Pyxis Chip Functional Specification

1 SUMMARY OF FEATURES

Feature	Details
Neural Network Accelerator	4 ***Clusters*** per chip, with 12 Universal Inference Engines (UIEs) per Cluster Performance @1.33 GHz for 3x3, 1x1 or MatMul: Per UIE: 44 TFLOPS FP8, 22 TFLOPS FP16 Per Cluster: 525 TFLOPS FP8, Ser 263TFLOPS FP16 Total per chip: 2100 TFLOPS FP8 1050 TFLOPS FP16
	Custom vector processor: 8170 GFLOPS FP8 4085 GFLOPS FP16 4085 GOPS Int16
Accelerator SRAM	256MBytes unified SRAM
HBM DRAM	4 x HBM3E stacks Capacity: Up to 36GBytes per HBM : total 144 GBytes Bandwidth: up to 1229 GBytes/s per HBM : total 4915 GBytes/s
Fabric Interface	36 x 112G SERDES, supporting >400GB/s effective throughput Any-to-any connectivity for up to 144 Pyxis chips using Juniper Cell fabric
On-chip CPUs	16x Si-Five P670 @1.5GHz ISA: RV64I+MAFDCVH+ Z* + S* L1 I$/D$: 32KB/32KB (4-way) L2$: 256KB (8-way) L3$: 16MB (18-way)
PCIE	Gen5 x16
Misc interfaces	I2C (xN), SPI GPIO (xN)
Clock Domains	UIEs & AMEM: 1.2-1.4GHz HBM Fabric Interface CPU complex PCIE complex

2 Block Diagrams

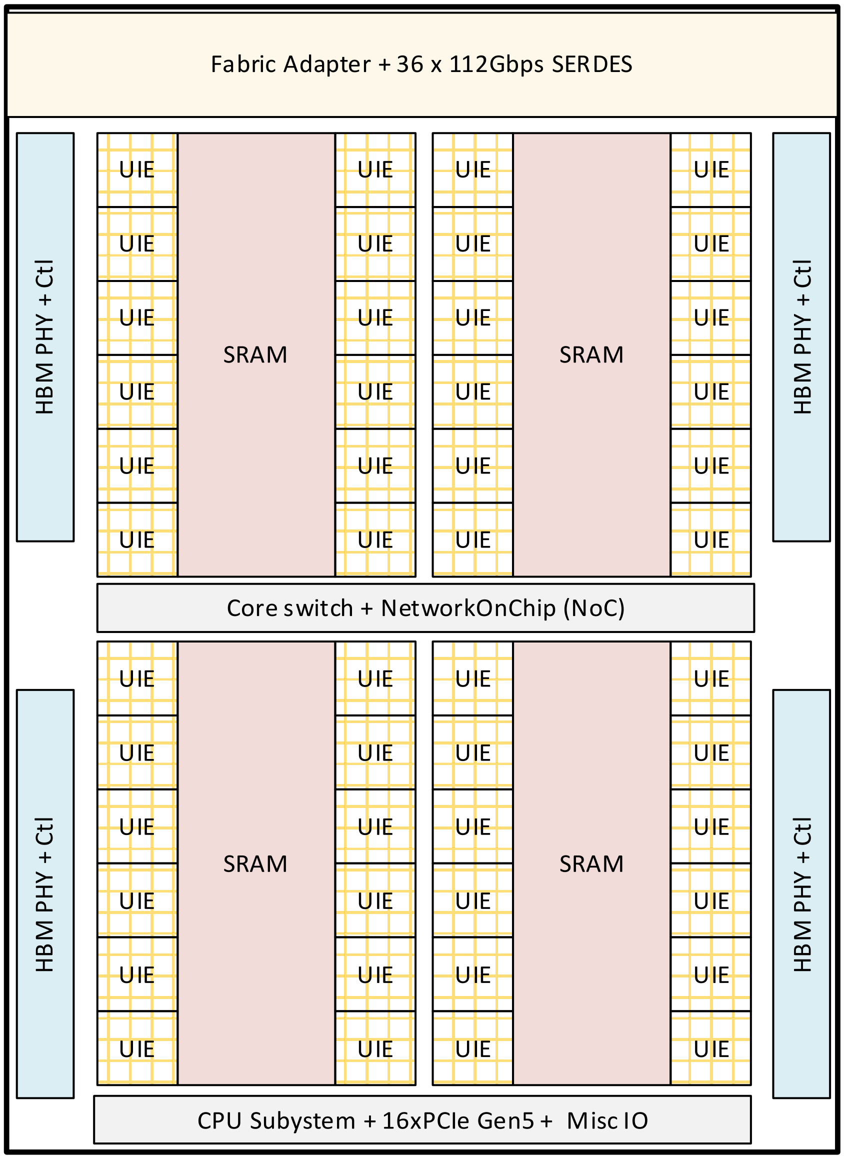

Fabric Adapter + 36 x 112Gbps SERDES
HBM PHY + Ctl
HBM PHY + Ctl
HBM PHY + Ctl
HBM PHY + Ctl
UIE
SRAM
SRAM
SRAM
SRAM
Core switch + NetworkOnChip (NoC)
CPU Subystem + 16xPCIe Gen5 + Misc IO

Figure 2-1: Pyxis Block Diagram

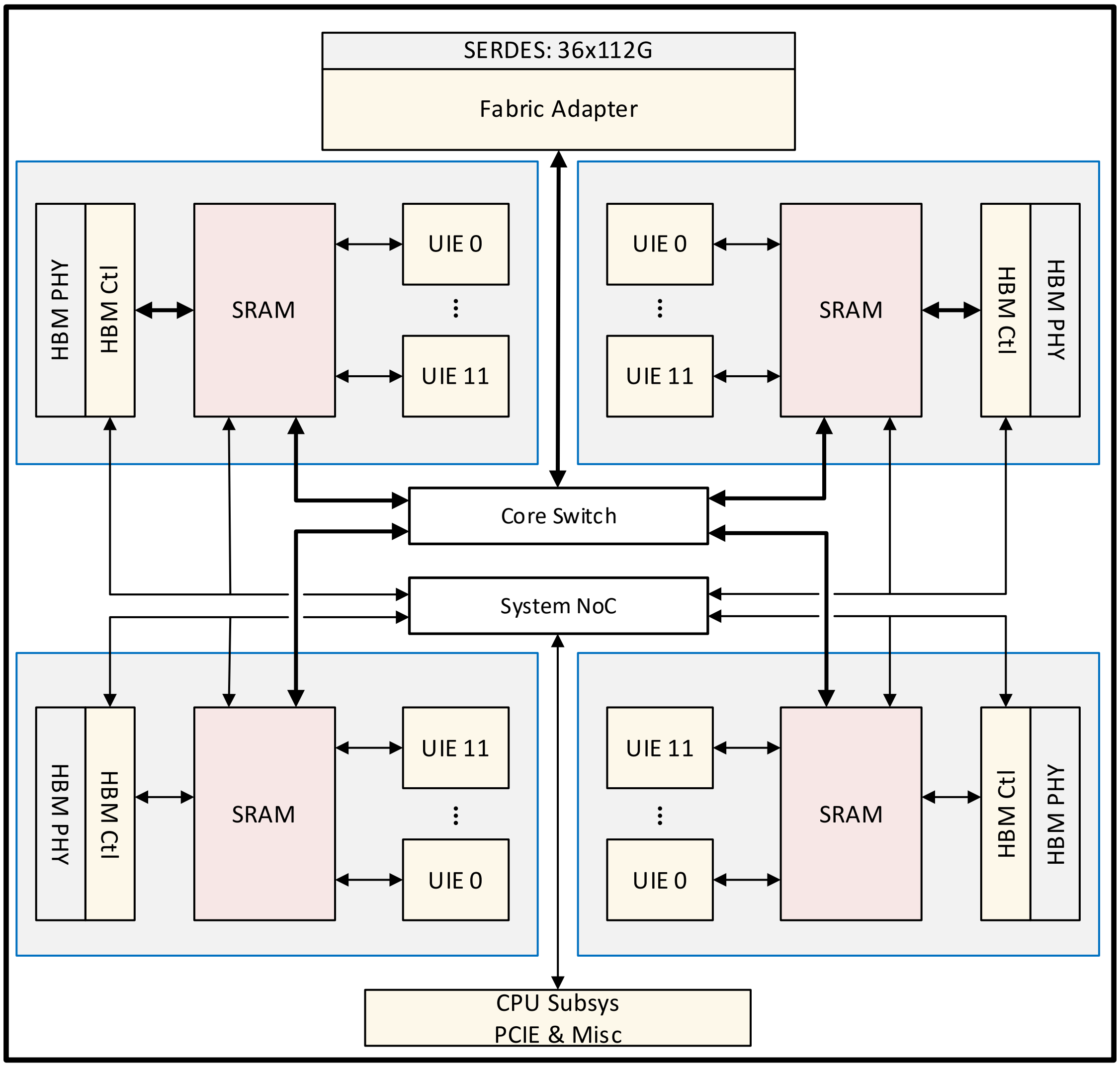


Figure 2-2: Pyxis Data Flow Diagram

3 GLOSSARY

UIE	Universal Inference Engine
UIE Cluster	A block of 12 UIEs, one HBM interface and 64MB of AMEM. There are 4 UIE Clusters per Pyxis
AMEM	Accelerator Memory (SRAM) – a multi-banked, multi-ported SRAM structure providing high bandwidth to the UIEs and HBM interface, as well as supporting inter-cluster and inter-chip communication. There is one AMEM per UIE Cluster.
HBM	High Bandwidth Memory – high speed external DRAM.
NoC	Network on Chip – internal bus network for interconnecting on-chip CPUs, PCIE ports and certain engines to the memories and memory-mapped Control and Status Registers
VPU	Vector Processing Unit. One instance of VPU per UIE
LLC	Leaf Level Convolver. The atomic processing element of the UIE which performs both 3x3 convolutions and general matrix-matrix dot product computations natively
LLC Grid	Grid of 16x128 LLCs forming a systolic array for convolution or matrix multiplication. For Matrix multiplication, this operates as 128x128. One LLC Grid per UIE
LNS	Logarithmic Number System, described as LnsI**x**F**y**, where **x** is number of integer bits, and **y** is number of fraction bits of the exponent value
LLM	Large language Model
Trip	Execution of (typically) one round-trip for data read from AMEM, processed by the UIE, with results written back to AMEM. A trip roughly corresponds to one layer of a neural network (or the portion of a layer executed by one UIE). For convenience, a "Trip" is also used to describe one DMA job for the specialized DMA engines
EB	Exponent Bias. Pyxis floating point and LNS numbers use configurable EBs to scale the representable range of values.
MatMul	Matrix Multiplication for **data x data matrices**. In this document, this is distinguished from **weights x data** matrix multiplication, which is referred to as a 1x1 convolution or linear layer.
DMA	Direct Memory Access: copying data directly from one memory to another via a hardware engine.
ICDMA	Inter-Cluster DMA: copies data from one UIE Cluster's AMEM to another Cluster's AMEM
H2A, A2H	HBM-to-AMEM or AMEM-to-HBM DMA: copies data from HBM to AMEM or form AMEM to HBM (in the same cluster).
RDMA	Remote DMA: a DMA transfer to a remote Pyxis chip. For Pyxis, RDMA transfers are always AMEM-to-AMEM
VOQ, OQ	Virtual Output Queue: a queue used to reach a remote Pyxis chip across the cell-fabric. A VOQ identifies the destination chip, UIE Cluster, and class-of-service. VOQs live on the "ingress" or Tx side of the fabric Output Queue: a target queue on the "egress" or Rx side of the fabric. An OQ is associated with a UIE Cluster and class-of-service.
TJID, RJID	Transmit Job ID & Receive Job ID – used to establish an RDMA communication channel between two Pyxis chips
CSR	Control & Status Register. Software-visible, memory-mapped register to provide control and/or status of the hardware.

4 Data Types

Pyxis Universal Inference Engines (**UIEs**) support several data types optimized for hardware: Recogni-custom `FP8`, `FP16`, and logarithmic variants of these, denoted `LnsI4F3` and `LnsI5F10`. The Vector Processing Unit in each UIE also supports `Int16`.

In the following descriptions, ***saturating maximum*** **values act to clamp overflows, but are not sticky unless stated otherwise**. If a computation results in a value larger than the maximum representable (magnitude), it is clamped to the maximum value for that particular computation. But for subsequent computations using that value, it is simply the maximum value and not sticky or “infinity.” However, the sum of a positive maximum value and negative maximum value (or subtracting two maximum values of with the same sign) will result in `NaN`.

`NaN` (Not A Number) *is* sticky, and any computation with NaN as an input will result in NaN output.

4.1 FP8

The base data format for activations in Pyxis hardware is Recogni-custom **FP8**:

(1)	(4)	(3)
S	E	F
sign	Exponent	Mantissa (Fraction)

Value is `(-1^S)* 1.F * 2^(E+ExponentBias).` Exponent Bias is a user-defined integer value and described in section 4.5 below. EB is usually a negative value.

