The designs of the master and slave CCB FPGAs

[Document number: A48001N004, revision 11]

Martin Shepherd, California Institute of Technology

June 15, 2005

This page intentionally left blank.

Abstract

The aim of this document is to detail the design of the firmware in the CCB slave and master FPGAs, and define their interfaces to the rest of the CCB hardware and software. The design is presented in a hierarchical manner, starting with block diagrams of major components and their interconnections, and ending with either low level schematics, or with VHDL components.

Contents

List of Figures

Chapter 1

Introduction

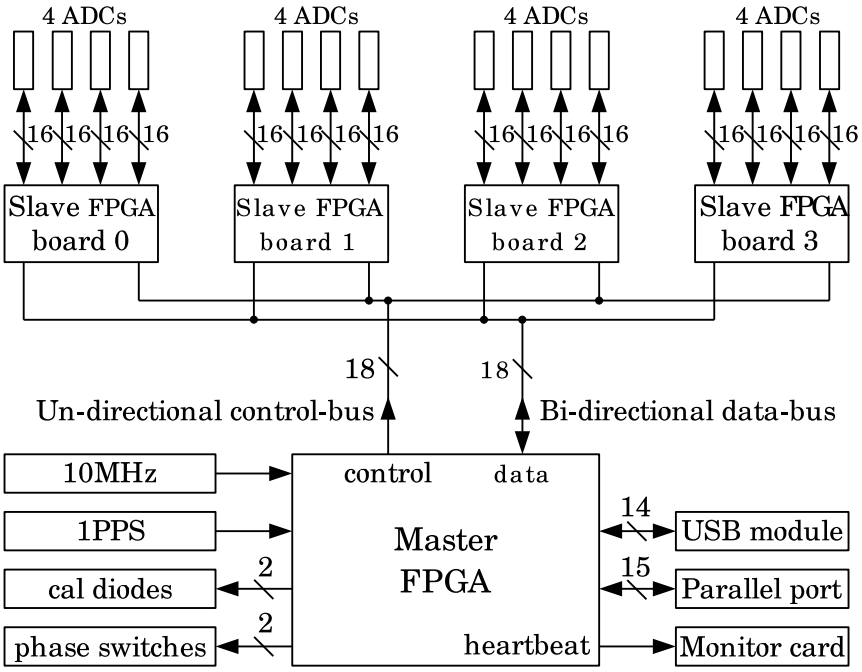

Figure 1.1: An overall summary of the FPGA connections

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, communicates with a host computer via a USB link and an EPP-enabled parallel port, and generates signals that control the calibration diodes and phase switches in an external differential radiometer. All of its timing signals are derived from the Green Bank 10MHz and 1PPS reference signals.

There are 16 ADCs in the CCB, partitioned equally between the four slave FPGAs. Each

slave FPGA simultaneously clocks out 14-bit samples from its 4 ADCs, at a continuous 10MSPS, and either integrates these samples until told to deliver the integrations to the master FPGA, or is told to deliver them individually to the master FPGA. In either case, the resulting data are first streamed to the master FPGA, over the master-slave data-bus, and are then streamed to the computer via the USB link. Note that although the master-slave data-bus is bi-directional, the CCB treats it as a uni-directional bus, directed from the slaves to the master FPGA. The slave FPGAs use 16 bits of the 18-bit bus to send integrated or raw data to the master FPGA and 1 bit to send a heartbeat signal to the master FPGA. This leaves one bit currently unused.

The EPP parallel-port is used by the host computer to send configuration information and commands to the master FPGA, as well as to acknowledge interrupts that the master FPGA generates on the EPP-port's interrupt pin. The host computer can also optionally read back configuration values over the same link.

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 by the main FPGA clock-signal, `clock`, into input registers within the associated *Sampler* components. The configurable phase shift between the `adc_clock` and `clock` signals 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 presented at the `raw` outputs of the *Sampler* components, as well as being integrated.

Within the individual *Sampler* components, each new sample is integrated by adding it to one of 4 phase-switch bins. The appropriate phase-switch bin is specified by the master FPGA, via the `phase` control input. When the master FPGA commands the start of a new integration period, by asserting the `start` signal, the contents of the phase-switch bins

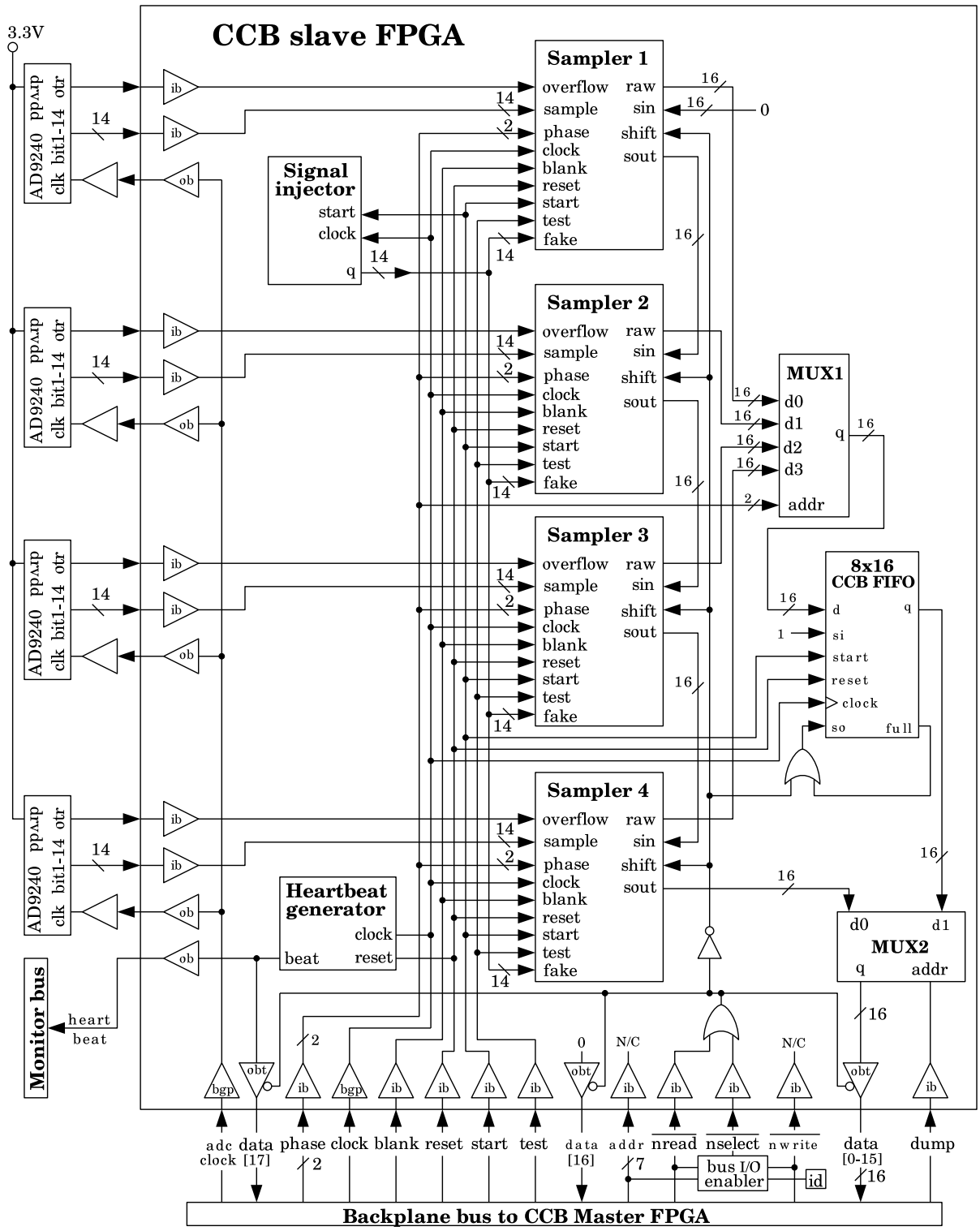

Figure 2.1: The top-level design of the slave FPGA

from the previous integration period, are transfered into I/O buffers, ready for transmission to the master FPGA. Simultaneously, within each *Sampler*, the bin that is selected by the `phase` signal, is initialized with the first ADC sample of the new integration period, while the remaining bins are zeroed.

The I/O 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 FPGA, 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.

In dump mode, when a pulse at the `start` input indicates the start of a new integration

period, any existing contents of the `CCB FIFO` component are replaced with the first raw sample of the integration period. The purpose of this FIFO is to allow the *Data Dispatcher*, in the master FPGA to take a few clock cycles to start reading out raw samples, without missing the corresponding number of samples at the start of the integration period. If the master FPGA takes more than 8 clock cycles to read the first sample from the FIFO, then the `full` output of the FIFO causes the first sample of the integration period to be shifted out, as though it had been read by the master FPGA, and thus frees up room for the latest raw sample to be shifted in. Thus the master FPGA has 8 clock cycles to start reading out dump-mode raw samples from the slave.

All input and output signals from the slave FPGA have to pass through buffers in the FPGA's I/O blocks. These buffers are shown in the diagram. Buffers marked `ib` are Xilinx `ibuf` input buffers, those marked `ob` are Xilinx `obuf` output buffers, those marked `bgp` are Xilinx `bufgp` global-clock-network input buffers, and those marked `obt` are Xilinx `obuft` tri-state output buffers. All of these buffers have been explicitly configured to accommodate the 3.3v low-voltage CMOS I/O standard. To maximize the number of outputs that can be simultaneously switching, without causing excessive ground-bounce, the output buffers have also been configured to use the lowest supported drive current, and the slowest supported slew time.

2.1.1 The Heartbeat Generator

The slave FPGAs generate a clock-like heartbeat signal that has two uses.

1. The external PC104 based monitoring system generates a leaky average of the `heartbeat` output signal, for monitoring by the computer. When the heartbeat signal is operating correctly, this average should be around half of the full-scale digital high voltage.

2. The heartbeat signal is also driven onto the master-slave data-bus, whenever the slave is selected, so that the master FPGA can determine if that slave is present and showing signs of life.

The circuit that generates the heartbeat signal is shown in figure 2.2. This generates a signal whose state alternates at the start of each FPGA clock cycle. It thus looks like a 5MHz clock signal, whose edges are synchronous with the main 10MHz clock signal.

When a particular slave FPGA is selected for readout, the master FPGA latches a copy of its `heartbeat` signal at the start of each clock cycle, and keeps the latched values from the two most recent successive clock cycles. Since the state of the heartbeat signal should alternate from one clock cycle to the next, the master FPGA then compares the two states with an XOR gate. If the two successive states aren't opposites, then the originating slave is flagged in the output data that are sent to the CCB computer.

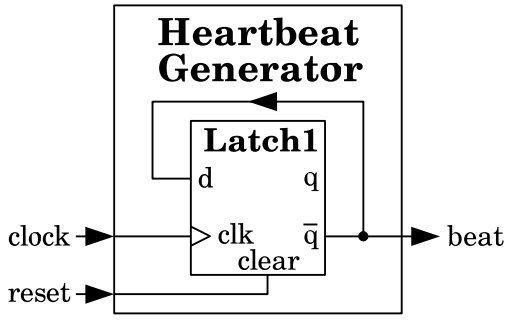

Figure 2.2: The Heartbeat Generator component

2.1.2 The Signal Injector

The job of the *Signal Injector* is to generate repeatable pseudo-random fake ADC samples, that can be used 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 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 circular 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 this would cause a break the otherwise repeatable test-sequence.

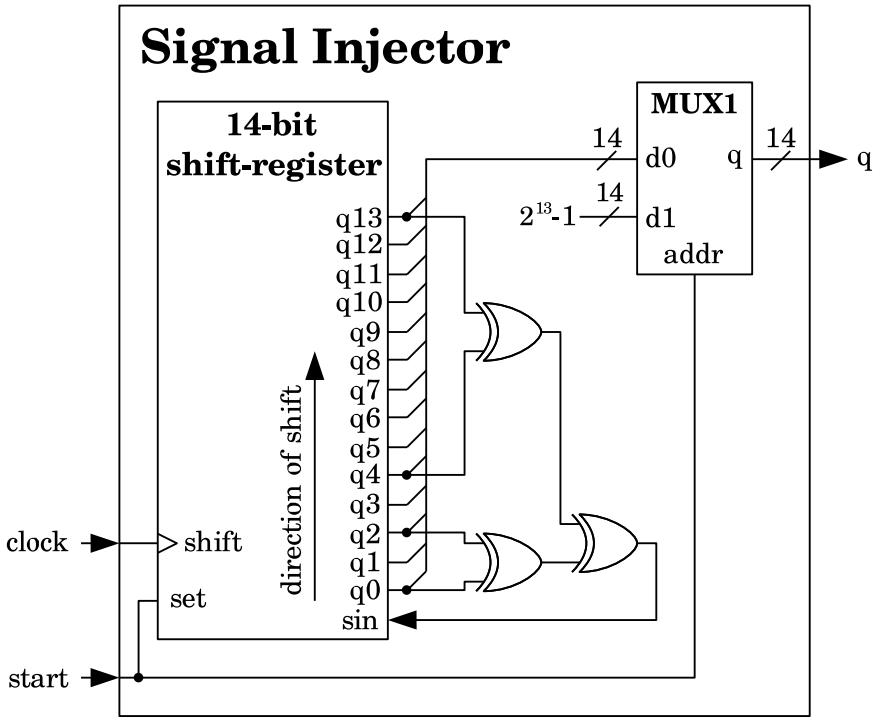

Figure 2.3: The Signal Injector component

2.1.3 The Sampler component

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

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 `Blanker1` module. `Blanker1` module 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 be added to one of its 4 internal accumulators (phase-switch bins), according to the states of the phase switches. 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.

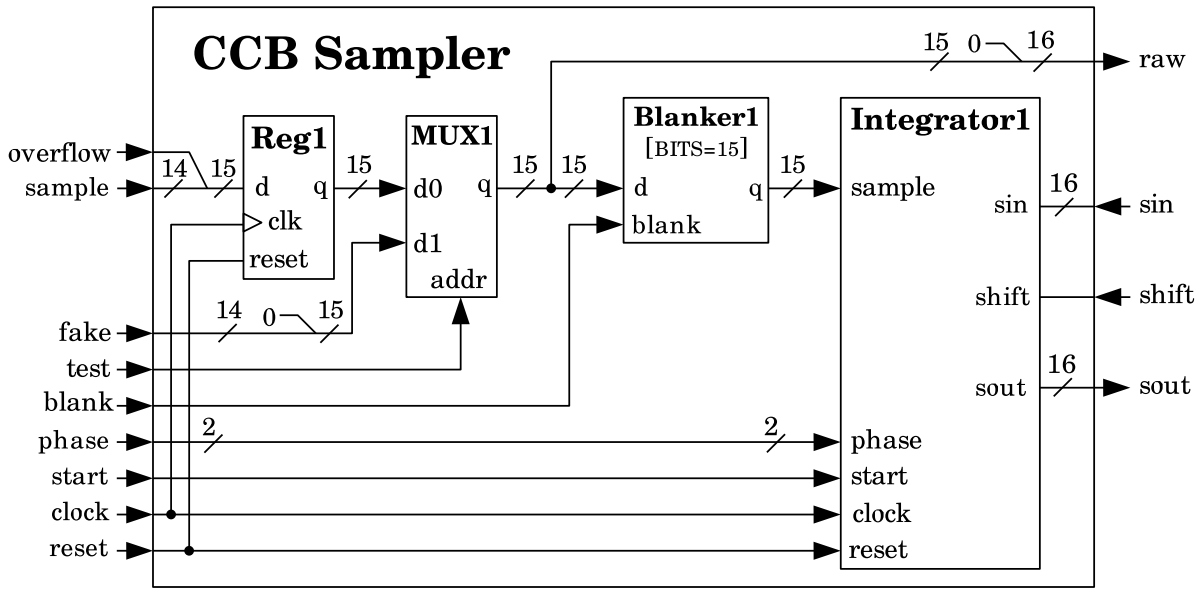

Figure 2.4: The Sampler component

The `start` input signal, which the master FPGA asserts for one clock cycle, indicates the end of one integration period, and the start of the next. When this is asserted, the contents of the integration bins are copied into a PISO within `Integrator1`, 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 output value of `Blanker1`, and zeroing the accumulators of the remaining 3 phase-switch bins.

Although the outputs of the accumulators are 32 bits wide, the data-bus that connects the slaves to the master FPGA is only 16 bits wide. Thus the PISOs are 16-bits wide, and each integrated sample is split into two parts before being loaded into this PISO, ordered such that the least significant 16-bits emerge from the `so` output, before the 16 most significant bits. The PISOs within neighboring *Sampler* components are chained via their `so` and `si` ports, and when being read-out, they are all simultaneously clocked via their `shift` inputs.

2.1.4 The Blanker component

Blanker components take a multi-bit input signal, `d`, and either present this unchanged at the `q` output, or, if the `blank` input is asserted, set all the bits of the `q` output to zero. They are trivially implemented by the VHDL code shown in figure 2.5.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity blanker is
  generic(BITS: std_logic_vector := 32);
  Port ( d : in std_logic_vector(BITS-1 downto 0);
         q : out std_logic_vector(BITS-1 downto 0);
         blank : in std_logic);
end blanker;

architecture Behavioral of blanker is
begin
  blank_bits: for i in BITS-1 downto 0 generate
    q(i) <= d(i) and not blank;
  end generate blank_bits;
end Behavioral;
```

Figure 2.5: The VHDL implementation of the Blanker component

2.1.5 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 2.6.

Most of the work of an *Integrator* component is performed by four embedded *Accumulator* components, each of which represents one of 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. At the start of each clock cycle, the decoded `phase` input thus determines which *Accumulator* gets the latest sample. 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.

2.1.6 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 2.7.

In the diagram, the combination of the `Adder` component and register `Reg1`, form the ac-

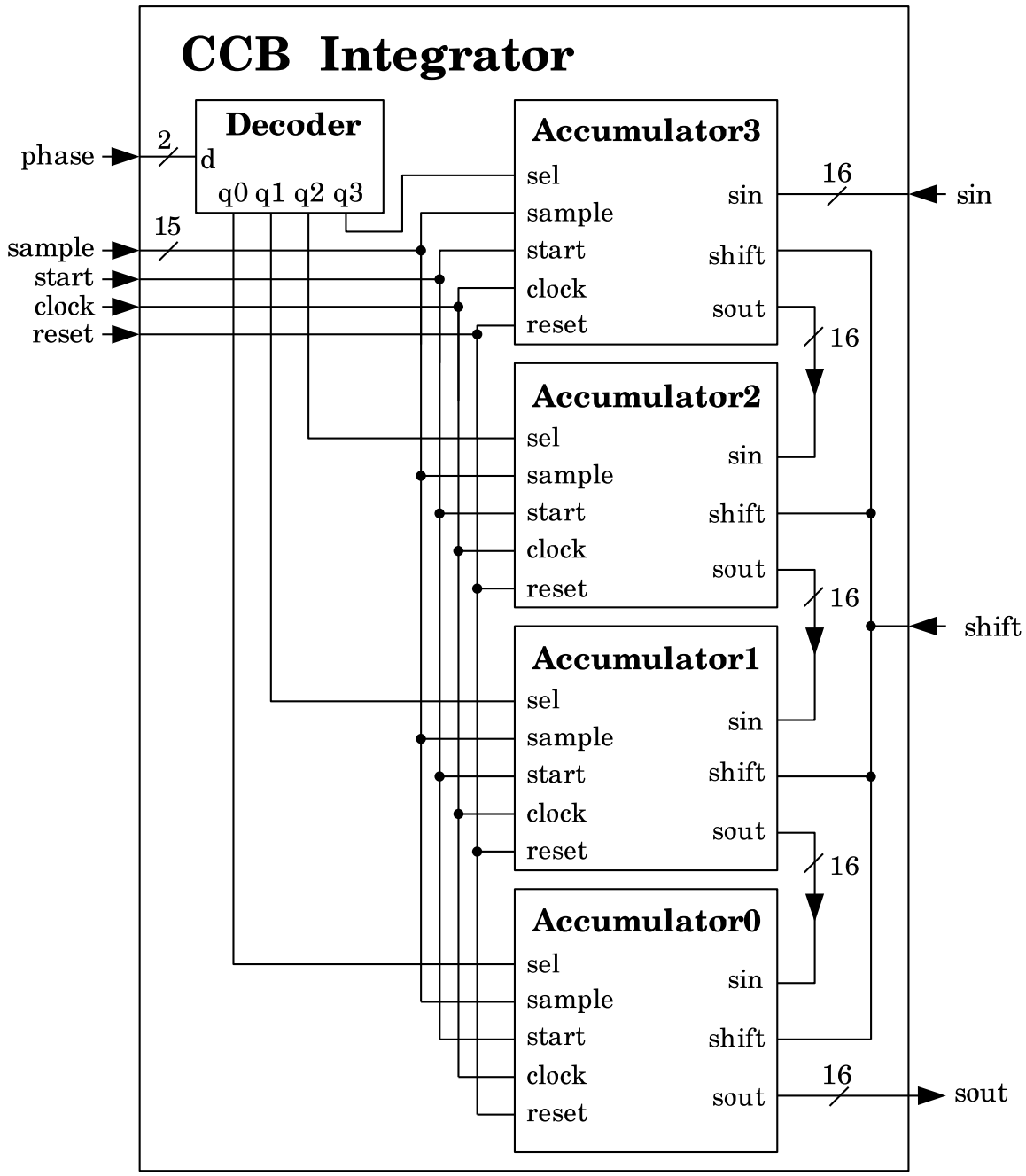

Figure 2.6: The Integrator component

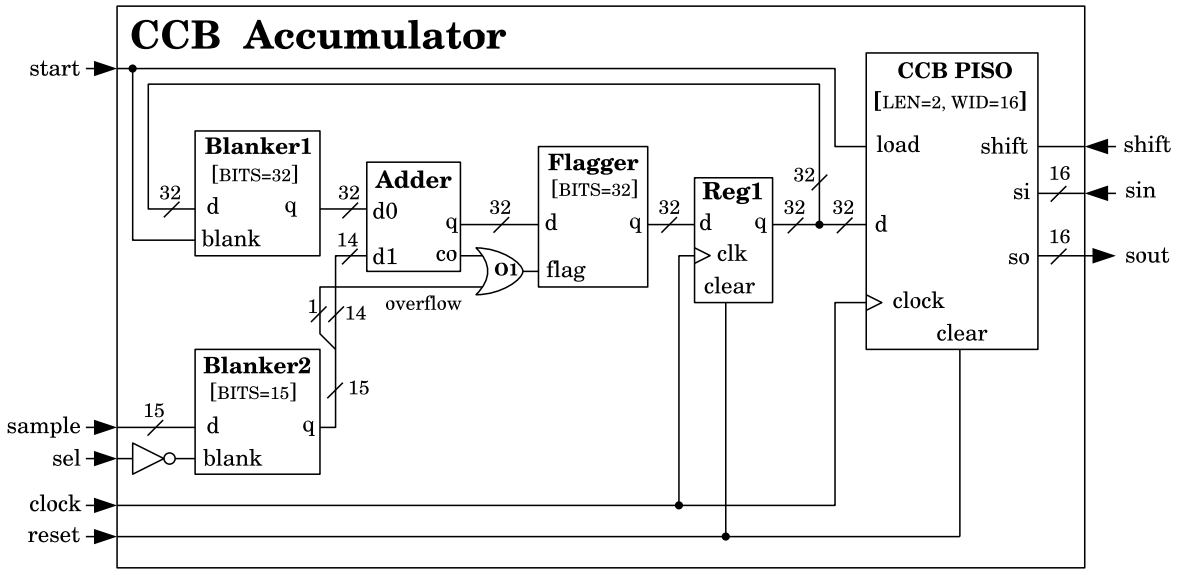

Figure 2.7: The Accumulator component

cumulator 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, when the input sample should be ignored, `Blanker2` arranges that zero be added, instead of a new sample.

At the start of a new integration period, as indicated by the `start` input being asserted for one clock cycle, `Blanker1`, 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 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 `Reg1`, 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 `Flagger` component initializes the accumulator 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 $2^{32} - 1$ by the `Flagger` component. 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 the `Flagger` component 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 `Blanker1`, 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 integration. The second time that it happens, the least significant 16 bits of the preceding *Accumulator* in the chain of *Accumulator* PISOs, is presented, etc.

The Flagger component

The *Flagger* component takes a multi-bit input signal, `d`, and either presents this unchanged at the `q` output, or, if the `blank` input is asserted, sets all the bits of the `q` output to one. Its is trivially implemented by the VHDL code shown in figure 2.8.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity flagger is
  generic(BITS: std_logic_vector := 32);
  Port ( d : in std_logic_vector(BITS-1 downto 0);
         q : out std_logic_vector(BITS-1 downto 0);
         flag : in std_logic);
end flagger;

architecture Behavioral of flagger is
begin
  flag_bits: for i in BITS-1 downto 0 generate
    q(i) <= d(i) or flag;
  end generate flag_bits;
end Behavioral;
```

Figure 2.8: The VHDL implementation of the Flagger component

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 determines the timing and states 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, and to the receiver. 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.

All input and output signals from the master FPGA have to pass through buffers in the FPGA I/O blocks. These buffers are shown in the diagram. Buffers marked `ib` are Xilinx `ibuf` input buffers, those marked `ob` are Xilinx `obuf` output buffers, those marked `ibg` are Xilinx `ibufg` global-clock-pin input buffers, and those marked `iob` are Xilinx `iobuf` tri-state bi-directional buffers. All of these buffers have been configured to accommodate the 3.3v low-voltage CMOS I/O standard.

When many output pins of an FPGA simultaneously go high or low, the resulting slew currents in the ground pins can cause the effective ground level within the FPGA to significantly rise or fall. This causes otherwise constant voltage levels at input pins to appear to change, and if these changes are sufficiently large, this can generate phantom pulses at asynchronous inputs. To reduce ground-bounce, 21 pins, spread around the FPGA, but marked collectively as `vg` (virtual ground) in the diagram, are driven low internally, and tied to ground externally. These effectively increase the number of ground-return pins. To facilitate measurements of the ground levels in each of the I/O banks, an additional pin per I/O bank is driven low internally, but left floating externally. These pins are collectively denoted in the diagram by the name `qp`, which stands for "*quiet pins*".

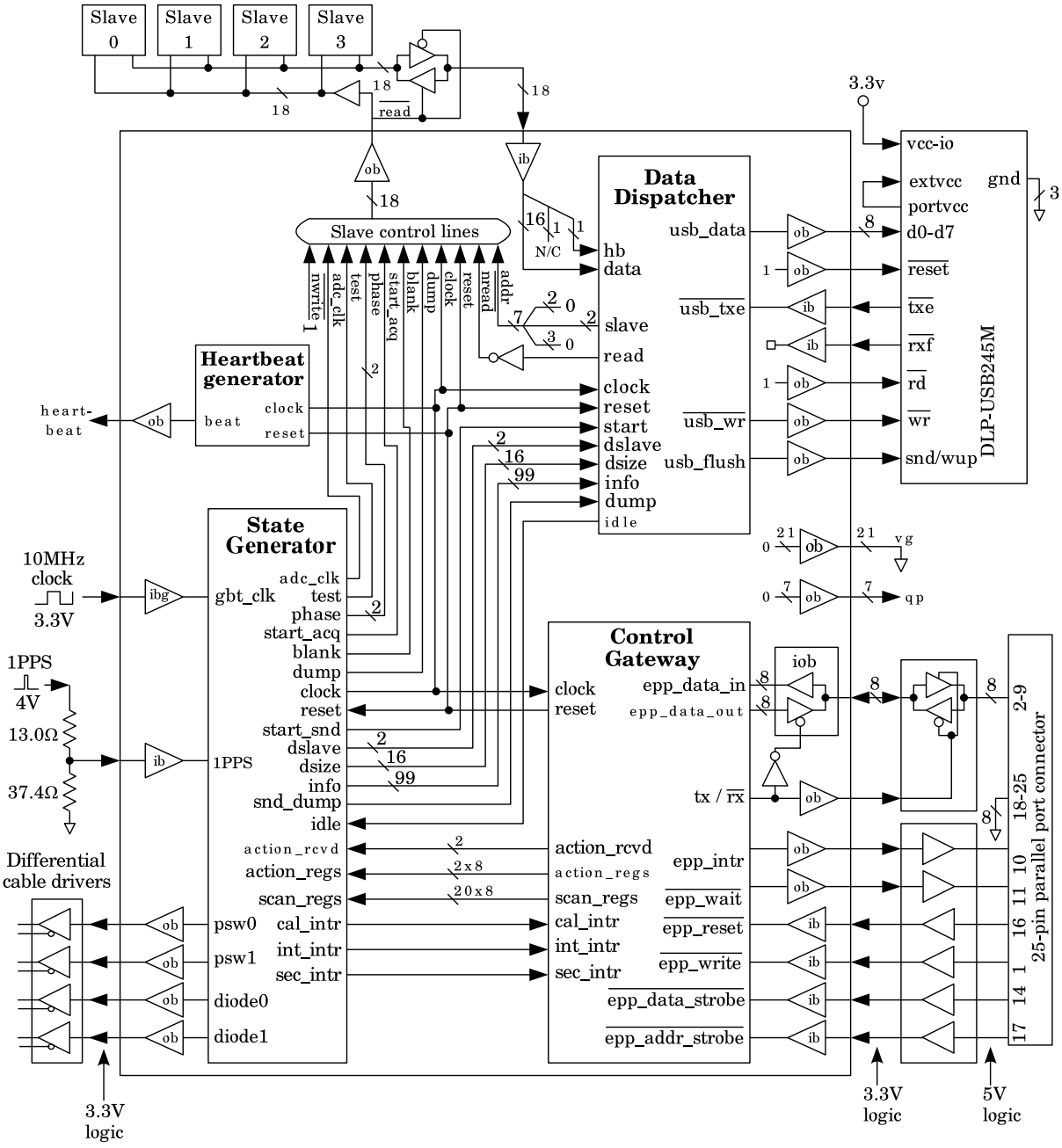

Figure 3.1: The top-level design of the master FPGA

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 is simplified by the fact that EPP-enabled parallel ports can generate 8-bit address and data I/O cycles in hardware. The 4 types of bus cycles are interpreted by the CCB firmware as follows:

- **The address-write cycle**

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

- **The data-write cycle**

  The byte that this sends to the FPGA, is copied into the register that was last selected by an address-write cycle.

