Atlantic Interface

June 2002, ver. 3.0 Functional Specification 13

Features

- Flexible interface for packet-oriented data of arbitrary length
- Interfaces all Altera® cell and packet MegaCore® functions
- Synchronous point-to-point connection
- High throughput with flexible flow control
- Master source/slave sink or master sink/slave source relationships
- Scalable clock frequency and data path width
- Fixed start of packet (SOP) alignment simplifies packet handling

Functional Description

The direction of data flow on the Atlantic™ interface can be either from master to slave (master source) or slave to master (slave source).

Figure 1 shows a block diagram of the Atlantic interface, transmitting from master source to slave sink.

Figure 1. Master Source to Slave Sink Block Diagram

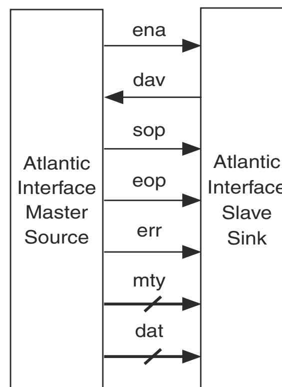


Figure 2 on page 2 shows a block diagram of the Atlantic interface, transmitting from slave source to master sink.

A-FS-13-3.0

Figure 2. Slave Source to Master Sink Block Diagram

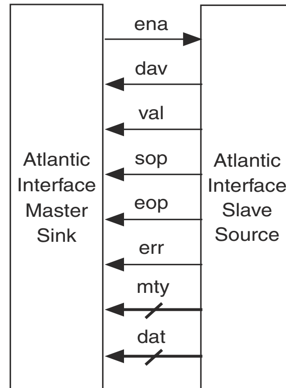
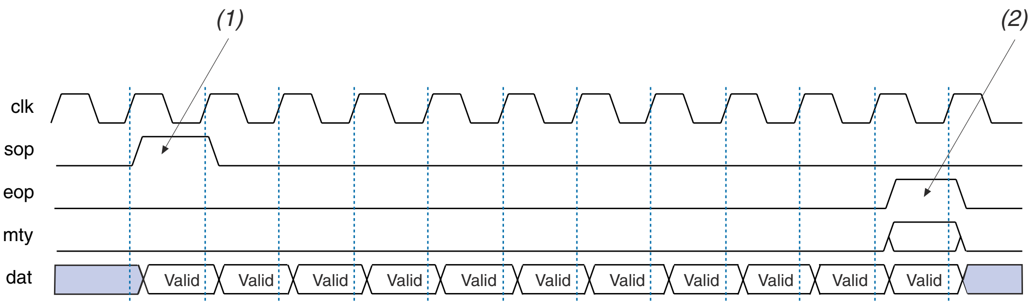


Figure 3 shows an example data packet in either master source or slave source configurations. In this case, `ena` and `val` are continuously asserted.

Figure 3. Example Data Packet



Notes:

(1) `sop` marks the start of the data packet.
(2) `eop` marks the end of the data packet, and `mty` indicates the number of invalid bytes.

Master Source

A slave sink interface responds to write commands from a master source interface.

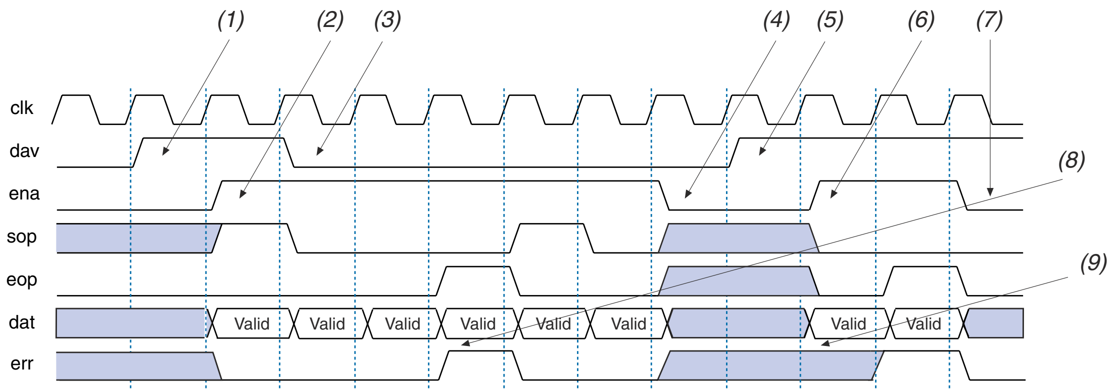
The master asserts ena and drives new data on dat. On the following rising edge of clk, the slave observes dat and ena. If ena is asserted, the slave accepts and processes data; if not, it discards dat. No val is required in this direction because ena indicates when dat contains new, valid data. If ena is deasserted, dat is undefined. The slave sink has no cycle-by-cycle flow control; it uses dav to request the master to stop data transfer. However, the master may take several clock cycles to stop transferring data, depending on the application.

dav indicates that the slave can accept a significant amount of data. The amount of data is application dependent. If the master continues to assert ena for an extended period of time after dav is deasserted, the slave may overflow.

