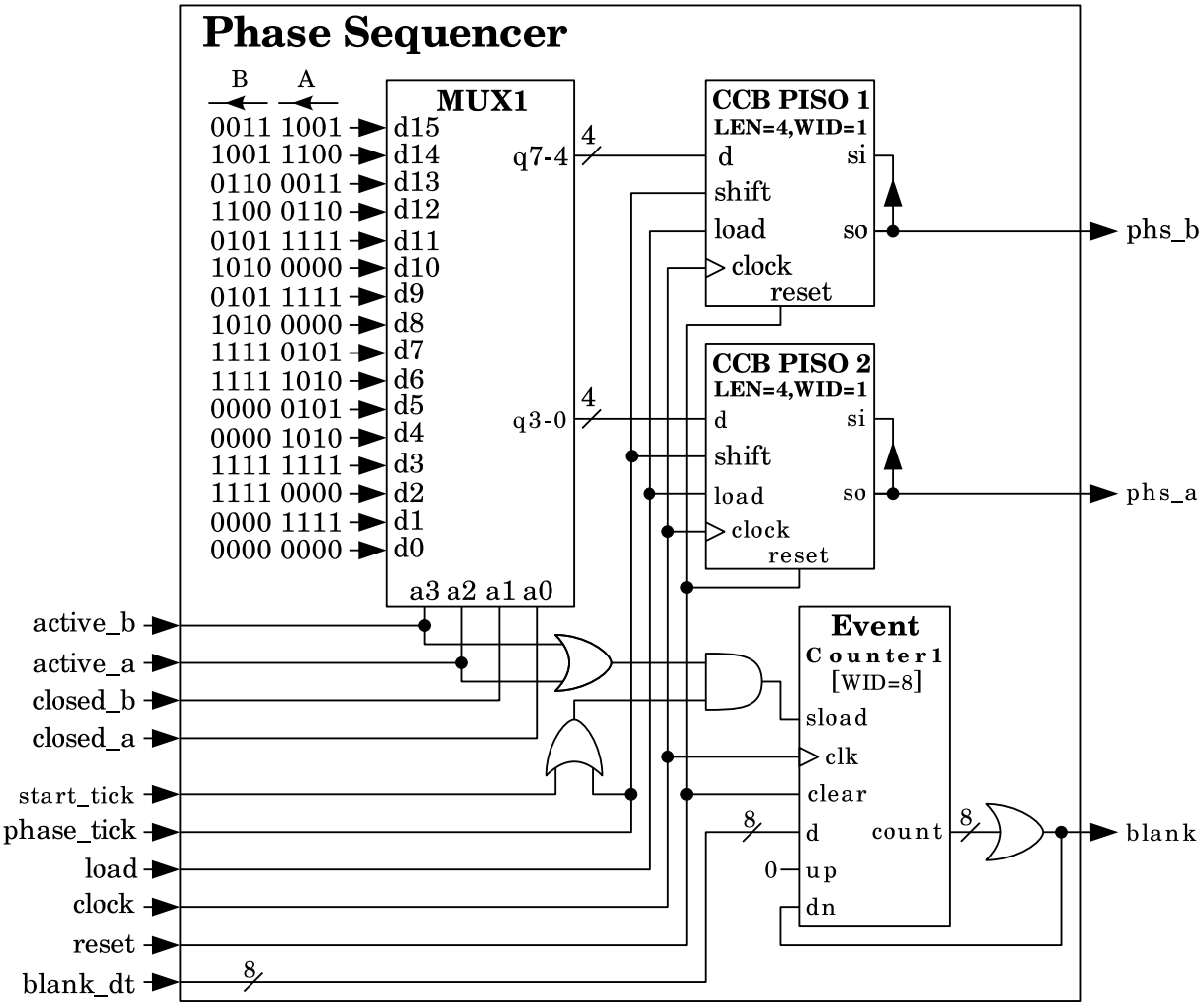

Figure 3.25: The Phase Sequencer

sequence starts on the rising clock-edge that occurs during the single clock-cycle pulse at the `start_tick` input, which happens at the official start of the scan, a few clock cycles after the `load` input is deasserted.

At the start of a new scan, and whenever the PISOs are clocked to command a new pair of phase-switch states, if either switch is configured to be switching, `Event Counter1` is loaded with the phase-switch blanking time, provided in the scan configuration. Thereafter, this counter counts down by one at the start of each clock cycle, until it reaches zero. While it remains non-zero, the `blank` output is asserted, to tell the slave FPGAs to discard any ADC samples that are acquired during this time. Thus ADC samples that are acquired while either phase switch is in the midst of changing state, are excluded from the integrated data.

3.3.3 The Slave Controller

The *Slave Controller* generates the set of signals that control how and when the four slave FPGAs acquire and optionally integrate data arriving from the receivers. The implementation of the *Slave Controller* is shown in figure 3.26.

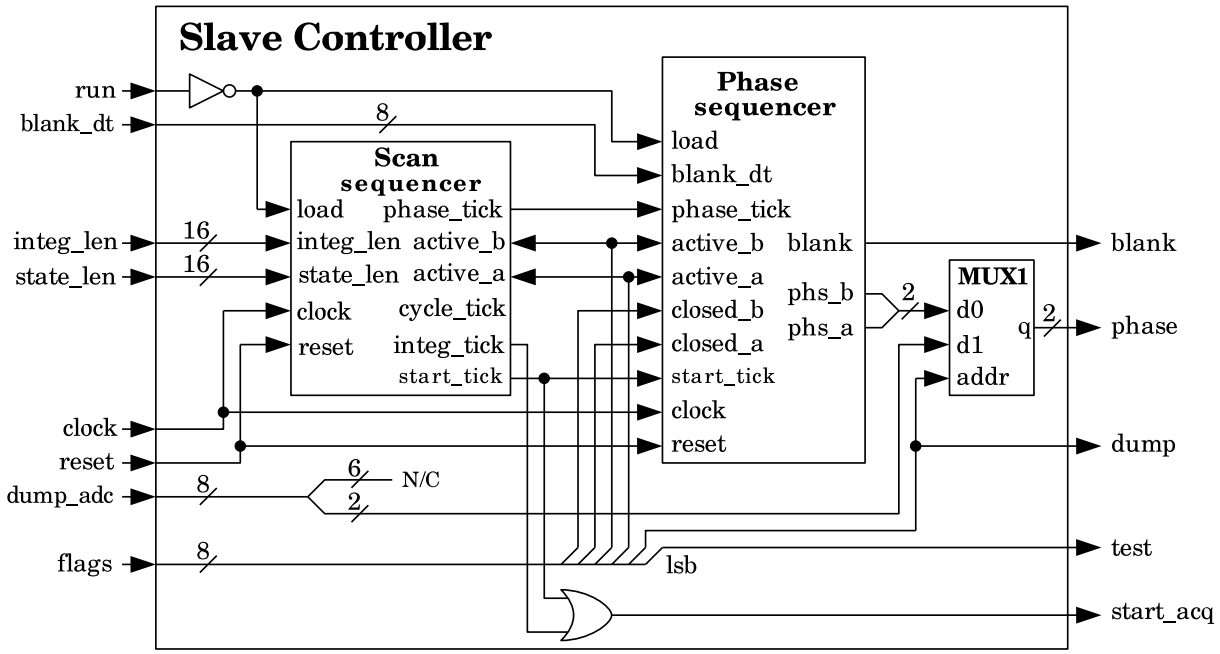

Figure 3.26: The Slave Controller

The *Scan Sequencer* and *Phase Sequencer* modules have already been documented in sections 3.3.2 and 3.3.2, respectively. However, whereas in the *Receiver Controller* module, the `phase` output of the *Phase Sequencer* is always used to command the states of the phase-switches; in the *Slave Controller* it is usually used to route the resulting samples to the

