Lattice Memory Mapped Interface and Lattice Interrupt Interface

User Guide

FPGA-UG-02039-1.2

January 2020

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS and with all faults, and all risk associated with such information is entirely with Buyer. Buyer shall not rely on any data and performance specifications or parameters provided herein. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. No Lattice products should be used in conjunction with mission- or safety-critical or any other application in which the failure of Lattice's product could create a situation where personal injury, death, severe property or environmental damage may occur. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Contents

Figures

Tables

1. Introduction

This document provides a description of the Lattice Memory Mapped Interface (LMMI) and Lattice Interrupt Interface (LINTR). Many FPGA IP blocks provide a set of dynamically programmable configuration, control, and status bits or provide interfaces for transactional data transfer. LMMI is a common interface which supports these operations in a consistent and well-defined manner. Some FPGA IP blocks also provide asynchronous status information or service requests through interrupts. LINTR defines a set of functional standards and naming conventions for IP blocks which generate interrupts.

This document covers the top-level definitions of LMMI and LINTR which apply to all Lattice FPGA IP blocks. For more detailed interface or register information for a given FPGA IP block, please refer to the User's Guide for that block.

2. Lattice Memory Mapped Interface (LMMI)

LMMI is a simple memory-mapped address/data interface. It defines a standard set of interface signals for register/memory access and supports both single and burst transactions with a maximum throughput of one transaction per clock cycle. LMMI supports optional wait states for slave interfaces that need more than one clock cycle to complete a transaction, and supports multiple outstanding transactions (also known as pipelined transactions). Bus transactions are completed in order; LMMI does not support out of order transactions.

Although the LMMI protocol is capable of supporting all of the listed features, a given implementation may choose to support all, some, or none of these features. For example, a simple LMMI master may choose to implement only single transactions and not support burst transactions or pipelined transactions. To see which features a given FPGA IP block supports (e.g., wait states, pipelined transactions), please refer to the User's Guide for that block.

2.1. Signal Definitions

Table 2.1 below defines the LMMI signals. FPGA IP blocks may have additional ports, as appropriate to their functionality, but all register/memory access is handled through the LMM interface.

Table 2.1. LMMI Signal Definitions

Signal	Slave Direction	Description
lmmi_clk	In	Clock
lmmi_resetn	In	Reset (active low) Resets the LMM interface and sets registers to their default values. Does not reset the internal (i.e., non-user-accessible) registers of the IP block.
lmmi_request	In	Start transaction
lmmi_wr_rdn	In	Write = HIGH, Read = LOW
lmmi_offset[n:0]	In	Offset (0-32 bits) – register offset within the slave, starting at offset 0. Bit width is IP dependent.
lmmi_wdata[n:0]	In	Write data (0-32 bits) Bit width is IP dependent.
lmmi_rdata[n:0]	Out	Read data (0-32 bits) Bit width is IP dependent.
lmmi_rdata_valid	Out	Read transaction is complete and lmmi_rdata[] contains valid data.
lmmi_ready	Out	Slave is ready to start a new transaction. Slave can insert wait states by holding this signal low.

2.2. Reset

The lmmi_resetn signal can be asserted asynchronously, but deassertion must be synchronous after the rising edge of lmmi_clk. The minimum duration for reset assertion is one full clock cycle.

Reset is asynchronous. When lmmi_resetn is asserted, all output ports on the slave drive their reset value (0) immediately.

2.2. Transaction Descriptions

All LMMI signals are sampled on the rising edge of lmmi_clk. The timing of signal transitions shown in the waveforms in this document are intended as examples only. The only constraints on the timing of signal transitions are the setup and hold requirements around rising clock edges.

2.2.1. Write Transactions

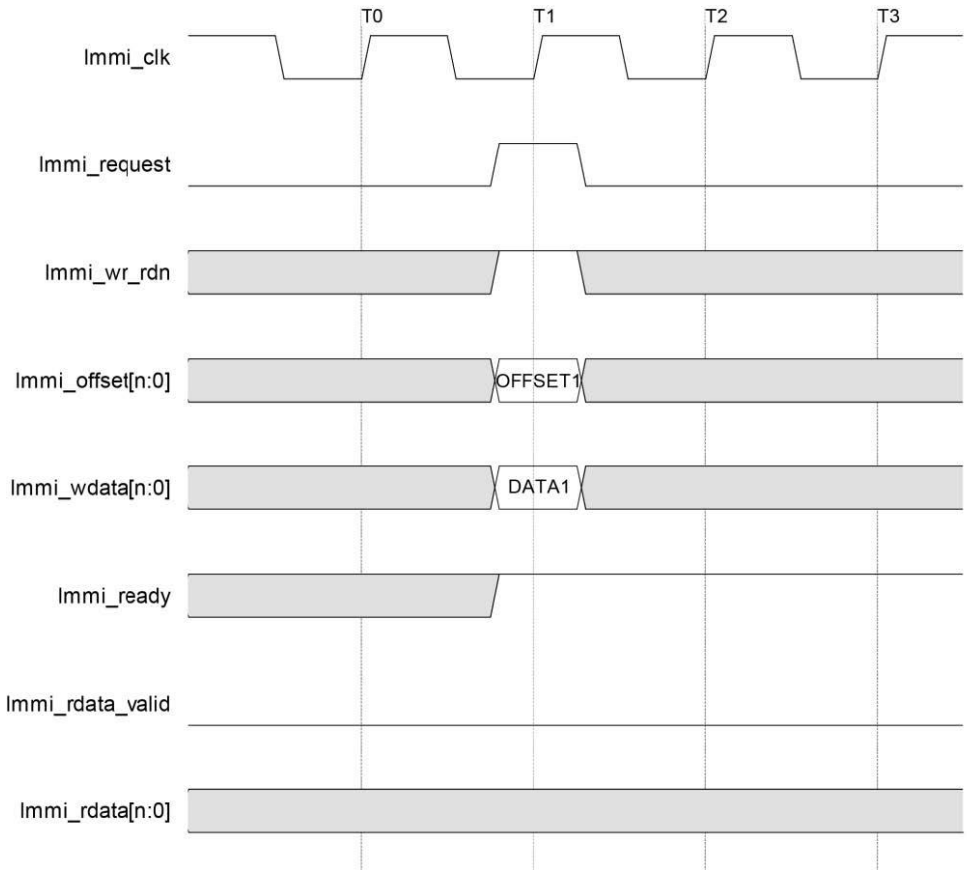

Figure 2.1. Single Write with No Wait States

Protocol Description (Figure 2.1)

T0: Master decides to start a write transaction, asserts lmmi_request and lmmi_wr_rdn, and drives lmmi_offset[] and lmmi_wdata[] with values for the new transaction. Slave is ready to start a new a transaction in the next clock cycle and asserts lmmi_ready.

T1: Master sees lmmi_ready high which signals that the slave has accepted the write transaction. After the appropriate hold time, Master deasserts lmmi_request and may change lmmi_wr_rdn, lmmi_offset[] and lmmi_wdata[].

T2: Slave signals that it is ready to start a new transaction by asserting lmmi_ready.