The following FP8 codepoints are reserved for special purposes:

S	**E**	**F**	**value**
0	0000_2	000_2	zero
1	0000_2	000_2	NaN
0	1111_2	111_2	saturating maximum positive value
1	1111_2	111_2	saturating maximum negative value

Note that Pyxis FP8 does not support subnormal values.

4.2 FP16

For higher precision, Pyxis supports a customized version of FP16 (not fully IEE754 compliant) with the following format

(1)	(5)	(10)
S	E	F
sign	Exponent	Mantissa Fraction

Value is `(-1^S)* 1.F * 2^(E+ExponentBias).` Exponent Bias is a user-defined integer value and described in section 4.5 below. EB is usually a negative value.

The following FP16 codepoints are reserved for special purposes:

S	**E**	**F**	**value**
0	00000$_2$	00_0000_0000$_2$	zero
1	00000$_2$	00_0000_0000$_2$	NaN
0	11111$_2$	11_1111_1111$_2$	saturating maximum positive value
1	11111$_2$	11_1111_1111$_2$	saturating maximum negative value

Note that Pyxis FP16 does not support subnormal values.

Computations with FP16 **operate at half of the execution rate of FP8**, since it takes twice as long to access the 2-bytes-per-element tensor data from memory.

Data inputs to the UIE can be either FP8 or FP16, and the output written back may use either format. This may be selected on a per layer or even per-channel basis. The input & output formats are independently configured. However, the output format from one layer (FP8 or FP16) must always be chosen as the input format for the next layer.

4.3 LNS I4F3

Weights used for convolution in Pyxis hardware are stored in Logarithmic Number System (LNS) form, with **LnsI4F3** as the baseline type. This is the logarithmic-encoded counterpart to FP8.

(1)	(4)	(3)
S	I	F
sign	Exponent integer	Exponent Fraction

Value is `(-1^S)*2^(I.F + ExponentBias)`

The S (sign) bit applies to the value in linear space, for representing $\pm 2^{E}$.

The exponent has integer and fractional parts, representing $2^{(Exp\_int.Exp\_fract)}$. Since this is a logarithmic format, successively numbered values have a quantized *ratio*, not a quantized *difference*. With 3 bits of exponent fraction, the ratio between successive values is $2^{1/8}$: 1, approximately 1.09:1, or 9% ratio between adjacent values.

Exponent Bias is a user-defined integer value and described in section 4.5 below. EB is usually a negative value.

LnsI4F3 uses the same reserved codepoints as FP8, though substituting "I" and "F" fields for FP's "E" and "F" fields:

S	I	F	value
0	0000_2	000_2	zero
1	0000_2	000_2	NaN
0	1111_2	111_2	saturating maximum positive value
1	1111_2	111_2	saturating maximum negative value

NOTE: weights do **not** support NaN values, and the NaN codepoint must not be used for weights

4.4 Extended LNS I5F10

For higher precision internal representation of FP8 or FP16 data, Pyxis supports 16-bit ***LnsI5F10***, or ***LNS16*** for short, and is the logarithmic-encoded counterpart to FP16.

(1)	(5)	(10)
S	I	F
sign	Exponent integer	Exponent Fraction

The LNS16 format is **ONLY** used internally in the LLC Grid, or for higher precision weights. Linear FP8/16 numbers are converted into LNS16 automatically by the hardware before the data is forwarded to the LLC Grid. The log-domain representations are then used for multiplication in the LLCs.

LnsI5F10 uses the same reserved codepoints as FP16, though substituting "I" and "F" fields for FP's "E" and "F" fields:

S	I	F	value
0	00000$_2$	00_0000_0000$_2$	zero
1	00000$_2$	00_0000_0000$_2$	NaN
0	11111$_2$	11_1111_1111$_2$	saturating maximum positive value
1	11111$_2$	11_1111_1111$_2$	saturating maximum negative value

4.5 Exponent Bias

Recogni Floating Point formats uses a biased exponent. The “*true”* exponent value is obtained by adding a signed-integer *Exponent Bias* (**EB**) to the nominal exponent value stored in memory:

$$E_{true} = E_{nominal} + EB.$$

or

$$E_{nominal} = E_{true} - EB$$

For Recogni **FP8**, EB is used to scale the *true* values to be representable within the *nominal* exponent range [0, 15]. Unlike IEEE754 floating point numbers, the EB is not fixed, and may be set differently for each layer in a network. For example, if the desired range of *true* exponent values for FP8 is [–12, +3], we set EB to -12.

For Recogni **FP16**, the nominal exponent range is [0,31], and the EB range is similarly adjustable. For example, if the desired range of true exponent values for FP16 is [-16,+15], we set EB to -16. Note that the equivalent to the IEEE754 FP16 standard EB is -15.

Similar math applies for weights encoded in LnsI4F3 format. EB is used to scale the *true* values to be representable within the *nominal* integer-exponent range [0, 15]. For example, if the desired range of *true* exponent-integer values for LNS is [–12, +3], we set EB to -12. Similarly, LnsI5F10 has the same EB range and interpretation as FP16

When converting between FP8 and FP16 (or between any of the Pyxis types with 4-bit and 5-bit exponent integers), it is necessary to adjust the (nominal) Exponent when changing the EB. These type conversions are performed automatically by hardware, but the type converters must be programmed with EB Adjustment values which effectively scale the values to “fit” in the target type dynamic range. This is described in detail for each type-converter in the Universal Inference Engine (UIE) sections below, and summarized in Section 6.5.

4.6 MNS 5.2.N

Accumulation in the LLC Grid uses another, slightly different custom Floating Point data type, referred to as MNS. This data type has a 5-bit exponent, and encodes the mantissa in 2’s complement format.

The MNS format is as follows. The logical representation includes 2-bit mantissa.integer field:

(5)	(2)	(F=13-18)
E	I	F
Exponent	Mantissa integer	Mantissa Fraction

The hardware format drops one redundant mantissa bit, for a **5.1.N** format:

(5)	(1)	(F=13-18)
E	S	F
Exponent	Mantissa Sign	Mantissa Fraction

The effective value is

```
X_MNS = Mantissa * 2^(Exponent+EB_MNS)
```

There are two MNS formats used within the LLC Grid, corresponding to the two-stage accumulator scheme. The first, “active” accumulator uses F=13, or 5.1.13; the second accumulator stage uses F=18, or 5.1.18 format.

The MNS exponent bias is determined by adding the EB’s of the two operands (e.g., weights and data) used to form the products, and then adjusted by 16 (in the LLC multipliers) to avoid exponent-overflow in the intermediate results.

```
EB_MNS = EB_data1 + EB_data2 + 16
```

The specific bit fields are as follows

- MNS mantissa is a ***signed*** value, using **2’s complement** format. This adds one sign bit to the integer part of the mantissa, so there are logically two integer bits in the mantissa: effectively a sign-bit and a leading one (for positive) or a leading zero (for negative).
- MNS values are normalized before being stored in the accumulator. Normalization of a 2’s complement number requires finding the first 1 (if the number is positive) or the first 0 (if negative) from the left, shifting to place that in the 2^0 position, and adjusting the exponent accordingly.
- We can then ***discard the leading bit in the 2^0 position*** for savings in the accumulator since this is implied and equal to the inverse of the sign bit. Hence, we store only one bit for mantissa.integer, i.e., the sign bit.

The 2’s complement format for MNS does not have a “negative zero”, so the NaN and saturating-maximum values are encoded differently from the FP8/16 and corresponding LNS formats. The reserved codepoints for MNS (hardware format) are as follows for all sizes of MNS F bits:

S	E	F	value
0	00000_2	$0000\_...\_0000_2$	zero
0	11111_2	$1111\_...\_1111_2$	**sticky** maximum positive value, equal to $1.11\_1111\_1111_2 * 2^{31}$ nominal
1	11111_2	$0000\_...\_0001_2$	**sticky** maximum negative value, equal to $-(1.11\_1111\_1111_2 * 2^{31})_{\text{nominal}}$
1	11111_2	$0000\_...\_0000_2$	NaN (overloads $-2.0 * 2^{31}$)

Note that $-(1.0 * 2\hat{\ }0)_{\text{nominal}}$ cannot be represented in MNS because we store only the sign-bit, and `-1.0 * 2^E` is actually stored as `-2.0 * 2^(E-1)` in 2's complement form. This is impossible for E=0. This will result in an underflow to zero.

The normalization step may result in exponent underflow or overflow.

- In the case of underflow, the result is converted to zero.
- In case of overflow, the exponent and mantissa are **sticky** at the maximum positive or negative values in the table above. The maximum-magnitude values are "sticky" during accumulation, so adding or subtracting any weights*data values to this sticky value in the accumulator will not change the accumulator, which will remain at the maximum value.

4.7 Log to Linear conversion

Converting from log (LNS) to the linear domain (floating point), uses the approximation:

$$2^x \approx 1 + x + \mathrm{delta}(x)$$

where `delta(x)` is a "mapping correction", in the form of a four-piece linear approximation:

```
delta(x)  = -x/4              for x ∈ [0, 0.25)
          = -(12x + 5)/128    for x ∈[0.25, 0.5)
          =  (8x - 15)/128    for x ∈[0.5, 0.75)
          =  (9x - 9)/32      for x ∈[0.75, 1]
```

Given an LNS number `X_lns` with (exponent) value `I.F`, we convert to linear as

```
X_lin  = 2^X_lns
       = 2^(I.F)
       = 2^(I)* 2^(F)
       ≈ 2^(I)* (1+F+delta(x))
```

This math is carried out with 10 bits of precision for the result, consistent with FP16/LNS16 fraction sizes. The maximum relative error in this mapping is < 1%.

Hardware also provides the option to disable the mapping-correction (delta(x)) or to clip the resulting value to 3-bits of fraction. See 6.2.4.4, Linear-to-Log conversion for Horizontal MatMul Data

4.8 Linear to Log conversion

To convert from FP8/16 (linear) to LNS, we apply a log2 approximation which is the inverse of the log-to-linear mapping described above:

```
log2(1+x) ≈ x – delta'(x)      for x ∈[0,1]
```

where delta'(x) is the inverse "mapping correction" as follows:

```
delta'(x)     = -x/3                for x ∈[0, 3/16)
              = -(12x + 5)/116      for x ∈[3/16, 53/128)
              =  (8x - 15)/136      for x ∈[53/128, 87/128)
              =  (9x - 9)/41        for x ∈[87/128, 1]
```

Then, given a linear number `X_lin`, in the floating-point format `1.f * 2^e`, we apply the mapping:

```
X_lns         = log2(X_lin)
              = log2(1.f * 2^e)
              = e + log2(1+f)
              ≈ e + f – delta'(f)
```

This math is carried out with 10 bits of precision, consistent with FP16/LnsI5F10 fraction field widths.

4.9 IMPORTING/EXPORTING IEEE-754 FP16

Recogni FP16 ("RFP16") internal data format differs from IEEE-754 FP16 in a few details. When importing IEEE FP16 data from PCIE or the on-chip CPUs to UIEs, it is first necessary to convert to RFP16 before we can do any computations. Similarly, when exporting RFP16 to CPUs or over PCIE to any external processors, we must first convert back to IEEE FP16.

These conversions may be performed **in the UIEs when reading from AMEM SRAM (for IEEE->RFP16) and when writing to AMEM SRAM (for RFP16->IEEE)** under program control. The following sections list the details of these conversions.

4.9.1 Importing from IEEE FP16

This is an option when reading data from AMEM into the UIEs:

- If the IEEE FP16 value = +/- zero, convert to RFP16 zero
- If the IEEE FP16 value = +/- NaN, convert to RFP16 NaN.
- if the IEEE FP16 value = +/- infinity:
    - If using saturating mode, convert to RFP16 +/- max
    - If using non-saturating mode, convert to RFP16 NaN

  Saturating/non-saturating mode is a run-time configuration.

Otherwise:

- Adjust the exponent to convert from IEEE FP16 standard exponent-bias (-15) to the desired RFP16 exponent bias . This is under software control to program the run-time EB adjustment.

- Normalize the output of the exponent adjustment. If the result (magnitude) is smaller than the minimum normal RFP16, set the result to zero, since RFP16 does not support subnormal values. (Note however, that properly configuring the exponent-bias for RFP16 can avoid this conversion underflow.)

- If the output of the above is larger than the maximum normal RFP16 value, set to +/- max.

4.9.2 Exporting to IEEE FP16

This is an option when writing data from the UIEs to AMEM:

- If the RFP16 value = zero, convert to IEEE FP16 +zero
- if the RFP16 value = NaN, convert to IEEE FP16 +NaN.
- If the RFP16 value = +/- max, optionally convert this to IEEE +/- infinity. (This option is under CSR control). Otherwise, treat RFP16 +/-max as a normal value before conversion as follows:

Otherwise:

- Adjust the exponent to match the IEEE FP16 standard exponent-bias for = -15. This is under software control to program the run-time EB adjustment.

