The designs of the master and slave CCB FPGAs

`[Document number: A48001N004, revision 5]`

Martin Shepherd, California Institute of Technology

August 25, 2004

This page intentionally left blank.

Abstract

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

[This remains incomplete]

Contents

List of Figures

Chapter 1

Introduction

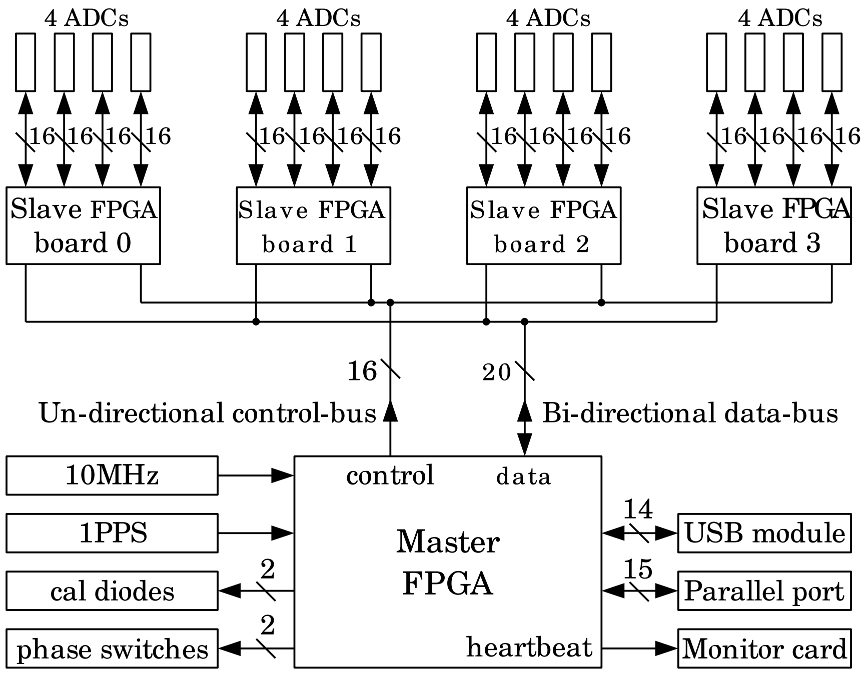


Figure 1.1: An overall summary of the FPGA connections

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

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

Note that although a bi-directional 20-bit data-bus is shown in the diagram, the current design only uses 16 of these bits, and only transfers data over them in one direction, directed from the slave FPGAs to the master FPGA.

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

Chapter 2

The slave FPGAs

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

2.1 An overview of the internals of a slave FPGA

Starting from the left hand-side of the diagram, DCM1 generates a phase-shifted copy of the main FPGA clock-signal. This signal clocks the 4 external ADCs, and latches their samples into input registers within the associated *Sampler* components. The *Sampler* components take either these latched samples, as their input samples, or fake pseudo-random samples from the *Signal Injector* component, according to the state of the `test` control-signal. The selected input samples are then both reproduced at the `raw` outputs of the *Sampler* components, and integrated between `start`-signal pulses.

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

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

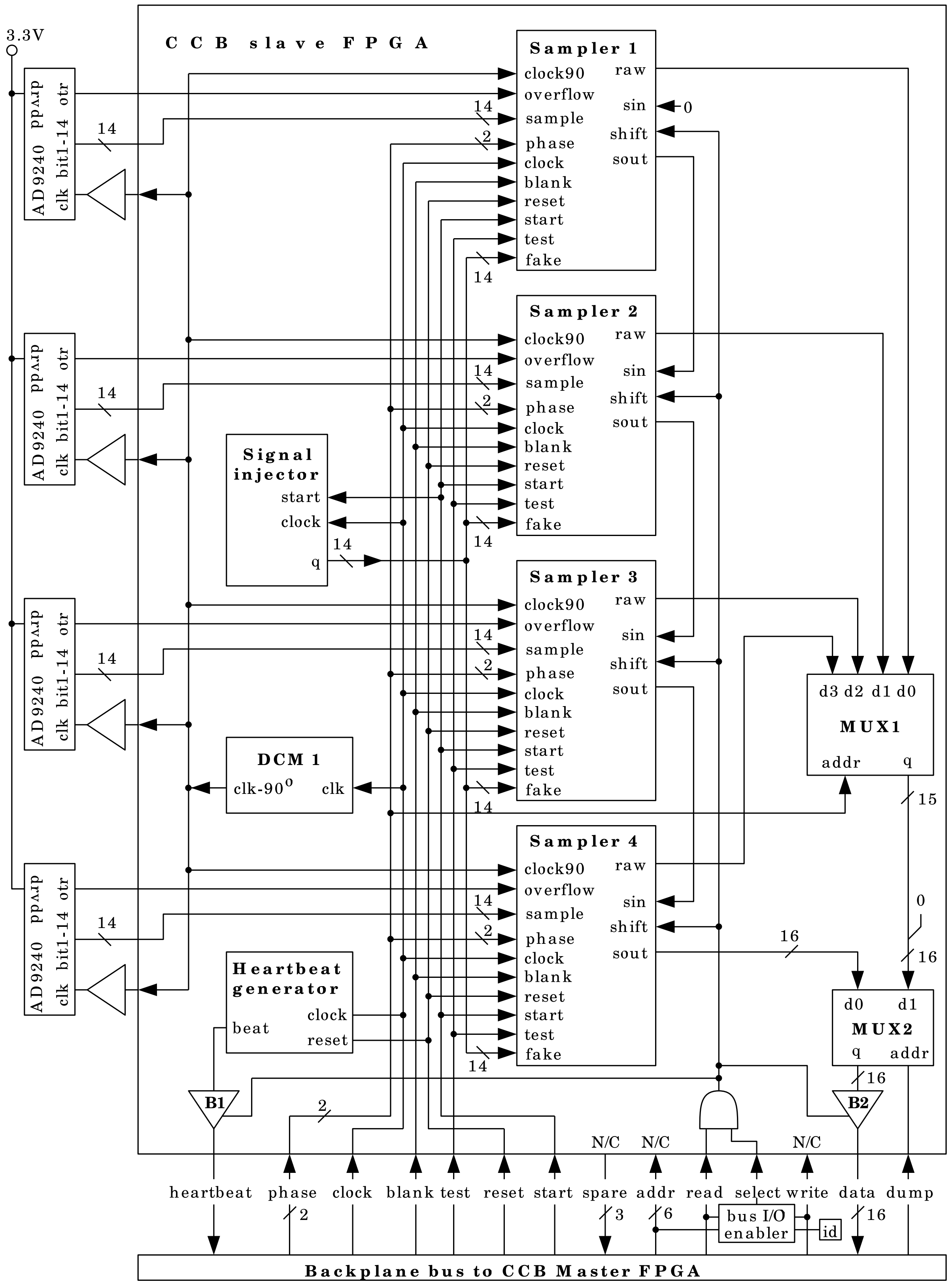


Figure 2.1: The top-level design of the slave FPGA

The `select` control-signal is asserted when the `addr` signal contains the board-ID of the slave, and either the `read` or `write` 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 `read` signal or the `write` signal is asserted. In the current design the master never sends anything to the slaves over the data-bus, so the `write` strobe is simply ignored by the slave FPGAs.

When the `read` signal is asserted, the addressed slave responds to this by sending the master integrated, or dump-mode, ADC samples. The master asserts the `read` 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 `read` strobe, and clock out the next data sample, ready to be read by the master, another clock cycle later. Subsequently, samples continue to be clocked out on the rising edges of the clock, until the `read` strobe is deasserted again by the master.

The asserted `read` strobe also causes the addressed slave to drive its `heartbeat` and `spare` output signals 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 also has different semantics in the two modes. In normal integration mode, it identifies the phase-switch bin that the latest sample should be added to. In dump mode it identifies the *Sampler* whose raw samples are to be passed to the `data` output, via `MUX2`.

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

2.1.1 The Heartbeat Generator

The `heartbeat` output signal of the currently addressed slave, is examined by the master FPGA, to determine if that slave is present and showing signs of life. The generation of this heartbeat signal is shown in figure 2.2.

Since an FPGA that has been fried, or has failed to load its firmware, could unpredictably present any signal on its I/O pins, the `heartbeat` output is a dynamic signal, with a known pattern that the master FPGA can check for. The pattern that is used, is simply a signal which toggles its state at each successive rising edge of the global clock. The master FPGA

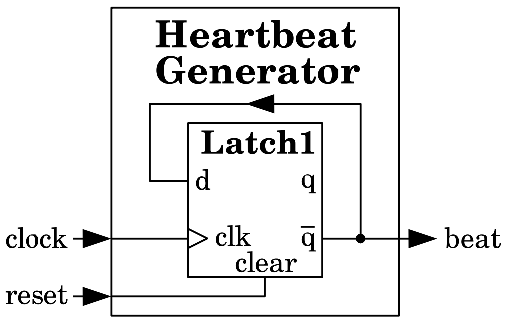


Figure 2.2: The Heartbeat Generator component

checks this at the start of each clock cycle, simply by using an XOR gate to compare a latched copy of the previous state of the `heartbeat` signal, to its current state. If the old and new heartbeat values of a given slave, aren't opposites, then that slave is flagged in the output data that are sent to the CCB computer.

2.1.2 The Signal Injector

The job of the *Signal Injector* is to generate repeatable pseudo-random fake ADC samples, for optional use by the *Sampler* components, in place of real ADC samples. The implementation, as shown in figure 2.3, is 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.

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 would be started anew at the start of the next integration period, so automatic restarting hasn't been included. Automatic restarting would be of dubious utility anyway, since the operator wouldn't see the repeatable test-sequence that they were expecting, if the sequence were restarted in the middle.

2.1.3 The Sampler component

The job of the *Sampler* component is to acquire raw samples from the ADC, integrate either these samples or fake ADC samples, into phase-switch bins, and present both the

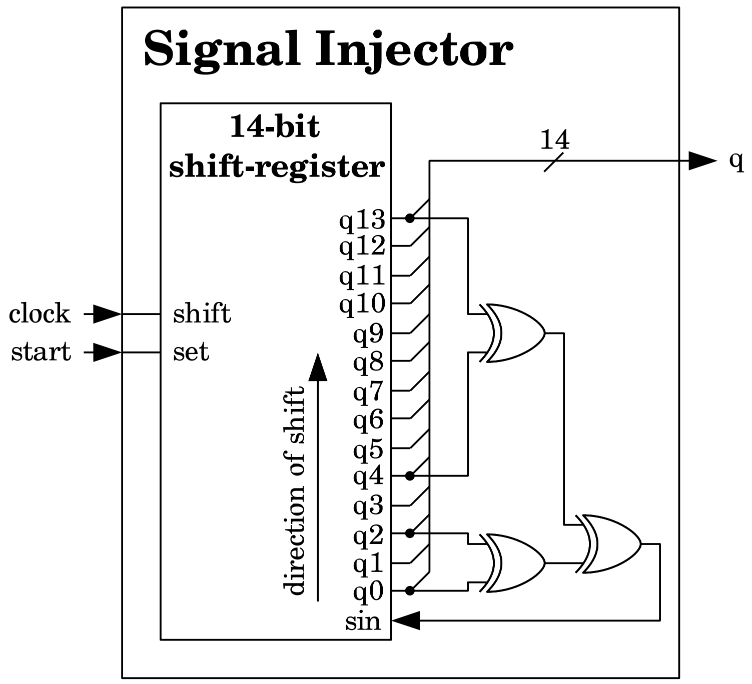


Figure 2.3: The Signal Injector component

resulting integrations, and the real or fake samples, for collection by the master FPGA. The implementation is shown in figure 2.4.

Register *Reg1* uses the phase-shifted ADC clock to acquire each new ADC sample and overflow signal from the external ADC. Multiplexer *MUX1* then takes either this sample and overflow, or a fake sample, with no overflow, and presents these to integrator, *Integrator1*, for integration into phase-switch bins. It also routes them to the `raw` output of the *Sampler*, for collection in dump mode.