corresponding phase-switch bins. In addition, in dump mode, it is ignored entirely, because in this mode there is no integration into phase-switch bins, and instead multiplexer `MUX1` substitutes the address of the ADC whose samples are to be dumped. Similarly, whereas in the *Receiver Controller*, the `blank` output of the *Phase Sequencer* is ignored; in the *Slave Controller*, it is passed on to the slaves, to command them to drop samples during phase-switch transitions.

The `dump` output signal of the *Slave Controller*, is taken directly from the configuration flags register associated with the scan. When asserted, this flag switches the slaves into dump mode. In this mode, one slave is selected for readout by the *Data Dispatcher*, and the `phase` outputs identify a particular ADC whose raw samples must be presented, one at a time, on the data bus, for collection by the *Data Dispatcher*.

Similarly, the `test` output signal is taken from the scan's configuration flags register. When asserted, this flag tells all of the slaves to integrate or dump faked pseudo-random samples instead of the real ADC samples.

The `start_acq` output signal is asserted for one clock cycle at the start of each integration period. This tells the slaves to clear their integration accumulators, after transferring the previous contents of these accumulators to readout queues within the slaves. It also resets the pseudo-random sample-generator that is used when the `test` flag is asserted.

3.3.4 The Dispatch Controller

The *Dispatch Controller* controls the *Data Dispatcher* module. It tells it when to start collecting and transmitting the data of a new integration period to the computer. It also presents the *Data Dispatcher* with appropriately timed header information to send to the computer, along with the data of each integration period.

The implementation of the *Dispatch Controller* is shown in figure 3.27.

As in the *Receiver Controller* and *Slave Controller*, the timing signals of the *Dispatch Controller* are generated by a *Scan Sequencer* module. In particular, at the start of each integration, this causes `Event Counter1` to increment by one, after being initially zeroed before the start of the scan. Thus at the start of each integration, the output count of `Event Counter1` denotes the ordinal number of the following integration period. This is one of the required header items.

Before discussing the functions of the other components in this schematic, it is necessary to explain a fundamental difference in the timing of data collection in dump-mode and normal integration-mode.

- In dump mode, raw samples start to be collected by the *Data Dispatcher*, and dispatched to the computer, at the start of each integration period. This means that

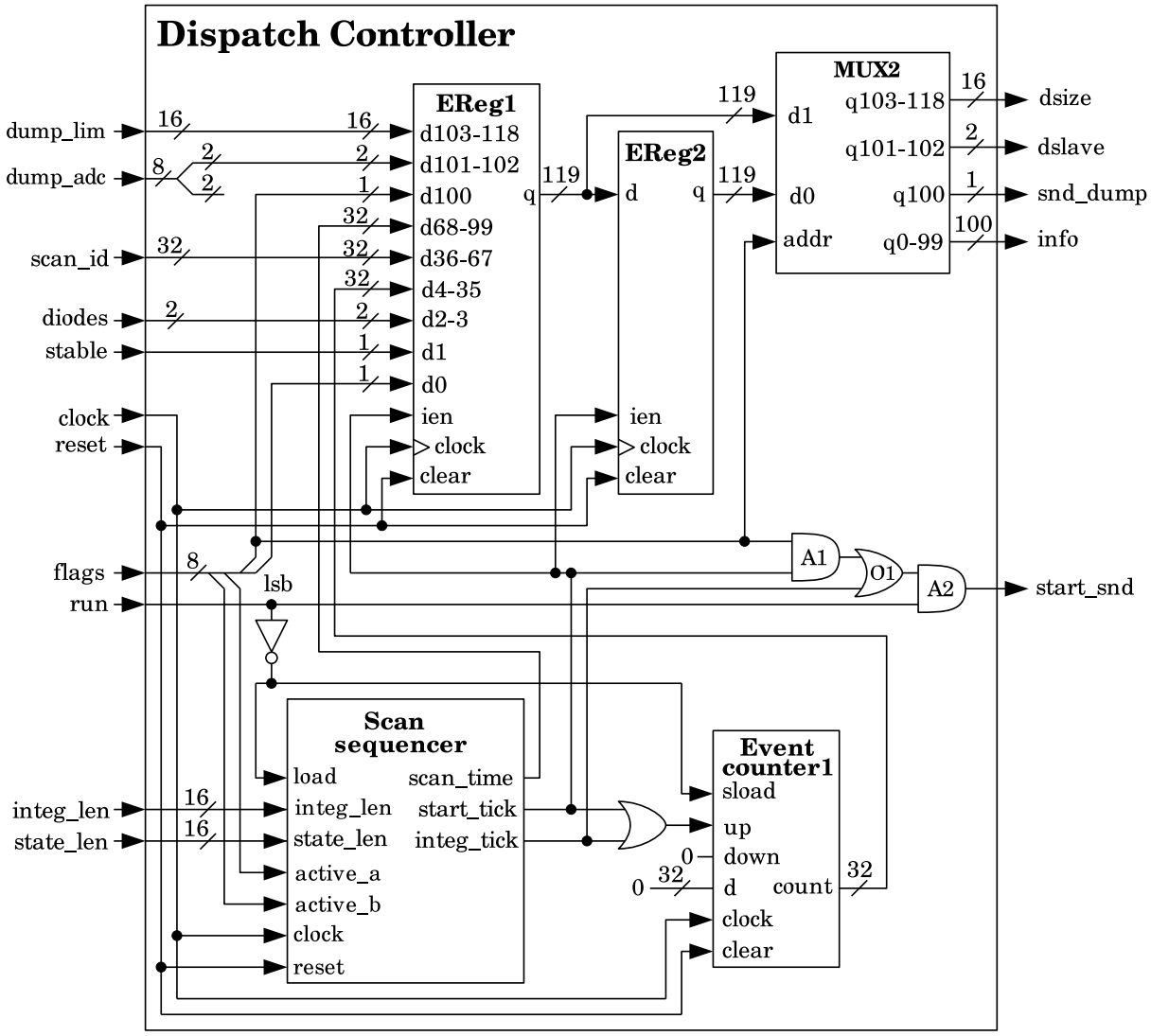

Figure 3.27: The Dispatch Controller

Dispatch Controller must present the *Data Dispatcher* with configuration and header information that is pertinent to the integration period that is just starting.

- In normal integration mode, on the other hand, the slaves integrate samples for a whole integration period, before the *Data Dispatcher* collects them and dispatches them to the computer. This means that *Dispatch Controller* must present the *Data Dispatcher* with configuration and header information that is pertinent to the integration period that has just ended.

