www.opencapi.org

OpenCAPI LPC (Memory Agent) Reference Design Guide

Version 6.32
17 July 2020

Approved

Work Group OpenCAPI LPC (Memory Agent) Reference Design Guide

OpenCAPI Enablement Work Group
OpenCAPI Consortium

Version 6.32 (17 July 2020)

Abstract

This document provides details and specifications on the OpenCAPI Lowest Point of Coherency (LPC). It is the work product of the OpenCAPI Consortium Enablement Work Group.

This document is handled in compliance with the requirements outlined in the OpenCAPI Consortium Work Group (WG) process document. Comments, questions, etc. can be submitted to membership@opencapi.org.

Contents

List of figures

List of tables

Revision log

Each release of this document supersedes all previously released versions. The change history log lists all significant changes made to the document since its initial release.

Revision date	Summary of changes
17 July 2020	Version 6.32. • Updated for atomic operations.
18 September 2018	Version 6.31. • Updated for the OpenCAPI Work Group template.

About this document

This document provides details and specifications on the OpenCAPI Lowest Point of Coherency (LPC), which is also known as the memory agent.

This section contains some general format information and document conventions.

Architecture compliance terminology

In architecture descriptions, certain terms carry meaning in addition to their normal use in English. The following terms are used within this architecture specification to describe the requirements an implementation must meet to be considered compliant.

Table i-1. Architecture terms

Term	Description
invalid	Used for multi-bit fields where the contents are not reliable. The field or bus shall not be examined for any functional or error checking actions.
may	An architectural option indicating that an implementation is allowed to have this behavior or characteristic.
reserved	With respect to a field of a register or bus: ● A reserved field shall be set to 0 by an implementation. ● A reserved field shall not be examined by an implementation. With respect to a code point: ● A reserved code point shall not be issued by a compliant implementation ● A reserved code point shall cause a bounded undefined response (that is, it won't hang the system). ● A reserved code point may be used in future revisions of the architecture. The architecture may specify that the use of a reserved code point is an error condition.
shall	An architectural requirement indicating a required behavior or characteristic.
uncertain	Used for single-bit fields where the contents are not reliable. The field or bus shall not be examined for any functional or error checking actions.
undefined	When the value of a field or a bus is undefined, the value may vary between implementations and may vary for a particular implementation for different actions. An implementation shall not examine a field when its value is undefined for functional purposes. However, the field may be checked for errors in those cases where an implementation includes error checking (that is, parity, ECC and so on).

Conventions

The OpenCAPI Consortium documentation uses several typesetting conventions.

Bit and byte numbering

Throughout this document, little-endian notation is used, which means that bits and bytes are numbered in descending order from left to right.

Thus, in the description of a 4-byte field, bit 31 is the most significant bit (MSb) and bit 0 is the least significant bit (LSb). The corresponding byte numbering is also shown.

MSb																													LSb
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
byte 3								byte 2								byte 1								byte 0							

Big-endian and little-endian byte ordering are shown in *Table i-2* and described in the POWER ISA, version 3.0, Book I.

Table i-2. Big- and little-endian comparisons

LE	7	6	5	4	3	2	1	0
	Bit numbering within a byte							
BE	0	1	2	3	4	5	6	7

4-byte field with character data shown

LE	3	2	1	0
Content:	M	I	K	E
BE	0	1	2	3

Illustrating the difference between little endian and big endian storing to memory of the 4-byte field shown to the left.

Memory offset	LE stored	BE stored	
0	E	M	
1	K	I	
2	I	K	
3	M	E	

Representation of numbers

Numbers are generally shown in decimal format, unless designated as follows:

- Hexadecimal values in running text are usually preceded by 0x.

  For example: 0x600F4.

- Binary values in running text are usually preceded by “0b” or enclosed in single quotation marks.

  For example: 0b1 or ‘1010’.

- Binary and hexadecimal values may also be shown using Verilog style representation, where the number of bits precedes the designator. For instance, 16x1234 is a 16 bit number with the value 0x1234. 5b11011 is a 5 bit number with the value 0b11011.
- A bit value that is immaterial, which is called a “don't care” bit, is represented by an “X.”

Bit significance

The bit on the left represents the most-significant bit of a field. The bit on the right represents the least-significant bit of a field. For example, in CTL[0:31], 0 is the most-significant bit. In BUS[11:0], 11 is the most-significant bit.

Notes

This section describes Engineering and Developer notes.

Engineering notes

Engineering notes provide additional implementation details and recommendations not found elsewhere. The notes might include architectural compliance requirements. That is, the text might include Architecture compliance terminology. These notes should be read by all implementation and verification teams to ensure architectural compliance.

Engineering note:

This is an example of an Engineering note. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin cursus hendrerit enim, vel tempus nibh ornare ut. Quisque ac augue eu augue convallis hendrerit. Mauris iaculis viverra ipsum nec dapibus. Nunc at porta libero. Curabitur luctus ultrices augue non pulvinar. Vestibulum mattis non ipsum at venenatis. Suspendisse euismod, neque et suscipit luctus, odio metus semper lectus, quis volutpat est libero quis nunc. Vivamus rutrum mauris sed tristique malesuada.

Developer notes

Developer notes are used to document the reasoning and discussions that led to the current version of the architecture. These notes might also include recommended changes for future versions of the architecture, or warnings of approaches that have failed in the past. These notes should be read by verification teams and contributors to the architecture.

Developer note:

This is an example of a Developer note. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin cursus hendrerit enim, vel tempus nibh ornare ut. Quisque ac augue eu augue convallis hendrerit. Mauris iaculis viverra ipsum nec dapibus. Nunc at porta libero. Curabitur luctus ultrices augue non pulvinar. Vestibulum mattis non ipsum at venenatis. Suspendisse euismod, neque et suscipit luctus, odio metus semper lectus, quis volutpat est libero quis nunc. Vivamus rutrum mauris sed tristique malesuada.

Terms

The following terms are used in this document.

AFU	Attached functional unit. Architecturally, this term refers to an endpoint unit or resource. Communication from the processor to the AFU goes through a protocol stack, transaction layer (TL), data link layer (DL), and physical medium layer (PHY). Command and data packets at the AFU interface are specified by the AFU command/data interface, which is the interface between the AFU protocol stack and the AFU.
BAR	Base Address Register.
BDI	Bad data indicator
BRAM	Block random access memory (RAM).
CAM	Content addressable memory.
CAPI	Coherent Accelerator Processor Interface.
CFG	Configuration sub-system.
DL	OpenCAPI data link layer on the host processor.
DLX	OpenCAPI data link layer on the OpenCAPI device.
DMA	Direct memory attach.
ERY	Error vector array.
FIFO	First-in, first-out.
FLIT	Flow control digit.
LPC	Lowest point of coherency.
MMIO	Memory-mapped input/output.
OCSE	OpenCAPI Simulation Engine.
PASID	Process address space ID

PHY	The PHY layer interfaces to the DL and the network. This is the bit stream level that specifies the electrical and optical transmission medium as well as the network interconnect topology. The current specification for the network is a point-to-point connection.
PHYX	On the OpenCAPI device, the PHYX layer interfaces to the DLX and the network. This is the bit stream level that specifies the electrical and optical transmission medium as well as the network interconnect topology. The current specification for the network is a point-to-point connection.
RAM	Random access memory.
RO	Read only.
RW	Read/write.
SAM	Sparse array map.
TL	OpenCAPI transaction layer found on the host processor. ● Interfaces to the DL and the protocol layer. Responsible for command-packet formation and response-packet handling and formation. Ensures that the order of data sent to the DL matches the command and response packet order sent to the DL. ● Manages data flits from the DL and associates the data with the command or response packet that was received prior to the arrival of the data. The command and response packets contain data descriptors that enable this association. ● Provides flow control. ● Provides error handling and control. ● Manages virtual channels, virtual queues, and service queues associated with the virtual channels. Order is retained within virtual channels.
TLX	OpenCAPI transaction layer found on the OpenCAPI device. ● Interfaces to the DLX and the protocol layer. Responsible for command-packet formation and response-packet handling and formation. Ensures that the order of data sent to the DLX matches the command and response packet order sent to the DLX. ● Manages data FLITs from the DLX and associates the data with the command or response packet that was received prior to the arrival of the data. The command and response packets contain data descriptors that enable this association. ● Provides flow control. ● Provides error handling and control.
VC	Virtual channels.

VPD	Vital product data.
WTC	Write-to-clear.

1. Overview

The purpose of the lowest point of coherency (LPC), also known as the memory agent, is to allow memory implemented in the attached functional unit (AFU) to act as a coherent extension of system memory.

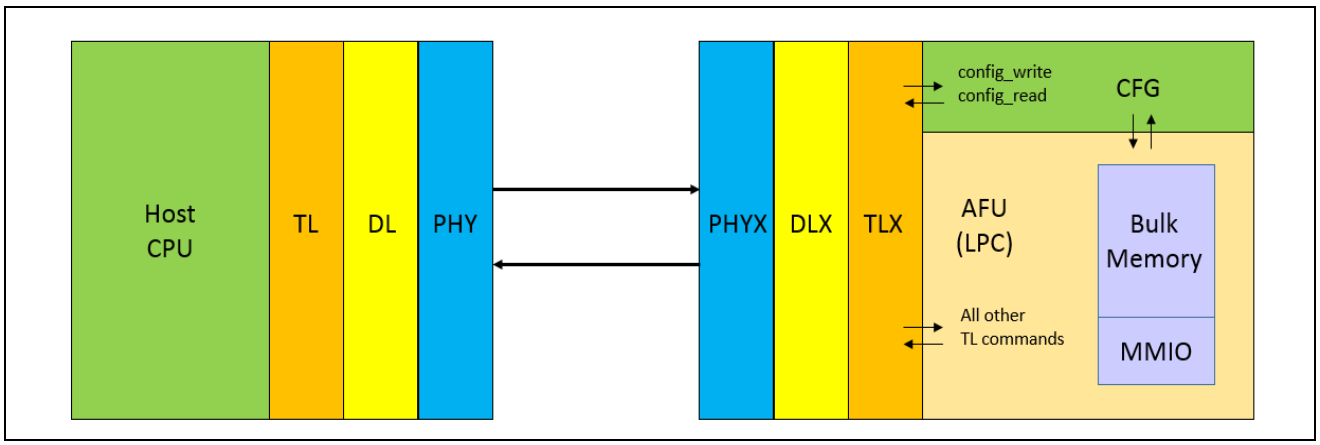
The reference design has three goals:

1. To provide a bus functional model for use in simulation, to verify host and OpenCAPI Simulation Engine (OCSE) functionality
2. To provide an FPGA implementation for use in lab bring up, to validate host hardware
3. To provide a reference design to OpenCAPI partners interested in implementing their own AFU

The LPC acts primarily as a slave device to the host. This means in most cases it responds to host issued read and write commands to configuration, MMIO, and memory spaces. The LPC only initiates commands to the host in one situation, to present an interrupt to report an error.

End-to-end, the connections look like Figure 1-1.

Figure 1-1. End-to-end connections



Several layers exist between the Host CPU and AFU. The TL ↔ TLX layer implements the commands and responses that make up the OpenCAPI protocol. It also provides the packing and unpacking of commands, responses, and data using templates to utilize the bandwidth between the end points in the most efficient manner. The DL ↔ DLX layer ensures non-bottlenecked delivery by incorporating a retry buffer, packet sequence checking, and credits allowing pipelined traffic. The PHY ↔ PHYX implement the physical drivers/receivers that handle the electrical signaling between the two ends of the link.

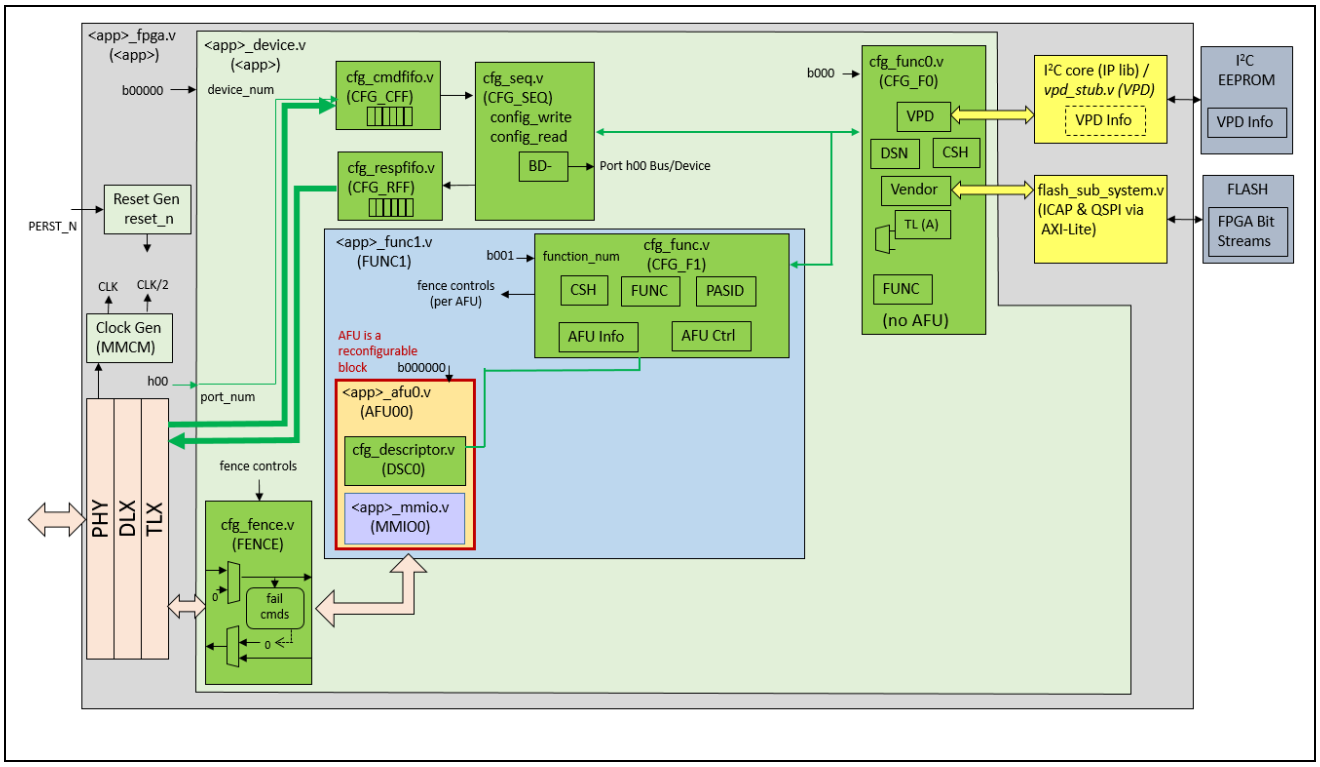
The configuration sub-system (CFG) handles software configuration of the AFU. It is described in the document *OpenCAPI 3.0 Configuration Sub-System Reference Design Specification*. As such, very little about it is discussed here with two exceptions.