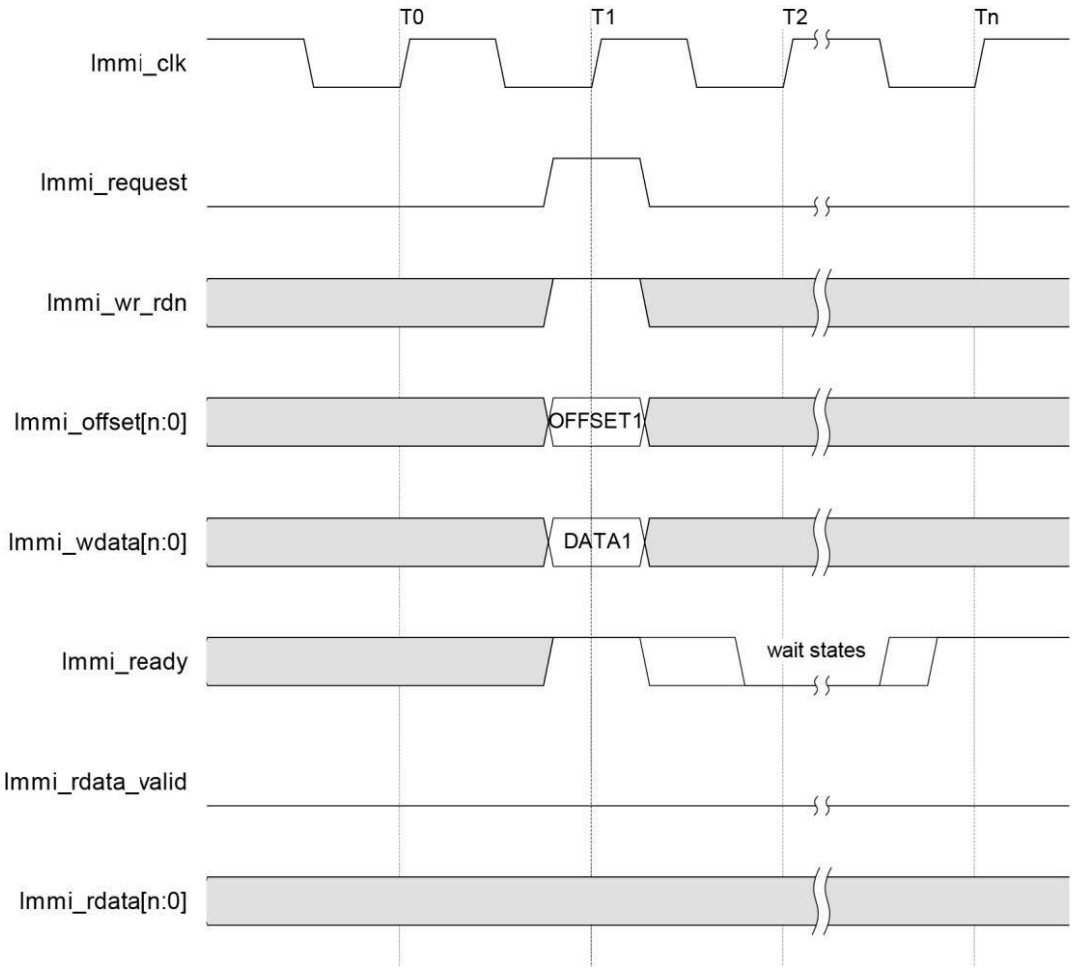

Figure 2.2. Single Write with Wait States

Protocol Description (Figure 2.2)

T0, T1: Same as single write with no wait states.

T2: Slave deasserts lmmi_ready to insert one or more wait states.

Tn: Slave signals that it is ready to start a new transaction by asserting lmmi_ready.

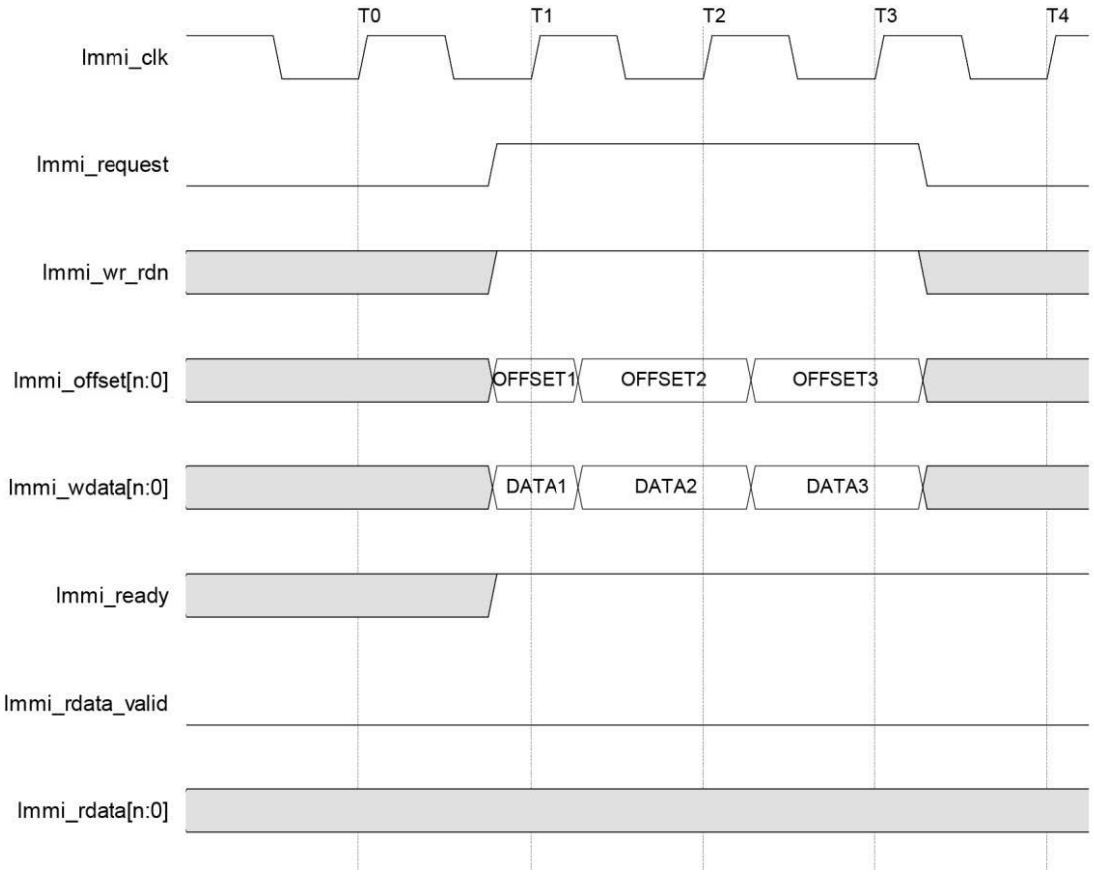

Figure 2.3. Burst Write with No Wait States

Protocol Description (Figure 2.3)

T0: Same as single write with no wait states.

T1, T2: Master sees lmmi_ready high which signals that Slave has accepted the write transaction. After the appropriate hold time, Master changes lmmi_offset[] and lmmi_wdata[] to new values for the next transaction.

T3: Master sees lmmi_ready high which signals that Slave has accepted the write transaction. After the appropriate hold time, Master deasserts lmmi_request and may change lmmi_wr_rdn, lmmi_offset[] and lmmi_wdata[].

T4: Slave signals that it is ready to start a new transaction by asserting lmmi_ready.

