# ThymesisFlow disaggregated memory hardware design and deployment on Power 9 (AC922)

**Dublin Research Lab, IBM Research Europe**

Version 1.1
11 March 2020

## Contents

## List of Tables

## List of Figures

# 1 Overview

This design documents discusses the details of ThymesisFlow design which comprises a full stack prototype for disaggregated memory on IBM Power9 systems and is built over Open the Coherent Accelerator Processor Interface (OpenCAPI) stack. Apart from the design documentation details, this document describes how to bring up the system on a Power9 AC922 setup.

The document is structured as follows: First it describes the overall approach based on OpenCAPI v3.0 . Then it describes the hardware stack down to low-level design details and how to build and deploy it. Finally it describes the software stack support and how to deploy it. The document assumes reader familiarity with the OpenCAPI V3.0 transaction layer protocol specification, Verilog HDL and previously opensourced OpenCAPI reference designs (primarily the LPC reference design).

## 1.1 Overall Approach

In Figure 1 below the overall approach is presented.

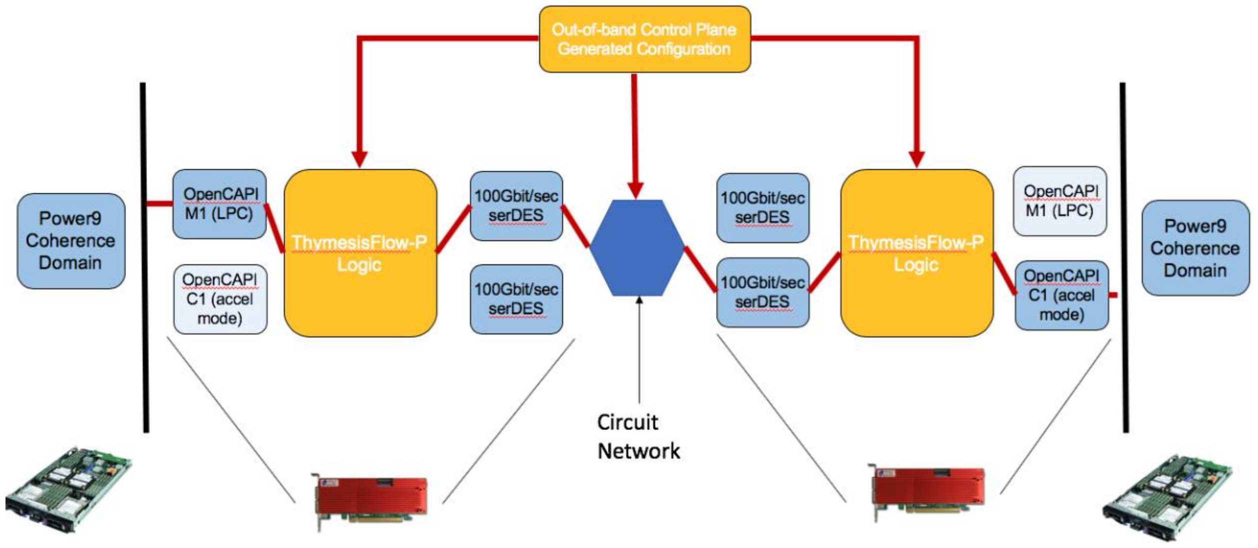


*Figure 1 Disaggregation over OpenCAPI. Architectural blueprint*

The design takes advantage of the OpenCAPI modes as follows : 1. M1 or LPC or memory Host Agent at the Compute side (i.e. the side where remote memory is mapped to local physical address space) depicted on the left and C1 at the remote memory side (i.e. the side where local system memory is stolen and provided as remote) depicted on the right, to enable disaggregated memory.

The hardware datapath depicted implements the physical attachment of remote memory. Operating system actions are further required for the logical attachment of disaggregated memory (i.e. linux kernel sparse memory model-based memory hotplug support).

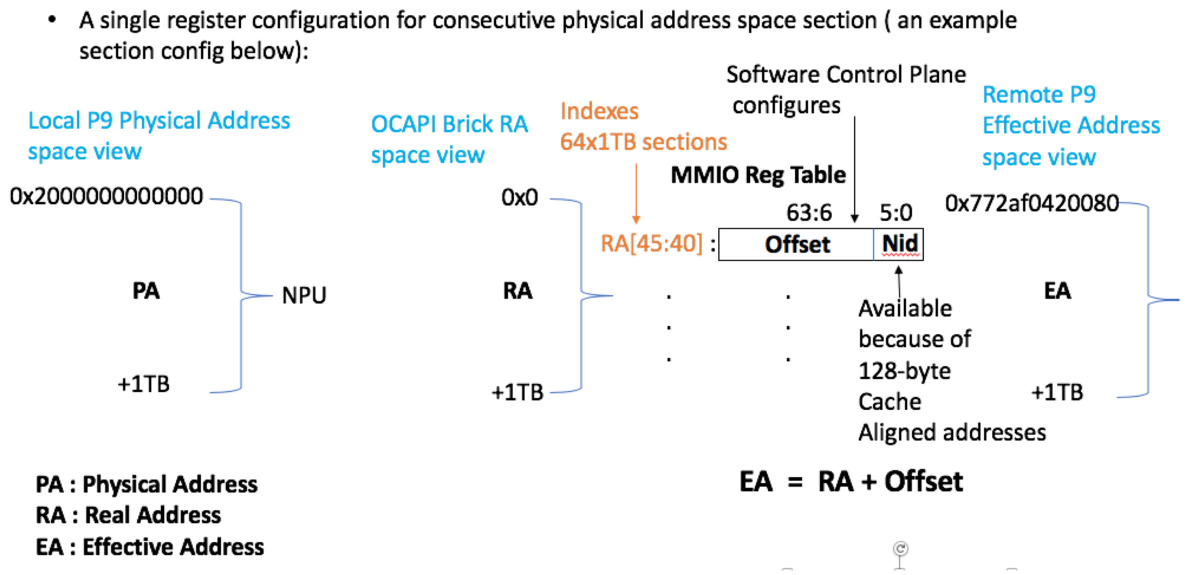
This design bridges directly processor cacheline traffic, without any further intermediate caching support) but not coherence traffic. Therefore it enables the scale-up of memory resources from the perspective of the compute node, *but cannot be used to further expand its SMP domain with remote CPUs.*

Finally, based on the locally available network facing serDES interfaces, the design can be configured to steal memory from one or more hosts concurrently and also has the option of bonding serDES interfaces (channel bonding happens at the transaction layer) towards the same host improving this way the available remote memory bandwidth.

## 1.2 OpenCAPI transaction modifications for remote mastering

As it is explained in the OpenCAPI Transaction Layer spec, the M1 mode (or LPC) is working with Real memory addresses (like PCI-e peripherals) and it uses specific transaction operation codes (opcodes). The real address map from the perspective of the M1 design starts from 0x0. Thus the peripheral can infer it's address space without being dependent on the Physical address

base address where it is mapped on a given system bus memory layout. E.g. OpenCAPI brick might be mapped to a Physical Adress base 0x200000000, but when P9 processor masters a request to that address, the internal stack of the brick will receive the address 0x0. Subsequently if P9 masters request at 0x200000100 the brick stack will receive 0x100 address and so on.
On the other side, the memory is stolen in accelerator mode (C1 mode). That means the OpenCAPI brick in (C1) mode is posing itself as an accelerator and asks from standard application to provide access to its allocated memory address space. These type of accesses require effective (or virtual) addresses. In addition, the (C1) mode uses its own transaction operation codes, different from the LPC (M1) ones though the operations and transaction header format are identical from any other perspective. Therefore, it is needed to change both the addresses and the transaction opcodes to successfully issue a transaction that comes from an M1 design to a remote C1 design.
While opcode changes are standard and can be hardcoded, the Real Address to Effective Address (or virtual address) depends on which address spaces are paired at runtime and what addresses are available then. It is a control planes' responsibility to determine what offset has to be applied to convert RA to EA. In the current implementation, continuous linear physical address space regions are referred-to as sections and are of a fixed size that has to be decided at hardware building time are employed by the M1-side (compute side) of the design. It is paramount that the memory side application is providing access section-size continuous address space (and cacheline aligned) allocated regions for the currently implemented approach to work.
An example of this core approach is depicted in Figure 2 below:



- A single register configuration for consecutive physical address space section ( an example section config below):

*Figure 2 Real Address to Physical Address Conversion Approach*

Therefore this approach maps a continuous well-defined physical address region of the compute side to a continuous well-defined virtual address space memory region of the remote memory provider that is of the same size. Any transaction mastered to the compute side physical address space agreed region ends up being served by the remote memory virtual address space region.

In addition to the above, the transaction feature network identifiers/credentials to get forwarded over the network, which we discuss next.