For a master source, there is no delay after ena is asserted or deasserted and dataflow on dat (and associated data interface signals) starts or stops.

Figure 4 shows the timing of the Atlantic interface with a master source.

Figure 4. Atlantic Interface Timing—Master Source to Slave Sink



Notes:

(1) Slave sink indicates it has space for threshold words.
(2) The master source begins writing data to the slave sink.
(3) Slave sink indicates it no longer has space for threshold words. Master source can continue to send data, but must ensure that the slave sink does not overflow.
(4) Master source stops sending data.
(5) Slave sink indicates it has space for threshold words.
(6) Master source begins writing data to the slave sink.
(7) Slave sink indicates it still has space, but the master source has run out of data.
(8) Master source detects an error, and asserts err.
(9) Master source identifies a data error, it asserts err and holds it until eop is deasserted.

Slave Source

A master sink interface generates read commands to a slave source interface.

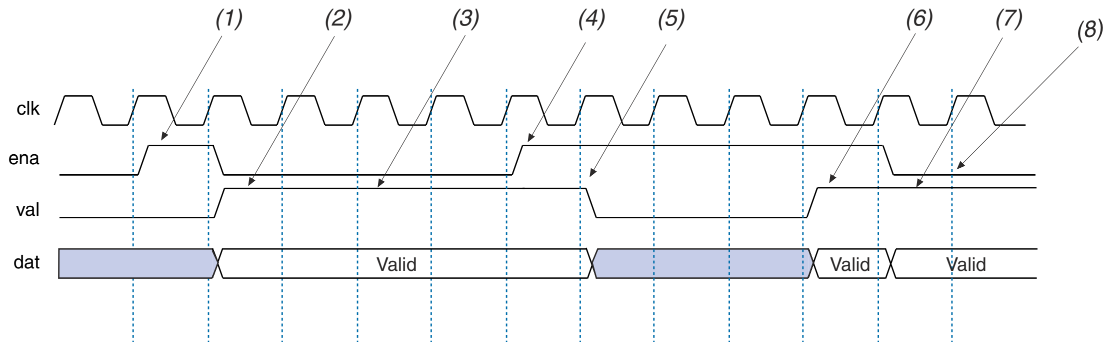
The master asserts `ena` when it is ready for data. On the following `clk` rising edge, the slave observes `ena` asserted. The slave immediately either drives new data on `dat` and asserts `val`, or deasserts `val`. On the following `clk` rising edge, the master observes the state of `val` and samples `dat`. If `val` is asserted, the master uses the value from `dat`; if not, `dat` is undefined and the master discards the contents of `dat`.

If `ena` is asserted on a previous clock edge and `val` is asserted on the current clock edge, then `dat`, `sop`, `eop`, and `err` are also valid on the current clock edge, and `dat` contains new data. If `val` is deasserted, `sop`, `eop`, `err`, and `dat` are undefined.

`dav` indicates that the slave can supply a significant amount of data. The amount of data is application dependent. If the master continues to assert `ena` for an extended period of time after `dav` is deasserted, the slave may underflow.

Figure 5 shows the general slave source flow control.

Figure 5. Slave Source Flow Control

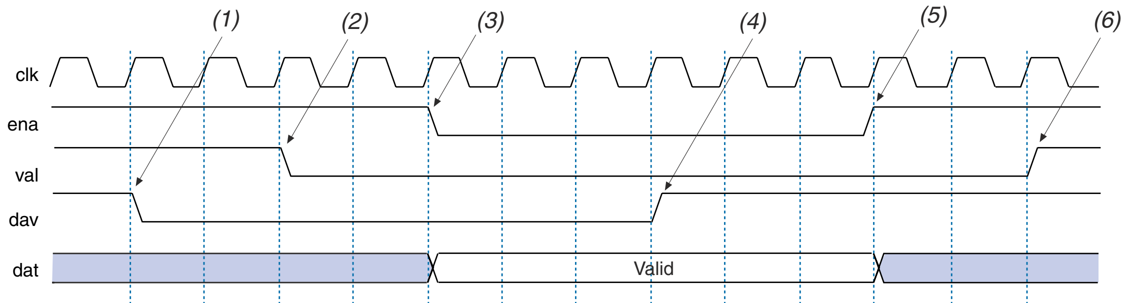


Notes:

(1) Master asserts `ena` to request new data.
(2) Slave drives `dat` and asserts `val` in response to the request.
(3) Slave holds `dat` and `val` because the master is not requesting additional data. Since `dat` is being held constant, the master can sample the data at any point.
(4) Master reasserts `ena` to request new data.
(5) Slave has no data and deasserts `val`.
(6) Slave has new data. Slave drives `dat` and asserts `val`.
(7) Slave has new data. Slave drives `dat` and asserts `val`.
(8) Slave holds `val`, `sop`, `eop`, `err`, and `dat` constant.

Figure 6 shows the slave source with underflow handling.

Figure 6. Slave Source Underflow Handling

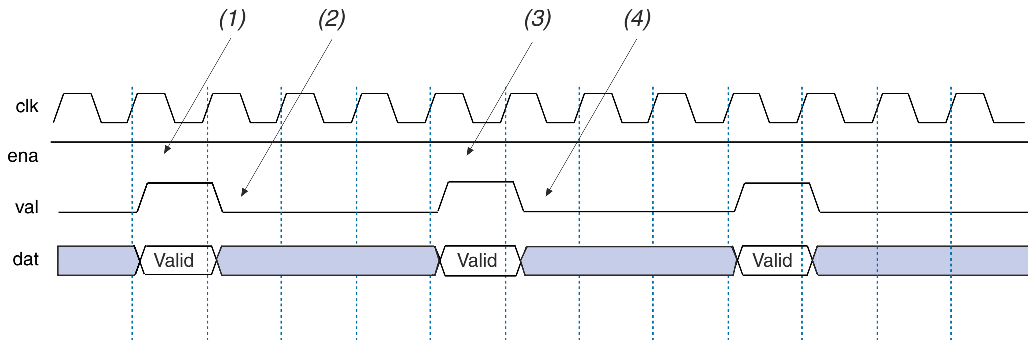


Notes:

(1) Slave deasserts `dav` to indicate potential underflow.
(2) Slave underflows and deasserts `val`. `dat`, `sop`, `eop`, and `err` become undefined.
(3) Master deasserts `ena` to stop data flow. `val` holds its value. `dat`, `sop`, `eop`, and `err` are undefined.
(4) Slave asserts `dav` to indicate it has data available. Because `ena` is deasserted, `val` stays deasserted and `dat`, `sop`, `eop`, and `err` are undefined.
(5) Master asserts `ena` to restart data flow.
(6) Slave takes one cycle to fetch data then asserts val, new data on `dat`, `sop`, `eop`, and `err`.

The master may assert `ena` continuously. The slave can then issue data on `dat` at will, using `val` to indicate if `dat` contains new data. Figure 7 shows the slave source with slave flow control.

Figure 7. Slave Source With Slave Flow Control

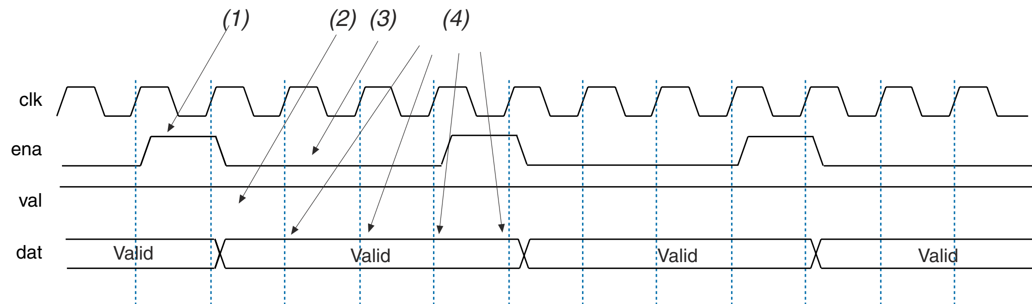


Notes:

(1) Slave has new data. Slave drives `dat` and asserts `val`.
(2) Slave has no new data; `dat` becomes undefined. Slave deasserts `val`.
(3) Slave has new data. Slave drives `dat` and asserts `val`.
(4) Slave has no new data; `dat` becomes undefined. Slave deasserts `val`.

In general, the master only accepts new data if it observes val asserted on the current rising edge of clk and ena was asserted on the previous rising edge of clk. However, if ena is deasserted, the slave holds dat, sop, eop, err, and val at the same values. The master may then sample val and dat at any time until the clk edge following the reassertion of ena. Figure 8 shows the slave source with master flow control.