- **The data-read cycle**

  When the FPGA is asked for a data-byte, it sends the contents of the register that was last selected by an address-write cycle.

- **The address-read cycle**

  When the FPGA is asked for an address-byte, it sends a byte whose individual bits indicate which FPGA event-sources have requested interrupts since the last time that the computer executed an address-read cycle. This also has the side effect of acknowledging any previously unacknowledged interrupt.

There are two occasions that the computer writes data to the master FPGA registers.

1. To start a new scan, the computer first changes the configuration for the new scan, by writing to the scan-configuration registers, then sends a start-scan command, by writing to the `start_scan_reg` register. This tells the master FPGA to stop the current scan as soon as possible and start the new scan.

   When the start-scan command is received, the *State Generator* takes a snapshot of the scan configuration registers, and subsequently uses this snapshot to configure the new

scan. This ensures that the computer can write to the scan configuration registers in any order, and at any time, in the secure knowledge that only the values that pertain when the start-scan command is sent, will ever be used. The snapshot configuration is used to configure the scan during the short period between the end of the previous scan and the start of the new scan. This ensures that between the moment when the start-scan command is received, and the moment when the previous scan actually ends, the operation of the previous scan isn't affected by the new configuration. This is important, because the previous scan doesn't end until any pending data, from that scan, have been safely sent to the computer.

2. Since the on/off states of the calibration diodes potentially change at the start of each new integration period, and the new states need to be known in advance of each integration period, the master FPGA uses a FIFO to hold a time-ordered in-advance list of bytes, whose values specify successive cal-diode states and their durations. To initially fill this FIFO, and thereafter keep it full, the computer writes one such cal-diode configuration byte to the master FPGA, whenever it receives a `cal_intr` interrupt from the master FPGA.

   The master FPGA generates the first `cal_intr` interrupt of a new scan, as soon as the corresponding start-scan command is received. Once the computer has responded to this interrupt, by sending the cal-diode configuration of the first integration period, or periods, then the master FPGA generates a new `cal_intr` interrupt, to request the cal-diode configuration that should follow the first. It continues to request cal-diode configuration bytes until the FIFO is full. Thereafter, a new `cal_intr` interrupt is sent whenever space becomes available in the FIFO. Since individual cal-diode configuration-bytes last for one or more integration periods, this will happen at most once per integration period.

3.1.1 The internals of the Control Gateway

The implementation of the *Control Gateway* is shown in figure 3.2.

Since only one 8-bit register can be read from or written to by the CPU in a single EPP transaction, it is necessary to send the target address for subsequent read and write operations, as a separate EPP transaction. As previously mentioned, to do this, the CPU uses an EPP address-write transaction to send the 8-bit address of the target register. 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 event-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 event-sources requested the interrupt. The *EPP Interrupter*, which is told about the

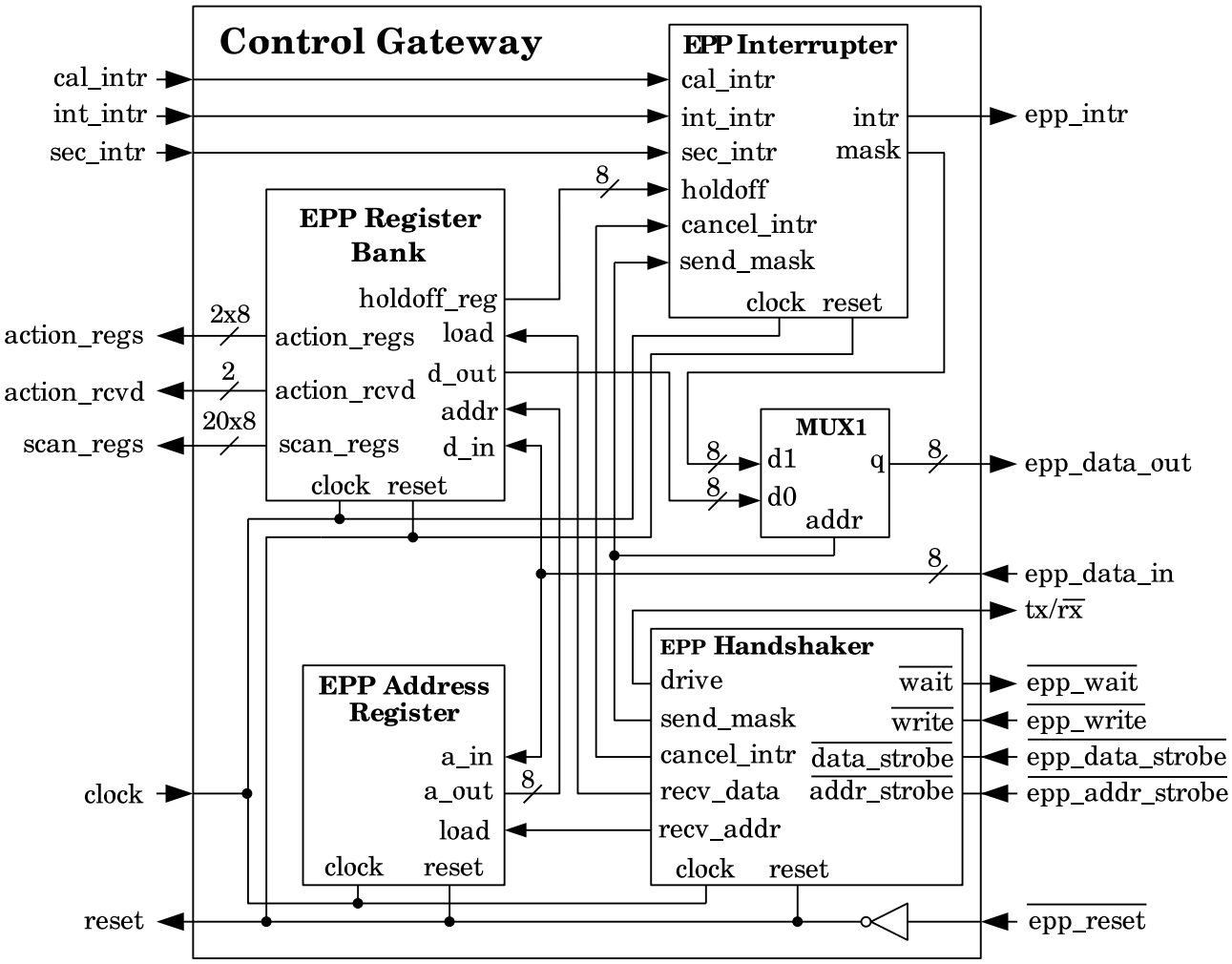

Figure 3.2: The Control Gateway

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 determines the minimum number of clock cycles to wait between sending one interrupt, before sending another. This prevents interrupts from being sent too frequently for the CPU to handle, and also sets the rate at which unacknowledged interrupts are to be re-sent. If the computer fails to acknowledge receipt of an interrupt within one holdoff interval, a new interrupt pulse is generated. There is no danger that such a re-sent interrupt will be interpreted by the CPU as indicating a second distinct event in the FPGA, since it is the contents of the interrupt mask, rather than the number of interrupts received, that indicates when an event has occurred, and the interrupt mask is automatically cleared as part of the read-address operation.

To avoid a tug-of-war with the CPU, the FPGA only drives the EPP `data` lines when explicitly requested. Thus the tri-state output buffers in the I/O-blocks of the `data` 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.

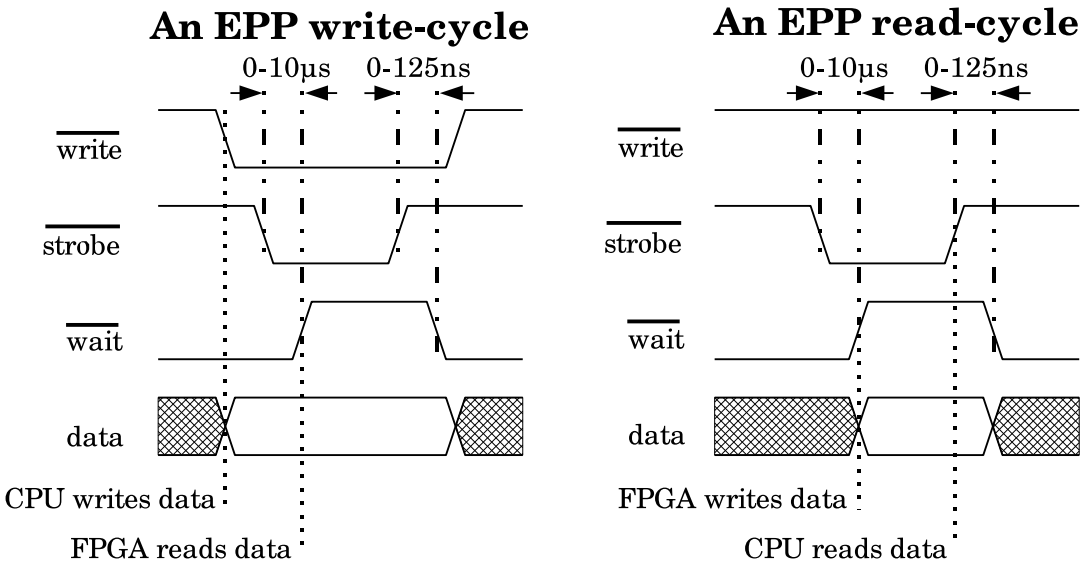

Figure 3.3: The standard EPP I/O cycles

The standard timings of the read and write EPP I/O cycles are shown in figure 3.3. Note

that the $\overline{\texttt{strobe}}$ signal in this diagram 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 the computer requests an EPP read-cycle.

When looking at this circuit it is important to note that in order to safely convert an external asynchronous input signal into a synchronous internal signal, it is widely recommended that one use a chain of two latches, to synchronize the signal, instead of just one. The reason is that occasionally the external input signal will violate the setup and hold times of the input latch, and place the input latch in a metastable state. The use of two latches, gives this state an extra clock cycle to resolve itself, before the rest of the circuit sees the synchronized signal.

According to the IEEE-1284 EPP standard, the $\overline{\texttt{wait}}$ signal needs to go high either when data written by the CPU have been latched by the peripheral, in the case of a write-cycle, or once data placed on the data lines by the peripheral data have stabilized, in the case of a read-cycle. Due to the above requirement that one use two latches to synchronize an asynchronous input signal, and the fact that data are latched in the FPGA to and from the data lines synchronously with the FPGA clock signal, the minimum delay that we can insert before driving the $\overline{\texttt{wait}}$ signal high, is up to one FPGA clock cycle between the time that the strobe goes low and the next rising edge of the FPGA clock, plus one extra clock cycle for the synchronization overhead added by the necessary second latch. This delay is thus between 100ns and 200ns, which is well within the maximum of 10μs dictated by the IEEE-1284 standard. This minimum delay is what is implemented by the *EPP Handshaker*. The $\overline{\texttt{wait}}$ signal is driven high by the outputs of the rightmost of each pair of latches in the diagram, between one and two FPGA clock cycles after either of the EPP strobe lines goes low. At the corresponding rising edge of the FPGA clock, data are either in the process of being latched by the FPGA, in the case of an EPP write cycle, or are guaranteed to have stable register or interrupt data on them, ready to be read by the CPU, in the case of an EPP read cycle. The implementation of these guarantees will be described shortly.

The $\overline{\texttt{wait}}$ signal must return low within 125ns of the active strobe signal being returned high by the CPU. This can not be done using synchronous logic, due to the necessary delay of up to two 100ns clock cycles to resolve the potential metastable states caused by the asynchronous strobe signals. Thus the timing of the falling edge of the $\overline{\texttt{wait}}$ signal is determined by the asynchronous logic formed by gates `N1` and `A5`, which pull the $\overline{\texttt{wait}}$ signal low as soon as the strobe signal goes low. According to the FPGA's data-sheet, the resulting I/O delays should be less than 10ns, which is clearly well within the maximum of 125ns.

The EPP data lines are handled differently for each of the 4 possible EPP I/O cycles.

- **The EPP address-read cycle**

  Address-read cycles are used by the CCB host to both acknowledge and receive informa-

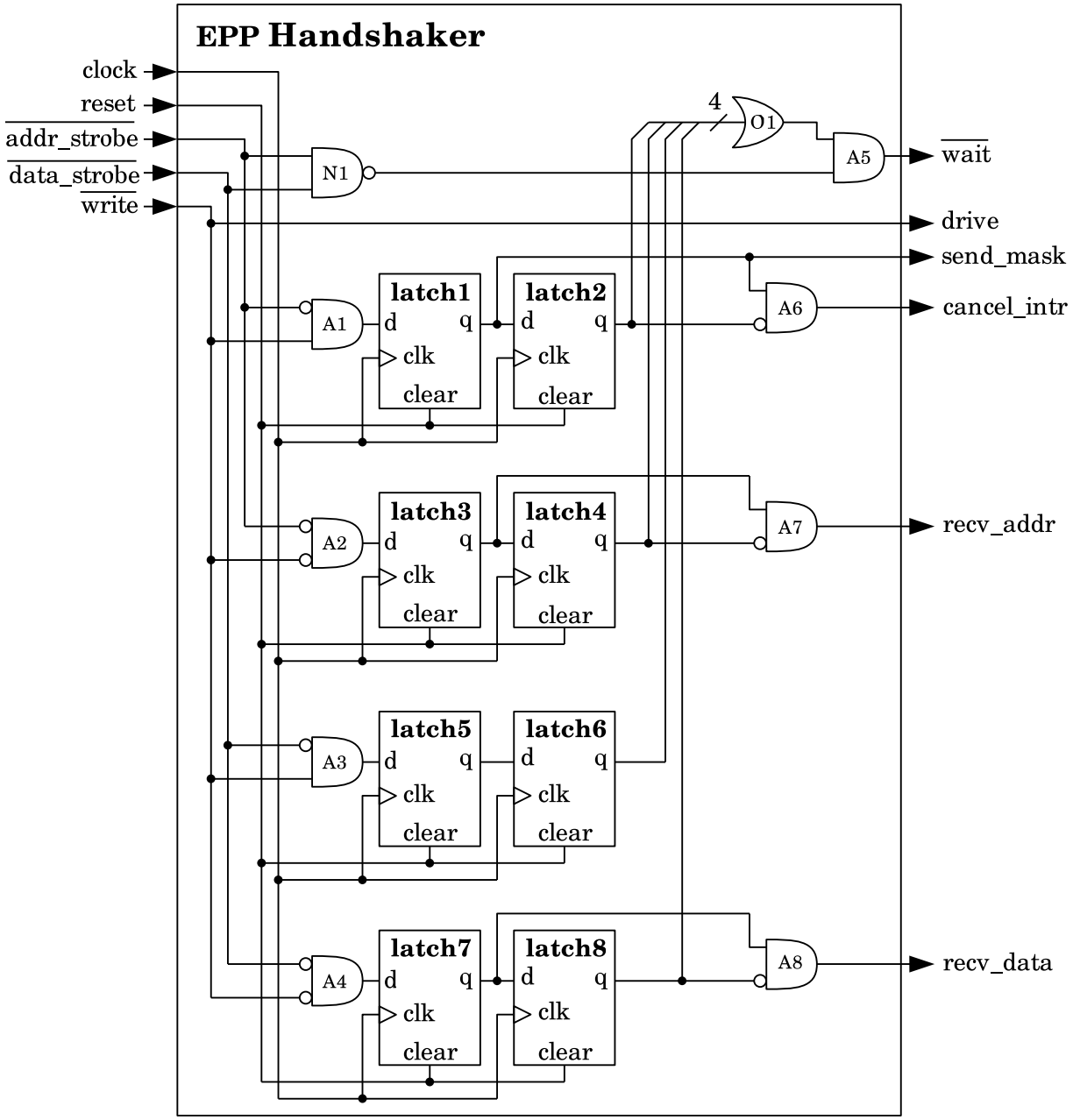

Figure 3.4: The EPP Handshaker

tion about which interrupt-requesting events have occurred since the last address-read. The parallel port initiates such cycles, by pulling the $\overline{\texttt{addr\_strobe}}$ signal low, while holding the $\overline{\texttt{write}}$ signal high. This causes the input of `latch1` to go high, such that on the next rising edge of the FPGA clock, its `q` output starts to go high, potentially slowed by metastability problems. This output does three things.

1. It drives the `send_mask` signal high. This deasserts the clock-enable input of the output latch of the *EPP interrupter*, and thus prevents that latch output from changing its value at the next rising edge of the clock (by which time any metastable state should have resolved itself). This output signal remains asserted, until one clock cycle after the strobe goes low again, and thus ensures that the interrupt mask that is driven onto the data lines remains stable until the CPU has read it.
2. It also drives the `cancel_intr` signal, which drives the clock-enable input of a different latch in the *EPP interrupter* to clear the interrupt condition that is being reported. Again, this isn't seen until the next rising edge of the clock, such that any metastable state has time to resolve itself.
3. Finally, the output of `latch1` also drives the input of `latch2`, whose output both drives the $\overline{\texttt{wait}}$ signal high, and deasserts the temporarily raised `cancel_intr` signal.

Thus, the `send_mask` output becomes asserted within one clock cycle of $\overline{\texttt{addr\_strobe}}$ going low, and subsequently becomes deasserted within one clock cycle of $\overline{\texttt{addr\_strobe}}$ returning high.

One clock cycle after the `send_mask` output is asserted, the $\overline{\texttt{wait}}$ signal is driven high, to tell the CPU that the data-lines are presenting stable data to be read.

Just like the `send_mask` output, the `cancel_intr` output becomes asserted within one clock cycle of $\overline{\texttt{addr\_strobe}}$ going low, but unlike the `send_mask` output, it then remains asserted for precisely one clock cycle, regardless of when $\overline{\texttt{addr\_strobe}}$ returns high.

Throughout the cycle, the $\overline{\texttt{write}}$ signal is also routed directly to the `drive` output, to enable the transmit buffers to drive the data onto the EPP data lines.

- **The EPP address-write cycle**

  EPP address-write cycles are used by the CCB to send the address of the CCB register that will next be read from or written to. They are initiated by the CPU by pulling both the $\overline{\texttt{addr\_strobe}}$ and $\overline{\texttt{write}}$ lines low. This causes the input of `latch3` to go high, and on the following rising edge of the FPGA clock, the output of `latch3` starts to go high, possibly delayed by any metastability problems. This, in turn does two things.

  1. It drives the `recv_addr` signal high. This drives the clock-enable input of the *EPP Address Register*, such that one clock cycle later this register latches the contents of the data lines.

2. It drives the input of `latch4`, whose output both drives the $\overline{\texttt{wait}}$ signal high, and deasserts the `recv_addr` signal.

Thus the `recv_addr` signal is asserted for one clock cycle, starting up to one clock cycle after the $\overline{\texttt{write}}$ and $\overline{\texttt{addr\_strobe}}$ signals go low, and the $\overline{\texttt{wait}}$ signal goes high one clock cycle later.

- **The EPP data-read cycle**

  EPP data-read cycles are used by the CCB to read-back the value of the currently addressed CCB register. The CPU initiates such a cycle by pulling the $\overline{\texttt{data\_strobe}}$ line low, while the $\overline{\texttt{write}}$ signal is high. In this case, the circuit doesn't need to tell the rest of the *Control Gateway* about the transaction, because the default value of the data-bus output of the *Control Gateway* is the value of the currently addressed register, and since register values can only change during EPP write transactions, there is no need to explicitly freeze this register during a read. Thus, latches 5 and 6 simply respond by driving the $\overline{\texttt{wait}}$ signal high between one and two clock cycles after the $\overline{\texttt{data\_strobe}}$ signal goes low and the $\overline{\texttt{write}}$ signal is high, at which time the CPU can safely read the register value from the data lines. The output is driven onto the EPP data-bus by way of the tri-state output buffers that are enabled by the $\overline{\texttt{write}}$ signal, routed to them via the `drive` output.

- **The EPP data-write cycle**

  EPP data-write cycles are used by the CCB to write values into the currently addressed CCB register. They are initiated by the CPU by driving both the $\overline{\texttt{data\_strobe}}$ and $\overline{\texttt{write}}$ signals low. This has the same effect on the `recv_data` output as previously described for the address-write cycle. In particular, the `recv_data` signal is asserted for one clock cycle, at the end of which the currently addressed register latches the contents of the EPP data lines, just as the $\overline{\texttt{wait}}$ signal is raised to tell the CPU that the data have been received.

The response of the *EPP Handshaker* to the 4 I/O cycles is illustrated in the timing diagrams of figure 3.5.

The EPP address register

The *EPP Address Register*, as shown in figure 3.6, 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 enable input is not asserted, so the register retains its current value. However, when the `load` input indicates 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.

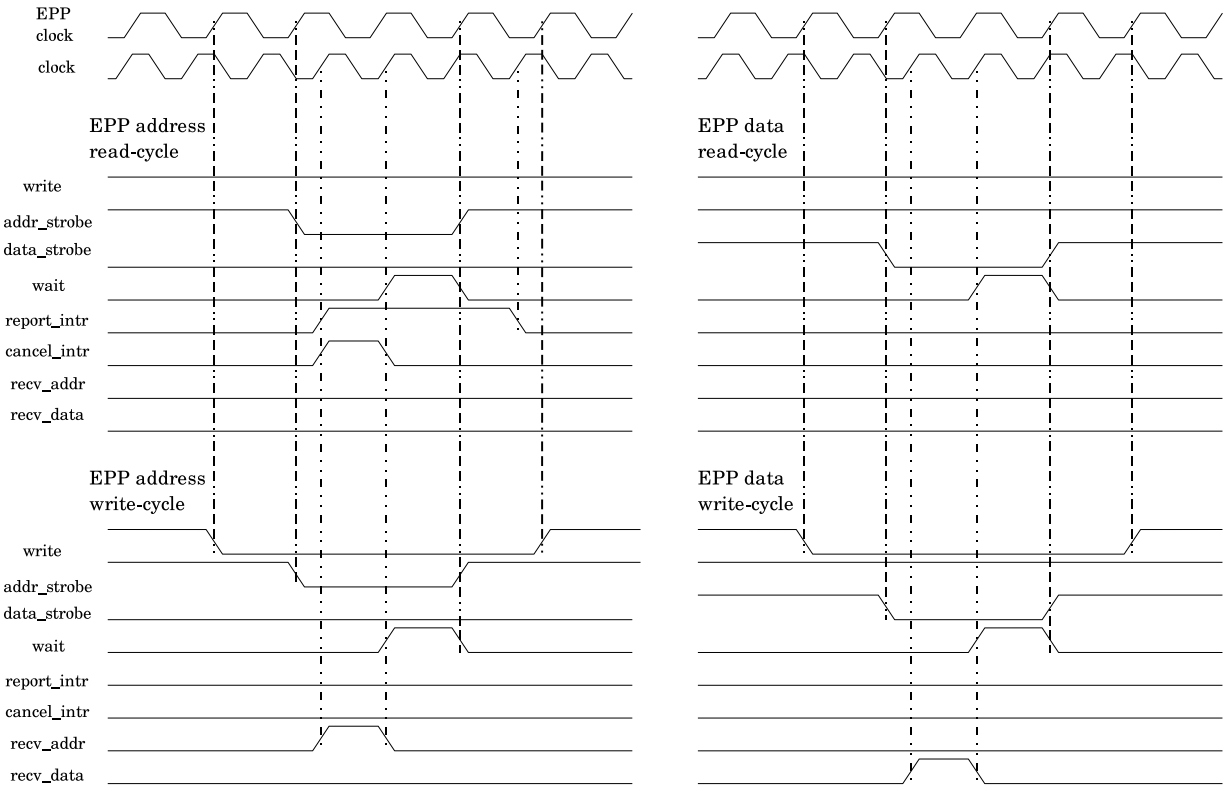

Figure 3.5: Timing diagrams of the EPP Handshaker

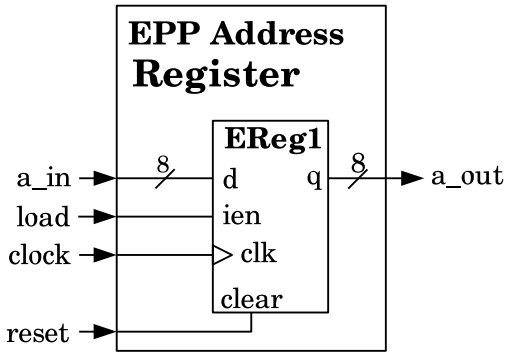

Figure 3.6: The EPP Address Register

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*, whose VHDL implementation is shown in figure 3.7, contains the registers that are used to record and provide read-back of configuration parameters and commands sent by the CPU. It also supplies a read-only CCB identification byte in EPP register 0. 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 *EPP Register Bank* holds 4 distinct groups of registers.

1. Register zero is an identification register. This has the arbitrary value of `27`. As a basic sanity check, when the device driver on the computer starts running, it attempts to read this register, and verify its value. Note that if the computer attempts to write to this register, the new value will be ignored, and the register will retain its special ID value.

2. The interrupt holdoff delay is held in register 1. This is a normal read-write register, but because its value is used locally, within the *Control Gateway*, it's value is separately output to the *Control Gateway*, via the `holdoff_reg` argument.

3. Action registers are register which are written to, to solicit an immediate reaction within the *State Generator*. Examples are the `start_scan_reg` register, which commands the start of a new scan, when written to, and the `cal_diode_reg` register, which adds a new value to the queue of calibration configurations, when written to. Whenever one of these registers is written to by the computer, the corresponding bit in the `action_rcvd` output signal, is asserted for one clock cycle, to tell the *State Generator* to perform the associated action. Note that this ensures that the *State Generator* notices the write operation, even if the value of the register stays the same.

   Action registers, and their receipt-notification signals, are passed to the *State Generator* via the `action_regs` and `action_rcvd` signals.

4. Scan configuration registers hold configuration values for the next scan. Changes to their values are not noticed by the *State Generator* until the next time that the computer writes to the `start_scan_reg` register. Thus the configuration of the next scan can be sent before the previous scan has ended, without affecting it, and multi-byte registers can be written, one byte at a time, without any danger of a partially changed configuration value being unexpectedly used.