First, it is important for software to recognize that config_write and config_read commands are extracted from other Host → AFU commands. While they execute in order with respect to each other, config_write/config_read commands can execute before, in parallel, or after other TL commands to the AFU. Under most circumstances this should be acceptable because initialization and configuration are usually complete before functional commands are executed. However, if software decides to mix the two command streams (configuration and

AFU), it must take this into account whenever there is an ordering dependency between them.

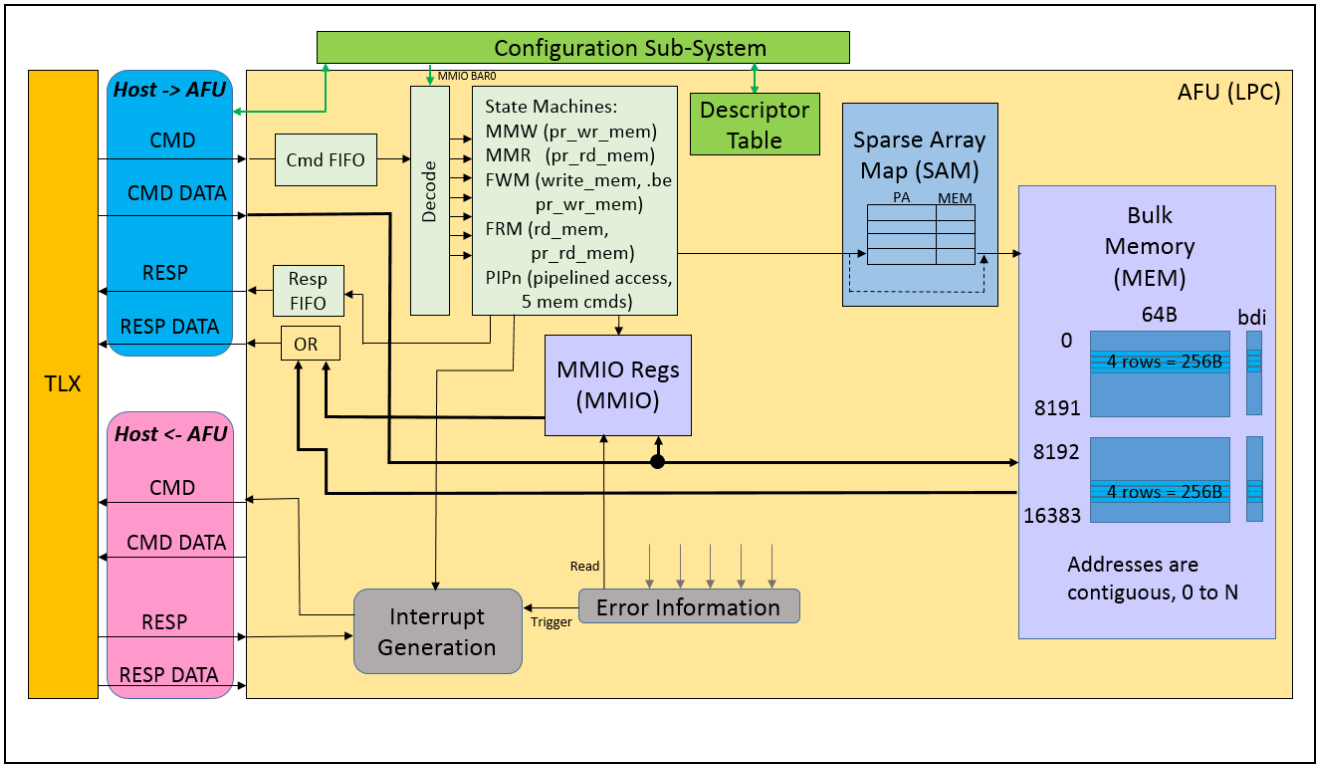
Second, Figure 1-2 shows the pieces of the configuration sub-system and their location in a single AFU system like the LPC. This is important to mention as CFG files are part of the overall LPC delivery and initialization procedure; therefore, they are introduced at a high level.

Figure 1-2. Locations of configuration sub-system in single AFU



The major components of the LPC AFU itself can be seen in Figure 1-2 (within <app>_afu0.v in Figure 1-3).

Figure 1-3. Configuration sub-system



1.1 Logical representation LPC

There are two groups of signals between the TLX and AFU. One is for Host-to-AFU operations, the other for AFU-to-Host operations. Each group consists of a command and response interface, along with command data (data travels on this during write commands) and response data (data travels on this during read commands). In the OpenCAPI TL architecture, the completion of each command is acknowledged by a response, which indicates success or failure of the command.

The LPC application uses the Host-to-AFU group in most cases. AFU-to-Host is only used for interrupts.

1.2 Operation presentation from TLX

When a command and associated data have been received by the TLX, the command is presented on the command interface to the AFU if the TLX has at least one command credit from the AFU. If the LPC state machines were connected directly to the TLX interface, they would have to capture the command when it was presented because that is the only cycle in which the signals are valid (when tlx_afu_cmd_valid = ‘1’). However, the LPC state machine that implements pipelined writes and reads to bulk memory prefers the interface stay valid over several cycles, because it might have to wait until there are enough response credits to complete the command before accepting it from the TLX. To provide this behavior, a command FIFO (CMD FIFO) is implemented to receive and store multiple commands from the TLX. This has two advantages: the latency between completing one command and starting the next one is less than reaching back into the TLX’s command array, and the state machines can stall a command from execution as long as it needs.

To get started, after reset and an indication that the TLX is ready (tlx_afu_ready= ’1’), the command FIFO pulses the command credit signal back to the TLX (afu_tlx_cmd_credit) for as many cycles as it has FIFO entries. This means that no initial credits need to be supplied to the TLX (afu_tlx_cmd_initial_credit[6:0]= 0). Commands flow from the TLX into the command FIFO until the credits are exhausted. As the LPC state machines process a command, they issue a “cmd_complete” pulse to the CMD FIFO, who then returns a command credit to the TLX to allow it to send another command to the CMD FIFO.

In the LPC, command dispatch logic polls on command valid out of the CMD FIFO (cff_cmd_valid). When it sees this, it decodes the opcode (cff_cmd_opcode) and determines which other signals on the interface are used by this command type.[1]

If the command is an MMIO or atomic operation, it waits for all other commands in progress to finish (that is, the pipeline needs to drain), then allows the current command to execute. The state machines executing MMIO operations must signal their completion back to the CMD FIFO (via cmd_complete) before the CMD FIFO presents the next command in the queue. This means MMIO and atomic operations are executed one at a time.

If the command targets bulk memory and pipelining is not enabled, then a similar situation occurs where the current command is completely executed before the CMD FIFO presents the next command in the queue.

If the command targets bulk memory and pipelining is enabled, the response credits are checked. If enough response credits are present to let the command complete, it is passed into the bulk memory write/read pipeline and the CMD FIFO is told to present the next command on the next cycle. If enough response credits are not present yet, the command stays in the CMD FIFO until enough response credits arrive. Assuming things are not gated by response credits, commands can enter the bulk memory write/read pipeline every cycle; that is, they can execute as fast as the TLX can present them.[2]

If the command carries data (i.e., write), the AFU must request it by pulsing a read request to the TLX data buffer (afu_tlx_cmd_rd_req) while simultaneously providing the size of the data it wants to obtain (afu_tlx_cmd_rd_cnt). The first 64-byte FLIT of data appears two[3] cycles later on the command data bus (tlx_afu_cmd_data_bus[511:0]) along with the qualifying valid (tlx_afu_cmd_data_valid). Data is held in the TLX FIFO until the AFU requests it, so that the AFU can take as long as it needs to process the command before requesting data. It also is up to the AFU how to read it out. For example, if the command has 256 bytes of data associated with it (4 FLITs), the AFU can obtain all of it at once by requesting 256 bytes during the single request. In this case, the TLX streams the four FLITs to the AFU, one per cycle. However, if the AFU needs to insert some delay between each request, it can also retrieve the data using four FLIT requests of 64 bytes each, two FLIT requests of 128 bytes each, or any combination it wishes. This gives the AFU flexibility in timing between receipt of the command and processing data, as well as between processing each FLIT of data.

When the AFU is done with the command, it returns a credit to the TLX to get another one by pulsing a command credit signal (afu_tlx_cmd_credit).

In the TL 3.0 architecture version of the TLX, different virtual channels (VC) are used for commands and responses. Operations arriving from host to TLX use VC1 for commands and VC0 for responses. Operations

[1] For timing purposes, opcode decode execution is done as the command enters the CMD FIFO from the TLX. The decode signals are added to the CMD FIFO and carried through with the command. This is needed because decoding the command and determining when to dispatch it is too much logic to execute in one clock cycle. Conceptually, though one can think of decode and dispatch occurring together.
[2] Depending on the mix of commands, stalling can occur between bursts of commands. Stalling can happen because it takes two cycles for the TLX to fetch the next command after receiving a command credit. If the command mix happens to receive or return multi-FLIT data, then there will be natural breaks in the command stream to allow the TLX to absorb two cycles of latency. If the commands are all 1 FLIT operations though, stalling can occur to accommodate this latency whenever the command flow stops and restarts.
[3] Strictly speaking, data arrives two or more cycles after being requested from the TLX. If there is no data transfer in progress, it arrives two cycles later. If the TLX → AFU data interface is in use when the request is made, data appears as soon as the bus is available, as long as two cycles have expired after the request. The TLX needs a minimum of two cycles to retrieve data from its buffer.

going to the host from the TLX use VC3 for commands and VC0 for responses. The AFU does not need to be concerned about virtual channels because the TLX has separate interfaces for commands and responses in both directions.

All logic in the LPC is synchronous to a single clock, which must be the same clock used by the TLX.

1.2.1 Matching bandwidth to the host

When implemented in an FPGA, the LPC clock is targeted to run at ¼ the clock frequency of the POWER9 host (for example, 400 MHz[4] versus 1.6 GHz). This means to match aggregate bandwidth assuming an equal mix of commands and data in both directions, the AFU would need to receive and supply one command and one response in each direction on every cycle, as the POWER9 host can work with only 1 command or response at a time. If the mix of operations is not balanced, the AFU subsystem can fall behind the POWER9's capability for processing or operations. Because under normal circumstances, the LPC does not generate commands or receive responses to generated commands, it theoretically has a maximum bandwidth of about ½ the host. Implementation details that create pauses in the stream of operations will degrade this more.

1.3 Nonpipelined operations

Write and read commands associated with MMIO operations always execute in a non-pipelined fashion. This is because they are not performance critical; therefore, design resources can be conserved by making their implementation as simple as possible. Because the order of these commands is preserved, software can use an MMIO read as a flush mechanism to ensure all preceding MMIO write commands have completed.

In nonpipelined mode, writes and reads to bulk memory are also executed one at a time. This can be useful to simplify debug or to exercise variations in the execution of a command. Adding variation to the pipelined logic can be difficult depending on the variation; therefore, all test options that vary things like how data is read from the TLX data buffer or responding to commands in multiple parts is planned to be added to the non-pipelined version of the command implementation.

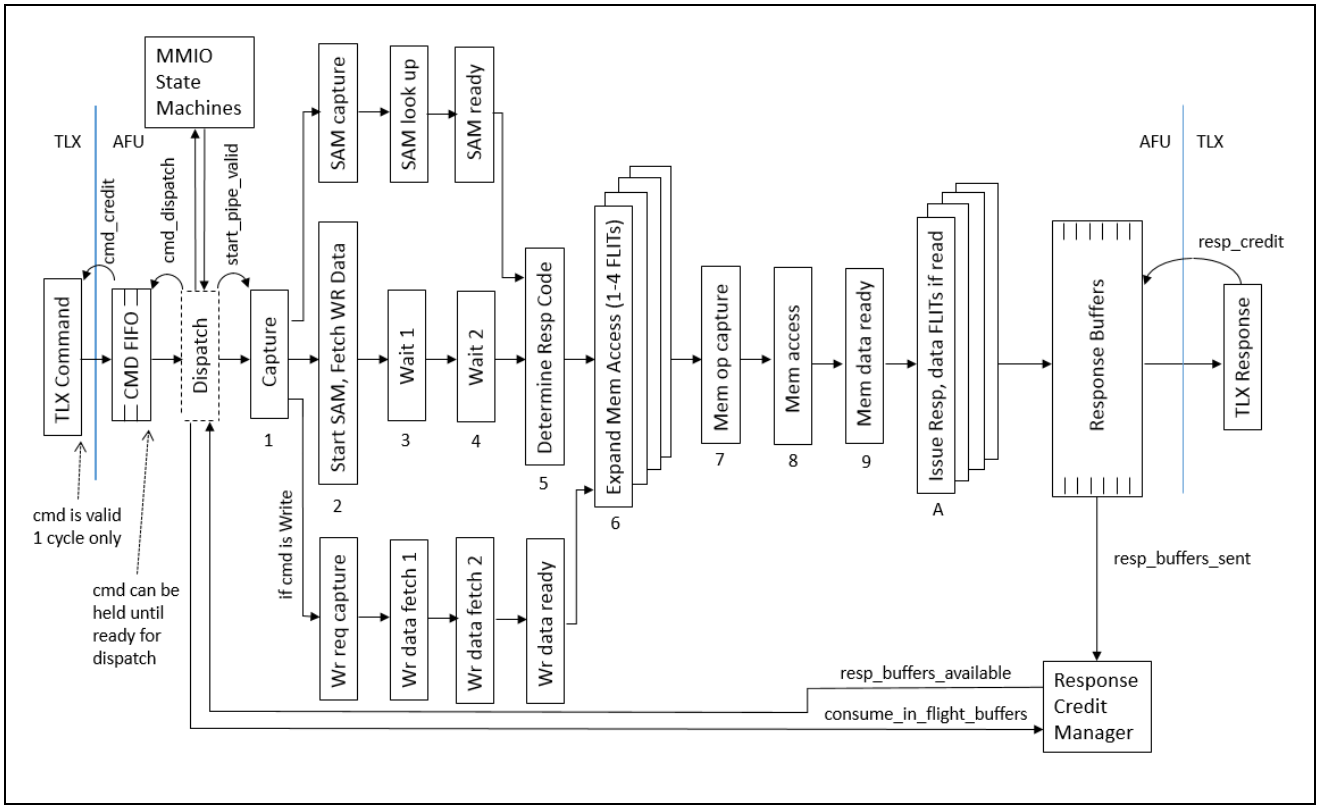
All atomic operations are also executed in a non-pipelined fashion.

1.4 Pipelined operations

To support bandwidth testing of the OpenCAPI link and TLX, the LPC has a mode to enable pipelined processing of write and read commands to bulk memory. When enabled, three types of writes (write_mem, pr_wr_mem, write_mem.be) and two types of reads (rd_mem, pr_rd_mem) can be executed, one per cycle. This assumes an appropriate command mix to prevent gaps in command presentation and enough response credits from the TLX exist to prevent stalls in the dispatching of commands. Figure 1-4 shows the states of the pipeline.

[4] The FPGA frequency goal is slightly higher, 402.83 MHz (2.482 ns period) to match a serial link running at 25.78125 Gbps.

