The designs of the master and slave CCB FPGAs

[Document number: A48001N004, revision 2]

Martin Shepherd, California Institute of Technology

May 5, 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 the VHDL code that synthesizes the lowest level components displayed, and connects them together.

[This is still a work in progress]

Contents

List of Figures

Chapter 1

Introduction

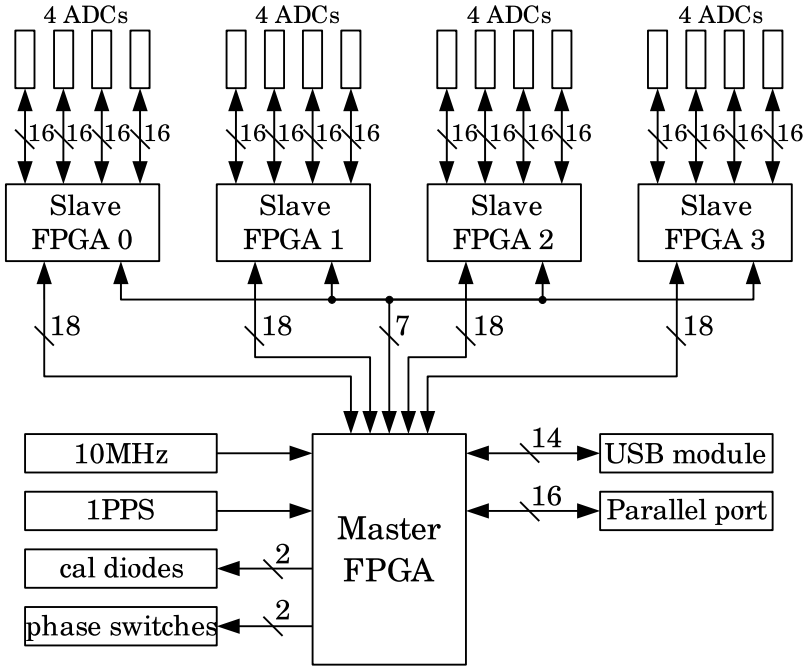

Figure 1.1: An overall summary of the FPGA connections

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 from the computer's EPP parallel port, delivers interrupts to the computer over the same EPP parallel port, dispatches observed data to the computer over a USB link, and controls calibration diodes and phase switches in the receiver. 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 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.

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, shown 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 14 bit data-signals and 1-bit overflow signal of each of the 4 ADCs, are simultaneously latched into a corresponding 15-bit register at the start of every FPGA 10MHz clock cycle. Simultaneously, the previous values of these registers are read by integrator components. At the end of each integration, the outputs of the integrators are flash loaded into a PISO, ready to be read out by the master FPGA. In dump mode, the output of one of the 15-bit ADC inputer-registers is siphoned off, to be read directly by the master FPGA, and the outputs of the integrators and its PISO are ignored.

2.1.1 Normal integration mode

When not in dump mode, the output of each ADC, delayed by one clock cycle by the input register, is read into an integrator component.

The integrator component, which will be detailed shortly, either ignores the new sample, if the `drop` signal from the master FPGA is asserted, or adds it to the accumulation bin specified by the 2-bit `phase` signal, as received from the master FPGA.

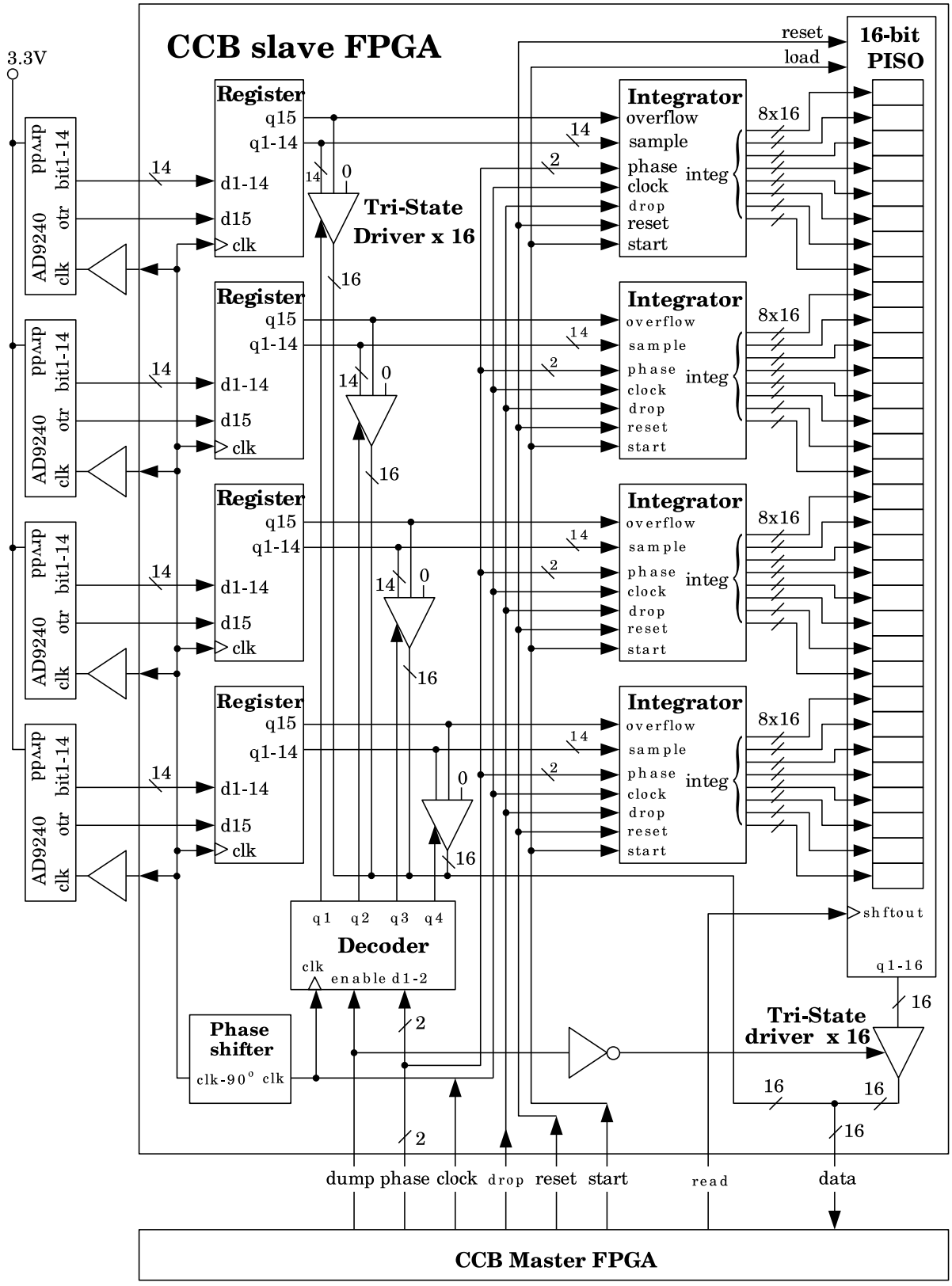

Figure 2.1: The top-level design of the slave FPGA

At the end of each integration period, the master FPGA asserts the `start` signal for one clock cycle. This causes the output PISO, on the right of the diagram, to flash-load the contents of all of the 32-bit accumulator output bins of the integrators, and at the same time causes the integrator components to zero all of their accumulators, and add the latest ADC sample to the appropriate empty bin, unless the `drop` signal is asserted.

The accumulator bins are 32-bits wide, but there are only 16 pins available for clocking the data out of the slave FPGA. Therefore a 16-bit wide PISO is used, with two entries assigned to each accumulator bin. Since there are 4 accumulator bins per integrator, 4 integrators, and 2 PISO entries per accumulator, the 16-bit wide PISO will be 32 entries in length.