Integration into phase-switch bins is performed by `Integrator1`. Within this component, each new sample is either ignored, if the `blank` signal is asserted, or added to the phase-switch integration bin that is specified by the 2-bit `phase` input. If the input sample either has its overflow bit asserted, or its addition to the integration would overflow the 32-bit integration-bin, then the contents of the integration-bin are replaced with a 32-bit number having all bits set to 1, and thereafter, this state persists until the bin is reset for the next integration period.

The end of one integration period, and the start of the next, is signaled by the `start` input signal. When this is asserted, the contents of the integration bins are copied into an output PISO, within the integrator component, as the integration bins are being reset for the new integration. All bins, except possibly for the currently selected bin, are reset by zeroing their contents. If the `blank` input is asserted, then the number in the currently selected bin is similarly reset to zero. Otherwise it is replaced with the value of the first sample of the new

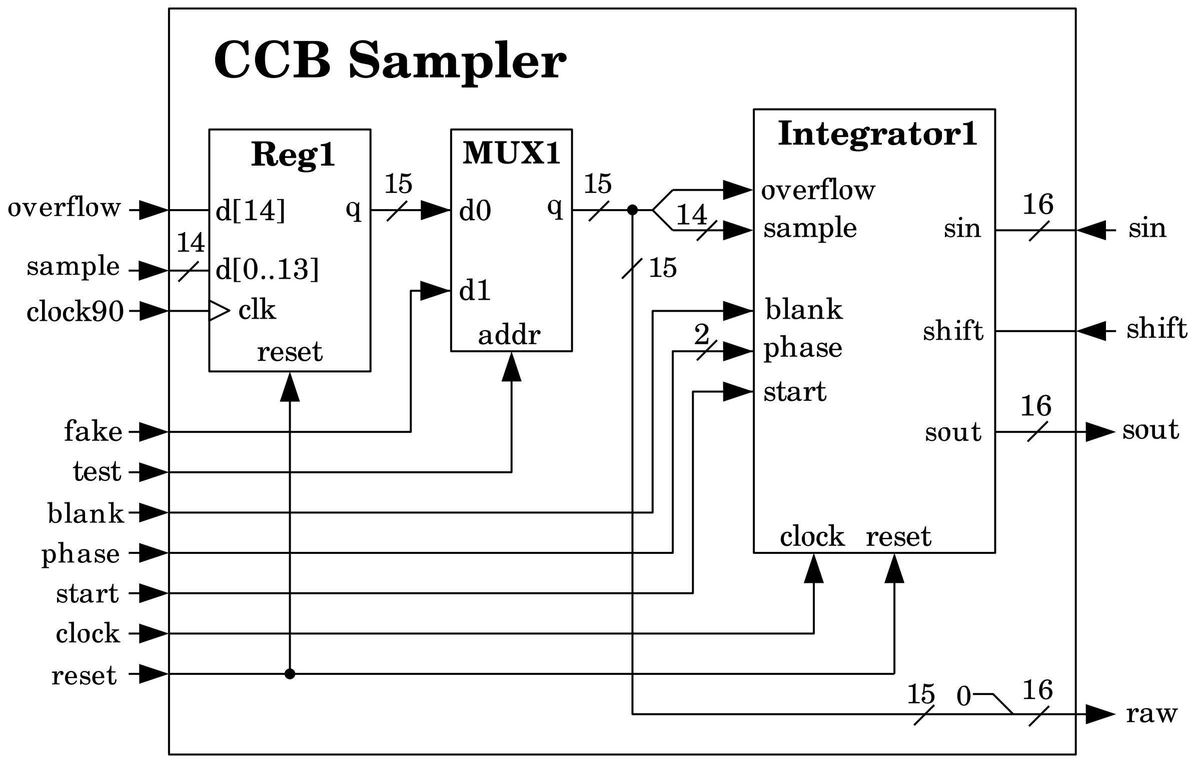


Figure 2.4: The Sampler component

integration period.

Although the integration bins are 32 bits wide, there are only 16 I/O pins available for transmitting data from each slave FPGA to the master FPGA. Thus, within the slaves, the chain of 16-bit PISOs of each of the *Sampler* components, is used to feed the master one 16-bit half of an integration result, each time that the master clocks the `strobe` input of the slave. For each integration-bin, the 16 least significant bits of the bin contents are read out first, followed by the 16 most significant bits.

2.1.4 The Integrator component

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

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

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

2.1.5 The Accumulator component

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

In the diagram, the *Adder* component and register, `Reg0`, form the accumulator cell used to integrate successive samples. This updates every clock cycle, despite samples only being added when the `select` input is asserted. On clock cycles when either the `select` input is not asserted, or the `blank` input is asserted, AND gate `A2` replaces the input-sample with zero, so that nothing gets added to the accumulator. The previous value of the accumulator cell is fed to the `d0` input of the adder, to be added to, except when the cell is being reset for a new integration. In the latter case, AND gate `A1`, changes the value at the `d0` input to zero. Thus, when the `start` signal is asserted, to start a new integration, the adder replaces the accumulator value with either zero, if either the accumulator isn't currently selected, or the `blank` input is asserted, or with the first sample of the new integration, otherwise.

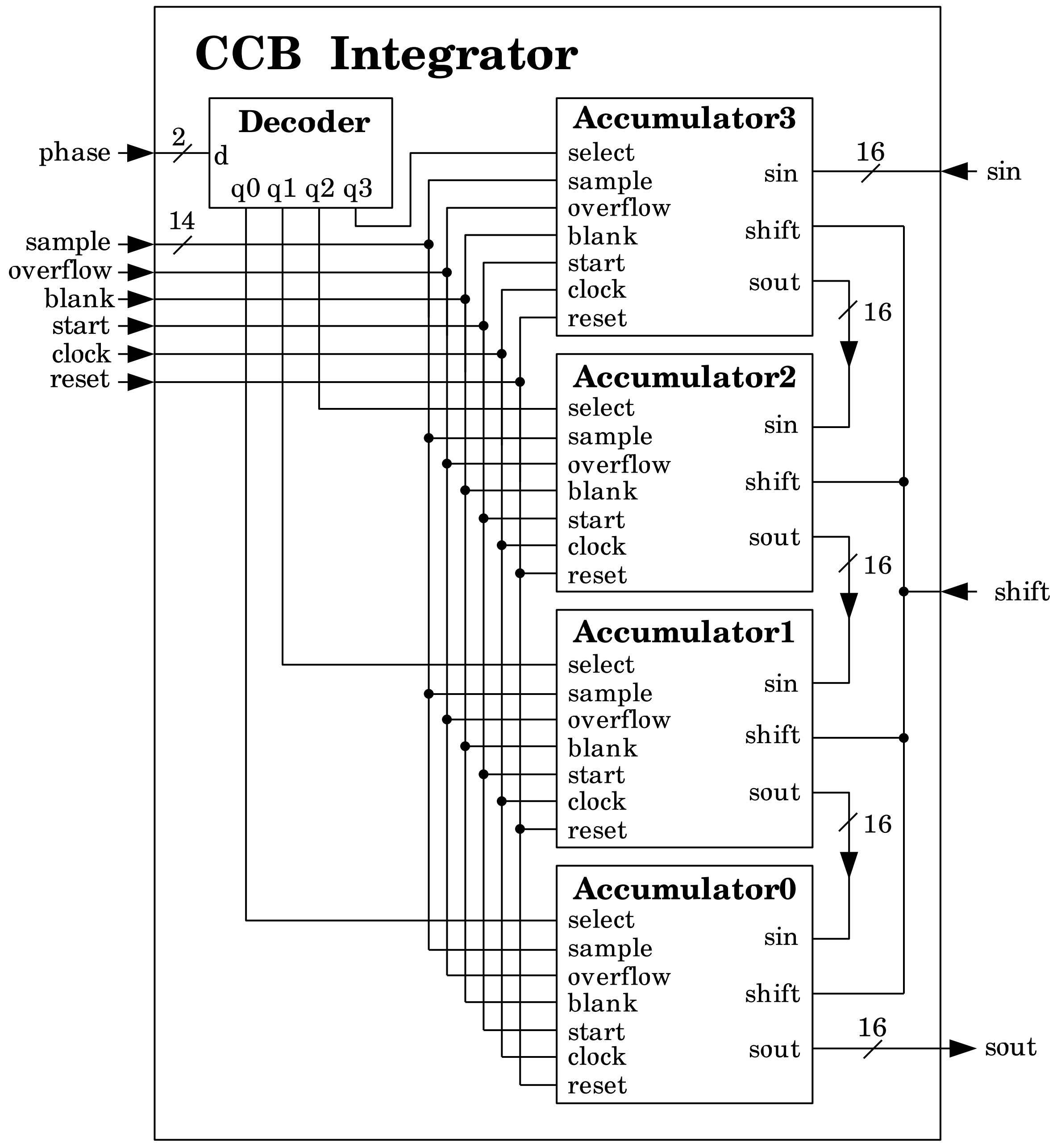


Figure 2.5: The Integrator component

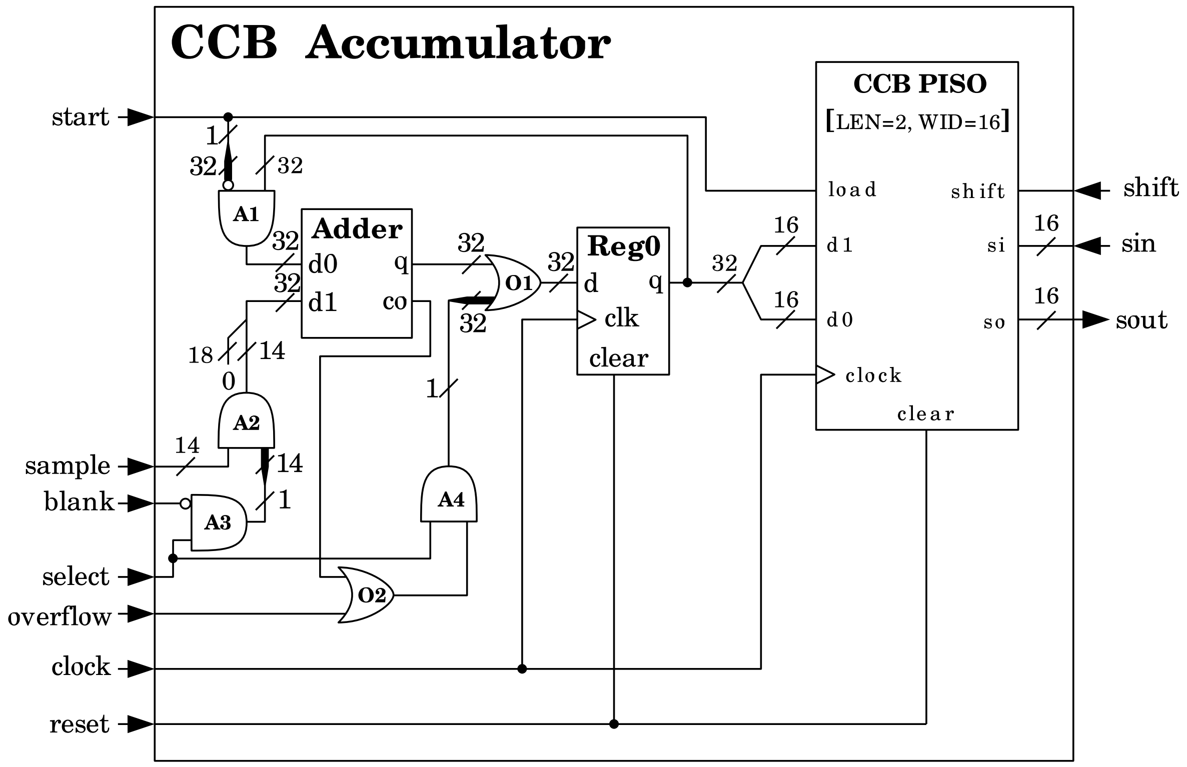


Figure 2.6: The Accumulator component

When the *Accumulator* is selected, if either its `overflow` input is asserted, or the output of the adder overflows, OR gate `O1` replaces the output of the adder with a 32-bit value with all bits 1. Since all further non-zero additions to this maximal value re-trigger this feature, this value persists until the start of the next integration period, when the accumulator cell is reset.