Figure 1-4. Pipeline states



If the command presented by the CMD FIFO is decoded as an MMIO write or read, it is dispatched as soon as the pipeline no longer contains any commands. Draining the pipeline is required to preserve the order of commands. The MMIO command must completely finish before another command is dispatched from the CMD FIFO.

If the command is a write or read to bulk memory, it is allowed to enter the pipeline (also known as dispatched) if two conditions are met:

- There are enough free entries in the Response Buffer to hold the response and all response data associated with the command.
- No write data FLIT is expected to use this cycle. A write command can have 1 - 4 data FLITs associated with it and because each FLIT consumes a cycle in the pipeline, commands are not allowed to enter until a preceding write includes enough separation to contain all the data FLITs.

By pre-determining when it is appropriate to dispatch a command, once it enters the pipeline, a command is guaranteed to finish in a known number of cycles. No backpressure mechanism is used within the pipeline itself, simplifying the transfer of information from stage-to-stage. Instead, back pressure from the TLX response interface is applied through Response Buffer credits and checked by the Dispatch stage before allowing a command to enter. Designing the CMD FIFO to present and indefinitely hold a command is critical to avoid losing commands when there are not enough response buffer credits. While this can result in the CMD FIFO filling up, back pressure from it to the TLX is achieved by throttling AFU-to-TLX command credits to prevent a CMD FIFO overflow.

The Response Buffer is required because the TLX response buffer size is too small (that is, four deep) to cover all the stages of the pipeline. Without it, commands can only enter in bursts of four, and then must wait for the TLX to provide response credits back to the head of the pipeline before more commands can enter.

Within the pipeline, Sparse Array Mapping (SAM) address translation and the fetching write data can occur simultaneously. The two forks merge before accessing bulk memory. Depending on the number of data FLITs, write accesses can be expanded from 1 - 4 cycles; likewise, read accesses can be extended from 1 - 4 cycles. Eventually responses and response data end up in the Response Buffer, where they are sent out to the TLX as fast as it can accept them.

1.5 Command order

In the version of LPC using BRAM blocks as bulk memory, the order of all commands is preserved at all times.

1.6 Operation decode

The LPC has different state machines for each operation type it supports. When decoding a valid command, the LPC decides which state machine to start based on the opcode. The signals that kick off the correct state machine have a unique naming convention (‘start_*’), making them useful to observe when debugging a test. Also state machines and signals/registers used in each state have a 3-character string to identify it with a particular state and to avoid collisions of similar named signals in different states.

Table 1-1. Opcodes for non-pipelined state machines

State Machine	Abbreviation	OPcode	Start Signal Name
N/A	NOP	0x00 and BAR match	Start NOP
SM_MMW	MMW	0x86 and BAR match	start_mmio_write
SM_MMR	MMR	0x28 and BAR mismatch	start_mmio_read
SM_FWM*	PWM	0x86 and BAR mismatch	start_pr_wr_mem
SM_FRM*	PRM	0x28 and BAR mismatch	start_pr_rd_mem
SM_FWM	FWM	0x81 and BAR mismatch	start_write_mem
SM_FRM	FRM	0x20 and BAR mismatch	start_rd_mem
SM_FWM*	WMB	0x82 and BAR mismatch	start_write_mem_be
SM_AMO	AMO_RD	0x30 and BAR mismatch	start_amo_rd
SM_AMO	AMO_W	0x40 and BAR mismatch	start_amo_w
SM_AMO	AMO_RW	0x38 and BAR mismatch	start_amo_rw

Table 1-1 is for the non-pipelined state machines. While commands are decoded individually, to reduce logic the execution of `start_pr_rd_mem` is folded into the SM_FRM state machine and the execution of `start_pr_wr_mem` and `start_write_mem_be` are folded into the SM_FWM state machine.

Pipelining is implemented differently, because each stage of the pipeline has the prefix PIP#_ where # = 1,2,…,9,A and each character represents a different stage of the pipeline. The start signals are grouped into a vector start_pipe_type[4:0], making it easier to pass them from stage-to-stage. Each bit of the vector represents a different command.

Table 1-2. Pipelining

Bit Number	OPcode	Start Signal Name
4	0x00 and BAR mismatch	start_pipe_type[PIPE_TYPE_RD_MEM]
3	0x28 and BAR mismatch	start_pipe_type[PIPE_TYPE_PR_RD_MEM]
2	0x81 and BAR mismatch	start_pipe_type[PIPE_TYPE_WRITE_MEM]
1	0x86 and BAR mismatch	start_pipe_type[PIPE_TYPE_PR_WR_MEM]
0	0x82 and BAR mismatch	start_pipe_type[PIPE_TYPE_WRITE_MEM_BE]

Partial write and partial read commands can go to either MMIO registers or bulk memory. What determines the target is a match or mismatch of the upper address bits. It is expected that the host will configure Base Address Register 0 (BAR0) and set the Memory Space bit in the Configuration Space Header attached to the AFU (that is, Function 1) before attempting a partial write or read. This BAR register determines the starting address of the MMIO space. If a partial write or read operation falls within the MMIO space, it is routed to the MMIO registers. If it falls outside the space, it is routed to bulk memory.

All full write and read operations (`write_mem, write_mem.be, rd_mem`) are routed to bulk memory. However, if the address targets the MMIO space, it is treated as an unrecognized opcode because the host should not be accessing MMIO space with full-sized commands.

1.7 MMIO BAR0 register

The BAR0 register sets the start of the MMIO space. It is a 64-bit address that is formed by concatenating Configuration Space Header registers 0x014 and 0x010 in the configuration sub-system:

```
cfg_f1_csh_mmio_bar0[63:0] =

  {reg_csh_014_q[31:0], reg_csh_010_q[31:4], 4b0100};
```

The size of the MMIO space implemented in the LPC is passed to software using the lower bits of register 0x010. The idea is for software to discover the size of the implemented MMIO space by writing the BAR registers with all ‘1’s and reading them back. Bit positions which come back as ‘1’ are treated as upper address bits that software can change. Bit positions that read back as ‘0’ are assumed to be part of the MMIO space. The number of bits that read back ‘0’ determine the MMIO space size implemented by hardware.

Note: The lower 4 bits have a different meaning. They are assumed to be 0b0000 when filling in the lower 4 bits of BAR0 address and MMIO size.

For example, assume software writes both registers (0x014 and 0x010) to all 1s and that reading them returns:

```
0x014 = 0xFFFF_FFFF, 0x010 = 0xFFF0_0004
```

Software interprets the lower 20 bits, where [19:0] (with [3:0] assumed to be 0b0000) are ‘0’ to mean the MMIO space is 1 MB in size. Software can change the upper address bits (63:20) to move the MMIO space wherever it wants to in AFU memory. The lower address bits [19:0] are used as an offset within the MMIO space to select a specific MMIO register.

The intent is for software to position MMIO space so that it and bulk memory do not overlap. Because bulk memory starts at 0x0000_0000_0000_0000 and increases to some address (such as N), software should position MMIO space above address N.

1.8 Memory mapped I/O registers

Memory mapped (MMIO) registers are implemented in the file `lpc_mmio_regs.v`. Because they are specific to a particular AFU, the file is part of the LPC source, but it was intentionally designed as a separate module so that a version can be copied and modified as a jump start on other AFU designs.

MMIO accesses are done using partial write memory (`pr_wr_mem`) and partial read memory (`pr_rd_mem`) commands. Data accesses can be 1, 2, 4, or 8 bytes wide, with the address naturally aligned to the size.[5]

The following tables describe the registers implemented by the LPC.

- Address offsets are in hexadecimal.
- Access can be Read Only (RO), Read/Write (RW), and Write-to-Clear (WTC - meaning write the bit to '0' to clear it).
- Read Only register fields have no reset value because the state of the hardware determines it.
- <unassigned> means the register field is implemented but has no assigned purpose at this time. While software can write and read <unassigned> fields, it is recommended to leave them alone as they can be re-tasked at any time.
- Software should use the 'Scratch Pad' instead for experimental or test purposes.

The 1 MB of MMIO space is divided into two sections, 0 to (512 KB - 1) is the 'Global MMIO space' and 512KB to (1 MB -1) is reserved for 'Per PASID' registers. This means 0x0_0000 to 0x7_FFFF is for Global MMIO space and 0x8_0000 to 0xF_FFFF is for Per PASID space.

1.8.1 Global MMIO space

Table 1-3. Control register

Offset (hex)	Register Name			
00000	CONTROL These bits control the behavior of the LPC.			
Field Name	Bits	Access	Reset Value	Description
<unassigned>	63:6	RW	58b0	
enable_retries	5	RW	1b0	Enables "false retries." When '1' there is a 25% chance of retry (12.5% regular and 12.5% lightweight retry) assigned as response to R's and W's. Useful for simulation and verification but should be avoided when the LPC is connected to a real host.
enable_pipeline	4	RW	1b0	When '1', pipelining is enabled on bulk memory writes and reads. When '0', one write or read is processed at a time.
<unassigned>	3	RW	1b0	
ignore_nomatch_on_read	2	RW	1b0	When '0', reading from bulk memory that has not been written to previously (that is, sam_no_match=1) will discard the read and cause a bad response code. When '1', this check is suppressed and uninitialized read data is returned as good data (that is, BDI might be '0'). This mode can be useful for random simulation but should be

[5] Natural alignment means the address and size start on a boundary where the lowest address bits are '0'. For instance, a 1-byte size points to any byte address. A 2-byte size requires address[0] = '0'. A 4-byte size requires address[1:0] = '00'. An 8-byte size requires address[2:0] = '000'.

				avoided when the LPC is connected to a real host. Note that read data is still sourced from bulk memory, therefore will be 'X', '0', or 'other default value loaded in the array.
trigger_sfw_intrp	1	RW	1b0	Write to '1' to trigger loading the contents of SFW_INTRP_HI and SFW_INTRP_LO into the error vector array, which triggers an interrupt if interrupts are enabled. Note: Hardware automatically clears this bit after the interrupt is in the error vector array.
sam_disable	0	RW	1b0	When '0', mapping of input address to bulk memory is performed. When '1', no mapping is performed; the input address is passed directly through as the bulk memory address.

Table 1-4. SFW_INTRP_HI register

Offset (hex)	Register Name			
00008	SFW_INTRP_HI			
Field Name	Bits	Access	Reset Value	Description
sfw_intrp_loadsrc[15:0]	63:48	RO	16x4000	Error source identifier for software generated interrupt
sfw_intrp_hi_data[47:0]	47:0	RW	48b0	User defined value, passed as interrupt error data

Table 1-5. SFW_INTRP_LO register

Offset (hex)	Register Name			
00010	SFW_INTRP_LO			
Field Name	Bits	Access	Reset Value	Description
sfw_intrp_lo_data[63:0]	63:0	RW	64b0	User defined value, passed as interrupt error data

Table 1-6. Unassigned register

Offset (hex)	Register Name			
00018	< implemented but unassigned>			
Field Name	Bits	Access	Reset Value	Description
<unassigned>	63:0	RW	64b0	

Table 1-7. Status register

Offset (hex)	Register Name			
00100	STATUS Allows software to query LPC status.			
Field Name	Bits	Access	Reset Value	Description
tlx_afu_ready	63	RO	--	When ‘1’, TLX is telling the AFU it is ready to operate.
sam_entries_open[4:0]	62:58	RO	--	Number of available mapping resources in Sparse Array Map.
ery_data_valid	57	RO	--	When ‘1’, valid error information is ready for software to read in ERR_INFO1 and ERR_INFO2.
<unassigned>	56:0	RO	--	

Table 1-8. ERROR register

Offset (hex)	Register Name			
00108	ERROR[6] Inputs are captured and held until the register bit is written to ‘0’. Each bit holds a different error condition. The input source can be a pulse or steady signal, the register will capture and hold it once set to ‘1’. If software writes a bit to ‘1’ for test purposes, it only affects this register and not the rest of the LPC. Note that writing the bit to ‘0’ only clears the register bit, not the original source of the error, so that it can be changed back to ‘1’ immediately after the write to ‘0’ is completed.			
Field Name	Bits	Access	Reset Value	Description
detect_bad_op	63	WTC	1b0	When ‘1’, the host command contains an unrecognized opcode or unrecognized opcode + MMIO BAR combination.
write_mem_DL_is_reserved	62	WTC	1b0	When ‘1’, a write_mem command was received with an invalid DL value of ’00’.
<unassigned>	61	WTC	1b0	<Was ‘read_mem_DL_is_reserved’, but this is not a hang condition. A bad resp_code covers this.>
<unassigned>	60	WTC	1b0	<Was ‘afu_cfg_bad_op_or_align’, but this is not a hang condition. A bad resp_code covers this.>
<unassigned>	59	WTC	1b0	<Was ‘mmio_bad_op_or_align’, but this is not a hang condition. A bad resp_code covers this.>
received_bad_op	58	WTC	1b0	When ‘1’, the TLX presented a response with an unrecognized opcode value.
err1_lock_afu_to_host_intf	57	WTC	1b0	When ‘1’, the interrupt generator or test mode logic tried to obtain the AFU → Host interface when it was already in use.