One clock-cycle after the master FPGA asserts the `start` signal, the output of the PISO is valid, and the master FPGA can then use the `read` input signal to clock out each new PISO entry over the output `data` lines.

2.1.2 Dump mode

In dump mode, although the integrators still integrate data, the tri-state output driver, following the PISO, is placed into a high impedance state, to enable one of the 16-bit tri-state drivers at the outputs of the ADC input registers, to instead direct the output of a chosen ADC, directly to the `data` output of the slave FPGA. In this mode, the 2-bit `phase` signal is re-interpreted as the address of the ADC to be dumped, so an address decoder, which is only enabled when the `dump` signal is asserted, is used to enable the chosen tri-state buffer, according to this address.

At the end of dump mode, the master FPGA de-asserts the `dump` input and then, when a new scan is ready to start, it briefly asserts the `start` signal to flush the contents of the integrators, and start a new integration. At this point the PISO contains the garbage contents flushed from the integrators, so its contents are not sent to the computer, and are simply left in the PISO, where they are subsequently overwritten when the `start` signal is again asserted at the end of the new integration.

2.1.3 The ADC clock signal

Note that the clock signal that is transmitted to the ADCs 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.

2.1.4 External connections

The FPGA I/O pins should be configured for 3.3V logic levels.

Provided that the `DRVDD` pins of the ADCs are connected to a 3.3V power-supply, then the output data pins of the ADCs can directly drive the corresponding input pins of the FPGA. This may not be the case if there are cables or filters in between. Unfortunately, the ADC clock input appears to require higher CMOS levels, regardless of the voltage at the ADC `DRVDD` pin. Hence the individual 3.3V to CMOS buffer amplifiers between the FPGA clock output pins and the ADC clock input pins. If the clock output pins of the FPGA are close together, then presumably the number of FPGA clock output pins could be reduced; as could the number of separate buffer chips.

Note that an assumption in this design, which may not be correct, is that the interconnections between master and slave FPGAs, and between the slave FPGAs and the analog board, will be via daughter-board connectors, rather than via cables and EMI filters. The scenario in mind has a central board housing the master FPGA, the USB interface and the parallel port interface and, on the component side, connectors for 4 daughter-boards holding the slave FPGAs, and on the other side of the board, shielded by the ground plane, 4 daughter-boards housing the analog electronics. The connections from the ADCs to the slave FPGAs would thus go via two levels of daughter card connectors. If a significantly different scheme is adopted, which includes inter-board cabling and/or EMI filtering, then extra buffer amplifiers may need to be added for the data lines, along with some means to compensate for any rise time delays added by EMI filters.

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 central brain of this design, is the *State Generator* component, which orchestrates the timing and the values of all control signals that go to the other components and 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.

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:

- **The address write-cycle**

  The associated data-byte is interpreted as the address of one of the registers in the

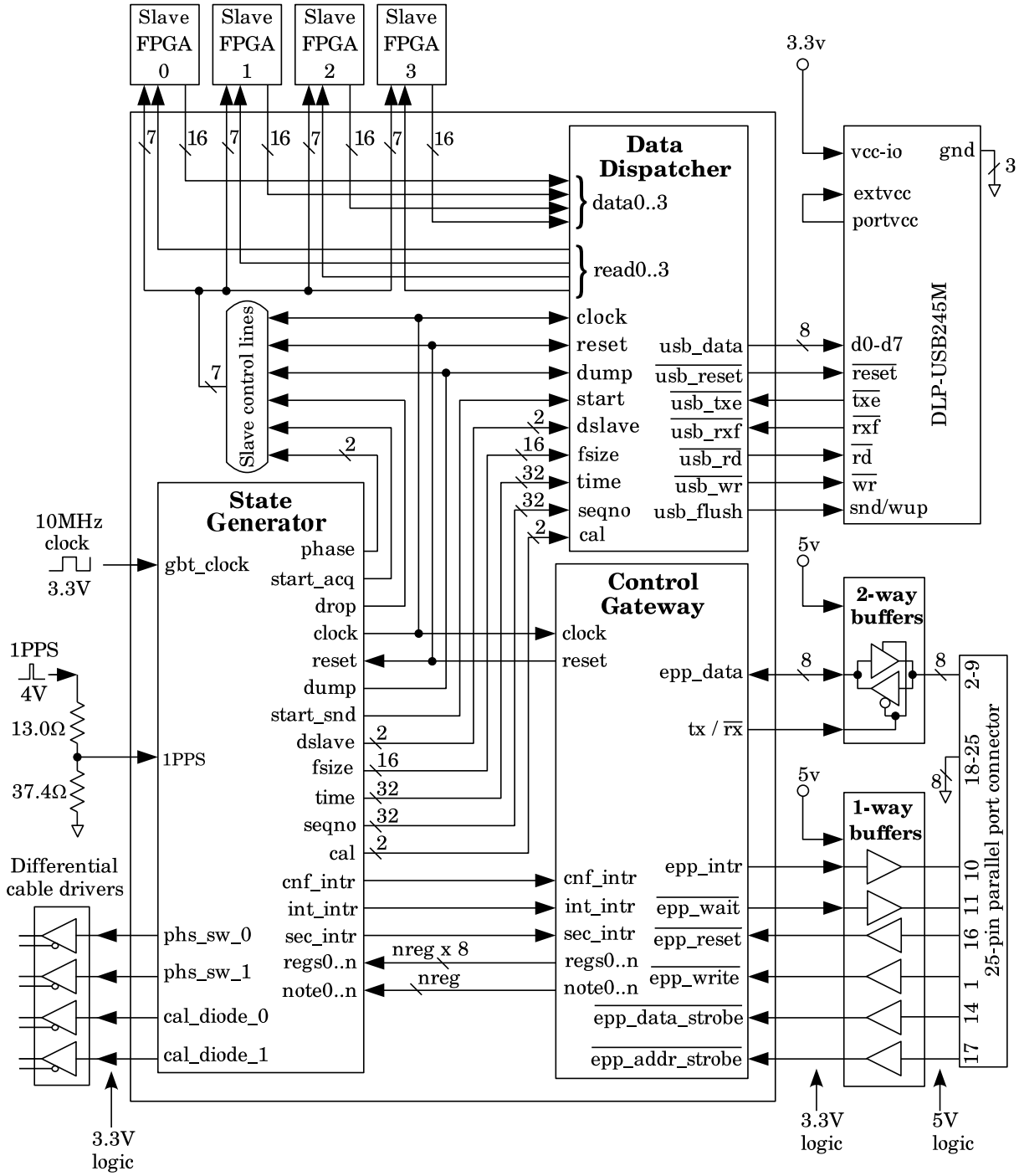

Figure 3.1: The top-level design of the master FPGA

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 previously indicated during the last address write.

- **The data read-cycle**

  The returned data-byte is the value of the register that was previously 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 be 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 will happen, on average 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 rest of the CCB, where the 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, followed, one cycle later, by a corresponding output flag which indicates, for one clock cycle, when a new value for that register has been received. The bytes in the *EPP Register Bank* can also be read-back by the CPU, via EPP data-reads.