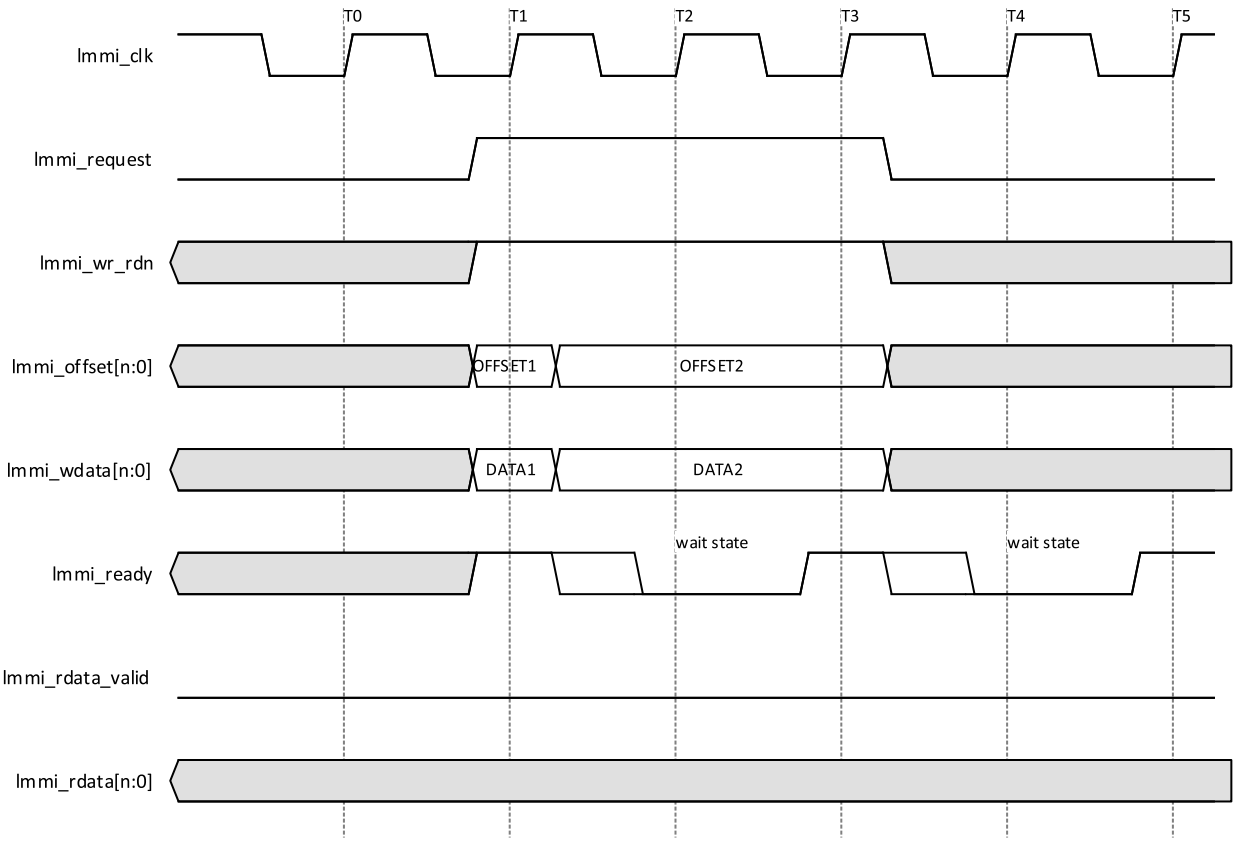

Figure 2.4. Burst Write with Wait States

Protocol Description (Figure 2.4)

T0, T1: Same as burst write with no wait states.

T2: Slave deasserts lmmi_ready to insert a wait state.

T3: Slave signals that it is ready to start a new transaction by asserting lmmi_ready. Master sees lmmi_ready high which signals that Slave has accepted the write transaction. After the appropriate hold time, Master deasserts lmmi_request and may change lmmi_wr_rdn, lmmi_offset[] and lmmi_wdata[].

T4: Slave deasserts lmmi_ready to insert a wait state.

T5: Slave signals that it is ready to start a new transaction by asserting lmmi_ready.

2.2.2. Read Transactions

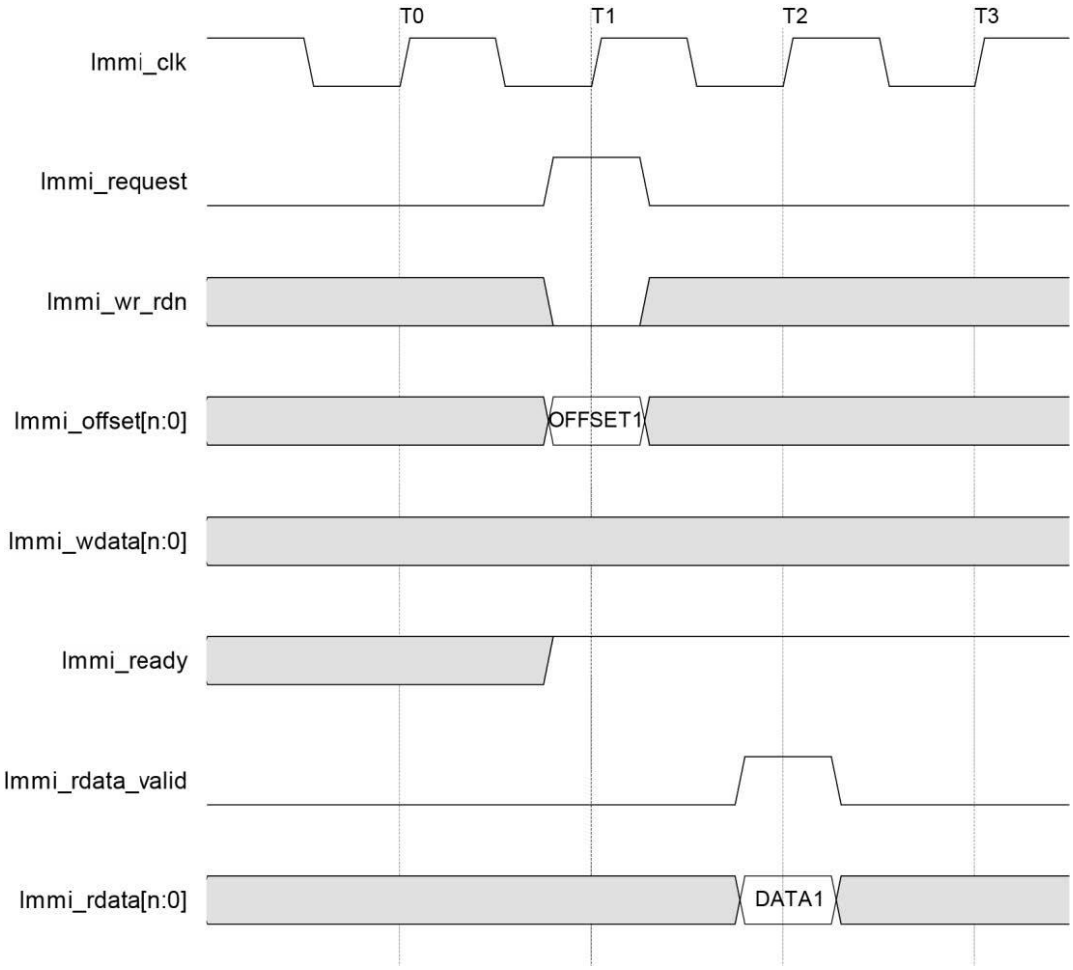

Figure 2.5. Single Read with No Wait States

Protocol Description (Figure 2.5)

T0: Master decides to start a read transaction, asserts lmmi_request, deasserts lmmi_wr_rdn, and drives lmmi_offset[] with a value for the new transaction. Slave is ready to start a new a transaction in the next clock cycle and asserts lmmi_ready.

T1: Master sees lmmi_ready high which signals that Slave has accepted the read transaction. After the appropriate hold time, Master deasserts lmmi_request and may change lmmi_wr_rdn and lmmi_offset[].

T2: Slave drives lmmi_rdata[] with the result of the read transaction, asserts lmmi_rdata_valid to signal that lmmi_rdata[] is valid, and asserts lmmi_ready to signal that Slave is ready to start a new transaction.