## 1.3 Network approach

Current design features support to drive a circuit network. Outgoing transactions get annotated, based on the memory address range they belong, with a network identifier which is used by the thymesisflow routing layer. In the current implementation, the identifier designates the outgoing serDES pipeline that has to be used for transmission. Notably, the routing layer can be configured to aggregate one or more network pipelines under the hood. In the current implementation each network pipeline delivers 100Gbit/sec (bundle 4x25Gbit/sec transceivers) using the datalink layer protocol Xilinx Aurora, which offers framed flit transmission with CRC check support. Likewise, at the reception side, the design annotates the network pipeline from which the transaction arrived and sends the response back from the same network pipeline. More details will be provided in the description of the thymesisflow routing layer logic.

## 1.4 Software-defined memory concept

Combining the memory and network configuration described in sections 1.2 and 1.3, we introduce the concept of software defined remote memory, which is summarized in the Figure 3 below. An out-of-band configuration interface sets appropriate hardware datapath configuration support to modify transactions and route them towards the appropriate network pipeline. The remote memory is regarded as software-defined, as the design provides a software-based way of describing a remote memory node and hotplugging it to a running machine. Currently both the address that needs to be applied for converting from local RA to remote EA and the network identifier are pushed to the same register – one per segment. The current design supports 12 x1 TB sections that can be configured independently by different registers.

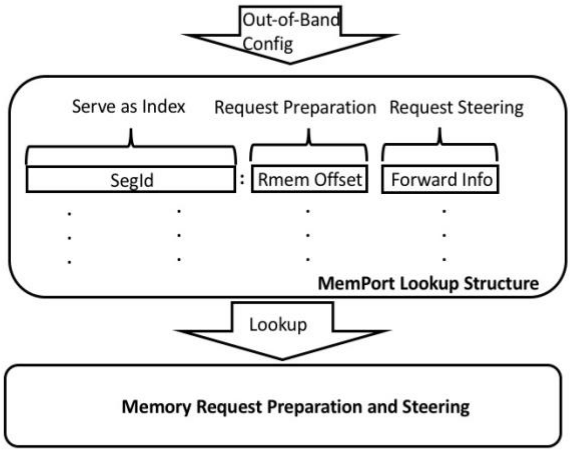


*Figure 3 Software-defined memory concept*

# 2 ThymesisFlow Design

Current version of ThymesisFlow hardware design can assume either the memory or a compute role (by compiling in different modules) and use both QSFP ports available on the Alpha Data 9V3 FPGA.

## 2.1 Design Blueprint

In Figure 4 below the high level blueprint of the design is depicted.

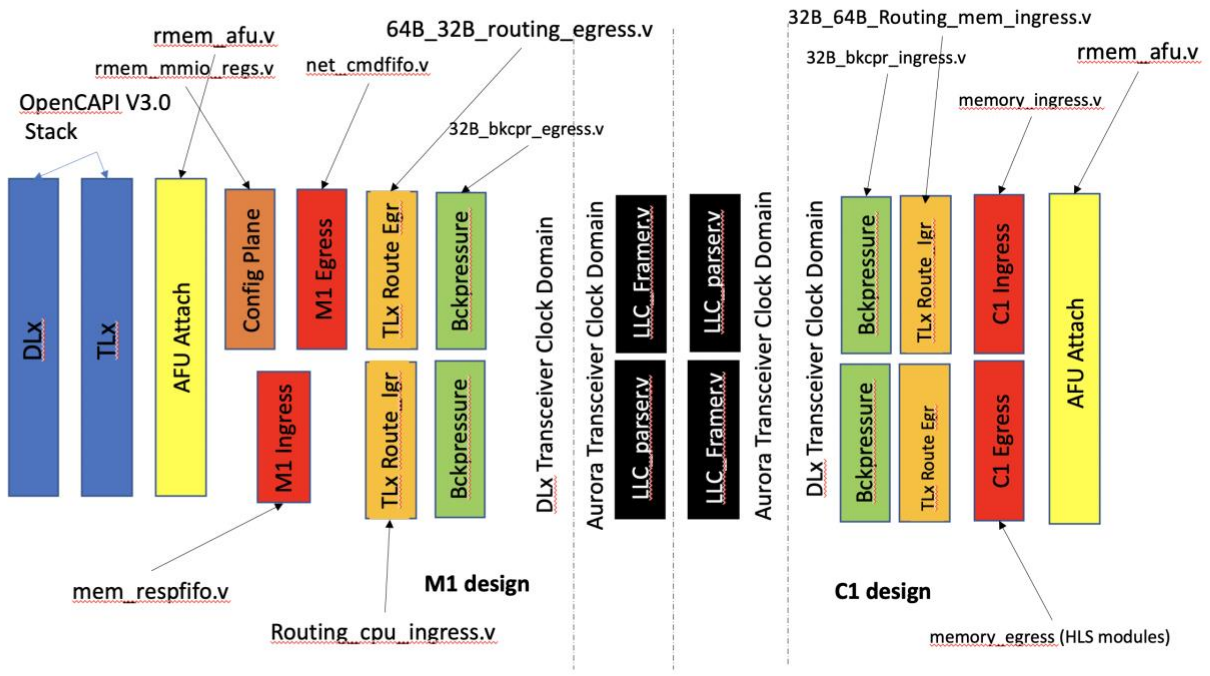


*Figure 4 High-level ThymesisFlow design blueprint*

The starting reference design for this implementation is the opensourced OpenCAPI LPC reference design provided.

The LPC reference design, illustrates the OpenCAPI transaction request/response interaction with the TLx Parser and Framer engines that are needed to receive and send OpenCAPI DL frames towards the P9 host, the credit management to handle backpressure on command and data channels, it completely abstracts the PCI-scan commands (cfg_*) which are important for device discovery and, finally, separates MMIO traffic that reads/write to global registers from main system memory traffic.

This document will not get into details about subsystems and techniques that have been already demonstrated (and thus copied from) the LPC reference design, it will just provide targeted references where needed.
ThymesisFlow design is a dataflow design that transports OpenCAPI 128B transactions only (not 256B and not partials with WEN), their response headers and data flits, between compute and one or more memory endpoints that are interconnected over a circuit network.
All Transaction header (command or Responses) flit formats used throughout the design are described and justified in a dedicated section. Subsequently, the design description that follows describes all engines of the pipeline, starting from the LPC slave side where the memory access requests are initiated by the Power 9 compute node. Then, the path of the request is described until the transported transaction is mastered at the memory side (excluding the network interactions). In the same spirit, the response pipeline description, starts at the memory side where responses are generated and comes back to the compute side (excluding the network as well). In the context of each node role there are two independent pipelines: Egress pipeline that pushes flits towards the network and ingress pipeline that receives from the network. This document refers to them as compute egress, compute ingress, memory egress and memory ingress pipelines.

The network facing interfaces (i.e. the low-level control protocol or LLC), the backpressure support and the configuration/performance counter MMIO interface are described independently at the end of the description of the design.

**Design dataflow module interfaces**

With the exception of the data passing interface of TLx Framer and Parser which takes place with the appropriate exchange of credits, all other ThymesisFlow modules use the **AMBA AXI-4** stream protocol to exchange flits. When specific module interfaces are referred the name and the bit width of the TDATA field of each AXI4-Stream port will be reported.
In addition to AXI4-Stream some modules interface Xilinx FPGA BRAM using the native BRAM interface as it defined by Xilinx.

**Design clock domains**

The ThymesisFlow design runs all the OpenCAPI related operations in the OpenCAPI link transceiver clock domain. Each network interface designates its own clock domain and it is interfaced to the rest of the design via appropriate clock-domain crossing queues.

## 2.2 Code availability and source structure

ThymesisFlow-P is realized as an extension of the LPC reference design and the OpenCAPI v3.0 stack. The design is coded in Verilog with an exception of 2 vivado HLS modules, and, in addition, the building approach favors a command line invocation of the Vivado toolchain – block design diagram methodology is not used. The tcl scripts provided will create a new project, create and register IP instances. Then vivado can be started in GUI mode to execute the build.

All the ThymesisFlow-related code is under the folder afu/thymesisflow. Under these folder there are the Thymesisflow verilog modules that operate in the OpenCAPI DLx transceivers clock domain. Folder hls_modules contains 2 HLS modules that handle the memory egress side and also run in the OpenCAPI DLx transceiver clock domain. Finally, the thymesisflow_llc folder contains the link control protocol modules that run in the network facing transceivers clock domain.