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 3.4.1. On the first rising edge of the clock that follows the `start` signal going high, at the start of a new integration, the accumulator register is initialized with the output of the adder, at the same time that the previous output of the accumulator register is being latched into the PISO. One clock cycle later, the output of the PISO will have settled to hold the least significant 16 bits of the accumulated integration. Thus integrated data can safely start to be read out from the accumulators two clock cycles after the `start` signal goes high.

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

2.1.6 The ADC clock signal

Note that the clock signal that is transmitted to the ADCs, and to the input registers within the `Sampler` components, is a phase shifted version of the FPGA clock, and is generated by one of the "Digital Clock Managers" of the parent Spartan-3 FPGA. On the CCB mailing list a preference for a 90° phase shift was expressed. This could be +90°, or as tentatively shown in the diagram, −90°.

The data-sheet of the AD9240 ADC says that the time taken between a rising clock edge at the ADC clock input, and a valid new sample being available at the ADC data outputs, ranges from between 8ns to 19ns. Thus if the ADC clock were generated by shifting the FPGA clock by +90° (ie. 25ns), then this would leave a minimum of 6ns between the time that valid data appeared at the data outputs of the ADC, and the time that the input registers in the FPGA attempted to latch this data. This seems rather a short time, given that the data outputs will presumably have to traverse PCB tracks, connectors, and the input capacitance of the FPGA pins, before arriving at the inputs of the registers. Thus, in the diagram, the alternative −90° phase-shift is indicated instead. This means that the registers latch the data at least 56ns after they become valid, and 25ns before the ADC next samples its inputs.

In practice, the *Digital Clock Managers* can be programmed to generate practically any phase shift, so the choice of phase-shift need not be set in stone at this point, and can be changed if testing proves that the initial choice was a bad one. This also means that the choice of whether to use inverting or non-inverting external clock buffer-amplifiers is unimportant, since either can be accommodated by selecting a different phase shift.