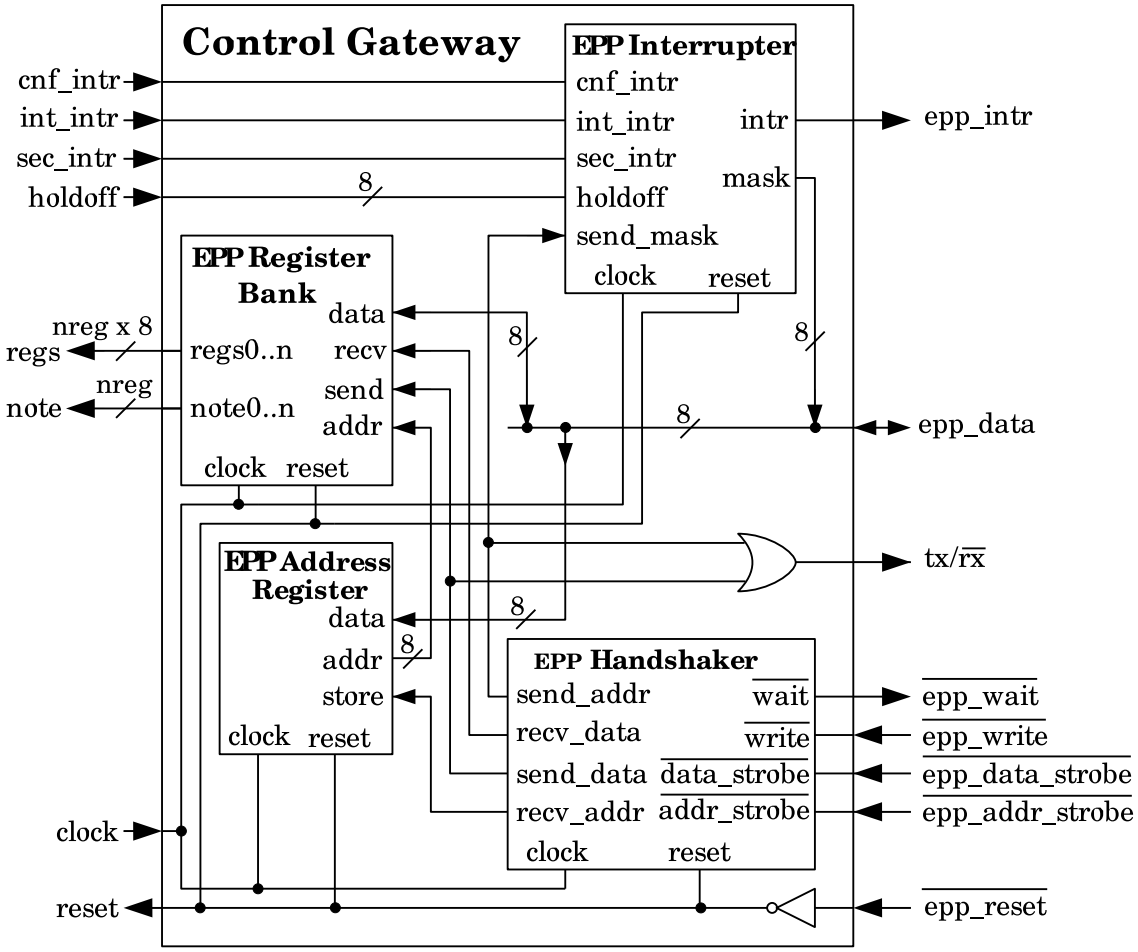

Figure 3.2: The Control Gateway

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. 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 whose output is used to route subsequent EPP data transactions to the specified register in the *EPP Register Bank*.

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 resent. Note that there is no danger that a resent 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 this is cleared as it is read.

To avoid a tug-of-war with the CPU, the FPGA only drives the `data` lines when explicitly requested, as indicated by either of the `send_data` or `send_addr` outputs of the *EPP Handshaker* being asserted. Thus the external `data` line transceivers are configured to passively receive data from the computer, except when either of the former signals are 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, the value of the $\overline{\texttt{write}}$ signal, and the value of the 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, to 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,

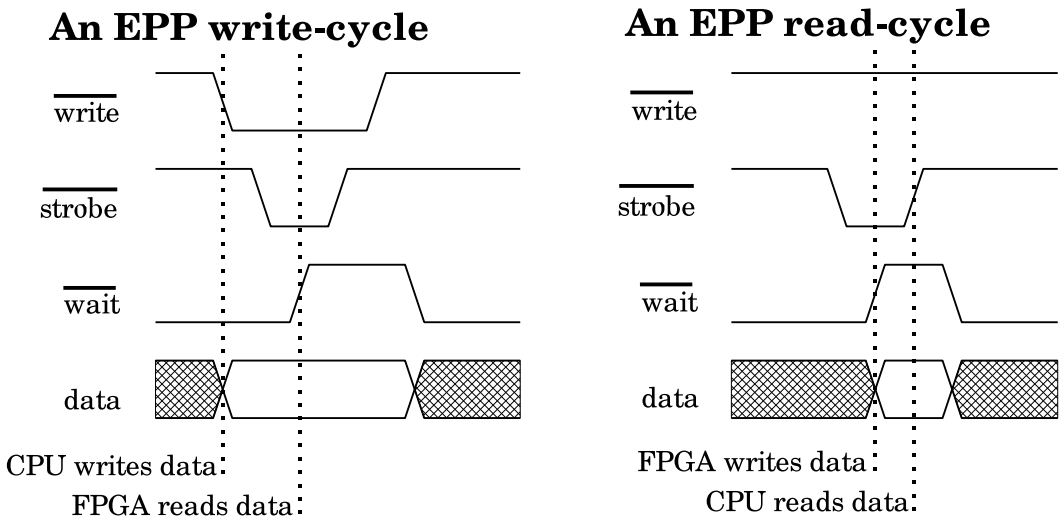

Figure 3.3: The standard EPP I/O cycles

and invert them. All of the logic paths that follow them have a full clock cycle to settle, before being latched by latches 3-7. In particular, when a rising edge of either of the EPP $\overline{\texttt{data_strobe}}$or $\overline{\texttt{addr_strobe}}$signals violates the setup and hold times of latches 1 and 2, the corresponding latch could go into a metastable state, so this extra clock-cycle for the metastable state to work itself out, should greatly increase the reliability of the circuit. Indeed, apparently the conventional recommendation for interfacing asynchronous logic to clocked logic is to use two chained latches like this. The drawback of this is that it adds a clock cycle 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. If it were a problem, the simplest solution would be to increase the FPGA clock frequency to make one FPGA clock cycle less than half the duration of the standard EPP 8MHz clock period, although this would of course involve a trade off, since the probability of a metastable state persisting for the shorter period of the higher frequency clock, would then be higher.

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, which corresponds to 0.8 and 1.6 EPP 8MHz clock cycles. Thus most of the time, the standard 4-cycle EPP I/O transaction will be lengthened to 5 8MHz cycles, and thus last 0.625μs instead of 0.5μs.

Note that the $\overline{\texttt{wait}}$ and $\overline{\texttt{write}}$ signals aren't pre-latched, like the $\overline{\texttt{strobe}}$signals, 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.

While it drives the EPP $\overline{\texttt{wait}}$ signal high, the *EPP Handshaker* also asserts one of the `send_data`, `recv_data`, `send_addr`, and `recv_addr` outputs, both to indicate what type of