[6] ERROR register outputs are also visible on a signal in lpc_afu.v called mmio_out_captured_errors[63:0].

err2_lock_afu_to_host_intf	56	WTC	1b0	When ‘1’, the interrupt generator or test mode logic tried to release the AFU → Host interface when it was already free.
cff_fifo_overflow	55	WTC	1’b0	When ‘1’, Command FIFO was written with more entries than it can hold.
pip2_illegal_dl	54	WTC	1b0	When ‘1’, pipeline detected dL=’00’ on write_mem or rd_mem.
rff_fifo_overflow	53	WTC	1b0	When ‘1’, Response FIFO was written with more entries than it can hold.
err_incorrect_endianness	52	WTC	1b0	When ‘1’, a TLX command in big-endian format was received. Except for atomic operations, the LPC only supports little-endian format. Big-endian format is supported only for atomic operations, and if these operations are enabled, this error check must be disabled.
vpd_err_unimplemented_addr	51	WTC	1b0	When ‘1’, indicates an access to VPD targeted an unimplemented address.
desc_err_unimplemented_addr	50	WTC	1b0	When ‘1’, indicates an access to the AFU Descriptor Table targeted an unimplemented address.
SM_IRQ is in error state	49	WTC	1b0	When ‘1’, fatal error in Interrupt state machine.
<unassigned>	48	WTC	1b0	
TLX → AFU command credit overflow	47	WTC	1b0	When ‘1’, AFU received too many command credits from TLX.
TLX → AFU command credit underflow	46	WTC	1b0	When ‘1’, AFU received too few command credits from TLX.
TLX → AFU response credit overflow	45	WTC	1b0	When ‘1’, AFU received too many response credits from TLX.
TLX → AFU response credit underflow	44	WTC	1b0	When ‘1’, AFU received too few response credits from TLX.
TLX → AFU command data credit overflow	43	WTC	1b0	When ‘1’, AFU received too many command data credits from TLX.
TLX → AFU command data credit underflow	42	WTC	1b0	When ‘1’, AFU received too few command data credits from TLX.
TLX → AFU response data credit overflow	41	WTC	1b0	When ‘1’, AFU received too many response data credits from TLX.
TLX → AFU response data credit underflow	40	WTC	1b0	When ‘1’, AFU received too few response data credits from TLX.
Response Buffer credit overflow	39	WTC	1b0	When ‘1’, response buffer credit manager overflowed
Response Buffer credit underflow	38	WTC	1b0	When ‘1’, response buffer credit manager underflowed
TLX Port 0 CFG Command FIFO Overflow	37	WTC	1b0	When ‘1’, the configuration command FIFO on TLX Port 0 overflowed
TLX Port 1 CFG Command FIFO Overflow	36	WTC	1b0	When ‘1’, the configuration command FIFO on TLX Port 1 overflowed
TLX Port 0 CFG Response FIFO Overflow	35	WTC	1b0	When ‘1’, the configuration response FIFO on TLX Port 0 overflowed
TLX Port 1 CFG Response FIFO Overflow	34	WTC	1b0	When ‘1’, the configuration response FIFO on TLX Port 1 overflowed
<unassigned>	33:32	WTC	2b0	

Bulk Memory Error: simultaneous read and write operations	31	WTC	1b0	When ‘1’, logic attempted to do read and write operation to bulk memory at the same time.
Bulk Memory Error: natural alignment error on multi-FLIT read	30	WTC	1b0	When ‘1’, the combination of starting address and number of FLITs exceeds the 256-byte boundary at the bulk memory interface.
Bulk Memory Error: internal error	29	WTC	1b0	When ‘1’, bulk memory encountered a fatal internal error managing number of FLITs.
SAM error: mapping resources exceeded (sam_overflow)	28	WTC	1b0	When ‘1’, the Sparse Array Map logic ran out of mapping resources. This is a fatal error but caused by an implementation choice of the LPC and not something illegal with respect to the AFU architecture.
ERY error: simultaneous load	27	WTC	1b0	When ‘1’, multiple errors tried to save error information in the Error Array at the same time. Information, possibly associated with an interrupt, was lost.
ERY error: overflow	26	WTC	1b0	When ‘1’, the Error Array lost information because all the locations were full when another save to it was attempted.
IRQ error: irq_acs_settings_err	25	WTC	1b0	When ‘1’, config space settings are not right to allow the LPC to send an interrupt. 1. AFU Control DVSEC x0C[24] ‘Enable AFU’ must be ‘1’
PASID error: pasid_acs_settings_err	24	WTC	1b0	When ‘1’, config space settings are not right to use the PASID value set in the AFU’s PASID MMIO space 1. AFU Control DVSEC x10[12:8] ‘PASID Length Enabled’ must be ≥ 0 2. AFU Control DVSEC x14[19:0] ‘PASID Base’ must be ≤ intrp_pasid
<unassigned>	23:0	WTC	24b0	

Table 1-9. ERR_INFO_HI register

Offset (hex)	Register Name			
00110	ERR_INFO_HI Upper bits of error information associated with current interrupt.			
Field Name	Bits	Access	Reset Value	Description
ery_data_out[127:64]	64	RO	--	See Interrupt section for details. If there is no error information pending, ‘0’s are returned.

Table 1-10. ERR_INFO_LO register

Offset (hex)	Register Name			
00118	ERR_INFO_LO Lower bits of error information associated with current interrupt. **Caution:** The action of reading this register changes the contents of ERR_INFO_HI and ERR_INFO_LO to the information associated with the next interrupt. Therefore, only read this register once per interrupt. Unexpected reads can cause improper operation of the error recording / reporting mechanism. Use 'ery_data_valid' in the STATUS register to determine if there is any useful information in ERR_INFO_HI/LO.			
Field Name	Bits	Access	Reset Value	Description
ery_data_out[63:0]	64	RO	--	See Interrupt section for details. If there is no error information pending, '0's are returned.

Table 1-11. SCRATCH_PAD register

Offset (hex)	Register Name			
00200 00208 00210 … 002F0 002F8	SCRATCH_PAD Thirty-two contiguous registers form a 256-byte scratch pad area. Software can write or read these registers at will, the LPC takes no action based on their contents			
Field Name	Bits	Access	Reset Value	Description
<not applicable>	63:0	RW	64b1	All bits in the scratch pad are initialized to '1'.

1.8.2 Per PASID space

There is 64 KB of memory reserved for each PASID.

Table 1-12. INTRP_ADDR register

Offset (hex)	Register Name			
80000	INTRP_ADDR			
Field Name	Bits	Access	Reset Value	Description
intrp_ea[63:2]	63:2	RW	62b0	Interrupt message address[63:2]
Reserved	1:0	RO	2b00	Interrupt message address[1:0] These bits are set to '00' ensuring doubleword alignment.

Table 1-13. INTRP_CTRL_DATA register

Offset (hex)	Register Name			
80008	INTRP_CTRL_DATA			
Field Name	Bits	Access	Reset Value	Description
Reserved	63:33	RO	31b0	These bits are reserved.
intrp_vec_mask	32	RW	1b1	Interrupt vector control '0' = interrupt vector is enabled '1' = interrupt vector is disabled Note: LPC only uses bit [32].
intrp_data[31:0]	31:0	RW	32b0	Interrupt message data[31:0] Note: Not used in LPC application.

Table 1-14. Pending Bit Array register

Offset (hex)	Register Name			
80010	INTRP_PBA (Pending Bit Array)			
Field Name	Bits	Access	Reset Value	Description
Reserved	63:1	RO	63b0	LPC only implements one interrupt. The remaining bits are tied to '0'.
intrp_is_pending	0	RO	--	Indicates that the LPC wants to send an interrupt, but interrupts are masked. '0' = No interrupt is pending or interrupts are not masked. '1' = An interrupt is waiting to be sent, but interrupts are masked.

Table 1-15. INTRP_PASID register

Offset (hex)	Register Name			
80018	INTRP_PASID LPC-specific PASID Entry, one per PASID			
Field Name	Bits	Access	Reset Value	Description
intrp_afutag[15:0]	63:48	RW	16b0	Used as AFUTag[15:0] field in intrp_req's issued by the LPC.
intrp_stream_id[3:0]	47:44	RW	4b0	Used as STREAM_ID[3:0] in intrp_req's issued by the LPC.
intrp_cmd_flag[3:0]	43:40	RW	4b0	Used as CMD_FLAG[3:0] in intrp_req's issued by the LPC.
<unassigned>	39:20	RW	20b0	
intrp_pasid[19:0]	19:0	RW	20b0	Used as PASID[19:0] on the assign_actag command issued by the LPC before sending intrp_req's. Note: This must be greater than or equal to the value in 'PASID Base' in the configuration space AFU Control DVSEC x14[19:0].

1.9 Sparse array map

When bulk memory is implemented in the FPGA block RAM (also known as, BRAM), the LPC is limited to a relatively small amount of memory compared to the address range allowed by the OpenCAPI architecture. This can be a problem if device drivers want to use address ranges across a larger span like 4 TB or 8 TB, which might be more reasonable in a real LPC adapter.

The LPC therefore implements a Sparse Array Map (SAM), which gives the appearance of spanning a large address range using a much smaller amount of total memory. It does this by dividing the physical memory into 256-byte chunks of storage (64 bytes x 4 FLITs). Each chunk, or group of chunks, is associated with a translation register that assigns a larger number of address bits to it. As long as chunks of memory and SAM translation registers are available, the LPC can map a larger address range into contiguous bulk memory.

Strictly speaking, this might be in violation of the OpenCAPI AFU architecture. Per the architecture, the AFU is supposed to implement contiguous memory starting from address 0 and increasing to some maximum address. However, if the LPC gives the appearance of contiguous memory to the host, Sparse Array Mapping can be used. The only difficulty is when the SAM is out of translation resources, the LPC has a problem even if the host issues a legal address that would fall within the advertised contiguous memory range. In this case, the LPC enters an unrecoverable fatal error state. Thus, when allowed by the test conditions, the user is advised to keep accesses grouped together using nearby addresses to avoid consuming mapping resources.

For example, assume the physical (bulk) memory implemented in the LPC is 1 MB, organized as 16,384 rows of 64-byte FLITs. At four FLITs per chunk, this is 4096 chunks that can have translated addresses associated with them. If the host is trying to access addresses over a 4 TB range, it uses address bits [41:0]. In this case, the SAM has 4096 registers each containing 34 bits. The registers are arranged as a contiguous list, so that each one has an assigned number ranging from 0 - 4095 matching the 4096 memory chunks one-for-one. The 34 bits are used to hold address bits [41:8] of the incoming address associated with the chunk, bits [7:6] select a FLIT within the 256 bytes, and low-address bits [5:0] select a byte within the 64-byte FLIT.

As a host read or write enters the SAM, all 4096 registers marked “in use” are searched for a match. If there are no matches, the next unused register is loaded with address bits [41:8], marked as ‘in use’, and the chunk associated with the assigned number is accessed by the read or write operation. From this point forward, all accesses to that address use that chunk. [This is a Content Addressable Memory (CAM) function.]

Because architecturally the host does not know about the SAM, there is no way for it to change a chunk from ‘in use’ to ‘unused’. Therefore, when all translation registers in the SAM are used, the LPC is at maximum capacity and the next non-matching access will cause a fatal error. Hopefully, simulation and lab exerciser software can live within this restriction.

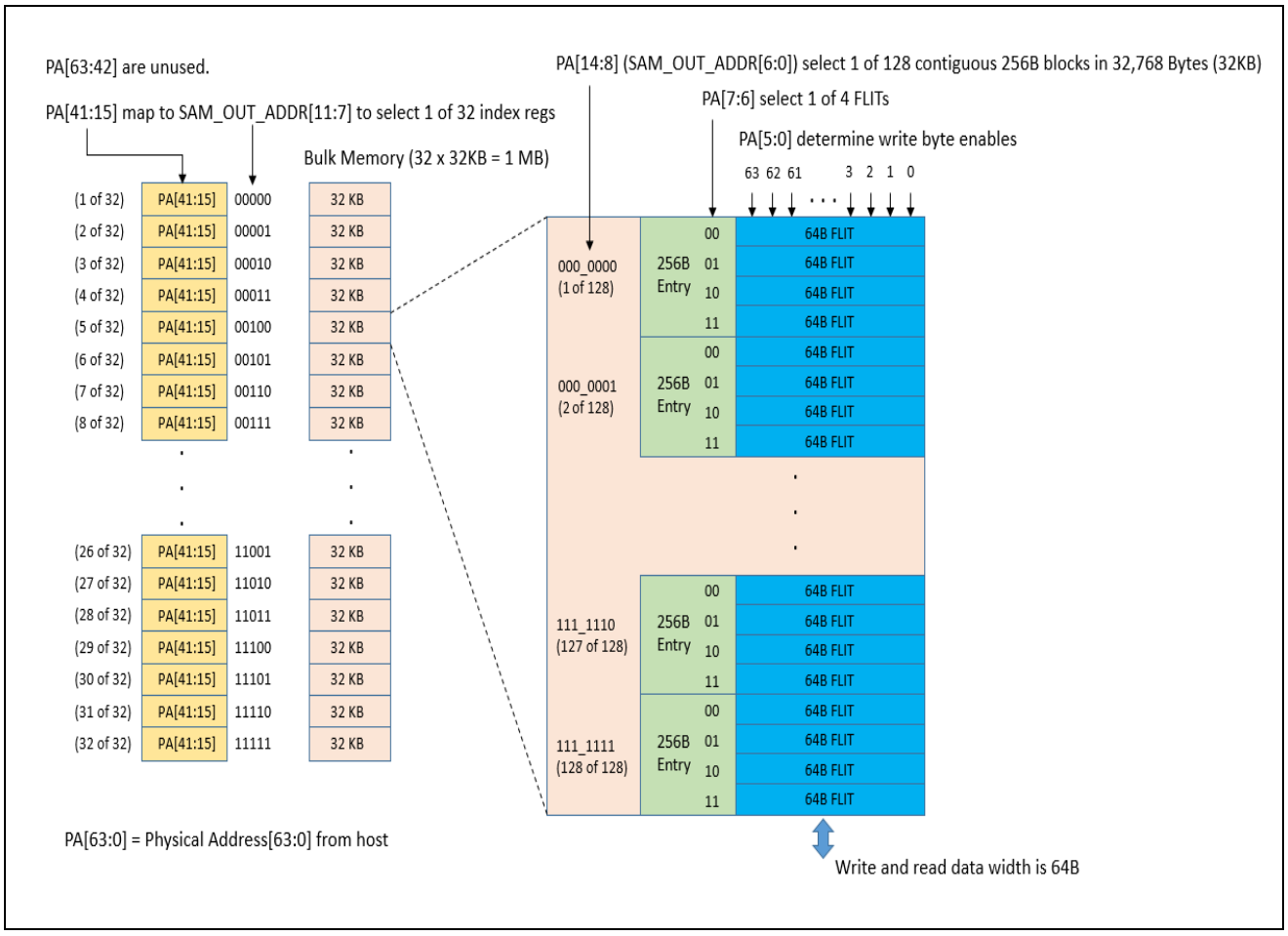
While this example used 4096 translation registers, this is too many for an FPGA implementation as the simultaneous match queries and combining of match results is quite hardware intensive. However, the same concept can be applied by using fewer translation registers and increasing the chunk size associated with it. For instance, 4096 registers with 256-bytes per chunk is the same amount of physical memory as 32 registers with 32,768-bytes per chunk, as is 16 registers with 65,536 bytes per chunk. All of them use the same 1 MB of physical memory, but with a fewer number of translation registers containing less upper address bits to translate. The extra address bits on the lower side simply get passed through to index into a larger chunk size.

After experimentation using preliminary FPGA timing rules in Vivado synthesis, place, route, and timing, it was determined 32 registers is the practical limit in the selected FPGA technology at the target speed of 400 MHz. Thus, in the initial versions of the LPC, 1 MB of bulk memory is divided into 32 chunks of 32,768 (32K) bytes per chunk. To support 4 TB of addressing, the physical address from the host (PA[63:0]) is broken out thusly:

- PA[63:42] are ignored. Aliasing can result if these bits are used
- PA[41:15] are associated with 1 of 32 mapping registers, used to select 1 of the 32 chunks

- PA[14:8] are used to select 1 of 128 contiguous entries of 256 Bytes in a chunk
- PA[7:6] are used to select 1 of 4 FLITs in a 256 Byte entry
- PA[5:0] are used to determine the write byte enables, one for each byte

Figure 1-5. Address mapping



Address mapping can be disabled by setting the MMIO bit SAM_DISABLE to 1. When this bit is set, PA[19:8] is directly connected to SAM_ADDR_OUT[11:0], which effectively makes bulk memory a single contiguous space. If using SAM_DISABLE, be sure to set it before any bulk memory accesses take place because turning it on or off in the middle of testing can cause problems.