Chapter 3

The master FPGA

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

3.1 The Control Gateway

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

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

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

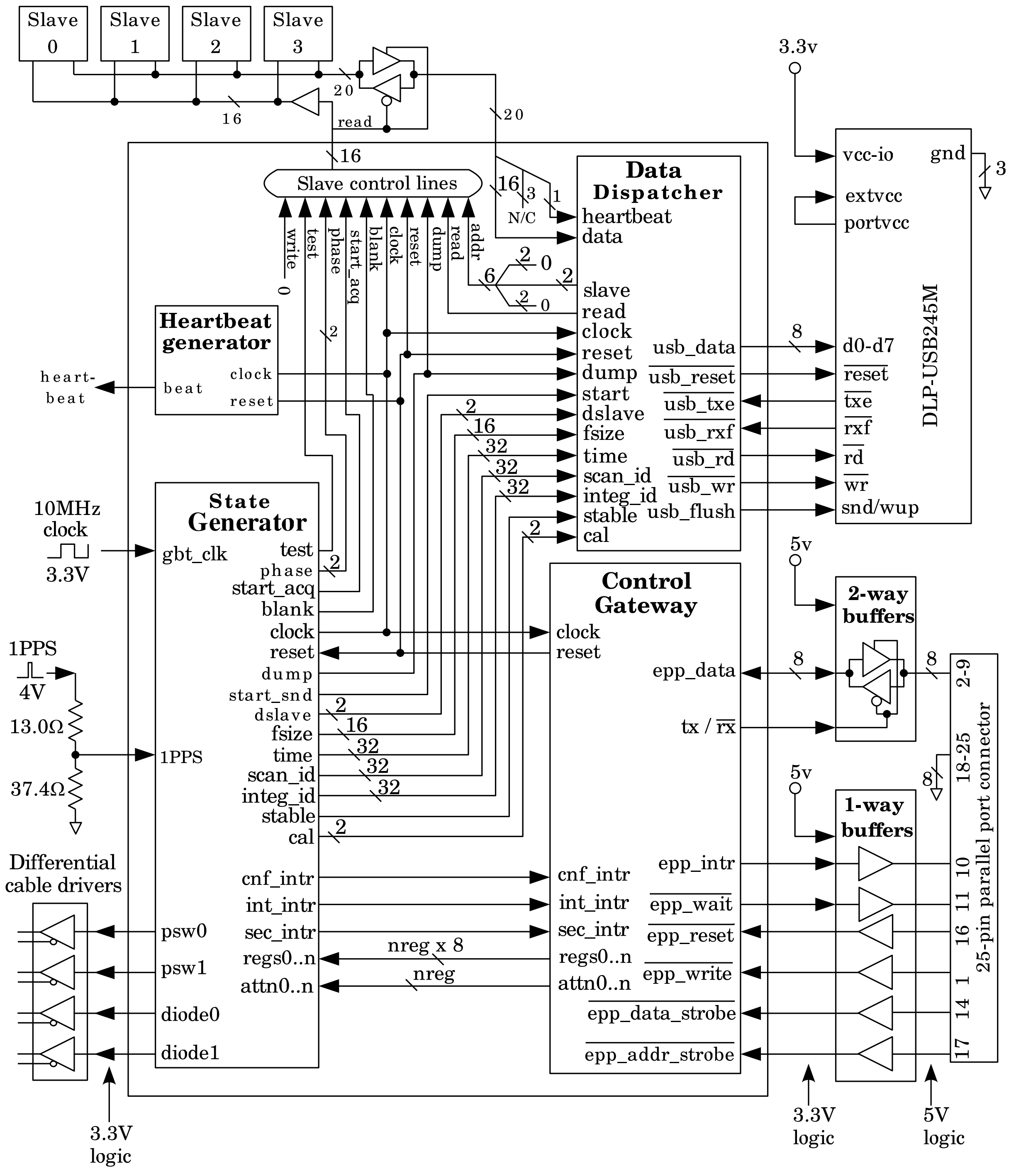


Figure 3.1: The top-level design of the master FPGA

- **The address write-cycle**

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

- **The data write-cycle**

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

- **The data read-cycle**

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

- **The address read-cycle**

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

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

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

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

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

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

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

3.1.1 The internals of the Control Gateway

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

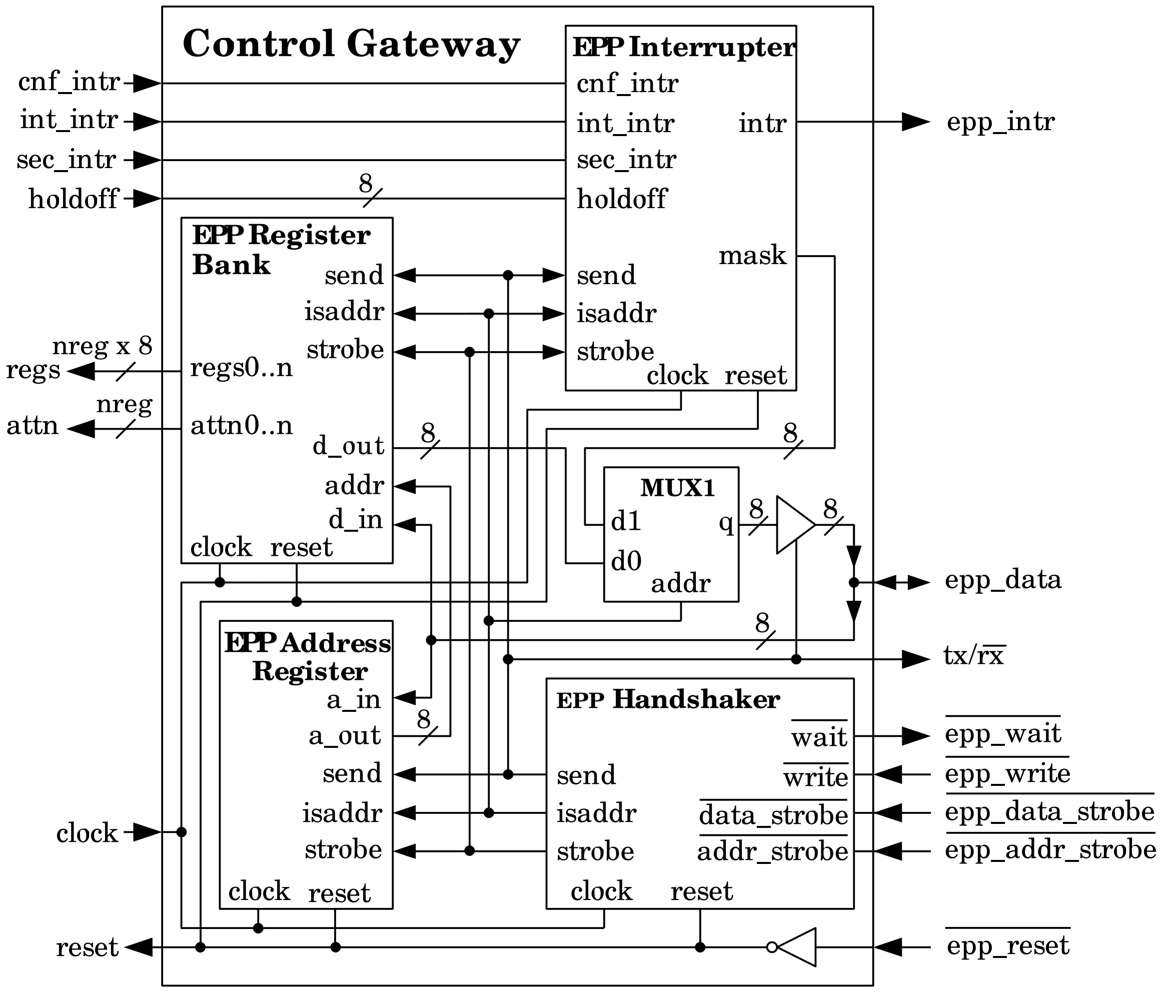


Figure 3.2: The Control Gateway

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

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

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

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

The EPP Handshaker

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

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

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

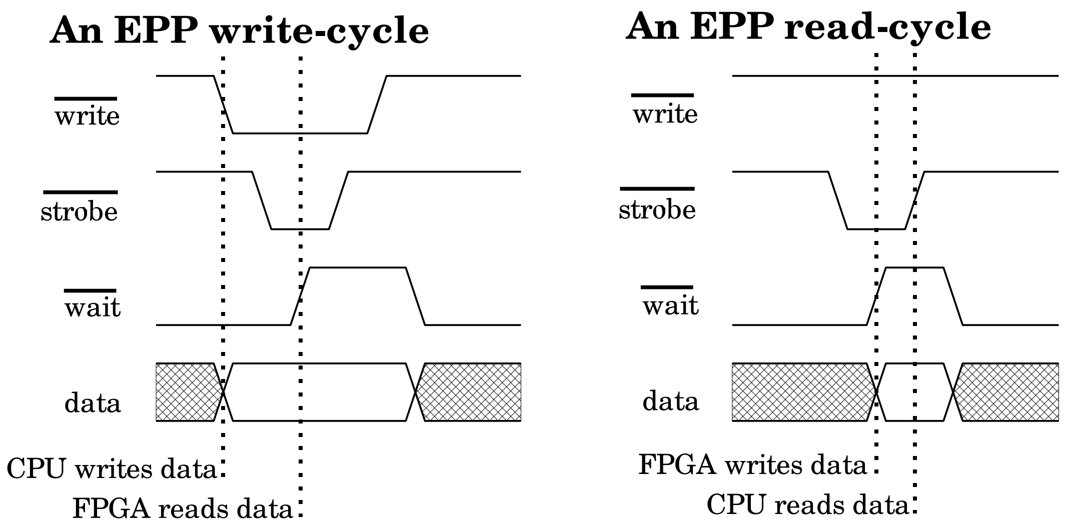
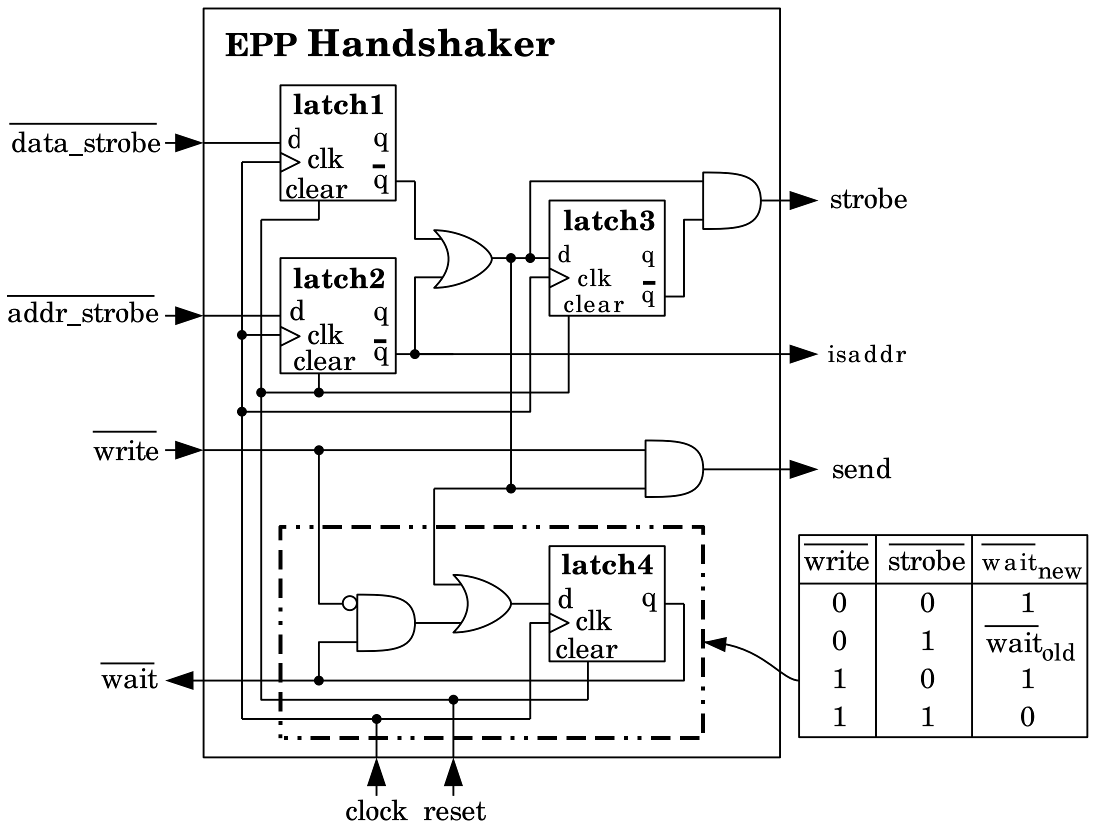


Figure 3.3: The standard EPP I/O cycles



$\overline{\text{write}}$	$\overline{\text{strobe}}$	$\overline{\text{wait}}_{new}$
0	0	1
0	1	$\overline{\text{wait}}_{old}$
1	0	1
1	1	0

Figure 3.4: The EPP Handshaker

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

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

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

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

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

The EPP address register

The *EPP Address Register*, as shown in figure 3.5, holds the address of the target data-register of subsequent EPP data-write and data-read cycles. It is implemented as a simple 8-bit register, which is updated synchronously at the start of each clock cycle. At the start of most clock cycles, the existing value of the register is simply re-latched, but when the `strobe`,

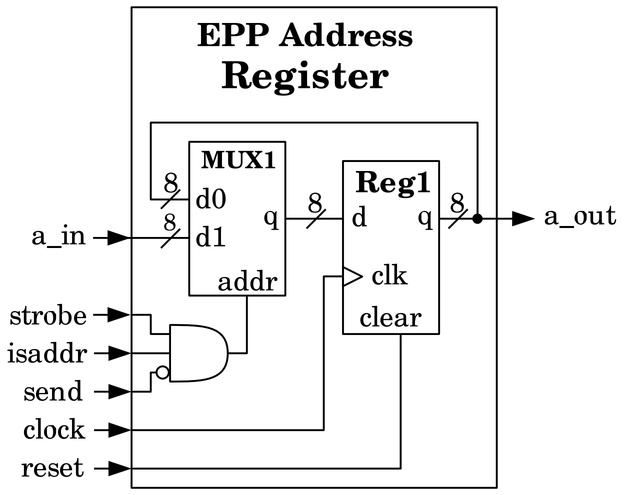


Figure 3.5: The EPP Address Register

`isaddr` and `send` inputs indicate that an EPP address-write transaction is in progress, the asserted address input of `MUX1`, causes the signals on the EPP data lines (at the `a_in` input) to be latched instead.

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

The EPP Register Bank

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

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

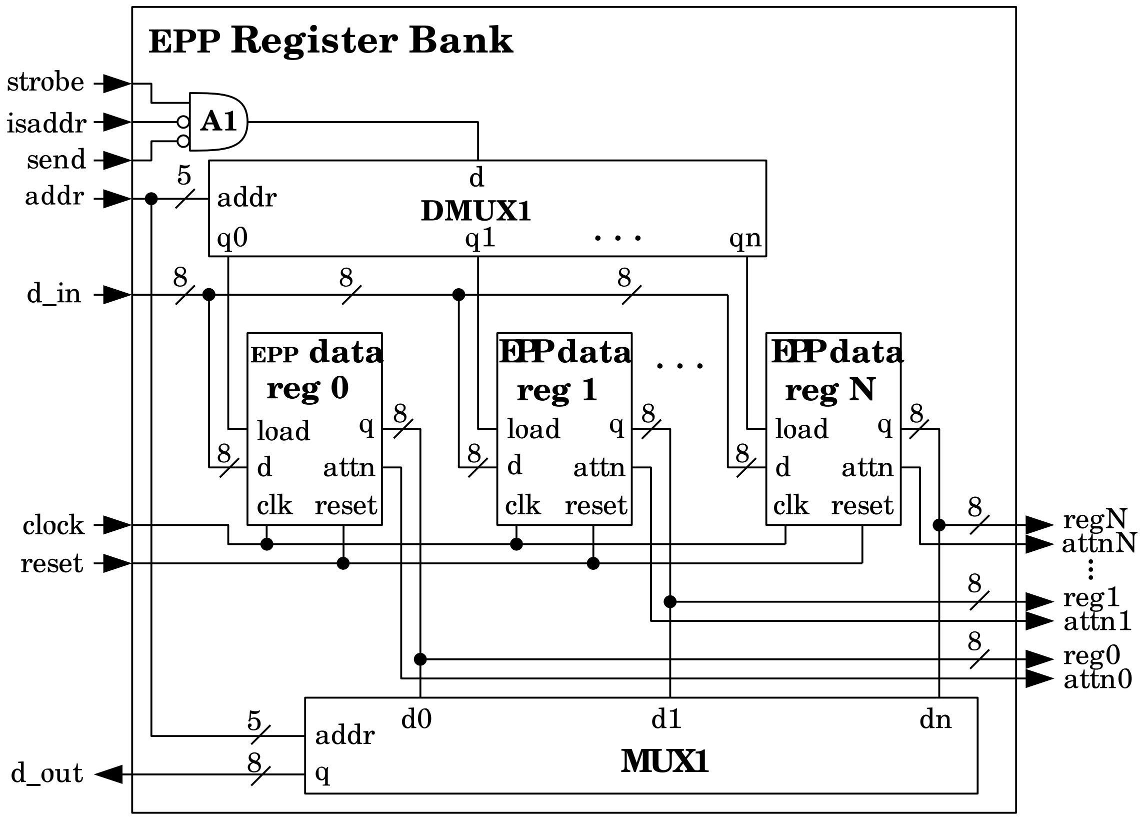


Figure 3.6: The EPP Register Bank

Note that although the 8-bit width of the `addr` input would allow up to 256 registers to be addressed, much fewer registers are actually needed, so the smaller number of bits shown going to `MUX1` and `DMUX1`, has been chosen to accomodate the preliminary list of registers given in section 3.3.1.

The individual data registers are implemented as *EPP Data Register* modules, implemented as shown in figure 3.7. At the start of most clock cycles, the existing value of the register is simply re-latched through `MUX1`, but when the `load` input indicates that an EPP data-write transaction to this register is in progress, the asserted address input of `MUX1`, causes the signals on the EPP data lines((at the `d` input) to be latched instead. Simultaneously, `latch2` latches the asserted `load` input to the `attn` output, and thus indicates to the *State Generator* whenever the register is updated.