- If the EB-adjusted value (magnitude) is smaller than the minimum normal value, adjust to use subnormal values if possible. If smaller than the minimum subnormal, set the result to **+/-** zero.

- If the EB-adjusted value (magnitude) is larger than the maximum IEEE FP16 normal value, set the result to +/- infinity.

4.10 Importing/Exporting OCP FP8

Recogni FP8 (“RFP8”) internal data format differs from OCP FP8 (“OFP8”) specification[1] in a few details, and OFP8 cannot be processed natively by UIEs. Similar to the IEEE F16 conversions discussed above, conversion may be performed in the UIEs when reading from AMEM (import OFP8) and when writing to AMEM (export OFP8). Both of OFP8 **E4M3** and **E5M2** formats are supported, and both saturating and non-saturating conversion modes are supported by run-time configuration.

4.10.1 Importing from OFP8

The UIEs use Recogni FP16 (RFP16) internally, and may be programmed to convert from **OFP8** to **RFP16** when reading data from AMEM. Both of the OFP8 E4M3 and E5M2 formats are supported. The recipe is as follows:

- If the OFP8 value = +/- zero, convert to RFP16 zero
- If the OFP8 value = +/- NaN, convert to RFP16 NaN.
- if the OFP8 value = +/- infinity (E5M2 only)
    - If using saturating mode, convert to RFP16 +/- max
    - If using non-saturating mode, convert to RFP16 NaN

  Saturating/non-saturating mode is a run-time configuration.

Otherwise:

- Adjust the exponent to convert from OCP FP8 standard exponent-bias (-7 for E4M3 or -15 for E5M2) to the desired RFP16 exponent bias. This is under software control to program the run-time EB adjustment.

- Normalize the output of the exponent adjustment. If the result (magnitude) is smaller than the minimum normal RFP16, set the result to zero, since RFP16 does not support subnormal values. (Note however, that properly configuring the exponent-bias for RFP16 can avoid this conversion underflow.)

- If the output of the above is larger than the maximum normal RFP16 value, set to +/- max.

[1] OCP 8-bit Floating Point Specification (OFP8) Revision 1.0, June 20, 2023

4.10.2 Exporting to OFP8

The UIEs use Recogni FP16 (RFP16) internally, and may be programmed to convert **RFP16** to **OFP8** when writing data to AMEM. Both of the OFP8 E4M3 and E5M2 formats are supported. The recipe is as follows:

- If the RFP16 value = zero, convert to OFP8 +zero
- if the RFP16 value = NaN, convert to OFP8 +NaN.

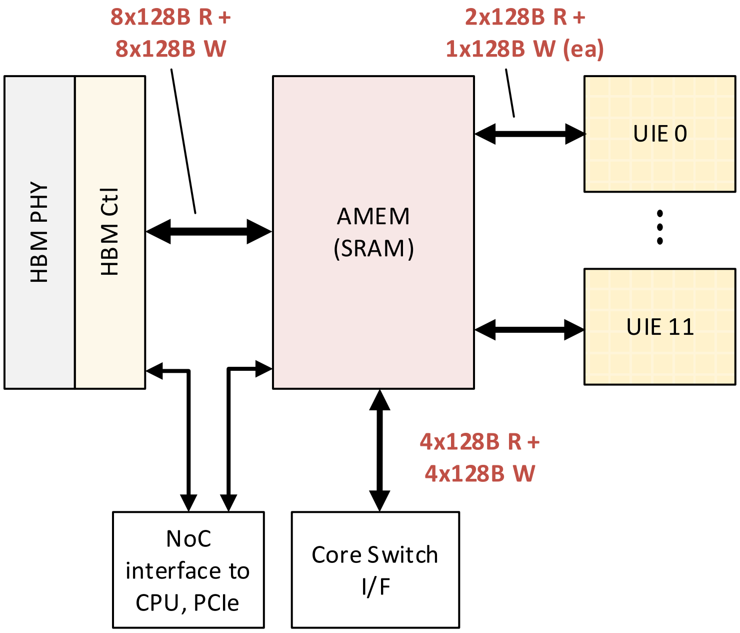
Otherwise:

- Adjust the exponent to match the OCP FP8 standard exponent-bias for -7 for E4M3 or -15 for E5M2. This is under software control to program the run-time EB adjustment.
- If the resulting value (magnitude) is smaller than the minimum normal value, adjust to use subnormal values if possible. If smaller than the minimum subnormal, set the result to **+/-** zero.
- After possibly aligning the mantissa for subnormal values, round the mantissa from RFP16 (E5M10) to OFP8 E4M3 or E5M2. This uses a round-to-even policy.
- If the post-rounding/EB-adjusted value (magnitude) exceeds the maximum OFP8 normal value :
    - If using saturating mode, convert to +/- maximum normal OFP8.
    - if using non-saturating mode:
        - E4M3 target: convert to +/- NaN
        - E5M2 target: convert to +/- infinity

      Saturating/non-saturating mode is a run-time configuration.

5 Major Functional Blocks

5.1 UIE Cluster



Each UIE Cluster includes

- 12 Universal Inference Engines (**UIE**s). Each UIE includes:
  - 16 row x 128 column LLC Grid
  - 1 x 128 Vector Processing Unit
  - 8x128x128 Vector Buffer memory
- Accelerator Memory (**AMEM**), 64 Mbytes of SRAM
- HBM3E DRAM interface and controller
- Core-switch interface for inter-cluster and inter-chip DMA transfers
- System Network-on-Chip (**NoC**) interface for CPU and PCIE access to HBM DRAM

The UIEs are designed to execute complex neural networks including convolutional neural networks, transformers, and general matrix multiplication tasks.

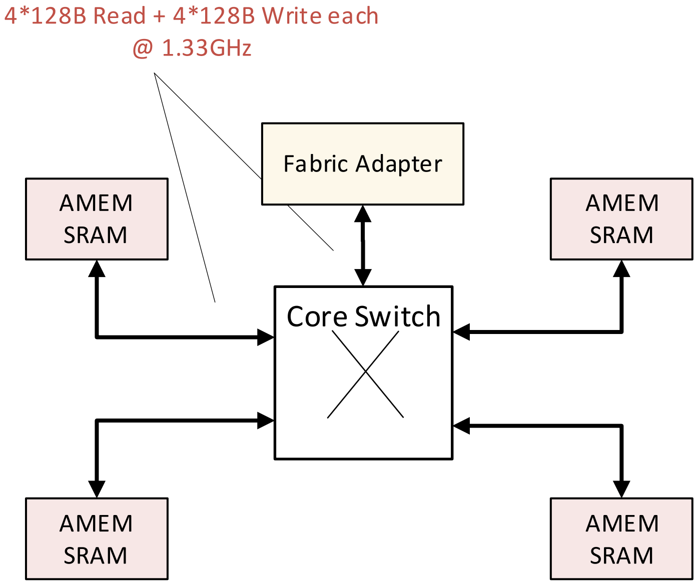
Execution of a neural network inference starts by transferring the input data (image or token embeddings) and weights from HBM into AMEM via the DMA controller. Input data may be received from the fabric or PCIe (outside the cluster) and DMA'ed into the cluster's AMEM or HBM. The network layers are executed as a series of ***trips*** through the UIEs, where each trip is roughly equivalent to a network layer or node in a network graph. The tensor processing may be tiled, and the tiles distributed among the UIEs.

Execution of each trip (or network layer) may include loading trip microcode into the UIEs, reading the weights and tensor data from AMEM, executing the specific task (e.g., matrix multiplication or convolution), and writing the resulting tensor data back to AMEM. These tasks are all executed by hardware under software supervision. A single network can consist of hundreds of trips in series (per UIE) and distributed across the UIEs. For parallel operation, data can be shared among the clusters by transferring from AMEM to another cluster's AMEM under DMA control; data may also be shared with other Pyxis chips by transferring to/from the clusters via the Fabric Adapter.

5.2 Core Switch

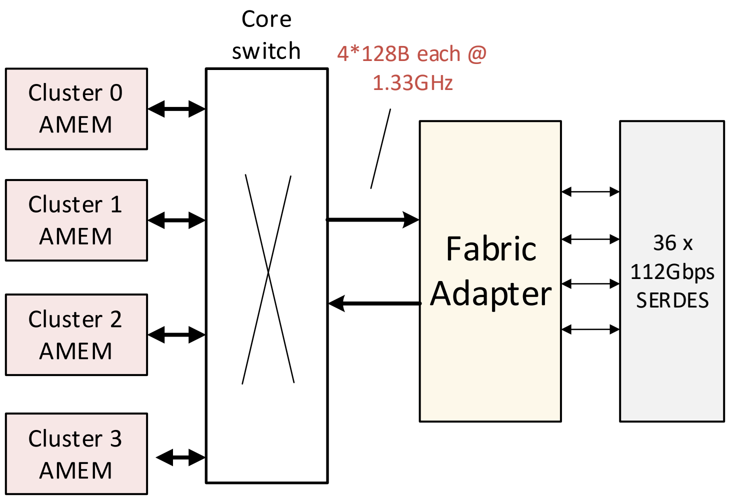
Pyxis enables high-bandwidth data exchange among the four UIE Clusters, and the Fabric Adapter, via a Core Switch. This is a buffered crossbar to transfer data from AMEM in any cluster to AMEM in any other cluster. There are 4 x 128Byte ports to/from each cluster, and 4 x 128B ports to/from the Fabric Adapter. At 1.33GHz, this supports up to 680 GBytes/s to/from each cluster or the fabric.

Transfers from cluster-to-cluster are executed by the Inter-Cluster DMA engine, which can be programmed to transfer data blocks from any Cluster AMEM to any other (but not to itself). Transfers between any Cluster AMEM and the cell fabric interface are controlled by the Remote DMA engines in the Fabric Adapter.



5.3 FABRIC ADAPTER

Pyxis provides 36x112Gbps SERDES to a cell-based switch fabric, and the fabric-interface controller to interoperate with the Juniper “BF” Fabric chips. This can carry over 400GB/s of payload traffic (in each direction) to and from the fabric. The transfers are divided into 4 lanes of ~100GB/s each, with each lane typically corresponding to one UIE Cluster.



The Fabric Adapter provides a reliable RDMA-write service. The RDMA copies data directly from memory (SRAM) on one Pyxis into memory (SRAM) on another Pyxis chip, and provides a lossless “reliable” transport service, including selective retry for the infrequent case of an error in communication. This supports low-latency for collective data-sharing applications, such as `all_reduce`, `reduce_scatter`, and `all_gather`.

The Fabric Adapter also supports a datagram service for sending data or messages from a transmit queue (on a source chip) to a receive queue (on a destination chip). These queues are software managed. This is not a lossless “reliable” transport, and allows for dropped datagrams; the CPUs on each end are then responsible for implementing a reliable transport layer in software.

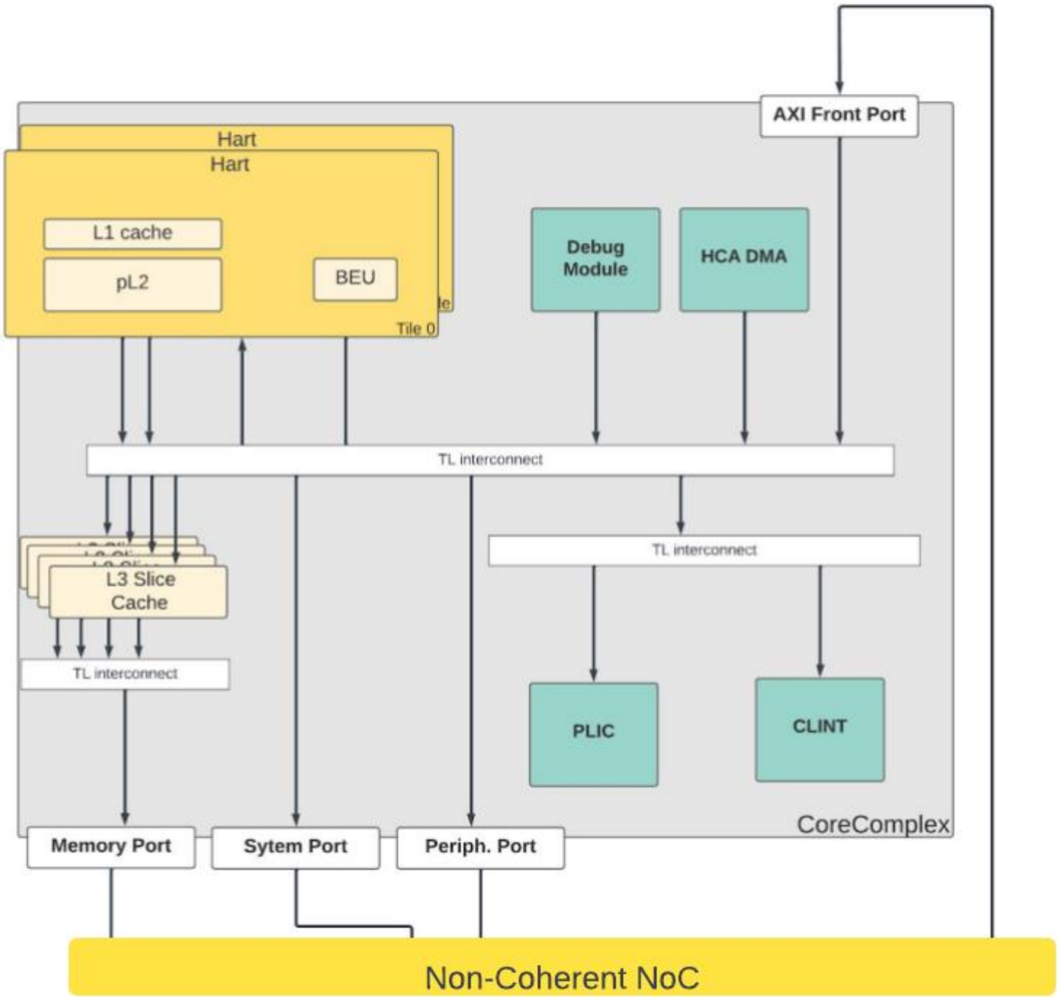
5.4 CPU SUBSYSTEM

5.4.1 Specifications

Item	Description
Core type/Version	Si-Five P670
# of Cores	16 core CPU cluster ~ 4x(4-core/micro-cluster)
Coherency	All cores within CPU cluster are coherent up to the L3$
ISA	RV64I+MAFDCVH+ Z* + S*