To accommodate both of these conflicting requirements, the *Dispatch Controller* uses register `EReg1` to gather header and configuration information at the start of each integration period, while simultaneously transferring a copy of the previous contents of `EReg1` to `EReg2`. Thus one clock cycle after the start of each new integration period, `EReg1` contains the configuration and header information of the period that is just starting, while `EReg2` contains the configuration and header information of the period that just ended. Multiplexer `MUX1` selects which of these to actually pass on to the *Data Dispatcher*, according to whether dump mode is in effect of not.

Since the *Data Dispatcher* takes 2 clock cycles from the start of each integration to actually start using the above information, the information is safely presented one clock cycle in advance of when it is first used.

The `start_snd` output is asserted for one clock cycle to tell the `Data Dispatcher` to start collecting data from the slaves, and dispatch them to the computer. In dump mode, this should occur at the start of each integration period, whereas in normal integration mode, it should occur at the end of each integration period. However, since the end of one integration period is also the start of the next integration period, the integration tick from the *Scan Sequencer* is used to generate `start_snd` pulses both in dump mode and normal integration mode, with the exception of the starting pulse of the first dump-mode integration period. The latter pulse is generated by the `start_tick` output of the *Scan Sequencer*, which is generated at the start of the first integration of the scan.

To satisfy the requirements of the *Scan Initiator*, documented in section 3.3.1, AND gate `A2` ensures that a new output frame will not be started unless the `run` input signal is asserted.

3.3.5 The 1PPS Gateway

The external GBT 1PPS signal is a train of 1μs pulses, with a period of 1 second, and an amplitude of 4V. Each pulse signals the start of a new second of UT. The job of the 1PPS gateway is to convert each 1μs pulse into a pulse whose rising edge coincides with a rising edge of the FPGA clock, and whose duration is a single FPGA clock cycle. The circuit shown in figure 3.28 does this.

Since the 1PPS input signal isn't synchronous with the FPGA clock, latch 1 is used both to

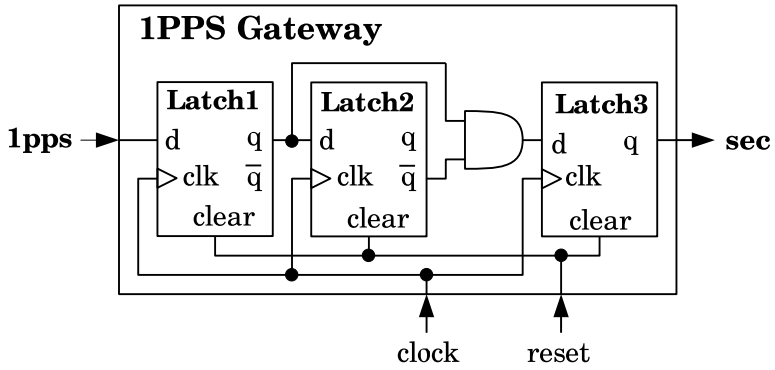

Figure 3.28: The 1PPS Gateway

synchronize the signal, and to allow one clock cycle for any metastable state in latch 1 to settle before latches 2 and 3 sample its output. Thus, one clock cycle after latch 1 latches the start of a new 1PPS pulse to its q output, latches 2 and 3 both latch a high value to their q outputs. On the following clock cycle, the $\overline{q}$ output of latch 2 is low, so latch 3 latches a low value to its output. Thus latch 3's q output goes high for precisely one clock cycle, regardless of how much longer the external input pulse lasts. Clearly, the rising edge of latch 3 trails the rising edge of the external 1PPS signal by between one and 2 FPGA clock cycles. This translates to a maximum delay of 0.2μs, which is an insignificantly small fraction of the CCB's 1ms minimum integration time.

3.3.6 Clock Conditioner

The *Clock Conditioner* takes the externally provided Green Bank 10MHz clock signal and converts it into two 10MHz clock signals, one being the main clock signal used to clock the logic within all of the FPGAs, and the other being a phase-shifted copy of this clock signal, used for clocking the ADCs. Both of these clock signals are conditioned to have 50% duty cycles.

Ideally, both the job of conditioning the main clock to have a 50% duty cycle, and the job of generating a phase-shifted version of this clock, would be performed by using the DLLs (Delay Locked Loops) provided in the Spartan-3 DCMs (Digital Clock Managers). Unfortunately, it turns out that these DLLs have a minimum clock frequency of 24MHz, meaning that they can't be used to condition the 10MHz clock frequency of the CCB. Fortunately, the DFS (Digital Frequency Synthesis) modules in the same DCMs don't have this restriction, and although these can only perform frequency multiplication, and duty-cycle correction, these features are sufficient for generating the two clock signals.

The basic idea is to take the 10MHz Green Bank reference signal, use a DFS to generate

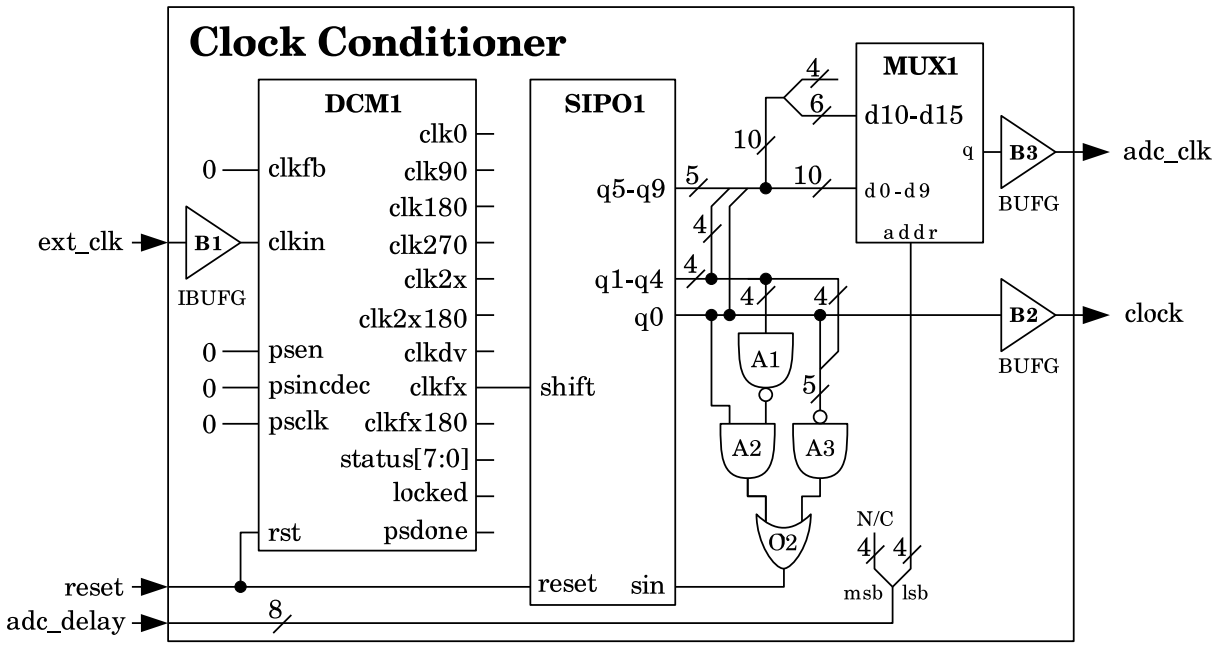

Figure 3.29: The Clock Conditioner

a 100MHz signal, divide this back down to 10MHz, to form the main duty-cycle-corrected FPGA clock signal, and delay this by a configurable number of 100MHz (10ns) clock cycles, to generate the phase-shifted ADC clock signal. The implementation is shown in figure 3.29.