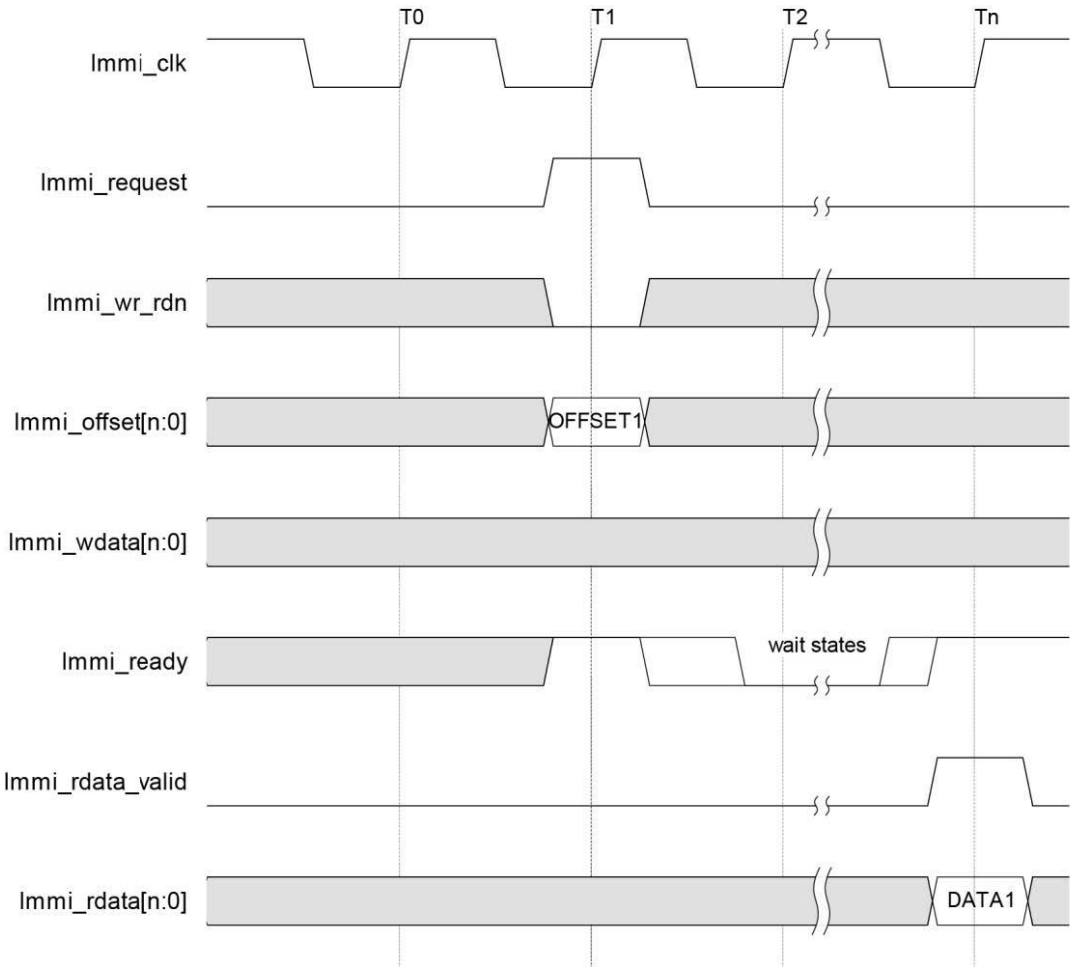

Figure 2.6. Single Read with Wait States

Protocol Description (Figure 2.6)

T0, T1: Same as single read with no wait states.

T2: Slave deasserts lmmi_ready to insert one or more wait states.

Tn: Slave drives lmmi_rdata[] with the result of the read transaction, asserts lmmi_rdata_valid to signal that lmmi_rdata[] is valid, and asserts lmmi_ready to signal that Slave is ready to start a new transaction.

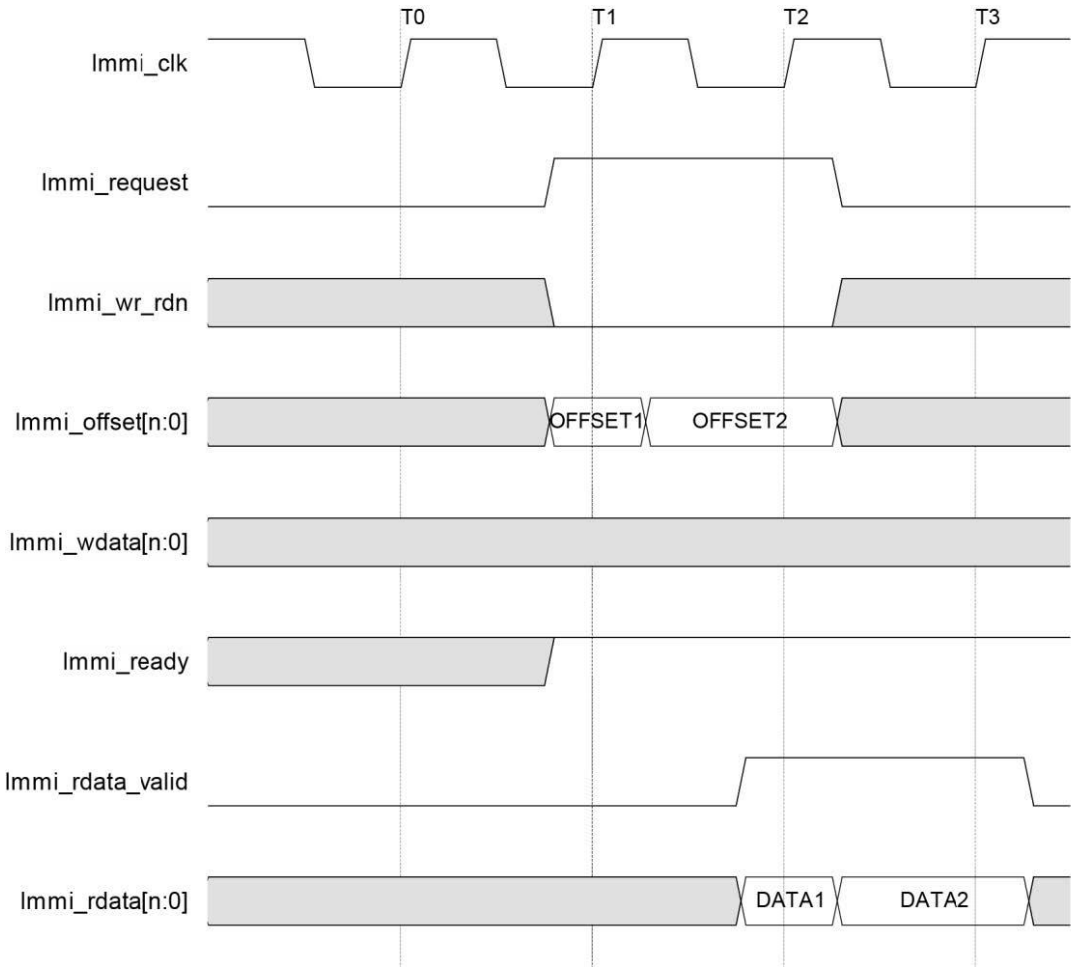

Figure 2.7. Burst Read with No Wait States

Protocol Description (Figure 2.7)

T0: Same as single read with no wait states.

T1: Master sees lmmi_ready high which signals that Slave has accepted the read transaction. After the appropriate hold time, Master changes lmmi_offset[] to a new value for the next read transaction.

T2: Slave drives lmmi_rdata[] with the result of the read transaction, asserts lmmi_rdata_valid to signal that lmmi_rdata[] is valid, and asserts lmmi_ready to signal that Slave is ready to start a new transaction. Master latches lmmi_rdata[]. After the appropriate hold time, Master deasserts lmmi_request and may change lmmi_wr_rdn and lmmi_offset[].