## 2.3 Design flit formats used

The OpenCAPI reference stack features a 512-bit (64B) datapath. ThymesisFlow retains this datapath size at all stages. Flit formats below are the formats used to represent various flit-types at different stages of the design. Explanation behind format choices are described where necessary.
**TLx Command Flit (Bit Width 199)**
TLx Command Flit contains all fields mandated by the OpenCAPI Transaction Layer specification. TLx commands are issued towards the P9 OpenCAPI Nest TL. This design elects TLx as the command format for the network, which means that all commands have to be converted to TLx Opcodes and proper effective addresses, before they get transmitted via a network interface. TL command opcodes and formats have to be handled/converted locally at each Compute endpoint. These flit format type for the command is being used by all M1/C1 modules prior to converting the flit to a 64B network format (which features a few additional fields at different offsets).

| Flit Field Type | Bit Range on the Flit (Big Endian) |
|---|---|
| TLx OpCode | (198:191) (8 bits) |
| TLx Effective Address | (190:127) (64 bits) |
| TLx CappTag | (126:111) (16 bits) |
| TLx DL | (110:109) (2 bits) |
| TLx PL | (108:106) (3 bits) |
| TLx BE | (105:42) (64 bits) |
| TLx actag | (41:30) (12 bits) |
| TLx Stream id | (29:26) (4 bits) |
| TLx PASID | (25:6) (20 bits) |
| TLx pgsize | (5:0) (6 bits) |

*Table 1 TLx Cmd Flit Format*

**TL Resp Flit (Bit Width 38)**

TL Resp Flit holds the TL response to a TLx Command. This is elected as the Response command flit format that should travel on the network. The fields are depicted below:

| Flit Field Type | Bit Range on the Flit (Big Endian) |
|---|---|
| TL Resp OpCode | (37,30) (8 bits) |
| TL Resp Capptag | (29:14) (16 bits) |

| | |
|---|---|
| TL Resp code | (13:10) (4 bits) |
| TL Resp DL | (9:8) (2 bits) |
| TL DP | (7,6) (2 bits) |
| TL pgsize | (5:0) ( 6 bits) |

*Table 2 TL Resp Flit Format*

**TLx Command Store Flit (Bit Width 215)**

This the TLx Command Flit prepended with an additional Capptag at the beginning, which is used by the command reissue engine at the memory ingress side.

| Flit Field Type | Bit Range on the Flit (Big Endian) |
|---|---|
| TLx Local Capptag | (214:199) (16 bits) |
| TLx OpCode | (198:191) (8 bits) |
| TLx Effective Address | (190:127) (64 bits) |
| TLx CappTag | (126:111) (16 bits) |
| TLx DL | (110:109) (2 bits) |
| TLx PL | (108:106) (3 bits) |
| TLx BE | (105:42) (64 bits) |
| TLx actag | (41:30) (12 bits) |
| TLx Stream id | (29:26) (4 bits) |
| TLx PASID | (25:6) (20 bits) |
| NetID | (5:0) (6 bits) |

*Table 3 TLx Command Store Flit*

**Data Flits**

Following the OpenCAPI specification, dataflit is equal to 64B and occupies the full datapath width. In selected stages of the pipeline, when relevant information is available, the dataflit is 513 bits wide, with the most significant bit ( location 512 ) being used to mark whether the dataflit is rogue or not (Bad data indicator or BDI).

**Casting TLx Cmd and Response flits to 64B and annotation space (Bit Width 512)**

Since the OpenCAPI stack datapath width is 512bits (64B) each of the aforementioned command and response flits are casted by this design to the **most significant** bit range of the 64B datapath flit starting at bit position 512. The largest Flit to be casted is the TLx Command flit with is 200 bits long. When a flit is either a Command or a TL Response flit flavor, the range that starts at bit offset 296 and down to 256 (296:256) is regarded as annotation space and can be used for passing information to other modules of the pipeline. The casting and annotation of the 64B flit is described in the table below. Please note that the lower half of the 64B flit (bits in the range 255:0) are not forwarded when the flit is an OpenCAPI transaction cmd or response. The lower part is only forwarded when the flit is data.

| Flit Field Type | Bit Range on the Flit (Big Endian) | |
|---|---|---|
| All Types of Cmd or Resp Flit | (511:297) (215 bits) | |
| Annotation Space | (296:256) (40 bits) | |
| | Net Id | (296:291) (6 bits) |

*Table 4 64B datapath Flit, flit casting and annotation space*

A comprehensive way of how casting happens for various flit type and data, is provided at the following HLS ocx_flit.cpp file, which defines the OCXFlit object resembling the 64B flit and features a variety of functions that implement the casting. This is a core file used is by the memory egress HLS modules and can be found in the respective folder of the project.

## 2.4 Compute egress pipeline modules

The LPC loads and stores get submitted to the Compute egress pipeline. Starting from the LPC reference design (lpc_afu.v) the *rmem_afu.v* file includes all the code that interfaces TLx Framer and TLx parser to both M1 and C1 Egress/Ingress pipelines of ThymesisFlow.
The default *lpc_cmdfifo.v* implementation is used to handle credit returns with the TLx and receive the LPC (Tlx->Afu) command flits and data. These are classified between MMIO commands, which in our case are used to read/write to MMIO registers that define design behavior, and the standard LPC load/store memory traffic. The reader should refer to the LPC reference design to understand how the commands are handled.

### 2.4.1 net_cmdfifo.v

**Inputs:**

1. fifo pull interface (Xilinx ap_fifo)

**Outputs**

2. AXI4-Stream (handles TLx->Afu cmdflits)

In the sequel , ThymesisFlow introduces a new fifo, that it is named net_cmdfifo.v and has the following 4 basic differences from lpc_cmdfifo.v:

1. The Read side has been converted by a Flit-Pull interface (i.e. return credit to get next flit) to a Flit-push interface following the AMBA AXI-4 Stream protocol. According to AXI4-Stream in each and every cycle where this fifo contains valid data, if the other side has the ready signal asserted, the flit is considered delivered (push semantics).
2. Upon Flit reception, it converts TL *read_mem* and *write_mem* command opcodes, to TLx *read_wintc* and *dma_w* to prepare them for the network transmission. ***In the current implementation transcoding of partials and .n commands is not supported.***
3. Looks at the proper bits of the read_mem or write_mem Physical Address (PA) to determine to which PA section it belongs to apply the proper offset (that has been passed through MMIO registers) . In the current configuration, section size is hardcoded at build time. The code defines

as example 1TB consecutive sections and supports up to 12 different sections (which is well beyond 4TB that the current OpenCAPI brick implementation can do). Changing the number of registers in *rmem_mmio_regs.v* it is possible to increase the number of supported sections, and, in addition, by examining different ranges of bits the section size can be increased or decreased. **Note:** In the current prototype the MMIO register contents are used directly by the datapath.

4. Based on the section where the command belongs, a network identifier is added to the annotation space and is to be used later from the routing layer for forwarding. Currently, taking advantage of the 128-byte address alignment requirement for OpenCAPI commands, the network identifier is squeezed at the lower order bits (5:0) of the memory offset that is delivered via MMIO.

On exiting the net_cmdfifo, the command flits have been converted to be properly issued to remote memory and also includes a network identifier annotation that can be used later by network adaptation logic for transmission. *rmem_afu.v* is features a state machines that pull the data flits that need to be sent when net_cmdfifo receives a write transaction. The dataflits are being directly stored to a separate standard Xilinx fifo.

Subsequently, the compute egress path exits the rmem_afu.v module, with 2 AXI4-S stream channels:

| Signal Name | Type (Output) |
|---|---|
| ncff_cmd (command flits) [198:0] | AXI4- Data Stream |
| cmddata_Fifo (dataflits) [512:0] includes bdi. | AXI4- Data Stream |

### 2.4.2 thymesisflow_64B_compute_egress_adapter

**Inputs:**

1. AXI4-Stream (200bit cmd flits)
2. AXI4-Stream (64B data flits)

**Outputs**

1. AXI4-Stream 64B (cmd flits and data flits multiplexed in time)

The main task of this module is to multiplex the received flits in time and format the incoming cmd flits to the required 64B format for the network. When a write cmd flit is read from port 1 then the respective dataflits are forwarded from port 2 towards the output.