L1 I$/D$	32KB/32KB (4-way) with ECC
L2$	256KB (8-way) with ECC
L3$	16MB (18-way) with ECC
MMU	Supported
Interrupts	AIA
Modes	User, Machine, Supervisor, Hypervisor
Clock Frequency	Core clock 2GHz

5.4.2 Block Diagram



5.4.3 Port Definitions

Item	Description
Memory Port	2 AXI-4 ports each port 32B wide
System Port	1 AXI-4 port 16B wide
Front port	1 AXI-4 port 16B wide
Peripheral port	1 AXI-5 port 8B wide

6 UNIVERSAL INFERENCE ENGINE (UIE)

Pyxis Universal Inference Engines are tuned for neural network inference workloads, and are equally adept at convolution, matrix multiplication, or transformer applications, hence “universal”.

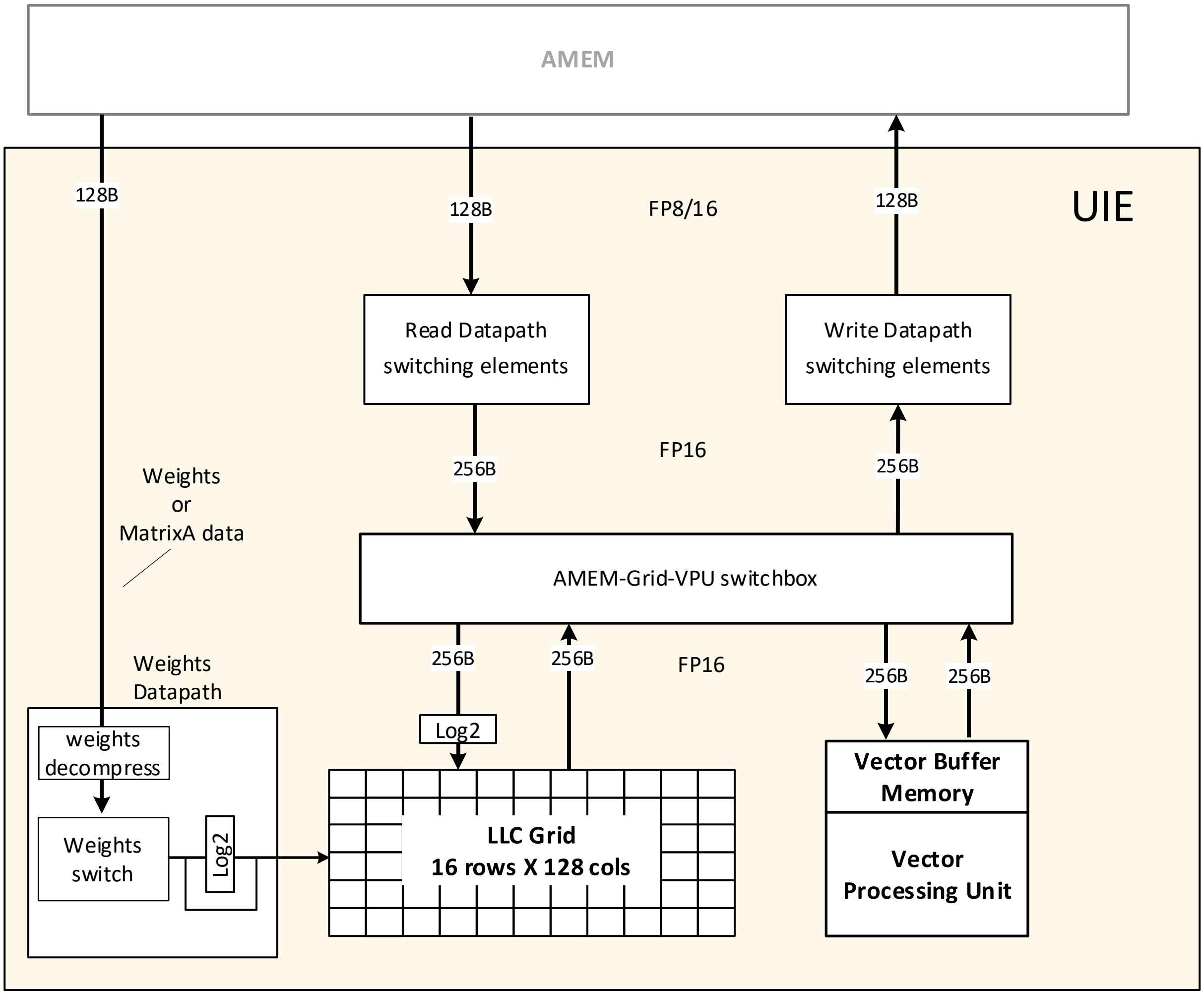


Figure 6-1: UIE Overview

6.1 UIE Theory of Operation

The Pyxis Universal Inference Engines (**UIE**s) execute neural networks as a sequence of ***trips***, where each trip comprises reading a tensor (or one tile of a tensor) from AMEM, executing an operation such as 1x1 convolution[2] on that tensor (or tile) and writing the results back to AMEM, performing one round-trip from AMEM and back. In the simple case, one trip can correspond to one network layer. For some operations, multiple layers can be fused together in a single trip – the most common case is fusing a Conv2D or MatMul with an activation function such as ReLU or GeLU in the same trip. In other cases, a single network layer may require multiple trips, such as for processing "wide" tensors with spatial-width greater than the UIE aperture of 128 columns; in that case, a tensor is processed in tiles of 128 columns, with one trip per tile.

For each trip, the hardware is programmed to read each tensor data-row from AMEM, stepping through the rows in a prescribed order, at up to 128 bytes/cycle. In the nominal case, one row of a tensor is stored in each 128-column word of AMEM, though for wider tensors, this may be tiled to process only 128 bytes per row, and for narrower tensors, we may read multiple rows concurrently, up to 128 bytes per cycle.

The data rows read from AMEM are streamed through the UIE **Read datapath** to align data to the computational elements. Computation flow depends on the type of operation being performed:

- For **3x3 Convolution**: data-rows then stream through the array of **Leaf Level Convolvers** (**LLC**s); these are arrayed in a grid structure referred to as the **LLC Grid**. The tensor data enters the LLC Grid from the "top", while the weights enter on the "side" as drawn in the figure. Each LLC Grid-row implements one or more filters to natively execute 3x3 convolutions, with multiple accumulators in each LLC. This allows for streaming a block of contiguous input tensor-rows for one input channel, and compute a block of contiguous output tensor rows, while holding the 3x3 kernel weights steady for the duration of that block.
- For **Matrix Multiplication or 1x1 Convolution**: there are two read ports on AMEM for each UIE. One port will read the 1st matrix operand in blocks of 128 rows x 128 bytes/row, and feed these in from the "side" of the LLC Grid one column at a time. This first operand may be either weights (for linear layers or 1x1 convolutions) or activation data (for data x data MatMuls). The other AMEM port reads the 2nd matrix operand, 128 bytes (1 row) per cycle, and feeds this data from the "top" of the LLC Grid.

  This implements a 128 x 128 *output stationary systolic array*, where each LLC computes the dot-product for its respective row & column of the output matrix. Since there are only 16 grid-rows to compute the 128 output rows of the matrix, each grid-row is multiplexed to operate as **8 *virtual* grid-rows**, or equivalently 8 virtual output-channels. This is described further in section 6.1.3 below.
- For many types of **element-wise tensor operations**, data may instead be directed from the UIE datapath to the **Vector Processing Unit**, a SIMD processing array with 128 ALU elements (one per column). This processor supports a rich set of math operators including

[2] 1x1 convolutions are exactly equivalent to linear layers performing weights x data matrix multiplication

addition/subtraction, multiplication/division, logical operations, and lookup-table based transforms to execute any arbitrary function of one variable. The Vector Processing Unit also supports more complex tasks such as SoftMax or pooling operations through programming.

As the results are produced from the LLC Grid or Vector Processing Unit, the results are streamed through the **Write datapath** to align data to the memory. Data is then written back to the AMEM, up to 128 bytes per cycle.

The following subsections provide a deeper dive into each of these operational flows.

6.1.1 3x3 Convolution Flow

The UIEs natively execute 3x3 convolutions in a single pass, employing an array of processing elements which computes 3x3 dot-products of 9 Activation and 9 Weight values, performing one 3x3 dot product per cycle. These processing elements are referred to as Leaf Level Convolvers or **LLC**s, and the 16x128 array of LLCs is referred to as the **LLC Grid**.

The processing flow is summarized as follows:

1. Activation data is read from AMEM row-by-row, in "**row-groups**" of 8 rows + 1 overlap row from each input channel. Up to 128 bytes are read per cycle, every cycle. Data is converted from FP8/16 into LNS format just before handing it off to the grid, since the grid wants log-domain data to perform the multiplication step. This linear-to-logarithmic mapping is described in section 4.8 above

2. The data is sent through the Read datapath to the LLC Grid.

    - In the simple case, the Read datapath just passes the data through to the LLC Grid.

    - For "narrow" tensors where the width (in columns) = 128/N, the AMEM read will fetch at most 128/N columns at a time, and the Read datapath will duplicate the data spatially, N times across the width of the datapath, to allow more concurrent execution using this data. (N is restricted to a power of two, with maximum value of 8). For example, if the tensor row width = 64, this can be duplicated 2x across the 128 columns of the datapath.

    - For "wide" tensors where width > 128 columns, the convolution is processed in multiple overlapping tiles. The overlap requires reading data which is not aligned to a 128-column boundary. The Read datapath will rotate the data to re-align the columns to the grid, with 16-column granularity

3. The LLC Grid is organized such that each grid-column corresponds to one column of the output to be computed, and each grid-row corresponds to one or more output channels. This differs from many systolic arrays where the array rows correspond to output tensor data-rows. In our implementation for 3x3 convolution, LLCs in a particular grid-row compute *all* the output data-rows for a given output channel, over time.

4. As the data arrives from AMEM and the Read datapath, each LLC in each column collects 9 data elements – three from consecutive data-rows for its column, and three from the adjacent east and west columns, to form the 3x3 data inputs.

5. Concurrently, the 3x3 kernel weights are streamed from AMEM into buffer memories adjacent to each grid-row, and relayed to each of the LLCs. Weights are stored in LNS format.

6. Each LLC computes the dot-product of the 9 data elements with the 9 weights, and stores the sum in an accumulator *slot*. Each cycle, the incoming data shifts down by one data-row, and the LLC can compute the 3x3 dot product for the next output data-row. This is repeated for 8 output rows, each cycle computing a partial sum for the next sequential output data-row and storing that in a separate accumulator slot. Each 3x3 dot-product is computed as follows:

    - multiply the weights x activations in the log domain by adding the LNS values, since log(A*B) = log(A) + log(B).

    - convert the nine products to a floating-point format in the linear domain, using the log-to-linear conversion method described in section 4.7 above

    - Align the mantissas of the 9 products and the accumulated value to a common exponent, and add the 10 values to form a new partial sum

    - Normalize the sum and store in the **MNS** format in the accumulator. MNS is a custom floating-point format used for accumulation in the LLCs, described in section 4.6 above.