In the diagram, buffer `B1` is the input-buffer of the global clock pin at which the Green Bank 10MHz reference signal enters the master FPGA. Buffers `B2` and `B3` are global clock buffers, which connect the two output clock signals to, to separate global clock networks within the master FPGA.

The DFS in digital Clock Manager, `DCM1`, is configured to multiply the frequency of the 10MHz clock signal, presented at its `clkin` input, by 10, and present the resulting 100MHz clock signal at its `clkfx` output. The only static configuration parameters of `DCM1` that need to be changed from their default values, are the following.

- `CLK_FEEDBACK = None`

  This turns off the DLL by disabling its feedback path. This is necessary since the input frequency is too low for the DLL to lock.

- `CLKFX_MULTIPLY = 10`

  This is the factor of 10 by which to multiply the input 10MHz clock-signal.

- `LOC = TBD`

Spartan-3 FPGAs contain 4 DCMs, each one located in a different physical corner of the FPGA. The `LOC` parameter specifies which one of the DCMs to use, and should be chosen to use the DCM closest to whichever pin is used for the clock input.

The 100MHz signal generated at the `clkfx` output, is used to clock the contents of a SIPO (Serial-In, Parallel Out) shift register. This shift register is used both to divide the signal back down to 10MHz, and to generate the 10 possible 10ns delays, as follows. In general, any SIPO (Serial-In, Parallel Out) shift register with at least N stages, can be used to divide the frequency of its shifting clock by $2N$, and present N differently delayed copies of this output. In particular, if the outputs of the successive stages of the shift-register are labeled $\mathtt{q}_0 ... \mathtt{q}_{N-1}$, then at the start of each input clock cycle, the serial input, `sin`, of the shift register must be given by,

$$\mathtt{sin} = \overline{\mathtt{q}}_0\overline{\mathtt{q}}_1 ... \overline{\mathtt{q}}_{N-1} + \mathtt{q}_0\overline{(\mathtt{q}_1\mathtt{q}_2 ... \mathtt{q}_{N-1})} \tag{3.1}$$

where as usual, logical AND is represented by multiplication, and logical OR is represented by summation. This equation is easy to understand when one realizes that the contents of the N stages represent half of the output clock-period. Basically, the equation keeps feeding the shift register input with the same value as it did on the previous clock cycle, except on the clock cycles when it sees that all of the stages have the same values, at which point it feeds the opposite value into the shift register, to start the next half of the output clock cycle. Note that if any of the shift-register stages somehow get toggled into an incorrect state, say by a power-glitch, then although initially this will generate a corresponding glitch in the output clock signal, the properties of the above equation are such that a clean clock signal will be restored within at most $N - 1$ clock cycles, with the only lasting effect being a constant shift in the arbitrary phase origin.

So, to divide the 100MHz clock frequency by 10, we need a SIPO with at least 5 stages. This will generate 5 delayed versions of the divided 100MHz clock, with delays every 10ns over the range 0ns...40ns. Since this only covers half a clock cycle, whereas we need a full clock cycle, a SIPO of 10 stages is actually needed to provide 5 more delay taps. Thus in figure 3.29, the first 5 stages of a 10 stage SIPO are used, along with gates `A1`, `A2`, `A3` and `O1`, to implement equation 3.1, while both these 5 initial stages, plus the remaining 5 stages, are used to generate delayed versions of the divided clock signal, every 10ns, over the range 0ns...90ns.

There are three reasons for delaying the ADC clock signal relative to the main FPGA clock signal.

- One reason to delay the ADC clock signal is to accomodate the delay between the ADC clock edge, and data being output by the ADC. The data-sheet of the AD9240 ADC says that the time taken between the rising clock edge at the ADC clock input, and a valid new sample being available at the ADC data outputs, ranges from between 8ns

and 19ns. Thus the rising FPGA clock-edge that is used to latch these outputs into the slave FPGAs, must occur more than 19ns after the rising ADC clock edge.

- Another reason to delay the ADC clock, is to arrange that the noisy period around the active edge of the FPGA clock, not occur while the ADC is at its most sensitive to external noise.

- Finally, potential clock-skew problems can be remedied by modifying the ADC clock-delay.

Since the optimal ADC clock delay will need to be determined empirically, `MUX1` allows any of the 10 delays to be selected dynamically, according to the address contained in the `adc_delay_reg` register. Since the 4-bit MUX can select between 16 values, whereas there are only 10 possible delays, the MUX address is interpretted modulo 10, with the highest 6 addresses thus selecting the same delays as the lowest 6 addresses. As such, the value of the `adc_delay_reg` register selects the delay as a multiple of 10ns, modulo 100ns.

A reasonable starting point for the value of the `adc_delay_reg` register would be 2, to select a 20ns delay. This would leave 20ns for all FPGA operations to cease, after each rising edge of the FPGA clock, followed by a 80ns quiet period, during which the ADC would take up to 20ns to output its previous value, while simultaneously starting to measure its next value.

3.4 Custom generic components

The components that are described in this section are custom components that are used in more than one part of the CCB.

3.4.1 The ELatch component

There are many occasions when one wants to latch values synchronously with the clock, but only at particular clock cycles. The naive way to handle this is to AND the clock signal going to the edge-sensitive input of a latch, with an enabling signal. The problem with this is that if your clock-enabling signal is itself generated by the output of a synchronous latch, whose output changes just after the rising edge of the clock, then the output of the AND gate may end up generating a small glitch between the rising edge of the clock signal and the rising edge of the enabling signal, and this glitch will either cause the following latch to erroneously latch its input, or it will push the latch into a metastable state, due to the shortness of the glitch. Although one could work around this by using a negative edge-triggered latch to assert the enabling signal before the rising edge of the clock, this can lead to conflicts with other components that need the same enabling signal to be generated synchronous with the rising edge of the clock. The use of both active edges also doubles the amount of time that

the FPGA generates switching transients, and thus would make it harder to find a quiet time for the ADCs to capture their inputs.

The solution to this problem is to implement a latch with an enable input, containing an embedded D-type latch that always captures its input value at the rising edge of the clock, but with a multiplexer in front of its input, which simply directs the existing value of the latch back to this input, when the latch value should remain unchanged. The implementation is shown in figure 3.30, where the input-enable input is labeled `ien`.