   Scan configuration registers are presented to the *State Generator* via the `scan_regs` output.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity epp_register_bank is
    generic (NID: integer := 1;     -- The number of identification registers.
             NLOCAL: integer := 1;  -- The number of control-gateway registers.
             NACTION: integer := 2; -- The number of action registerss.
             NSCAN: integer := 20;  -- The number of scan-config registers.
             WID: integer := 8);    -- The number of bits per register.
    Port (clock, reset : in std_logic;
          load : in std_logic;                                        -- When high, assign d_in to the addressed register.
          d_in : in std_logic_vector(WID-1 downto 0);                 -- The byte to assign when load is asserted.
          addr : in std_logic_vector(WID-1 downto 0);                 -- The address of the target register.
          d_out : out std_logic_vector(WID-1 downto 0);               -- The current value of the addressed register.
          holdoff_reg : out std_logic_vector(WID-1 downto 0);         -- The value of the interrupt-holdoff register.
          action_regs : out std_logic_vector((WID*NACTION)-1 downto 0); -- The values of the action registers.
          scan_regs : out std_logic_vector((WID*NSCAN)-1 downto 0);   -- The values of the scan-configuration registers.
          action_rcvd : out std_logic_vector(NACTION-1 downto 0));    -- The receipt-notification signals of the action registers.
end epp_register_bank;

architecture Behavioral of epp_register_bank is
  constant NREG : integer := NID + NLOCAL + NACTION + NSCAN;             -- Total number of registers.
  constant ID_REG_ADDR : integer := 0;                                   -- The address of the CCB identification register.
  constant HOLDOFF_REG_ADDR : integer := 1;                              -- The address of the interrupt-holdoff register.
  constant CCB_ID_VALUE : std_logic_vector(WID-1 downto 0) := "00011011"; -- The value of the CCB identification register.
  constant REG_ZERO : std_logic_vector(7 downto 0) := (others => '0');   -- The zero-valued byte, used to initialize registers.
  --
  -- Create the array of registers.
  --
  type regarray is array (NREG-1 downto 0) of std_logic_vector(WID-1 downto 0);
  signal regbank : regarray;
  --
  -- Create an internal array of per-register received signals.
  --
  signal regrcvd : std_logic_vector(NREG-1 downto 0);
  --
  -- An integer version of the addr input.
  --
  signal reg_addr : integer range 0 to NREG;
begin
  --
  -- Convert the addr argument to an integer, for use as an array index.
  --
  reg_addr <= conv_integer(addr) when conv_integer(addr) < NREG else 0;
  --
  -- Export the value of the currently addressed register at d_out.
  --
  d_out <= regbank(reg_addr);
  --
  -- Export the value of the holdoff register at the holdoff output.
  --
  holdoff_reg <= regbank(HOLDOFF_REG_ADDR);
  --
  -- Export the values of the action registers and their notification signals.
  --
  EXPORT_ACTION_REGS: for i in 0 to NACTION-1 generate
    action_regs(((i+1)*WID)-1 downto i*WID) <= regbank(NID+NLOCAL+i);
    action_rcvd(i) <= regrcvd(NID+NLOCAL+i);
  end generate EXPORT_ACTION_REGS;
  --
  -- Export the values of the scan configuration registers.
  --
  EXPORT_SCAN_REGS: for i in 0 to NSCAN-1 generate
    scan_regs(((i+1)*WID)-1 downto i*WID) <= regbank(NID+NLOCAL+NACTION+i);
  end generate EXPORT_SCAN_REGS;

  write_reg_proc: process (clock, reset)    -- Perform register assignments.
  begin
    if reset = '1' then                     -- Active-high asynchronous reset.
      regbank(ID_REG_ADDR) <= CCB_ID_VALUE;
      regbank(NREG-1 downto 1) <= (others => REG_ZERO);
    elsif clock'event and clock = '1' then  -- Rising clock edge
      if load='1' and reg_addr /= ID_REG_ADDR then
        regbank(reg_addr) <= d_in;       -- Assign a new register value.
        regrcvd(reg_addr) <= '1';        -- Flag the register as updated.
      else
        regrcvd <= (others => '0');      -- Deassert any register update flag.
      end if;
    end if;
  end process write_reg_proc;
end Behavioral;
```

Figure 3.7: The VHDL implementation of the Register Bank component

To synchronously write a new value into the currently addressed register, the new value is first presented at the `d_in` input, then the `load` input is asserted for one clock cycle.

The list of defined registers can be found in appendix A.

The EPP Interrupter

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

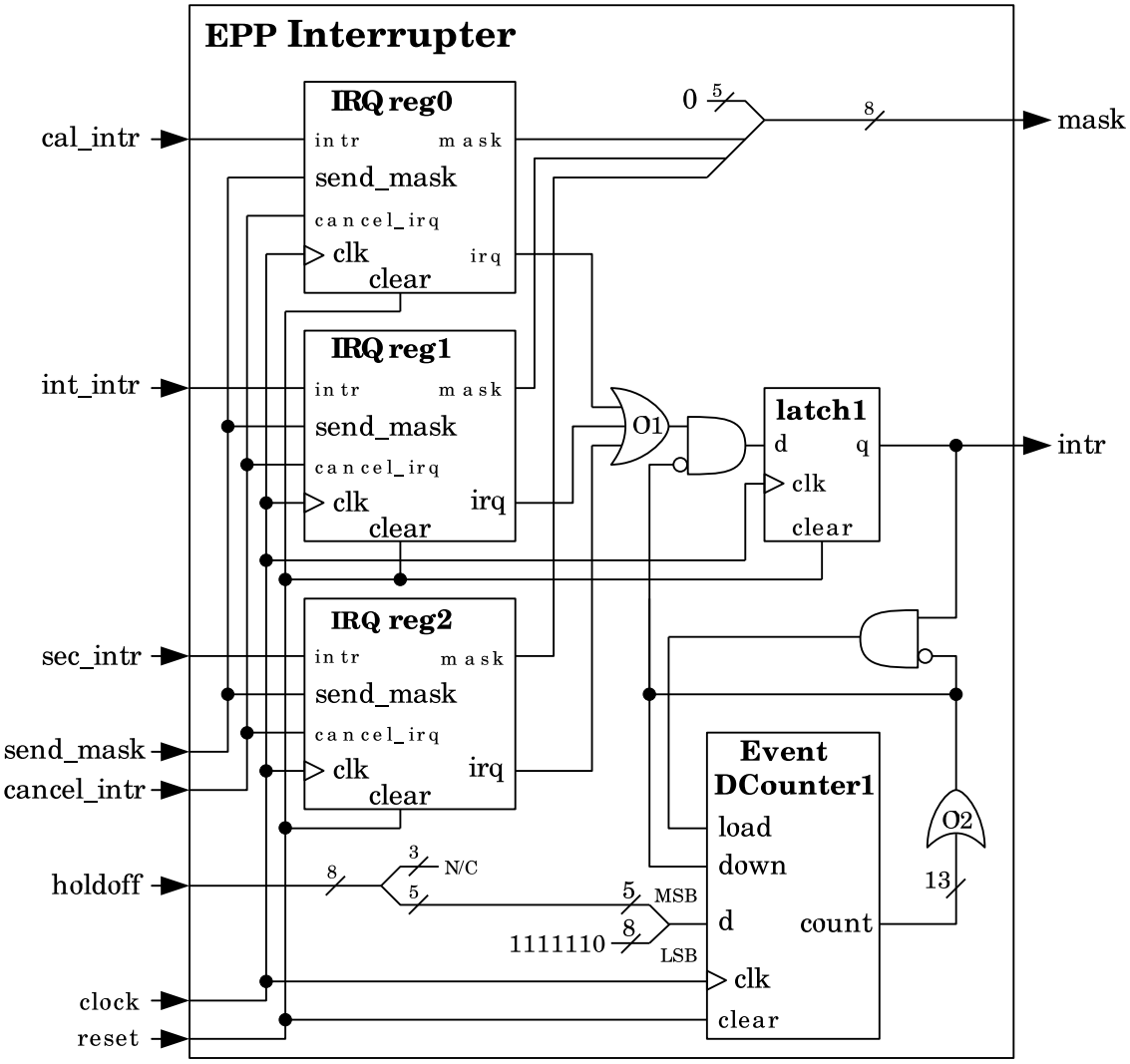

Figure 3.8: The EPP Interrupter module

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 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 $256 \times (\texttt{holdoff} + 1)$ clock cycles. In particular, once any interrupt source has requested an interrupt, a new CPU interrupt is sent every time this number of clock cycles have passed, until the computer performs an EPP address-read to get the bit-mask of previously unreported events. Hardwiring a minimum value of 256 clock cycles, ensures that the computer doesn't get swamped with interrupts if the `holdoff` input is set to a small value. The holdoff countdown is implemented by `Down Counter1`. This is a down-counter with a synchronous load capability, and a count-enable input (`ce`) which, when asserted, tells the counter to count down by one at each rising edge of the clock. Figure 3.9 is a timing diagram that illustrates the behavior of the holdoff counter. It shows the case where an interrupt request is pending, and the `holdoff` input happens to be zero. It can be seen that the diagram would repeat every 256 clock cycles in this case, and that an interrupt of two clock cycles would be raised anew, each time around.

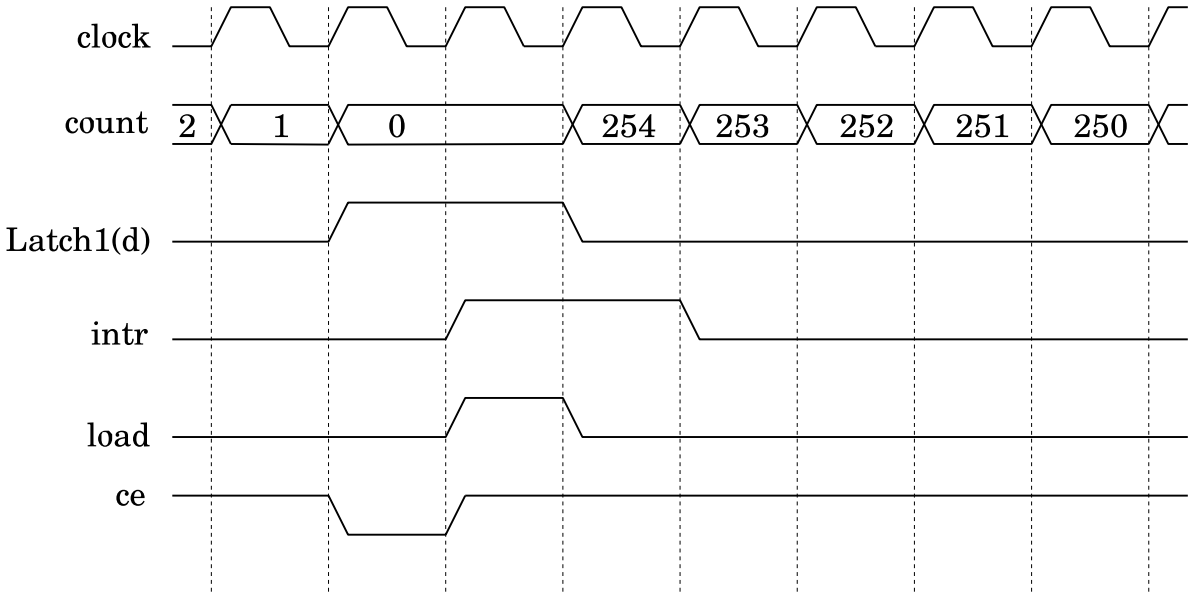

Figure 3.9: A timing diagram of the interrupt holdoff counter

When a particular event-source in the FPGA wishes to notify the computer of a new event, it synchronously asserts the respective 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, then 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.

Figure 3.10, shows the internals of a single IRQ register.

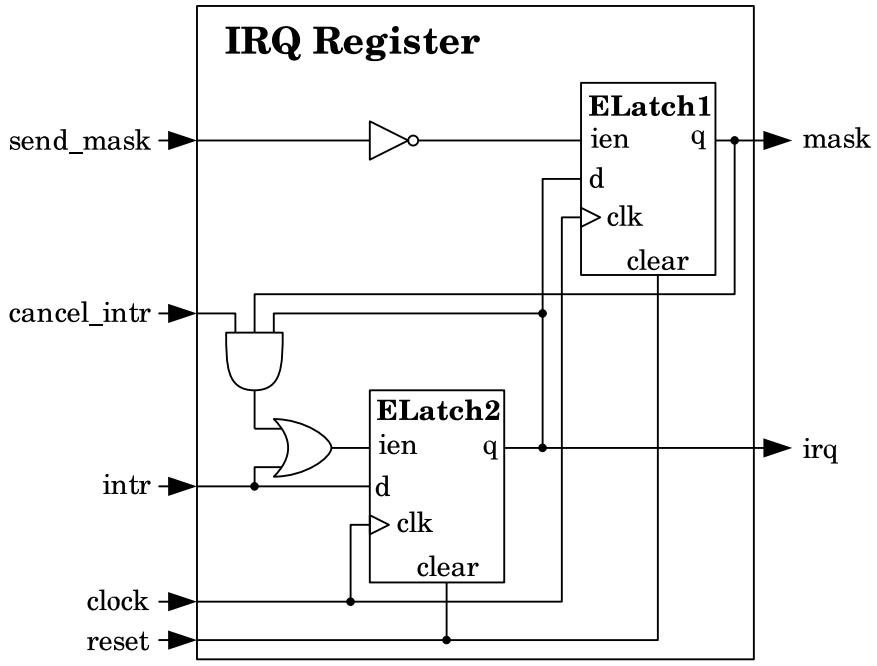

Figure 3.10: An Interrupt Request (IRQ) Register

When the CPU acknowledges the receipt of an interrupt, by performing an EPP address-read, the *EPP Handshaker* asserts the `send_mask` input for the duration of the time that the `mask` output is required to be frozen for reading by the CPU, and it asserts the `cancel_intr` input for one clock cycle, to tell the IRQ registers to synchronously forget the sub-set of interrupts that are being reported in the `mask` output. The register that drive the `mask` outputs of the individual IRQ registers normally update at the start of each FPGA clock cycle, latching copies of the neighboring `irq` outputs. The asserted `send_mask` input, however, prevents

these updates, by disabling the clock-enable inputs of the `mask` output registers. Thus, one clock cycle after `send_mask` input goes high, when the CPU is first told that it is safe to read the EPP data lines, the `mask` outputs, along with the EPP data lines that they drive, are guaranteed to have been stable for one clock cycle. Meanwhile, on the clock cycle at which the `cancel_intr` input is seen to be asserted, the IRQ registers look at the frozen `mask` outputs, to see if there is a reported interrupt to be canceled, and if so replace the contents of the registers that drive the `irq` outputs, with the current value of the `intr` input. If an interrupt is being requested at that point, this again drives the `irq` output high, to request a new EPP interrupt. Otherwise, it drives it low, and thus stops that IRQ register from generating EPP interrupts, until the corresponding event-source next asserts its `intr` input. The aim of this is to guarantee that if any interrupt-worthy event occurs one or more times between two CPU reads of the interrupt mask, the second read will report it to the CPU.

Note that if an event-source requests a new interrupt while its IRQ register is still asserted from a previously unacknowledged request, the new request does nothing. Thus, each bit in the bit-mask of interrupts that are sent to the CPU, only says that one or more of the corresponding event has occurred since the mask was last read. There is no provision for keeping track of how many times the interrupt occurred. A way to keep track of the number of this 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 the design of the IRQ registers did just this. 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 by the CPU, the CCB would quickly queue hundreds of unacknowledged events. When interrupts were subsequently re-enabled, this would then keep the CPU busy for a while acknowledging stale events. For these reasons, the unnecessarily complex 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 event-source, per EPP address-read.

Figure 3.11 illustrates the behavior of an IRQ register via a timing diagram.

The three interrupt sources that have been implemented, 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 a start-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, a FIFO has been implemented that contains 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,

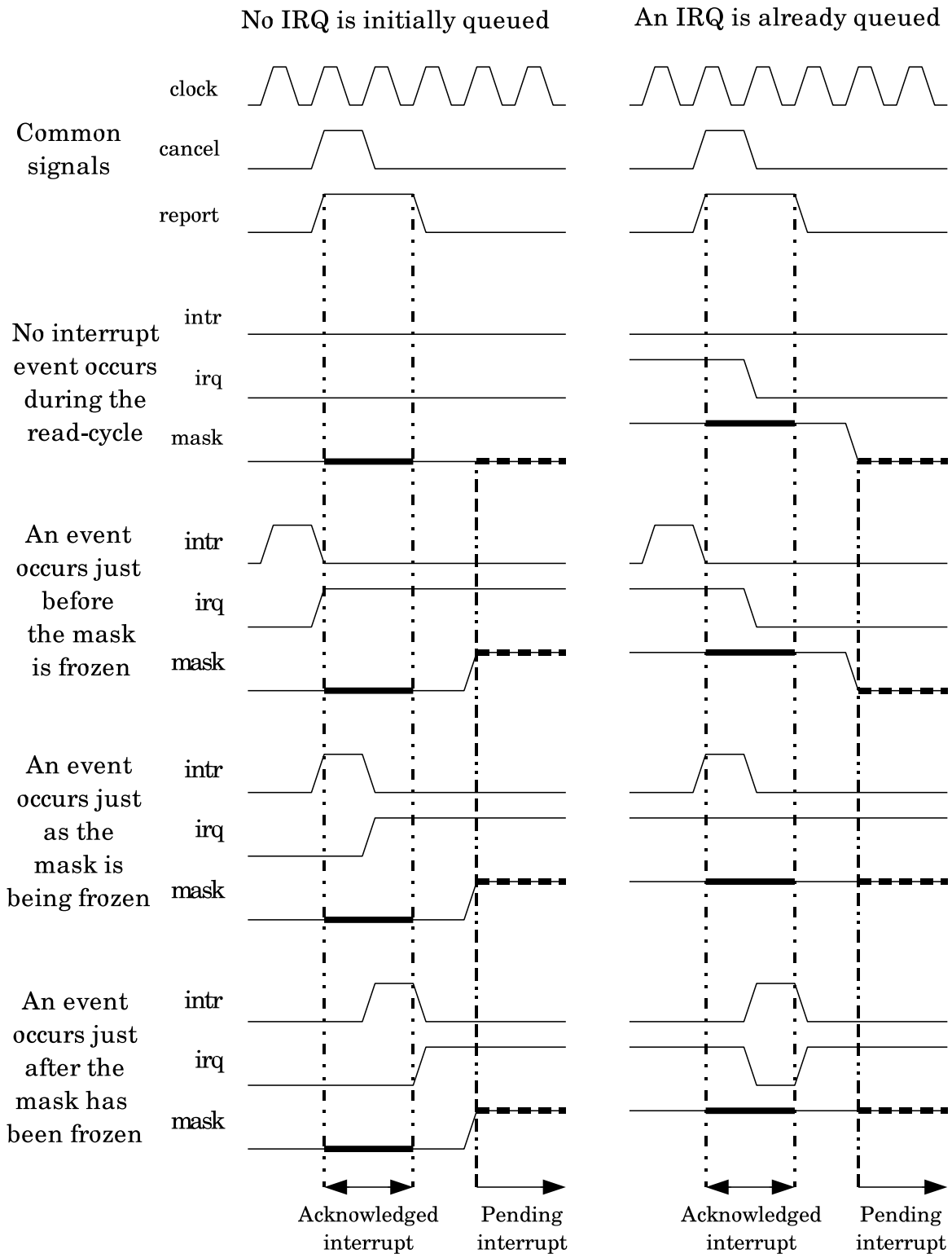

Figure 3.11: Timing diagrams of an IRQ Register during an EPP address-read

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` - Start-of-integration interrupts.

  Start of Integration interrupts are sent at the start of each integration period during a scan. In the interval between a start-scan command being received, and the resulting new scan starting, these interrupts cease, starting again at the start of the first integration period of the new scan.

  Note that in dump-mode scans, it can take longer than an integration period to dispatch the data that is collected during a single integration period, but start-of-integration interrupts continue to be generated for each consecutive integration period, even if the collection of a new frame of dump-mode data is not started at that time. Thus the number of these events that are reported won't always match the number of data frames sent to the computer over the USB interface.

  When a new integration starts before the interrupt from the start of the previous integration has been acknowledged by the computer, the new start-of-integration interrupt request is ignored, but the previous, unacknowledged 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, start-of-integration 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 possibly for debugging.

- `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, then the new one is simply ignored, while the *EPP interrupter* continues resending 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's 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 once interrupts become enabled.

3.2 The Data Dispatcher

In normal integration mode, data that are integrated during each integration period, are ready to be dispatched to the computer at the end of the integration period. In dump-mode, unintegrated data are immediately ready to be sent to the computer, at the start of each integration period. It is the *Data Dispatcher*'s responsibility to collect these integrated or dump-mode data and send them to the computer. In both cases, the *Data Dispatcher* reads the data 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 connected to pins of the master FPGA, there are currently no plans to use them.

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.

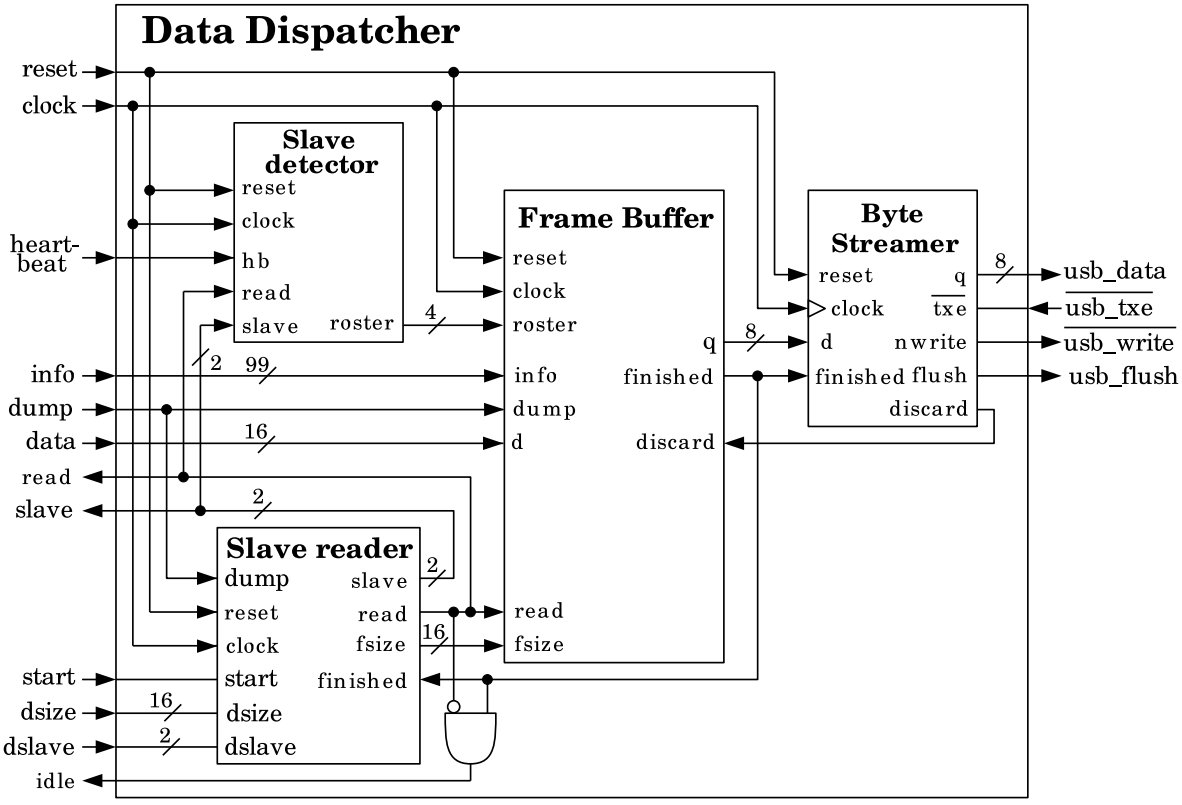

Figure 3.12: The Data Dispatcher

3.2.1 The internals of the Data Dispatcher

Figure 3.12 shows the building blocks of the *Data Dispatcher*, and how they are interconnected.

One clock cycle after the `start` input 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*.

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 the address of the highest numbered slave FPGA, and then, after all of that slave's samples have been transfered, it is set to that of the next lower numbered FPGA. This continues, until the samples of all of the FPGAs have been transfered to the *Frame Buffer*. In dump mode, the `slave` output is persistently given 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 master/slave data-bus into the *Frame Buffer*'s FIFO, the *Byte Streamer* streams the contents of the *Frame Buffer* to the USB interface chip, 8 bits at a time, starting with a header.

Once the contents of the *Frame Buffer* have been delivered to the USB chip, the `finished` output of the *Frame Buffer* is asserted, to tell the *Slave Reader* that it is okay for it to start collecting a new frame. One clock cycle after this, the USB chip's `flush` input is strobed low for one clock cycle, to tell it not to wait until it has a full packet, before, before sending any previously buffered data to the computer.

The *Slave Reader* de-asserts its `read` output, to terminate collection of the current data-frame, after the required number of samples have been read from the slaves. In dump-mode, if the number of dump-samples requested by the `dsize` input exceeds the size of the *Frame Buffer*, the *Slave Reader* limits the number of samples to be read, to the actual size of the 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 *Slave Reader* is careful to halt dump-mode data collection before the FIFO has a chance to overflow, there is no danger, either of gaps in the collected data, caused by temporary overflow conditions, 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 3.13.

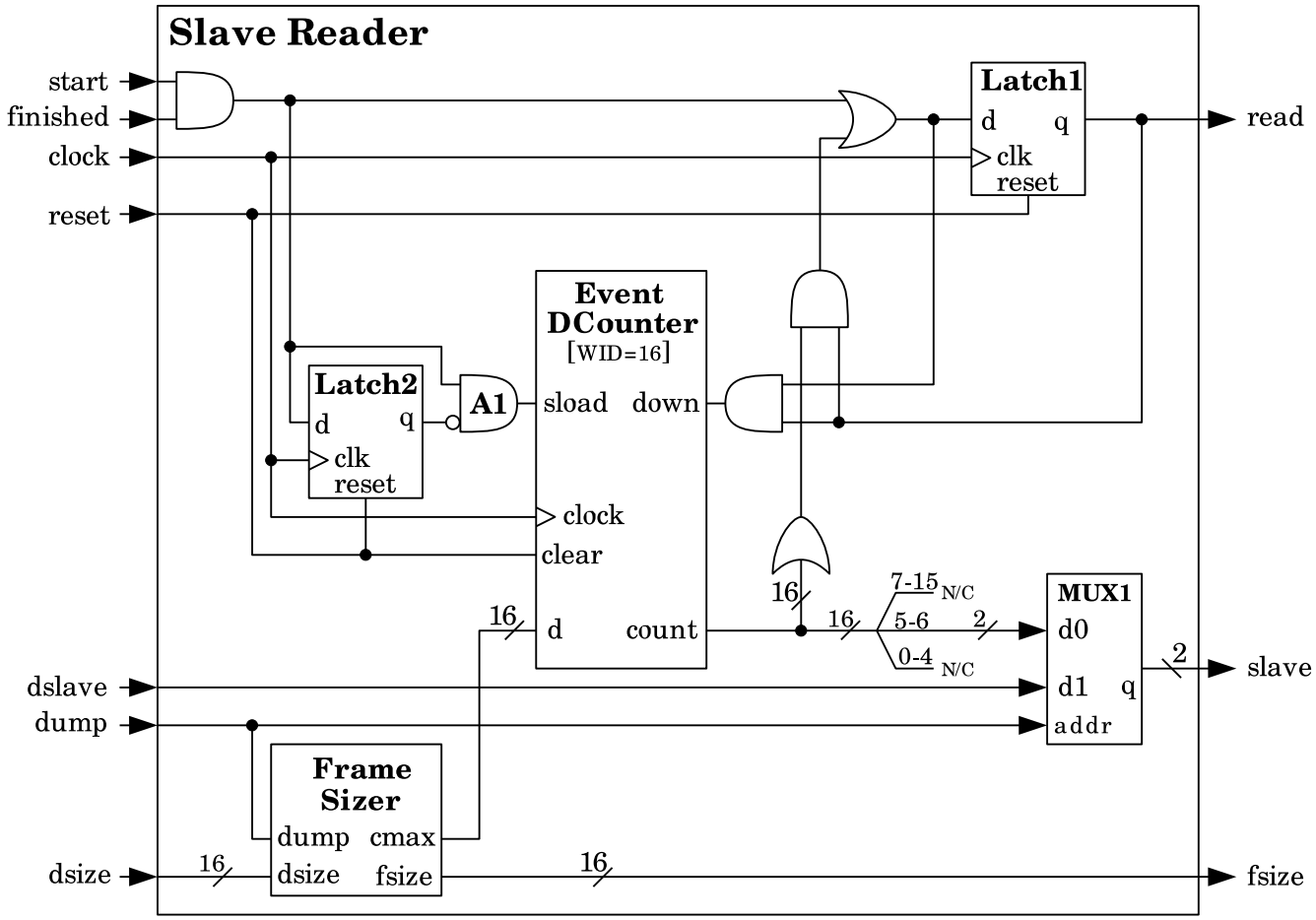

Figure 3.13: The Slave Reader

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.

When the `finished` 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 goes high.

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 integration period that is just starting.

Alternatively, if the `finished` 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 value of the `finished` input, 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 `finished` signal is asserted, although the `start` pulse only lasts for a few clock cycles (see later), `Latch1` and its surrounding logic thereafter hold the `read` signal high until 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 `finished` input, also loads a down-counter with the number of samples that are to be read. Since the `start` signal generally is asserted for a few clock cycles, `Latch2` and AND gate `A1` are used to generate a load pulse that rises asynchronously with the `start` signal, and then falls on the next rising edge of the clock, after being heeded by the `load` input. One clock cycle later, the `read` signal goes high, and the counter starts counting down by one at the start of each clock cycle, until one clock cycle before the `read` input is due to go low again. This happens when the output count of the counter finally reaches zero, after the requested number of samples have been read.

The number of samples that are to be read from the slaves, is computed by the *Frame Sizer*, whose simple VHDL implementation is shown in figure 3.14.

In normal integration mode, the *Frame Sizer* specifies, via its `fsize` output, that 128 16-bit words should be read from the slaves. This corresponds to 64, 32-bit integrations, with each sample split into pairs of 16-bit words. In dump-mode, if the value of the dump-lim register, which is presented by the `dsize` input, is less than or equal to the size of the FIFO in the *Frame Buffer*, then the *Frame Sizer* specifies that this number of samples be read. Otherwise, it specifies the limiting size of the FIFO.

The `fmax` output of the *Frame Sizer* is always one less than the value presented at the `fsize` output, and thus one less than the number of samples that are to be read. This is used as the preload-value of the `Event DCounter` component, which counts down by one, each time that a new sample is read. The preload value needs to be one less than the number of samples that are to be read, because it counts down to zero, before ending the reading of samples, rather than 1.

In normal integration mode, the 2 most significant bits of the output count, are used as the address of the slave that is to be read, such that 32 16-bit samples are read from one slave at a time, starting with the 4th slave, and working down to the 1st slave. Note that this relies on the use of a zero-relative countdown.

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

entity frame_sizer is
    Port ( dsize : in std_logic_vector(15 downto 0);
           dump : in std_logic;
           fsize, fmax : out std_logic_vector(15 downto 0));
end frame_sizer;

architecture Behavioral of frame_sizer is
  --
  -- Parameterize the total size of the FIFO in the Frame Buffer module.
  --
  constant buffer_size : std_logic_vector(15 downto 0) := "0100000000000000";
  --
  -- Create a wire for internal routing of the computed frame size.
  --
  signal frame_size : std_logic_vector(15 downto 0);
begin
  --
  -- Compute the number of 16-bit samples to be read from the slaves.
  --
  frame_size <=
    "0000000010000000" when dump='0' else  -- 64 integrated 32-bit samples.
    dsize when dsize < buffer_size else    -- dump_lim_reg dump-mode samples
    buffer_size;                           -- Limit to the size of the buffer.
  --
  -- Present the result at the fsize output.
  --
  fsize <= frame_size;
  --
  -- Compute the preload value for a zero-relative down-counter of samples.
  --
  fmax <= frame_size - 1;
end Behavioral;
```

Figure 3.14: The VHDL implementation of the Frame Sizer

In dump mode, the slave that is to be read, is not selected by the output count. Instead, `MUX1` arranges that all of the dump-mode samples be loaded from the single slave that is specified by the `dslave` input.