Caution: If the intent is to use SAM_DISABLE to treat bulk memory as a 1 MB contiguous space, be aware the configuration space field ‘Mem Size’ in the AFU Descriptor Table might not match. The hardcoded value for this field is 4 TB, because the default value of SAM_DISABLE is ‘0’. This can cause a conflict in what software thinks the LPC memory size is, because at configuration time it is presented as 4 TB while at run time it is changed to 1 MB. If the user wishes, they can change this default to 1 MB by making a change to the AFU Descriptor Table input found in `lpc_afu.v`. While this works easily in simulation, in hardware an FPGA rebuild must be done and a new bit image applied for the change to be observed.

Figure 1-6. AFU descriptor table input

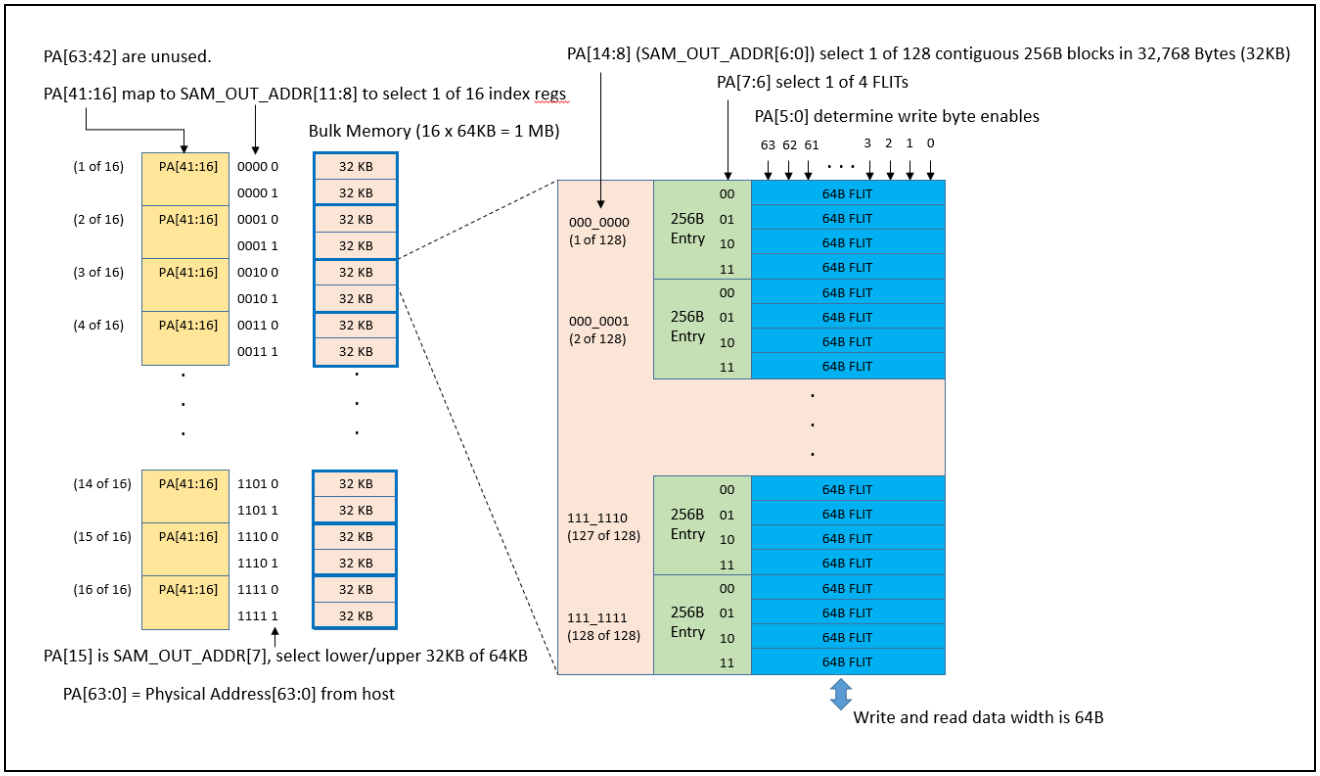
```
4452 cfg_descriptor DESC (
4453    .clock                                ( clock                                )
4454  , .reset                                ( reset                                )  // (positive active)
4455    // READ ONLY field inputs
4456                                          // 22222111111111111000000000
4457                                          // 43210987654321098765432l   Keep string exactly 24 characters long
4458  , .ro_name_space                        ( "IBM,LPC................."          )
4459  , .ro_afu_version_major                 ( `AFU_VERSION_MAJOR                   )
4460  , .ro_afu_version_minor                 ( `AFU_VERSION_MINOR                   )
4461  , .ro_afuc_type                         (   2'b01                              ) // Type C1 issues commands to the host but does not cache host data
4462  , .ro_afum_type                         (   2'b01                              ) // Type M1 contains host mapped addresses (i.e. MMIO or memory)
4463  , .ro_profile                           (   8'h01                              ) // Device Interface Class
4464  , .ro_global_mmio_offset                (  {60'h0000_0000_0000_000, 1'b0}      ) // MMIO space starts at BAR offset 0 ([2:0]=000)
4465  , .ro_global_mmio_bar                   (   3'b000                             ) // MMIO space is contained in BAR0
4466  , .ro_global_mmio_size                  (  32'h0008_0000                       ) // LPC MMIO size is 1 MB, but Global MMIO section is 512 KB
4467  , .ro_per_pasid_mmio_offset             (  {60'h0000_0000_0008_000, 1'b0}      ) // PASID space start at BAR 0+512KB address ([2:0]=000)
4468  , .ro_per_pasid_mmio_bar                (   3'b000                             ) // PASID space is contained in BAR0
4469  , .ro_per_pasid_mmio_stride             (  32'h0001_0000                       ) // Stride is 64KB per PASID entry (Linux requirement)
4470 //, .ro_mem_size                         (   8'h14                              ) // Default is 1 MB (2^20, x14 = 20 decimal) - SAM disabled
4471  , .ro_mem_size                          (   8'h2A                              ) // Default is 4 TB (2^42, x2A = 42 decimal) - SAM enabled
4472  , .ro_naa_wwid                          ( 128'h0000_0000_0000_0000_0000_0000_0000_0000 ) // LPC has no WWID
4473    // Hardcoded 'AFU Index' number of this instance of descriptor table
4474  , .ro_afu_index                         ( ro_afu_index                         ) // Each AFU instance under a common Function needs a unique index number
4475    // Functional interface
4476  , .cfg_desc_afu_index                   ( cfg_desc_afu_index                   )
4477  , .cfg_desc_offset                      ( cfg_desc_offset                      )
4478  , .cfg_desc_cmd_valid                   ( cfg_desc_cmd_valid                   )
4479  , .desc_cfg_data                        ( desc_cfg_data                        )
4480  , .desc_cfg_data_valid                  ( desc_cfg_data_valid                  )
4481  , .desc_cfg_echo_cmd_valid              ( desc_cfg_echo_cmd_valid              )
4482    // Error indicator
4483  , .err_unimplemented_addr               ( desc_err_unimplemented_addr          )
4484 );
```

1.9.1 SAM reduced to 16 entries

Starting with version LPC 6.2, the Sparse Array Mapper was reduced from 32 to 16 entries, with each entry now representing a 64 KB block. This was done to achieve timing closure at 25.78125 Gbps on the OpenCAPI link. Experiments run when the LPC was being designed used early timing rules for UltraScale+ FPGA gates and wires, because that was the only data available at the time (that is, Vivado 2016.x). Production timing rules were added starting with Vivado 2017.1, which caused timing closure issues in the SAM block. Improvements to Vivado during 2017 improved timing somewhat, but by 2017.4 it was evident problems were going to persist. Therefore, experiments were re-run, including reducing the SAM size by half and going to a 2 cycle SAM lookup. Both showed timing closure can be achieved, but reducing the SAM was the less invasive option to implement.

The new SAM organization is shown in Figure 1-7. There are now 16 entries, with each one representing a 64 KB block of bulk memory.

Figure 1-7. SAM organization



1.10 Bulk memory

Bulk memory is where data FLITs are stored and retrieved. It is organized as N rows x 512 bits, so that a full 64-byte FLIT can be written or read each cycle.

In the FPGA, Block RAM (BRAM) primitives are used as memory. After several experiments, it was determined the block memory generator can most efficiently create BRAM blocks in the size 8192 x 512. Two of them are used to create a 16,384 x 512 memory, implementing a total of 1 MB of storage. If the implementation allows it, more BRAM blocks can be added to increase the amount of bulk memory in the LPC.

Bulk memory addressing is contiguous by row, starting from row 0 increasing to row 16,383.

These BRAMs hold the FLITs; however, there are two more BRAMs used to hold the ‘bad data indicator’ (BDI) bit that is associated with each FLIT. These BRAMs are 8192 x 1, and like FLIT storage two of them are used to make a 16,384 x 1 array. The BDI bit received during writes is stored in these side arrays and is provided along with the FLIT when it is read.

Other implementation notes:

- While the Xilinx devices also allow UltraRAMs to implement large memories, BRAMs were chosen instead because UltraRAMs are physically located next to the 25 Gbps PHY and it is desired to keep that area as quiet as possible to minimize noise that can propagate into the link.
- A single BRAM can be 1K rows x 36 bits when 2 ports are used (1R, 1W), or 512 rows x 72 bits when the 2 ports can be reduced to a single RW port.

- BRAMs can be internally cascaded to appear as a bigger RAM, but after chaining approximately four of them together timing problems can arise causing there to be a register stage inserted. This can make it difficult for surrounding logic to manage, especially when considering a pipelined design, so cascading was turned off in the LPC usage. Instead user defined logic around the BRAMs handles all MUX'ing of outputs.
- To meet the aggressive timing goals, address, controls, and data are registered just before going into the BRAM and data is registered immediately after the BRAM.
- Registers around the BRAM are implemented in external logic because if implemented inside the BRAM, the array enable signal must be held for multiple cycles. This can complicate the surrounding logic when pipelining is considered; therefore, the decision was made to put registers in explicitly around the BRAM to keep the enable signal active for only a single cycle. Vivado synthesis might pull registers into the BRAM block, but as long as the behavior stays the same this is not a problem.
- ECC and parity are not implemented on bulk memory contents. If this were a product, or if memory was implemented by external DRAM, then ECC would be required. But being used only as a reference design and lab exerciser, the added protection was not deemed necessary.

1.10.1 Simulating BRAMs

In simulation, the model created by Vivado (Xilinx tool set) can be used for event simulation (that is designer simulation using Cadence tools or the simulator in Vivado). However, this model is not compatible with cycle simulation, therefore a replacement behavior model was created for IBM's cycle simulators Mesa & Awan.

To use with Mesa/Awan simulation, the variable **BEHARY** must be defined during Portals compilation. The default is to invoke the Xilinx BRAM primitive as the memory model, because this is used by Vivado for synthesis, placement, and timing. Defining this variable replaces the Xilinx model with the cycle simulation compatible behavior. The following is an example of the proper declaration for *tvc* (run on an AIX machine).

```
AIX> tvc -C "-DBEHARY" -src lpc_device.v -dbout dbout -mixed -verilog -module
lpc_device -clean
```

The Mesa/Awan behavior array models reside in the same directory as the LPC source.

2. Commands and features

2.1 Supported commands and features

TL CAPP Commands (Host → AFU or Host → CFG)

- pr_rd_mem, pr_wr_mem — Partial memory read and write
- rd_mem, write_mem — Full cache line read and write
- write_mem.be — Write with byte enables
- amo_rd, amo_rw, amo_w — Atomic memory operations
- nop — Implied, although AFU does not see it because absorbed in TLX

TLX AP Responses (AFU → Host or CFG → Host)

- mem_rd_response, mem_wr_response
- mem_rd_fail, mem_wr_fail
- nop, return_tl_credits — Implied, although AFU does not see it because absorbed in TLX

TLX AP Commands (AFU → Host)

- intrp_req — Needed to pass along errors
- assign_actag — Needed for interrupts
- nop — Implied, although AFU does not see it because absorbed in TLX

TL CAPP Responses (Host → AFU)

- intrp_resp — Response for AFU issued interrupts
- intrp_rdy — Retry for suspended interrupts issued by AFU
- nop, return_tlx_credits — Implied, although AFU does not see it because absorbed in TLX

2.1.1 Features

This LPC design includes these features:

- Designed using little-endian format for all but the atomic operations. Big- and little-endian formats are supported only for the atomic operations. If atomic operations are enabled, the error check for err_incorrect_endianness described in the **Global MMIO space** section of this document must be disabled.
- Conformance to error scenarios described in section 9.1 ‘Error Events’ in the OpenCAPI TL specificiation and the **Error scenarios** section of this document
- **config_write** and **config_read** commands are supported by the LPC because it includes the CFG sub-system, even though these commands are not passed into the “AFU” itself.

2.2 Unsupported commands and features

The following commands are not supported by this LPC implementation.

TL CAPP Commands (Host → AFU)

These commands are not part of this version of the LPC.

- force_evict
- return_adr_tag
- wake_afu_thread
- write_meta
- xlate_done
- ‘Os’ field support for write_mem

TLX AP Responses (AFU → Host)

- mem_rd_response.ow, mem_rd_response.xw, read_response.xw
- wake_afu_resp
- intrp_req.d

TLX AP Commands (AFU → Host)

These commands are not part of this version of the LPC.

- adr_tag_release
- amo_rd.t, amo_rw.t, amo_w.t
- castout, castout.push
- mem_pa_flush
- upgrade_state, upgrade_state.t

The LPC does not issue commands to the host; therefore, these do not apply.

- dma_pr_w, dma_pr_w.t, dma_w, dma_w.be, dma_w.be.t, dma_w.t
- pr_rd_wnitc, pr_rd_wnitc.t, rd_wnitc, rd_wnitc.t
- read_exclusive, read_exclusive.t, read_shared, read_shared.t
- xlate_touch
- wake_host_thread

TL CAPP Responses (Host → AFU)

These commands are not part of this version of the LPC.

- touch_resp
- upgrade_resp
- cl_rd_resp, cl_rd_resp.ow
- mem_flush_done

The LPC does not issue commands to the host, therefore these do not apply.

- read_failed, read_response, read_response.ow, read_response.xw
- write_response, write_failed
- wake_host_resp

2.2.1 Features

This version of the LPC does not support:

- 32-byte and 8-byte datum transfers; only 64-byte FLITs are supported to carry data
- .s (synchronization) formats of commands
- segment ordering
- critical OW first on rd_mem

2.3 Special command fields

The OpenCAPI TL specification must be closely reviewed when implementing commands and responses. Different commands provide different fields, or parts of fields, with some values assumed or implied. This can be misleading to the AFU designer, especially because the TLX provides signals for all fields of all commands whether they are valid or not. The following is a summary of where the LPC gets values for all the fields needed in successful or failing responses.