Figure 8. Slave Source With Master Flow Control



Notes:

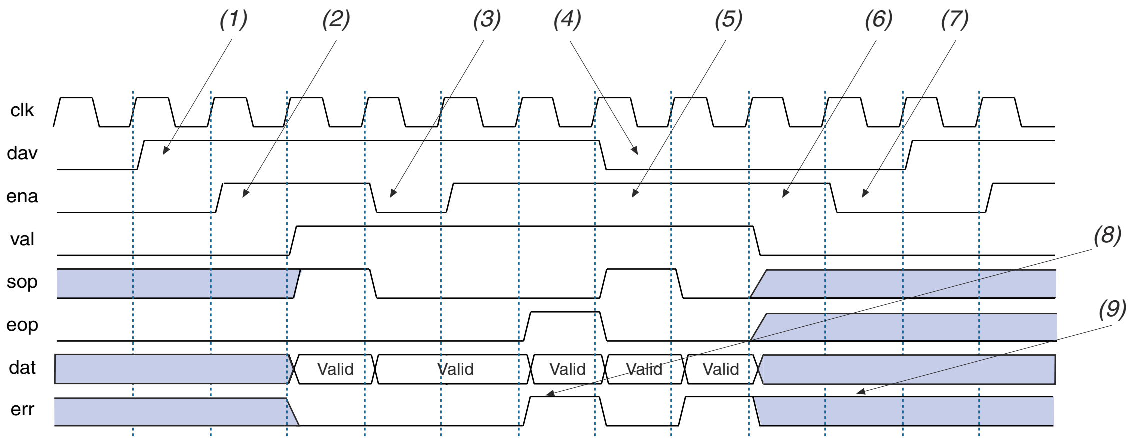
(1) Master requests data by asserting ena.
(2) Slave drives new data on dat.
(3) Slave observes ena deasserted and holds dat and val at their current value.
(4) Master may sample dat at any of these edges.

For a slave source to master sink there is a one-cycle delay after ena is asserted or deasserted and dataflow on dat (and associated data interface signals) starts or stops. When the master asserts ena, no new data is available on dat until after the following rising edge of clk. Similarly, when the master deasserts ena, dat and val will have new values on the following rising edge of clk and hold their values until ena is asserted. However, the interface is pipelined, so the delay does not affect the net throughput of the interface.

dav indicates that the slave has a significant amount of data available. However, the master may assert ena if dav is negated. If the slave does not have data immediately available when ena is asserted, it deasserts val.

Figure 9 shows the timing of the Atlantic interface with a master sink.

Figure 9. Atlantic Interface Timing—Slave Source to Master Sink



Notes:

(1) Slave source indicates that data is available (either `threshold` words available, or EOP).
(2) Master sink begins reading data.
(3) Master sink decides to stop reading the data for one clock cycle. `val` remains asserted and `data`, `sop`, and `eop` hold their current values.
(4) Slave source indicates that it has less than `threshold` words available. The master sink can continue to read data until it detects `val` deasserted.
(5) Master sink continues to read data, validates data with `val`.
(6) Slave source cannot supply any more data, so deasserts `val`.
(7) Master sink goes idle until `dav` is reasserted.
(8) Slave source identifies an `eop` error, and asserts `err`.
(9) Slave source identifies a data error, and asserts `err` until `eop` is deasserted.

Compatibility

To ensure that individual implementations of an Atlantic interface are compatible they must have:

- The same data bus width
- The same parity, or none
- The same address signal, or none
- Compatible data directions (data source connecting to data sink)
- Compatible control interfaces (master interface connecting to slave interface)
- Compatible threshold levels (slave sink can overflow, and slave source can operate inefficiently if thresholds are incorrectly set)

Signals

To support multiple Atlantic interfaces on the same module, the user should differentiate each interface by prefixing the signal names: <*prefix*>_<*signal name*>.

Table 1 describes the Atlantic interface global signals.

Table 1. Global Signals

Name	Direction	Description
`clk`	Input	Clock, rising edge active. The Atlantic interface uses single-edge clocking. All signals are synchronous to `clk`, and master and slave are in the same clock domain.

Table 2 describes the Atlantic interface control signals.