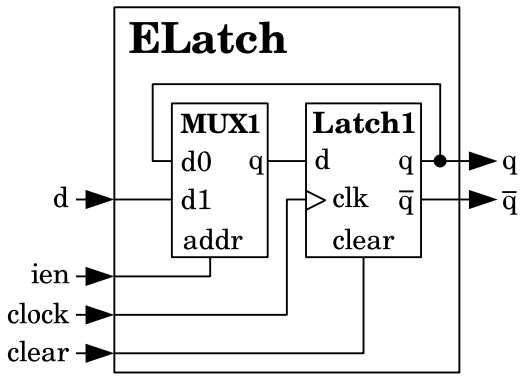

Figure 3.30: A D-type latch with a synchronous input-enable input

3.4.2 The EReg component

The *EReg* component is simply a multi-input version of the *ELatch* component described in section 3.4.1. Thus it is simply a synchronous register with a synchronous enable input (`ien`). Its implementation is shown in figure 3.31.

3.4.3 The CCB PISO component

Conventional PISO (Parallel In Serial Out) components respond to the active edge of the clock by either loading parallel data, or shifting out serial data, depending on the state of a single `load/`$\overline{\texttt{shift}}$ input. The problem with this is that without disabling the clock input, one can't keep the contents of the PISO unchanged for one or more clock cycles. Disabling the clock isn't as trivial as it sounds. The only reliable way to do it without introducing false clock edges, is to use a latch, triggered off the opposite edge of the clock, to enable or disable the clock signal while it is known to be low. This, unfortunately means that the enabling signal is sampled half a clock cycle earlier than the normal `load/`$\overline{\texttt{shift}}$ input, which can break timing expectations.

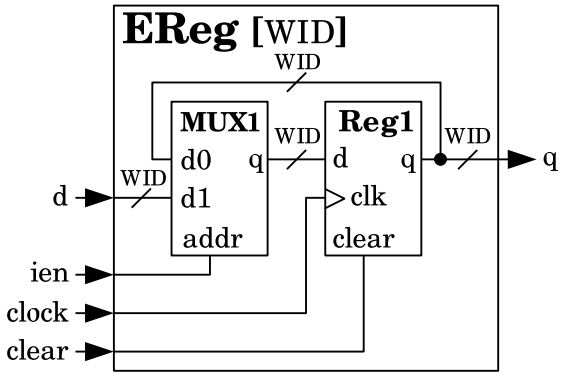

Figure 3.31: A register with a synchronous input-enable input

A better method is to create a custom PISO with the following properties. The PISO should have separate `load`-enable and `shift`-enable inputs, rather than one combined `load/`$\overline{\texttt{shift}}$ input, such that there can be more choices than just shifting or loading. As in the conventional PISO, these inputs should be acted on at the active edge of the clock, but unlike a normal PISO, the contents of the PISO should remain unchanged unless at least one of them is asserted.

A single node of such a PISO is shown in figure 3.32.

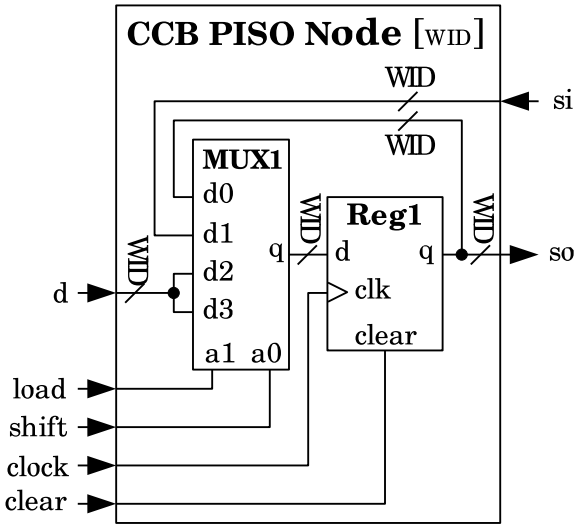

Figure 3.32: One node of a CCB PISO component

Note that the parameter `WIDTH` specifies the number of bits serially shifted in and out of the serial inputs and outputs, `si` and `so`, or parallel-loaded via the `d` input. At the rising edge of

the clock, the PISO performs the following operations, according to the states of the `shift` and `load` inputs.

- `load`=0, `shift`=0

  When neither of the enable inputs are asserted, multiplexer `MUX1` routes the current value of register `Reg1` back to its input, such that the contents of the latch remain unchanged.

- `load`=0, `shift`=1

  When just the `shift` input is asserted, multiplexer `MUX1` routes the value at the serial-in (`si`) input to the input of the register `Reg1`, such that the contents of the register are replaced by whatever value is at the `si` input. Usually this input is connected to the `so` output of the previous node in the PISO.

- `load`=1, `shift`=0 or 1

  Whenever the `load` input is asserted, regardless of the state of the `shift` input, multiplexer `MUX1` routes the value at the `d` input of the node to the input of register `Reg1`. Thus the contents of the register get replaced with whatever value was at the `d` input of the node.

*CCB PISO Node*s are strung together in a chain to form a PISO, as shown in figure 3.33.

The width of the PISO nodes is set by the `WID` parameter, while the number of nodes in the PISO, is set by the `LEN` parameter.

3.4.4 The Event Counter component

In several places a counter is required that does one of 4 things synchronously with the rising edge of the clock, and does nothing at any other time. These things are:

- Load a new count into the counter if the `sload` input is asserted, regardless of the states of the `up` and `down` inputs.
- Increment the output of the counter by 1 if the `up` input is asserted, and neither the `sload` nor the `down` inputs are asserted.
- Decrement the output of the counter by 1 if the `down` input is asserted, and neither the `sload` nor the `up` inputs are asserted.
- Leave the output count unchanged if none of the `sload`, `up` or `down` inputs are asserted.

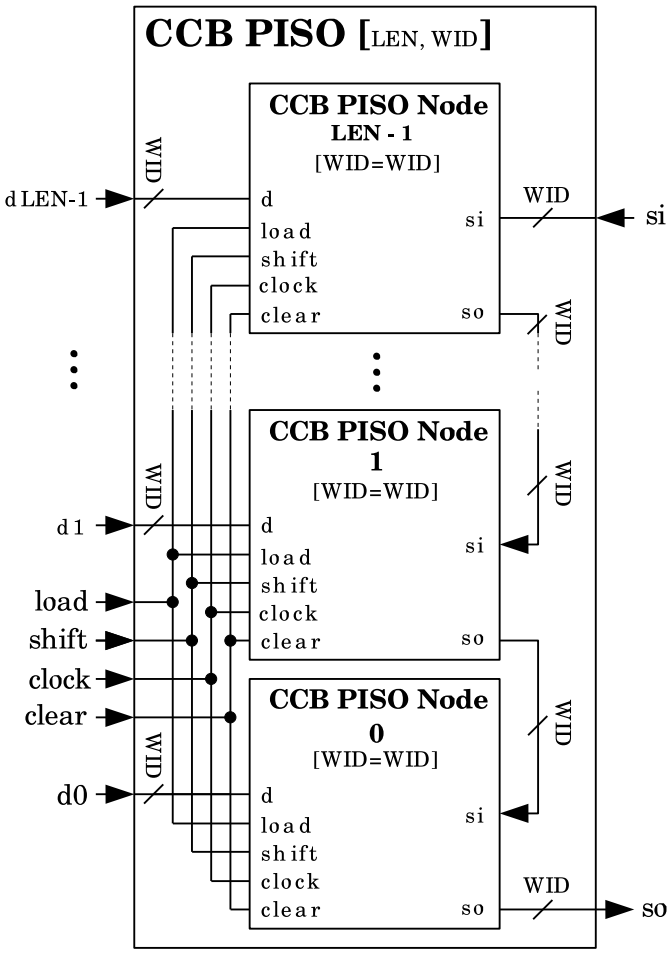

Figure 3.33: A complete CCB PISO component

The *Event Counter* component, shown in figure 3.34, shows the inputs and outputs of this counter.