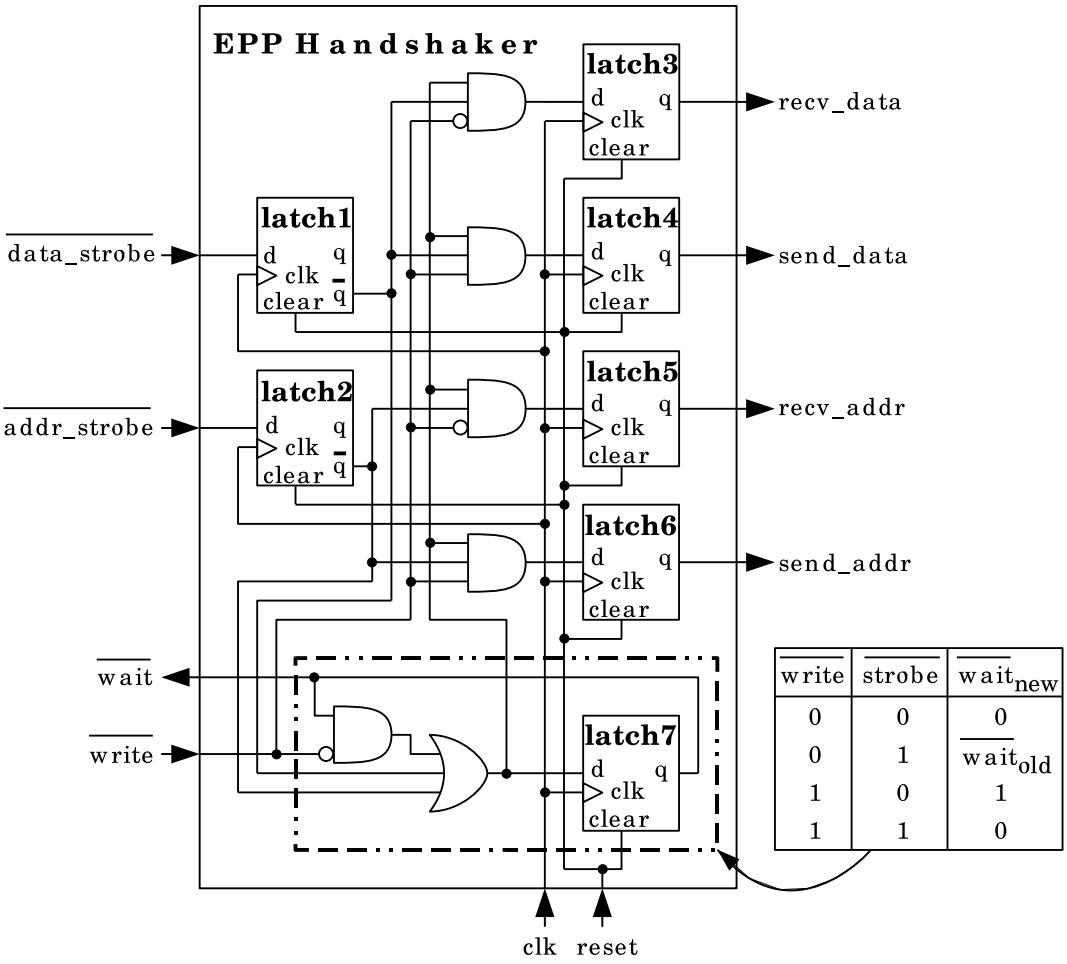

Figure 3.4: The EPP Handshaker

I/O transaction has been requested, and to signal the period during which the `data` signal should be valid. The `send_data` and `send_addr` outputs indicate when the FPGA should drive a data byte or the interrupt mask onto the shared `data` lines, respectively, and the `recv_data` and `recv_addr` signals respectively indicate when a data or address byte should be read from the `data` lines. Data are expected to be latched to or from the `data` lines on the rising edges of these signals, in accord with the diagram of the standard EPP I/O cycles. In the case of FPGA data-read and address-read transactions, the corresponding source of the requested data-byte is expected to drive the `data` lines while the associated `send_data` or `send_addr` signal is asserted.

The EPP address register

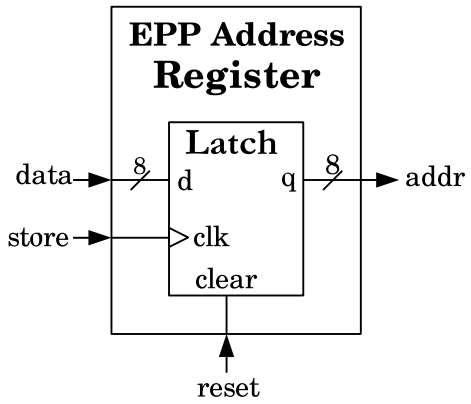

Figure 3.5: The EPP Address Register

The *EPP Address Register*, as shown in figure 3.5, holds the address of the data register that should be written to, or read from, by subsequent EPP data-write and data-read cycles. It is implemented as a simple octal latch which latches in the contents of the `data` lines on the rising edge of the `store` input signal. Note that the `store` input signal is driven by the `recv_addr` output of the *EPP Handshaker*. Thus the data-byte of each EPP address-write transaction ends up in this register. The `addr` output is the index of the addressed data register, and is thus connected to the `addr` input of the *EPP Register Bank* module.

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 and command opcodes sent by the CPU. The `addr` input, which comes from the *EPP Address Register* module, selects which register should latch data from the `data` lines when the `recv` input goes high, or which register should drive its value onto the `data` lines, while the `send` signal is high. Register selection is implemented

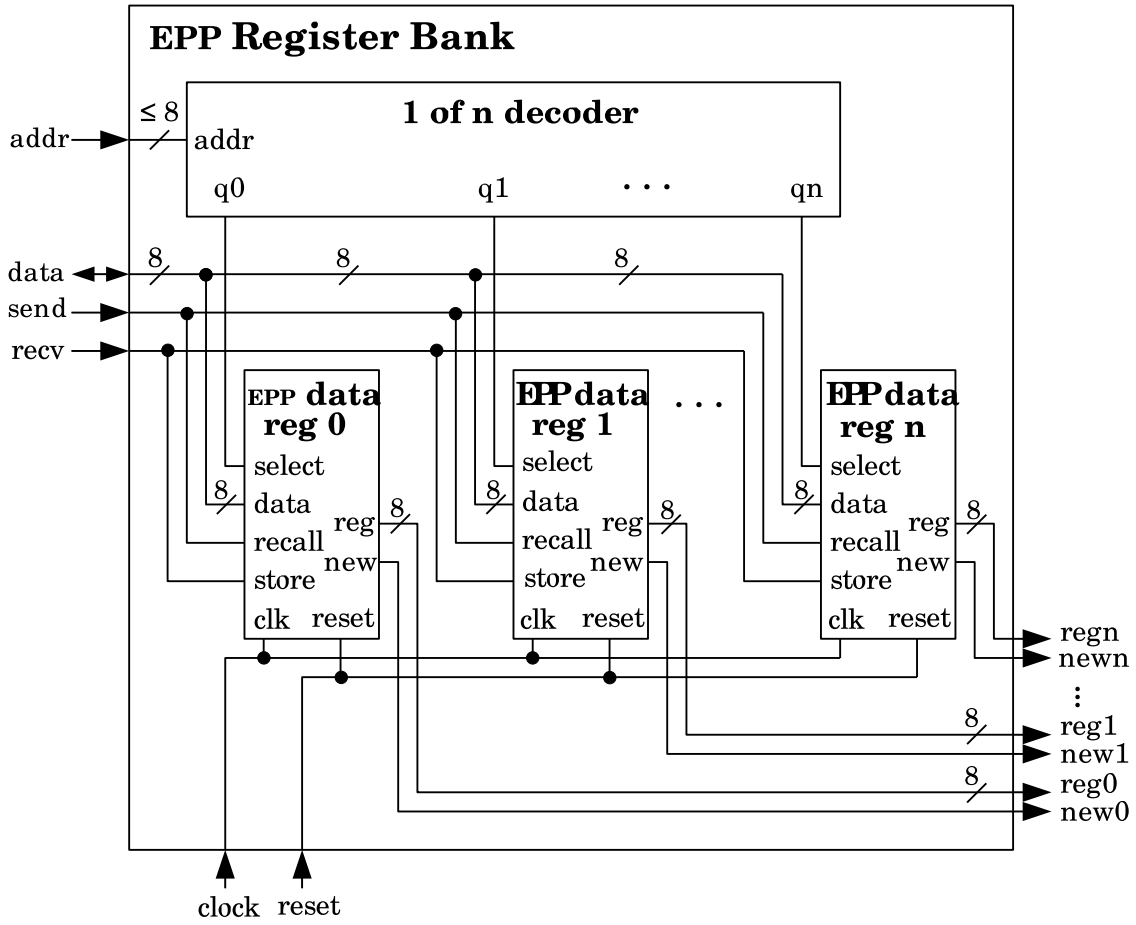

Figure 3.6: The EPP Register Bank

via an conventional address decoder, which asserts its *i*'th q output when the address has the value *i*. Although the `addr` input provides an 8-bit address, which allows up to 256 registers; in practice much fewer registers will be needed, so a smaller decoder that ignores some of the most significant bits, can be used.