Above it was mentioned that the `start` signal is asserted for more than one clock cycle, at the start of each new dispatch period. What actually happens is that after asserting the `start` signal, the *State Generator* doesn't subsequently deassert it until it sees the `idle` output of the *Data Dispatcher* go low, indicating that the commanded operation has actually started. Thus, since the `idle` signal goes low when the `read` signal goes high, one clock cycle after the `start` input goes high, and the *State Generator* notices that the `idle` signal has gone low a further one clock cycle after this, in practice, the `start` signal is asserted for two clock cycles. Note that the number of clock cycles that the `start` signal remains asserted for, is irrelevant to the correct operation of the circuit.

To better illustrate the operation of this circuit, a timing diagram of it, derived by hand, is shown in figure 3.15.

The internals of the Frame Buffer

The *Frame Buffer* component assembles one frame of data each integration period. This consists of a descriptive header, and the samples of the integration period. It presents this

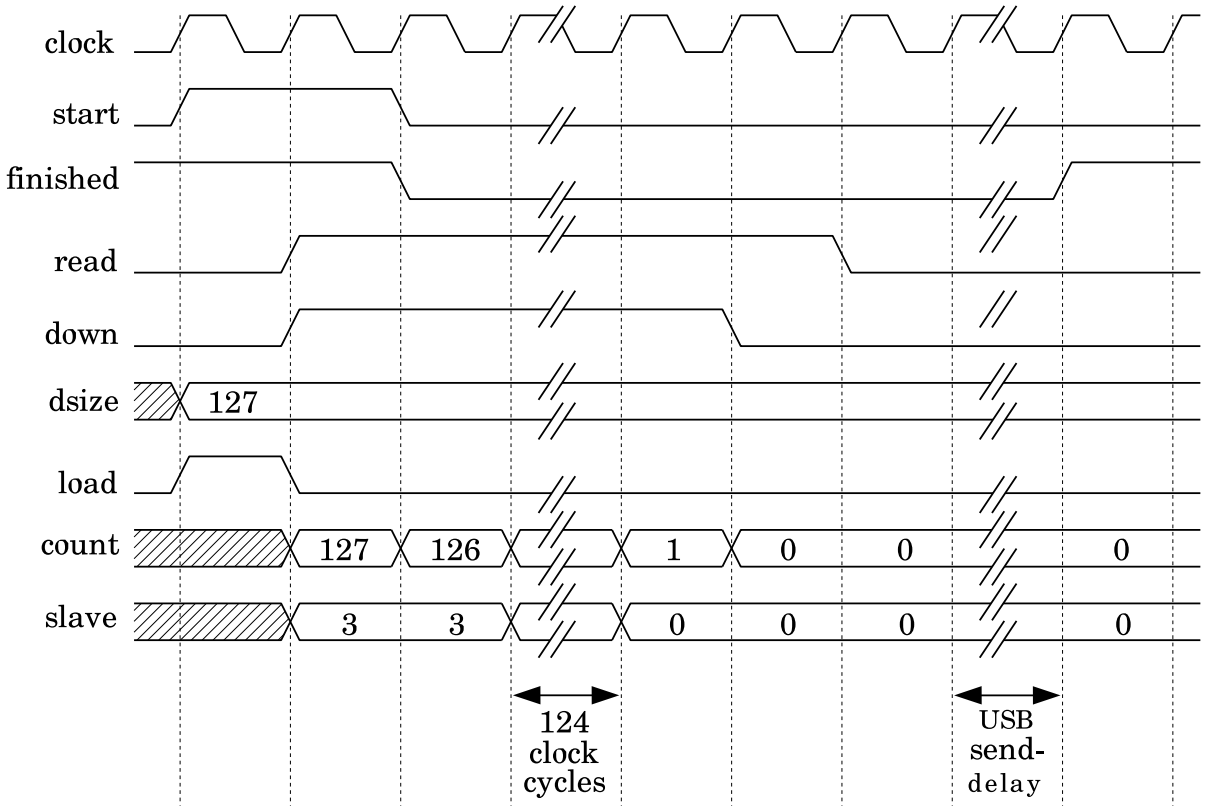

Figure 3.15: A timing diagram of the Slave Reader

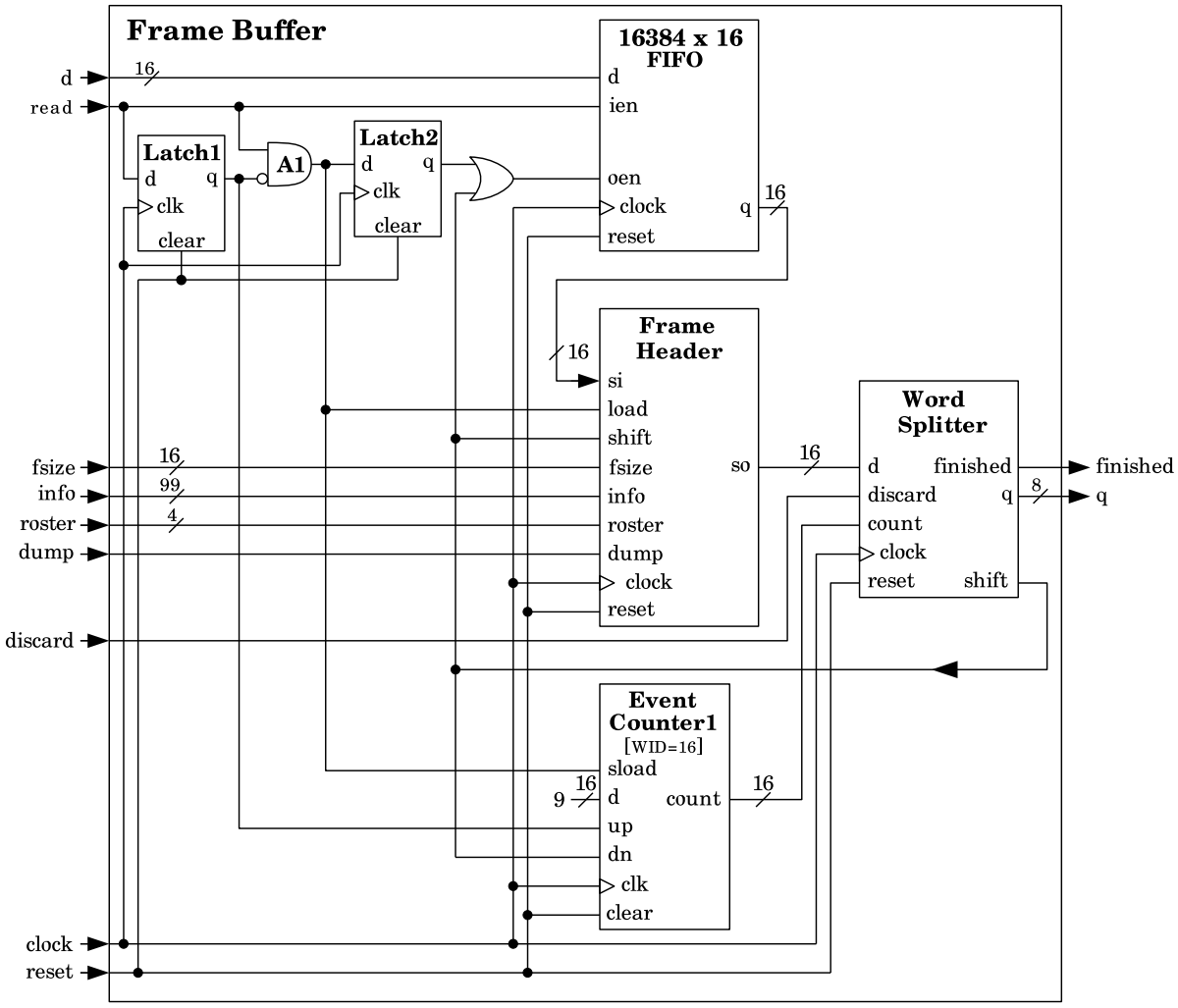

Figure 3.16: The Frame Buffer

frame, one 16-bit word at a time to the internal *Word Splitter* component, which in turn sequences first the least significant byte of this word, followed by the most significant byte, to the separate *Byte Streamer* component, for transmission to the computer, via the external USB chip.

The rate at which data can be sent to the computer over the USB bus, is much slower than the rate at which data are received from the slave FPGAs. So once data start to arrive from the slaves, the *Frame Buffer* remains occupied until the completed frame has been sent to the computer. During this time, `Event Counter1`, which keeps a record of the number of unsent 16-bit words in the *Frame Buffer*, has a non-zero output.

The `read` input is the same signal that is sent to the slave FPGAs to tell them when to send data. It stays high until all of the data that are to be part of the data frame, have been received from the slave FPGAs. The first sample that arrives from a slave is ready to be latched into the FIFO component, at the first rising clock edge that follows the `read` signal going high. On the same clock edge, the `Frame Header` component and `Event Counter1` are initialized, via the one clock-cycle load pulse that is generated by the combination of `Latch1` and AND gate `A1`. As will be described later, the `Frame Header` contains a 9-entry 16-bit wide PISO, and the result of the load pulse is to load these 9 words with header information that describes the contents of the frame. Since this immediately adds 9 words to the unsent contents of the `Frame Buffer`, `Event Counter1` is initialized with the value 9.

One clock cycle after the first sample has been latched by the FIFO, the load pulse, delayed by one clock cycle, by `Latch2`, asserts the output enable (`oen`) input of the FIFO, to move the first sample that was received, to the output of the FIFO. Subsequently, whenever the *Word Splitter* component pulses its `shift` output to indicate that it has finished with the current output word of the header-PISO, both the output-enable input of the FIFO, and the `shift` input of the *Frame Header* are pulsed. This results in the header-PISO shifting in the value at the output of the FIFO, and the FIFO and header-PISO components both replacing their output values with the next oldest values that they contain. In this manner, the `so` output of the *Frame Header*, clocked by the `shift` input, outputs an initial stream of header words, followed by a stream of samples from the FIFO.

At the start of each new clock cycle, `Event Counter1` counts up by one, if the delayed `read` input is still asserted, and counts down by one if the *Word Splitter*'s `shift` output is asserted. If both of its up-count and down-count inputs are asserted simultaneously, then its output count remains unchanged. The reason for counting up with a delayed version of the `read` input is to give the counter time to load its initial value, during the load pulse, before it starts counting down received samples. Since the rate at which data arrive always exceeds the rate at which those data are subsequently forwarded to the *Byte Streamer*, the `read` input will have returned low, and the counter stopped counting up, long before all of the data have been sent. So the time at which the counter's output subsequently falls to zero, is not delayed by the delay inserted between the `read` signal and the `up` input of the counter.

A timing diagram that illustrates the operation of the *Frame Buffer*, is shown in figure 3.17. Note that the behavior of the `discard` input is taken from the timing diagram of the *Byte Streamer* component, which will be shown later.

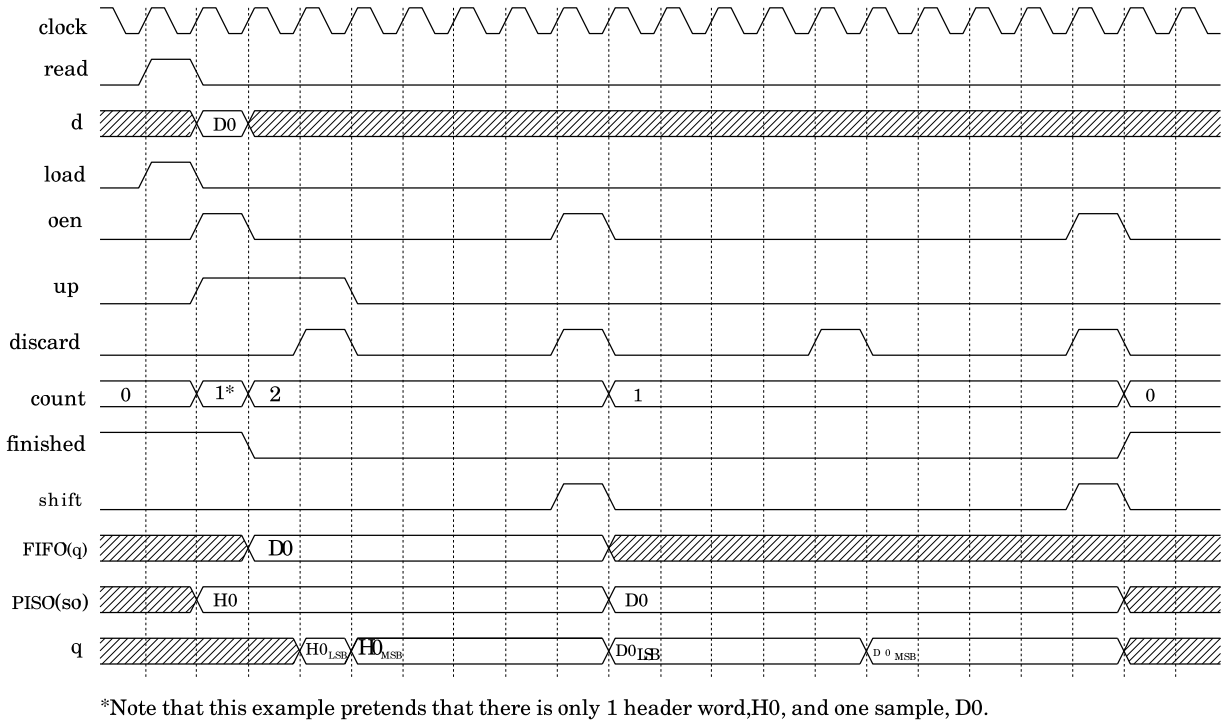

Figure 3.17: A timing diagram of the Frame Buffer

The internals of the Frame Header

As shown in figure 3.18, the *Frame Header* is basically a 9-entry, 16-bit synchronous PISO.

Note that instead of using a conventional PISO, an instance of the customized *CCB PISO*, described in section 3.4.3, is used. This has separate `load`-enable and `shift`-enable inputs, which are heeded at the rising edge of the clock. Unlike a conventional PISO, which always either serially its contents or loads parallel data on every clock cycle, the contents of *CCB PISO*'s 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 rising edge of the clock. This fills the nine 16-bit entries in the PISO with the information that is merged by the *Header Data* component.

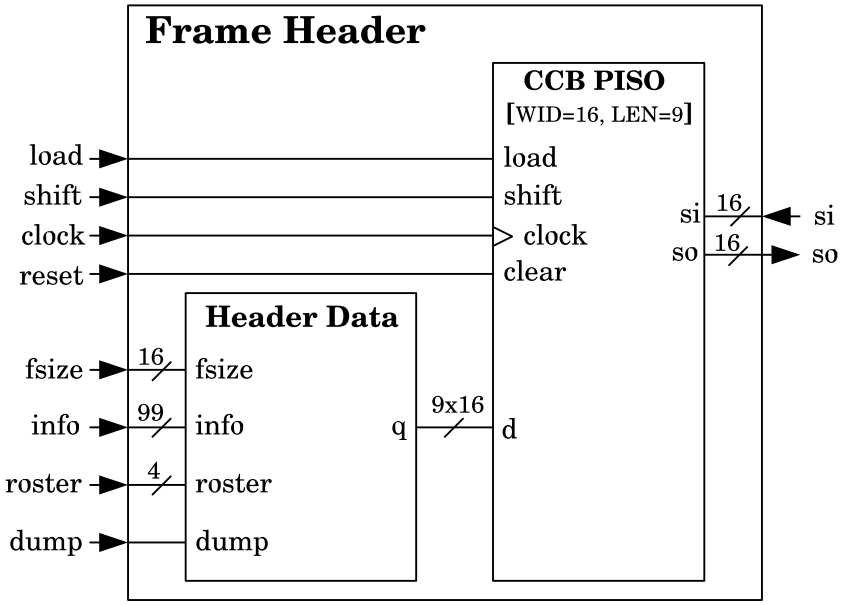

Figure 3.18: The Frame Header

This is the simple VHDL component, shown in figure 3.19.

The *Header Data* component combines the header information at its inputs into the single, multi-bit `q` output, which consists of a multiple of 16-bit words, consisting of the following items.

- The least significant 16-bit header word 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 indicated that they were present and functioning, while they were being read out during previous frames. Bit 4 is asserted

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

entity header_data is
  port (
    fsize  : in std_logic_vector(15 downto 0); -- The frame size.
    info   : in std_logic_vector(98 downto 0); -- The info vector from the
                                               --  Dispatch Controller.
    roster : in std_logic_vector(3 downto 0);  -- The roster of present slaves.
    dump   : in std_logic;                     -- The dump-mode flag.
    q      : out std_logic_vector((9*16)-1 downto 0)); -- The output header.
end header_data;

architecture header_data_arch of header_data is
begin
  --
  -- Header word 8.
  --
  q(143 downto 128) <= fsize;              -- The frame size.
  --
  -- Header words 7 and 6.
  --
  q(127 downto 96) <= info(98 downto 67);  -- The scan ID.
  --
  -- Header words 5 and 4.
  --
  q(95 downto 64) <= info(66 downto 35);   -- The start-time of the integration.
  --
  -- Header words 3 and 2.
  --
  q(63 downto 32) <= info(34 downto 3);    -- The integration ID.
  --
  -- Header word 1.
  --
  q(31 downto 23) <= (others => '0');      -- The 9 msbs are always zero.
  q(22 downto 21) <= info(2 downto 1);     -- The cal-diode target states.
  q(20) <= info(0);                        -- The cal-diode stability flag.
  q(19 downto 16) <= roster;
  --
  -- Header word 0.
  --
  q(15 downto 2) <= (others => '0');       -- The 14 msbs are always 0.
  q(1) <= dump;                            -- The dump-mode flag.
  q(0) <= '1';                             -- The lsb is always 1.
end header_data_arch;
```

Figure 3.19: The VHDL implementation of the Header Data component.

if the cal-diode switches were stable throughout the integration. Bits 5 and 6 report the commanded states of the cal-diode switches during the integration. The remaining 9 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 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 scan, with a resolution of 100ns. This counter wraps around every 430 seconds.

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

- The 7th and 8th header words are the least and most significant 16 bits of the 32-bit number which identifies the parent scan. The ID value that is sent, is the value that was in the `ccb_id_reg` register when the start-scan command for the scan was received.

- The 9th header word specifies how many 16-bit words of data follow the header.

Once the PISO has been loaded with the output of the *Header Data* component, 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 simultaneously shifting in a new value from the `si` input.

The internals of the Word Splitter

The *Word Splitter* module, embedded within the *Frame Buffer* component, takes one 16-bit word at a time from the *Frame Buffer*, splits this into 2 bytes, and sequences these, least significant byte first, followed by the most-significant byte, to the *Byte Streamer*, to be forwarded to the USB chip.

The *Word Splitter* outputs a new byte at its `q` output each time that the *Byte Streamer* asserts the `discard` signal, to indicate that it has transmitted the previous byte. This continues until the `count` input, which tells it how many unsent bytes remain in the *Frame Buffer*, reaches zero. At this point, instead of presenting a new byte for transmission, the *Word Splitter* asserts its `finished` output, to tell the *Byte Streamer* to send any partially full USB packet to the computer, as soon as possible.

The word that the *Word Splitter* splits into two bytes is the current output of the *Frame Header* component. At the start of a new frame, this automatically acquires the value of the first header word, but thereafter the *Word Splitter* tells the *Frame Header* when to present a new word, by asserting its `shift` output. The asserted `shift` signal also decrements the count of the number of unsent words at the `count` input.

The *Word Splitter* component is implemented as a finite state machine. This has the behavior that is shown in state diagram of figure 3.20.

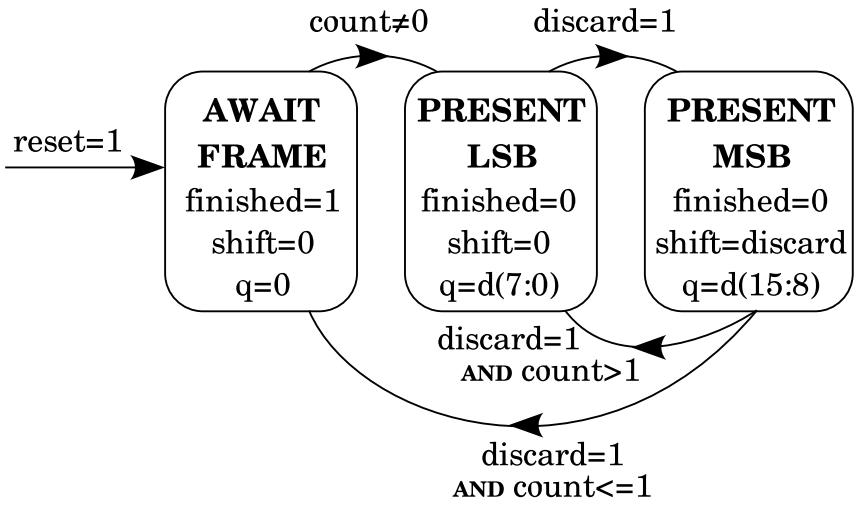

Figure 3.20: The state diagram of the Word Splitter FSM

The meanings of the states are as follows.

- `AWAIT FRAME`

  This state is assumed in the period that separates the completion of the transmission of one frame of data, and the start of the transmission of the next frame. During this time the `count` input, which reports the number of unsent bytes in the *Frame Buffer*, is zero. As soon as a new frame starts, this count becomes non-zero, and stays non-zero until the whole of the new frame has been sent. Thus, when `count` becomes non-zero, the state machine advances to the `PRESENT LSB` state, to start sending the first byte of the new frame.

- `PRESENT LSB`

  In this state, the least-significant byte of the 16-bit `d` input is presented at the `q` output, for transmission to the CPU. This value continues to be presented until the *Byte Streamer* asserts the `discard` input. At that point the state machine advances to the `PRESENT MSB` state.

- `PRESENT MSB`

  In this state, the most-significant byte of the `d` input is presented at the `q` output, for transmission to the CPU. Again, this value continues to be presented until the *Byte Streamer* asserts the `discard` input again. At this point the state-machine asserts the `shift` output, to tell the *Frame Buffer* that it is finished with the word at its `d` input.

  Then, if the `count` input indicates that there will still be unsent words, once the `shift` output has discarded the latest word, then the state machine returns to the `PRESENT LSB` state, to start sending the first byte of that new word.

  Alternatively, if `count` input will fall to zero, in response to the `shift` output pulse, then this means that the latest frame has been sent. Thus the `finished` output is asserted, to report this to the *Byte Streamer*, and the state-machine returns to the `AWAIT FRAME` state, to wait for the start of the next frame.

The state machine of the *Word Splitter* component, is implemented in VHDL, as shown in figure 3.21.

The internals of the Byte Streamer

The *Byte Streamer* performs the hand-shaking that is required to send one byte at a time from the *Frame Buffer* to the external USB chip, for subsequent transmission to the CPU. The timing requirements of the signals that interface with the USB chip are shown in figure 3.22.

The behavior of the state machine that implements this is illustrated in the state diagram of figure 3.23.

As elaborated by the small circuit beneath this state diagram, The `settling_txe` and `stable_txe` signals implement a synchronizer chain of two flip-flops, whose input is the asynchronous `txe` signal from the USB chip. A snapshot of the raw asynchronous `txe` signal is latched into the `settling_txe` flip-flop at the rising edge of each FPGA clock cycle. Since the raw `txe` signal could change state during the setup or hold time of this flip-flop, this will occasionally put the flip-flop into a metastable state. Thus to ensure that its value isn't consulted for at least one clock cycle, to give the metastability time to settle, the output of this flip-flop is latched into the `stable_txe` flip-flop one clock cycle later. The value of the `stable_txe` flip-flop is then used a further clock cycle later, such that any metastability in one or both of the flip-flops has time to resolve itself into a stable value at the `stable_txe` output, before this is used.

The necessary drawback of synchronizing the asynchronous `txe` signal, is that `stable_txe` lags changes in `txe` by between one and two clock cycles, depending on the phase of the change, relative to the FPGA clock. As will be seen below, accommodating this lag, involves the addition of a wait-state to the state machine.

The meanings of the states in the state diagram, are as follows.

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

entity word_splitter is
  port (clock, reset, discard : in  std_logic;
        d, count              : in  std_logic_vector(15 downto 0);
        q                     : out std_logic_vector(7 downto 0);
        finished, shift       : out std_logic);
end word_splitter;

architecture word_splitter of word_splitter is
  --
  -- Enumerate the states that the state machine goes through.
  --
  type states is (AWAIT_FRAME, PRESENT_LSB, PRESENT_MSB);
  signal state, next_state : states; -- The current and pending states.
  --
  -- Constants for assignments/comparisons with the count input and q output.
  --
  constant ZERO_COUNT : std_logic_vector(15 downto 0) := (others => '0');
  constant UNITY_COUNT : std_logic_vector(15 downto 0) := ZERO_COUNT(15 downto 1) & '1';
  constant ZERO_BYTE : std_logic_vector(7 downto 0) := (others => '0');
begin

  -------------------------------------------------------------------------
  -- The following procedure updates the registers of the state machine.
  -------------------------------------------------------------------------
  word_splitter_sequential_proc: process (clock, reset)
  begin
    if reset = '1' then                     -- Asynchronous reset?
      state <= AWAIT_FRAME;
    elsif clock'event and clock = '1' then  -- Rising clock edge?
      state <= next_state;
    end if;
  end process word_splitter_sequential_proc;

  -------------------------------------------------------------------------
  -- The following procedure generates purely combinatorial outputs.
  -------------------------------------------------------------------------
  word_splitter_combinatorial_proc: process (state, count, discard, d)
  begin
    --
    -- Generate the combinatorial outputs of each of the defined states.
    --
    case state is
      when AWAIT_FRAME =>           -- Wait for the start of a new frame.
        finished <= '1';            -- The last frame has finished being sent.
        shift <= '0';               -- Don't shift the empty frame-buffer.
        q <= ZERO_BYTE;             -- The current value of 'q' is unused.
        if count /= ZERO_COUNT then -- Has a new frame started?
          next_state <= PRESENT_LSB;
        else
          next_state <= AWAIT_FRAME;
        end if;

      when PRESENT_LSB =>           -- Present the LSB of d at q.
        finished <= '0';            -- A frame is still being sent.
        shift <= '0';               -- We haven't finished with the value at d.
        q <= d(7 downto 0);         -- Present the least-significant byte.
        if discard='1' then         -- Move onto the second byte of the word?
          next_state <= PRESENT_MSB;
        else                        -- Continue presenting the first byte?
          next_state <= PRESENT_LSB;
        end if;

      when PRESENT_MSB =>           -- Present the MSB of d at q.
        finished <= '0';            -- A frame is still being sent.
        shift <= discard;           -- Have we finished with the word at d?
        q <= d(15 downto 8);        -- Present the most-significant byte.
        if discard='1' and          -- No more words to send?
          count(count'high downto 1)=ZERO_COUNT(count'high downto 1) then
          next_state <= AWAIT_FRAME;
        elsif discard='1' then      -- Present the first byte of the next word?
          next_state <= PRESENT_LSB;
        else                        -- Continue presenting the second byte.
          next_state <= PRESENT_MSB;
        end if;

      when others =>                -- Recover from illegal states.
        finished <= '1';
        shift <= '0';
        q <= ZERO_BYTE;
        next_state <= AWAIT_FRAME;
    end case;
  end process word_splitter_combinatorial_proc;
end word_splitter;
```

Figure 3.21: The VHDL implementation of the Word Splitter

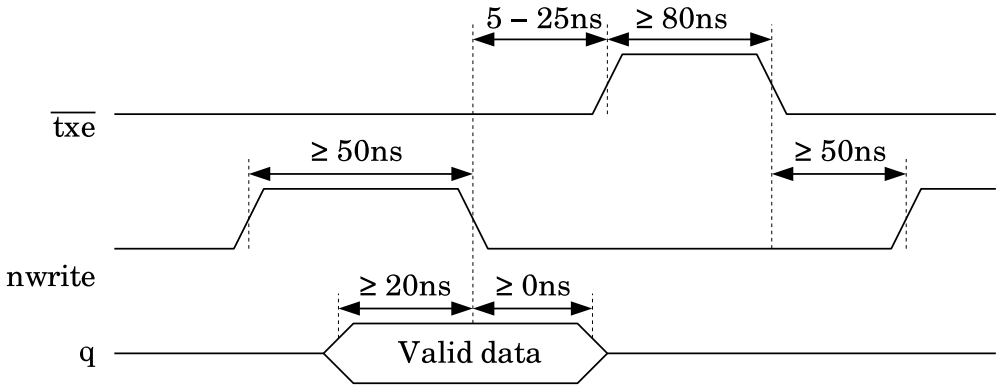

Figure 3.22: The timing specifications of a write-cycle to the USB chip's FIFO

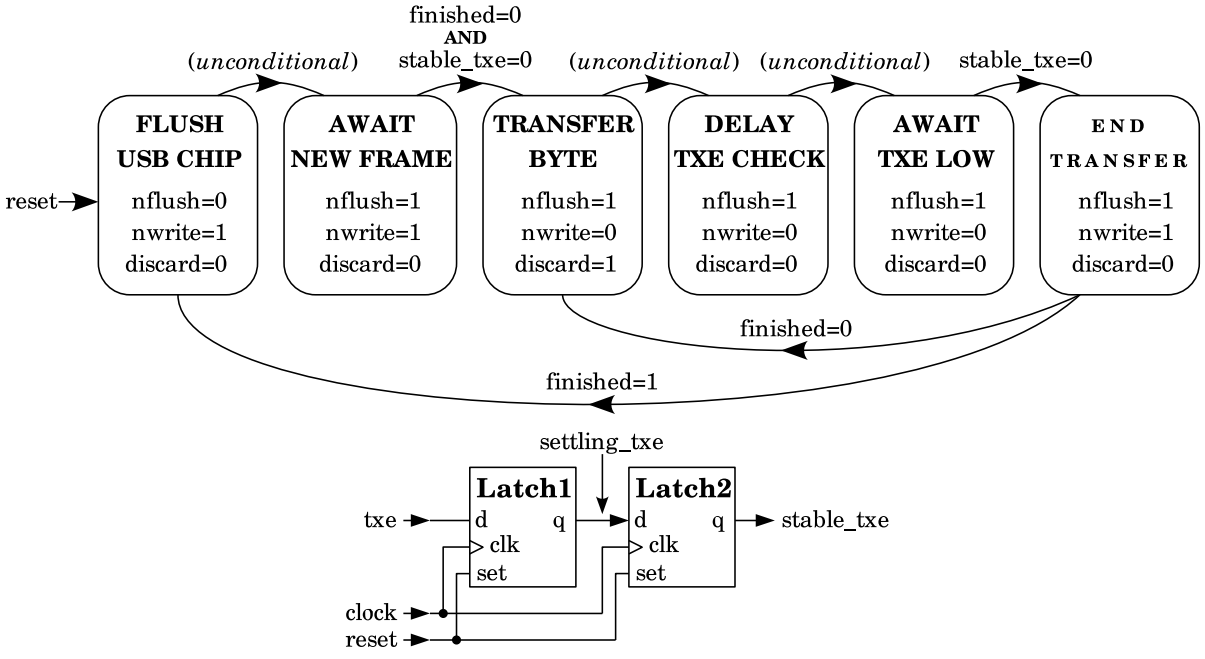

Figure 3.23: The state diagram of the Byte Streamer state machine

- `FLUSH USB CHIP`

  The state machine occupies this state until the first clock cycle after a firmware reset, and for one clock cycle after sending all of the data of the latest data-frame to the external USB chip. It asserts the active-low `nflush` signal, to tell the USB chip to send any unsent data in the chip's I/O buffer, as soon as possible, without waiting for its packet buffer to become full.