Table 2. Control Signals (Part 1 of 2)

Name	Direction	Description
`ena`	Master to slave	Data transfer enable signal. `ena` is driven by the interface master and used to control the flow of data across the interface. If the master is the source, `ena` behaves as a write enable from master to slave. The master asserts `ena` and `dat` simultaneously. When the slave observes `ena` asserted on the `clk` rising edge it immediately captures the Atlantic data interface signals. If the slave is the source, `ena` behaves as a read enable from master to slave. When the slave observes `ena` asserted on the `clk` rising edge it drives the Atlantic data interface signals and asserts `val`. The master captures the data interface signals on the following `clk` rising edge. If the slave is unable to provide new data, it deasserts `val` for one or more clock cycles until it is prepared to drive valid data interface signals.
`val`	Slave to master	Data valid signal. Present only on a slave source and master sink interface. `val` indicates the validity of the data signals. `val` is updated on every clock edge where `ena` is sampled asserted, and holds its current value along with the `dat` bus where `ena` is sampled de-asserted. When `val` is asserted, the Atlantic data interface signals are valid. When `val` is deasserted, the Atlantic data interface signals are invalid and must be disregarded. To determine whether new data has been received, the master must qualify the `val` signal with the previous state of the `ena` signal.

Table 2. Control Signals (Part 2 of 2)

Name	Direction	Description
`dav`	Slave to master	Data available signal. When the `dat` bus is in slave to master direction, if `dav` is high, the slave has at least `threshold` words available to be read, or the data can be read up to an end of packet without risk of underflow. When the `dat` bus is in master to slave direction, if `dav` is high, the slave has enough space for `threshold` words to be written. *(1)*

Note:
(1) *`threshold`* is implementation dependent, and typically corresponds to FIFO buffer almost full/empty levels.

Table 3 describes the Atlantic interface data signals.

Table 3. Data Signals (Part 1 of 2)

Name	Direction	Description
`sop`	Source to sink	Start of packet signal. `sop` is used to delineate the packet boundaries on the `dat` bus. When `sop` is asserted, the start of the packet is present on the `dat` bus and aligned to the most significant byte. `sop` is asserted on the first transfer of every packet.
`eop`	Source to sink	End of packet signal. `eop` is used to delineate the packet boundaries on the `dat` bus. When `eop` is asserted, the end of the packet is present on the `dat` bus. `mty` indicates the number of invalid bytes the last word is composed of when `eop` is asserted. `eop` is asserted on the last transfer of every packet.
`err`	Source to sink	Error indicator signal. `err` is used to indicate that the current packet is aborted and should be discarded. `err` can be asserted at any time during the current packet, but when asserted it can only be deasserted on the clock cycle after `eop` is asserted. Conditions that can cause `err` to be set can be, but are not limited to, header error correction (HEC) or frame check sum (FCS) error, FIFO buffer overflow, parity error, or abort sequence detection.

Table 3. Data Signals (Part 2 of 2)		
Name	**Direction**	**Description**
mty[m-1:0] (m=0, No mty) (m=1,2,3,...)	Source to sink	Word empty bytes. mty indicates the number of invalid (empty) bytes of data in dat. The mty bus is all zero, except during the last transfer of a packet on dat. When eop is asserted, the number of invalid packet data bytes on dat is specified by mty. The definition of mty is compatible with the mod signal in the POS-PHY level 2 and 3 specifications. This table is an example of m=3 or mty[2:0]. m values are dependent on the dat field values. mty = ‘000’, All dat bytes are valid mty = ‘001’, dat[7:0] are invalid mty = ‘010’, dat[15:0] are invalid mty = ‘011’, dat[23:0] are invalid mty = ‘100’, dat[31:0] are invalid mty = ‘101’, dat[39:0] are invalid mty = ‘110’, dat[47:0] are invalid mty = ‘111’, dat[55:0] are invalid An 8-bit dat bus does not have an mty signal A 16-bit dat bus requires an mty[0] signal (m=1) A 32-bit dat bus requires an mty[1:0] signal(m=2) A 64-bit dat bus requires an mty[2:0] signal (m=3) An 8×2^m dat bus requires an mty[m-1:0] signal mty can only be non-zero when eop is asserted.
par	Source to sink	Parity signal (optional). The par signal indicates the parity calculated over the dat bus. Odd and even parity are supported.
adr[n:0] (n=0,1,2,3,...)	Source to sink	Address bus (optional). The adr bus is a defined extension to the dat bus. It carries the associated address information for each packet for use in multi-port implementations. The adr bus must be valid at the same time as the dat bus and remain constant throughout a complete packet.
dat[8×2^m-1:0] (m=0,1,2,3...)	Source to sink	Data bus. This bus carries the packet octets that are transferred across the interface. The data is transmitted in big-endian order on dat. m determines the bus size, for example: if m=2, dat is [31:0]; if m=3, dat is [63:0].

Note:

(1) The direction is reversed for source interfaces.

Timing

The clock frequency is application dependent.