The individual data registers are implemented as *EPP Data Register* modules, implemented as shown in figure 3.7. Since only one register at a time can be addressed by EPP data-read and data-write cycles, each data register ignores its `store` and `recall` signal inputs except when its `select` input is asserted. When both this and the `recall` signal are high, the embedded octal register drives its current value onto the `data` lines, via a tri-state buffer. Alternatively, when the `select` signal is asserted, and the `store` input signal goes high, it does 2 things. First the rising edge of this signal causes the byte on the `data` lines to be latched into the embedded register. Then one clock cycle later, after the register has settled, latches 2 and 3 generate a single pulse, one clock-cycle in length at the `note` output. Thus the `note` output of each EPP data register is used to signal other parts of the firmware when it has been updated. For example, the FIFO which queues integration-specific configuration data will use this to shift each new value of the configuration register into the FIFO. Similarly registers that are to be interpreted as command opcodes will use this signal to trigger a command, without the opcode value within the register needing to change.

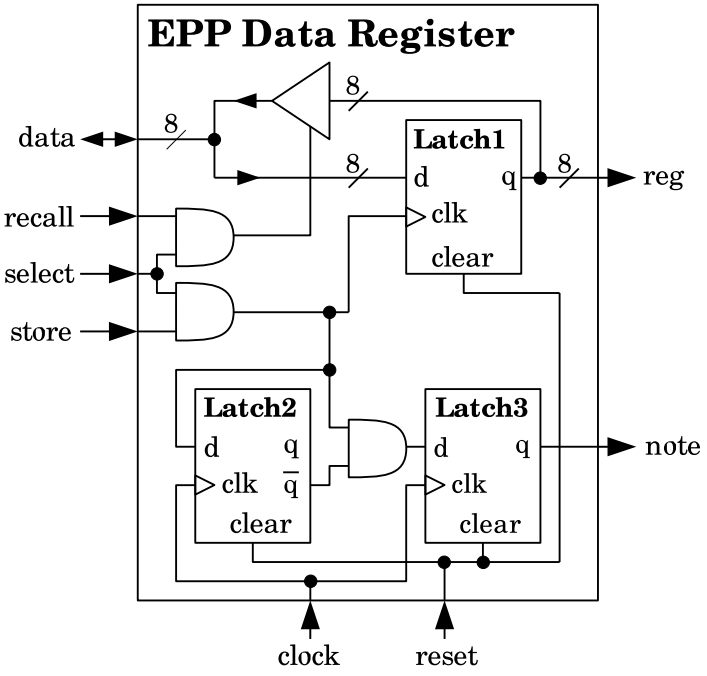

Figure 3.7: An EPP Data Register

The EPP Interrupter

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

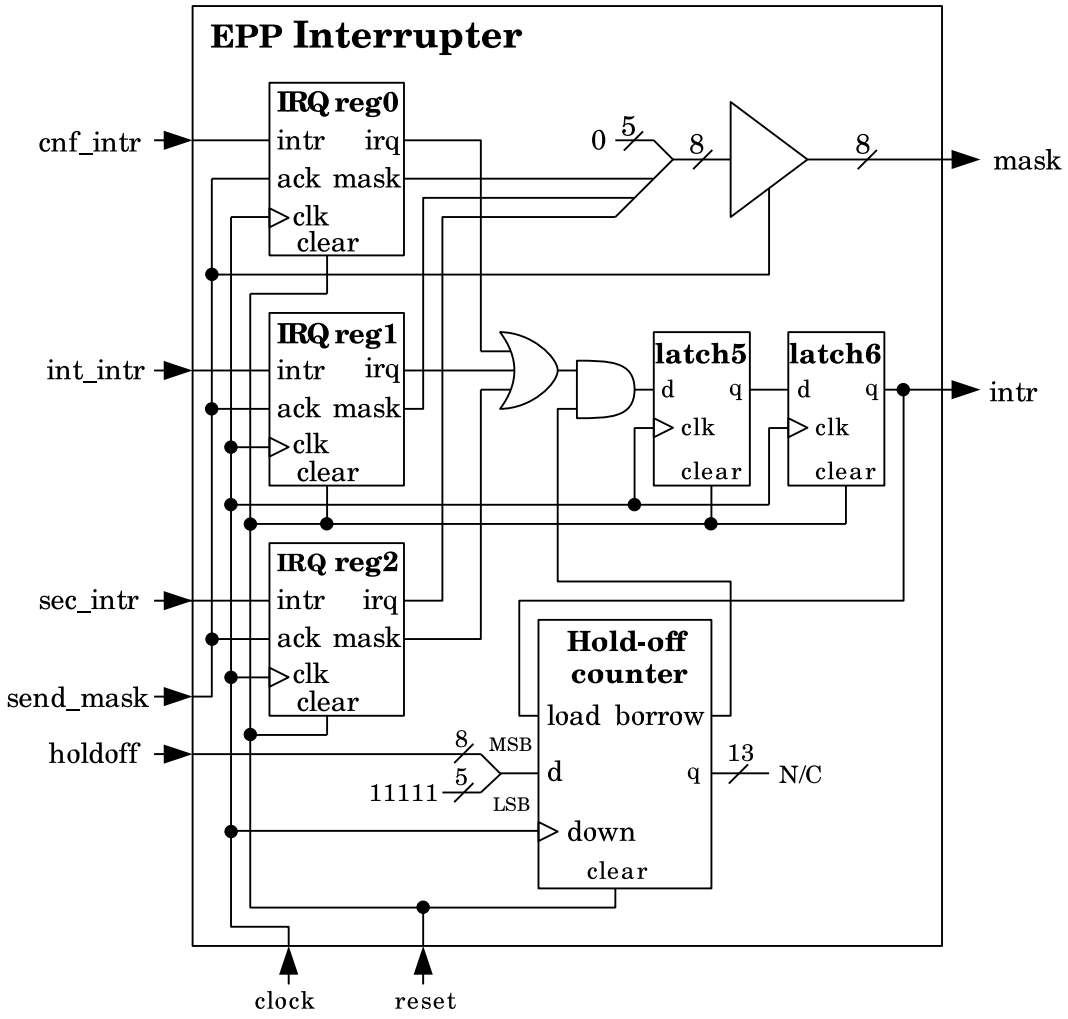

Figure 3.8: The EPP Interrupter module

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 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 bog-down the CPU, 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 which event sources have requested interrupts.

When a particular event-source in the FPGA wishes to notify the computer of a new event, it asserts the associated one of the `cnf_intr`, `int_intr` or `sec_intr` interrupt-request inputs of the *EPP Interrupter* for at least one clock cycle. On the following 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 holdoff 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 cycle (ie. 1.6 EPP 8MHz clock cycles). Simultaneously, it reloads the holdoff 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 rising edge of the `send_mask` input latches the states of the IRQ registers to their respective `mask` outputs, which are then driven onto the parallel-port `data` lines via the tri-state buffered `mask` output of the *EPP Interrupter*. One clock cycle later, all of the IRQ registers whose latched `mask` outputs are asserted, are de-asserted, ready to register a new event. Note that the `mask` outputs remain unchanged when the register is de-asserted, since they must continue to drive the `data` lines until the `send_mask` input goes low.

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 would then keep the CPU busy for a while acknowledging stale events, after interrupts were re-enabled. 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.