7. After 8 partial-sum values have been accumulated (for 8 output data-rows), the AMEM read switches to the next input channel, and the same subset of input rows (but from the next input channel) are transmitted to the LLCs. The next input-channel's 3x3 kernel is read from the local weight buffer memories adjacent to each grid row, and transmitted into to the LLCs on that grid-row. This kernel update can be done once per 8 cycles for the entire grid.

8. When switching to the next channel, it takes 3 cycles to collect the first input data rows in the LLCs, so there are two "bubble" cycles required (with no computation performed) at the start of each set of 8 partial computations. Hence, each set of 8 computations occurs over 10 cycles.

9. Again, the dot-products are computed with the 3x3 kernel, and the partial sums continue to be accumulated using the same 8 accumulator slots.

10. This process repeats for each input channel, 8 data-rows at a time. The computations take place in a pipelined but concurrent manner over all 16 grid-rows, computing 16 output channels concurrently in a nominal case.

11. When all the input channels have been processed, the accumulation is complete for the current group of 8 output data-rows. The final accumulated sum is stored in a second set of 8 accumulator slots which are reserved for completed results. These 8 "writeback" accumulator slots are subsequently written back to the AMEM, from each of the 16 grid-rows in the LLC Grid. This writeback transfer can occur in the background while the primary "active" accumulator slots are re-used for computing results for the *next* set of data rows.

12. Data sent back from the LLC Grid is converted from the accumulators' MNS format to FP8 or FP16. The data can optionally be routed to the VPU to perform some fused operations, most notably an Activation Function (such as ReLU, etc.), then back to the Writeback datapath. The Writeback datapath performs data switching and alignment as needed, and the results are then written to AMEM.
13. The next row-group of 8 output data rows is then computed, processing the *next* 8 input data-rows from every input channel. Two overlapping rows are also required (so it takes 10 input rows to compute 8 outputs, as noted above). This next row-group can be computed concurrently while the prior row-group's accumulators are being written back to AMEM.
14. This process repeats for each row-group: read 8(+2) data rows from each input channel, compute 8 output data-rows in parallel on each of the 16 grid-rows (with separate output channels computed on each grid-row), until the entire output has been computed.

6.1.2 1x1 Convolution, or Weights x Data Flow

The LLC Grid also supports 1x1 convolution, deploying eight of the nine multipliers and adders in each LLC to compute 8 dot-product terms each cycle. This is equivalent and identical to a Linear layer (weights x data)

The processing flow is summarized as follows, focusing on the differences from 3x3 convolution described above:

1. Activation data is read from AMEM for one data-row (per cycle) across all input channels over successive cycles. I.e., for one tensor-row index, the input channel index is incremented every cycle. This is tiled into widths of up to 128 bytes, sent to the grid on the "vertical" data path.
2. When the data arrives at the LLC Grid, each LLC in each column collects 8 data elements over 8 cycles. These 8 data elements (at each LLC) correspond to the same tensor row for ***8 input channels***.
3. For the next 8 cycles, each LLC will compute the dot-products for these 8 data elements for ***8 output channels***, i.e., each LLC computes 8 different output channel contexts in a time-multiplexed manner. **For Linear MatMuls** (weights * data) the operation is identical; conceptually **each output channel is a separate row of the output matrix.**
    a. Each cycle, 8 individual weights are read from the buffer memories adjacent to each grid-row, and relayed to *all* of the LLCs on that grid-row. These 8 weights correspond to 8 input channels for the current output channel context. Note that the 8 weights are updated for every LLC every cycle, and all LLCs in a particular grid-row use the same weights.
    b. The LLCs multiply the 8 data elements with the 8 weights for the current output channel, using eight of the nine multipliers in the LLC structure. The 8 products are then added with one accumulator value using 8/9ths of the adders in the LLC. This uses the same log-multipliers, log-to-linear conversions, floating-point addition, and normalization used for 3x3 convolution.

c. The sum is then written to the LLC accumulator. Each accumulator slot corresponds to a different output channel (context) for the same output data row. This can be contrasted with 3x3 convolution, where each accumulator slot corresponds to a *different data-row* for the *same output channel*.

In summary, the LLCs collect 8 data values over 8 cycles and hold them steady for the next 8 cycles. During those next 8 cycles, they compute the dot-products for those 8 data with 8 different sets of weights to implement 8 output channels.

4. When all the input channels have been processed, the accumulation is complete for one tensor data row (for this tile). The accumulated sum is written to a second set of accumulators slots which are reserved for writeback. This data may now be written back to AMEM for the one data-row over 128 output channels (or equivalently, 128 output matrix rows). This writeback can occur in the background while the primary "active" accumulator slots are re-used for computing results for the next output data row.
5. This process repeats for each tensor data-row: read all input-channels for that one data-row index, compute 8 output channel results in parallel on each of the 16 grid-rows (for that data-row), until the entire output tensor has been computed.

➔ Since each grid-row supports 8 output channels, it operates as eight ***virtual grid-rows***, and the entire 16-grid-row LLC array appears as **128 virtual grid-rows**. Each virtual grid-row is equivalent to one output channel.

Note that this level of speedup requires hardware to supply 8 weight parameters per cycle, to each grid-row, totaling 128 weights per cycle "horizontally". This matches the AMEM bandwidth available, namely 128 bytes/cycle with 1-byte weights.

6.1.3 Data * Data Matrix Multiplication Flow

The UIEs can implement general matrix * matrix multiplication for two activation-data operands (also referred to as "MatMul" in this document). In this operating mode, each LLC in the Grid is used to compute one element of the output matrix in a tiled fashion. The (virtual) grid-rows are the rows of the output matrix, and grid-columns are the columns of the output matrix. This is a version of an output-stationary systolic array model.

The Grid computes **C** `= MatMul(`**`A, B`**`)`, generating a base tile size of 128 rows x 128 columns. The inputs to this tile are matrix **A** of size `128xK`, and matrix **B** of size `Kx128`, where `K` is the common dimension.

To execute MatMul, we must send one *column* per cycle of matrix **A** from the "side" of the grid, and one *row* of matrix **B** from the "top". This uses the two AMEM read ports (per UIE) to supply both the **A** and **B** data concurrently. The MatMul flow is as follows:

1. Matrix **A** data is read from AMEM one *row* at a time, but must be sent into the grid one *column* at a time. This data is read in blocks of 128 rows * 128 bytes, and then shifted "horizontally" into the side of the LLC Grid over 128 cycles. The switch for this is described in section 6.2.4.2

below.

Note that this delivers 128 bytes per cycle in the horizontal direction, so each of the 16 grid-rows will receive an average of 8 bytes per cycle from the "side".

2. Concurrently, matrix **B** data is read from AMEM at a rate of 128 bytes per cycle, and delivered to the "top" of the grid.
3. The computation in each LLC is identical to the scheme described above for 1x1 convolution:
   a. Every 8 cycles, the LLC collects 8 input data values shifted from the top from matrix **B** (corresponding to one column, and 8 rows of tensor data for that one column of **B**).

   b. For the next 8 cycles, buffers at the side of each grid-row provide 8 data values from matrix **A** and broadcast these values to all LLCs (columns) in that grid-row. These 8 data values correspond to 8 columns of a particular data-row of matrix **A**.

      Each of those 8 cycles corresponds to a different data-row of matrix **A**. In effect, 8 data-rows of matrix **A** are time-multiplexed onto each grid-row, one per cycle. Each grid-row therefore computes results for ***8 output data-rows***.

   c. In those 8 cycles, the LLC computes the dot-product of the 8 values from **A** and **B**, using the same math and structures described for 1x1 convolution. The results are written to an accumulator slot for each data row, i.e., using 8 accumulator slots for 8 output data-rows in each LLC.
4. If the common dimension "K" is greater than 128, this process is repeated for each set of 128 data-rows from **A** and **B**, continuing the accumulation of the dot-products in each LLC. The common dimension K can be any value, but is most efficient when K is a multiple of 128.
5. When the accumulation for this tile is complete, the accumulator data can be written back to AMEM, from each of the 16 grid-rows * 8 output data-rows/grid-row. Data is transmitted from the LLC Grid, 128 bytes per cycle, through the Write datapath to AMEM. The accumulators are double-buffered, so computing the next tile (in case of a larger matrix) can begin while the accumulator writeback is in progress for the prior tile.

➔ Since each grid-row supports 8 output data-rows for matrix multiplication, it operates as 8 ***virtual grid-rows***, and the entire 16-grid-row LLC array appears as **128 virtual grid-rows**.

6.1.3.1 Matrix A Transpose Option

In the description above, Matrix A data was supplied to the Grid one column at a time, by reading blocks of 128 rows * 128 columns row-wise, and shifting into the grid column-wise.

Matrix **A** can also be transposed on-the-fly between AMEM and the (side of the) LLC Grid, using the AMEM weights-reader data switch described in section 6.2.4.2 below. In this mode, each row of **A** read from AMEM would be a logical column of $\mathbf{A}^\mathbf{T}$, and so as we read one row x 128 columns from **A**, this is effectively one column x 128 rows for $\mathbf{A}^\mathbf{T}$. We can directly feed the 128 bytes of $\mathbf{A}^\mathbf{T}$ to 128 virtual-grid-rows each cycle. Byte 0 goes to virtual-grid-row 0, byte 1 to virtual grid-row 1, … byte 127 to virtual-grid-row 127.

In this mode, there is no need to read in chunks 128 rows, and the common dimension 'K' may be any size without degrading performance.

A matrix may also be transposed and written back to AMEM in transposed form using this facility. The method is to read and transpose Matrix **A** on-the-fly, and multiply $\mathbf{A}^T$ by an identity matrix, thereby capturing $\mathbf{A}^T$ in the LLC grid accumulators, then writing this result back to AMEM. This also supports transposing a larger matrix in units of 128x128 tiles

6.1.3.2 FP16 Matrix Multiplication

FP16 requires 2 cycles per data element to transfer from AMEM, since FP16 (two-byte) values are serialized over two cycles per (1-byte-wide) column lane. This limits throughput to 50% of FP8. The two-byte FP16 values are converted to (two-byte) LnsI5F10 values prior to sending data to the grid.

- Every 8 cycles, the LLC collects ***4*** LnsI5F10 data values shifted from the top from matrix **B** (corresponding to one column x 4 rows of tensor data for that one column of **B**).
- For each of the next 8 cycles, logic at the side of each grid-row collects 4 LnsI5F10 data values from matrix **A** and broadcasts these values to all LLCs (columns) in that grid-row.
- Each of those 8 cycles corresponds to a different data-row of matrix **A**. In effect, 8 data-rows of matrix **A** are time-multiplexed onto each grid-row, one per cycle. Each grid-row therefore computes results for ***eight output data-rows***, which is the same logical organization used for FP8xFP8 MatMul. However, each cycle only 4 dot-product values are computed instead of 8/cycle with FP8

This arrangement achieves the maximum possible FP16 MatMul throughput of 128 bytes/cycle from both horizontal and vertical data flows, and computes 128x128 tiles at 50% of the throughput compared to FP8.

6.1.4 Vector Processing Unit Flow

The Vector Processing Unit (VPU) provides Single-Instruction-Multi-Data (SIMD) processing on one data-row at a time. This includes an array of 128 ALUs, with one ALU per each of the 128 datapath columns, and each of the ALUs execute the same program on the data received in their respective columns. The VPU has its own custom microcoded instruction set and general-purpose registers, and can execute multiple microinstructions on each data element (row).

The VPU may operate concurrently with (or instead of) the LLC Grid. The VPU can do dedicated vector-wise operations on activation data, or the ALU operation may be "fused" to the end of a convolution or MatMul from the Grid. The VPU operates on **FP16** data natively.