  Note that after a reset, this state drives the edge-sensitive `nwrite` signal into the inactive high state. Since the USB chip is documented to pull up the `nwrite` input during its firmware reset, this should prevent a downward edge being generated on this input, which could otherwise cause a bogus byte to be latched into the USB chip.

  After staying in this state for one clock cycle, the state machine moves on the `AWAIT NEW FRAME` state.

- `AWAIT NEW FRAME`

  The state machine stays in this state until the *Frame Buffer* tells it that the first byte of a new frame is available at the `d` input, and the assertion of the synchronized version of the `txe` signal, tells it that the USB chip is able to receive a new byte for subsequent transmission. It then advances to the `TRANSFER BYTE` state.

- `TRANSFER BYTE`

  This state lasts one clock cycle. It drives the `nwrite` output low, from its previously high state, and thus triggers the negative-edge-sensitive latching of the latest byte into the USB chip. Since this happens at least one clock cycle after the *Frame Buffer* indicated that a new byte had been presented, the data has by this time been presented to the USB chip for at least 100ns, which comfortably exceeds the 20ns minimum setup time required by the USB chip.

  At the start of the next clock cycle, the state machine advances to the `DELAY TXE CHECK` state.

- `DELAY TXE CHECK`

  This is the wait-state of one clock cycle which was briefly mentioned in the previous discussion of the two clock-cycle synchronization delay of the `stable_txe` signal.

  According to the data-sheet of the USB chip, it takes up to 20ns after the falling edge of the `nwrite` signal, for `txe` to go high, and a further 80ns or more for it to go low again. Thus the minimum time between the start of the preceding `TRANSFER BYTE` state (in which `nwrite` is driven low), and `txe` returning low (ie. asserted), is one FPGA clock cycle. The change in `txe` then takes a further two clock cycles to propagate through the synchronizer latches of this module and be reflected in the `stable_txe` signal. Thus we can't safely look at the `stable_txe` signal, to see if it is safe to send a new byte, for at least three clock cycles after the start of the `TRANSFER BYTE` state.

  If no wait-state were inserted between the `TRANSFER BYTE` state, at whose start `nwrite` went low, and the `AWAIT TXE LOW` state, then the value of `stable_txe` at the end of the first clock cycle of the `AWAIT TXE LOW` state, would be that of the moment when

the `TRANSFER BYTE` signal was transitioning low, instead of the required one clock cycle later. Thus a delay of one clock cycle is required.

After staying in this wait-state for one clock cycle, the state machine advances to the `AWAIT TXE LOW` state.

- `AWAIT TXE LOW`

  The state machine remains in this state until the `stable_txe` signal indicates that the USB chip has received the latest byte, and has room in its buffer for a new byte. As per the requirements illustrated in figure 3.22, `nwrite` continues to be held low during this time.

- `END TRANSFER`

  In this state, the single-byte transfer is ended by returning the `nwrite` output signal high. Because entering this state requires that `stable_txe` be low, and because the value of `stable_txe` lags changes in the value of the `txe` signal by between one and two clock 100ns cycles, the required minimum time of 50ns between `txe` going low and `nwrite` being returned high, is comfortably exceeded.

  After being in this state for one clock cycle, the state machine advances to either the `TRANSFER BYTE` state, to start transferring a new byte, or to the `FLUSH USB CHIP` state, to terminate the frame, and prepare to transfer the next frame.

A timing diagram that illustrates the operation of the *Byte Streamer*, is given in figure 3.24. In this diagram, the example frame has only one header entry, instead of 9, and only one data sample. Note that the state of the state machine, during each clock cycle, is indicated below the timing signals.

The VHDL implementation of the *Byte Streamer* state-machine is shown in figure 3.25.

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 3.26.

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.

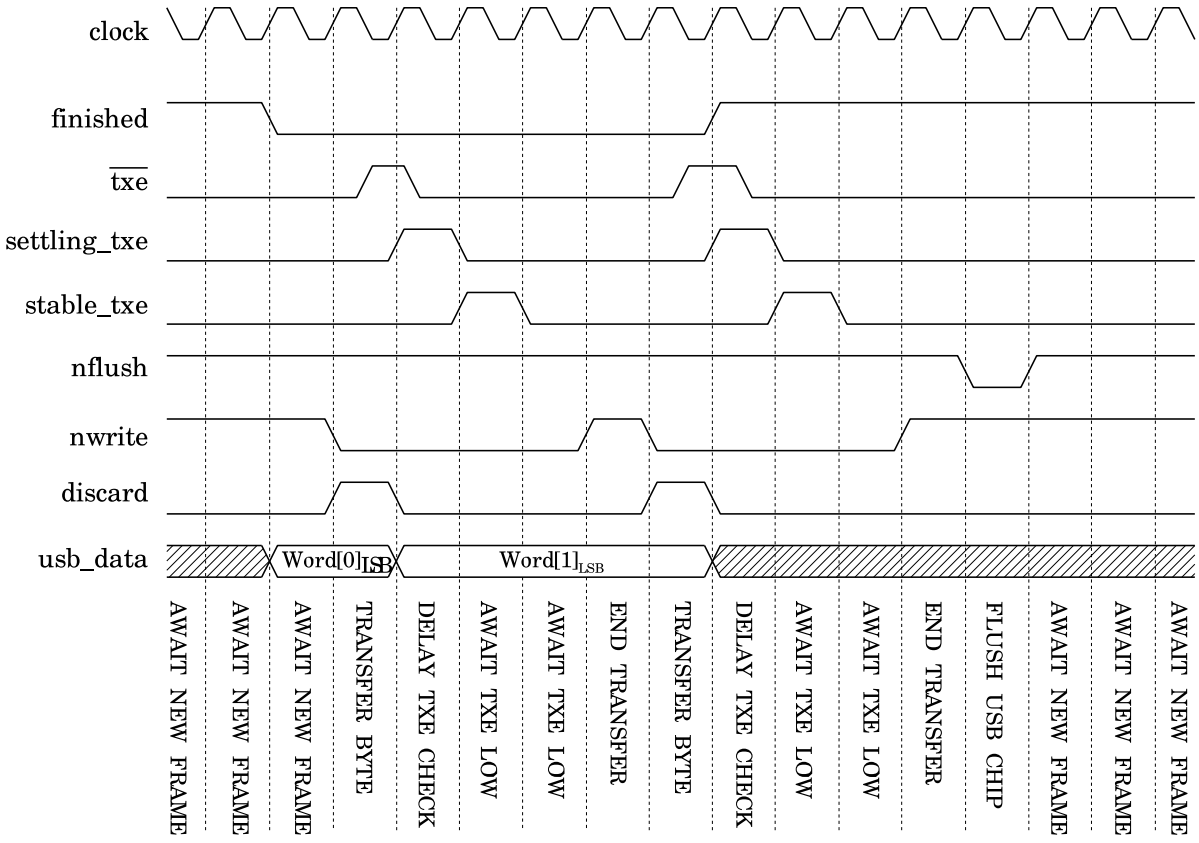

Figure 3.24: A timing diagram of the Byte Streamer

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

entity byte_streamer is
  port (clock, reset, finished, txe : in  std_logic;
        d                           : in  std_logic_vector(7 downto 0);
        q                           : out std_logic_vector(7 downto 0);
        nflush, discard, nwrite     : out std_logic);
end byte_streamer;

architecture byte_streamer of byte_streamer is
  --
  -- Enumerate the states that the state machine goes through.
  --
  type states is (FLUSH_USB_CHIP, AWAIT_NEW_FRAME, TRANSFER_BYTE,
                  DELAY_TXE_CHECK, AWAIT_TXE_LOW, END_TRANSFER);
  signal state, next_state : states;
  --
  -- Declare the signals that represent the txe synchronization flip-flops.
  --
  signal settling_txe, stable_txe : std_logic;
begin

  q <= d;  -- The byte to be sent comes directly from the Frame Buffer output.

  -------------------------------------------------------------------------------
  -- The following process implements the reqisters of the state machine.
  -------------------------------------------------------------------------------
  byte_streamer_register_proc: process (clock, reset)
  begin
    if reset = '1' then                     -- Asynchronous reset?
      state <= FLUSH_USB_CHIP;
      settling_txe <= '1'; stable_txe <= '1';
    elsif clock'event and clock = '1' then  -- Rising clock edge?
      state <= next_state;
      settling_txe <= txe; stable_txe <= settling_txe;
    end if;
  end process byte_streamer_register_proc;

  -------------------------------------------------------------------------------
  -- The following procedure generates the combinational outputs of the FSM.
  -------------------------------------------------------------------------------
  byte_streamer_combinatorial_proc: process (state, finished, stable_txe)
  begin
    --
    -- Generate the combinatorial outputs of the current state.
    --
    case state is
      when FLUSH_USB_CHIP =>  -- Tell the USB chip to send any unsent data.
        nflush <= '0'; nwrite <= '1'; discard <= '0';
        next_state <= AWAIT_NEW_FRAME;
      when AWAIT_NEW_FRAME => -- Wait for a new frame.
        nflush <= '1'; nwrite <= '1'; discard <= '0';
        if finished='0' and stable_txe='0' then
          next_state <= TRANSFER_BYTE;
        else
          next_state <= AWAIT_NEW_FRAME;
        end if;
      when TRANSFER_BYTE =>   -- Transfer the latest byte to the USB chip.
        nflush <= '1'; nwrite <= '0'; discard <= '1';
        next_state <= DELAY_TXE_CHECK;
      when DELAY_TXE_CHECK => -- Delay checking stable_txe.
        nflush <= '1'; nwrite <= '0'; discard <= '0';
        next_state <= AWAIT_TXE_LOW;
      when AWAIT_TXE_LOW =>   -- Await transmit-enable from the USB chip.
        nflush <= '1'; nwrite <= '0'; discard <= '0';
        if stable_txe = '0' then
          next_state <= END_TRANSFER;
        else
          next_state <= AWAIT_TXE_LOW;
        end if;
      when END_TRANSFER =>    -- Complete the transfer of the latest byte.
        nflush <= '1'; nwrite <= '1'; discard <= '0';
        if finished='0' then
          next_state <= TRANSFER_BYTE;  -- Start sending the next byte.
        else
          next_state <= FLUSH_USB_CHIP; -- Terminate the frame.
        end if;
      when others =>          -- Recover from illegal states.
        nflush <= '1'; nwrite <= '1'; discard <= '0';
        next_state <= FLUSH_USB_CHIP;
    end case;
  end process byte_streamer_combinatorial_proc;
end byte_streamer;
```

Figure 3.25: The VHDL implementation of the Byte Streamer's state-machine

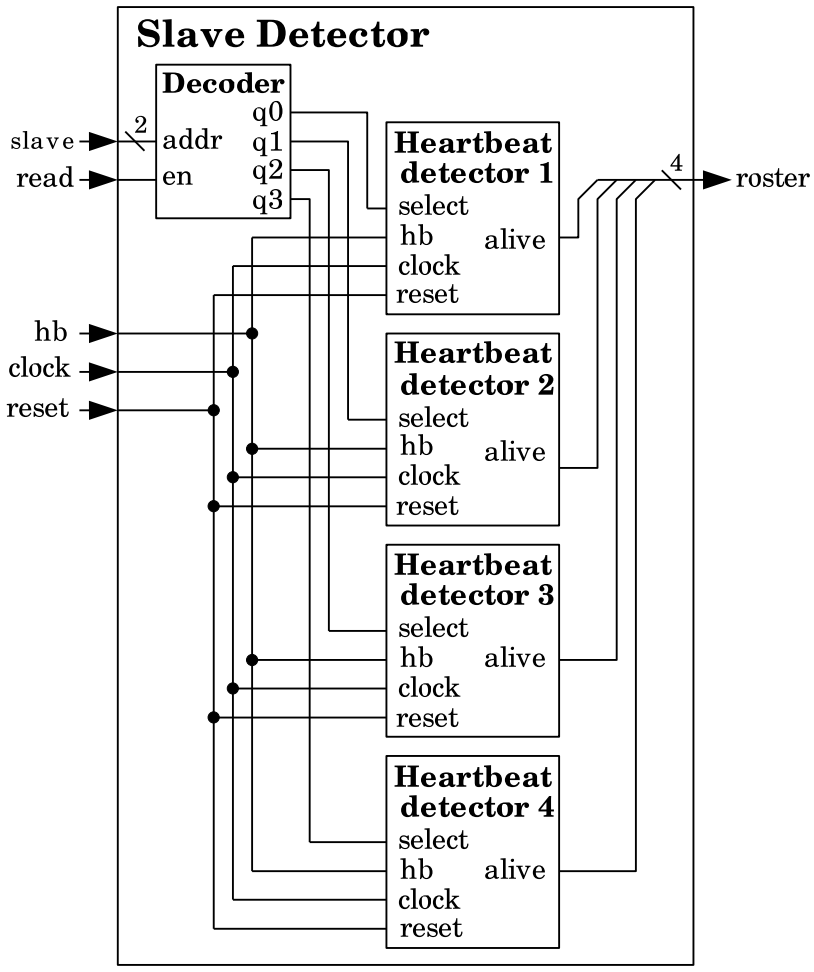

Figure 3.26: The Slave Detector

The internals of the Heartbeat Detector modules

The implementation of the individual *Heartbeat Detector* modules is shown in figure 3.27.

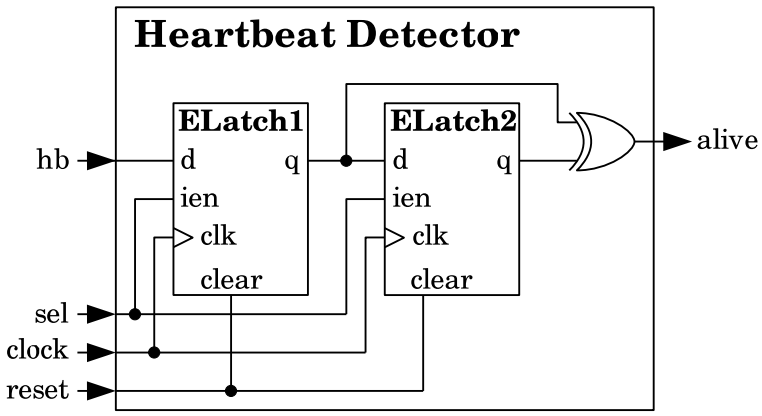

Figure 3.27: The Heartbeat Detector

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, from 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 another 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.

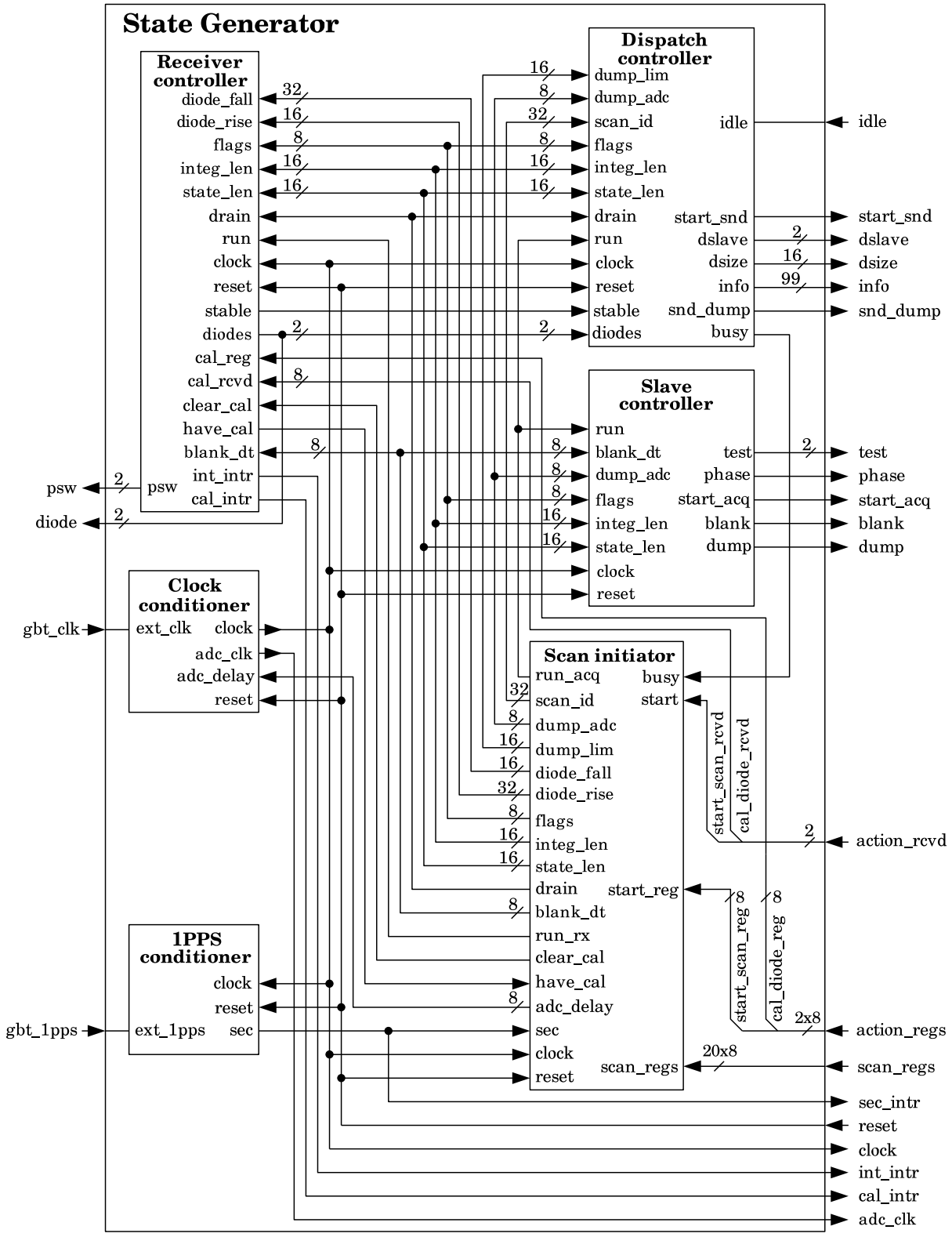

Figure 3.28: The State Generator

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 3.28.

Along the lower half of the left edge of figure 3.28, 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*, and the `idle` signal which tells the *State Generator* when the *Data Dispatcher* is not busy sending data to the computer.

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 values of the registers that configure each scan, enter via the `scan_regs` input. Command registers that trigger responses when written to, enter via the `action_regs` input, and are accompanied by corresponding receipt-notification signals at the `action_rcvd` input.

The official list of CCB registers, together with their contents and effects can be found in appendix A.

The main components within the *State Generator* are as follows:

- 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.

- The *Clock Conditioner* takes an external 10MHz clock signal, forces its mark-space ratio to unity, and outputs both this clock signal, as the master clock used by all modules in both the master FPGA and the slave FPGAs, as well as a configurably phase-shifted version of this signal, which is used by the slave FPGAs, to clock the external ADCs.
- The *1PPS Conditioner* generates a synchronized pulse of one clock cycle duration, each time that it sees a pulse from the 1-pulse-per-second input. This is used both to generate an interrupt to the CPU, and optionally to synchronize the start-times of scans.

The above components will now be described in detail.

3.3.1 The Scan Initiator

Within the *State Generator*, the *Scan Initiator* has the job of responding to start-scan commands from the computer. On receiving a start-scan command, it orchestrates the orderly completion of any currently running scan, reconfiguration of the *State Generator* for the requested new scan, and starting up the next scan at the appropriate time. The implementation is shown in figure 3.29.

Whenever a new start-scan command is received, `Ereg1` takes a snapshot of the scan configuration registers. This is later used to configure the new scan, after the current one has completed.

Latch 1, together with its surrounding logic, implements a bistable latch. This goes high when the *Scan Synchronizer* briefly asserts its `start_rx` output, and thereafter remains high until the *Scan Synchronizer* asserts its `config` output. The resulting `run_rx` output controls when the *Receiver Controller* should be running, and when it should be stopped and reconfiguring itself for the next scan.

Similarly, `latch2` implements another bistable, which goes high when the *Scan Synchronizer* briefly asserts its `start_acq` output, and thereafter remains high until the *Scan Synchronizer* asserts its `config` output. The resulting `run_acq` output controls when the *Slave Controller* and *Dispatch Controller* modules should be running, and when they should be stopped and reconfiguring themselves for the next scan.

The `drain` output tells the other modules of the *State Generator* that the current integration period is the last one of the scan that is just ending, and in response, the `busy` input is required to be deasserted by the *Dispatch Controller*, when it has finished dispatching the data of that final integration.

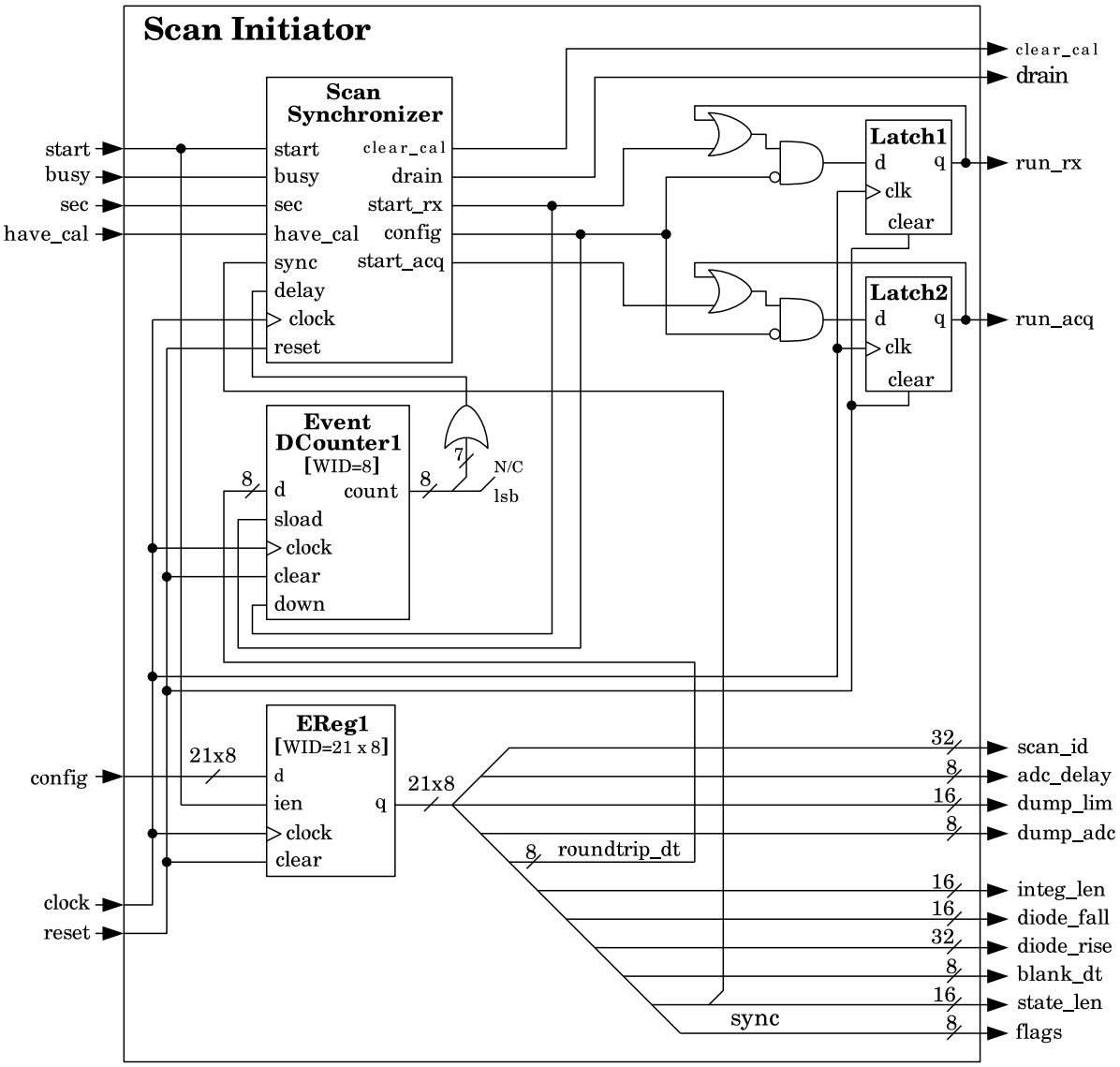

Figure 3.29: The Scan Initiator

The `clear_cal` output is asserted when it is time for the *Cal Controller* to throw away obsolete queued calibration configurations from the previous scan, and ready itself to accept new calibration configurations for the next scan. In response, the *Cal Controller* drives the `have_cal` input low, when it has done this. Subsequently, after the `clear_cal` output is returned low, to allow the *Cal Controller* to solicit and store new calibration configurations, the *Cal Controller* asserts the `have_cal` input as soon as a calibration configuration for the first integration has arrived.

The purpose of `Event DCounter1` is to count down the predicted time that it takes for the slave FPGAs to see the results of any change in the cal-diode or phase-switch control signals.

The step by step operation of the *Scan Initiator* is governed by the *Scan Synchronizer* module. This is implemented as a finite state machine. It's actions are illustrated by the state diagram in figure 3.30.

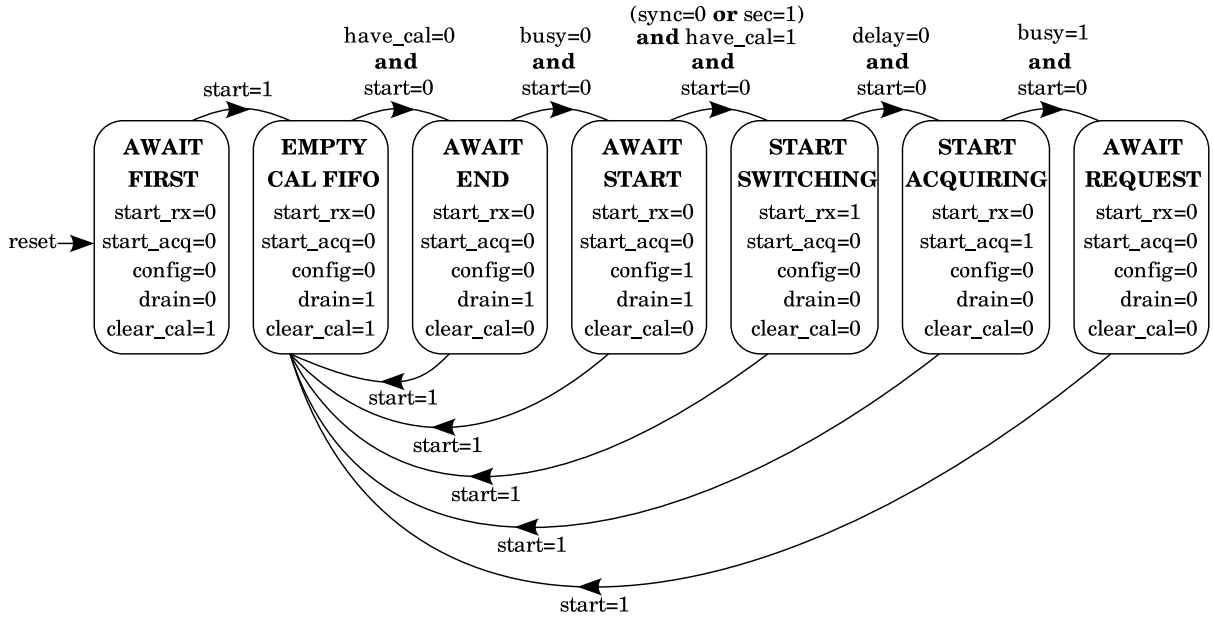

Figure 3.30: The state diagram of the Scan Synchronizer FSM

The meanings of the states are as follows.

- `AWAIT FIRST`

  After a firmware reset, the state machine remains in the `AWAIT FIRST` state until the first start-scan command is received. In this state the `clear_cal` output is held high, to prevent cal-diode interrupts from being generated until after the first start-scan command is received.

The `start_rx` and `start_acq` outputs are held low, such that the external bi-stables that drive the corresponding `run_rx` and `run_acq` outputs continue to hold their post-reset, deasserted values.

The `drain` output is held low, because there has been no previous scan to clean up after up.

The `config` output is held low, since there is no need to explicitly deassert the already deasserted bi-stables that drive the `run_rx` and `run_acq` outputs of the *Scan Initiator*, and there is no pending scan to configure yet.

- `EMPTY CAL FIFO`

  Whenever the computer sends a start-scan command, the `start` input is asserted for one clock cycle. This causes the state machine to switch to the `EMPTY CAL FIFO` state, regardless of its existing state. This is the first step in ending an old scan, if any, and starting a new one.

  In this state the `clear_cal` output is asserted, to tell the *Cal Controller* to discard any unused cal-diode configurations from the terminating scan, and then prepare itself to receive new ones for the pending scan. The `drain` output is also asserted, to tell both the *Cal Controller* and the *Dispatch Controller* that the current integration period is the last of the current scan. This stops the *Cal Controller* from incorrectly using up the first cal-diode configuration when the current integration ends. Similarly it tells the *Dispatch Controller* not to start collecting and dispatching a new frame at the end of the current integration period, and to deassert its `busy` output once the data of the current integration period have been safely dispatched to the computer. Finally, it tells the *Receiver Controller* not to request any more start-of-integration interrupts.

  The state machine remains in this state until the *Cal Controller* deasserts its `have_cal` output, to indicate that it has emptied its FIFO.

- `AWAIT END`

  In this state, the `clear_cal` output is deasserted, to allow the *Cal Controller* to use cal-diode interrupts to tell the computer to start sending cal-diode configurations for the pending new scan.

  Meanwhile, the `drain` output remains asserted, to continue telling the *Dispatch Controller* and *Cal Controller* modules to finish up the final integration of the previous scan.

  The state machine remains in this state until the *Dispatch Controller* indicates, by deasserting its `busy` output, that the data of the final integration period have been sent to the computer. It then advances to the `AWAIT START` state.

- `AWAIT START`

  In this state, while waiting to start the new scan, the `config` output is driven high, to halt the external controller modules, and force them to load new scan configuration parameters from the scan-configuration snapshot that is held in `EReg1` of the *Scan Initiator*. This order is communicated to the controller modules by deasserting the

external bi-stables that drive the `run_rx` and `run_acq` outputs of the *Scan Initiator*. Note that it is important that the controller modules not directly consult the scan-configuration snapshot at any other time during a scan, because the contents of the snapshot will change as soon as the next start-scan command is received, before the current scan has finished.

The state machine remains in the `AWAIT START` state for one or more clock cycles, until the following two conditions for starting a new scan are satisfied.

1. The cal-diode configuration of the first integration period has been received from the computer.
2. If the synchronize-with-1PPS flag was included in the flags of the start-scan command, then a new scan can start once the next 1-second tick has been seen. If the synchronization flag was not included, then the scan should start as soon as possible.

Once these conditions have been met, then the state machine advances to the `START SWITCHING` state.

- `START SWITCHING`

  This state tells the external *Receiver Controller* module to start cycling the phase switches and cal-diodes through the states that pertain to the new scan. It tells it this by asserting the `start_rx` output, which in turn asserts the bistable that drives the `run_rx` output of the *Scan Initiator*.

  The state machine remains in this state until enough time has passed for the effects of the initial cal-diode and phase-switch control signals that the *Receiver Controller* outputs, to propagate to the receiver, and start affecting the samples that are subsequently received by the slave FPGAs. This roundtrip control delay is counted down by `Event DCounter1`, in the *Scan Initiator*, while the `run_rx` output remains high. Note that this counter was initialized with the configured roundtrip delay during the preceding `AWAIT START` state.

  The roundtrip counter in the *Scan Initiator* starts counting down one clock cycle after the `start_rx` output first goes high, and thus, provided that the `roundtrip_dt` configuration value was greater than zero, the output count reaches 1 at the start of the `roundtrip_dt`'th clock cycle after the `start_rx` signal went high. This is the point at which the state machine should advance to the `START ACQUISITION` state.

  In the special case of the `roundtrip_dt` configuration value being incorrectly configured to be zero, then the counter starts at zero, and the state machine should advance to the `START ACQUISITION` state as soon as possible.

  To cater to both of the above cases, the state machine advances to the `START ACQUISITION` state when the roundtrip countdown value is either 1 or 0.

- `START ACQUISITION`

This state starts the external *Slave Controller* and *Dispatch Controller* modules acquiring the first samples of the new scan. It does this by temporarily asserting the `start_acq` output, which in turn, causes the `run_acq` output of the parent *Scan Initiator* to go high, and stay high until the end of the scan.

The `start_rx` output is returned low in this state, leaving the bistable in the *Scan Initiator* module to continue asserting the `run_rx` output.

The state machine remains in the `START ACQUISITION` state until the *Dispatch Controller* asserts the `busy` input, to indicate that it has started acquiring data. It then returns to the `AWAIT START` state.

- `AWAIT REQUEST`

  Once the state machine has delegated the running of the new scan to the *Receiver Controller*, *Slave Controller* and *Dispatch Controller* modules, it remains in the `AWAIT REQUEST` state until the computer sends a new start-scan request. In this state, all of the outputs of the *Scan Synchronizer* are held low, since no new operations need to be requested during this time.

  Subsequently, when a new scan is requested by the computer, the state machine returns to the `EMPTY CAL FIFO` state, to prepare to start the new scan.

Note that at the cores of each of the *Receiver Controller*, *Slave Controller* and *Dispatch Controller* modules, are identical copies of a `Scan Sequencer` module, which generates the timing signals that control transitions between the various states within a scan. Because the above state machine delays asserting the `start_acq` signal by the roundtrip control delay, with respect to the `start_rx` signal, the `Scan Sequencer` in the *Receiver Controller* starts this much earlier than the `Scan Sequencer`s in the *Slave Controller* and *Dispatch Controller* modules. Thus the timing ticks that govern data-acquisition are correctly delayed with respect to those that generate the receiver control signals, by the amount of time that it takes for the effects of toggling phase-switches and cal-diode states to propagate to the slave FPGAs.

The state machine that implements the state-diagram of figure 3.30, is implemented by the VHDL code shown in figure 3.31. Note that in this figure most comments have been stripped, and the code has been reformatted, to make it fit on a single page.

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

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

entity scan_synchronizer is
  port (clock, reset, start, busy, sec, delay, have_cal : in std_logic;
        start_rx, start_acq, config, drain, clear_cal   : out std_logic);
end scan_synchronizer;

architecture scan_synchronizer_arch of scan_synchronizer is
  type states is (AWAIT_FIRST, EMPTY_CAL_FIFO, AWAIT_END, AWAIT_START,
                  START_SWITCHING, START_ACQUIRING, AWAIT_REQUEST);
  signal state, next_state : states;
begin
  process (clock, reset)  -- Synchronously advance the state machine.
  begin
    if reset = '1' then                     -- An asynchronous reset.
      state <= AWAIT_FIRST;
    elsif clock'event and clock = '1' then  -- The rising edge of the clock.
      state <= next_state;
    end if;
  end process;

  process (state, start, busy, sync, sec, delay, have_cal)  -- Set the outputs of each state.
  begin
    case state is
      when AWAIT_FIRST =>   -- Await the first start-scan command.
        start_rx <= '0'; start_acq <= '0'; config <= '0'; drain <= '0'; clear_cal <= '1';
        if start='1' then
          next_state <= EMPTY_CAL_FIFO;
        else
          next_state <= AWAIT_FIRST;
        end if;
      when EMPTY_CAL_FIFO => -- Empty the cal-diode FIFO of any relics of the previous scan.
        start_rx <= '0'; start_acq <= '0'; config <= '0'; drain <= '1'; clear_cal <= '1';
        if start='0' and have_cal='0' then
          next_state <= AWAIT_END;
        end if;
      when AWAIT_END =>  -- Await the completion of the previous scan.
        start_rx <= '0'; start_acq <= '0'; config <= '0'; drain <= '1'; clear_cal <= '0';
        if start='1' then
          next_state <= EMPTY_CAL_FIFO;
        elsif busy='0' then
          next_state <= AWAIT_START;
        else
          next_state <= AWAIT_END;
        end if;
      when AWAIT_START =>    -- Halt and reconfigure, while waiting to start the new scan.
        start_rx <= '0'; start_acq <= '0'; config <= '1'; drain <= '1'; clear_cal <= '0';
        if start='1' then
          next_state <= EMPTY_CAL_FIFO;
        elsif (sync='0' or sec='1') and have_cal='1' then
          next_state <= START_SWITCHING;
        else
          next_state <= AWAIT_START;
        end if;
      when START_SWITCHING =>    -- Start asserting run_rx.
        start_rx <= '1'; start_acq <= '0'; config <= '0'; drain <= '0'; clear_cal <= '0';
        if start = '1' then
          next_state <= EMPTY_CAL_FIFO;
        elsif delay = '0' then  -- Wait for the roundtrip countdown to finish.
          next_state <= START_ACQUIRING;
        else
          next_state <= START_SWITCHING;
        end if;
      when START_ACQUIRING =>   -- Start asserting run_acq.
        start_rx <= '0'; start_acq <= '1'; config <= '0'; drain <= '0'; clear_cal <= '0';
        if start = '1' then
          next_state <= EMPTY_CAL_FIFO;
        elsif busy='1' then
          next_state <= AWAIT_REQUEST;
        else
          next_state <= START_ACQUIRING;
        end if;
      when AWAIT_REQUEST =>   -- Await the next start-scan request.
        start_rx <= '0'; start_acq <= '0'; config <= '0'; drain <= '0'; clear_cal <= '0';
        if start='1' then
          next_state <= EMPTY_CAL_FIFO;
        else
          next_state <= AWAIT_REQUEST;
        end if;
      when others =>      -- Handle illegal states.
        start_rx <= '0'; start_acq <= '0'; config <= '0'; drain <= '0'; clear_cal <= '0';
        if start = '1' then
          next_state <= EMPTY_CAL_FIFO;
        else
          next_state <= AWAIT_REQUEST;
        end if;
    end case;
  end process;
end scan_synchronizer_arch;
```