First, identify which fields are needed in responses.

```
mem_wr_response   Opcode[7:0], CAPPTag[15:0], dL[1:0], dP[1:0]

mem_wr_fail       Opcode[7:0], CAPPTag[15:0], dL[1:0], dP[1:0], Resp_code[3:0]

mem_rd_response   Opcode[7:0], CAPPTag[15:0], dL[1:0], dP[1:0]

mem_rd_fail       Opcode[7:0], CAPPTag[15:0], dL[1:0], dP[1:0], Resp_code[3:0]
```

The amo_rd and amo_rw operations use mem_rd_response and mem_rd_fail; the amo_w operation uses mem_wr_response and mem_wr_fail. While LPC logic fills in the Opcode field based on the response type and Resp_code field based on the success or failure of the operation, the other fields need to be provided or assumed by the command as shown in Table 2-1. Note that atomic operations require the cmd_flag field and the E bit for endianness; see the OpenCAPI 4.0 Transaction Layer Specification for details.

Table 2-1. LPC fields

Command	Provided by Command[7]	Assumed Values
MMIO write (**pr_wr_mem**)	CAPPTag[15:0] PA[63:0], pL[2:0]	dL[1:0]=’01’ (64B FLIT) dP[1:0]=’00’ (no offset)
MMIO read (**pr_rd_mem**)	CAPPTag[15:0] PA[63:0], pL[2:0]	dL[1:0]=’01’ (64B FLIT) dP[1:0]=’00’ (no offset)

[7] Opcode[7:0] is not included in the list because all commands contain it.

Command	Provided by Command7	Assumed Values
Partial memory write (**pr_wr_mem**)	CAPPTag[15:0] PA[63:0], pL[2:0]	dL[1:0]='01' (64B FLIT) dP[1:0]='00' (no offset)
Partial memory read (**pr_rd_mem**)	CAPPTag[15:0] PA[63:0], pL[2:0]	dL[1:0]='01' (64B FLIT) dP[1:0]='00' (no offset)
write_mem	CAPPTag[15:0] PA[63:6] dL[1:0], Os	PA[5:0]='000000' dP[1:0]=Determined by AFU, it might break up response
rd_mem	CAPPTag[15:0] PA[63:5], dL[1:0], MAD[7:0]	PA[4:0]='00000' dP[1:0]=Determined by AFU, it can break up response
write_mem.be	CAPPTag[15:0] PA[63:6] Byte_enable[63:0]	PA[5:0]='000000' dL[1:0]='01' (64B FLIT) dP[1:0]='00' (no offset)
amo_rd	CAPPTag[15:0], PA[63:0] pL[2:0], E[0], cmd_flag[3:0]	dL[1:0]='01' (64B FLIT) dP[1:0]='00' (no offset)
amo_rw	CAPPTag[15:0], PA[63:0] pL[2:0], E[0], cmd_flag[3:0]	dL[1:0]='01' (64B FLIT) dP[1:0]='00' (no offset)
amo_w	CAPPTag[15:0], PA[63:0] pL[2:0], E[0], cmd_flag[3:0]	dL[1:0]='01' (64B FLIT) dP[1:0]='00' (no offset)

2.4 Byte alignment on data bus

The data bus is 64 bytes wide. When partial width operations occur, the bytes stay in their normal byte lanes and the lower address bits are used to select the byte lanes.

To help illustrate this, the lower 8 bytes of the data bus are shown. The same concept applies over the full data bus width.

'*' indicates the used bytes in the operation, 'pa' is the physical address bus, and the operation length in bytes is determined by the pL (partial length) field of the command.

Table 2-2. Byte alignment

Byte Lane	7	6	5	4	3	2	1	0
pa[2:0]	'111'	'110'	'101'	'100'	'011'	'010'	'001'	'000'
data[63:0]	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
8B op, pa='000'	*	*	*	*	*	*	*	*
4B op, pa='100'	*	*	*	*				
4B op, pa='000'					*	*	*	*
2B op, pa='110'	*	*						

Byte Lane	7	6	5	4	3	2	1	0
2B op, pa='100'			*	*				
2B op, pa='010'					*	*		
2B op, pa='000'							*	*
1B op, pa='111'	*							
1B op, pa='110'		*						
1B op, pa='101'			*					
1B op, pa='100'				*				
1B op, pa='011'					*			
1B op, pa='010'						*		
1B op, pa='001'							*	
1B op, pa='000'								*

For this example, all other combinations of length (pL) and pa[2:0] are illegal, as they violate 'natural alignment' (e.g., 2B op with pa[2:0]='001').

2.5 Interrupts

2.5.1 Causes of interrupts

Within the LPC, interrupts are generated for the following reasons (when properly enabled):

- When a host command results in a failed response
- When the LPC detects a potentially fatal error
- When triggered by software via an MMIO write (for test purposes to inject an interrupt)

Per the OpenCAPI TL specification, interrupts can also be generated if a 'Bad Data Indicator' (BDI) is received on write data to bulk memory or if present on read response data from bulk memory to the host. However, the spec also allows for the BDI bit to simply be carried with the data with no interrupt generated.

To minimize latency and conserve resources, the LPC does not buffer write data before saving it into bulk memory nor does it buffer data when reading it from bulk memory. Thus, it only allows BDI to affect the response code (and interrupts) when data is guaranteed to be a single FLIT.

As such, the LPC behaves in this fashion:

- On MMIO writes, if BDI is present on the single FLIT, a bad response code is generated. The failing response code in turn generates an interrupt. The BDI bit is checked to make sure bad data does not overwrite an MMIO register contents, potentially causing unexpected behavior on subsequent operations.

- On MMIO reads, the BDI is always set to 0. This is because the MMIO registers are treated as uncorrupted, which is reasonable because writing bad data to the registers is prevented. Soft errors can change a register value, but in FPGA implementations this effect is ignored on general logic.
- On bulk memory writes, BDI does not factor into the response code; therefore, no interrupt is generated. Write data and received BDI are saved in bulk memory even if BDI is set on a FLIT. This ensures that the data will be marked as corrupted if the host reads it later. The problem with BDI affecting the response code is it would require a data buffer to temporarily hold and discard all data if the BDI happens in any FLIT. To minimize latency and implementation resources, FLITs are written directly into memory as they arrive. This means a late arriving BDI cannot 'un-write' the memory locations that are already written.
- On bulk memory reads, the response code is determined before data is fetched from memory because it must be sent with the first data FLIT. Because data passes directly from bulk memory to the TLX, BDI cannot affect the response code unless a buffer is introduced. However, this is avoided to minimize latency and resources. Because a bad response code is not generated, no interrupt is triggered on BDI.

The LPC only reports one interrupt at a time but does buffer up a small number of errors that can generate interrupts. This allows a limited number of error conditions that occur before interrupts are enabled or while an interrupt is ‘in flight’ to be saved for later presentation to the host.

2.5.2 Generating an interrupt

Before the first interrupt can be issued, an acTag must be registered with the host. The acTag is formed from two sources, the Bus/Device/Function information assigned to the AFU and a PASID stored in the MMIO space. The AFU can choose the value of acTag, but it must be between the ‘acTag Base’ and ‘acTag Length Enabled’ boundaries set in the configuration space (see AFU Control DVSEC). To work with the POWER9 host, these values must be set so that the tag range is between 0 and 63. The LPC chooses the lowest value of the tag range as the acTag.

When an interrupt must be sent, the LPC checks whether an **assign_actag** instruction must be issued to the host prior to the interrupt request. It is an OpenCAPI TL requirement that operations using the acTag field must register the acTag value with the host prior to executing the operation. This is because the acTag is used as a short hand for the larger number of bits representing BDF and PASID values that will not fit in most TL commands. If an **assign_actag** was not already sent, it is issued before the interrupt. Once the acTag is registered, any number of interrupts (or other AFU initiated operations) can be issued until either the AFU is reset or the Bus/Device/Function/PASID values are changed.

Other fields of the interrupt command (**intrp_req**) also come from MMIO registers. The LPC expects these to be set by the host to the desired values just after reset. The fields include a 64-bit effective address (passed in the Obj_handle field), a 4-bit cmd_flag[8], a 4-bit stream_id[9], and 16-bit AFUTag[10]. While the LPC can use any value it wanted for stream_id or AFUTag, allowing test software to change them adds more flexibility. Other fields also need to be set up to enable interrupts. Refer to section 'Configure interrupt generation’ on page 51 for more details.

[8] cmd_flag tells the host how to interpret the Obj_handle field, because the AFU and its driver software can pass different types of information in that field. For the LPC AFU however, Obj_handle only carries the effective address of the interrupt so cmd_flag only needs one value. For a POWER9 host, ‘0000’ means treat the Obj_handle as an EA.
[9] stream_id is a way to keep interrupts separate on the PowerBus, which can be useful if the AFU had multiple engines generating interrupts. Since the LPC AFU only processes one interrupt at a time, only one value is needed.
[10] AFUTag is used to pair interrupt requests and responses. It has no other use except to be a unique identifier for multiple outstanding AFU to host operations.

2.5.3 Responses to interrupts

After the AFU sends **intrp_req** to the host, the host responds with an **intrp_resp** that has a matching **AFUTag**. The response code field determines what happens next.

- resp_code = ‘0000’ Interrupt Request accepted

  The interrupt is complete. The LPC can process new interrupts as needed.

- resp_code = ‘0010’ Retry Request

  The host is momentarily busy, therefore asks the AFU to resend the interrupt again. The LPC reissues the same **intrp_req** command after a ‘long delay’, as specified by the OpenCAPI TL spec. The long delay value is provided by the configuration space.

- resp_code = ‘0100’ Interrupt Resources Pending

  The host needs software intervention to process the interrupt. Because this can take some time, the host will notify the AFU when it can resend the interrupt. When the LPC sees an **intrp_rdy** from the host (AFUTag must match), it reissues the same **intrp_req** command. Note that **intrp_rdy** arrives as a Host –> AFU response, not a command.

- resp_code = ‘1001’ Unsupported Operand Length, ‘1011’ Bad object handle, or ‘1110’ Failed

  An unrecoverable error occurred. The LPC logs that a fatal error happened and hangs the interrupt state machine.

2.5.4 Interrupt states

The various combinations of **intrp_resp** and **intrp_rdy** and their response codes are implemented using this state machine in the LPC (SM_IRQ).

IDLE:

When interrupt is triggered, move to GET_LOCK.

GET_LOCK:

When the AFU→Host interface is available and the TLX can receive a new AFU command, lock the AFU→Host interface and issue **assign_actag** if the BDF or PASID changed because the last time it was issued. Move to SEND_INTRP.

SEND_INTRP:

When the TLX can receive a new AFU command, issue **intrp_req** and move to WAIT_FOR_RESP.

WAIT_FOR_RESP:

When intrp_resp is received, make one of the following choices.

a) If received AFUtag <> AFUtag sent in **intrp_req**, record error & move to ERROR
b) If resp_code=’0000’, move to CLEAN_UP (interrupt was accepted)
c) If resp_code=’0010’, move to RETRY_AFTER_DELAY (reissue intrp_req after delay)
d) If resp_code=’0100’, move to WAIT4_INTRP_RDY (wait for intrp_rdy from host)

```
      e) If none of the above are taken, record error & move to ERROR

RETRY_AFTER_DELAY:

      Wait the number of cycles programmed as a "long delay", then move to
      SEND_INTRP.

WAIT4_INTRP_RDY:

      When intrp_rdy is received, make one of these choices.

      a) If received AFUtag <> AFUtag sent in intrp_req, record error & move to ERROR
      b) If resp_code='0000', move to SEND_INTRP (host is ready to receive intrp_req)
      c) If resp_code='0010', move to RETRY_AFTER_DELAY (reissue intrp_req after delay)
      d) If none of the above are taken, record error & move to ERROR

CLEAN_UP:

      Release lock on AFU→Host interface and indicate interrupt was reported
      successfully. Move to CLEAN_UP2.

CLEAN_UP2:

      Wait 1 cycle for clearing of lock to take effect. Return to IDLE.

ERROR:

      Freeze the interrupt state machine by staying in this state. If MMIO reads
      are still working, software can retrieve error vector(s) to gain insight
      into what went wrong. An AFU reset is needed to return to normal operation.
```

In support of the states above, the LPC triggers sending an interrupt when an entry is written into the error vector array. Because errors can occur while an interrupt is in progress, it uses a counter to accumulate the number of triggers detected. Therefore, the host can see interrupts issued back to back, although not overlapping, if the counter is non-zero after an interrupt request has been completed.

2.5.5 Additional interrupt information

Each AFU has the choice of determining what, if any, information to save with each interrupt to help software determine and resolve its cause. The LPC saves a 128-bit vector of error diagnosis information with each interrupt. The 128 bits are readable by software from two MMIO registers: ERR_INFO_HI[63:0] and ERR_INFO_LO[63:0]. It is expected that software will read ERR_INFO_HI first, then read ERR_INFO_LO. This order is important because reading ERR_INFO_LO replaces the present error information with the next one in line if multiple interrupts are pending. In other words, the present error information is discarded from the saved error array when ERR_INFO_LO is read.

The LPC intends the flow of information to go as follows:

- Error is detected in the logic.
- Diagnosis information is collected and logged in the saved error array.
- An interrupt is generated based on a new entry written into the error array.
- The host receives and responds to the interrupt, per the OpenCAPI TL procedure.
- Interrupt handler software reads the MMIO registers to obtain the diagnosis information.

- The act of reading ERR_INFO_LO clears the diagnosis information from the error array.
- If another error has been saved in the array, repeat the process.

Note that the intention is for one interrupt to be paired with one read of error diagnosis information. If the interrupt handler chooses to “read ahead”, meaning it reads ERR_INFO_HI/LO multiple times to retrieve all errors when processing a single interrupt, the read and write pointers in the saved error array can get out of sync.

Software can read ahead if it ensures ERR_INFO_HI/LO is only read once per interrupt. If software uses the value of ‘ery_data_valid’ in the STATUS register to confirm if ERR_INFO_HI/LO contains valid information prior to reading it, then read ahead can be performed. Because the LPC will still generate one interrupt for every error information vector, software must ignore interrupts associated with any errors that are read ahead. However, if software reads a non-valid entry from ERR_INFO_LO, the read pointer of the error array will move beyond the write pointer and subsequent errors can be missed. Therefore, software must use read ahead carefully and only read entries marked as ‘valid’ from the ERR_INFO_LO register.

The LPC implements an error vector array (ERY), allowing up to 8 error vectors to be recorded while an interrupt request is in progress or before interrupts are enabled. Error vectors can be added to the array by issuance of a bad response code to an OpenCAPI TL command, by detection of an internal error (ERROR register), or by a software generated interrupt (SFW_INTRP_HI, SFW_INTRP_LO registers).

Once enabled, an interrupt is sent if anything is present in the error vector array. If multiple errors have been recorded, the next pending interrupt is sent immediately after a successful interrupt response code is received for the current interrupt. This can occur before software has a chance to retrieve the error diagnosis information for the first interrupt. To maintain the proper pairing of interrupts and diagnosis information, software must understand that the generation of interrupts operates independently from the removal of error diagnosis information from the saved error array even though there is a one for one relationship between them.

2.5.6 Decoding error information

The information in ERR_INFO_HI and ERR_INFO_LO is decoded differently based on the cause of the error. The upper 16 bits determine the error cause, which subsequently determine how to decode the remaining bits. Each source has a unique bit in the vector. If multiple source bits are set, simultaneous errors occurred. In this case, because the rest of the information is the OR of the source vectors, extra care is needed during interpretation.