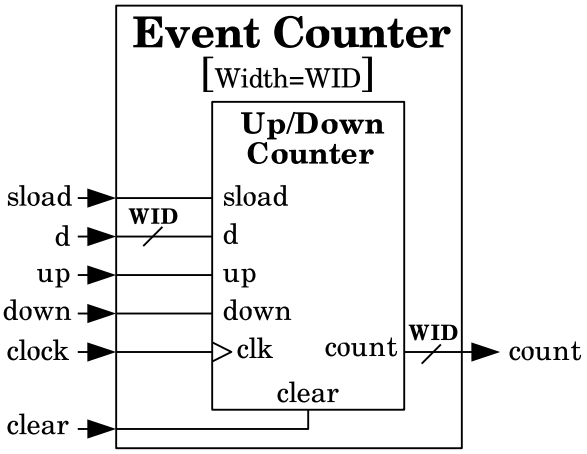

Figure 3.34: An up/down counter with synchronous parallel load capability

The VHDL code shown in figure 3.35 is an example of how this counter could be implemented, assuming that there isn't anything equivalent in Xilinx's library of counters.

Note that this is one case where using VHDL makes much more sense than schematic capture, because of the VHDL addition and subtraction operators, which take advantage of the fast-carry networks built into Spartan 3 FPGAs.

3.4.5 The Metronome component

Metronome components periodically generate a pulse, one clock-cycle in length, every time that they have synchronously counted a specified number of events at their `step` input. They are implemented by the circuit shown in figure 3.36,

The period between pulses must initially be set by asserting the `load` input for one or more clock cycles. This takes the period, specified by the `nstep` input, uses it to set the initial count of the counter, and copies it into register `EReg1`. The value of this register is thereafter used to reload the counter for a new countdown after each output tick.

The terminal count at which the component outputs a pulse, and prepares to reload its internal counter, depends on whether the `step` input was asserted when the counter last reloaded itself. If it was asserted, then the reload operation will have masked one countdown event, and the count should only go down to 1, whereas if it wasn't asserted, then the reload didn't represent a countdown event, and the count should go all the way down to 0. NOR gate `N1` generates the terminating output pulse when the most significant `WID-1` bits of the

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity event_counter is
  generic(WID: std_logic_vector := 16);
  port(
       clk, sload, clear, up, down: in std_logic;
       d, q: out std_logic_vector(WID-1 downto 0)
  );
end event_counter