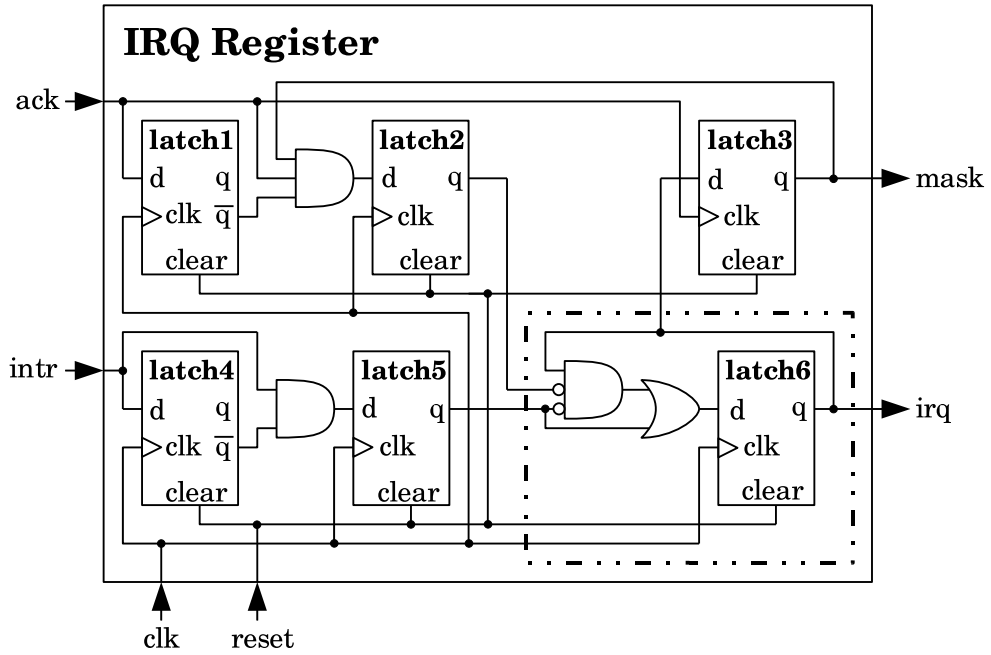

Figure 3.9: An Interrupt Request (IRQ) Register

The part of the circuit in the bottom right of the diagram, enclosed in a dashed box, is the IRQ register itself. It takes interrupt requests from the output of latch 5, and interrupt acknowledgements from the output of latch 2. An asserted interrupt-request input causes the `irq` output to go high, and thereafter stay high until the interrupt acknowledgement input is next asserted. If both a new interrupt request and an acknowledgement signal arrive on the same clock cycle, the new interrupt request takes precedence, and asserts the `irq` output. The value of the `irq` output is latched to the `mask` output, on the rising edge of the `ack` signal, ready to be driven onto one of the EPP `data` lines by a tri-state buffer in the parent *EPP Interrupter*.

The purpose of latches 4 and 5 is to generate a single pulse, one clock cycle in length, each time that the `intr` input goes high, regardless of how long the input signal remains high. This is needed to avoid a race condition with the acknowledgement signal. The rising edge of the single-cycle pulse actually occurs one clock cycle after the `intr` signal first goes high.

Similarly, the purpose of latches 1 and 2, is to convert a potentially multi-cycle pulse at the `ack` input into a single-cycle pulse at the acknowledgement input of the dashed circuit. In this case however, the one cycle delay in the initiation of this pulse, and an extra input to the AND gate, are exploited to ensure that the delayed acknowledgement input to the dashed circuit is not generated at all, unless the interrupt mask bit that was latched to the `mask` output, one clock cycle earlier, is high. In other words the IRQ register isn't actively de-asserted by an acknowledgement signal, unless it was asserted when its value was reported

to the computer. This guarantees that the `mask` output can be trusted to always report if at least one interrupt request occurred since the last time that the `mask` was latched. If, instead, the IRQ register were de-asserted without regard for whether an interrupt event had been reported to the computer, then any interrupt request that arrived at the `d` input of latch 6, on the same clock edge that latched the corresponding `q` output into latch 3, would get discarded by the `ack` signal one cycle later, and the event would never be reported to the computer.

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 resent 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 link. All communications over the USB bus are directed from the FPGA to the computer. Thus, although the read (`rd`) and read-enable (`rxf`) pins of the USB interface are shown as inputs to the *Data Dispatcher*, there are no plans to use them at the moment.

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

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

The two of these modules that I bought for testing the FT245BM, I got from a company called Saelig (`www.saelig.com`), which is an official US distributor for the FT245BM. The

modules arrived overnight. Since then, I have noticed that Mouser Electronics carries them as well. Their catalog number at Mouser is 626-DLP-USB245M, and they cost $25.