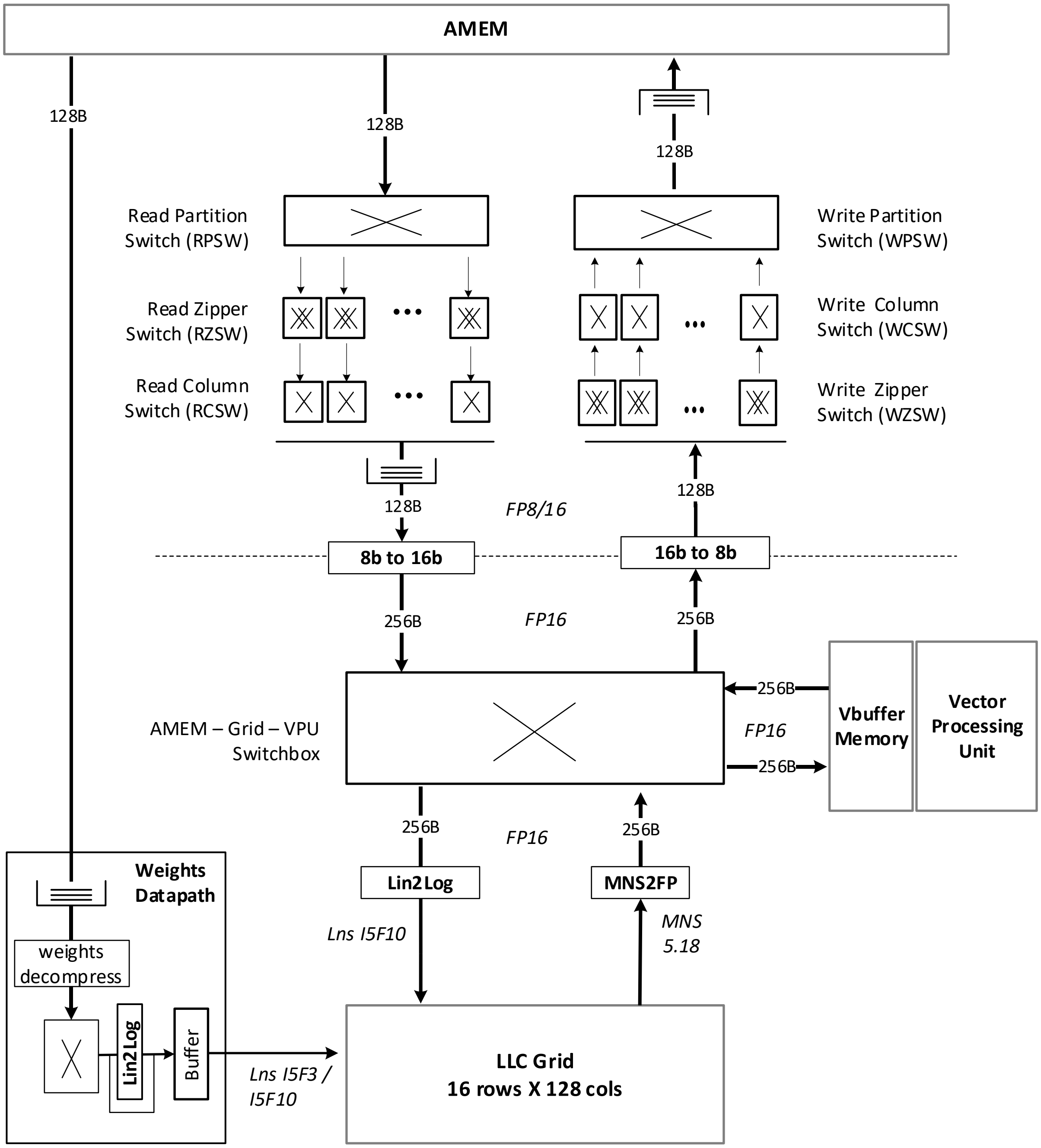
Data read from AMEM may be routed to the VPU for pre-processing, and data offloaded from the Grid may be routed to the VPU for post-processing, or both. The VPU has a *VectorBuffer* (**Vbuffer**) memory which can buffer multiple 128 x 128 data tiles transferred to and from the VPU. This serves as a decoupling buffer for data read from AMEM, or writeback data from the Grid. This allows an application which computes a tile's worth of data in the grid over a number of clock cycles, to quickly offload the tile to the Vbuffer, allowing the Grid to immediately commence computation for the next tile. Concurrently with the Grid's work on the 2nd tile, the VPU can perform additional computations for the first tile.

For dedicated vector-wise operations, data is fetched from AMEM as required by the specific algorithm, and sent directly to the VPU, bypassing the LLC Grid. Data is first written into a Vbuffer memory for use by the VPU algorithms. When complete, the results are sent back to AMEM via the Write datapath. The flow of data is AMEM→(Vbuffer)→VPU→(Vbuffer)→AMEM. Note that another portion of the Vbuffer is used to decouple the VPU outputs from the transfer to AMEM

For fusing activation functions to convolution or MatMul operations, the natural flow of data is AMEM→Grid→(Vbuffer)→VPU→(Vbuffer)→AMEM . In this arrangement, the VPU can execute a fused activation function on the writeback data before forwarding it to AMEM.

For more sophisticated algorithms, data may be recirculated from the VPU to the LLC Grid, bypassing the AMEM. This speeds up operations such as Softmax which may need to make multiple passes over the data, while the data stays in the higher-precision FP16 format at full performance.

6.2 UIE DATAPATH



The UIE Datapath has the following major blocks

- Read datapath: transfers data from AMEM to the LLC Grid or VPU
- Write datapath: transfers data from the Grid or VPU to AMEM

- Weights datapath: transfers weights and data from AMEM to the Grid, including weights decompression
- The LLC Grid (16 rows x 128 columns of Leaf Level Convolver elements)
- Vector Processing Unit (VPU) including Vector Buffer

In the middle of the read & write datapaths is the AMEM-Grid-VPU switchbox, which can route data from any source to any destination from among AMEM, LLC Grid or VPU.

6.2.1 UIE Read Datapath

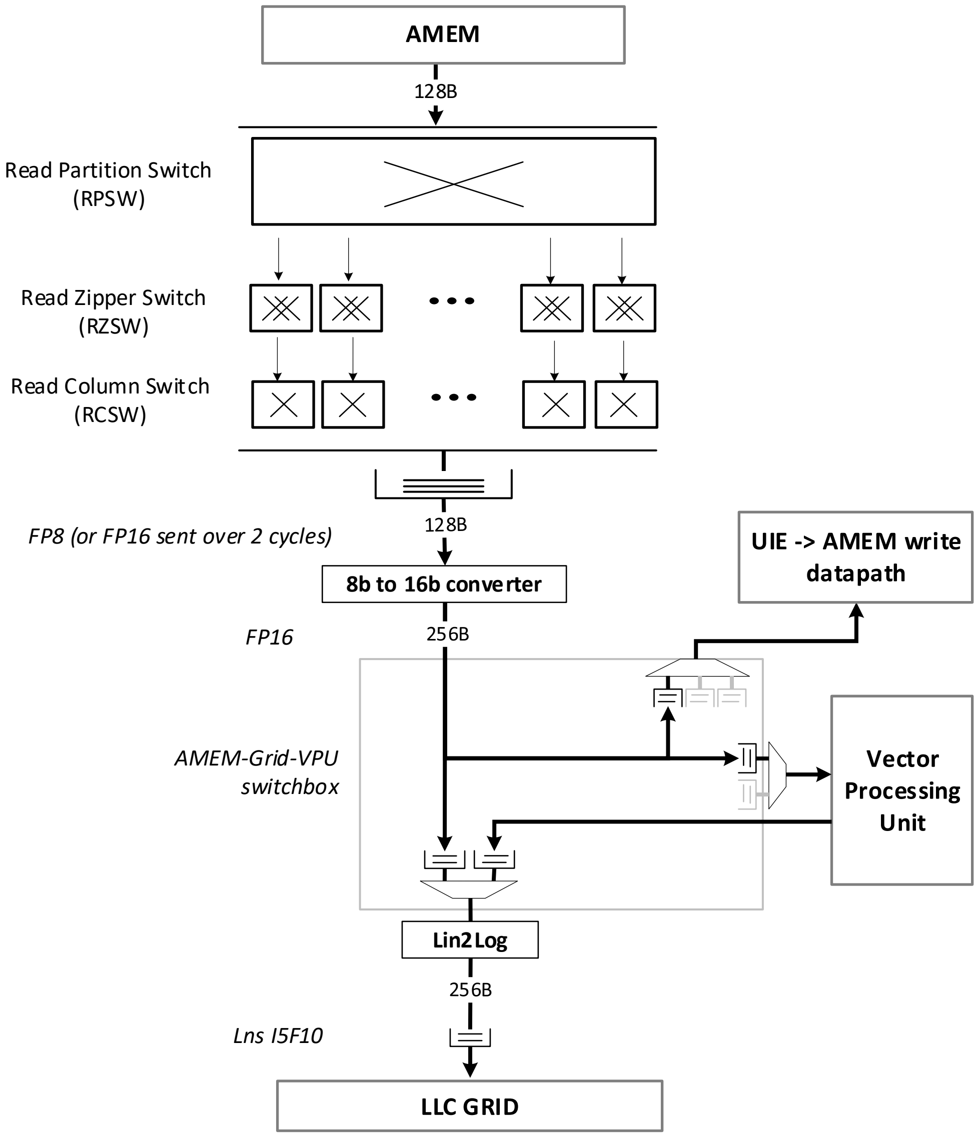

AMEM
128B
Read Partition Switch (RPSW)
Read Zipper Switch (RZSW)
Read Column Switch (RCSW)
FP8 (or FP16 sent over 2 cycles)
128B
8b to 16b converter
UIE -> AMEM write datapath
FP16
256B
AMEM-Grid-VPU switchbox
Vector Processing Unit
Lin2Log
256B
Lns I5F10
LLC GRID

Figure 6-2: UIE Read Datapath

For the most basic operations, the UIEs can read one tensor-row at a time from AMEM and stream the rows directly to the LLC Grid for processing. However, there are numerous cases where we need to do some data manipulation prior to processing by the LLC Grid, and this is done on the fly as the data travels from AMEM to the Grid. This is handled by the Read datapath blocks. This includes:

- Partition Switch: crossbar-style switch with partition-level switching granularity.
- Zipper Switch: performs column-level switching to “unzip” or “zip” even & odd data columns into (or from) partition-pairs for space-to-depth / depth-to-space transforms.
- Column Switch: Performs a second stage of column-level switching, for shifting, switching and broadcasting columns within and across adjacent partitions. This is a crossbar-style switch with column-level switching granularity within each partition
- 8b to 16b Lane Converter: adapts received 8-bit (per column) data from AMEM, to 16-bit physical data lanes to be sent to downstream datapaths.
- FP to LNS conversion: for data destined to the LLC Grid, we convert from floating point (FP8/16) to LNS16 before sending to the grid

6.2.1.1 Partition Switch (PSW)

The Partition Switch (PSW) performs *partition-granular* switching functions between AMEM and the LLC Grid. There is one instance of PSW on each of the Read and Write datapaths, denoted **RPSW** and **WPSW**.

This is a non-blocking switch to support virtually any partition-to-partition switching pattern. Capabilities include:

- Replication of any power-of-two subset of the input partitions, or any power-of-two length *sequence* of partitions.
- Rotation of the 8 data partitions, on a partition basis. This supports unaligned reads of any 8 contiguous logical partitions from the AMEM.
- Partition-level swizzling to support unzipping and re-zipping even & odd data columns, needed to prepare the data for dilated convolutions (“Space to Depth” and “Depth to Space”). In effect, this unzips or re-zips even/odd partitions.

6.2.1.1.1 PSW Structure

The switch is implemented with a stacked butterfly network, with 5 ranks. The mux-switching “distance” of the ranks is 1-2-4-2-1, as seen below.

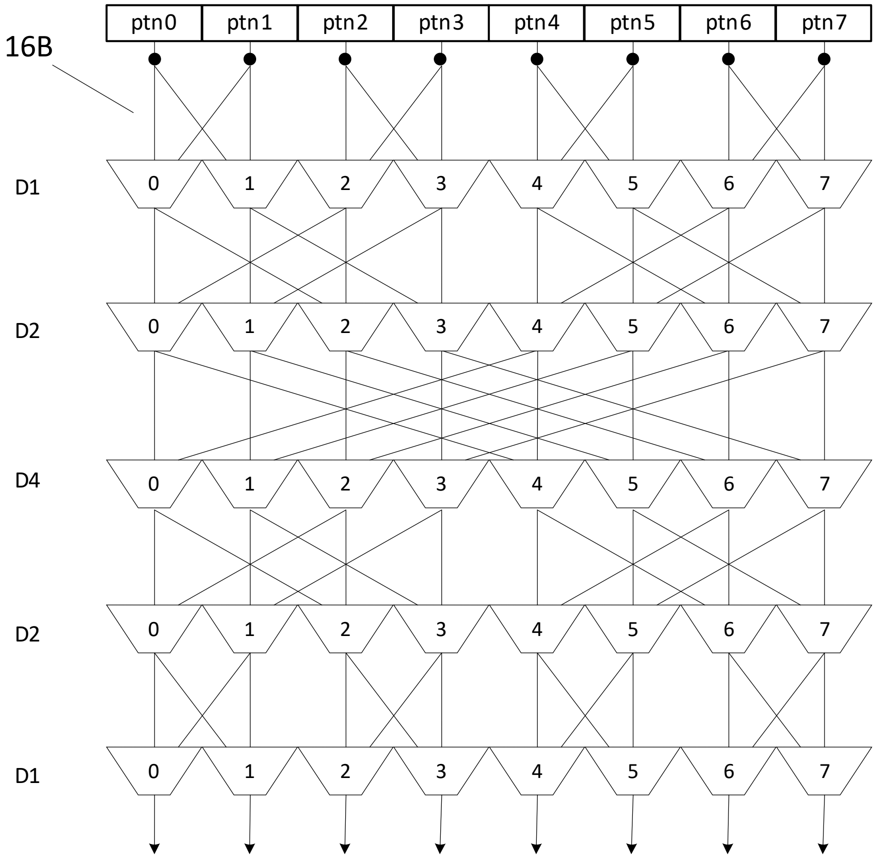


Figure 6-3: Partition Switch

6.2.1.1.2 Configuration and programming:

PSW may be programmed to update switch settings **every cycle**. This is a higher efficiency than in Scorpio, which permitted PSW switching to change at most once per 16 cycles. This is required by the more compact Pyxis AMEM tensor mapping, which allows arbitrary alignment of successive rows or channels, and hence, may require a different alignment from AMEM to the Grid each cycle.