### 2.4.3 thymesisflow_64B_32B_routing_egress

**Inputs:**

1. AXI4-Stream (64B multiplexed in time cmd and data flits)

**Outputs**

1. AXI4-Stream (32B cmd flits and data flits multiplexed in time for output port0)
2. AXI4-Stream (32B cmd flits and data flits multiplexed in time for output port1)

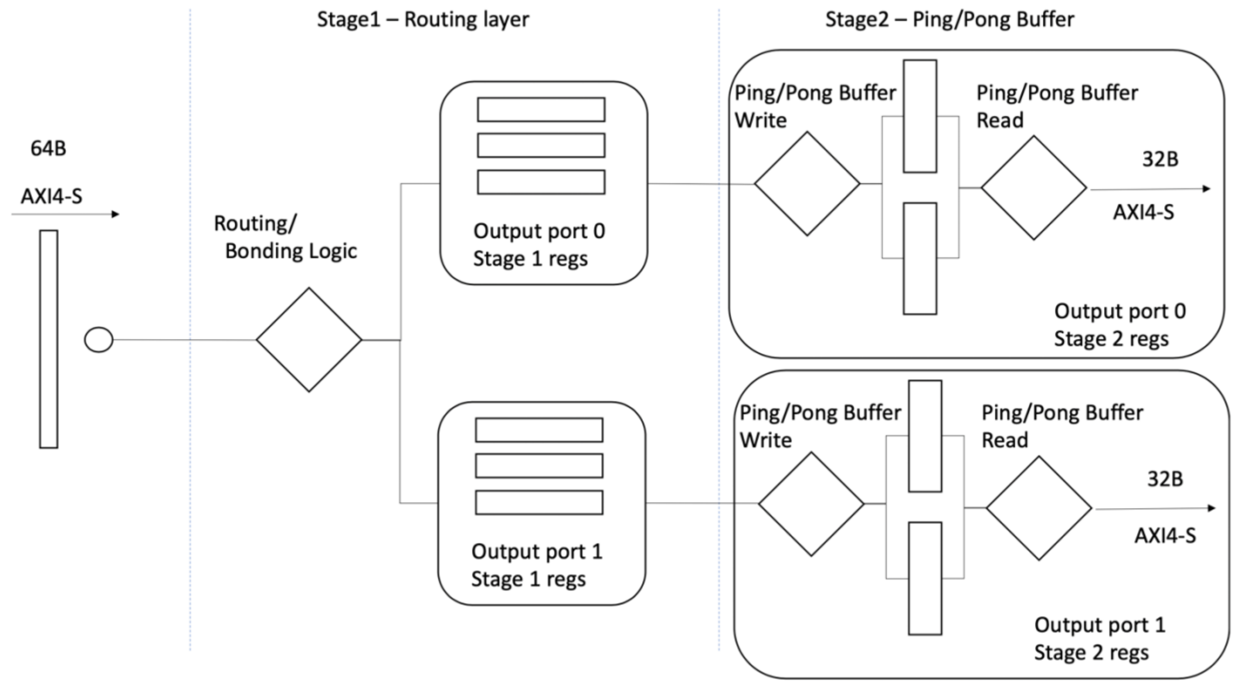


*Figure 5 routing layer egress design*

This module design is depicted in Figure 5. It receives a 64B multiplexed traffic at the input and according to each command network identifier field is routing the traffic to the appropriate output that drives a different network pipeline.
The design features 2 stages. Firstly, it accepts the command and routes it to the proper set of registers of the first stage. The second stage is a ping pong buffer (one per output) that absorbs output stalls and stages the propagation of AXI4-Stream TReady signal to relax timing constraints. This module also performs downsizing at the output. The emitted flits are downsized to 32B to match the datapath width of the network pipelines (4x25Gbit). The command flits just get the

upper 32B transmitted (as the lower 32B do not have valid information anyway) whereas the 2 64B dataflits are split in 4x32B flits with the upper 32B being transmitted first (511:256) and the lower (255:0) immediately following. This module can also handle responses and therefore it is used as is at the response path of the C1 design side (memory egress).

## 2.5 Compute ingress pipeline modules

### 2.5.1 thymesisflow_32B_64B_routing_compute_ingress

**Inputs:**

1. AXI4-Stream (32B multiplexed in time cmd and data flits from port0)
2. AXI4-Stream (32B multiplexed in time cmd and data flits from port1)

**Outputs**

1 AXI4-Stream (32B cmd flits and data flits multiplexed in time for output port0)

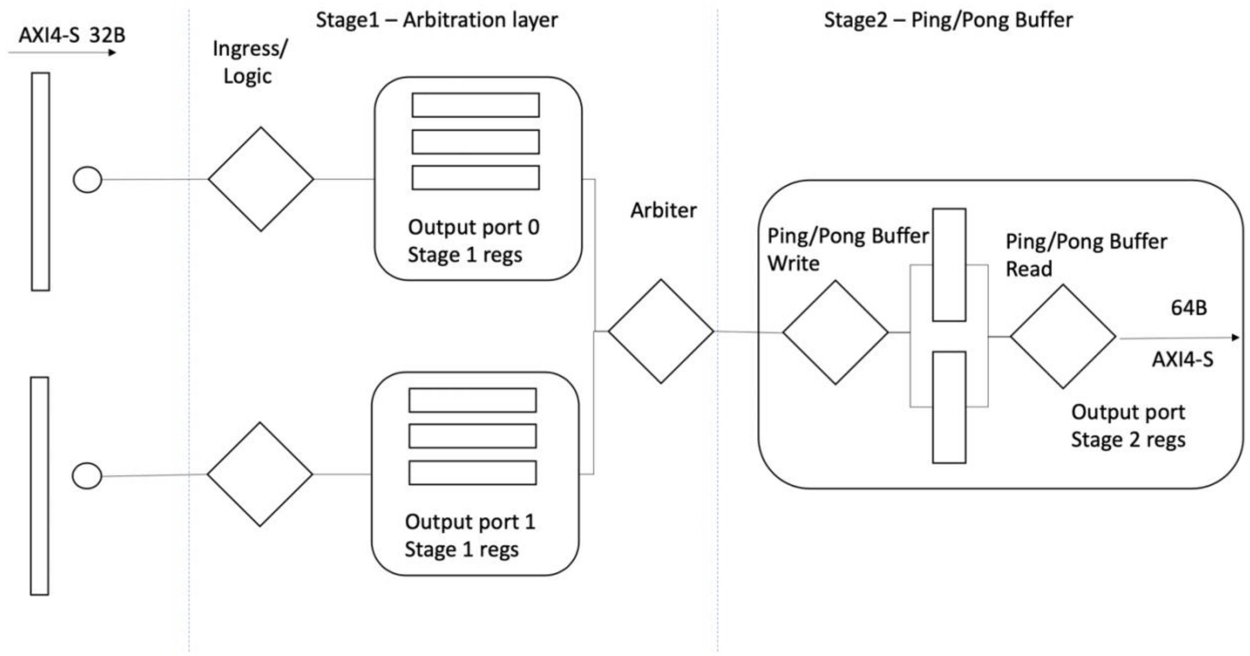


*Figure 6 Thymesisflow Compute/Memory routing Ingress*

Memory transaction responses arrive from the network 32B pipelines to the computer ingress pipeline. This module is leveraging a round-robin arbiter that receives entire transactions from each port input 32B pipeline and upsizes the transaction flits to 64B towards the single 64B output. The modules employs 2 stages. The first stage is 32B and receives independently flits from each port and the second stage is accepting based on the arbiter decision entire transactions (i.e. when the transaction of a given port is a read response, the dataflits are also received before the arbiter decision is taken into account again). The first stage also converts the response opcodes which have the C1 values ( TLx->AFU resp codes) to M1 resp opcodes (AFU->TLx). The second stage upsizes the transaction flits and delivers them to the next module according to the TLx Framer v0.3 implementation dataflow rules. The dataflits are returned to *rmem_afu.v* which features logic that drives the *mem_respfifo.v*.

Because of TLx Framer v0.3 implementation dataflow rules (page 19) the responses need to be delivered with resp command and first dataflit co-incidented, with the next dataflit brick-walled in the next cycle or the framer will mark them as rogue. The *mem_respfifo.v* (instance named RFF) provides special signals to implement this delivery scheme and it needs the flits submitted with control bits asserted according to their type. The RFF delivers flits to the framer at *rmem_afu.v* by appropriate combinational logic that is driven by the signals (rff_out_rdata_valid and rff_out_resp_valid) that indicate what type of data are contained in the flit.

## 2.6 Memory design overview

At the memory side, while the routing layer operations are identical with the compute ingress/egress pipelines (and the same modules are used with the exception of 32B_64B_routing_memory_ingress.v which does not need to perform the opcode transcoding like the compute side counterpart), there are additional requirements that demand ingress/egress cooperation. Therefore, before elaborating on the module design for each side, we firstly present the global memory side (C1) design and the reasons for the ingress and egress interactions.