architecture event_counter_arch of event_counter is
begin  -- event_counter_arch
  signal tmp : std_logic_vector(WID-1 downto 0);
  begin
    process(clk, clear)
    begin  -- process
      if(clear='1') then
        tmp <= (others => '0');
        elseif(clk'event and clk='1') then
          if(sload='1') then
            tmp <= d;
          elsif(up='1' and down='0') then
            tmp <= tmp + 1;
          elsif(down='1' and up='0') then
            tmp <= tmp - 1;
          end if
        end if
    end process;
    q <= tmp;
end event_counter_arch;
```

Figure 3.35: A VHDL implementation of the Event Counter component

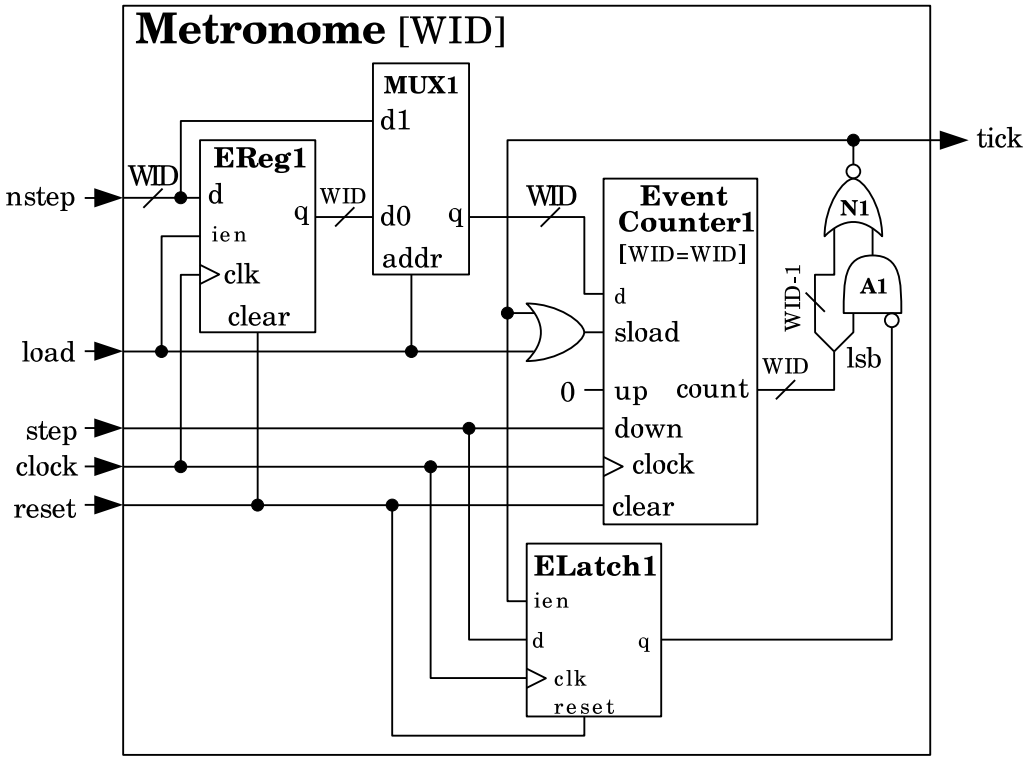

Figure 3.36: The Metronome periodic pulse-generator

output count are all zero, and the least significant bit AND'd with $\overline{\texttt{q}}$ is zero. Thus when the `step` input was asserted during the previous reload of the counter, $\overline{\texttt{q}}$ is zero, and the counter thus stops at 1, whereas when the `step` input was not asserted during the previous reload of the counter, $\overline{\texttt{q}}$ is 1, and the counter doesn't stop until 0.

The output pulse of the *Metronome* component is also used internally, to reload the counter for the next count-down interval, via the synchronous `sload` input of the counter. At this time, the counter is preset to the value stored in `Ereg1`, rather than from the external `nstep` input.

While the `load` input signal is asserted, the counter is held with the value of its `nstep` input, which is the value that it normally has one clock cycle after outputting a tick. During this time, the signal at the `step` input, is ignored. The `step` input is not heeded until the first rising clock edge that follows the `load` input going low, at which point it causes the counter to count down, if it is asserted. Beware that there is a pipeline delay of one clock cycle between a synchronously generated pulse at the `step` input, and any corresponding change in the `tick` output.

Appendix A

CCB control and configuration registers

The CCB firmware is controlled and configured by a set of 8-bit registers written to via the EPP parallel port of the CCB control computer. The computer interface to these registers is implemented by the *Control Gateway*, and their values are used by the *State Generator*. Whereas all registers are treated identically by the *Control Gateway*, the *State Generator* divides them into "action" and "configuration" registers.

When an action register is written to by the computer, the *State Generator* immediately does something in response. An example is the register whose value tells the FPGA to start a new scan or intra-scan. In this case the attention signal of the action register sets off the action, while the register value contains any information associated with the command.

When a configuration register is written to, the attention signal of the register is ignored, since the updated configuration isn't relevant until a new scan is started. To prevent configuration updates from affecting the current scan, the configuration registers in the *Control Gateway* are only copied into working registers within the *State Generator* at the start of each new scan.

A summary of all of the registers is given in table A.1.

The list of action registers

- `start_scan_reg` – Start a new scan or intra-scan.

  Whenever this register is written to by the computer, the *State Generator* first halts any existing scan, then, depending on the revised contents of the register, starts a new one, either immediately, or at the start of the next second.

Name	Type	Address of MSB	Address of LSB
start_scan_reg	action	00	00
cal_diode_reg	action	01	01
scan_flags_reg	config	02	02
state_len_reg	config	03	04
blank_dt_reg	config	05	05
diode_rise_reg	config	06	09
diode_fall_reg	config	10	11
integ_len_reg	config	12	13
roundtrip_dt_reg	config	14	14
holdoff_dt_reg	config	15	15
dump_adc_reg	config	16	16
dump_lim_reg	config	17	18
adc_delay_reg	config	19	19
scan_id_reg	config	20	23

Figure A.1: A list of all CCB registers

The bit assignments of the register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
00	X	X	X	X	X	X	X	sync

The meanings of the bit-value names are:

- **sync**
  If 0, start the new scan as quickly as possible. If 1, start the new scan at the next rising edge of the 1PPS signal.
- **X**
  The value of this bit is currently unused, and should be assigned 0.

- cal_diode_reg – Queue a new cal-diode configuration.

Append one entry to the queue of future multi-integration calibration-diode configurations. This register is to be written to whenever the computer receives a cal_intr interrupt. In particular, immediately after the computer writes to the start_scan_reg register, to initiate a new scan, the master FPGA generates a cal_intr CPU-interrupt to ask the computer for the configuration of the first integration of the new scan. The computer should then respond by writing the desired initial cal-diode states, and for how many integrations these states should be commanded, in the cal_diode_reg register. Thereafter, each time that a cal_intr interrupt is received by the computer, it should send the configuration of the next one or more integrations that follow the integrations that were last configured in the previous write to cal_diode_reg.

Since the receipt of a `cal_intr` interrupt is the only race-condition-free way that the computer can reliably know when there is space for a new configuration byte within the queue of cal-diode configurations, the computer must not write to the `cal_diode_reg` register at any other time.

The configuration bits within the `cal_diode_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
01	`n5`	`n4`	`n3`	`n2`	`n1`	`n0`	diode_b	diode_a

Where the meanings of the bit-value names are:

- **diode_a**
  If 0, calibration diode A should be commanded off at the start of the target integration. If 1, calibration diode A should be commanded on at the start of the target integration.
- **diode_b**
  If 0, calibration diode B should be commanded off at the start of the target integration. If 1, calibration diode B should be commanded on at the start of the target integration.
- **n0..n5**
  These bits together form a positive integer count, with bit `n0` denoting the least significant bit, and bit `n5` denoting the most significant bit. This count specifies for how many consecutive integrations the specified calibration diode states should be used. Thus, up to 64 consecutive integrations can be configured to have the same cal-diode state, in a single write to the `cal_diode_reg` register.

The list of configuration registers

- `scan_flags_reg` – Scan configuration flags.

  This register contains miscellaneous scan-specific single-bit configuration flags.

  The bit assignments of the register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
02	`X`	`X`	`close_b`	`close_a`	`switch_b`	`switch_a`	`dump`	`test`

The meanings of the bit-value names are:

- **test**
  If 0, use real ADC samples as the input to the CCB. If 1, use pseudo-random fake samples as the input to the CCB.

- **dump**
  If 0, send integrated samples to the computer. If 1, send raw ADC samples to the computer.
- **switch_a**
  If 0, phase-switch A should be held in the state specified by the `close_a` bit, for the duration of each phase-switch cycle. If 1, phase-switch A should be toggled on and off during each phase-switch cycle.
- **switch_b**
  If 0, phase-switch B should be held in the state specified by the `close_b` bit, for the duration of each phase-switch cycle. If 1, phase-switch B should be toggled on and off during each phase-switch cycle.
- **close_a**
  If 0, phase-switch A should be opened at the start of each phase-switch cycle. If 1, phase-switch A should be closed at the start of each phase-switch cycle.
- **close_b**
  If 0, phase-switch B should be opened at the start of each phase-switch cycle. If 1, phase-switch B should be closed at the start of each phase-switch cycle.
- **X**
  The value of this bit is currently unused, and should be assigned 0.

- `state_len_reg` – The number of samples per phase-switch state.

  This register specifies how long a single combination of phase-switches should last within a a phase-switch cycle. It is expressed as a number of 100ns ADC samples, and is a 16 bit number which extends across two 8-bit registers. It has minimum and maximum supported values of 250 and 65535, which correspond to durations of 25μs and 6.5ms.

  The configuration bits within the `state_len_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
03	b15	b14	b13	b12	b11	b10	b9	b8
04	b7	b6	b5	b4	b3	b2	b1	b0

  The meanings of the bit-value names are:

  - **b0..b15**