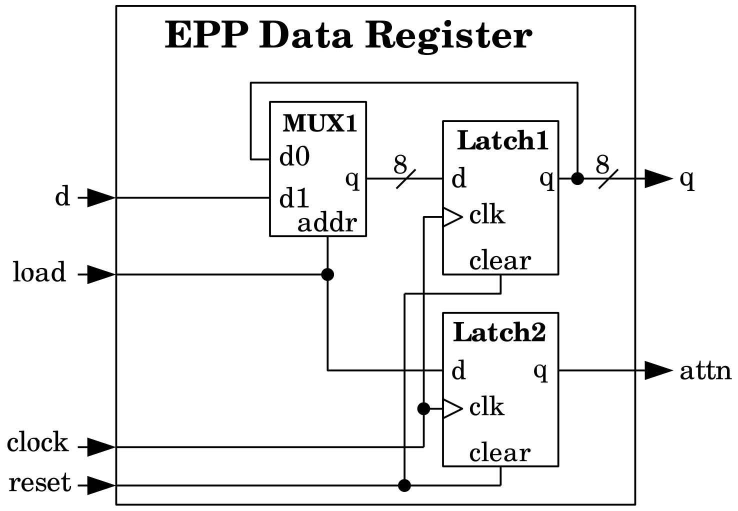


Figure 3.7: An EPP Data Register

The EPP Interrupter

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

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

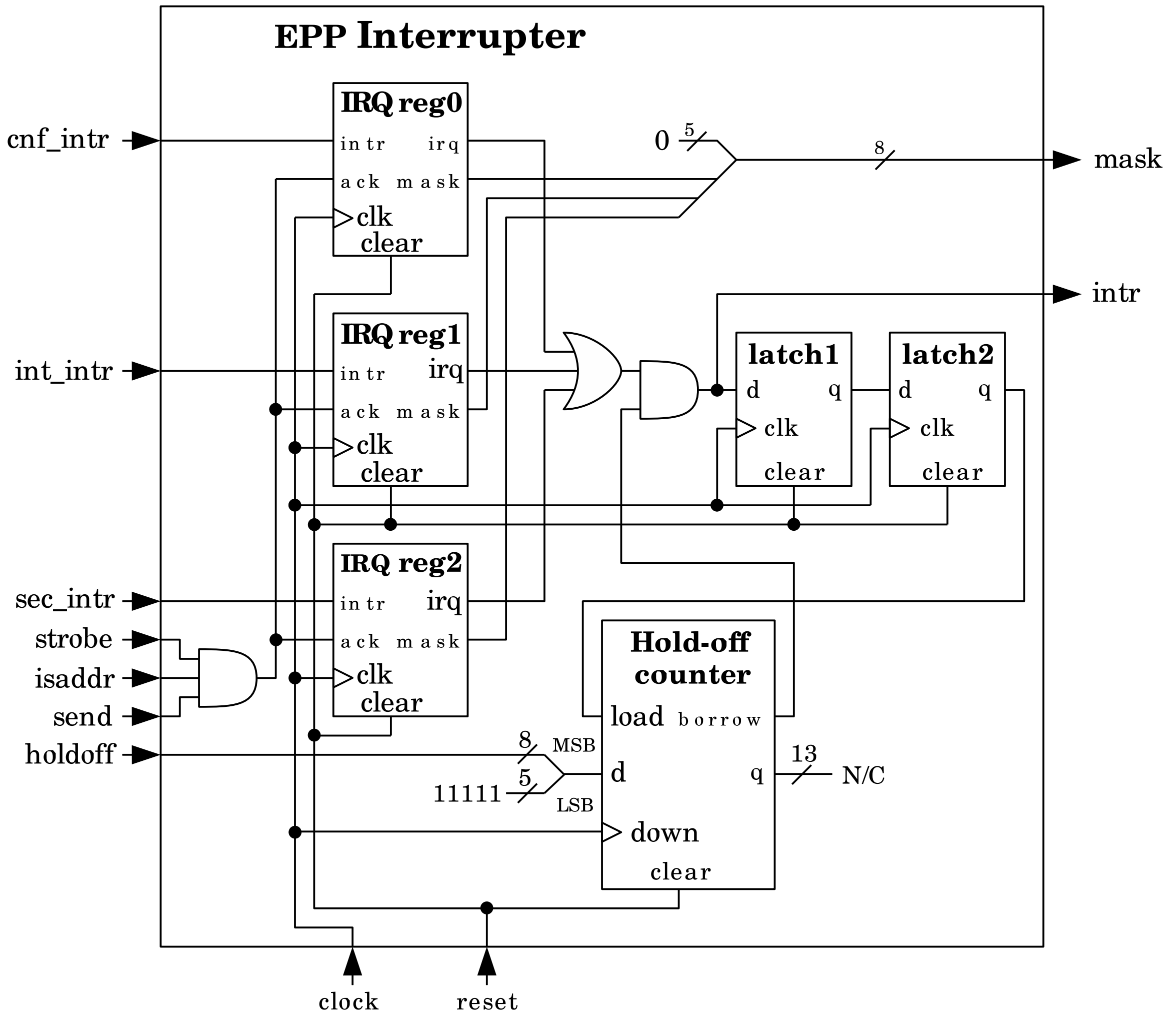


Figure 3.8: The EPP Interrupter module

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

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

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

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

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

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

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

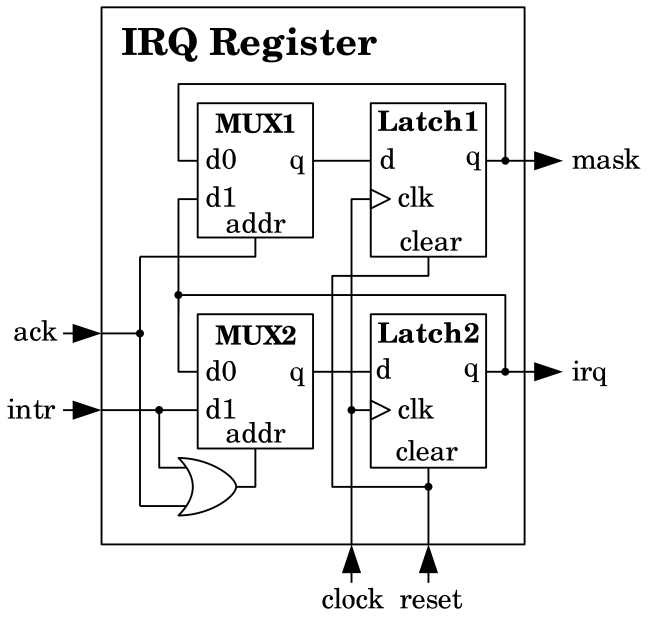


Figure 3.9: An Interrupt Request (IRQ) Register

When the event source associated with this register wishes to send an interrupt, it asserts the `intr` input for one clock cycle. At the end of this cycle (ie. the start of the next clock cycle), `Latch2` becomes asserted. It stays asserted until the `ack` (acknowledge) input is subsequently asserted for one clock cycle, at which point, the value of `Latch2` is transfered to `Latch1`, while `Latch2` adopts the current value of the `intr` input. Thus, whereas normally an asserted `ack` signal causes `Latch2` to be cleared; if a new interrupt is requested at the same time as the `ack` input is asserted, `Latch2` will record the new interrupt, instead of the new interrupt request being lost.

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

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

- `cnf_intr` - Integration configuration interrupts.

  Before the start of each new integration, the *State Generator* needs to know the desired on/off states of the cal-diodes. In principle this could be sent one integration in advance, from an end-of-integration interrupt handler. That was the original plan. However, to soften the real-time requirements placed on the device driver in the CCB embedded computer, and thereby make the CCB insensitive to occasional transient anomalies in Linux's interrupt latency, the current plan is to instead implement a FIFO containing the configurations of many integrations in advance, instead of just one. Keeping this FIFO filled is the job of the `cnf_intr` interrupt. At the start of a scan, to fill the FIFO, multiple `cnf_intr` interrupts are generated, each one telling the computer to send the configuration of the next un-configured integration. Thereafter at the start of each new integration, one entry is removed from the FIFO, and a new entry is requested by sending another `cnf_intr` interrupt.

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

- `int_intr` - Integration-done interrupts.

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

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

- `sec_intr` - 1 second interrupts.

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

By default, at boot time, EPP interrupts are disabled, and a write to the parallel-port configuration register is needed to enable them. While they are disabled, signals on the `intr` interrupt line are simply ignored by the computer. Thus the FPGA doesn't redundantly provide its own way to enable and disable the generation of interrupt signals on the `intr` line. Note that the resending of unacknowledged interrupts every `holdoff` clock-cycles, ensures that interrupts that are missed while the parallel-port has interrupts disabled, get re-sent and acknowledged as soon as interrupts become enabled.

3.2 The Data Dispatcher

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

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

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

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

3.2.1 The internals of the Data Dispatcher

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

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

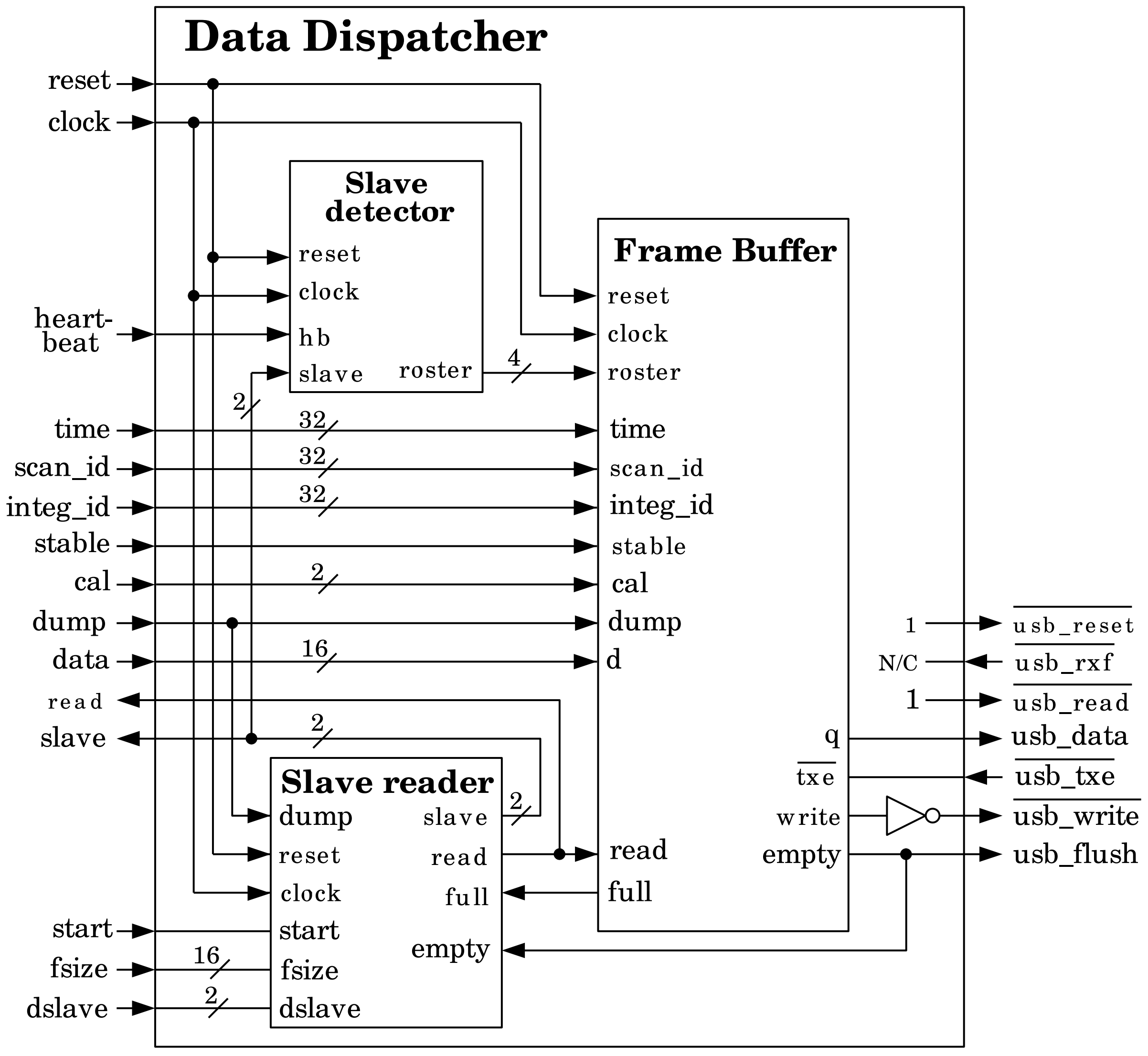


Figure 3.10: The Data Dispatcher

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

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

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

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

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

The internals of the Slave Reader

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

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

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

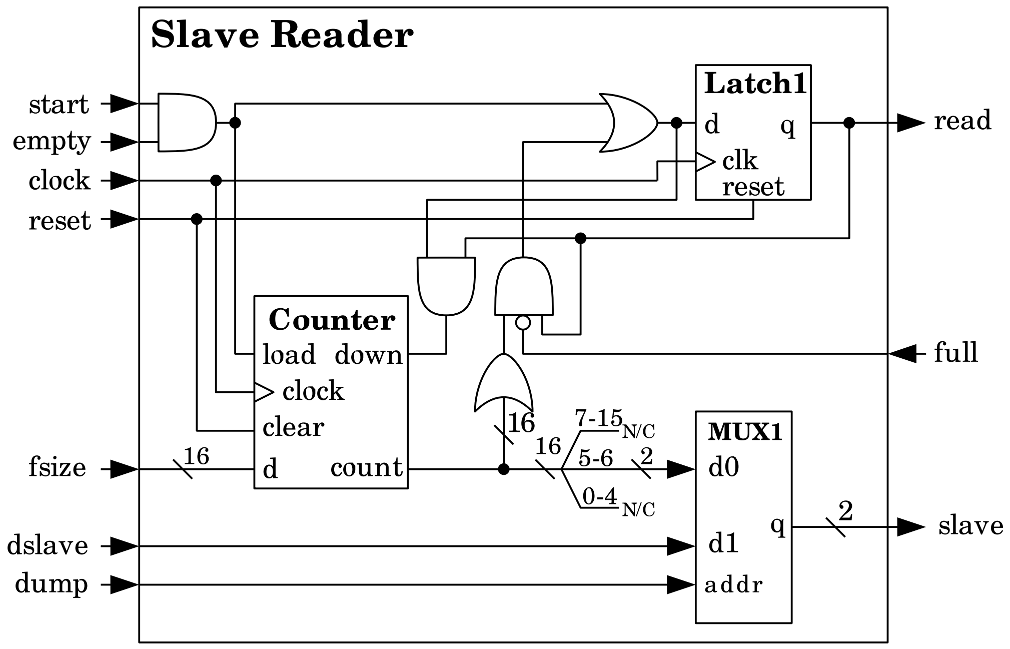


Figure 3.11: The Slave Reader

raw ADC samples for the subsequent dump-frame period.

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

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

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

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

In dump-mode, `MUX1` bi-passes the output-count's control of which slave is selected, and instead arranges that `fsize` samples be loaded from the slave specified by the `dslave` input. In this mode, the `fsize` input can specify any 16-bit number of samples, although if this exceeds the capacity of the *Frame Buffer*'s FIFO, the actual number of samples collected and sent to the computer will be truncated to fit.