6.2.1.2 Zipper Switch (ZSW)

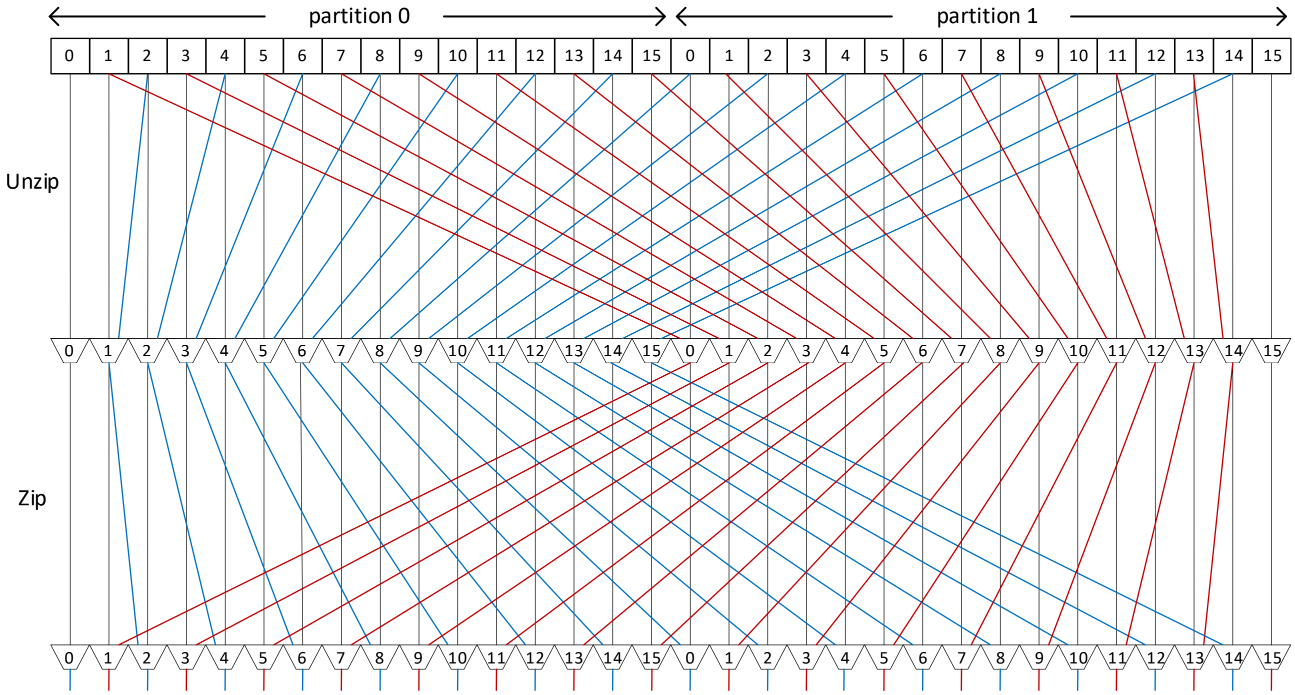


Figure 6-4: Zipper Switch multiplexing per 2 partitions

This block performs local column-level “zip” and “unzip” function necessary for “**space to depth**” and “**depth to space**” transforms by zipping or unzipping even/odd pairs of columns across two adjacent partitions in the datapath. This can also be used to segregate all the even and odd columns from a partition-pair.

- **Unzip**: this function takes pairs of partitions’ data (2 partitions x 16 columns/partition) and “unzips” the columns, so all the even (input) data columns are contiguous in the partition-0 output, and all the odd (input) columns are contiguous in the partition-1 output. This is instantiated 4 times for the 8 partitions-wide datapath.
- **Zip**: this function takes pairs of partitions’ data (2 partitions x 16 columns/partition) and “zips” the columns, interleaving the columns from each of the adjacent partitions, so the output contains the 16 input columns from each of two partitions, interleaved column-by-column like a zipper.

There is one instance of Zipper Switch on each of the Read and Write datapaths, denoted RZSW and WZSW.

6.2.1.2.1 Unzip operation for Space to Depth

Unzipping prepares data for dilated convolution, supporting **space to depth** operation by collecting all even columns and odd columns into separate output channels. This is further segregated among even and odd rows, creating four output channels with combinations of (even, odd) rows and (even, odd) columns.. Once the even/odd columns are segregated in each partition pair, the data may be further unzipped by a partition switch (RPSW or WPSW), reorganizing the partitions such that all even columns are contiguous, and all odd columns are contiguous across the entire width of the datapath. This requires a minimum tensor width of 2 partitions (32 columns) to support dilation-rate=2 (since the unzipping function rearranges columns among two partitions). If data is unzipped in this manner multiple times on successive trips, each trip increases the dilation-rate by a power of 2.

6.2.1.2.2 Unzip for squeezing columns on writeback

When writing back a layer with /2 downsampling or stride2 convolutions, only every-other column would have meaningful data. We want to concatenate the meaningful data-columns (which will total half of the grid width) and write these as up to 8 contiguous partitions back to the AMEM. This is done in two parts: first concatenate the 16 meaningful columns out of 32 within each partition-*pair*, to create one full partition's worth from each grid partition-pair. Then concatenate all the meaningful partitions and write as a contiguous block of up to 4 partitions (64 columns) to AMEM.

The Unzip function performs the column-squeezing part: it can concatenate either all-even or all-odd columns from each partition-pair, to create one partition's worth, i.e., 16 contiguous data columns selected from 32. The even (input) columns are output from the lower-numbered partition within each pair, and odd (input) columns are output from the higher-numbered partition within each pair.

We then configure the Write Partition Switch to concatenate either the even or the odd (output) partitions into the desired block of 4 contiguous data partitions to be written to memory

6.2.1.2.3 Zip operation for Depth to Space

Zipping reverses the even/odd column segregation. After performing a "space to depth" transform, the even and odd-numbered data columns will have been (previously) segregated into data-partitions of all even or all odd tensor columns. This may be re-integrated (interleaving even and odd columns), performing a "**depth to space**" operation using this switching feature. Note that the Partition Switch must first swizzle the data *partitions* so that they have even-columns/odd-columns/even-columns/odd… pattern, then the zip swizzles the columns within even/odd partition pairs to interleave the even & odd columns back together

Configuration and programming:

Zipper Switch configuration may normally be set once per trip, and remains in effect until reconfigured, typically for a subsequent trip. Normally at most one of the zip and unzip muxing operations is selected.

6.2.1.3 Column Switch (CSW)

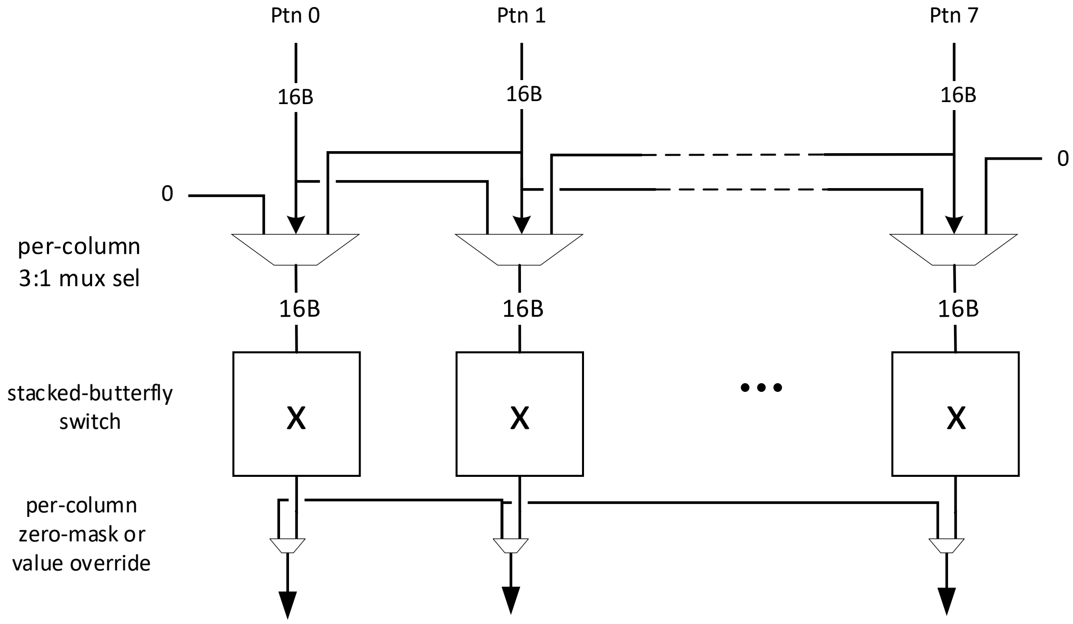


Figure 6-5: Column Switch

CSW provides column-level switching within a partition. It provides the following switching features:

- **Left or right shift by any number of columns** (up to 16). The data from the partitions to the left & right are shared and muxed. The 3:1 muxes in the figure select data either from partitions (*i-1*, *i)* for right-shift, or (*i*, *i+1)* for left-shift, independently per column. Each partition-pair can also swap columns in any selection.
- **Broadcasting** (replicating) any single input column, or any power-of-two-length sequence of columns, across the output columns of each partition
- **Swizzling columns** (with any-to-any mapping) within any partition or partition-pair
- **Per-column zero-masking or value override** allows forcing values to zero or -max on a per-column basis. This can mask invalid columns past the edge of a logical tensor row, or force `-max` for max-pooling support past the tensor edge. This is also used for upsampling and interpolation, to insert zeroes on all-even or all-odd columns as described above.

There is one instance of CSW on each of the Read and Write datapaths, denoted RCSW and WCSW.

6.2.1.3.1 Column-shift operation of CSW

To implement a left-shift by N columns, the input muxes in each partition **K** are configured to select columns [0:N-1] from partition **K+1**, and columns [N:15] from partition **K**. Then the switching logic is configured to rotate-left by N columns. If N=16, select all columns from partition K+1, and no further rotation is needed in the column switching logic.

Similarly, to implement a right-shift by N columns, the input muxes are configured to select columns [0:15-N] from partition **K**, and columns [15-N+1:15] from partition **K-1**. The switching logic is configured to rotate-right by N columns. If N=16, select all columns from partition K-1, and no further rotation is needed in the column switching logic.

For narrow tensors in which multiple rows or channels are present side-by-side in the datapath, the shifting function may shift-in invalid data from across the tensor edge (at partition boundaries). This can be corrected by zero-masking the columns which received invalid shift-in values.

6.2.1.3.2 CSW Structure

The CSW basic building block supports one partition, and includes

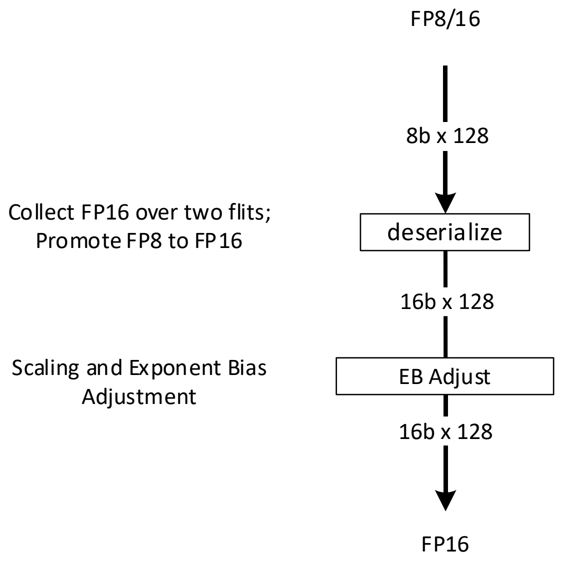
- **3:1 input mux with per-column source selection**: this selects data from the partition to the left, right, or this partition, for each column independently.
- **Stacked-butterfly 16x16 column switch**: this supports virtually any switching pattern from the 16 selected input columns to 16 output columns within a partition. The switch is implemented with a stacked butterfly network, with 7 ranks. The mux-switching “distance” of the ranks is 1-2-4-8-4-2-1 columns.
- **Per-column zero masking or override mask**: This is applied after the 16x16 switch, at which point all of the data have been steered to their destination columns to be delivered to the LLC Grid or Vector Processing Unit (on the Read datapath instance of this block). The mask is configured to force a value of zero, -max or no-masking on a per-column basis.