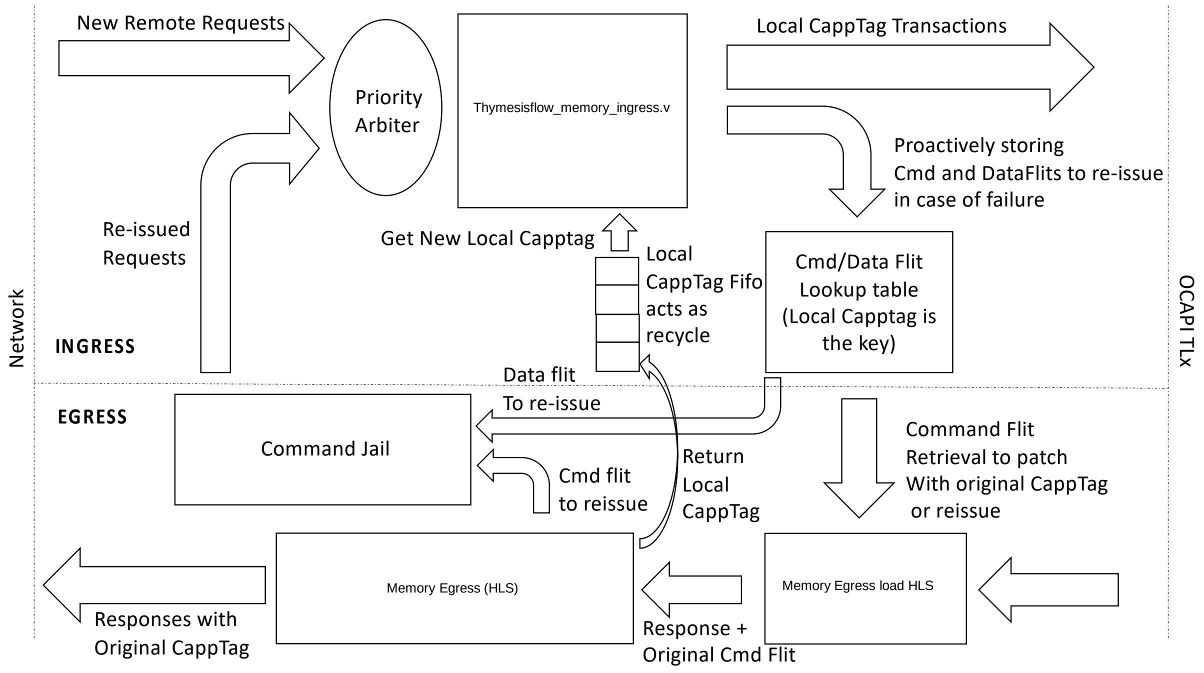

New Remote Requests
Priority Arbiter
Thymesisflow_memory_ingress.v
Local CappTag Transactions
Proactively storing Cmd and DataFlits to re-issue in case of failure
Re-issued Requests
Get New Local Capptag
Local CappTag Fifo acts as recycle
Cmd/Data Flit Lookup table (Local Capptag is the key)
Network
INGRESS
OCAPI TLx
Data flit To re-issue
EGRESS
Command Jail
Cmd flit to reissue
Return Local CappTag
Command Flit Retrieval to patch With original CappTag or reissue
Memory Egress (HLS)
Memory Egress load HLS
Responses with Original CappTag
Response + Original Cmd Flit

*Figure 7 Memory side ThymesisFlow design*

The memory side of Thymesisflow is mastering requests towards system memory (OpenCAPI C0 mode). One of the key challenges here, is that nMMU is in the address path of the OpenCAPI nest towards system memory, and therefore the design must be associated with an effective (i.e. virtual) address space (either from kernel or a user application context) and use virtual addresses to access system memory. Since the virtual address space is managed by the linux kernel, as it also happens with CPU accesses, the page tables might not be set for a variety of reasons when the access is attempted, thus causing a page fault to occur. In the case of OpenCAPI nest the system responds to transactions that ended up in a page fault with appropriate response error codes that indicate that a page fault happened, and the handling is in progress. The commands that are issued to the page that faulted need to be re-issued by C1 design. This is the first reason for which egress datapath should coordinate with ingress: A command failed and this is detected by egress modules that receive the failed response, which should interact with ingress to issue it again.

The second reason is the definition of an isolated CAPPTag domain. Since in ThymesisFlow commands from different remote nodes may arrive concurrently to the memory node, it might be the case that the CAPPTags of transactions that come from different hosts and are in-flight clash, which will result to catastrophic failure. For that reason, the memory side maintains a Fifo of local CAPPTags that is preloaded during system initialization with a number of CAPPTags (which also acts as the knob of the maximum allowed outstanding transactions for the memory side). The CAPPTags are consumed from the fifo by *thymesisflow_memory_ingress.v* module with each new transaction that arrives from the network and are returned back from memory_egress HLS module when successful responses are sent back to the network.

The local CAPPTags play another important role: They serve as indexes on a lookup table (based on BRAM) that acts as a temporary store for newly arriving command and data flits from the network. In case access failure happens, the CAPPTag of the failed response is used to retrieve the cmd and data flits from the BRAM store and re-issue them. In addition, in case a command succeeds, the store is used to get the original CAPPTag that was used by the remote host and patch the response so that the recipient can handle it. Obviously, upon reception of a new transaction, the original CAPPTag is replaced with the local one acquired from the fifo.

Figure 7 provides a high-level overview of how memory side modules interact. In the sequel we will describe each module separately.

### 2.6.1 Memory ingress pipeline modules

#### *2.6.1.1 thymesisflow_memory_ingress.v*

**Inputs:**

1. AXI4-Stream (64B multiplexed in time cmd and data flits from network
2. AXI4-Stream (32B multiplexed in time cmd and data flits from re-issue engine)
3. AXI4-Stream (2B Capptag Fifo )
4. Start ( 1-bit, kickstarts the module to send ACTAG command)

**Outputs**

1. AXI4-Stream (64B cmd flits demultiplexed )
2. AXI4-Stream (64B dataflits demultiplexed + BDI bit)
3. BRAM interface for cmd store (216bit)
4. BRAM interface for dataflit0 store (64B)
5. BRAM interface for dataflit1 store (64B)

The memory ingress module is depicted in Figure 7. It implements all the memory ingress tasks as described in 2.6. It receives flits from the network and local cmd re-issue module and it receives local CAPPTags. The *start* signal initiates the module, by returning an ACTAG (0) towards P9 to associate the commands that will be issuing with a context according to the OpenCAPI specification. Then it features 3 ports towards BRAM. There is the cmd_lookup port that stores command flits and 2 datalookup ports, each one storing 64B data flit as follows: If the TLx command is a 128B Write, then dataflit0 store receives the first dataflit and dataflit1 store receives the second dataflit. Finally the ingress module features two seperate AXI4-S ports that push demultiplexed command and dataflits towards the OpenCAPI nest.

The demultiplexed cmd and data flits end up in *rmem_afu.v* module. Subsequently they drive *mem_cmd_fifo.v* and *mem_data_fifo.v* that are accepting data via AXI4-S and export an ap_fifo interface towards TLx framer. *rmem_afu.v* features combinational logic that unloads the fifos towards the framer according to TLx framer v0.3 interface rules (page 19 of TLx framer/parser design document). There are plently of comments explaining the procedure.

### 2.6.2 Memory egress pipeline modules

#### *2.6.2.1 memory_egress_load and memory_egress HLS modules*

**Inputs (memory_egress_load):**

1. AXI4-Stream (60-bit responses)
2. Cmd BRAM interface (to load cmd flit from local store)

**Outputs(memory_egress_load)**

1. AXI4-Stream (32B combo flit that contains the Response header and the Cmd header counterpart that was retrieved from local store)

**Inputs (memory_egress):**

1. AXI4-Stream (32B combo resp/cmd flit from egress_loader)
2. Data0 BRAM interface (to load dataflit0)
3. Data1 BRAM interface (to load dataflit1)

**Outputs(memory_egress)**

1. AXI4-Stream (64B multiplexed response/data towards the network)
2. AXI4-Stream (64B multiplexed response/data toward the ingress to re-issue)
3. AXI4-Stream (2B return tag to CAPPTag Fifo)

Starting from interfacing the TLx parser in *rmem_afu.v*, the *mem_respfifo.v* receives command responses from memory and makes them available to the external modules through AXI4-Stream interface . In the case of successful reads, the dataflits are separately delivered to an external Xilinx Fifo (to meet timing). The width of this fifo is 64B and the depth 128 flits to prevent overflow due to all the cmd flits that can be transit in all the various cmd fifos that are used until dataflits start getting consumed by the routing layer.