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

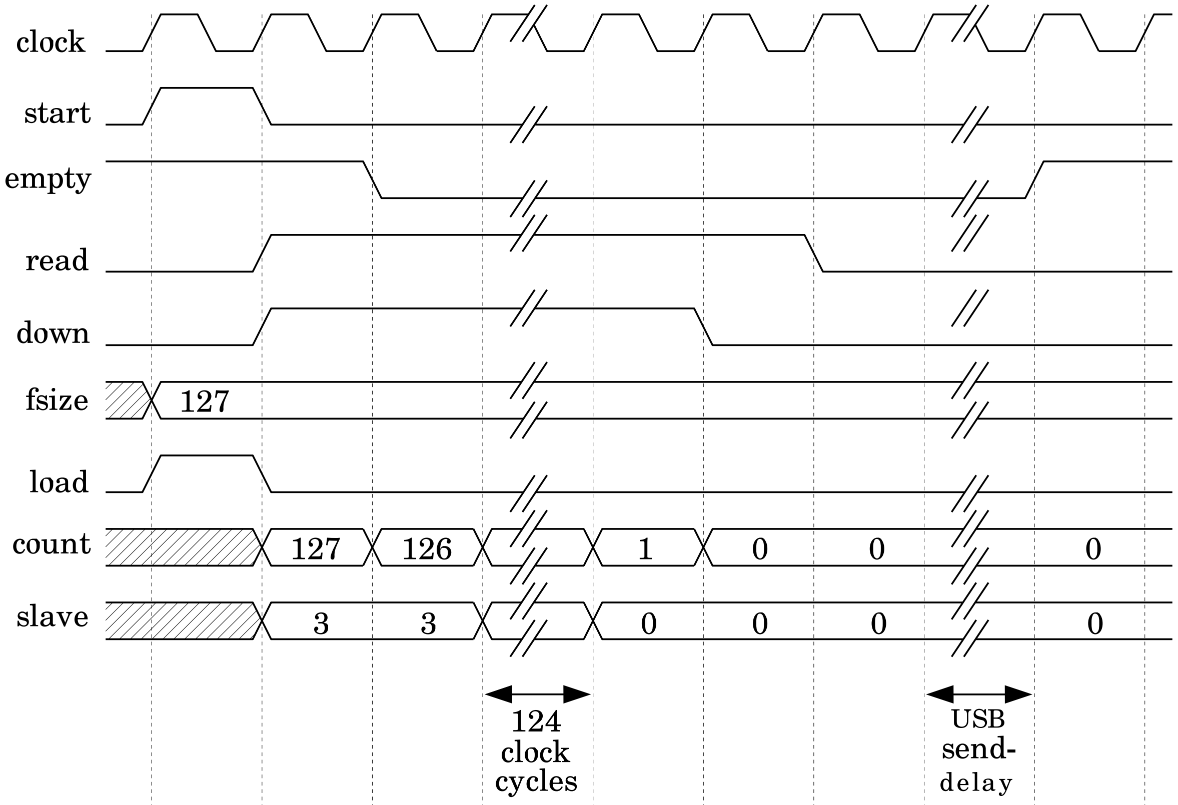


Figure 3.12: A timing diagram of the Slave Reader

The internals of the Frame Buffer

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

1. A *Frame Header*. This initially contains an 8 × 16-bit header describing the sample data.

2. A large FIFO, in which sample data are collected from the slave FPGAs at a faster rate than they can be sent to the computer.

3. A *Byte Streamer* component which transfers data, first from the header, then from the FIFO, to the USB interface chip.

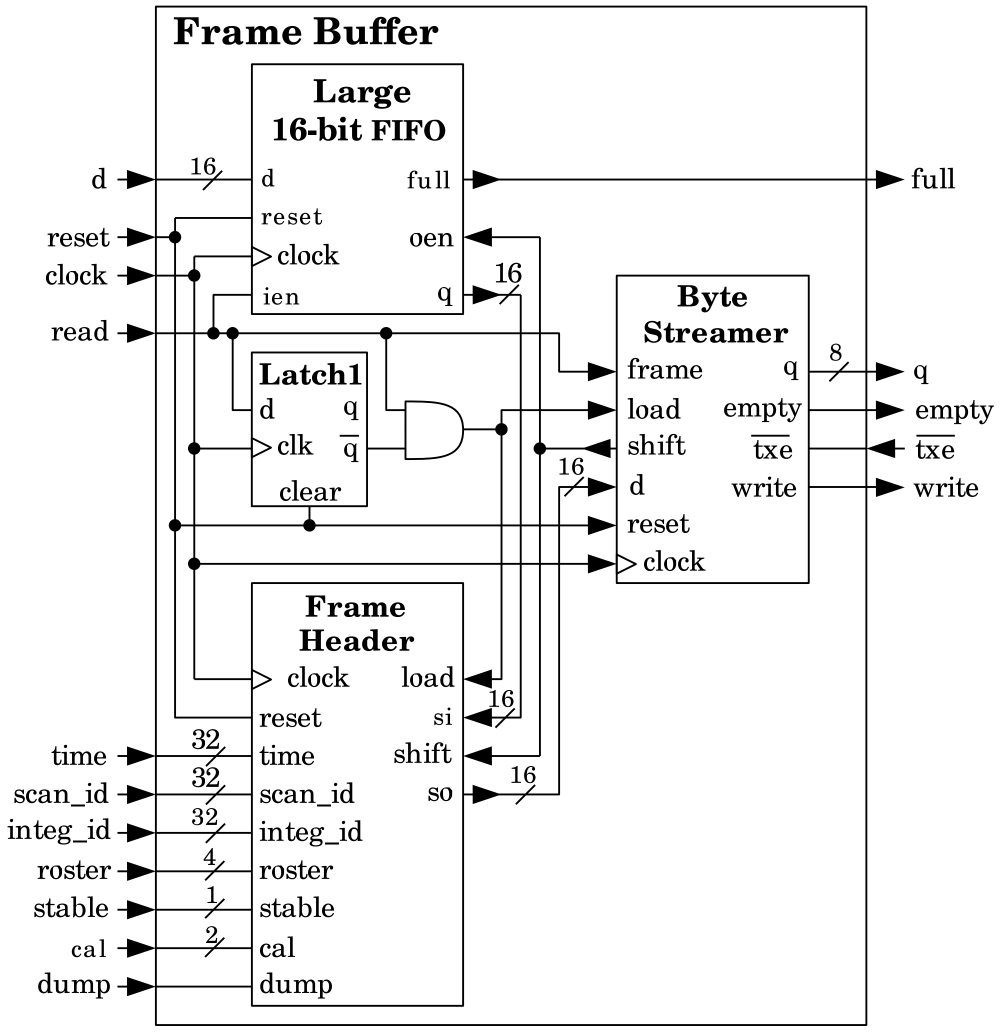


Figure 3.13: The Frame Buffer

Note that the contents of the FIFO are streamed through the *Frame Header*'s internal PISO. Thus, to ensure that as each sample gets shifted out of the *Frame Header*'s PISO, a new sample is shifted into it from the FIFO, the `shift` output of the *Byte Streamer* is connected to both the output-enable, `oen`, input of the FIFO and the `shift` input of the *Frame Header*. This is how the two data sources are combined into one stream.

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

The internals of the Frame Header

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

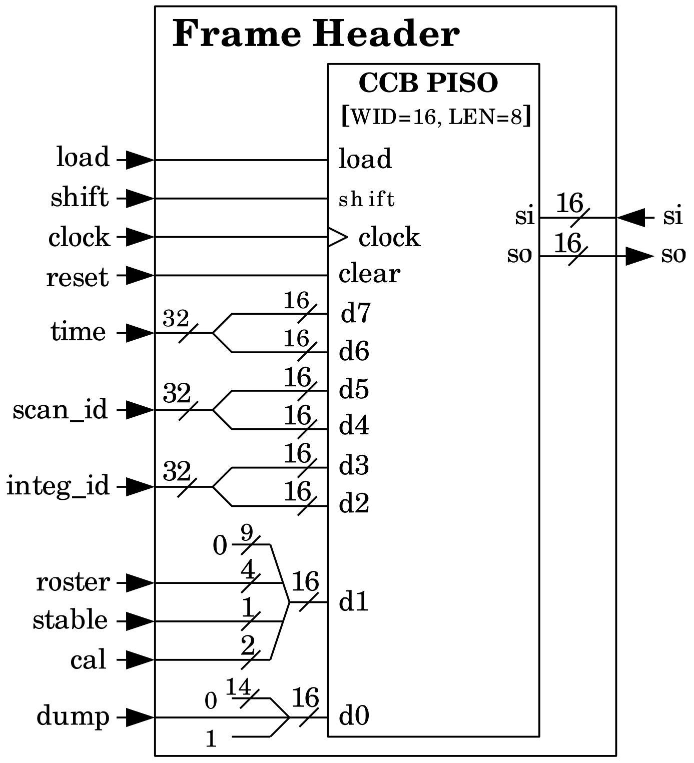


Figure 3.14: The Frame Header

Note that instead of using a conventional PISO, a copy of the customized PISO described in section 3.4.1, is used. This has separate `load`-enable and `shift`-enable inputs, which are acted upon at the rising edge of the clock. Unlike a conventional PISO, which either shifts serial data or loads parallel data on each clock cycle, the contents of the customized PISO remain unchanged during clock cycles when neither the `load` nor the `shift` signals are asserted. This is important, since the *Byte Streamer* doesn't want to be force-fed a new sample from the *Frame Header* every clock cycle, due to the handshaking overhead and flow-control delays imposed by the USB chip.

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

16-bit entries in the PISO with the following information.

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

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

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

- The second of the header words is a 16-bit word indicating various conditions that pertained while the data were being taken. Bits 0 and 1 report the commanded states of the cal-diode switches during the integration. Bit 2 tells the CCB manager that the phase and/or cal-diode switches were stable throughout the integration. Bits 3 through 6 is a boolean list of the slave FPGAs whose heartbeat signals indicate that they are present and functioning. The remaining 9 bits are unused.

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

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

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

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

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

The internals of the Byte Streamer

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

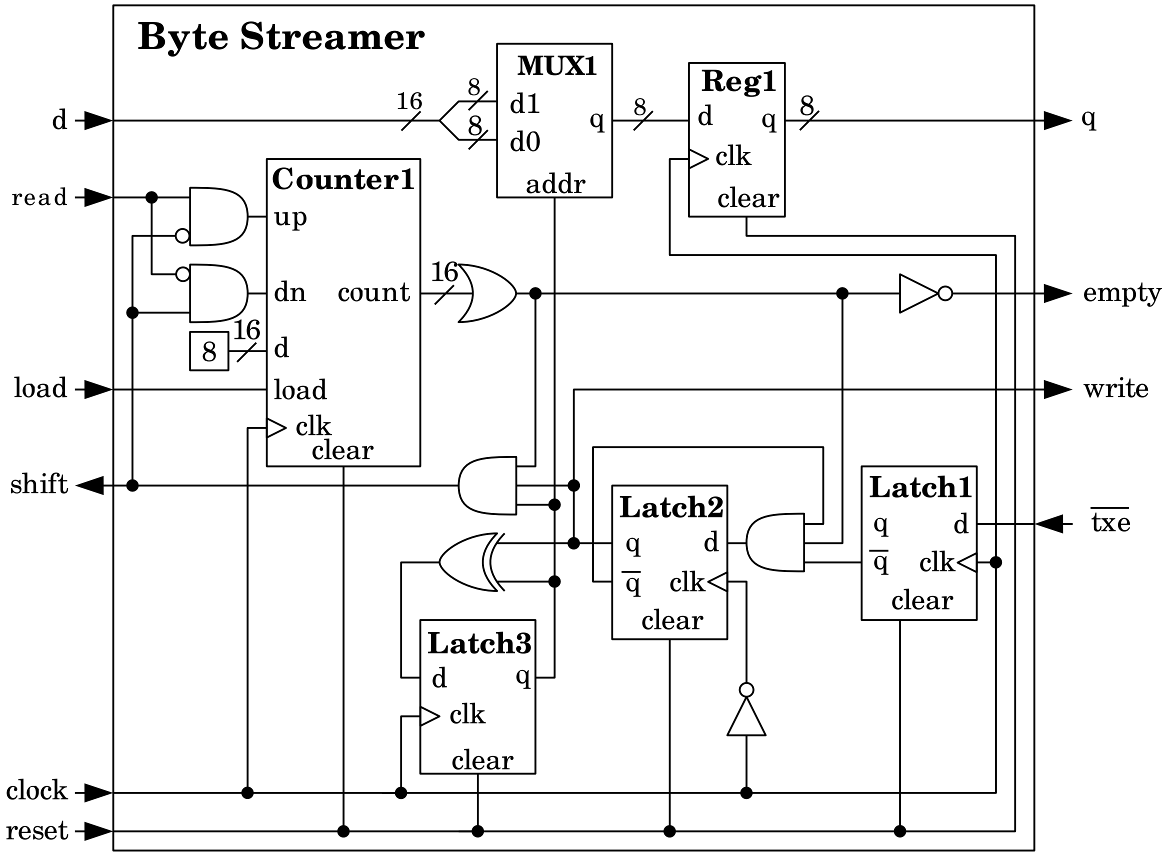


Figure 3.15: The Byte Streamer

The *Byte Streamer* takes one 16-bit sample at a time from its `d` input, and sends this to the USB chip, via the `q` output, starting with the least significant 8-bits, followed by the most significant 8-bits. Since the *Frame Buffer* fills up much faster than the USB chip accepts bytes for transmission, there is never any need for the *Byte Streamer* to wait for more data to arrive in the FIFO, once a new frame has been started. Thus the flow of data is solely regulated by the transmit-enable, $\overline{\texttt{txe}}$, output of the USB chip. From the point of view of the FPGA firmware, the $\overline{\texttt{txe}}$ signal is asynchronous, and thus needs to be passed through a pair of latches to stabilize and synchronize it with the FPGA clock. This is satisfied by latches 1 and 2. Note that `Latch2` is negative edge-triggered, while all other latches are positive edge-triggered. There are two reasons for this.