    Bits 0 to 15 of the 16-bit number, with `b0` denoting the least significant bit, and `b15` denoting the most significant bit.

- `blank_dt_reg` – The phase-switch settling time.

This register specifies how many 100ns ADC samples should be blanked to account for the settling time of the phase switches. The duration occupies a single 8-bit register, and thus allows for settling times of between 0 and 25.6μs.

The configuration bits within the `blank_dt_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
05	b7	b6	b5	b4	b3	b2	b1	b0

The meanings of the bit-value names are:

- **b0..b7**
  Bits 0 to 7 of the 8-bit number, with `b0` denoting the least significant bit, and `b7` denoting the most significant bit.

- `diode_rise_reg` – The rise-time of the cal diodes.

This register specifies how long it takes for the effects of turning a calibration diode on, to stabilize to below the limits of detectability. It is expressed as a number of 100ns ADC samples, and is a 32 bit number which extends across four 8-bit registers. The use of 32 bits establishes a maximum rise-time of about 7 minutes. This will hopefully be overkill, but the long duration allows for the potential that the diodes might need a significant amount of time to respond thermally to the change in loading when they are switched on.

The configuration bits within the `diode_rise_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
06	b31	b30	b29	b28	b27	b26	b25	b24
07	b23	b22	b21	b20	b19	b18	b17	b16
08	b15	b14	b13	b12	b11	b10	b9	b8
09	b7	b6	b5	b4	b3	b2	b1	b0

The meanings of the bit-value names are:

- **b0..b31**
  Bits 0 to 31 of the 32-bit number, with `b0` denoting the least significant bit, and `b31` denoting the most significant bit.

- `diode_fall_reg` – The fall-time of the cal diodes.

This register specifies how long it takes for the effects of turning off a calibration diode to stabilize to below the limits of detectability. It is expressed as a number of 100ns

ADC samples, and is a 16 bit number which extends across two 8-bit registers. The use of 16 bits establishes a maximum diode fall-time of 6.6ms.

The configuration bits within the `diode_fall_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
10	b15	b14	b13	b12	b11	b10	b9	b8
11	b7	b6	b5	b4	b3	b2	b1	b0

The meanings of the bit-value names are:

- **b0..b15**
  Bits 0 to 15 of the 16-bit number, with `b0` denoting the least significant bit, and `b15` denoting the most significant bit.

- `integ_len_reg` – The duration of an integration period.

  This register specifies the duration of each integration, as a multiple of the currently configured phase-switch cycle duration. This is a 16 bit value, which extends across two 8-bit registers. Since phase-switch states are required to persist for no less than 25μs, and up to 32 states are allowed per phase-switch cycle, the use of 16 bits establishes a minimum maximum of 52 seconds per integration. This far exceeds the roughly estimated 1 second duration at which a weak-signal would saturate the 32-bit integration accumulators.

  The configuration bits within the `integ_len_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
12	b15	b14	b13	b12	b11	b10	b9	b8
13	b7	b6	b5	b4	b3	b2	b1	b0

  The meanings of the bit-value names are:

  - **b0..b15**
    Bits 0 to 15 of the 16-bit number, with `b0` denoting the least significant bit, and `b15` denoting the most significant bit.

- `roundtrip_dt_reg` – The CCB/receiver round-trip delay.

  This register specifies an estimate of the length of time between the CCB toggling a switch control-signal, and the first effects of this operation being seen in the detected signal that arrives at the CCB. It should be on the short side of the actual estimated value, such that samples from when the switch began changing the signal, don't get incorrectly included with the pre-switch samples.

  While the actual number will have to be measured empirically, the major contributors can be estimated, as follows.

- The opto-isolators that drive the receiver control signals are likely to contribute a propagation delay of around 100ns, which corresponds to one FPGA clock cycle.
- Once the control signal arrives at the receiver, and the switches in the receiver start to respond to it, the perturbed astronomical signal has to go through an 8-pole 2MHz Bessel low-pass filter in the receiver, which delays the perturbed switched signal by 250ns.
- The next major contributor is the 4-stage pipeline in the ADCs delays, which add another 300ns.
- The input latch in the slave FPGAs further delays the use of the digitized signal by another 100ns.

Adding these figures up, one gets a lower bound of 750ns. In practice RFI filters, cables etc, will add further delays, so 1μs seems like a reasonable estimate.

The `roundtrip_dt_reg` register is 8 bits wide, and expresses the delay as a multiple of the 100ns ADC sampling interval. Thus the maximum supported round-trip delay is 25.6μs. The bit-assignments within the register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
14	b7	b6	b5	b4	b3	b2	b1	b0

The meanings of the bit-value names are:

- **b0..b7**
  Bits 0 to 7 of the 8-bit number, with `b0` denoting the least significant bit, and `b7` denoting the most significant bit.

- `holdoff_dt_reg` – The interrupt hold-off delay.

  This register sets the maximum rate at which the CCB is allowed to generate interrupts, expressed as the minimum interval between interrupts. It is a 5-bit number which has 1 added to it, before being scaled by 25.6μs, to arrive at the actual holdoff interval. Thus the range of supported holdoff intervals is $25.6\mu\text{s} \leftrightarrow 819.2\mu\text{s}$

  The configuration bits within the `holdoff_dt_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
15	X	X	X	b4	b3	b2	b1	b0

The meanings of the bit-value names are:

- **b0..b4**
  Bits 0 to 4 of the 5-bit number, with `b0` denoting the least significant bit, and `b4` denoting the most significant bit.

– **X**
The value of this bit is currently unused, and should be assigned 0.

- `dump_adc_reg` – The ADC to sample in dump mode.

  This register specifies which of the ADCs is to sampled when dump mode is enabled. There are 16 ADCs, split equally between 4 slave FPGAs, so the specification of the ADC is a 4 bit number with 2 bits specifying a slave FPGA, and 2 bits specifying a particular ADC handled by that slave.

  Specifically, the configuration bits within the `dump_adc_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
16	X	X	X	b4	slave1	slave0	sampler1	sampler0

  The meanings of the bit-value names are:

  – **slave0..slave1**
  Bits 0 and 1 is the 2-bit ID of the slave that handles the target ADC.

  – **sampler0..sampler1**
  Bits 1 and 2 is the 2-bit ID of the sampler that samples the target ADC.

  – **X**
  The value of this bit is currently unused, and should be assigned 0.

- `dump_lim_reg` – The maximum number of dump-mode samples to collect per integration period.

  When in dump mode, this register determines how many samples the CCB should attempt to collect, per integration period, before sending these to the computer. The actual number collected will be further limited to the size of the frame buffer.

  The configuration bits within the `dump_lim_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
17	b15	b14	b13	b12	b11	b10	b9	b8
18	b7	b6	b5	b4	b3	b2	b1	b0

  The meanings of the bit-value names are:

  – **b0..b15**
  Bits b0 to b15 denote a 16-bit unsigned integer, with b0 being the least significant bit. This integer specifies the maximum number of samples to attempt to collect, per integration period.

- `adc_delay_reg` – The ADC clock-delay.

  This register sets the delay between the rising edge of the main FPGA clock, and the rising edge of the ADC clock.

  The configuration bits within the `adc_delay_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
19	X	X	X	X	b3	b2	b1	b0

  The meanings of the bit-value names are:

  - **b0..b3**
    Bits b0 to b3 denote a 4-bit unsigned integer, with b0 being the least significant bit. The ADC clock-delay is set to this number, modulo 10, multiplied by 10ns.
  - **X**
    The value of this bit is currently unused, and should be assigned 0.

- `scan_id_reg` – The ID to assign the scan.

  This register specifies the identification number that is to be placed in the headers of data frames that are taken during the scan that is being configured.

  The configuration bits within the `scan_id_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
20	b31	b30	b29	b28	b27	b26	b25	b24
21	b23	b22	b21	b20	b19	b18	b17	b16
22	b15	b14	b13	b12	b11	b10	b9	b8
23	b7	b6	b5	b4	b3	b2	b1	b0

  The meanings of the bit-value names are:

  - **b0..b31**
    Bits 0 to 31 of the 32-bit idenfication number, with `b0` denoting the least significant bit, and `b31` denoting the most significant bit.