T3: Slave drives lmmi_rdata[] with the result of the read transaction, asserts lmmi_rdata_valid to signal that lmmi_rdata[] is valid, and asserts lmmi_ready to signal that Slave is ready to start a new transaction.

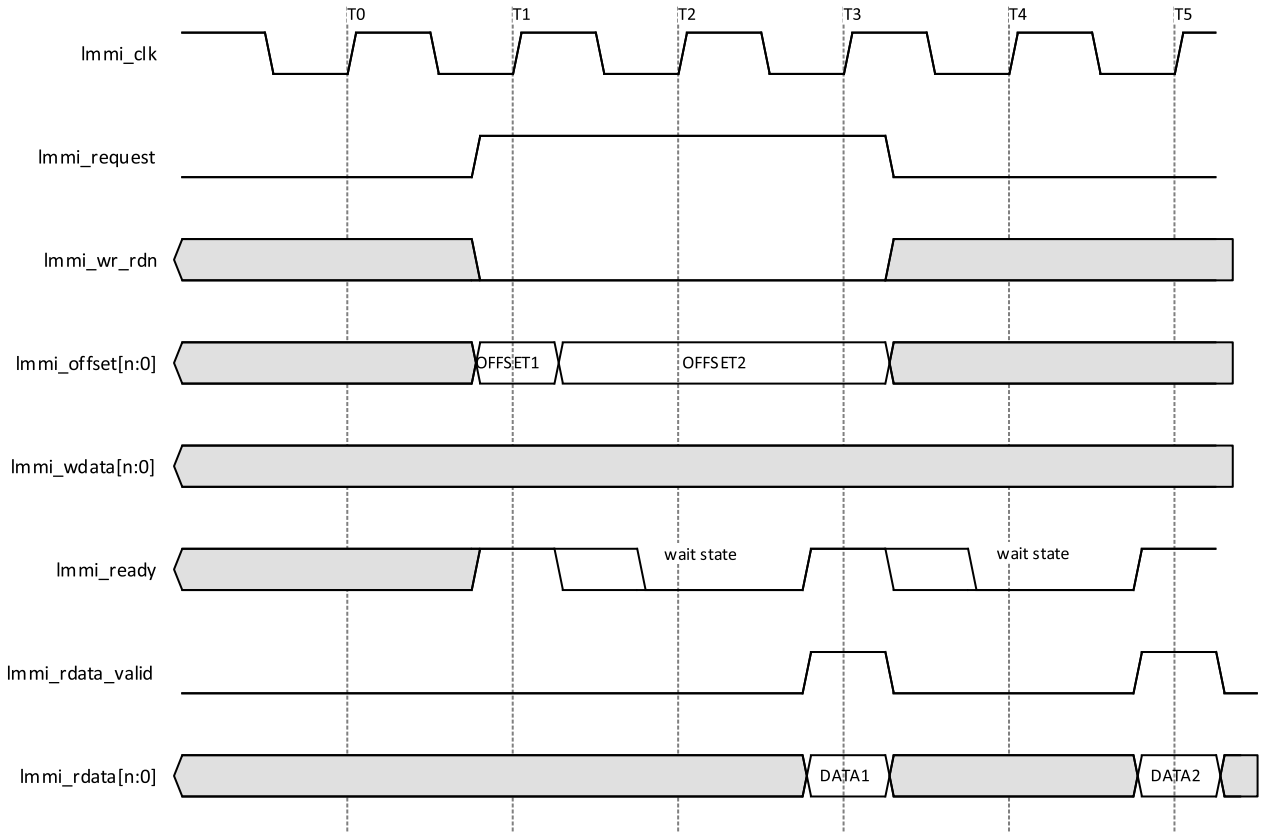

Figure 2.8. Burst Read with Wait States

Protocol Description (Figure 2.8)

T0, T1: Same as burst read with no wait states.

T2: Slave deasserts lmmi_ready to insert a wait state.

T3: Slave drives lmmi_rdata[] with the result of the read transaction, asserts lmmi_rdata_valid to signal that lmmi_rdata[] is valid, and asserts lmmi_ready to signal that Slave is ready to start a new transaction. Master latches lmmi_rdata[]. After the appropriate hold time, Master deasserts lmmi_request and may change lmmi_wr_rdn and lmmi_offset[].

T4: Slave deasserts lmmi_ready to insert a wait state.

T3: Slave drives lmmi_rdata[] with the result of the read transaction, asserts lmmi_rdata_valid to signal that lmmi_rdata[] is valid, and asserts lmmi_ready to signal that Slave is ready to start a new transaction.

2.2.3. Back-to-Back Transactions

This section shows examples of back-to-back read and write transactions. These examples are purely informative; there are no special protocol rules for back-to-back transactions.