3.2.1 The internals of the Data Dispatcher

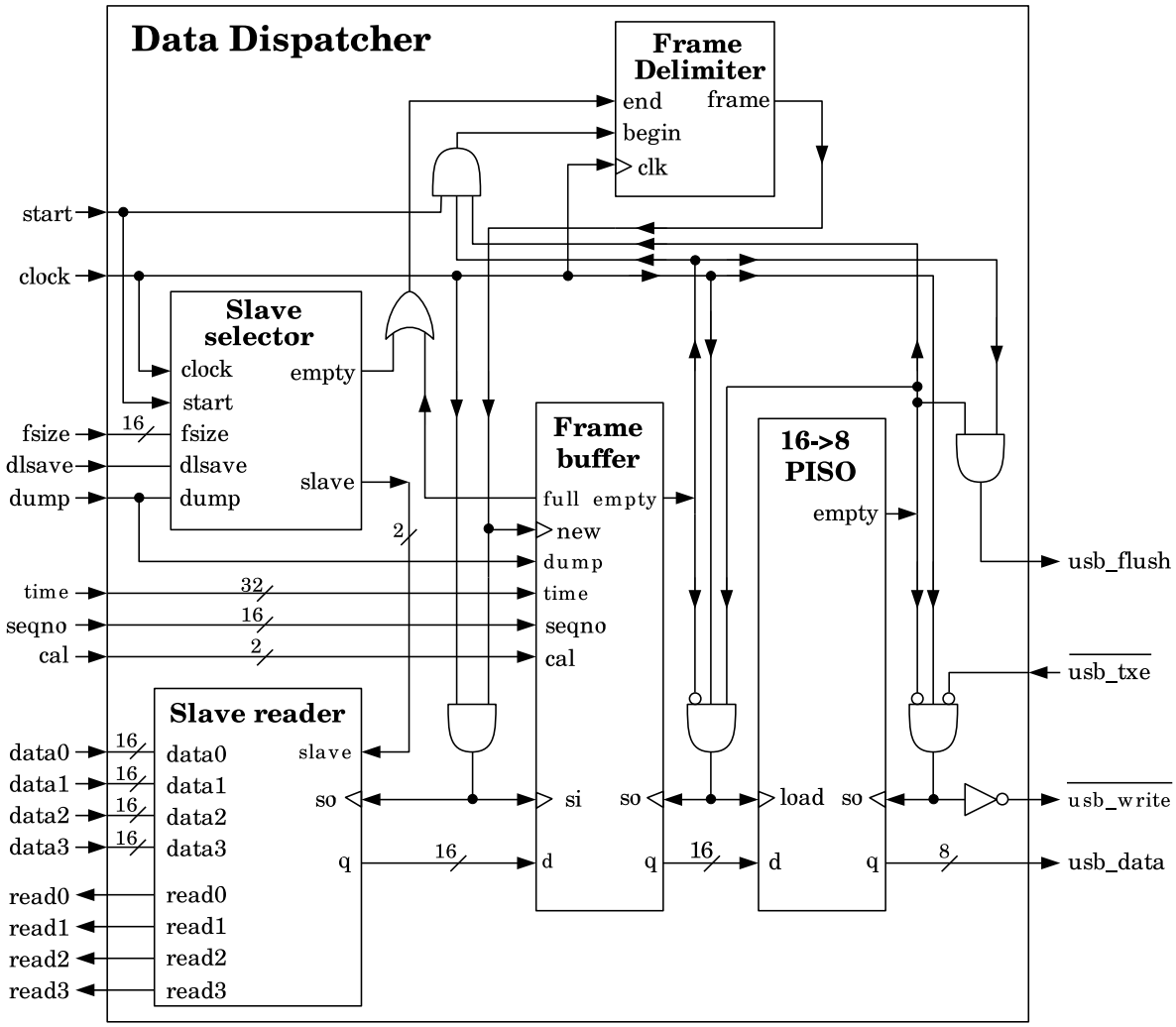

Figure 3.10: The Data Dispatcher

Figure 3.10 shows the building blocks of the Data Dispatcher, and how they are interconnected. All data available from one slave at a time, are read by the Slave Reader and passed on to the Frame Buffer. The current slave to read from, is selected by the *Slave Selector*, which cycles through the slaves in reverse numerical order, when reading out integrated data, or continually reads from the slave specified by the `dslave` input signal, when in dump mode. The `frame` output signal of the *Frame Delimiter* controls the packaging of output

data frames within the *Frame Buffer*. When the `frame` signal goes high, a new output data frame is initialized, and data from the slaves start to be transfered to the Frame Buffer. When all data from the slaves have been transfered, or the frame-buffer becomes full, the `frame` signal goes low, to terminate the frame. The `frame` signal doesn't go high again until the next rising edge of the `start` signal from the `start_snd` output of the *State Generator*, and even then, it only goes high if the previous contents of the *Frame Buffer* have been completely transfered to the CPU over the USB link. These measures prevent a new frame from trampling on an incompletely sent frame, and prevent temporary buffer-full conditions, when in dump mode, from creating unpredictable sampling gaps within a frame.

The *Frame Buffer* contains a large FIFO for the slave data, plus a small PISO in which the frame header is assembled. The frame header, which is flash loaded into the PISO, on the rising edge of the `frame` signal, consists of a time-stamp, a scan sequence number, the states of the cal-diodes during the just-completed integration, frame-start and dump-mode indicators.

Once a new frame has been started, the contents of the *Frame Buffer* are clocked out, 16-bits at a time, starting with the frame header in the PISO, and followed by the slave data from the FIFO. Each chunk is loaded into a 2 entry, 8-bit wide PISO, such that 8-bits at a time then can be clocked out to the 8-bit FIFO in the USB chip, whenever the USB chip has space. When both the *Frame Buffer* and the latter 8-bit PISO have been emptied of all data, the `usb_flush` signal is asserted, to tell the USB chip to send all remaining data to the CPU, as soon as possible, without waiting for enough data to precisely fill up the final USB block.

The internals of the Frame Delimiter

The implementation of the *Frame Delimiter* is shown in figure 3.11.

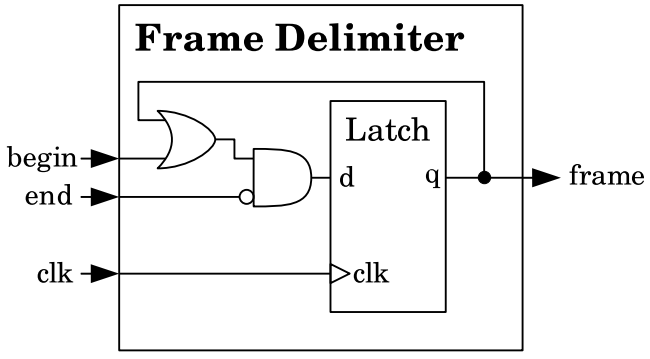

Figure 3.11: The Frame Delimiter

When the `end` input signal is asserted at the start of a clock cycle, the `frame` output becomes

de-asserted, regardless of the state of the `begin` input signal. This terminates the assembly of an output frame. The `frame` signal does not go high again, until the start of a clock cycle when the `begin` signal is newly asserted, after the `end` signal has been de-asserted.

This means that a new frame will not begin if the `begin` signal is asserted while the `end` signal is still asserted, and that if the `end` signal becomes asserted during a frame, the frame is considered to be complete, regardless of the state of the `begin` input.

The internals of the Slave Selector

The implementation of the *Slave Selector* is shown in figure 3.12.

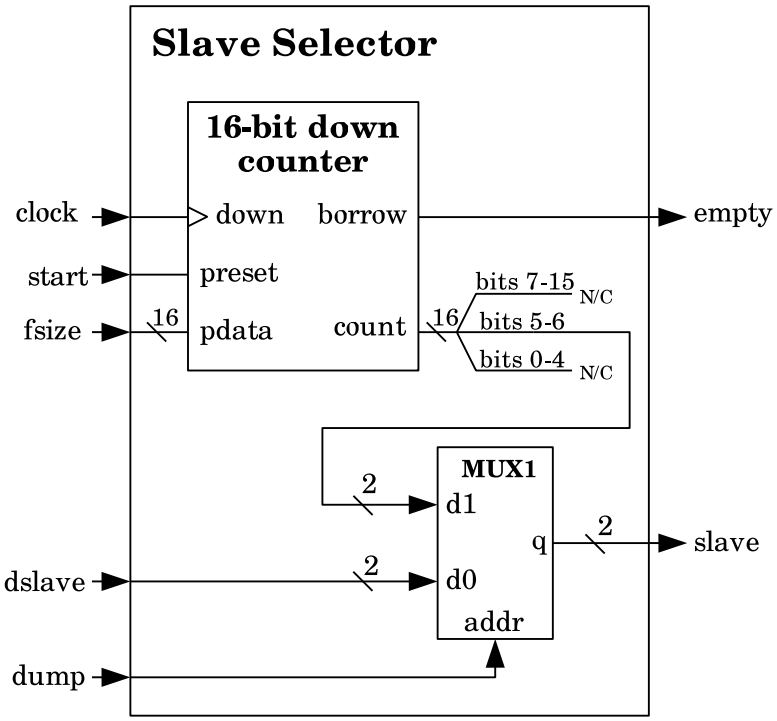

Figure 3.12: The Slave Selector

At the start of each clock cycle the *Slave Selector* tells the *Slave Reader* which slave to read the next 16-bit sample from, as well as indicating when readout should stop. In particular, in normal integration mode, it arranges that all data from slave 3 be read out first, followed by all data from slave 2, then all data from slave 1 and finally all data from slave 0. Alternatively, in dump mode, it indicates that data should only be read from the slave that is indicated by its `dslave` input. In both of these cases, once a total of `fsize` samples have been read-out, since the start of the frame, *Slave Selector* asserts its `empty` output signal, to indicate that readout should stop. In normal integration mode, the *State Generator* thus sets the `fsize`

input such that all of the slave PISOs are emptied before the *Slave Selector* asserts the `empty` signal, whereas in dump mode, it sets the `fsize` input according to the currently configured size of a dump-mode frame.

As evident in the diagram, the major functionality of the *Slave Selector* is implemented using a down-counter. This is preset to the value of the `fsize` input at the start of a new frame, when the `start` input is asserted. One clock-cycle later the output of the counter has settled to this value, and the *Data Dispatcher* reads the first sample from the specified slave, and the down-counter decrements by one, before the next sample is read on the next clock-cycle. The actual number of samples that get read before the asserted `borrow` output prevents further readout, is `fsize` + 1. Therefore the value of `fsize` actually represents one less than the number of samples that are to be read from the slaves.