The *mem_respfifo.v* module fifo is consumed by memory_egress_load module (depicted in Figure 7), which for each response flit,  retrieves (using the local response CAPPTag) the command flit counterpart that has been issued from the BRAM store. Interacting with BRAM introduces a cycle delay. The combination of the command and response flits at the output of memory_egress module results in a 32B wide signal that is mapped as follows (**Error! Reference source not found.**):

| Field name | Egress Loader 32B bit range (BE) |
|---|---|
| Not Used | 255:253 |
| TLResp | 252:215 |
| TLx Cmd issued | 214:0 |

*Figure 8 Egress Loader Internal flit format*

In the sequel, response flit together with the original command flit, are received from ocx_memory_egress module which features the main state machine in ocx_resp_eng.cpp

During initialization (enable_V gets permanently asserted) the state machine is spending a number of cycles to load locally attached CAPPTag fifo with local CAPPTags.  Local CAPPtags start from 1 and are incremented to load the attached fifo up to maximum number of local CAPPTags. The CAPPTags are also used as indexes for the store (so a resp flit with CAPPTag equal to 1 has the original cmd flit stored at offset 1 in the BRAM store). Therefore, the number of CAPPTags decided here take into account the attached CAPPTag fifo size, and the BRAM store size that has been used. After measurements of the maximum number of transactions that may be in-flight during intensive system operation, the number of local CAPPTags  has been fixed to 512. Therefore, the INIT state of this state machine loads 512 CAPPTags before it locks in the start state where it serves traffic.

In the GET_CMD state, the module receives the response flit and classifies it based on the response code: It can be one of the following:

- Successful read or write command, so the response needs to be forwarded towards the network.
- Failed command with code that is not related to page faults. The response is forwarded towards the network
- Failed command with Xlate_pending (i.e. page fault occured). These commands are sent to jail and should not be sent to the re-issue datapath before xlate_done arrives

- Failed command with late_retry (i.e. this command cannot be served because the translation failed but you need to retry it at a later time anyway because a page fault serving is in progress and a second cannot be currently initiated). This command is sent to jail as well.

ThymesisFlow design follows a naive approach regarding the page fault handling that will be fine-tuned in later versions. Each time a page fault occurs and a command comes back with a xlate_pending, all subsequent commands that come with xlate_reply are blocked until xlate_pending is served. As soon as xlate_done arrives the jail door is open and all blocked commands get served (some of them of course may still result in page faults that will be served when they are issued again). Once a cmd flit is sent to the jail, the corresponding dataflits are also retrieved from BRAM storage and are send to jail as well. The local CAPPTags of commands that are sent to jail are not recycled so the command may be re-issued using the previously assigned local CAPPTag. This also serves as security measure so that jail does not overflow, in case translations take too long (which typically don't ~ 1.6us ): Local CAPPTags will get depleted and *thymesisflow_memory_ingress.v* module will stop accepting cmds from the network.

The jail state machine is embedded in the memory egress module at ocx_rty_eng.cpp and takes care of the re-issuing step.

All successful commands are multiplexed in time and are forwarded towards the network via net_resp_out.

## 2.7 Network architecture and design details

With the primary focus being on low-latency, this version of ThymesisFlow integrates a circuit network. The modules described up to this point are totally oblivious to the network architecture. Their only responsibility is to route the flits towards the correct 32B egress network pipeline.

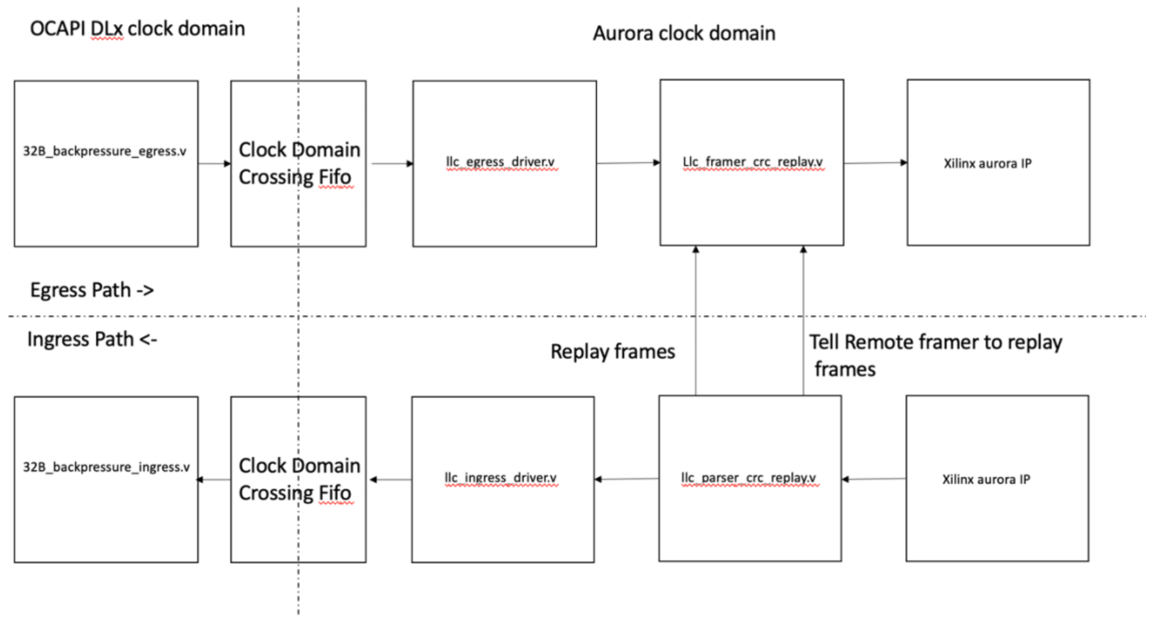


*Figure 9 Thymesisflow circuit network design*

ThymesisFlow stack requires a reliable network attached (i.e. the network should never allow a rogue flit to get delivered). If the network is brought down all together, both ends of the connection (compute and memory) will ultimately result in OpenCAPI command time out, which will cause the Power9 OpenCAPI nest to get fenced.

In **Error! Reference source not found.** the modules that are involved in the network transfer are depicted. Firstly, the backpressure modules (ingress/egress) that are in place to protect the ingress queue (clock domain crossing fifo) of the other side so it does not overflow. For this task these modules implement a credit mechanism that will be explained in the sequel. Then, entering the network facing transceiver clock domain a set of framer/parser modules along with driving logic comprise the lower-level stack of the link control protocol.

Thymesisflow LLC is reliability is based on a CRC-replay scheme. There is no Forward-Error-Correction (FEC) functionality involved as it hinders design timing closure. The backpressure messages (that carry credits) enjoy the channel reliability function. In synergy, the backpressure scheme together with the CRC-replay support provide the required network channel reliability.

### 2.7.1 thymesisflow_bkcpressure_egress/ingress.v and credit mechanism

**Inputs (for both):**

1. AXI4-Stream (32B multiplexed in time cmd and data flits)
2. Local ingress queue credit manager interface
3. Remote ingress queue credit manager interface

**Outputs (for both)**

1 AXI4-Stream (32B multiplexed in time cmd and data flits)
2. Local ingress queue credit manager interface
3. Remote ingress queue credit manager interface

Backpressure scheme is based on the exchange of credits between the endpoints of a serdes channel. The ultimate goal is to prevent the remote ingress queue from overflowing. The egress side is employing a credit manager that at the beginning of time is initialized with the number of credits that reflect the number of free slots of the remote ingress queue. As flits get through the backpressure egress module, these credits get consumed by pulsing proper signals to the credit manager that does the housekeeping for the remote ingress queue credits. At the same time, there is another credit manager that counts the local ingress queue credits. At the beginning of time, the latter credit manager is initialized to zero (by the backpressure ingress module) and as flits are departing from the ingress fifo queue (clock domain crossing fifo see Figure 9) it counts the credits that need to be returned to the remote egress path. The credits are returned by piggy backing them in the annotation space of OpenCAPI transaction header flits at the bit range (34:27). Each transaction header flit carries all the currently counted credits that need to be returned to the other side. In the same spirit the backpressure ingress of the remote end point is receiving the credits that are piggy backed in the arriving OpenCAPI transaction headers and returns the to the local credit manager that protects the remote ingress queue. Figure 10 depicts the credit scheme.