1. This reduces the time taken for a change in the state of the $\overline{\texttt{txe}}$ signal to reach the

output of `Latch3`, by a full FPGA clock cycle. This correspondingly speeds up the handshake with the USB chip.

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

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

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

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

The internals of the Slave Detector

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

As can be seen, each slave has its own *Heartbeat Detector* module, which is enabled when

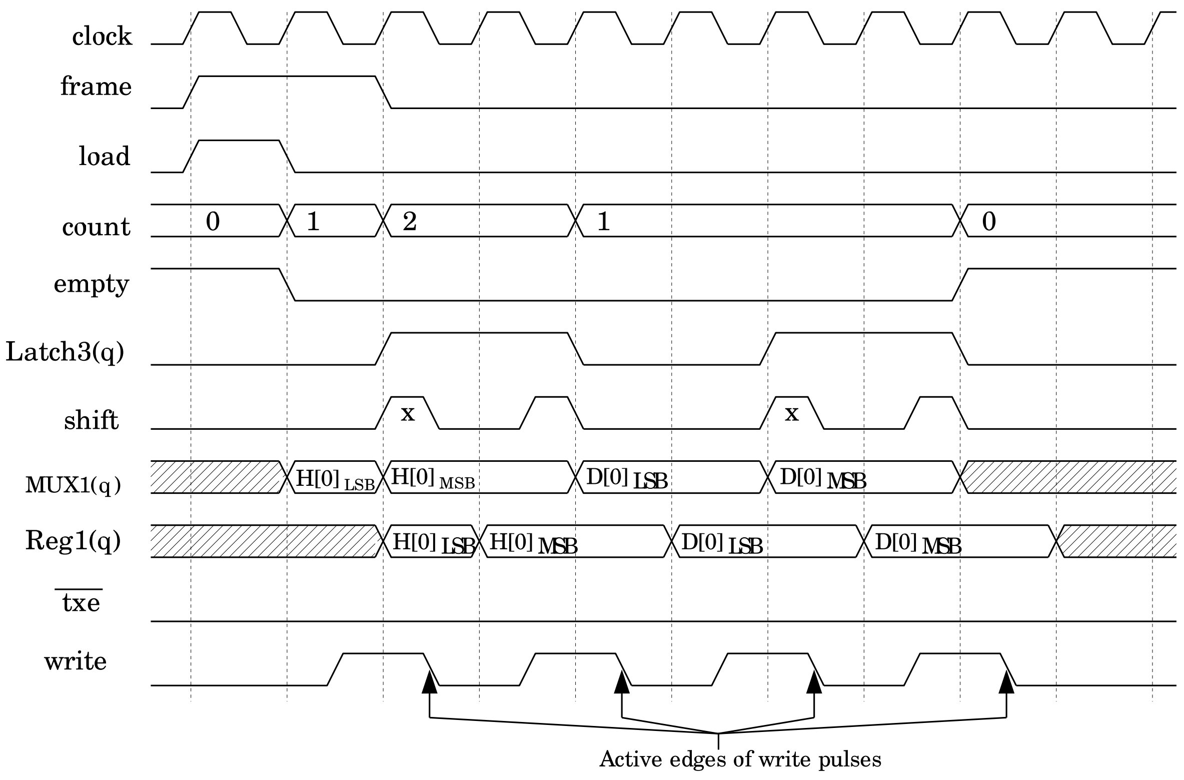


Figure 3.16: A timing diagram of the Byte Streamer

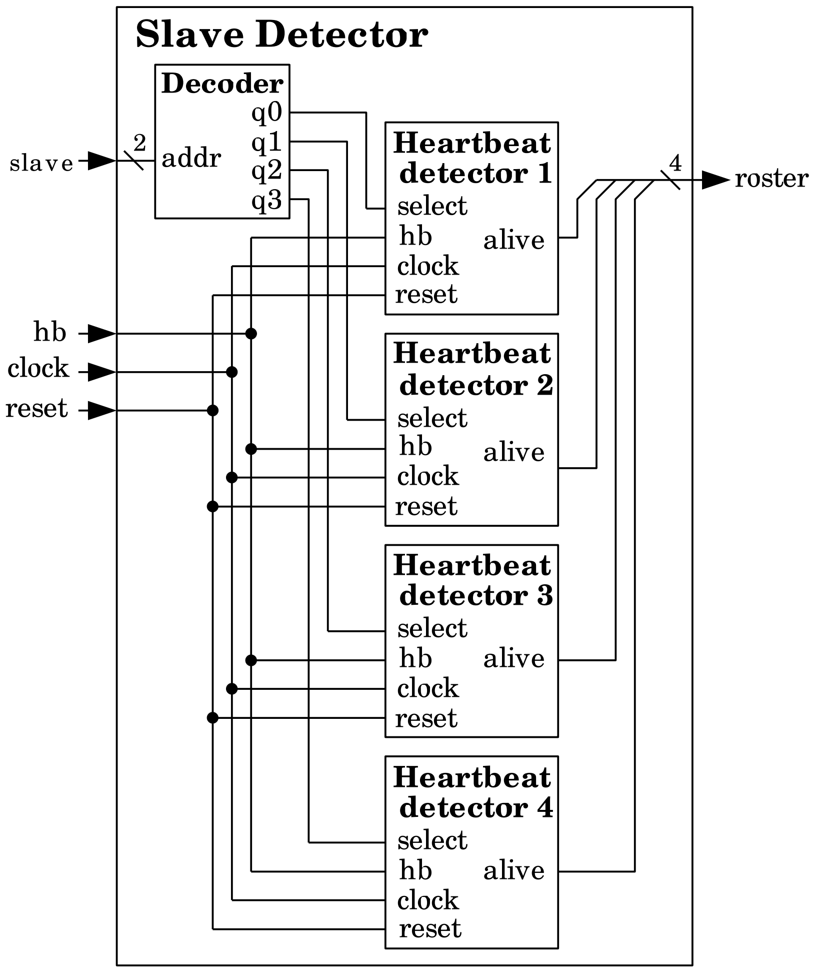


Figure 3.17: The Slave Detector

the slave is selected for readout by the *Slave Reader*. At the end of each integration the 4 `alive` outputs of the *Heartbeat Detector*s, combined into the 4-bit `roster` output, provide an indication of which slaves were alive during the integration period.

The internals of the Heartbeat Detector modules

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

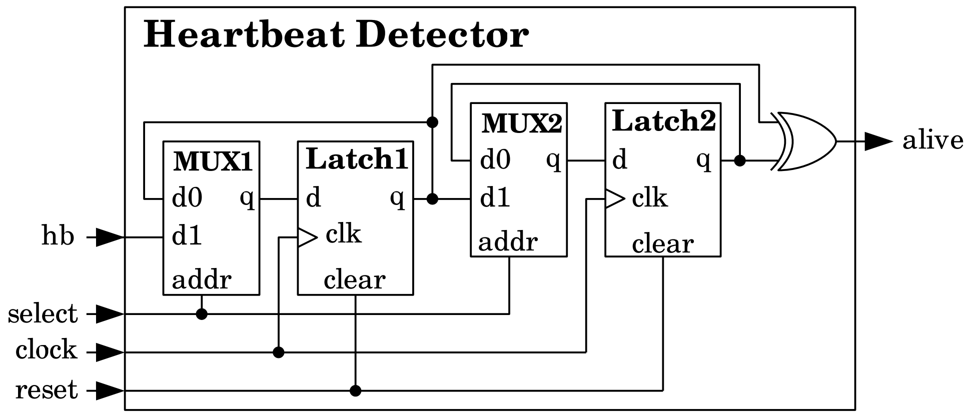


Figure 3.18: The Heartbeat Detector

When the `select` input of a *Heartbeat Detector* module is asserted, a rising edge at the clock input causes `Latch1` to acquire the state of the heartbeat signal, at the `hb` input. At the same time, the previous state of the heartbeat signal is transfered from `Latch1` to `Latch2`. 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 `select` 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, multiplexers `MUX1` and `MUX2` keep the contents of latches 1 and 2 unchanged, by routing the current values of the latches back to their inputs.

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 `select` input is not asserted, until just after all of the `alive` outputs are sampled for inclusion in the header of the latest data frame.

3.3 The State Generator

3.3.1 Registers

As previously documented, the *Control Gateway* manages an array of 8-bit registers that the computer can read and write via its parallel port. These registers are also visible to the *State Generator*, which can inteprett their contents however it wishes, and the *State Generator* is informed, by the *Control Gateway*, whenever a given register is written to by the computer, via a per-register transient flag. In practice the *State Generator* divides its intepretation of the registers into two distinct classes.

- **Action registers**

  When an action register is written to by the computer, the *State Generator* immediately does something in response. An example is the register whose value tells the FPGA to start a new scan or intra-scan.

- **Configuration registers**

  When a configuration register is written to, the only thing that happens immediately is that the new value is stored within the register bank of the *Control Gateway* for later use. The new value has no effect on the currently running scan or intra-scan. Subsequently, when a new scan or intra-scan is commanded, the updated values of these registers are copied into active configuration registers within the *State Generator*, and used to determine the behavior of the new scan. Thus, while configuration registers can be written to at any time, their new values only take effect when the next scan or intra-scan is initiated.

The list of action registers

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

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

  The bit assignments of the register are as follows.

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

The meanings of the bit-value names are:

- **sync**
  If 0, start the new scan as quickly as possible. If 1, start the new scan at the next rising edge of the 1PPS signal.
- **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.
- **switch_a**
  If 0, phase-switch A should be held in the state specified by the `close_a` bit, for the duration of each phase-switch cycle. If 1, phase-switch A should be toggled on and off during each phase-switch cycle.
- **switch_b**
  If 0, phase-switch B should be held in the state specified by the `close_b` bit, for the duration of each phase-switch cycle. If 1, phase-switch B should be toggled on and off during each phase-switch cycle.
- **close_a**
  If 0, phase-switch A should be opened at the start of each phase-switch cycle. If 1, phase-switch A should be closed at the start of each phase-switch cycle.
- **close_b**
  If 0, phase-switch B should be opened at the start of each phase-switch cycle. If 1, phase-switch B should be closed at the start of each phase-switch cycle.
- **X**
  The value of this bit is currently unused, and should be assigned 0.

• `conf_integ_reg` – Queue a new integration configuration.