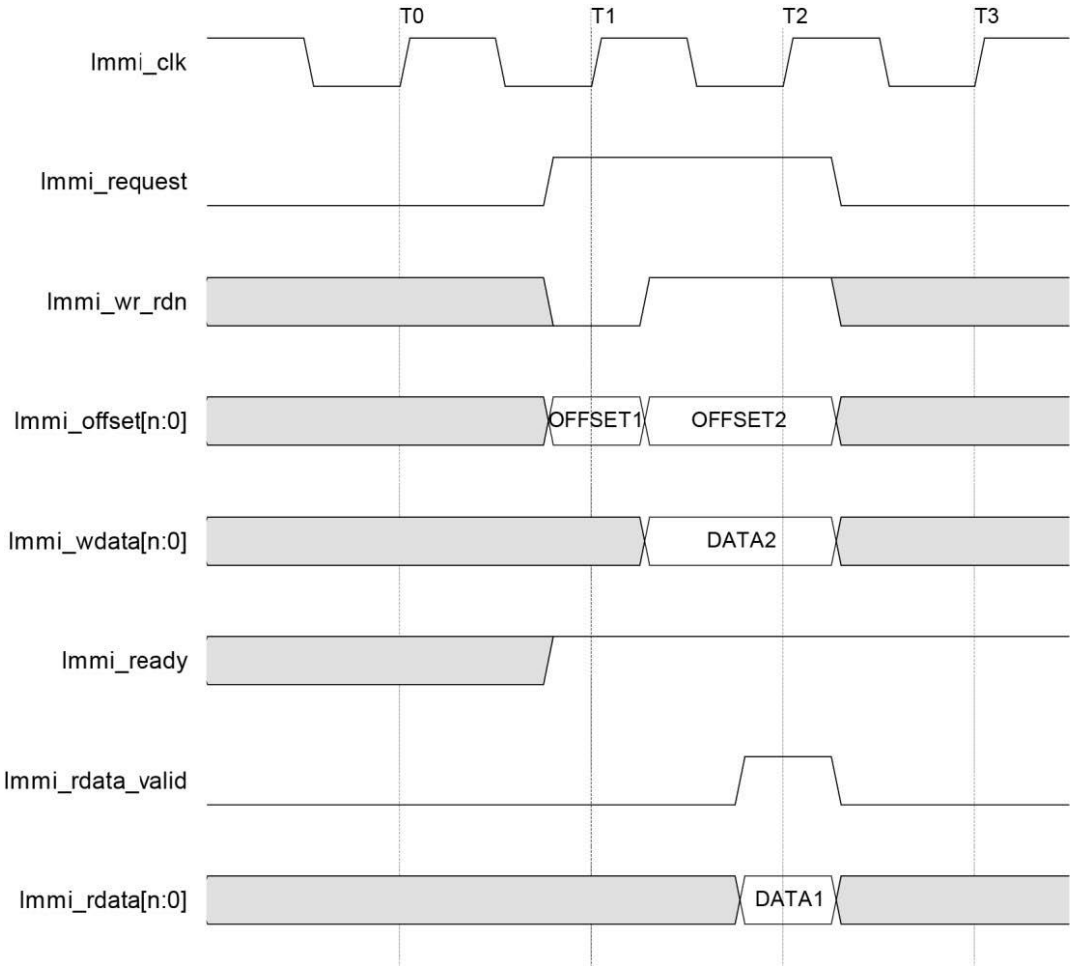

Figure 2.9. Back-to-Back Read and Write with No Wait States

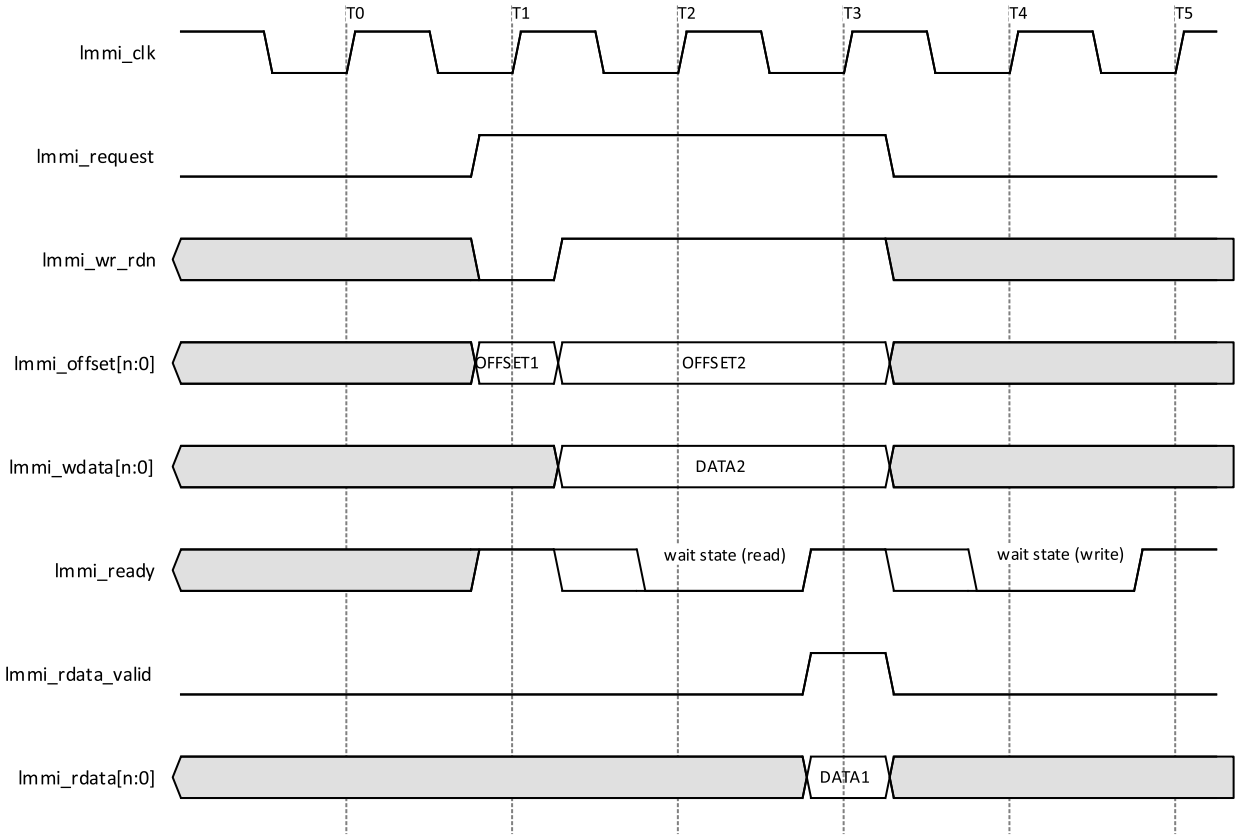

Figure 2.10. Back-to-Back Read and Write with Wait States

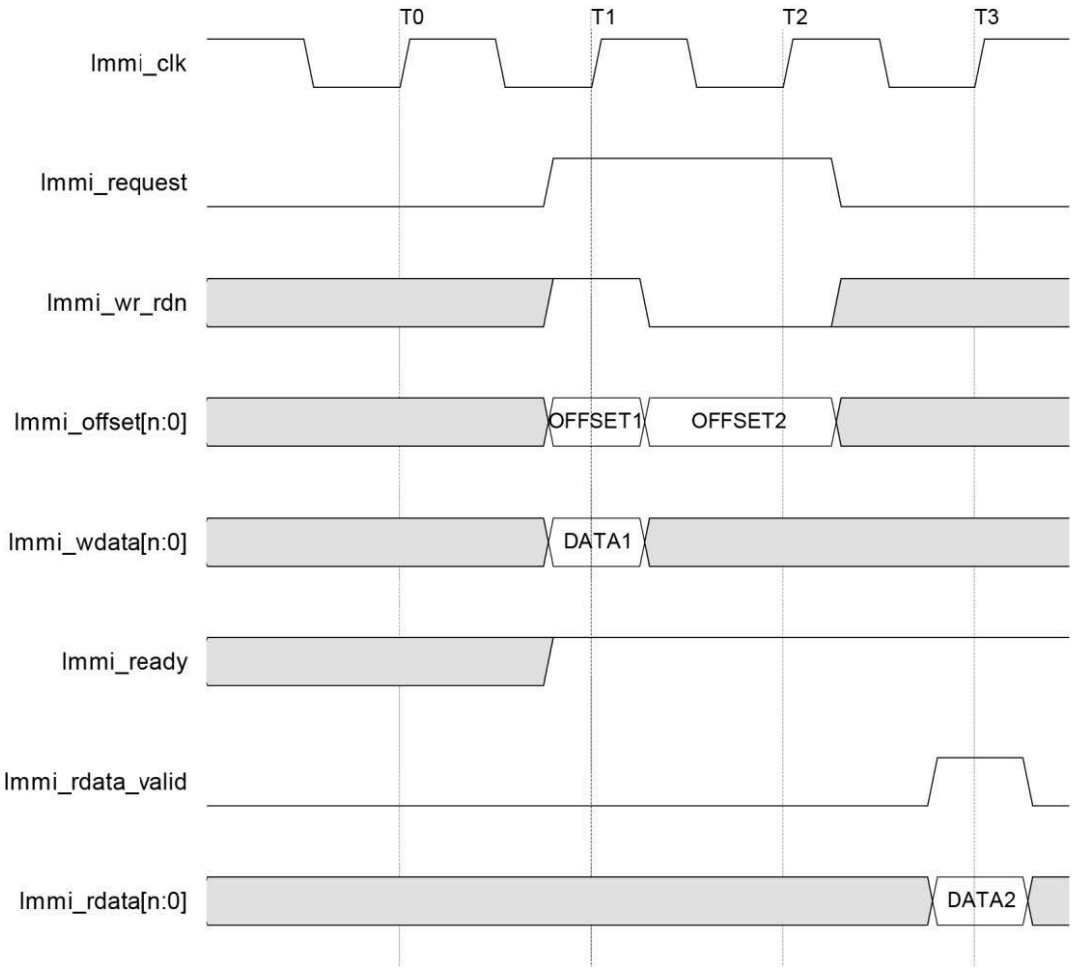

Figure 2.11. Back-to-Back Write and Read with No Wait States

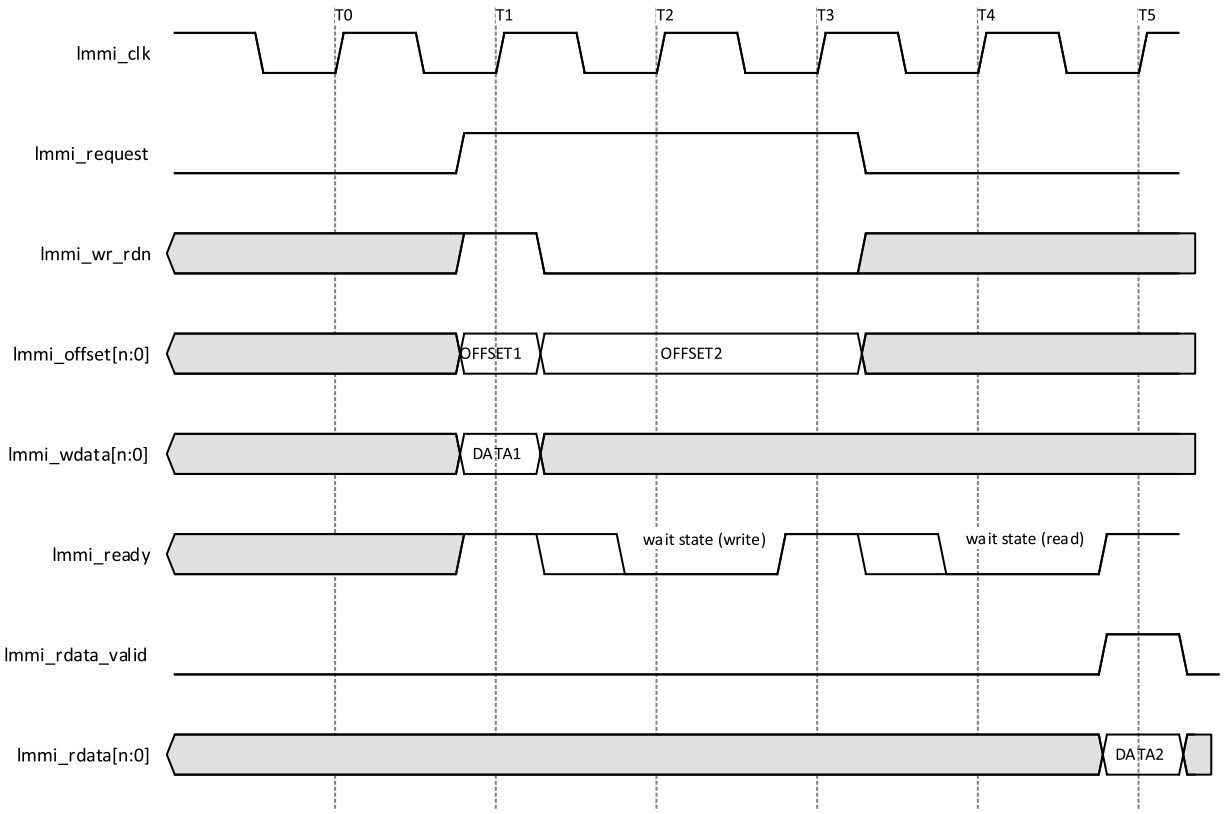

Figure 2.12. Back-to-Back Write and Read with Wait States

2.2.4. Pipelined Transactions

The LMMI protocol supports multiple outstanding transactions with in-order completion (also known as pipelined transactions). This feature is optional in both LMMI Masters and Slaves and is implementation dependent. Pipelined transactions can only occur when both the Master and Slave support this feature. If either side does not support pipelined transactions, then only one transaction at a time is supported. Currently there are no FPGA IP blocks which support pipelined transactions.

2.2.5. State/Flow Diagrams

2.2.5.1. Simple Master

The following state/flow diagram illustrates the behavior of a simple master which does not support pipelined transactions or burst transactions. The simple master always waits for one transaction to complete before starting the next transaction. This is only one example; many other possible implementations exist.

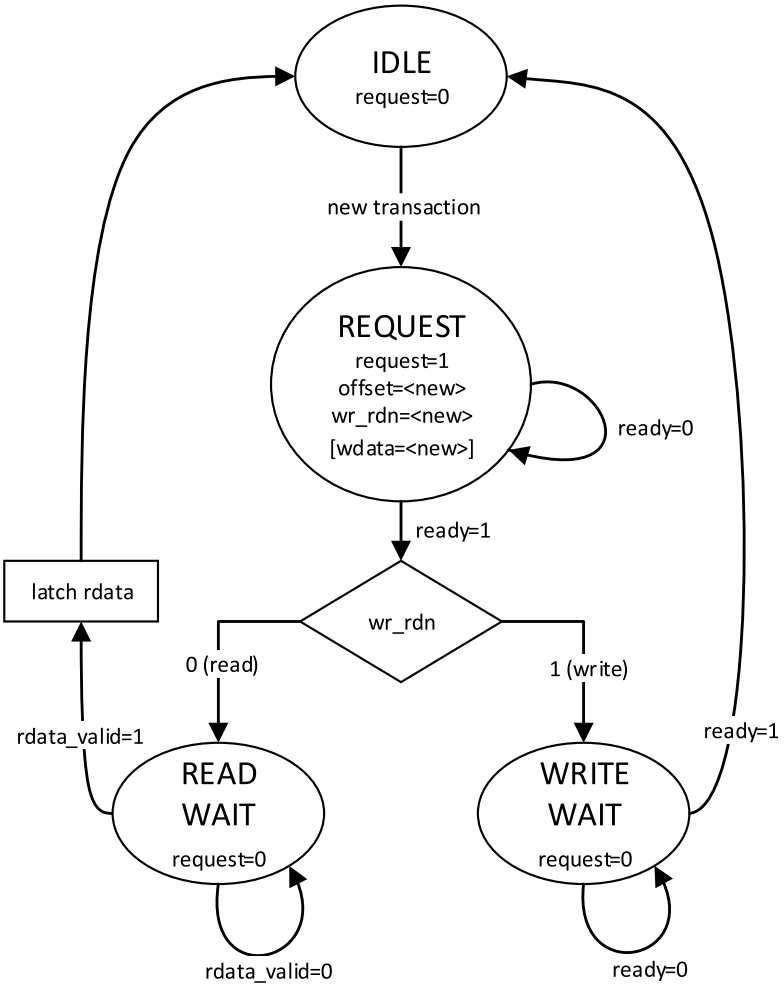

Figure 2.13. Simple Master Example

3. Lattice Interrupt Interface (LINTR)

The Lattice Interrupt Interface consists of an interrupt signal and a set of interrupt registers which are accessed through LMMI. These interrupt registers follow a standard functional definition, allowing users to implement common hardware/software to handle interrupts from a variety of IP blocks.

3.1. Signal Definitions

Table 3.1. LINTR Signal Definitions

Signal	IP Direction	Mandatory/Optional	Description
int	Out	M	Interrupt IP block has an interrupt which needs to be serviced. Active high, level sensitive. Stays high as long as any enabled interrupt is pending.

3.2. Interrupt Registers

IP blocks which support LINTR implement the following set of interrupt registers:

- Interrupt Status
- Interrupt Enable
- Interrupt Set

Each interrupt register has one or more bits which represent the interrupt sources in the IP block. The bit position of each interrupt source is the same in every interrupt register. For example, if int_src1 is assigned to bit 0 in the interrupt status register, it is assigned to bit 0 in the interrupt enable and set registers as well.

The number of interrupt source bits supported by a given IP block is dependent on the functionality of that block, but the minimum is 1 interrupt source bit. In other words, even if a block only supports one interrupt source, it implements interrupt status, enable, and set registers at least 1 bit wide.

3.2.1. Interrupt Status Register

The interrupt status register is named int_status and provides two functions.