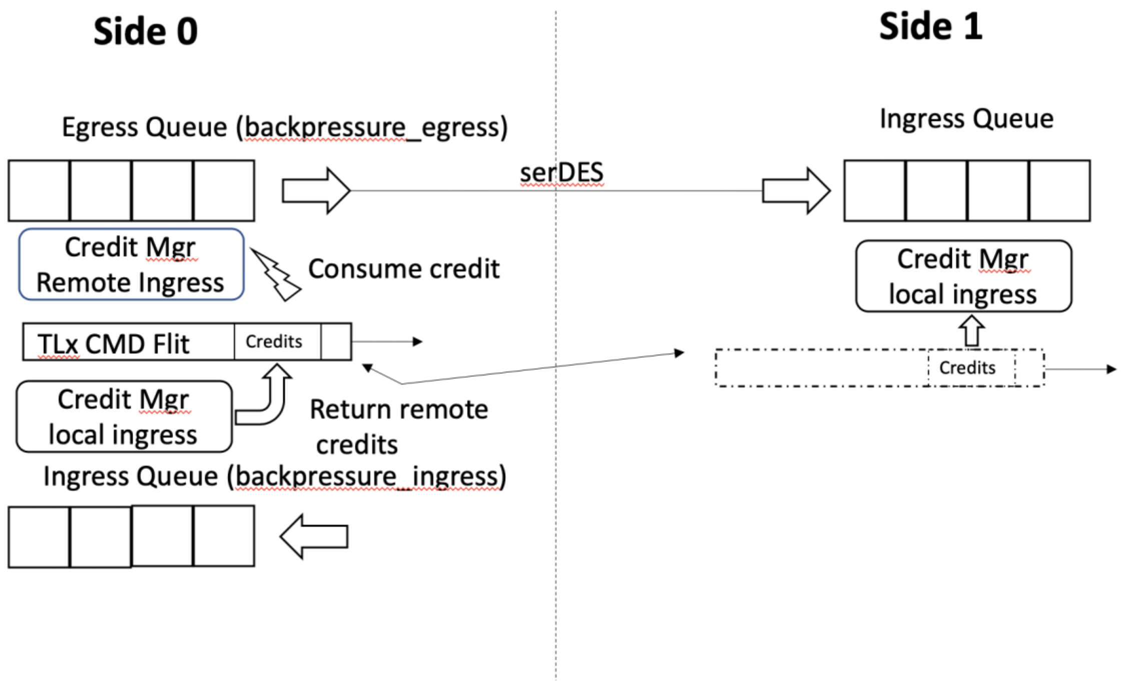


*Figure 10 Backpressure credit scheme*

### 2.7.2 thymesisflow_llc_ingress/egress/_driver.v

**Inputs (for both):**

1. AXI4-Stream (32B multiplexed in time cmd and data flits)

**Outputs (for both)**

1. AXI4-Stream (32B multiplexed in time cmd and data flits)

The ingress/egress driver just make sure that the framer-parser modules get padding flits when required to be able to always transmit 4-flit frames. Current network design is hardcoded to support 4-flit frames but that aims to be a compile time knob in the future. The driver egress logic just pads to required frame size with nop transaction headers (i.e. headers that use the opcode **0x40** which is currently not being used by the OpenCAPI Transaction layer protocol). Whenever incoming flits which are either OpenCAPI transaction headers or dataflits do not align to a 4-flit frame boundary, the egress driver module pads them by emitting nop flits. The ingress driver consumes nop flits without propagating them so the rest of the ingress logic is always expecting valid multiplexed OpenCAPI transactions.

After flits cross to the network domain, ThymesisFlow-P LLC (Logical Link Control) protocol is orchestrating the transmission. ThymesisFlow-P LLC is preparing flits to send them over Xilinx Aurora Datalink layer protocol which is widely used for serDES communications in the FPGA world. ThymesisFlow-P LLC provides the reliability support that is required by ThymesisFlow-P.

### 2.7.3 thymesisflow_llc_framer/parser_crc_replay.v

**Inputs (framer):**

1. AXI4-Stream (32B multiplexed in time cmd and data flits)
2. Local retransmit request (start retransmitting from given 6-bit frameid)
3. Remote retransmit request (tell the other side to start retransmitting from given 6-bit frameid)

**Outputs (framer)**

2. AXI4-Stream (frame-mode) (32B multiplexed in time cmd and data flits)

**Inputs (parser):**

1 AXI4-Stream (Frame mode) (32B multiplexed in time cmd and data flits)
2 Aurora module CRC check signals (crc_valid, crc_ok)

**Outputs (parser)**

1 AXI4-Stream (32B multiplexed in time cmd and data flits)
2 Local retransmit request (start retransmitting from given 6-bit frameid)
3 Remote retransmit request (tell the other side to start retransmitting from given 6-bit frameid)

ThymesisFlow-P LLC features a framer module that groups flits in fixed-size (currently 4) frames that get immediately transmitted. The framer features a circular buffer backed by BRAM that acts as a temporary storage for frames that have been transmitted, in case the other side reports CRC failure via special in-band message. Once a failure message is received the train of frames, starting from the failed one gets retransmitted, in-order, being replayed from the circular buffer. At the ingress side the parser receives the ThymesisFlow-P frames and a CRC pass or fail from the Xilinx aurora module. In case a failure is detected, the parser notifies local framer to send a retransmission message to the other side, and in the same spirit delivers a retransmission request from the other side to local framer. The in-band messages are enclosed on single-flit frames and this is how they get classified versus the normal OpenCAPI flit frame traffic. The code of these modules is heavily annotated to explain the various techniques employed.

## 3 ThymesisFlow-P Software design

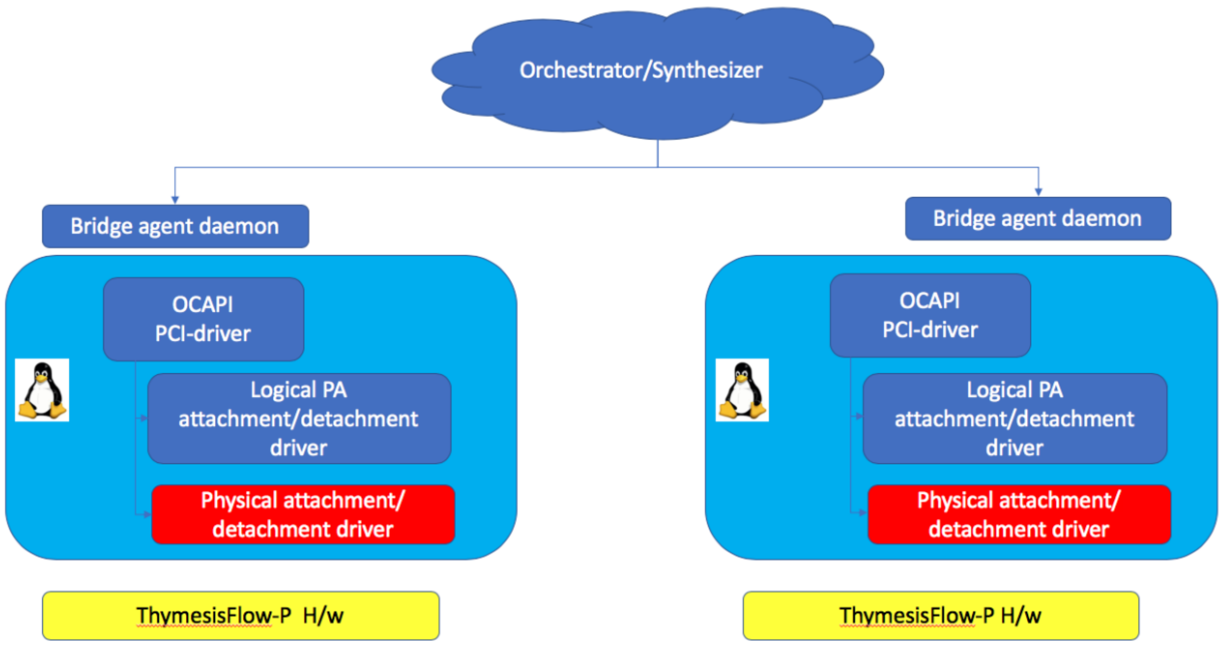

Orchestrator/Synthesizer
Bridge agent daemon
OCAPI PCI-driver
Logical PA attachment/detachment driver
Physical attachment/ detachment driver
ThymesisFlow-P H/w
Bridge agent daemon
OCAPI PCI-driver
Logical PA attachment/detachment driver
Physical attachment/ detachment driver
ThymesisFlow-P H/w

*Figure 11 Thymesis-P Software design*

ThymesisFlow-P is built on top of OpenCAPI software stack. *This section will be expanded in the future when all software matures. Currently refer to the next section (deployment) for instructions on how to bring up and execute a disaggregated memory application.*

## 4 Example Deployment on 2x AC922 with 9V3 FPGAs

To deploy a ThymesisFlow-P example you need to perform the following steps:

1. Make sure that both AC922 have a firmware stack installed that enables OpenCAPI C1 and M1 modes.
2. Make sure you have linux kernel with OpenCAPI support for both modes (ver >= 5.0)
3. Build and Install bitstreams on 9V3 FPGAs and cross connect them.
4. Execute an example libocxl remote memory application – like the appropriately modified stream benchmark.

## 4.1 AC922 firmware stack support

The skiboot code with modification is available at:

Please follow the README instructions of the repository above to build skiboot for Power according to your setup (if you need to cross compile it on an x86 or not). You will need the resulting **skiboot.lid.xz.stb** binary.