Append one entry to the queue of future integration configurations. This register is to be written to whenever the computer receives a `cnf_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 `cnf_intr` CPU-interrupt to ask the computer for the configuration of the first integration of the new scan. The first integration of the scan can not procede until this information has been received. The second time that a `cnf_intr` interrupt is received by the computer, after initiating a new scan, the configuration of the second integration of the scan should be written to the `conf_integ_reg` register, and so on. Since the receipt of a `cnf_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 integration configurations, the computer must not write to the `conf_integ_reg` register at any other time.

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

Register	The value of each bit							
address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
01	X	X	X	X	X	X	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.
– **X**
  The value of this bit is currently unused, and should be assigned 0.

The list of configuration registers

- `phase_state_dt_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 mimimum and maximum supported values of 250 and 65535, which correspond to durations of 25μs and 6.5ms.

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

Register	The value of each bit							
address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
02	`b15`	`b14`	`b13`	`b12`	`b11`	`b10`	`b9`	`b8`
03	`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 signicant bit, and `b15` denoting the most significant bit.

- `phase_switch_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 `phase_switch_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
04	`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 signicant bit, and `b7` denoting the most significant bit.

- `diode_rise_dt_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_dt_reg` register are as follows.

Register address	The value of each bit							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
05	`b31`	`b30`	`b29`	`b28`	`b27`	`b26`	`b25`	`b24`
06	`b23`	`b22`	`b21`	`b20`	`b19`	`b18`	`b17`	`b16`
07	`b15`	`b14`	`b13`	`b12`	`b11`	`b10`	`b9`	`b8`
08	`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 signicant bit, and `b31` denoting the most significant bit.

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

• `integ_period_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 accumlators.

The configuration bits within the `integ_period_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 signicant bit, and `b15` denoting the most significant bit.

• `roundtrip_dt_reg` – The CCB/receiver roundtrip 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.

This is an 8-bit value, expressed as a multiple of the 100ns ADC sampling interval. The maximum supported roundtrip delay is thus 25.6μs.

The configuration bits within the `roundtrip_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
13	`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 signicant bit, and `b7` denoting the most significant bit.

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

  This register sets the maximum rate at which the CCB is allowed to generate interrupts, expressed as the minimum interval between interrupts. It is a 5-bit number which has 1 added to it, before being scaled by 25.6μs, to arrive at the actual holdoff interval. Thus the range of supported holdoff intervals is 25.6μs $\leftrightarrow$ 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
14	`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 signicant bit, and `b4` denoting the most significant bit.
- **X**
  The value of this bit is currently unused, and should be assigned 0.

3.3.2 The 1PPS Gateway

The external GBT 1PPS signal is a train of 1μs pulses, with a period of 1 second, and an amplitude of 4V. Each pulse signals the start of a new second of UT. The job of the 1PPS gateway is to convert each 1μs pulse into a pulse whose rising edge coincides with a rising edge of the FPGA clock, and whose duration is a single FPGA clock cycle. The circuit shown in figure 3.19 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

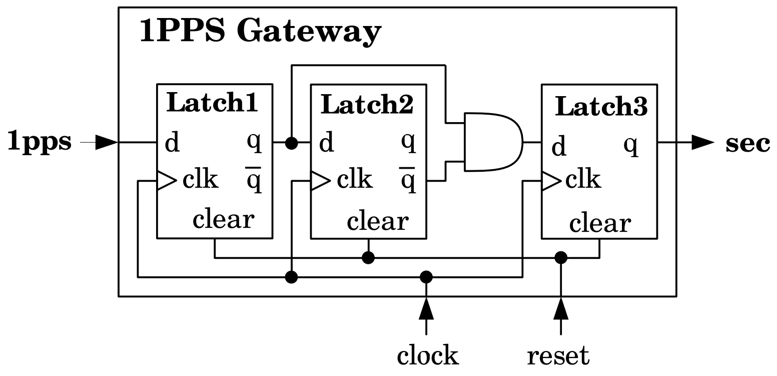


Figure 3.19: The 1PPS Gateway

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.

3.3.3 Dealing with round-trip delays and settling times

A serious complication in the design of the *State Generator* is the fact that there are significant round-trip delays and finite settling times involved in controlling the cal-diodes and phase switches.

The round-trip delay

When any of the receiver control signals are toggled, the initial effect of this on the detected signal is not seen by the slave FPGAs for a few clock cycles. First of all the digital switching control-signals are delayed by the RFI filters, opto-isolators and cables that separate them from the receiver. In particular, opto-isolators generally have propagation delays of around 100ns, which corresponds to one FPGA clock cycle. Then if the switches start to respond as soon as the control signal finally arrives, the 8-pole 2MHz Bessel low-pass filter in the receiver, delays the perturbed signal by another 250ns, the 4-stage pipeline in the ADCs delays it by another 300ns, and the input latch in the slave FPGAs delay the use of the digitized signal by another 100ns. Thus, even if one only accounts for these known delays, and ignores all other possible sources of delays, such as the RFI filters, the detectors, and the phase-shift in the ADC clock signals, it is clear that there will be a delay of at least 750ns, or 7.5 clock cycles, between the master FPGA toggling a receiver control signal and the slave FPGAs seeing any resulting perturbation in the digitized signal. Henceforth this delay will be referred to as the round-trip delay. The round-trip delay is assumed to be identical for all switching signals, and in practice, is configured as an integral number of clock cycles, via a single 8-bit configuration register.

Within the *State Generator*, the significant round-trip delay presents a challenge. Whereas the signals that it generates to control the phase switches and calibration diodes, can be sent to the receiver as they are generated, the corresponding control signals that are destined for the slave FPGAs and the *Data Dispatcher* must not be arrive at these modules until many clock cycles later, after the roundtrip delay has passed. It would require an impractical number of gates to implement this delay by delivering the slave and *Data Dispatcher* signals via variable-length shift-registers, clocked by the 10MHz system clock, since there are a lot of signals that would need to be delayed, and each signal would require a shift register of around 10 stages. Fortunately, only the `blank` signal actually potentially needs to change on every 100ns clock cycle, and the next smallest time step is the $\geq 25\mu$s duration of a single phase-switch state, which is much greater than the anticipated $\approx 1\mu$s roundtrip delay. A workable strategy is thus as follows.

1. Clock the the main state machine with the phase-switching clock, instead of directly from the main 10MHz clock. Use this to generate both the receiver switch-control signals, and the corresponding slave and *Data Dispatcher* signals, excluding the `blank` signal.

   Also have the rising edge of the phase-switching clock load the roundtrip delay down-counter, which is decremented at each 10MHz clock cycle, until it underflows.

2. Whenever the roundtrip down-counter underflows, latch the slave and *Data Dispatcher* control signals from the main state generator, into registers whose outputs go to the corresponding inputs of the slaves and the *Data Dispatcher*.

   At the same time, load a count of the number of samples that must be blanked, into a down-counter whose underflow output is used to drive the `blank` inputs of the slaves, via a *not* gate, and which is decremented by one at each 10MHz clock cycle, until the underflow output is asserted.

In summary, while most of the signals are delivered by what are essentially a single entry shift registers, clocked by the slow phase-switching clock signal, the `blank` signals are generated in the slave time-domain, from information latched from the start of each phase-switch cycle.

The settling time delays

The round-trip delay described above only encompasses the time that passes before the initial effects of a switch transition become apparent. It doesn't include the time taken for the effect of a switch to reach a settled state, after it starts changing. Unlike the round-trip delay, the settling-time delay depends on what is switching, and in the case of the cal-diodes, whether it is being switched on or off. In particular, it seems likely that the switch-on time of the cal-diodes could be significantly greater than their switch-off time, due to thermal and loading effects when they are turned on. In practice the rise and fall times of the diodes, and the settling times of the phase-switches, are configurable via configuration registers, so there are no built-in assumptions about their relative magnitudes. When one or more switches are toggled, the overall settling time is determined by the switch with the longest settling time.

When the calibration diode switches are toggled, all subsequent integrations that fall within the settling time of the cal signal, are flagged in the data stream that is sent to the computer. This is achieved by holding the `stable` input-signal of the *Data Dispatcher* low.

During the settling time of phase-switch transitions, the `blank` input signal of the slaves are asserted, to arrange for affected ADC samples to be excluded from the integration sums.

3.3.4 Clock Conditioner

still TBD

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 CCB PISO component

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

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

A single node of such a PISO is shown in figure 3.20.

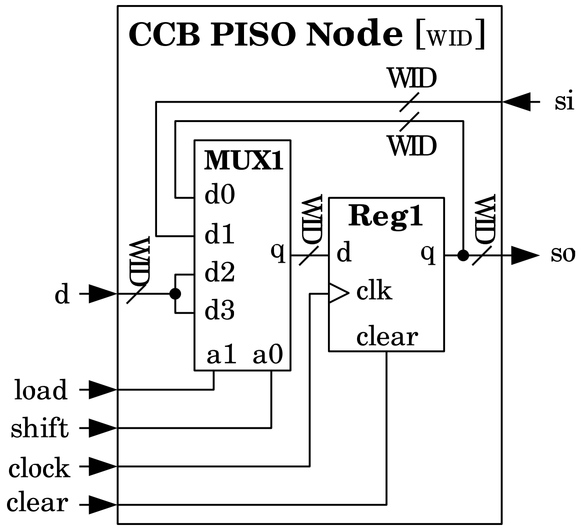


Figure 3.20: One node of a CCB PISO component

Note that the parameter `WIDTH` specifies the number of bits serially shifted in and out of the serial inputs and outputs, `si` and `so`, or parallel-loaded via the `d` input. At the rising edge of the clock, the PISO performs the following operations, according to the states of teh `shift` and `load` inputs.

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

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

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

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

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

  Whenever the `load` input is asserted, regardless of the state of the `shift` input, multiplexer `MUX1` routes the value at the `d` input of the node to the input of register `Reg1`.

Thus the contents of the register get replaced with whatever value was at the `d` input of the node.

*CCB PISO Node*s are strung together in a chain to form a PISO, as shown in figure 3.21.

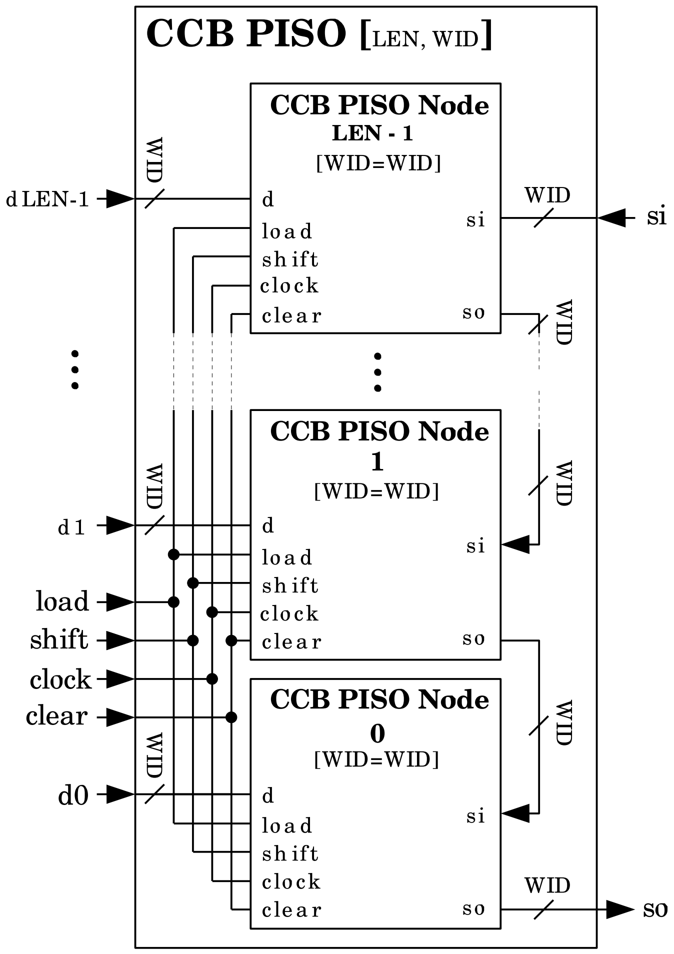


Figure 3.21: A complete CCB PISO component

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