Reading this register returns a set of bits representing all interrupts currently pending in the IP Block. The status bits are independent of the enable bits; in other words, status bits may indicate pending interrupts even though those interrupts are disabled in the int_enable register. In order to determine which interrupt source(s) generated an interrupt signal, the entity which services interrupts must mask int_status with int_enable (see the Interrupt Handling section for the recommended interrupt handling sequence).

Writing this register clears pending interrupts for each bit set to '1'. This is generally known as "write 1 to clear."

An example interrupt status register definition is shown below. Each IP Block defines its own interrupt bits and names them appropriately. For consistency among IP blocks, the bitfield names all end in _int.

Table 3.2. int_status Register Definition

7	6	5	4	3	2	1	0
RSVD	RSVD	RSVD	RSVD	src4_int	src3_int	src2_int	src1_int

Table 3.3. int_status Register Bitfield Definition

Bit	Field	Description
7:4	RSVD	**Reserved** Reads return 0 Writes are ignored
3	src3_int	**SRC3 Interrupt Status** – add interrupt name/description here Read Value: • no interrupt • interrupt pending • Write 1 to clear
2	src2_int	**SRC2 Interrupt Status** – add interrupt name/description here Read Value: • no interrupt • interrupt pending Write 1 to clear
1	src1_int	**SRC1 Interrupt Status** – add interrupt name/description here Read Value: • no interrupt • interrupt pending Write 1 to clear
0	src0_int	**SRC0 Interrupt Status** – add interrupt name/description here Read Value: • no interrupt • interrupt pending Write 1 to clear

3.2.2. Interrupt Enable Register

The interrupt enable register is named int_enable and it controls whether interrupts in the int_status register assert the int signal or not. It does not affect the contents of the int_status register. If one of the interrupt sources in the IP Block generates an interrupt, it sets the corresponding bit in the int_status register regardless of whether the interrupt is enabled or disabled in the int_enable register. See the Interrupt Signal Generation section for interrupt signal generation details.

An example interrupt enable register definition is shown below. Each IP Block defines its own interrupt bits and names them appropriately. For consistency among IP blocks, the bitfield names all end in _en.

Table 3.4. int_enable Register Definition

7	6	5	4	3	2	1	0
RSVD	RSVD	RSVD	RSVD	src4_en	src3_en	src2_en	src1_en

Table 3.5. int_enable Register Bitfield Definition

Bit	Field	Description
7:4	RSVD	**Reserved** Reads return 0 Writes are ignored
3	src3_en	**SRC3 Interrupt Enable** – add interrupt name/description here 0 – interrupt disabled 1 – interrupt enabled
2	src2_en	SRC2 Interrupt Enable – add interrupt name/description here 0 – interrupt disabled 1 – interrupt enabled
1	src1_en	**SRC1 Interrupt Enable** – add interrupt name/description here 0 – interrupt disabled 1 – interrupt enabled
0	src0_en	**SRC0 Interrupt Enable** – add interrupt name/description here 0 – interrupt disabled 1 – interrupt enabled

3.2.3. Interrupt Set Register

The interrupt set register is named int_set and it allows the user to set bits in the int_status register. Writing this register sets pending interrupts for each bit set to '1'.

It is not used in most applications, but is provided to give applications the ability to force individual interrupts.

An example interrupt set register definition is shown below. Each IP Block defines its own interrupt bits and names them appropriately. For consistency among IP blocks, the bitfield names all end in _set.

Table 3.6. int_set Register Definition

7	6	5	4	3	2	1	0
RSVD	RSVD	RSVD	RSVD	src4_set	src3_set	src2_set	src1_set

Table 3.7. int_set Register Bitfield Definition

Bit	Field	Description
7:4	RSVD	**Reserved** Reads return 0 Writes are ignored
3	src3_set	**SRC3 Interrupt Set** – add interrupt name/description here 0 – do nothing 1 – set src3_int in int_status
2	src2_set	**SRC2 Interrupt Set** – add interrupt name/description here 0 – do nothing 1 – set src2_int in int_status
1	src1_set	**SRC1 Interrupt Set** – add interrupt name/description here 0 – do nothing 1 – set src1_int in int_status
0	src0_set	**SRC0 Interrupt Set** – add interrupt name/description here 0 – do nothing 1 – set src0_int in int_status

3.2.4. Interrupt Signal Generation

The int signal is asserted whenever an interrupt status bit is set and the corresponding interrupt enable bit is also set. The signal is generated by a bitwise AND operation on int_status and int_enable and then a reduction OR of the result. The Verilog code is shown below.

```
int := | (int_status & int_enable);
```

3.3. Interrupt Handling

When interrupts occur, the IP Block sets the appropriate bit(s) in the int_status register, and the int signal is asserted if one or more of those interrupts is enabled. When int is asserted, the interrupt handler (either firmware running on a soft processor, or a state machine in logic) reads the int_status register, processes the interrupt (which may involve reading/writing various registers), and then writes to the int_status register to clear the interrupt.

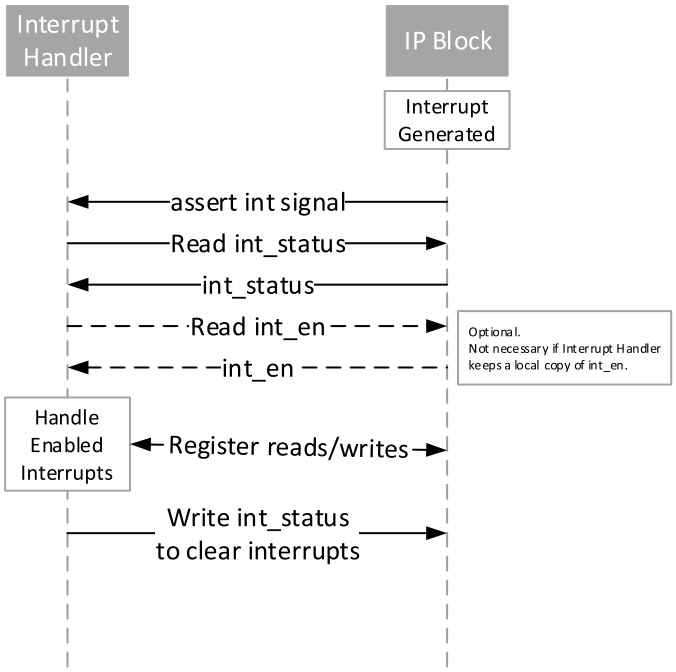

Figure 3.1. Interrupt Handling Sequence

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

Revision History

Revision 1.2, January 2020

Section	Change Summary
All	Updated document template.
Disclaimers	Added this section.
Revision History	Updated format.

Revision 1.1, February 2018

Section	Change Summary
Lattice Memory Mapped Interface (LMMI)	• Revised lmmi_ready description in Table 2.1. • Revised Reset section. • Revised Simple Master section. Added “burst transactions” to initial statement.
Interrupt Registers	In Interrupt Registers section, changed statements to: • For example, if int_src1 is assigned to bit 0 in the interrupt status register, it is assigned to bit 0 in the interrupt enable and set registers as well. • In other words, even if a block only supports one interrupt source, it implements interrupt status, enable, and set registers at least 1 bit wide. In Interrupt Status Register section, changed statement to: • For consistency among IP blocks, the bitfield names all end in _int. In Interrupt Enable Register section, changed statement to: • For consistency among IP blocks, the bitfield names all end in _en. In Interrupt Set Register section, changed statements to: • It is not used in most applications, but is provided to give applications the ability to force individual interrupts. • For consistency among IP blocks, the bitfield names all end in _set.

Revision 1.0, February 2018

Section	Change Summary
All	Initial release.

www.latticesemi.com