The process of installing the new skiboot on AC922 is automated. Clone the openpower python test framework and install it on any x86 or Power machine that can access the AC922 BMC over the network (obviously it should not be the AC922 itself..). You can get op-test from public github here:
https://github.com/open-power/op-test

Once installed, you just need to change to local directory to execute **op-test** binary. First create a text file which you can name for example "config_mymachine". Then edit it as follows

```
[op-test]
bmc_type=OpenBMC
bmc_ip= <ip or hostname of the target P9 bmc>
bmc_username=root
bmc_password=<password or default "0penBmc" – note the first digit is zero>
```

After you save the file you can directly issue the command:

```
user@host$ ./op-test  --config-file=./config_mymachine  --flash-skiboot ./skiboot.lid.xz.stb --only-flash
```

The op-test framework will flash skiboot. Please note the machine will be rebooted and switched off by the script, but it take a bit of time, so it is better that you turn the machine off by issuing:

```
obmcutil poweroff <or chassisoff>
```

at the bmc console.

**If skiboot does not allow AC922 to boot !!!!** log into AC922 bmc as root and navigate to /usr/local/share/pnor . Once you have previously installed skiboot using op-test tool, this folder will contain a file with the name **PAYLOAD**. Delete this file (using rm) and the default factory skiboot will boot instead.

## 4.2 Libocxl

Please download the libocxl code from here:
Use the examples config to configure the *compute* side and *memnode* to steal memory at the memory side. When configuration is done, use the provided python scripts to hotplug memory regions through sysfs.

## 4.3 Design registers

The design supports one context and features a set of registers that configure various functionalities. In the Figure 12 registers and their functionalities are listed. These registers are mapped to design global mmio space.

| Offset/range | Values | | Remarks |
|---|---|---|---|
| 0x0 /[63:0] | {0x0, 0x1} | | C1 design only, when 0x1 initiates capptag fifo and sends ACTAG cmd |
| 0x20/[63:0] | [63:6] EA offset | [5:0] netid | M1 design only configures the 1st entry of the netcmd_fifo.v section classifier (section 0) |
| 0x28/[63:0] | [63:6] EA offset | [5:0] netid | M1 design only configures the 2nd entry of the netcmd_fifo.v section classifier (section 1) |
| 0x30/[63:0] | [63:6] EA offset | [5:0] netid | M1 design only configures the 3rd entry of the netcmd_fifo.v section classifier (section 2) |
| 0x38/[63:0] | [63:6] EA offset | [5:0] netid | M1 design only configures the 4th entry of the netcmd_fifo.v section classifier (section 3) |
| 0x40/[63:0] | [63:6] EA offset | [5:0] netid | M1 design only configures the 5th entry of the netcmd_fifo.v section classifier (section 4) |
| 0x48/[63:0] | [63:6] EA offset | [5:0] netid | M1 design only configures the 6th entry of the netcmd_fifo.v section classifier (section 5) |
| 0x50/[63:0] | [63:6] EA offset | [5:0] netid | M1 design only configures the 7th entry of the netcmd_fifo.v section classifier (section 6) |
| 0x58/[63:0] | [63:6] EA offset | [5:0] netid | M1 design only configures the 8th entry of the netcmd_fifo.v section classifier (section 7) |
| 0x60/[63:0] | Tx flit counter | | It counts outgoing 32B Tx flits and resets to zero upon read |
| 0x68/[63:0] | Rx flit counter | | It counts incoming 32B Rx flits and resets to zero upon read |
| 0x70/[63:0] | [63:37] number of measurements | [36:0] accumulated latency in cycles | This register can be used to get average latency in M1 design cycles. Each time is read it provides the number of measurements along with the accumulated latency cycle so software can perform a division to determine the average. The latency counter calculates the number of cycles that elapse from when a random command exits the net_cmdfifo until the respective response arrives from remot memory to lpc_respfifo.v |
| 0x78[63:0] | {0x0,0x1,0x2,0x3} | | Aurora reset control. Bit[x] when asserted bring out of reset the respective aurora channel and when is deasserted it resets it. Currently the design features 2 aurora channels so only bit positions 0 and 1 are currently used in the 9v3 design. |

*Figure 12 register values and remarks*

netid value is one-hot encoded in this design. When netid[x] of netid is asserted then the corresponding transaction (and it's dataflits when available) should be routed to output port x. The design currently has two ports so only netid[0] and netid[1] are used. When both are asserted the datapath enters bonding mode and routes transactions among both ports in a round robin fashion. When bonding is used all transactions should have netid in bonding mode.

The memory design (C1) is not receiving any routing configuration. It is just designed to send the response to any arriving command transaction to the port from which it arrived.

### 4.3.1 Physical installation of 9V3 on AC922 server

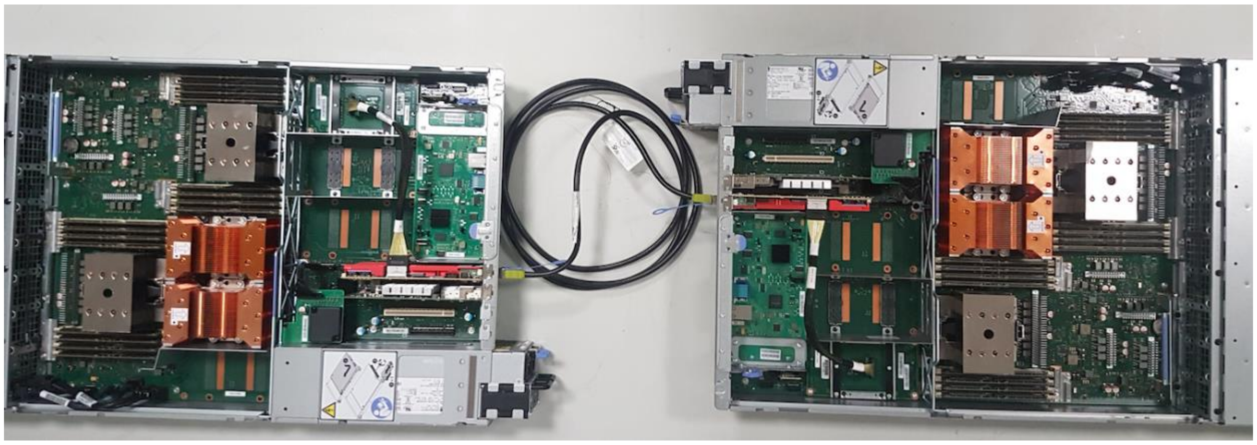


*Figure 13 AC922 physical setup*

As depicted in Figure 13 9V3 FPGAs are placed in the very first pci slot (that is located immediately after the BMC). In addition, via the special Acorn Adapter, and OpenCAPI cable
It occupies the first SXM2 slot of Processor 0. At the time of this writing, skiboot cannot handle GPUs being concurrently installed on the same processor, so they have to be connected to the other processor core as depicted, but this is likely to change in the future.

### 4.3.2 Build the bitstreams and configuration of 9V3

ThymesisFlow has been integrated with OpenCAPIBuilds repo, which is the proper channel to use for building OpenCAPI design and seamlessly incorporating the new functionality.

To build just use vivado toolchain (latest tested at the time of this writing is 2018.2 but should work with no issue with the latest as well).

The options for ThymesisFlow-P are as follows:

| **Parameter** | **Value** | **Remarks** |
|---|---|---|
| --afu | thymesisflow | Thymesisflow afu (mandatory) |

| | | |
|---|---|---|
| --speed | 20.0 or 25.78125 | Dlx transceiver speeds (both choices ok as the design meets timing) |
| --buffer | bypass | (elastic would work but no reason to use it |
| --card | ad9v3 | (mandatory for now) |
| --tftype | compute or memory | If ‘compute‘ is used, M1 side of the design will be built. If ‘memory’ is used C1 side of the design will be built. If nothing used both sides of the design will be built – currently not fully supported. |

Instructions on how to invoke the build command can be found on the README.

When the project is built, activate implementation number 10 (impl_10) to achieve timing closure.

Once the final bitstream is built, It can be either permanently flashed to FPGA or just get downloaded over JTAG while Power9 is booting hostboot. Do not push configurations later, as skiboot is training the OpenCAPI design links, so it does not make any sense to push any bitstream during or after this process. Hostboot bringup operations provide enough time margin to boot configure the 9V3 on the machine.

**Important!!!!**

Regarding the permanent flashing please follow the detailed 9V3 specifications manual. In addition, the very first time of setting a new 9V3 FPGA the GTY reference clock should be set to the proper clock rate: Following the 9V3 instructions here page 16 you can install the USB tools to change the FPGA QSFP28 reference clock. Please issue the following command:

C:\> **avr2util.exe /usbcom com4 setclknv 0 309375000**

This will change the QSFP28 cages reference clock to 309.375 Mhz that is used by the aurora core.