Figure 3.31: The VHDL implementation of the Scan Synchronizer

the interrupt-signal that signals the start of each integration period. The implementation of the *Receiver Controller* is shown in figure 3.32.

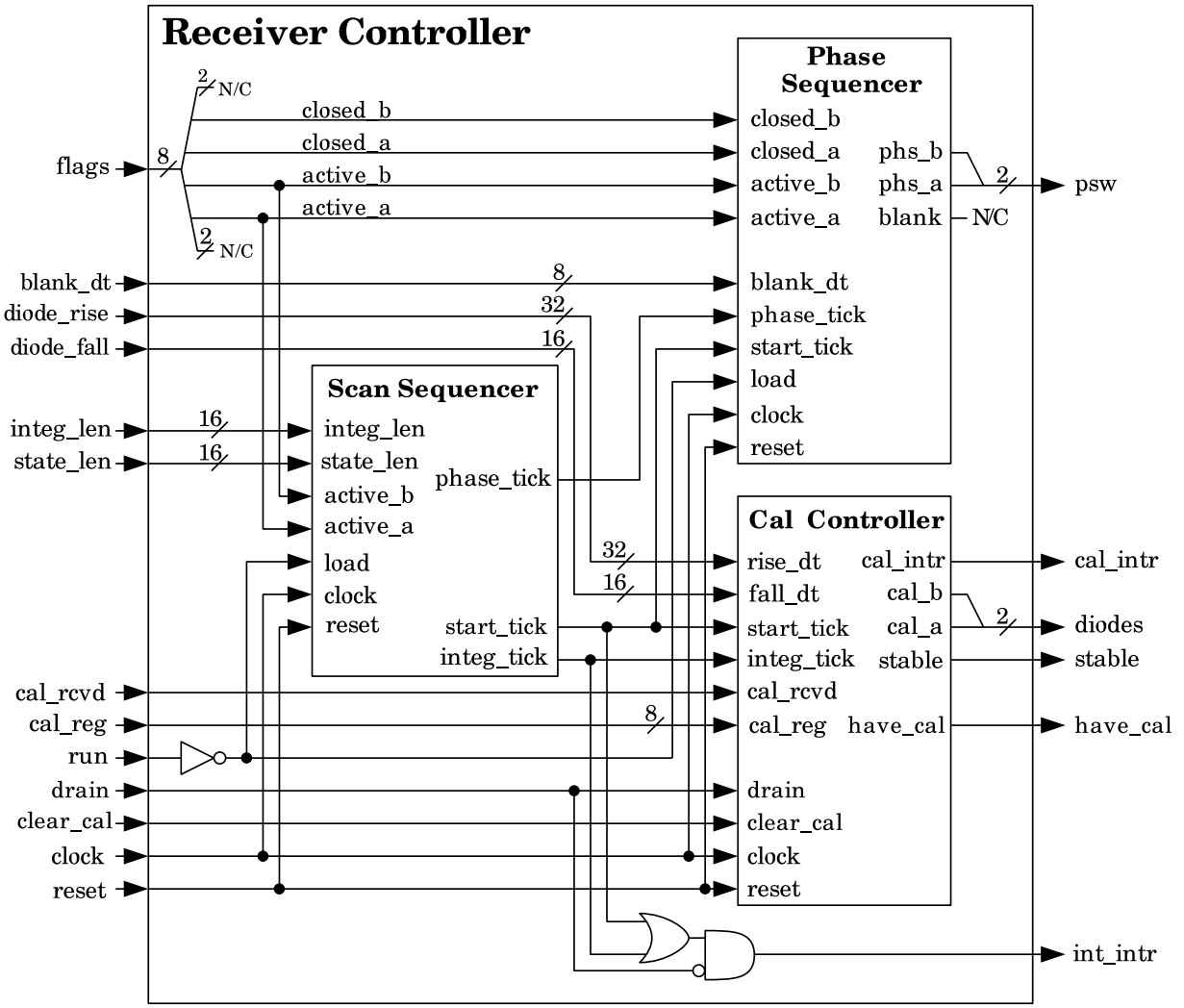

Figure 3.32: The Receiver Controller

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_rcvd` signals. The `cal_reg` signal brings in the contents of the `cal_diode_reg` register (see appendix A), and the `cal_rcvd` signal informs the *Receiver Controller* whenever the computer writes to this register.

The `drain` input is asserted by the *Scan Initiator* during the final integration of the current scan.

The `run` signal, which is asserted while a scan is running, is inverted to form a reconfiguration

signal, which is asserted for at least one clock cycle, between the end of one scan and the start of the next. During reconfiguration, the *Receiver Controller* stops switching the phase-switches and cal diodes. Subsequently, on the first rising clock edge after the `run` signal is reasserted to start a new scan, the frozen *Scan Sequencer*, *Phase Sequencer* and *Cal Controller* components all simultaneously proceed with the new scan.

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_diode_reg` register. Similarly, the *Phase Sequencer* uses the phase-switching tick to cycle the phase-switching control signals according to the flags in the `start_scan_reg` register.

In addition to controlling the cal-diodes, the *Cal Controller* outputs a cal-diode stability flag. This flag is included with the header information that the *Dispatch Controller* sends to the computer, and is used to tell the computer when the data that it is sending, are from an integration period during which the calibration diodes were in stable states throughout.

The *Cal Controller* also generates a `have_cal` output flag, which is used by the *Scan Initiator* to see when the first calibration configuration of a new scan has been received.

In addition to controlling the phase switches, the *Phase Sequencer* also generates a blanking signal, following each phase-switch transition. This signal is not used by the *Receiver Controller*, since it is designed for the other instance of the *Phase Sequencer*, 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 periodic pulses, of one clock cycle duration, at the ends of each of the various types of cycles within a scan. One periodic tick marks the end of each phase-switch state, another marks the end of each phase-switch cycle, and one marks the end of each integration period. In addition, there is a non-periodic tick which marks the start of all cycles at the start of a new scan. The *Scan Sequencer* also generates 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 3.33.

At the heart of the *Scan Sequencer* are the three *Metronome* components that generate the period integration, phase-switch cycle, and phase-switch state, tick outputs. The implementation of the *Metronome* components is described in section 3.4.6. Here it suffices to say that a *Metronome* component generates a pulse, of one clock cycle in duration, every time that it has synchronously counted `nstep` instances of its `step` input being asserted. This starts

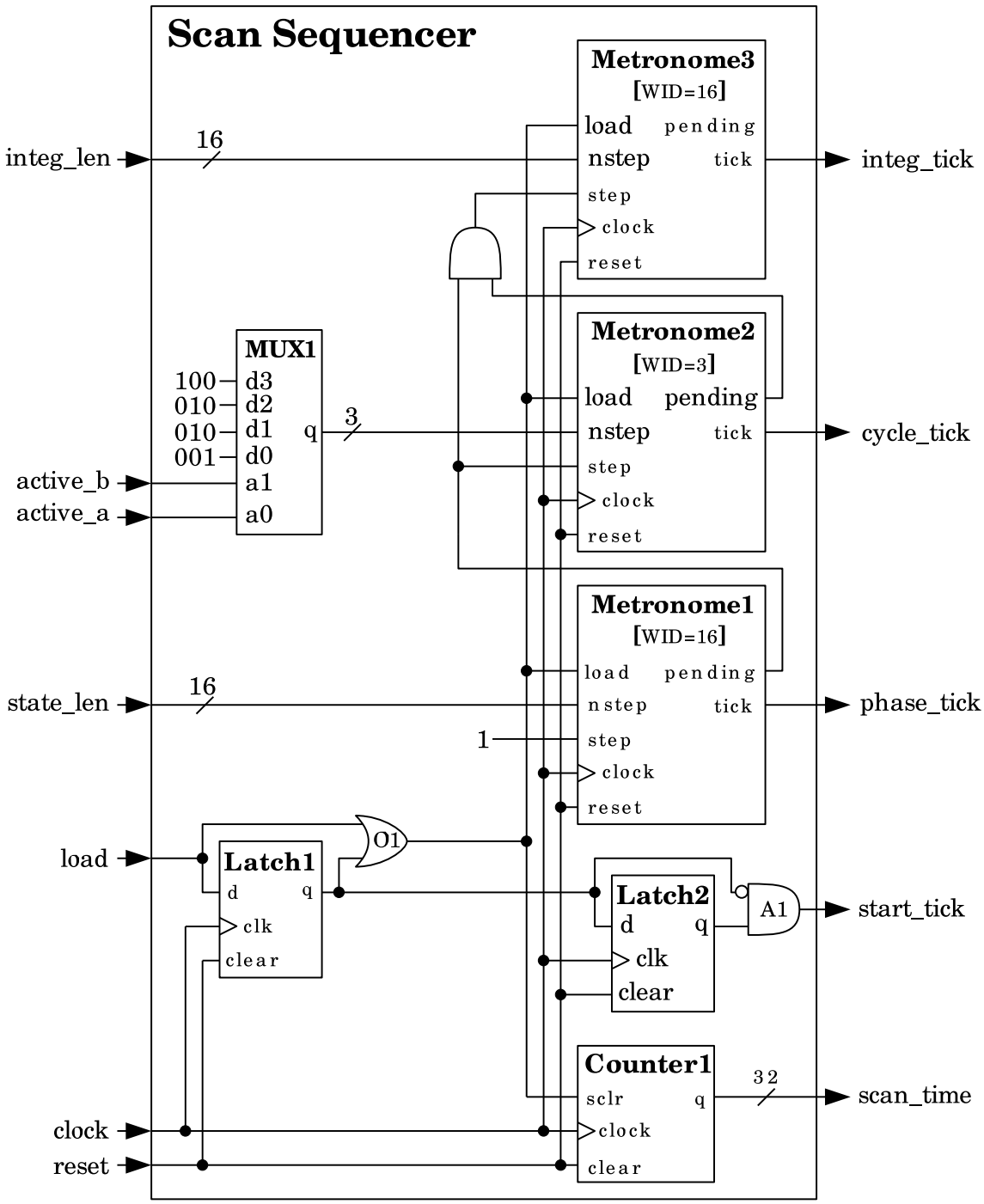

Figure 3.33: The Scan Sequencer

after the first `nstep` instances of `step` being asserted have been seen, after the synchronous `load` input has been used to load a repeat interval from the `nstep` input.

Immediately before the start of a new scan, the external *Scan Initiator* component asserts the *Scan Sequencer*'s `load` input for one or more clock cycles. This clears the scan time-stamp counter, resets and freezes the *Metronome* components, while loading new tick intervals via their `nstep` inputs, and arms `Latch2`, such that when the *Scan Initiator* returns the `load` input low, the `start_tick` output will go high for one clock cycle.

Since the rising edge of the `start_tick` signal is also the rising edge of the input of `Latch2`, `Latch1` is used to ensure that this occurs synchronously with the clock. OR gate `O1` then ensures that whereas the disabling and reconfiguration of the *Metronome* components occurs within one clock cycle of the `load` input going high, the same components get re-enabled just in time for `Metronome1` to start counting down at the rising edge that occurs at the end of the `start_tick` output pulse.

Since the `step` input of `Metronome1` is always asserted, and the interval of this *Metronome* is configured with the value of the `state_len` input, the `phase_tick` output outputs a pulse every `state_len` clock cycles, and the rising edge of the first of these pulses occurs `state_len` clock cycles after the rising edge of the `start_tick` pulse. Each of these pulses tell external *Phase Sequencer* components to advance to the next phase-switch state within current phase-switch cycle.

Multiplexer `MUX1` computes the number of phase-switch states per phase-switch cycle, according to the number of phase switches that are configured to be active, and *Metronome2* outputs a pulse at the `cycle_tick` output, every time that this many `phase_tick` pulses have been seen. In order that the tick at the `cycle_tick` output occur at the same time as the corresponding tick at the `phase_tick` output, the `pending` output of `Metronome1` is counted by `Metronome2`, rather than the `phase_tick` output. Since this goes high one clock cycle before the `phase_tick` output, both *Metronomes* count their events at the same time, and output tick's in sync with each other.

`Metronome3` outputs a pulse at the `integ_tick` output every time that `integ_len` ticks are generated at the `cycle_tick` output. These ticks mark the end of each integration period. Since the AND of the `pending` outputs of *Metronome*s 1 and 2 is high one clock cycle before both of these *Metronome*s output their ticks, this is used as the counter input by *Metronome3*, which thus outputs its ticks in sync with those of the other two *Metronome*s.

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 least-significant bits 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 the specified number of integration periods 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.

Following the start of the final integration of a scan, the queue of calibration-diode configurations is cleared by the `clear_cal` input-strobe, which goes high for one clock cycle. The resulting empty queue then prompts the *Cal Controller* to start sending the computer interrupts, to request new cal-diode configurations for the first few integrations of the pending scan. Once the first of these configurations has arrived, the `have_cal` output goes high, indicating that the *Cal Controller* is ready for the new scan.

On the same clock cycle that the queue is being cleared, the `drain` input goes high, and stays high until the next scan starts. This prevents the *Cal Controller* from attempting to start a new integration before the next scan starts.

At the start of each integration period, the *Cal Controller* is also responsible for generating a flag that indicates whether the calibration diodes have had time to settle by the start of that integration. This is computed by *Cal Switcher* components, which individually model the rise and fall characteristics of the two calibration diodes.

The implementation of the *Cal Controller* is shown in figure 3.34.

The queue of values written to the `cal_diode_reg` register, are held in `CCB FIFO1`. This FIFO receives one new value at a time, whenever the *Control Gateway* strobes the `cal_rcvd` 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, by being low, that there is still room for another value, the output of latch 1 again becomes asserted, and remains asserted until the next 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 `CCB FIFO1`, at the start of a new integration, different bits of this value are latched by `Event DCounter1`, and `Cal Switcher`s 1 and 2. In particular, the event down-counter reinitializes its count with the top 6 bits,

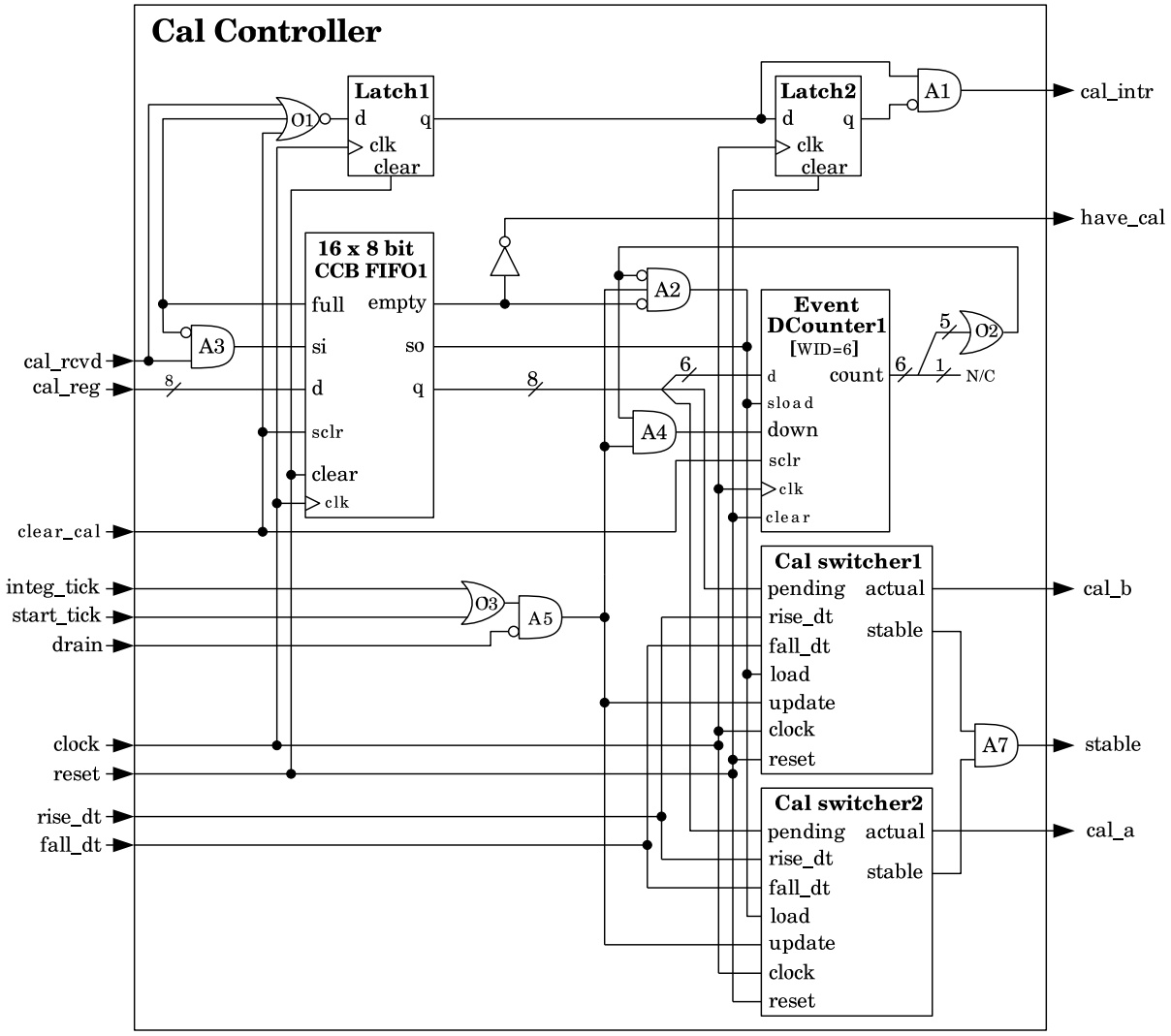

Figure 3.34: The Cal Controller

which contain an integer specifying for how many integrations the new calibration-diode states should 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 DCounter1` 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 `CCB 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 `CCB FIFO1` is found to be empty when a new value is needed, then 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 *Cal Switcher* components continue to drive the previous cal-diode states, until the next integration at which `CCB FIFO1` is found to be no longer empty.