For simplicity, ‘ery_src[127:0]’ referenced in the following description is formed by the concatenation of ERR_INFO_HI[63:0] and ERR_INFO_LO[63:0] (that is, ery_src[127] = ERR_INFO_HI[63], ery_src[0] = ERR_INFO_LO[0]).

Note: [127:112] = 0x2000 through 0x0200, and 0x0002 are reserved error sources, unused at this time.

2.5.6.1 Source: Internal (possibly fatal) error

```
[127:112] 0x8000 = Error source identifier

[111: 64] Reserved

[ 63:  0] ERROR[63:0] MMIO register at time of first error detection
```

2.5.6.2 Source: Software generated interrupt

```
[127:112] 0x4000 = Error source identifier
```

```
[111:  0] User defined value in SFW_INTRP_HI[47:0] || SFW_INTRP_LO[63:0]
```

2.5.6.3 Source: Pipelined writes/reads formed bad response code

```
[127:112] 0x0100 = Error source identifier

[111:108] Response Code field from Response

[    107] Reserved (‘0’)

[106: 93] Reserved (all ‘0’s)

[ 92: 88] Pipe Type (‘10000’=rd_mem, ‘01000’=pr_rd_mem, ‘00100’=write_mem,

                     ‘00010’=pr_wr_mem, ‘00001’=write_mem.be)

[     87] Reserved (‘0’)

[ 86: 84] PL[2:0] from host command

[ 83: 82] Reserved (‘00’)

[ 81: 80] DL[1:0] from host command

[ 79: 64] CAPPTAG[15:0] from host command

[ 63:  0] PA[63:0] from host command
```

2.5.6.4 Source: Received bad interrupt ready (intrp_rdy) - AFUTag mismatch or bad response code

```
[127:112] 0x0080 = Error source identifier

[111:108] Response Code field from intrp_rdy

[107: 32] Reserved (all ‘0’s)

[ 31: 16] AFUTag received in intrp_rdy

[ 15:  0] AFUTag sent in interrupt request
```

2.5.6.5 Source: Received bad interrupt response (intrp_resp) - AFUTag mismatch or bad response code

```
[127:112] 0x0040 = Error source identifier

[111:108] Response Code field from intrp_resp

[107: 32] Reserved (all ‘0’s)

[ 31: 16] AFUTag received in intrp_resp

[ 15:  0] AFUTag sent in interrupt request
```

2.5.6.6 Source: State machine formed bad response code - memory and MMIO

```
[127:112] 0x0020 = Error source identifier (Full/Partial Read from Memory)
          0x0010 = Error source identifier (Full/Partial/BE Write to Memory)
          0x0008 = Error source identifier (MMIO Read)
          0x0004 = Error source identifier (MMIO Write)
[111:108] Response Code field from Response
[    107] Bad Data Indicator (OR of FLIT BDIs when write, '0' when read)
[106: 87] Reserved (all '0's)
[ 86: 84] PL[2:0] from host command [Set to '111' on full read/write]
[ 83: 82] Reserved ('00')
[ 81: 80] DL[1:0] from host command [Set to '01' on partial read/write & BE]
[ 79: 64] CAPPTAG[15:0] from host command
[ 63:  0] PA[63:0] from host command
```

2.5.6.7 Source: State machine formed bad response code - configuration

```
[127:112] 0xh0001 = Error source identifier (Configuration state machine)
[111:108] Response Code field from Response
[    107] Bad Data Indicator (FLIT BDI when write, '0' when read)
[106:104] Opcode: '100'=config_write, '010'=config_read, '001'=invalid opcode
[103: 99] Device number received in command
[ 98: 96] Function number received in command
[     95] Device or Function number is not implemented
[     94] Command contained unrecognized opcode
[     93] T bit is '1'
[     92] Command data was marked bad (BDI='1') (config_write)
[     91] PL[2:0] is invalid
[     90] Address and size create an alignment error (config_write)
[     89] Target address is not implemented in selected Function
```

```
[     88] Read Data Valid (config_read), '0' (config_write)

[     87] T bit from host command

[ 86: 84] PL[2:0] from host command

[ 83: 82] TLX PORT NUM[1:0]

[ 81: 80] '01' (reserved, but set to one FLIT in size)

[ 79: 64] CAPPTAG[15:0] from host command

[ 63: 32] 0x0000_0000 (reserved, upper 32 bits of PA don't get to CFG)

[ 31:  0] PA[31:0] from host command
```

Note: The config_write/config_read/invalid_opcode error information is captured and passed along from the configuration sequencer block (cfg_seq.v).

3. Non-interrupt AFU to host traffic

Other than interrupts, the LPC initiates no other traffic to the host. However, place holder signals have been added to the source code to support special test modes, which might arise later during hardware bring up. One set of test mode place holder signals using the suffix *_tm1 exists in `lpc_afu.v`.

4. Error scenarios

Table 4-1 describes the LPC behavior in error situations.

Note: Per the OpenCAPI TL architecture, there is no data returned on a fail response. This means it is up to the host to interpret and convert the fail response to the desired data format. More specifically in the case of POWER9, section '*5.1.3 AFU MMIO BAR' of the OPPA spec* states, "If the AFU space is fragmented the AFU is expected to return all 1's on reads from the host or drop writes from the host to unimplemented addresses." Because the AFU itself cannot return data, the unit attached to the AFU (for example NPU) must convert the fail response code into all 1's on the PowerBus.

Table 4-1. LPC behavior in error situations

Error Scenario	**AFU Response**	
Unrecognized command opcode	Record receipt in error register, hang and issue interrupt.	
Full memory write (write_mem) or full memory read (rd_mem) to MMIO space	Treat as 'unrecognized command opcode' because only partial writes and reads are appropriate for MMIO register access	
Error Scenario	**MMIO Write**	**MMIO Read**
Access to 'reserved' address	Write has no effect. mem_wr_response	Data of all '0's is returned. mem_rd_response rdata = all '0's rdata_bad = '0' rdata_valid = '1'
Access to un-implemented address in BAR range	Write has no effect. mem_wr_fail resp_code = 0xE	No data is returned. mem_rd_fail resp_code = 0xE
Operation Length is not supported (that is PL field is not 1,2,4,8 bytes. DL is not used on pr_wr_mem and pr_rd_mem, so that there is no check on it for a reserved value.)	1 FLIT is assumed to come with the command. Extract it from the TLX but suppress the write therefore it has no effect. mem_wr_fail resp_code = 0x9	No data is returned. mem_rd_fail resp_code = 0x9
Address is not aligned properly with respect to the size	Write has no effect. mem_wr_fail resp_code = 0xB	Address is assumed to end in '000' therefore 8-bytes of data are returned mem_rd_response rdata = register contents rdata_bad = '0' rdata_valid = '1'
Access to register outside BAR range	Treated as operation to normal memory.	
Write data is marked bad (bdi='1')	Write has no effect. mem_wr_fail resp_code = 0x8	Not applicable
Error Scenario	**Memory Write**	**Memory Read**
Operation Length is not supported, but not reserved (DL<>'00')	DL field is assumed to correctly indicate the number of FLITs. Extract them from the TLX but suppress the write. Write has no effect. mem_wr_fail resp_code = 0x9	No data is returned. mem_rd_fail resp_code = 0x9

Operation Length is reserved (DL='00')	Record receipt in error register, hang or issue interrupt.	No data is returned. mem_rd_fail resp_code = 0x9
Address is not aligned properly with respect to the size	Write has no effect. mem_wr_fail resp_code = 0xB	Address is assumed to end in '000000', data at resultant location is returned mem_rd_response rdata = register contents rdata_bad = '0' rdata_valid = '1'
Access outside implemented range	Write has no effect. mem_wr_fail resp_code = 0xE	No data is returned. mem_rd_fail resp_code = 0xE rdata = n/a rdata_bad = n/a rdata_valid = n/a
Write data is marked bad (bdi='1')	Save bad indicator along with data, old data is overwritten	Return data as normal. Return saved bad indicator on 'bdi' output to TLX.
First access to a new SAM mapped block of memory	Write is performed normally, a new SAM table entry is allocated to the new physical address.	No data is returned. mem_rd_fail resp_code = 0xE If the first access to a new SAM block is a read, rdata and rdata_valid will be 'X's. Make this a fail code to prevent bad action(s) being taken on the data downstream by the host. (Note: This action can be changed via an MMIO bit.)
Error Scenario	**LPC Specific Response**	
Access to un-implemented address in LPC memory space[11]	Record receipt in error register, hang	

Possible resp_code's:

- mem_wr_fail = 0x8 (data error), 0x9 (unsupported length), 0xB (bad address), 0xE (general fail)
- mem_rd_fail = 0x8 (data error), 0x9 (unsupported length), 0xB (bad address), 0xE (general fail)

[11] This situation arises in the LPC reference design because of the SAM feature to allow simulation and hardware testing to have access to an address range much larger than the physical memory implemented in the FPGA. From an AFU architectural viewpoint, this might be illegal, so there is no defined way for the AFU driver to react. Therefore, if the LPC sparse memory table runs out of mapping entries, the LPC will log a failure bit and hang.

5. Initialization and configuration

5.1 Base initialization

The following steps are used to initialize and use the LPC.

1. Apply ‘reset_n’ (active low) for a minimum of 10 clocks.

This establishes proper initial values in all the state machines and control signals. It must be held active for several clock cycles to propagate a non-valid value through ‘valid’ registers used at each stage of the pipeline.

2. Establish credits as needed depending on the model contents.

There are 4 sets of credits to initialize that the AFU sees, although these can happen naturally if the model contains both AFU and TLX.

- AFU to TLX for commands, so TLX can issue commands to AFU
`afu_tlx_cmd_initial_credit[6:0],afu_tlx_cmd_credit` (commands)

- TLX to AFU for responses, so AFU can issue responses to TLX

  `tlx_afu_resp_initial_credit[3:0],tlx_afu_resp_credit` (responses)

  `tlx_afu_resp_data_initial_credit[5:0],tlx_afu_resp_data_credit` (response data)

- TLX to AFU for commands, so AFU can issue commands to TLX

  `tlx_afu_cmd_initial_credit[3:0],tlx_afu_cmd_credit (commands)`

  `tlx_afu_cmd_data_initial_credit[5:0],tlx_afu_cmd_data_credit (command data)`

- AFU to TLX for responses, so TLX can issue responses to AFU

  `afu_tlx_resp_initial_credit[6:0],afu_tlx_resp_credit` (responses)

3. TLX indicates it is ready to interact with AFU by setting ‘tlx_afu_ready’ to ‘1’.
4. Use ‘config_write’ to establish which templates and rates are used.

Enable templates 3, 1, and 0 for maximum throughput. Template 2 has a maximum throughput that is slightly less. (Function 0, OTL_BASE = 0x200).

- Enable transmit templates 3,2,1,0 config_write (addr 0x200+0x24,[3:0])= ‘1111’
- Enable maximum transmit rates config_write (addr 0x200+0x6C,[15:0])= 0x0100

5. Use ‘config_write’ to set MMIO BAR0, to identify MMIO space versus bulk memory (Function 1, CSH base = 0x000).

- Upper 4B MMIO Base Address config_write (addr 0x14) = MMIO base[63:32]
- Lower 4B MMIO Base Address config_write (addr 0x10) = MMIO base[31:0]

6. Read back the MMIO BAR0 values to ensure they are established within the LPC.

Because MMIO and bulk memory share the same commands (**pr_wr_mem**, **pr_rd_mem**), it is necessary for the MMIO BAR0 value to be firmly established before these commands arrive at the TLX→ AFU interface. If the BAR0 value is not established in time, the command can be interpreted incorrectly causing an intended MMIO operation to target bulk memory instead (or vice versa).

- Upper 4B MMIO Base Address config_read (addr 0x14) = MMIO base[63:32]
- Lower 4B MMIO Base Address config_read (addr 0x10) = MMIO base[31:0]

7. Enable the AFU to use BAR0 to distinguish MMIO operations (Function 1, CSH base = 0x000).

- Memory Space = 1 config_write (addr 0x04, [1]) = '1'

8. Read back 'Memory Space' to ensure it is set (Function 1, CSH base = 0x000).

- Read back 'Memory Space' config_read (addr 0x04, [1]) = '1'

5.2 Configure interrupt generation

1. Use 'config_write' to setup interrupt fields in configuration space (Function 1, OCTRL_BASE = 0x500)

- PASID Length Enabled = 0 config_write (addr 0x500+0x10, [12:8]) = '00000'
- PASID Base = 0[12] config_write (addr 0x500+0x14, [19:0]) = 0x00000
- acTag Length Enabled = 1 config_write (addr 0x500+0x18, [27:16]) = 0x001
- acTag Base = 0 config_write (addr 0x500+0x1C, [11:0]) = 0x000

2. Use 'pr_wr_mem' to set up interrupt fields in 'Per PASID' space

- Interrupt EA = <driver choice> pr_wr_mem (addr BAR0+0x80000, [63:0]) = <ea>
- Interrupt PASID = <driver choice> pr_wr_mem (addr BAR0+0x80018,[19:0]) = <pasid>
- Interrupt STREAM_ID = <driver choice> pr_wr_mem (addr BAR0+0x80018,[47:44]) =<stream>
- Interrupt AFUTag = <driver choice> pr_wr_mem (addr BAR0+0x80018,[63:48])=<afutag>

3. Read back last MMIO write to flush the above commands through the AFU before enabling interrupts in the configuration sub-system.

- Read back last MMIO write pr_rd_mem(addr BAR0+0x80018) = afutag,stream,pasid

4. Use 'config_write' to enable the AFU to initiate commands to the host (Function 1, OCTRL_BASE = 0x500).

- Enable AFU = 1 config_write (addr 0x500+0x0C, [24]) = '1'

5. Read back 'Enable AFU' to flush the above commands through the configuration sub-system before issuing traffic.

- Read back 'Enable AFU' config_read (addr 0x500+0x0C, [24]) = '1'

6. When software is ready, use 'pr_wr_mem' to enable general error interrupt to be issued by the AFU.

- Interrupt Vector Mask = 0 pr_wr_mem (addr BAR0+0x80008, [32]) = '0'

7. Read back last MMIO write to flush the above commands through the AFU.

- Read back last MMIO write pr_rd_mem(addr BAR0+0x80008, [32]) = '0'

[12] The 'PASID Base' must be less than or equal to the PASID value programmed in MMIO space. By setting the base to 0 though, any value programmed in the MMIO area will be allowed.

5.3 Optional settings

The following settings are independent of each other; they can be used in any combination. Pipelining is not supported for atomic operations.

1. Disable SAM

Sparse Array Memory (SAM) mapping is enabled by default. To turn it off, set the disable bit to '1'.

- sam_disable = 1 pr_wr_mem (addr BAR0+0x00000, [0]) = '1'

2. Ignore SAM 'no match' condition on memory read

When SAM is enabled, by default an error is flagged if an operation reads a location not found in the SAM. This is because the contents of the memory have not been initialized, the read will return unknown data. In this case, a bad response code is returned, and an internal error is raised.