Whereas in dump mode, only the `borrow` output of the counter is used by the *Slave Selector*, in normal integration mode, bits 5 and 6 of the output count (counting bits from 0), are also used to specify the slave that is to be read from. These are the two most significant bits of the count, given that in normal integration mode, the *State Generator* sets `fsize` to 127 (ie. 32PISO entries $\times$ 4slaves $-$ 1). Thus for the first 32 clock cycles, the two msb's of the count set the `slave` output to 3, then for the next 32 cock cycles they set `slave` to 2, then 1, then finally 0. In this way 32 samples are read from each slave, one slave at a time, starting with slave 3, and ending with slave 0.

Currently it isn't known how much space can be allocated for the FIFO in the *Frame Buffer*, so the maximum frame size is similarly unknown. As such, the `fsize` input has been arbitrarily assigned 16 bits, for now, which corresponds to a maximum size of a dump-mode frame of 65536 contiguous ADC samples (6.5ms). The actual size of the `fsize` input can be better tuned to the size of the FIFO obtained, when the design is complete, but in practice there is no harm in it being bigger than the available space in the FIFO, since the *Data Dispatcher* keeps an eye on the state of the *Frame Buffer* and terminates the frame early, as soon as the *Frame Buffer* becomes full.

The internals of the Slave Reader

As depicted in figure 3.13, the *Slave Reader* uses its `slave` input signal to connect the `data` and `read` signals of the correspondingly numbered slave FPGA, to the `so` input and `q` output signals of the *Slave Reader*. Thus a read-strobe on the `so` input of the *Slave Reader*, is routed through de-multiplexer, DMUX1, to the `read` output going to the selected slave, while the data that this returns from the selected slave is routed through the multiplexer, MUX1, to the `q` output of the *Slave Reader*.

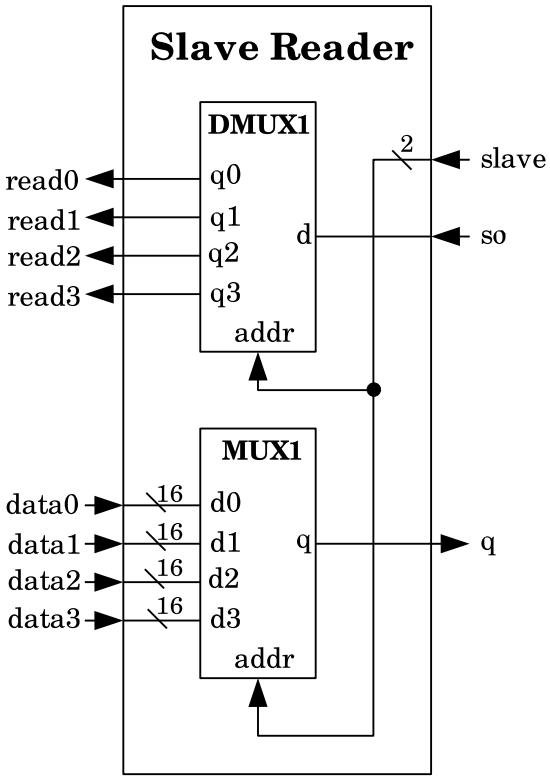

Figure 3.13: The Slave Reader

The internals of the Frame Buffer

As shown in figure 3.14, the *Frame Buffer* has two major parts, a 16-bit wide PISO containing a frame header, and a large 16-bit wide FIFO containing integrated or dump-mode data. The outputs of the *Frame Buffer* simulate the output of a virtual 16-bit wide FIFO, formed from the serialized concatenation of the contents of these two parts, with the header coming out first, followed by the data.

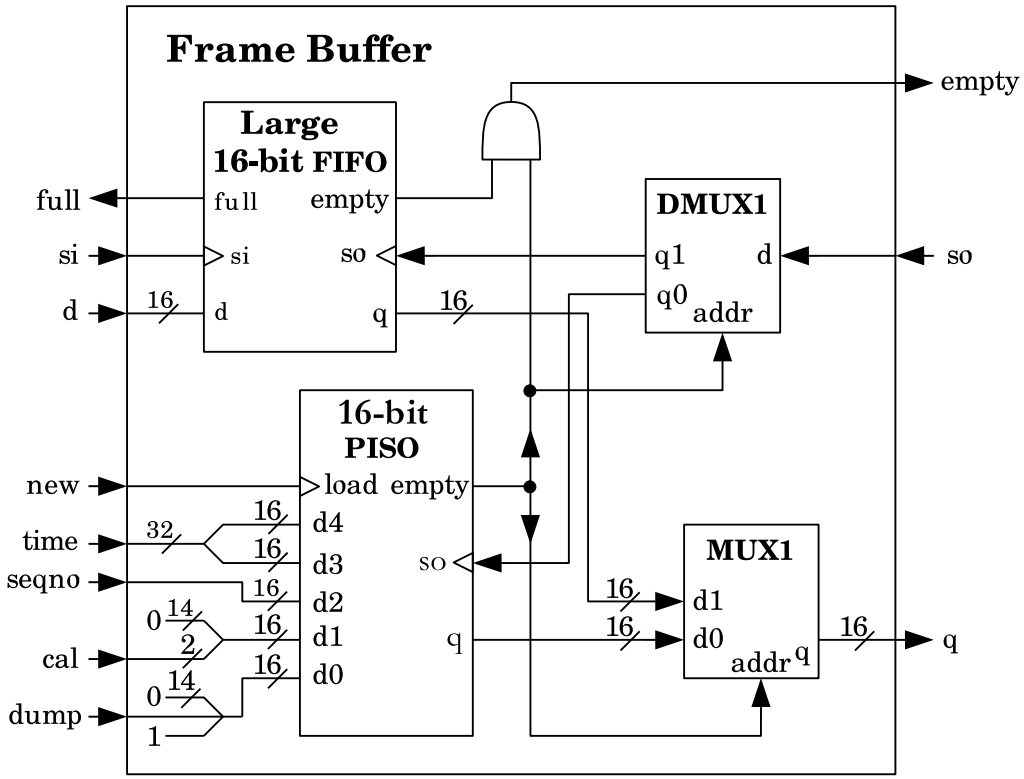

Figure 3.14: The Frame Buffer

The glue-logic with which the Data Dispatcher embeds the *Frame Buffer*, ensures that a new frame can not be started unless the *Frame Buffer* is empty. Thus whenever a new frame is successfully started, it is guaranteed that both the data FIFO and the header PISO of the *Frame Buffer* are empty. The start of a new frame is signaled by a rising edge on the `new` input, which is externally connected to the `frame` output of the *Frame Delimiter* in the Data Dispatcher.

At the start of a new frame, the rising edge of the `new` input-signal causes the header PISO to load the contents of the header, as derived from input signals received from the *State Generator*. The header currently consists of 5 16-bit words, which are used as follows.

- The first of the 16-bit header words 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.

  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 conditions that pertained while the data were being taken. Currently this consists of two bits, specifying the states of the cal-diode switches.

- The third of the header words is a 16-bit scan sequence number. This reflects the value of a counter in the *State Generator*, which is reset to zero, whenever the FPGA firmware is reset, and incremented by 1 whenever a parallel port command to start a scan or intra-scan is received. The CPU will use this both to watch for the first integration of a newly requested scan, and potentially to watch for missing scans.

- The 4th and 5th 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.

On the same clock edge during which the `new` signal goes high, data become available from the slaves. These data are synchronously clocked into the FIFO, 16-bits at a time, by the *Data Dispatcher*, using the `si`, shift-in, input and the `d`, data inputs. This continues until there are no data left to be read from the slaves, or the FIFO becomes full. In either case, the *Frame Delimiter* then disables further input to the *Frame Buffer*, until the next time that the *State Generator* asserts the `start` signal, after the contents of the *Frame Buffer* have all been sent to the real-time CPU.

3.3 The State Generator

still TBD