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 `O3`, at the start of each integration period.

The implementation of the *Cal Switcher* components, is shown in figure 3.35.

Each *Cal Switcher* component is responsible for driving one calibration diode. At the beginning of each 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 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`

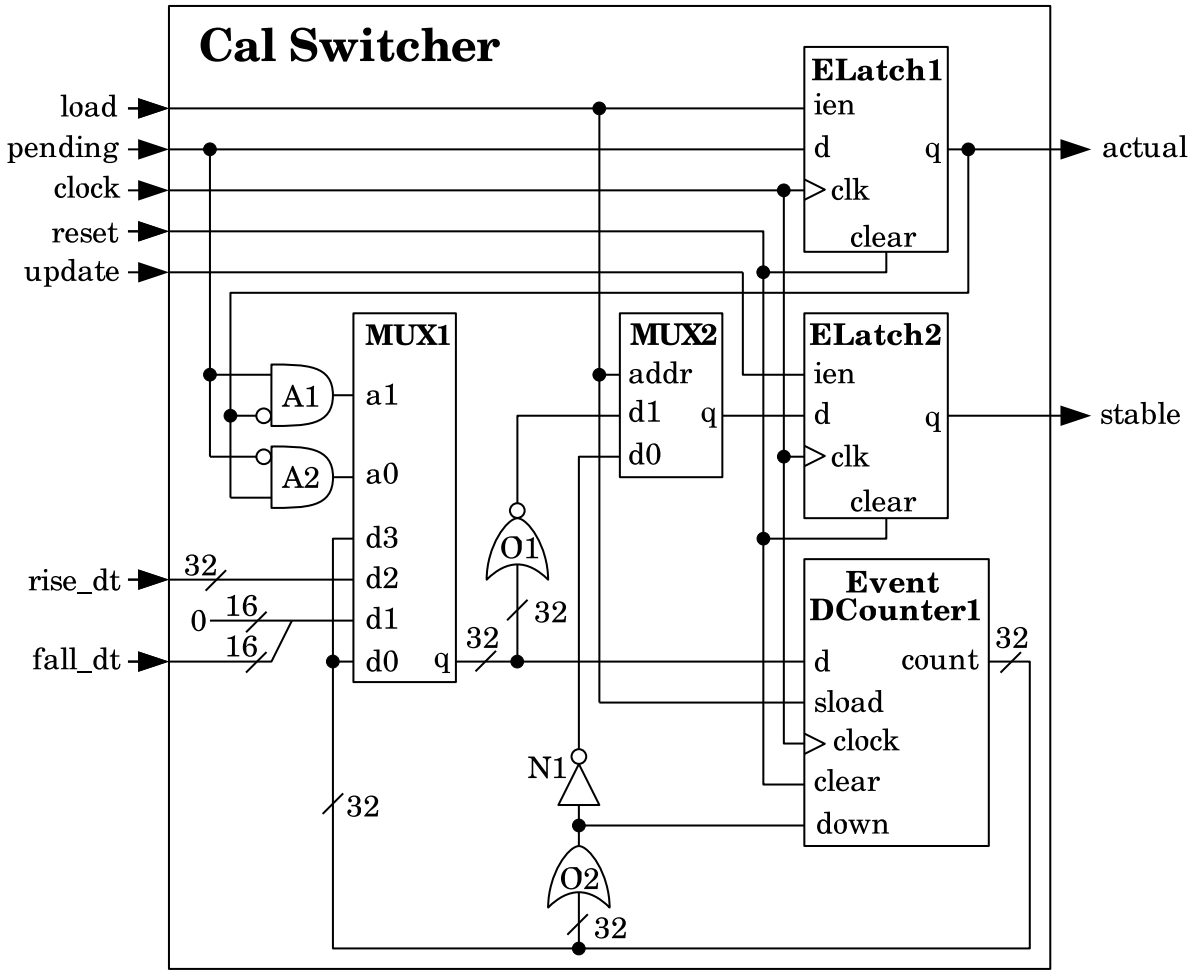

Figure 3.35: The Cal Switcher

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 one clock cycle in advance of whenever it is actually needed.

A new settling time is loaded into Event DCounter1 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 driven low, 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 DCounter1 is used directly. When a new value is being loaded, the output of the down-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

Phase Sequencer modules are used in two places. In the *Receiver Controller* a *Phase Sequencer* is used to generate the control signals that toggle the phase switches in the receiver. In the *Slave Controller* a *Phase Sequencer* is both used to direct samples into the appropriate phase-switch specific integration bin, and also to generate the blanking flag which causes samples to be discarded for a few samples after each phase-switch transition.

The implementation of the *Phase Sequencer* is shown in figure 3.36.

During a scan, the phase switches cycle through a fixed set of states, determined by the phase-switch configuration flags that the start_scan_reg register held when the scan started.

When two phase switches are configured to be actively switching, they take it in turns to switch, with just one of them switching state at the start of each new phase-switch state. When just one switch is configured to be actively switching, it switches state at the start of each new phase-switch state, while the other switch stays in the state that is dictated by the corresponding value of closed_a or closed_b. When neither switch is configured to be

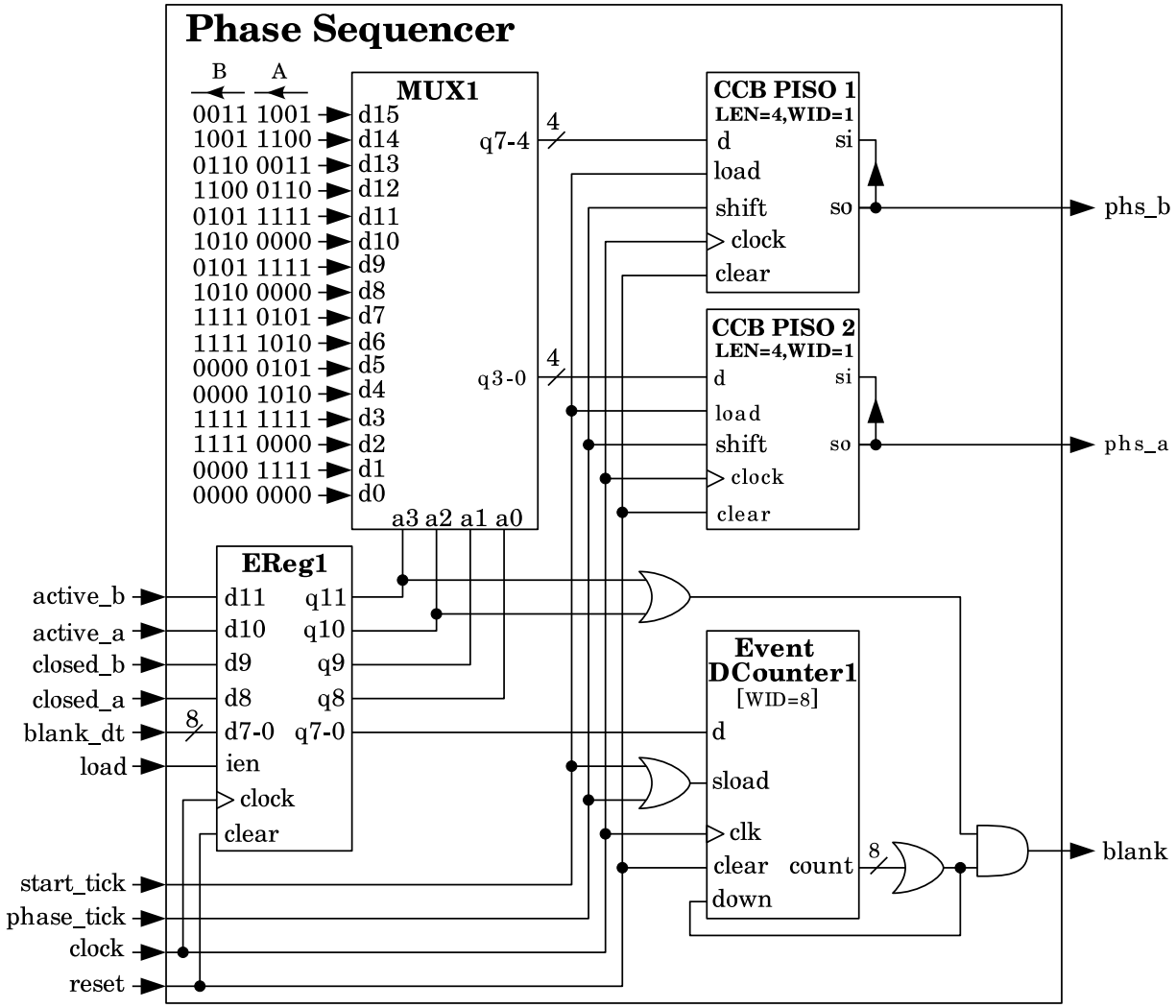

Figure 3.36: The Phase Sequencer

actively switching, then both switches remain in the states that are dictated by `closed_a` and `closed_b`.

Just prior to the start of a new scan, the *Scan Initiator* asserts the `load` input. This causes register `EReg1` to load phase-switching configuration flags from the *Scan Initiator*'s configuration-snapshot of the `start_scan_reg` register, and to load the desired phase-switch blanking time from the snapshot of the `blank_dt_reg` register. This configures the *Phase Sequencer* for the new scan.

Multiplexer `MUX1` implements a lookup table, indexed by the flags that configure which switches are active, and by the flags that specify which states they must have at the start of each phase-switch cycle. Each entry in this table contains 8 bits, of which the 4 most significant bits denote the 4 states that phase-switch B should repeatedly cycle through during the current scan, and the 4 least significant bits denote the parallel set of states that phase-switch B should simultaneously cycle through.

At the start of the first integration of a new scan, the `start_tick` input 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 rotates the sequence of states within the PISO, and presents each successive state at the serial outputs. The PISOs are clocked each time that the `phase_tick` input of the *Phase Sequencer* is asserted for one clock cycle.

At the start of each new phase-switch state, the `Event DCounter1` down-counter is loaded with the phase-switch blanking interval. This interval specifies how long it takes for the output of the receivers to settle, after either of the phase-switches changes state. The counter then counts down at the start of each clock cycle, until it reaches zero. While it is still counting down, if either switch is configured to be actively switching, the `blank` output is asserted, to indicate that ADC samples taken during this period should be discarded. If neither switch is configured to be actively switching, then there is no need to blank samples, so the output of the counter is ignored, and the `blank` output is driven low.

3.3.3 The Slave Controller

The *Slave Controller* generates the 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 3.37.

The *Scan Sequencer* and *Phase Sequencer* modules have already been documented in sections 3.3.2 and 3.3.2, respectively.

The `dump` output signal of the *Slave Controller*, is loaded from the `dump` configuration-bit of the `start_scan_reg` register, before each scan starts. This flag switches the slaves into dump mode. In this mode, one slave is selected for readout by the *Data Dispatcher*, and the

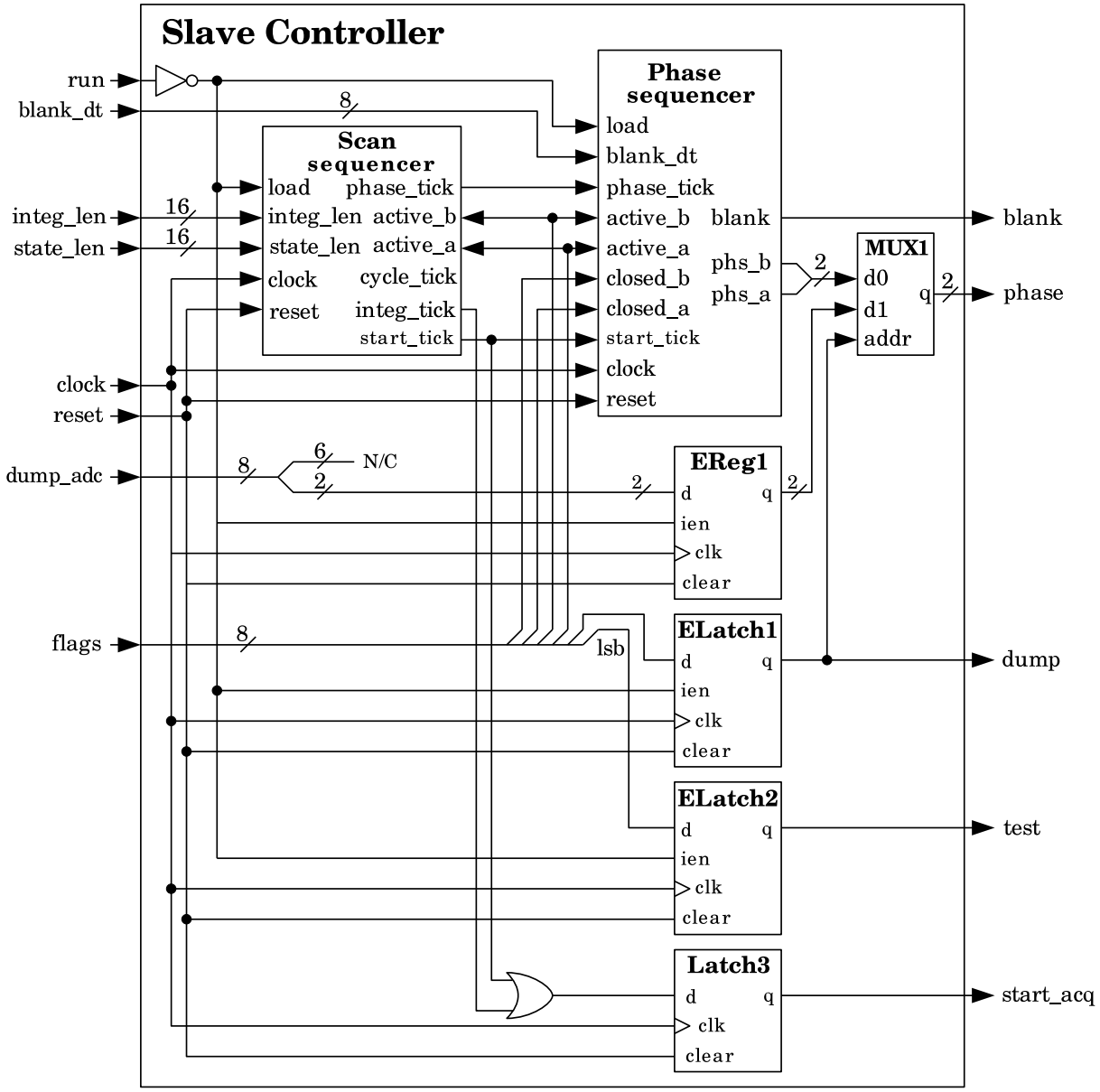

Figure 3.37: The Slave Controller

`phase` output, which is loaded from the `dump_adc_reg` register, tells the slave FPGAs which ADC's raw samples should be collected.

In normal integration mode, the `phase` output is used to tell the slave FPGAs which phase-switch bin to integrate the latest sample into, according to the phase-switch states output by the embedded *Phase Sequencer*.

The `blank` output of the *Phase Sequencer* is forwarded to the slaves, to command them to drop samples during phase-switch transitions.

The `test` output signal is loaded from the `test` configuration-bit of the `start_scan_reg` register. When asserted, this output tells all of the slaves to use internally generated pseudo-random samples instead of 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. Note that the pipeline delay of `latch3` matches that of the *Phase Sequencer* component, and thus causes the `start_acq` output to go high at the same time as the configuration of the first sample is driven onto the `phase` and `blank` outputs.

The `test` and `dump` outputs are loaded from the scan-flags register on the first rising clock edge that follows the `run` signal going low. Thus they aren't presented until the last sample of the previous scan has been read from the slaves by the *Data Dispatcher*, but they are presented before the `start_acq` output starts the collection of data for the next scan.

3.3.4 The Dispatch Controller

The *Dispatch Controller* controls the *Data Dispatcher* module. It tells the *Data Dispatcher* 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 3.38.

There are two types of information that are passed to the *Data Dispatcher*, for inclusion in the header.

1. **The configuration parameters of the current scan.**

   These are parameters that remain constant throughout the duration of each scan. They are thus loaded from the configuration snapshot registers in the *Scan Initiator*, immediately before each new scan.

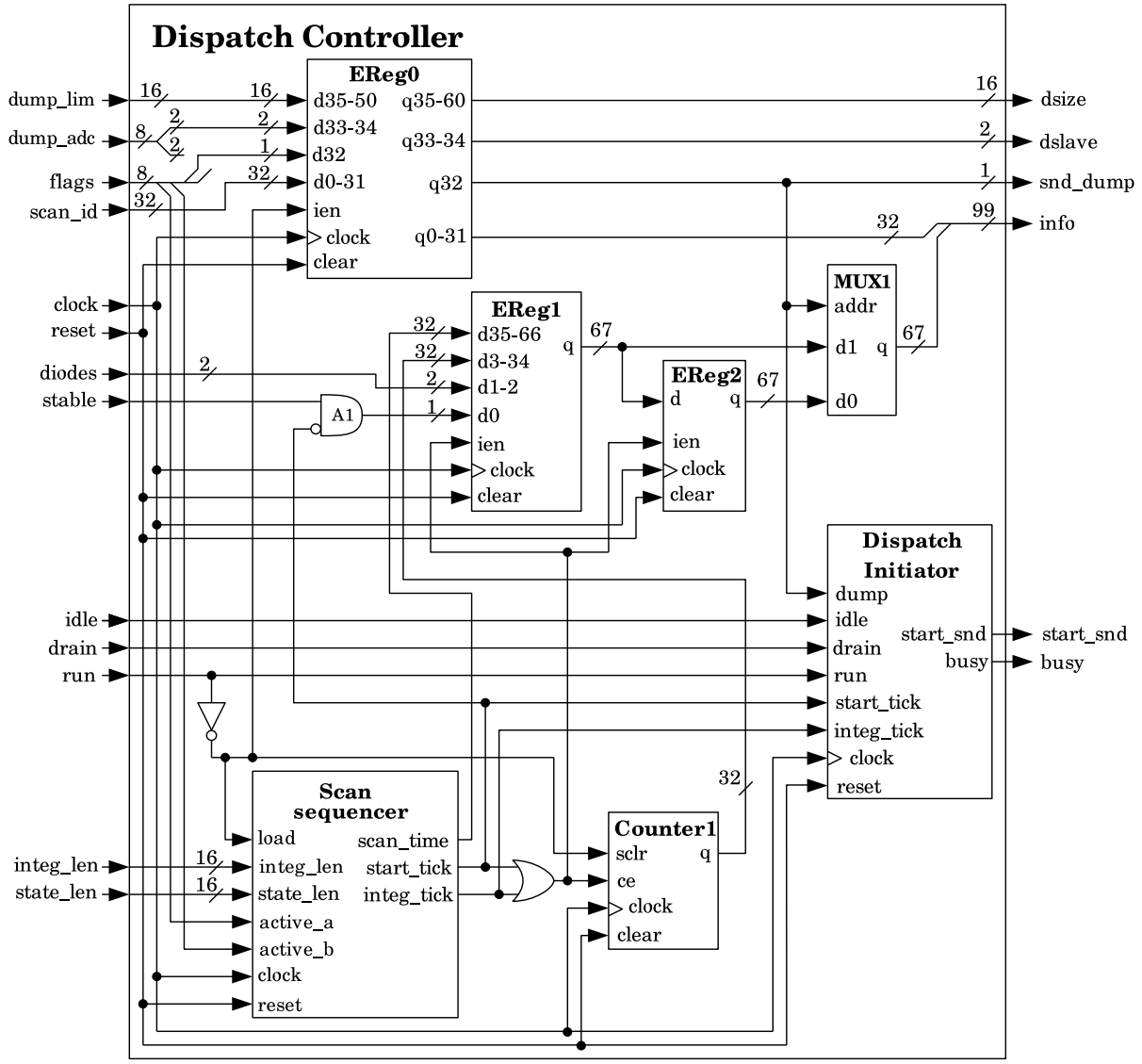

Figure 3.38: The Dispatch Controller

2. **Integration-specific parameters.**

   These are parameters that describe important characteristics of each integration. This includes the sequential number of the integration within the current scan, the time-stamp of the integration, the commanded states of the calibration diodes, and a stability flag which indicates whether the calibration diodes have had a chance to stably reach their commanded states.

   At the start of each integration period, the current values of these parameters are loaded into register `EReg1`, while the corresponding values that pertained to the start of the previous integration period, are transfered from this register to register `EReg2`.

   In dump mode, the *Data Dispatcher* constructs a header, and starts collecting data immediately, at the start of the new integration period. So in this mode, multiplexer `MUX1` sends the contents of `EReg1` to the *Data Dispatcher*, for inclusion in the header.

   In normal integration mode, however, the *Data Dispatcher* has to wait for the slaves to integrate samples for the duration of the integration period, before constructing a header and reading out the integrated data from the slaves at the start of the next integration period. Thus in this mode, `MUX1` sends the contents of register `EReg2` to the *Data Dispatcher* for inclusion in the header.

   Note that the purpose of AND gate `A1` is to always flag the first integration of a new scan. In particular, this accommodates the case of scans in which the phase-switches remain quiescent during the scan. In such cases, no phase-switch blanking is performed, because once the phase-switches have had time to settle into the quiescent state, at the start of the scan, they remain in that state throughput the scan. Although, in principle, phase-switch blanking could alternatively be temporarily enabled during the first phase-switch cycle, this would mean that the first integration would have a shorter effective integration time than the other integrations of the scan, which would make subsequent analysis more difficult.

   In scans where the phase-switches are actively switching, there is also a reason to flag the first integration. When the phase-switches are configured to be switching, optimal rejection of common-mode signals requires that the phase switches behave as similarly as possible during each phase-switch cycle of each integration. Given that the phase switch behavior preceding the first phase-switch cycle of the first integration, differs from that preceding the other phase switch cycles, it makes sense not to use at least the first phase-switch cycle, and since one integration period happens to be the smallest amount of data that can be flagged, it thus makes sense to flag the first integration period of each scan.

`Counter1` keeps a record of the sequential number of the current integration period. It is zeroed during the period immediately before the start of each new scan, when the `run` input is deasserted, and then counts up by one, first when the embedded *Scan Sequencer* outputs the `start_tick` to mark the start of the first integration, and thereafter at the end of each integration period, as marked by the `integ_tick` signal. Since the output of the counter is latched into `EReg1` during the same clock edge that increments the counter, `EReg1` loads the

value before the count is incremented, and thus, even though the counter is first incremented at the start of the first integration, the first integration is numbered 0, rather than 1.

As already mentioned previously, whereas in dump mode, the *Data Dispatcher* should construct a header and start streaming new raw data to the computer at the start of each integration period, in normal integration mode, it should wait until the start of the following integration period to stream the data that were integrated during the current integration period. Controlling this is the responsibility of the *Dispatch Initiator*.

The Dispatch Initiator

The *Dispatch Initiator* has a couple of responsibilities delegated to it by its parent *Dispatch Controller* module. It asserts the `start_snd` output at the appropriate times, to initiate data collection and dispatches to the computer, taking care not to attempt to do this when the *Data Dispatcher* is still dispatching data from a previous integration period, and it asserts the `busy` output, from the moment that a new scan starts, until the data of the final integration of the scan have been dispatched. It is told which integration is to be the final one, by the `drain` input going high.

The *Scan Initiator* is implemented as a *Finite State Machine*, which implements the state diagram of figure 3.39.

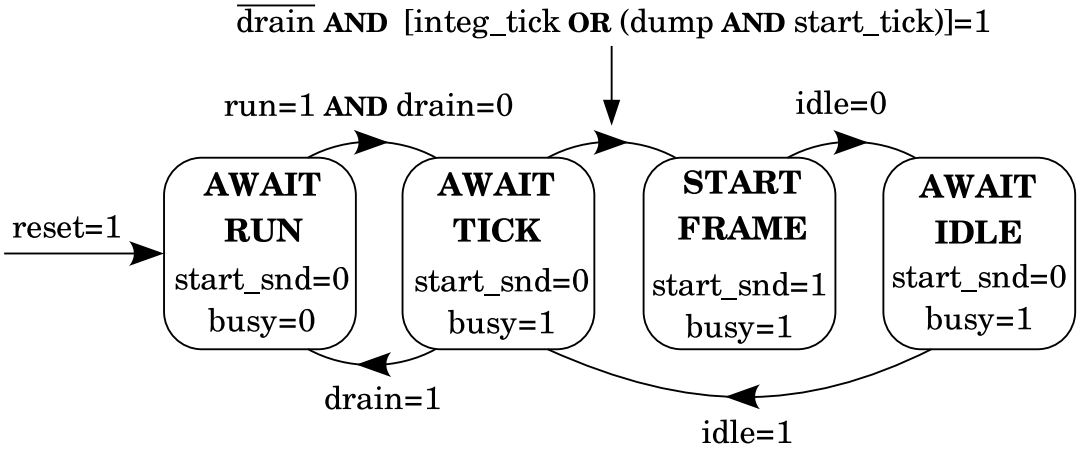

Figure 3.39: The state diagram of the Dispatch Initiator FSM

The meanings of the states are as follows.

- `AWAIT RUN`

  After a reset, and in the period between the end of one scan and the start of the next, the state machine sits in the `AWAIT RUN` state. In this state the `busy` output is held

low, to indicate that the *Dispatch Controller* isn't running a scan, and the `start_snd` output is held low, since no data should be dispatched to the computer while no scan is running.

It stays in this state while the `run` input is low, and advances to the `AWAIT TICK` state when the `run` input goes high.

Note that whenever the state machine is not in the `AWAIT RUN` state, the other states all assert the `busy` output, to indicate that the *Dispatch Controller* is busy acquiring and dispatching data of the current scan.

- `AWAIT TICK`

  The state machine sits in the `AWAIT TICK` state whenever the *Data Dispatcher* isn't in the process of collecting and sending data to the computer. It remains in this state until it is time to start either collecting and dispatching the dump-mode data of a new integration period, or time to start collecting and dispatching the integration-mode data of an integration period that has just finished. However, if the `drain` input either goes high while it is waiting in this state, or was already high when this state was entered, then the `busy` output is deasserted, to indicate that the scan has ended, and the state machine returns to the `AWAIT RUN` state, to await the start of the next scan.

  If the `drain` signal doesn't go high before the integration boundary when the *Data Dispatcher* should start collecting and dispatching a new frame of data, then the state machine advances to the `START FRAME` state.

- `START FRAME`

  In the `START FRAME` state, the *Dispatch Initiator* asserts the `start_snd` output, to tell the *Data Dispatcher* to start collecting and dispatching a new frame of data. It remains in this state, holding the `start_snd` output high, until the *Data Dispatcher* acknowledges the receipt of the request by deasserting the `idle` input.

  Note that in this state the state of the `drain` input is ignored, since once a dispatch has been started, the scan can't end until that dispatch ends.

  Once the `idle` input goes low, the state machine advances to the `AWAIT IDLE` state.

- `AWAIT IDLE`

  In this state, the `start_snd` output is deasserted, since the deassertion of the `idle` input by the *Data Dispatcher* indicates that the command was received. The state machine remains in this state until the *Data Dispatcher* subsequently asserts the `idle` input again, to indicate that it has finished collecting and sending the data from the latest integration period.

  Note that again, during this time, the `drain` input is ignored, since a scan can't end until the ongoing dispatch has ended.

  Once the `idle` input goes high, the state machine returns to the `AWAIT TICK` state, in which it is allowed to terminate the scan if the `drain` input is high, or wait for the appropriate moment to start collecting and dispatching data from another integration period.