However, in random simulation where the data is not checked, it can be desirable to let the read complete successfully and not flag an error. To enable this mode:

- ignore_nomatch_on_read = 1 pr_wr_mem (addr BAR0+0x00000, [2]) = '1'

3. Enable pipelining of write and read commands to bulk memory

By default, operations execute one at a time. This allows for simpler debug but means the bandwidth of the LPC is restricted. The ability to execute write and read commands to bulk memory exists but must be turned on. Note that write and read commands to configuration and MMIO space continue to execute one at a time.

- enable_pipeline = 1 pr_wr_mem (addr BAR0+0x00000, [4]) = '1'

4. Read back the CONTROL register to flush above commands through the command FIFO before issuing traffic

- Read back MMIO CONTROL register pr_rd_mem (addr BAR0+0x00000)

5.4 Shortcuts for simulation

To save time in each test, simulation can skip these above steps and set up the same conditions by using forces instead. In this case, the output of the configuration and MMIO registers can be forced to achieve similar effects. Some of the steps above can be replaced with the ‘force’s listed.

The following references assume ‘<hier>’ represents the upper qualifier string that gets down to the LPC AFU. A further assumption is ‘LPC’ is the instance name of the top level LPC file, `lpc_device.v` . Adjust the names accordingly for the specific model being simulated.

- Enable transmit templates and maximum rates

  force <hier>.LPC.cfg0_tlx_xmit_tmpl_config_0 = ‘1’

  force <hier>.LPC.cfg0_tlx_xmit_tmpl_config_1 = ‘1’

  force <hier>.LPC.cfg0_tlx_xmit_tmpl_config_2 = ‘1’

  force <hier>.LPC.cfg0_tlx_xmit_tmpl_config_3 = ‘1’

  force <hier>.LPC. cfg0_tlx_xmit_rate_config_0[3:0] = 0x0

  force <hier>.LPC. cfg0_tlx_xmit_rate_config_1[3:0] = 0x0

  force <hier>.LPC. cfg0_tlx_xmit_rate_config_2[3:0] = 0x1

  force <hier>.LPC. cfg0_tlx_xmit_rate_config_3[3:0] = 0x0

- Set MMIO BAR0

  force <hier>.LPC.FUNC1.cfg_f1_csh_mmio_bar0[63:0] = <desired MMIO Base Address>

- Set Memory Space

  force <hier>.LPC.FUNC1.cfg_f1_csh_memory_space = ‘1’

- Set values in configuration space related to interrupt generation

  force <hier>.LPC.FUNC1.cfg_f1_octrl00_pasid_length_enabled[4:0] = ‘00000’

  force <hier>.LPC.FUNC1.cfg_f1_octrl00_pasid_base[19:0] = 0x000

  force <hier>.LPC.FUNC1.cfg_f1_octrl00_afu_actag_len_enab[11:0] = 0x001

  force <hier>.LPC.FUNC1.cfg_f1_octrl00_afu_actag_base[11:0] = 0x000

- Set values used in interrupt request

  force <hier>.LPC.FUNC1.AFU00.mmio_out_intrp_ea[63:0] = <value>

  force <hier>.LPC.FUNC1.AFU00.mmio_out_intrp_vec_mask = ‘0’

  force <hier>.LPC.FUNC1.AFU00.mmio_out_intrp_pasid[19:0] = <value>

  force <hier>.LPC.FUNC1.AFU00.mmio_out_intrp_stream_id[3:0] = <value>

force <hier>.LPC.FUNC1.AFU00.mmio_out_intrp_afutag[15:0] = <value>

- Enable AFU

  force <hier>.LPC.FUNC1.cfg_f1_octrl00_enable_afu = ‘1’

- Disable SAM

  force <hier>.LPC.FUNC1.AFU00.mmio_out_sam_disable = ‘1’

- Ignore ‘no match’ on read

  force <hier>. LPC.FUNC1.AFU00.mmio_out_ignore_nomatch_on_read = ‘1’

- Enable pipelining

  force <hier>. LPC.FUNC1.AFU00.mmio_out_enable_pipeline = ‘1’

In addition, a useful signal to display and monitor during simulation is the vector of internal errors. Individual errors can pulse or be set and held, so that it is useful to monitor or display this signal over all cycles of the simulation.

<hier>. LPC.FUNC1.AFU00.mmio_in_captured_errors[63:0]

6. Other information

6.1 Credit re-synchronization

The configuration sub-system bit ‘AFU Unique[0]’ (AFU00 Control DVSEC, offset 0x0C, bit [28]) is designated as a control signal to re-synchronize credits. To have the TLX and LPC discard and re-exchange command, response, and cmd/resp data credits after the AFU has been re-configured or reset while the TLX remains active, software should write this bit to ‘1’ and then write it to ‘0’. When equal to ‘1’, credit counters are cleared. When the falling (‘1’ → ‘0’) edge is detected, LPC credit counters sample the initial credit value and start watching for credits from the TLX. Simultaneously the falling edge instructs LPC credit issuers to resend all buffer credits to the TLX, after a short delay to account for propagation delay differences in the distribution of the ‘resync_credits’ signal from the configuration register to LPC and TLX.

7. LPC organization

The top-level file for the LPC design is 'lpc_device.v'.

It instantiates several other files, each representing a major or reusable function. Those in blue are part of the configuration sub-system.

```
                                                (instance names)
lpc_device.v          (recommended instance name) LPC
+-> cfg_cmdfifo.v                                 CFG0_CFF
+-> cfg_seq.v                                     CFG_SEQ
+-> cfg_respfifo.v                                CFG0_RFF
+-> cfg_func0.v, cfg_func0_init.v                 CFG_F0
+-> cfg_fence.v                                   FENCE
+-> lpc_func.v                                    FUNC1
    +-> cfg_func.v, cfg_func_init.v               CFG_F1
    +-> lpc_afu.v                                 AFU00
        +-> cfg_descriptor.v                      DESC
        +-> lpc_tlx_afu_credit_mgr.v              TACC, TARC, TACDC,
                                                    TARDC, RFF_CREDITS
        +-> lpc_cmdfifo.v                         CFF
        +-> lpc_respfifo.v                        RFF
        +-> lpc_sparse.v                          SAM
        +-> lpc_mmio_regs.v                       MMIO
        +-> lpx_errary.v                          ERY
        +-> lpc_bulk_mem.v                        MEM
            For Mesa/Awan simulation:
            +-> bram_8192x512_noOutReg_beh.v        BRAM_00, BRAM_01
            +-> bram_8192x1_noOutReg_beh.v          BRAM_B00, BRAM_B01
            For Vivado processing:
            +-> bram_native_1P_noOutReg_8192x512_A  BRAM_00, BRAM_01
```

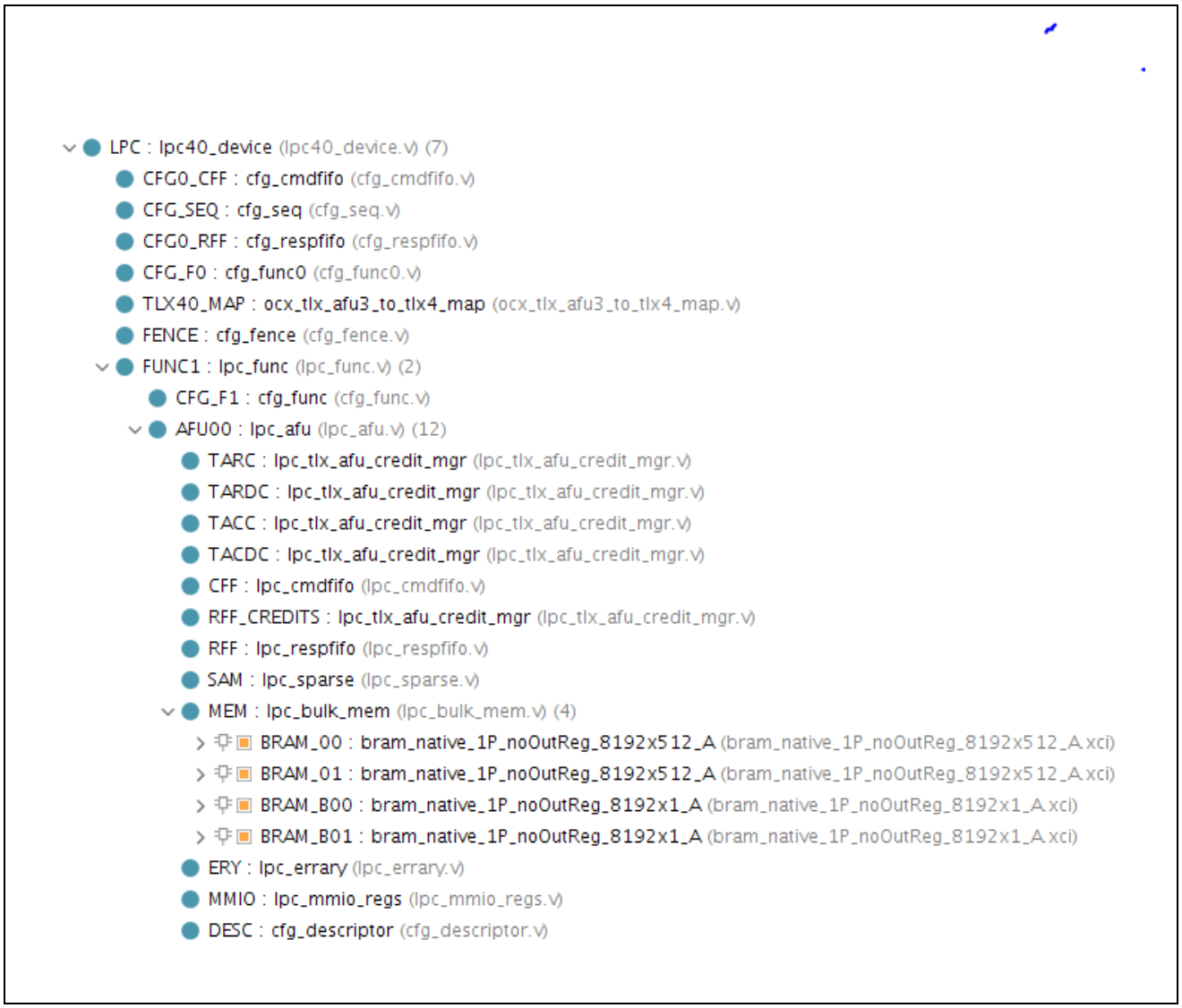
```
+-> bram_native_1P_noOutReg_8192x1_A     BRAM_B00, BRAM_B01
```

An extra file is included, to be connected as a peer to 'lpc_device.v'. This file contains sample VPD information and is intended for simulation only. In an actual FPGA, it is replaced by an I²C controller attached to an on-board FLASH or EEPROM that contains the Vital Product Data.

`vpd_stub.v` VPD

The hierarchy displayed in processing tools should match. Figure 7-1 is a screen capture of the source files from Vivado.

Figure 7-1. Source files from Vivado



8. Designer notes

The following section contains notes for the designer that can be useful when completing the LPC implementation. Nothing in here should be considered as committed features or functions of the LPC reference design. However, they are included to give the reader some sense of what can be possible or forthcoming.

8.1 Host read/write ordering

To improve performance, the LPC can allow host reads (loads) to be processed before host writes (stores); as long as queued loads and stores do not share the same address space and the store buffer is not full.

If a load arrives that shares the same address as a queued store operation, it should not be processed until the identified store operation is complete. Operations within each queue (that is, load queue or store queue) should be processed in the order they are received.

If the store buffer becomes full, the next one or more operations to memory should be stores to free up space for incoming store commands, as they can potentially block read commands that can be serviced.

Note: While allowed, this is not implemented as a high performing pipeline is used instead. However, it might become useful when accessing external DRAM instead of BRAMs inside the FPGA as bulk memory.

8.2 ‘TODO’ comments in source

Search for the string ‘TODO’ in the Verilog source to find more designer notes, ideas, and remaining things to implement.

8.3 Special mode: AFU to host bulk memory write

Someone doing a specific performance measurement asked for the ability for an AFU to initiate burst writes of data to the host so that bandwidth and latency of the OpenCAPI link can be measured. While this is really a DMA function that an AFU like ‘memcopy’ can do, it might be possible for the LPC to support this in a special test mode. The LPC would not support the ability to be programmed to initiate DMA operations in the same fashion as a real AFU but using MMIO fields it could be programmed with constant values to use for address, capptag, and so forth, that the host would recognize and support. Then it could read out of bulk memory to stream data to the host (that is repetitively send the 256-bytes at bulk memory address 0 or maybe a range from address 0 to N). Likewise, a stream of read data could be requested and either ignored or stuffed directly into a fixed region of bulk memory for later checking by the host. Performance counters (that is bytes per second, cycles waiting for TLX credits, and so forth.) would need to be included because performance measurement is the likely reason behind implementing this feature. A restriction could be imposed to say no other operations can be going on inside the LPC while this is engaged (except maybe accept MMIO writes to turn off the data stream). It is suggested to do whatever is easiest for LPC implementation and verification while being able to measure the data streaming performance.

8.4 Legal adjustments to TLX interface

As a test vehicle, it can be useful for the LPC to ‘stretch the boundaries’ of what is legally accepted as TL traffic between the TLX and AFU, and host and AFU. This includes things like varying timing between operations as well as injecting errors in responses.

To implement this, it would be too difficult to insert in pipelined operations. So instead, these would be added to a version of write_mem and rd_mem that do operations ‘one at a time’. In injection mode, pipelining would be turned off and all operations would happen individually. Programmable MMIO fields could control these features.

- Read TLX command data in bursts or individually (that is max_flit_read_burst = 1,2,3,4)
- Write TLX response data in bursts, with FLITs spaced apart or bunched (that is, max_flit_write_burst = 1,2,3,4)
- Varied interval between command acceptance or response generation (that is, min_cmd_accept_spacing = 0,1,2,3…, min_resp_gen_spacing = 0,1,2,3…)
- Hold off on response generation long enough to cause timeouts
- Inject periodic errors on responses (that is all fail response codes) and data (that is bdi)
- Send responses to command in a different order than the commands arrived (commands and responses should be paired by capptag, not order, on the host)
- Send or receive ‘Ordered Word’ first operations

This section should be considered ‘optional’, and not implemented if it causes complications with schedule or physical FPGA implementation (placement, timing).

8.5 Illegal adjustments to TLX interface

This section describes things the AFU can be programmed to do that are considered illegal, which can be useful to force the host to detect and handle error situations. To figure out what goes into this list, talk to the TLX and host teams to see what illegal situations (if any) they detect and report.

(The list of options is currently empty but kept as a placeholder as a reminder for future consideration.)

This section should be considered ‘optional’, and not implemented if it causes complications with schedule or physical FPGA implementation (placement, timing).

8.6 Miscellaneous information

This section contains miscellaneous information about related topics, like the FPGA selected or Vivado tools.

- (Xilinx AXI4 interface)
  http://www.xilinx.com/products/intellectual-property/axi_interconnect.html#overview