Each change in the state of the state machine is performed at the start of a clock cycle. Thus, given that the `start_snd` and `busy` signals are uniquely determined by the value of the `state` variable, they also change synchronously with the clock, and thus don't need to be latched.

Implementing a state machine is less cumbersome and error-prone in VHDL, than with a schematic, so the VHDL code in figure 3.40 is used to implement the *Data Initiator*.

Note that the displayed VHDL code has had most of the comments in the actual code, either significantly shortened, or eliminated, to conserve space. For more details please see the actual code.

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 immediately follows the rising edge of the FPGA clock, and whose duration is a single FPGA clock cycle. The circuit shown in figure 3.41 does this.

Since the 1PPS input signal isn't synchronous with the FPGA clock, latch 1 is used both to 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 $\overline{\texttt{q}}$ 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,

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

entity dispatch_initiator is
  port (
    clock, reset, drain, run, idle, integ_tick, start_tick, dump : in std_logic;
    start_snd, busy : out std_logic);
end dispatch_initiator;

architecture dispatch_initiator_arch of dispatch_initiator is
  type states is (AWAIT_RUN, AWAIT_TICK, START_FRAME, AWAIT_IDLE);
  signal state, next_state : states;
begin
  process (clock, reset)  -- Synchronously advance the state machine.
  begin
    if reset = '1' then                     -- An asynchronous reset.
      state <= AWAIT_RUN;
    elsif clock'event and clock = '1' then  -- The rising edge of the clock.
      state <= next_state;
    end if;
  end process;

  process (state, drain, run, idle, integ_tick, start_tick, dump)  -- Set the outputs of each state.
  begin
    case state is
      when AWAIT_RUN =>   -- Await the start of a new scan.
        busy <= '0';
        start_snd <= '0';
        if run='1' and drain='0' then
          next_state <= AWAIT_TICK;
        else
          next_state <= AWAIT_RUN;
        end if;
      when AWAIT_TICK =>           -- Wait to start a new dispatch.
        busy <= '1';
        start_snd <= '0';
        if drain='1' then          -- End the scan?
          next_state <= AWAIT_RUN;
        elsif (integ_tick='1' or (dump='1' and start_tick='1')) then
          next_state <= START_FRAME;
        else
          next_state <= AWAIT_TICK;
        end if;
      when START_FRAME => -- Tell Data Dispatcher to start a new frame.
        busy <= '1';
        start_snd <= '1';
        if idle='0' then
          next_state <= AWAIT_IDLE;
        else
          next_state <= START_FRAME;
        end if;
      when AWAIT_IDLE =>  -- Wait for the Data Dispatcher to finish.
        busy <= '1';
        start_snd <= '0';
        if idle='1' then
          next_state <= AWAIT_TICK;
        else
          next_state <= AWAIT_IDLE;
        end if
      when others =>      -- Handle illegal states.
        busy <= '0';
        start_snd <= '0';
        next_state <= AWAIT_RUN;
    end case;
  end process;
end dispatch_initiator_arch;
```

Figure 3.40: The VHDL implementation of the Dispatch Initiator

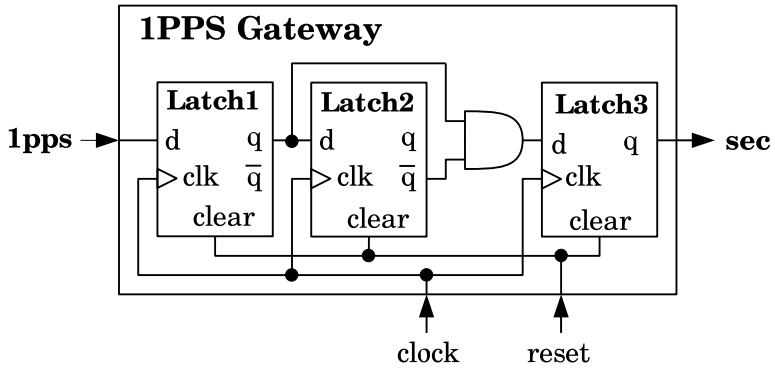

Figure 3.41: The 1PPS Gateway

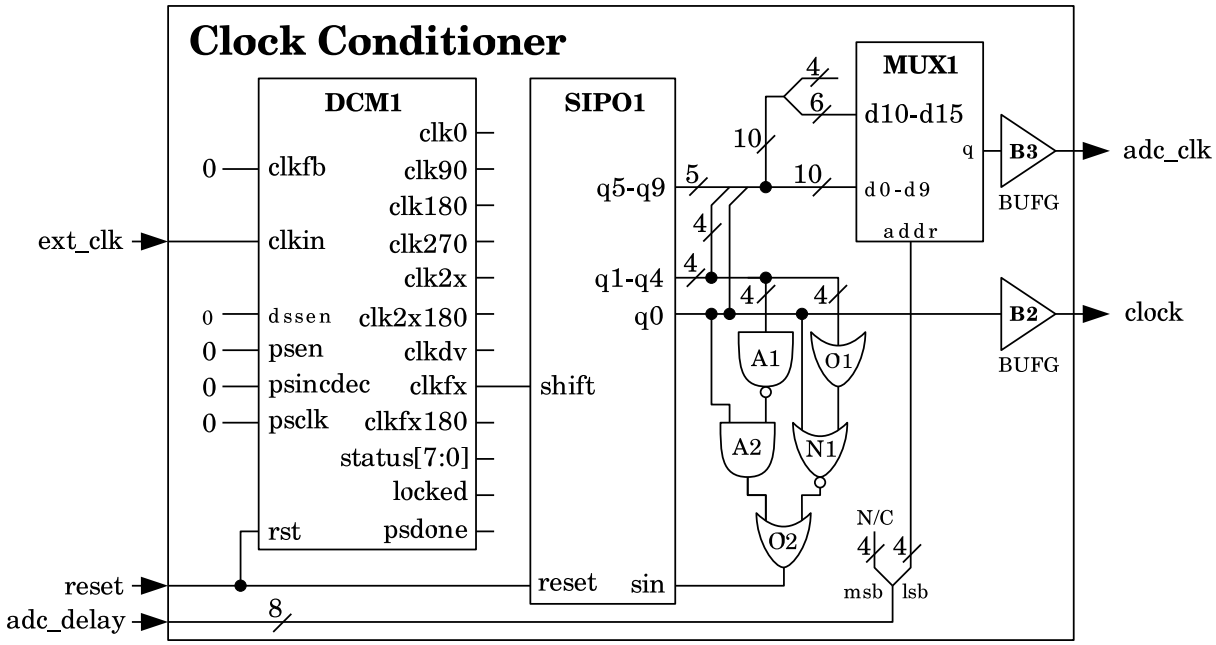

Figure 3.42: The Clock Conditioner

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 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 3.42.

Note that in the top-level master FPGA diagram, the `ext_clk` input of the *Clock Conditioner* comes from a global clock pin, via a global-clock input buffer. Similarly, buffers `B2` and `B3` are internal global clock buffers, which connect the two output clock signals 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.

- `CLKIN_PERIOD = 100.0`

  This tells the DCM the period of the clock input, expressed in floating-point nanoseconds.

- `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,

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

where as per convention, 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 3.42, 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 accommodate 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 interpreted 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 an 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. This can't be done reliably by AND'ing the clock signal with an enabling signal, since a synchronously generated enabling signal will change state just after the rising edge of the clock, and thus fail to disable the clock in time. A better way is to use a register that latches a new value on every clock cycle, together with a 2-input multiplexer which feeds the input of the register either with the required new value, when enabled, or with the previous value of the register, when not. In VHDL, the equivalent code to do this is shown in figure 3.43.

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

entity ccb_elatch is
  port (
    d    : in  std_logic;     -- The bit to be latched.
    q    : out std_logic;     -- The contents of the D-type latch.
    clk  : in  std_logic;     -- The clock.
    ce   : in  std_logic;     -- The active-high clock-enable signal.
    aclr : in  std_logic);    -- The asynchronous active-high clear input.
end ccb_elatch;

architecture ccb_elatch_arch of ccb_elatch is
begin
  ccb_elatch_proc: process (clk, aclr)          -- The latch assignment process.
  begin
    if aclr = '1' then                          -- Asynchronous reset?
      q <= (others => '0');
    elsif clk'event and clk='1' and ce='1' then  -- Is the clock enabled at the rising clock edge?
      q <= d;
    end if;
  end process ccb_elatch_proc;
end ccb_elatch_arch;
```

Figure 3.43: A D-type latch with a synchronous input-enable input

3.4.2 The EReg component

The *EReg* component is a multi-bit D-type register with a clock-enable input. The implementation of such a register, with a configurable number of bits, is shown in figure 3.44.

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

entity ccb_ereg is
  generic (WID : integer := 2);  -- The number of bits in the register.
  port (
    d    : in  std_logic_vector(WID-1 downto 0);  -- The bits to be latched.
    q    : out std_logic_vector(WID-1 downto 0);  -- The register contents.
    clk  : in  std_logic;                         -- The clock.
    ce   : in  std_logic;                         -- Active-high clock-enable.
    aclr : in  std_logic);                        -- Async active-high clear.
end ccb_ereg;

architecture ccb_ereg_arch of ccb_ereg is
begin
  ccb_ereg_proc: process (clk, aclr)            -- Register assignment process.
  begin
    if aclr = '1' then                          -- Asynchronous reset.
      q <= (others => '0');
    elsif clk'event and clk='1' and ce='1' then  -- Is the clock enabled at the rising clock edge?
      q <= d;
    end if;
  end process ccb_ereg_proc;
end ccb_ereg_arch;
```

Figure 3.44: The VHDL implementation of the ccb_ereg component

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. *CCB PISO* components add the possibility of holding the contents of the PISO unchanged. Instead of having a single input that either causes the PISO to load new parallel data, or shift the contents of the PISO, *CCB PISO*s have separate `load`-enable and `shift`-enable inputs. As in a conventional PISO, these inputs are heeded at the active edge of the clock, but unlike a normal PISO, the contents of the PISO remain unchanged, unless at least one of them is asserted.

A single node of the CCB PISO component is shown in figure 3.45.

The number of bits that are either serially shifted into the `si` input and shifted out of the `so` output, or that are parallel-loaded via the `d` input, is set by the `WID` parameter.

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, the value of the PISO node remains unchanged.

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

entity ccb_piso_node is
  generic (WID: integer := 16);                    -- The width of the node.
  Port ( d : in std_logic_vector(WID-1 downto 0);  -- The parallel data input.
         si : in std_logic_vector(WID-1 downto 0); -- Serial data input.
         load : in std_logic;                      -- When high, load 'd'.
         shift : in std_logic;                     -- When high, load 'si'.
         clock, clear : in std_logic;              -- Async clock and clear.
         so : out std_logic_vector(WID-1 downto 0)); -- The serial output.
end ccb_piso_node;

architecture ccb_piso_node_arch of ccb_piso_node is
begin
  process(clear, clock)
    variable tmp: std_logic_vector(WID-1 downto 0);
  begin
    if clear = '1' then                  -- Async clear?
      tmp := (others => '0');
    elsif clock='1' and clock'event then  -- Rising edge of clock?
      if load='1' then                   -- Synchronously load parallel data.
        tmp := d;
      elsif shift='1' then               -- Synchronously load serial data.
        tmp := si;
      end if;
    end if;
    so <= tmp;                           -- Output the contents of the node.
  end process;
end ccb_piso_node_arch;
```

Figure 3.45: One node of a CCB PISO component

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

  When just the `shift` input is asserted, the contents of the PISO node are replaced with the output value of the previous node of the PISO, taken from `si`.

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

  Whenever the `load` input is asserted, regardless of the state of the `shift` input, the contents of the PISO node are replaced with the value of the `d` input.

The CCB PISO component is constructed by stringing together a configurable number of *CCB PISO Node*s in a chain, as shown in figure 3.46. In this code, note that the `LEN` parameter specifies how many nodes are chained to form the PISO, and the `WID` parameter specifies the bit-width of each of these nodes.

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.

```
library ieee;
use ieee.std_logic_1164.all;
use work.ccb_schematic_pisos.all;  -- The package that contains ccb_piso_node.

entity ccb_piso is
  generic (LEN : integer; WID : integer);
  port (
    d : in std_logic_vector((LEN*WID)-1 downto 0);  -- The parallel inputs.
    load : in std_logic;                            -- When high, load d.
    shift : in std_logic;                           -- Shift when high.
    clock, clear : in std_logic;                    -- Async clock and clear.
    si : in  std_logic_vector(WID-1 downto 0);      -- Serial data input.
    so : out std_logic_vector(WID-1 downto 0));     -- Serial data output.
end ccb_piso;

architecture ccb_piso_arch of ccb_piso is
  --
  -- A signal vector for chaining the so outputs of each node to the si input
  -- of its successor. The first and last nodes are connected to the
  -- corresponding ports of the PISO, for external chaining.
  --
  type serial_vector is array(LEN downto 0) of std_logic_vector(WID-1 downto 0);
  signal serial_links : serial_vector;
begin
  --
  -- Create the LEN nodes, linked together such that serial data move
  -- from the si input of node(LEN-1) to the so output of node(0).
  --
  piso_nodes: for i in LEN-1 downto 0 generate
    piso_node : ccb_piso_node generic map (WID => WID) port map (
      si => serial_links(i+1), so => serial_links(i), load => load,
      shift => shift, clock => clock, clear => clear,
      d => d(WID*(i+1)-1 downto WID*i));
  end generate piso_nodes;
  --
  -- Connect the si input of the earliest node in the chain, to the si
  -- input of the PISO.
  --
  serial_links(LEN) <= si;
  --
  -- Connect the so output of the final node in the PISO, to the so
  -- output of the PISO.
  --
  so <= serial_links(0);
end ccb_piso_arch;
```

Figure 3.46: A CCB PISO of configurable length and width

- 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.

The *Event Counter* component, shown in figure 3.47, shows the inputs and outputs of this counter.

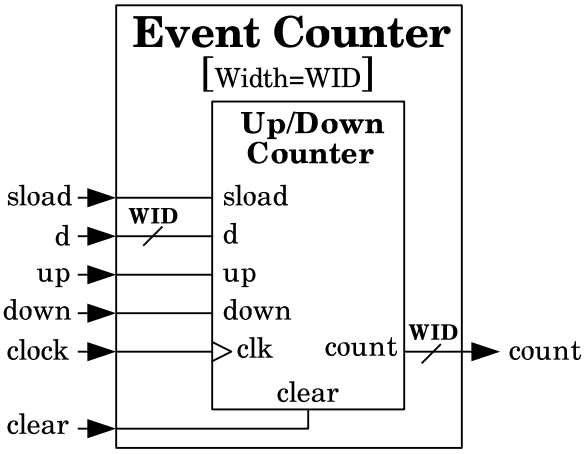

Figure 3.47: An up/down counter with synchronous parallel load capability

The VHDL code shown in figure 3.48 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 Event DCounter component

The *Event DCounter* component is essentially equivalent to an *Event Counter* component that has its `up` input tied permanently high. In other words this component only counts down. Note that `sclr` input is a synchronous clear input. It's VHDL implementation is shown in figure 3.49.

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

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

architecture event_counter_arch of event_counter is
  signal tmp : std_logic_vector(WID-1 downto 0);
begin  -- event_counter_arch
  process(clk, clear)
  begin  -- process
    if(clear='1') then
      tmp <= (others => '0');
    elsif(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 3.48: A VHDL implementation of the Event Counter component

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

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

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

Figure 3.49: A VHDL implementation of the Event DCounter component

3.4.6 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 `nstep=1` events. Metronomes of specific widths are derived from the implementation in figure 3.50, in which the bit-width is parameterized by the `WID` generic parameter.

After firmware-resets, *Metronome* components hold their `tick` and `pending` outputs low until configured with a finite value of `nstep`. Configuration is performed by asserting the synchronous `load` input for one clock cycle. This loads the value of `nstep`, which specifies the number of events to count between generating pulses at the `tick` output. The first of these pulses, which last one clock cycle each, is generated after `nstep` initial events have been counted, and thereafter after every subsequent `nstep` events are counted. There is a pipeline delay of one clock cycle, such that an external synchronous input will see the pulse one clock cycle after the event that triggered it. The `pending` output provides forewarning that the next time that `step=1`, a tick will be generated.

The implementation is somewhat similar to a state machine, in which the individual states are effectively encoded in the value of a down-counter. This down counter repeatedly counts down from the most recently loaded value of `nstep`, down to 1, counting down by one each time that the `step` input is high at the rising edge of the clock. Once the counter reaches 1, the `pending` output is asynchronously asserted, to indicate that a new tick will be output when a new event is seen, and thus the next time that the `step` input is asserted, the `tick` output is driven high for one clock cycle, the counter is reloaded with the previously loaded value of `nstep`, and this causes the `pending` output to return low.

Given that during normal operation, the counter never counts down to zero, a count value of zero is interpreted as indicating that the Metronome is un-configured and that it shouldn't generate any ticks. This is the state that the Metronome is initialized to by firmware resets, meaning that after a reset, a Metronome won't generate any ticks until after it has been explicitly configured, by synchronously asserting its `load` input, while there is a non-zero value at its `nstep` input. Subsequently, a Metronome can be returned to the idle state, by synchronously asserting the `load` input, while `nstep` is specified as zero.

3.4.7 The CCB FIFO component

The conventional RAM-based FIFOs that Xilinx provides aren't very convenient to use in some situations. For example, when one writes an entry into an empty FIFO of this type, although the `empty` output signal of the FIFO becomes asserted one clock cycle later, one then has to assert the output-enable input for another clock cycle, before the new occupant of the FIFO is presented at the output. In other words, to read an entry out of the FIFO, one has to first assert a read-strobe for one clock cycle, then wait one clock cycle for the requested output to appear at the output of the FIFO.

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

entity metronome is
  generic(WID: integer := 16);
  port(nstep: in std_logic_vector(WID-1 downto 0);
       load, step, clock, reset : in std_logic;
       tick, pending: out std_logic);
end metronome;

architecture metronome_arch of metronome is
  constant ZERO_COUNT : std_logic_vector(WID-1 downto 0) := (others => '0');
  constant UNITY_COUNT : std_logic_vector(WID-1 downto 0) := ZERO_COUNT(WID-1 downto 1) & '1';
  --
  -- This reload value of the metronome's down-counter.
  --
  signal reload_value : std_logic_vector(WID-1 downto 0);
  --
  -- The current value of the metronome countdown.
  --
  signal count_value : std_logic_vector(WID-1 downto 0);
  --
  -- The values to store count_value, reload_value and tick at the next rising edge of the clock.
  --
  signal next_count_value : std_logic_vector(WID-1 downto 0);
  signal next_reload_value : std_logic_vector(WID-1 downto 0);
  signal next_tick : std_logic;
begin
  --
  -- A new tick is pending on the next event after the count reaches unity.
  --
  pending <= '1' when count_value=UNITY_COUNT else '0';

  -------------------------------------------------------------------------
  -- The sequential part of the metronome, containing the only registers.
  -------------------------------------------------------------------------
  metronome_sequential_proc: process(clock, reset)
  begin
    if reset='1' then                        -- Active-high asynchronous reset.
      reload_value <= ZERO_COUNT;
      count_value <= ZERO_COUNT;
      tick <= '0';
    elsif clock'event and clock = '1' then  -- The rising edge of the clock.
      count_value <= next_count_value;
      reload_value <= next_reload_value;
      tick <= next_tick;
    end if;
  end process;

  -------------------------------------------------------------------------
  -- The combinational part of the Metronome. No registers should be
  -- infered in the following process.
  -------------------------------------------------------------------------
  metronome_combinational_proc: process (load, reset, step, nstep, count_value, reload_value)
  begin
    if reset='1' or (load='1' and nstep=ZERO_COUNT) then  -- Unconfigure?
      next_count_value <= ZERO_COUNT;
      next_reload_value <= ZERO_COUNT;
      next_tick <= '0';
    elsif load='1' then                      -- Reconfigure the metronome?
      next_count_value <= nstep;
      next_reload_value <= nstep;
      next_tick <= '0';
    elsif step='0' then                      -- No event to be counted.
      next_count_value <= count_value;
      next_reload_value <= reload_value;
      next_tick <= '0';
    elsif count_value=UNITY_COUNT then       -- Generate a tick and restart
                                             -- the countdown.
      next_count_value <= reload_value;
      next_reload_value <= reload_value;
      next_tick <= '1';
    elsif count_value=ZERO_COUNT then        -- Metronome disabled?
      next_count_value <= ZERO_COUNT;
      next_reload_value <= ZERO_COUNT;
      next_tick <= '0';
    else                                     -- Count a new event.
      next_count_value <= count_value - 1;
      next_reload_value <= reload_value;
      next_tick <= '0';
    end if;
  end process;
end metronome_arch;
```

Figure 3.50: The VHDL implementation of the Metronome component

An alternative approach to implementing a FIFO, is to base it on shift-registers instead of RAM. Whereas in a RAM based FIFO, one uses read and write addresses to read and write elements from a circular buffer, and thus is required to use explicit read and write strobes; in a shift-register based FIFO, one always presents the contents of the final register of the shift-register at the output, without the need for a read strobe, and one simply shifts the contents of the other registers down by one register, each time that the shift-out input is strobed. In other words, in a shift-register based FIFO, the first element written to an empty FIFO, automatically appears at the output, one clock cycle after it is written into the FIFO, and instead of having to strobe an output-enable input one clock cycle before reading the output, one reads the output on the same clock cycle as strobing a shift-out input to discard it from the FIFO and replace it with the next value in the FIFO.

Also, whereas the `empty` output signal of a RAM based FIFO, goes low once the only occupant of an almost empty FIFO is presented at the output, the `empty` output of a shift-register based FIFO goes low, once the final entry in the FIFO has been read and discarded from the output. Thus, whenever the `empty` output of a shift-register based FIFO is high, this means that the output is currently presenting a valid entry, whereas for a RAM based FIFO it means that the next read strobe will present a valid output, and gives no direct indication of whether the output is currently valid or not.

Shift-register based FIFOs thus have more convenient readout timing and output signals, with the only drawback being the loss of the option to exploit RAM to create very large FIFOs.

The VHDL code in figure 3.51 implements a shift-register based FIFO.

In this implementation, the FIFO can be emptied at any time by asserting either the asynchronous `clear` input or the synchronous `sclr` input. Asserting the synchronous `si` input causes whatever is at the `d` input to be latched into the FIFO, unless the FIFO is full. The `q` output presents the oldest value that is currently in the FIFO. In particular, if the `si` input is asserted when the FIFO is empty, the value latched from the `d` input appears at the `q` output, ready to be latched by an external circuit, on the following rising edge of the clock. The oldest value in the FIFO can be discarded, and its value at the `q` output replaced with the next oldest value, by synchronously asserting the `so` input. The `empty` and `full` outputs are asserted when the FIFO is empty or full, respectively.

Note that if both the `si` and `so` inputs are asserted when the FIFO is full, then, as one should expect, one new value is shifted in while the oldest value is discarded. In all other cases, if the `si` input is asserted when the FIFO is full, nothing happens.

Asserting the `so` output when the FIFO is empty, always does nothing, regardless of the value of the `si` input.

Simultaneously asserting both the `si` and `sclr` inputs, both clears the previous contents of the FIFO, and shifts in a new value to the cleared FIFO, from the `d` input.

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

entity ccb_fifo is
  generic (LEN : integer := 16; WID : integer := 8); -- The maximum occupancy and width of the FIFO.
  port (d                         : in  std_logic_vector(WID-1 downto 0);
        q                         : out std_logic_vector(WID-1 downto 0);
        clock, clear, sclr, si, so : in  std_logic;
        empty, full               : out std_logic);
end ccb_fifo;

architecture ccb_fifo_arch of ccb_fifo is
  constant ZERO : std_logic_vector(WID-1 downto 0) := (others => '0');
  --
  -- Declare the two arrays that respectively represent the registered
  -- values in the FIFO, and the combinational computation of what
  -- should be latched into them at the next active clock edge.  Note
  -- the extra element, which simplifies shifting zero into the final node.
  --
  type node_values is array(LEN downto 0) of std_logic_vector(WID-1 downto 0);
  signal node_val, next_node_val : node_values;
  --
  -- Declare internal wires for the empty and full output signals.
  --
  signal fifo_empty, fifo_full : std_logic;
  --
  -- The occupancy register counts the number of values currently in the FIFO.
  --
  signal occupancy : integer range 0 to LEN;
  --
  -- next_occupancy is the combinationally computated value that
  -- should be latched into the occupancy register at the next active
  -- clock edge.
  --
  signal next_occupancy : integer range 0 to LEN;
begin
  --
  -- The output of the FIFO is the value of the zero'th node.
  --
  q <= node_val(0);
  --
  -- Compute the full and empty indicators according to the FIFO's occupancy.
  --
  fifo_empty <= '1' when occupancy=0 else '0';
  fifo_full <= '1' when occupancy=LEN else '0';
  --
  -- Wire the above full and empty values to their formal outputs.
  --
  empty <= fifo_empty; full <= fifo_full;
  --
  -- Work out what value each FIFO register should latch next.
  --
  node_selector: for i in LEN-1 downto 0 generate
    next_node_val(i) <= d when si='1' and ((sclr='1' and i=0) or
   (sclr='0' and ((so='0' and occupancy=i) or (so='1' and occupancy=i+1)))) else
                       ZERO when sclr='1' else
                       node_val(i+1) when so='1' else
                       node_val(i);
  end generate node_selector;
  --
  -- The dummy extra node should always contain zero.
  --
  next_node_val(LEN) <= ZERO;
  --
  -- Work out what number the occupancy register should latch next.
  --
  next_occupancy <= 1 when sclr='1' and si='1' else
                    0 when sclr='1' else
                    occupancy+1 when si='1' and so='0' and occupancy/=LEN else
                    occupancy-1 when si='0' and so='1' and occupancy/=0 else
                    occupancy;
  --
  -- Synchronously update, or asynchronously reset all of the FIFO's registers.
  --
  fifo_update_proc: process (clock, clear)
  begin
    if clear = '1' then                    -- Asynchronously reset the FIFO?
      occupancy <= 0;
      node_val <= (others => ZERO);
    elsif clock'event and clock = '1' then -- Synchronously update all registers?
      occupancy <= next_occupancy;
      node_val <= next_node_val;
    end if;
  end process fifo_update_proc;
end ccb_fifo_arch;
```

Figure 3.51: The VHDL implementation of the CCB FIFO component

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 most of their values are used by the *State Generator*. There are 4 types of register.

- **info**

  These are informational registers, whose values are read-only to the CPU.

- **param**

  Registers of this type configure operating parameters that aren't related to individual scans. The new parameters are adopted as soon as practically possible, after they are received.

- **action**

  Writing to an action register causes something to happen, regardless of whether or not the write operation changes the value of that register.

- **config**

  These are scan-configuration registers. Although they can be written to at any time, their new values are not used immediately. Instead, a snapshot of their values is taken whenever a start-scan command is received, and this snapshot is used to configure the commanded scan.

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

Name	Type	Address of LSB	Bytes	Bits
ccb_id_reg	info	00	1	8
holdoff_dt_reg	param	01	1	8
cal_diode_reg	action	02	1	8
start_scan_reg	action	03	1	8
state_len_reg	config	04	2	16
blank_dt_reg	config	06	1	8
diode_rise_reg	config	07	4	32
diode_fall_reg	config	11	2	16
integ_len_reg	config	13	2	16
roundtrip_dt_reg	config	15	1	8
dump_adc_reg	config	16	1	8
dump_lim_reg	config	17	2	16
adc_delay_reg	config	19	1	8
scan_id_reg	config	20	4	32

Figure A.1: A list of all CCB registers

The CCB ID register (ccb_id_reg)

Whenever the FPGA firmware is reset, the *Control Gateway*'s EPP address register is reset to zero. This gives it the address of the CCB ID register (ccb_id_reg). Thus, if the computer performs an EPP data-read, immediately after using the EPP reset line to reset the FPGA firmware, then it will receive the value of the ID register, which is 27. Thus, by checking that an initial EPP data-read returns 27, the computer can check that the CCB hardware is connected to the parallel port, and that the *Control Gateway* is apparently functioning, without having to first perform an EPP address write. Thereafter, the ID register can also be read back at any other time, by first performing an EPP address write of zero. The bit assignments of this register are:

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

The list of general configuration registers

These are registers that aren't related to a particular scan.

- 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 25.6μs $\leftrightarrow$ 819.2μs

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
01	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.

The list of action registers

- `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
02	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.

- `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.

  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
03	X	sync	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.
- **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.

The list of configuration registers

- `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
04	b15	b14	b13	b12	b11	b10	b9	b8
05	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
06	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
07	b31	b30	b29	b28	b27	b26	b25	b24
08	b23	b22	b21	b20	b19	b18	b17	b16
09	b15	b14	b13	b12	b11	b10	b9	b8
10	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
11	b15	b14	b13	b12	b11	b10	b9	b8
12	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
13	b15	b14	b13	b12	b11	b10	b9	b8
14	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
15	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.

- `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 identification number, with `b0` denoting the least significant bit, and `b31` denoting the most significant bit.