6.2.1.3.3 Configuration and programming:

CSW provides a 64-deep table of configurations for the 3:1 mux selects and butterfly switch settings. The associated microprogram can apply mux select and butterfly settings (from the config table) as often as **every cycle**.

The per-column zero masking or value-override mask is configured in a smaller table, with depth of just 4 entries. This is expected to be updated at most once per trip, and to remain effective until the next trip.

6.2.1.4 8b to 16b converter



This block receives data as 8-bits per column, which may be FP8, or FP16 serialized over two cycles. Data is output 16-bits per column as FP16. There are two main sub-functions to this block:

6.2.1.4.1 FP16 Deserializer

FP16 data is sent from AMEM to the UIEs serialized over two cycles per column. This function collects two successive bytes into a single FP16 data element. In case of FP8 input data, no deserialization is needed, but each FP8 byte is promoted to FP16.

6.2.1.4.2 EB Adjuster

Conversion is based on the following formats:

	(1)	(4)	(3)
FP8:	S	E4	F3
	sign	Exponent	Mantissa (Fraction)

to

	(1)	(5)	(10)
FP16:	S	E5	F10
	Sign	Exponent	Mantissa (Fraction)

FP16 and FP8 typically use different EB's, since FP16 can represent a larger dynamic range of values. To convert FP8 to FP16, the data must be scaled by an integer power-of-two, which simply adds or subtracts an adjustment amount to the Exponent of the FP data. The FP8→ FP16 adjustment is:

```
       FP16_out.Exponent    = FP8_in.Exponent + EB_Adj_FP8
where
       EB__Adj_FP8          = EB_FP8 – EB_FP16
```

The `EB_Adj_FP8` scaling factor is a programmable parameter (per trip). This typically increments the exponent to convert to a more-negative EB when converting from FP8 to FP16.

EB Adjustment may also be used to **scale** incoming FP16 data by an arbitrary integer power of two by programming the `EB_Adj` to increment or decrement the exponent by the scaling factor. For example, to scale an FP16 value by 2^4, set the `EB_adj` value to -4. The FP16 → FP16 scaling is:

```
       FP16_out.Exponent    = FP16_in.Exponent + EB_Adj_FP16
where
       EB_Adj_FP16          = -log2(scaling_factor)
```

The `EB_Adj_FP16` scaling factor is also a programmable parameter (per trip)..

Note: Separate `EB_Adj_FP8 and EB_Adj_FP16` values may be applied for FP8 and FP16 in mixed-precision trips.

For non-FP8/16 data passed through the datapath (such as for relaying integers or computed addresses) the EB_Adj may be set to zero to pass opaque data.

6.2.1.5 Lin2Log converter

Before data is forwarded to the LLC Grid, it must be converted to LNS format. This is required because the LLCs perform multiplication via addition of log-domain numbers. In the main "vertical" datapath, the data is received at this block in FP16 format and converted to LnsI5F10. The conversion to LNS uses the linear-to-log mapping described in section 4.8 above.

The conversion options are listed in the following table for the "vertical" data flow

Source Data Type	Map to	include "mapping correction"?	Notes
FP16	LnsI5F10	yes	Typical case for FP16 for **1x1, 3x3 conv or MatMul**
	LnsI5F10	no	Simple mapping without mapping correction, for passing data through the grid unchanged
LnsI5F10	LnsI5F10	N/A	Only for weights datapath: supports receiving data already mapped to LnsI5F10 format.

*Table 6-1: Linear-to-Log mapping options for **vertical** data flow*

After the conversion to LnsI5F10, there is the option to truncate the fraction to fewer than 10 bits, and replace the truncated bits with zeros. This reduces the accuracy of the mapping but can save power in the LLC grid. This is appropriate when the incoming data was originally FP8 (and was promoted to FP16 in the datapath) so the extra log-fraction bits may carry excess precision.

A second instance of Lin2Log is used for the "Matrix A", or "horizontal" flow data from AMEM to the LLC Grid for MatMul. (See also section 6.2.4.4 for a description of how this is used in the horizontal flow). This Lin2Log instance has slightly different mapping support as shown below. Also, the FP8 data and LnsI4F3 weights on the "horizontal" path are converted to `LnsI5F3` (i.e., with 5-bit integer exponent). Along with the conversion to 5-bit exponent, the EB is adjusted accordingly as described previously.

Source Data Type	Map to	include "mapping correction"?	Notes
LnsI4F3	LnsI5F3	N/A	Weights are received in LNS format already, so linear-to-log mapping is bypassed. However, EB must be adjusted with 4-bit to 5-bit exponent conversion, so the LNS format is "promoted" to LNS5.3 before sending weights to the grid
FP8	LnsI5F3	yes	Typical case for FP8 for **MatMul** data. Output of mapping correction is rounded to 3 fraction bits
	LnsI5F3	no	Simple mapping without mapping correction. Used when data is passing through the grid unchanged, such as for Transpose operations
FP16	LnsI5F10	yes	Typical case for FP16 for **MatMul** data.
	LnsI5F10	no	Simple mapping without mapping correction, for passing data through the grid unchanged

*Table 6-2: Linear-to-log mapping options for **horizontal** data flow*

6.2.2 UIE Write Datapath

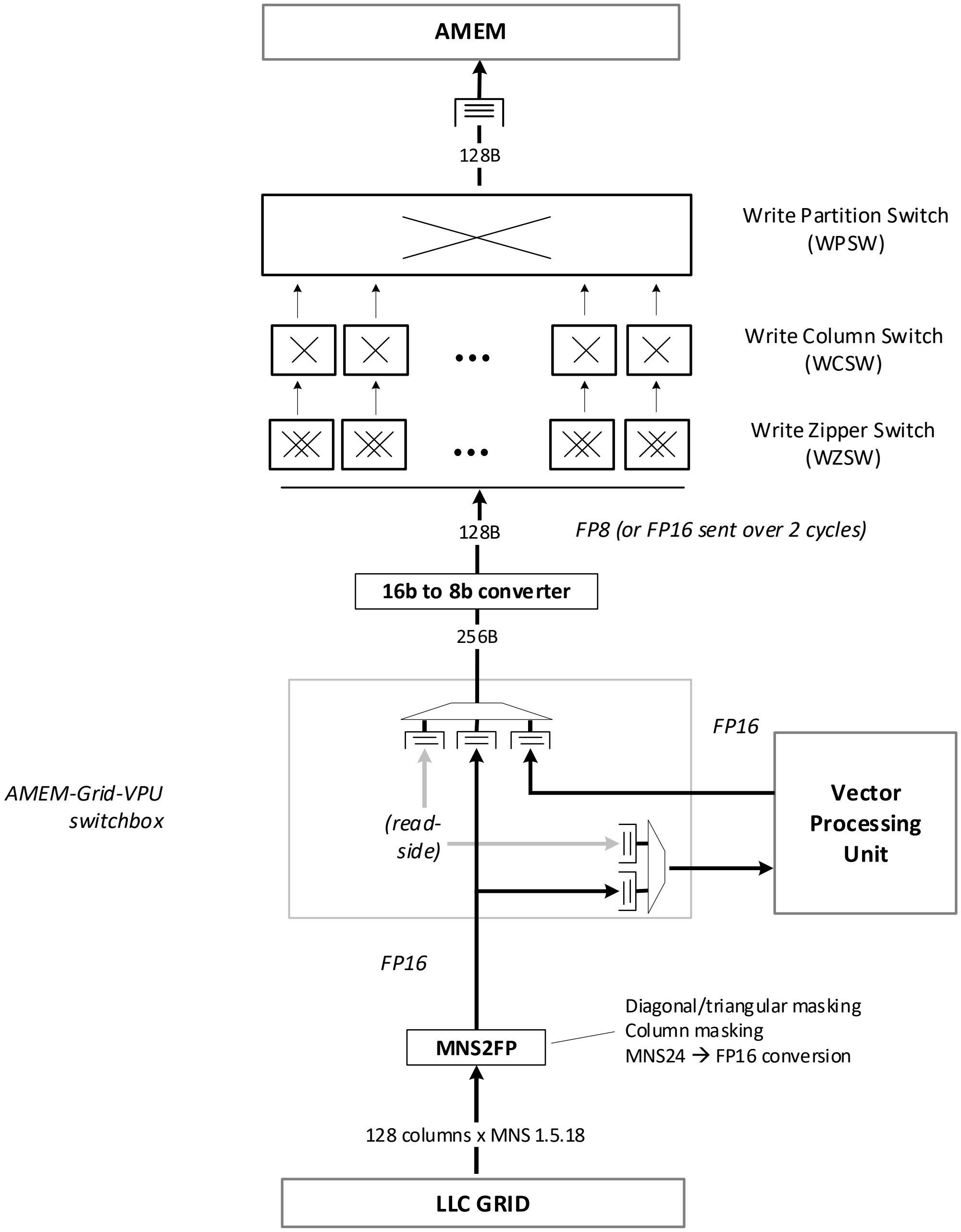


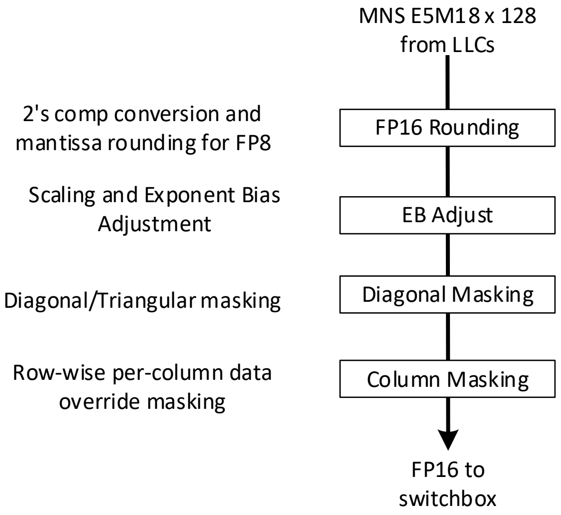
Figure 6-6: UIE Write Datapath

Data generated by the LLC Grid or Vector Processing Unit passes through the Write datapath on the way to AMEM. This datapath provides switching and some data processing functions to prepare the data to be stored in memory. This includes:

- MNS to FP converter and mask: converts MNS5.18 data from the Grid to FP16 for use in the datapath. Also applies masking and optional EB-adjustment

- 16b to 8b converter: Adapts received 16-bit (per column) data from the upstream datapaths, to 8-bit physical data lanes to be sent to AMEM.
- Zipper Switch: performs column-level switching to "unzip" or "zip" even & odd data columns for space-to-depth / depth-to-space transforms. Also "squeezes" out every other column for stride-2 convolution or 2x2 pooling. Same as read-side ZSW block
- Column Switch: Performs a second stage of column-level switching, for shifting, switching, and broadcasting columns within and across adjacent partitions. Same as read-side CSW block.
- Partition Switch: crossbar-style switch with partition-level switching granularity.

6.2.2.1 MNS to FP converter and Mask (MNS2FP)


MNS E5M18 x 128
from LLCs
2's comp conversion and mantissa rounding for FP8
FP16 Rounding
Scaling and Exponent Bias Adjustment
EB Adjust
Diagonal/Triangular masking
Diagonal Masking
Row-wise per-column data override masking
Column Masking
FP16 to switchbox

The MNS2FP block receives Writeback data from the LLC Grid, and converts this from MNS to FP16 format. This also includes column-wise masking for invalid columns since this is the first block downstream of the LLC Grid. This is a fixed-function pipelined block with throughput of one data row per cycle.

Operation includes the following steps:

6.2.2.1.1 MNS to FP16 mapping

1. Convert 2's complement mantissa (from MNS) into sign-and-magnitude format

MNS values use 2's complement encoding for the mantissa; FP16 representation is sign-and-magnitude. If the mantissa sign-bit is set, conversion is done by inverting the mantissa's fraction part and adding one in the fraction LSB. However, there are two corner cases to consider:

- Since MNS uses a 2’s-complement mantissa, the normalized MNS mantissa values are in the range [-2.0 , +1.1111…1_2]. The sign-and-magnitude representation covers [-1.1111…1_2, +1.1111…1_2]. So, we must convert the corner case of -2.0 * 2^E to -1.0 * 2^(E+1).

2. Round mantissa to 10 bits

MNS values on the writeback path have higher precision than FP16, which helps in accuracy during accumulation of dot-products. To convert to FP16, the mantissa must be rounded to 10 bits.

3. Convert special codepoints from MNS to FP16

The MNS value -2.0 * 2^E_{max} is reserved for NaN encoding, and is converted to Recogni-internal FP16 NaN encoding of “negative 0.

6.2.2.1.2 EB Adjuster

MNS values may have a different exponent bias from the EB desired for FP16 data. The exponent may be adjusted based on a programmable EB_Adjust value. Note: EB Adjustment is equivalent to scaling values by an integer power-of-two. See section 4.6 for a description of MNS EB

```
        FP16_out.Exponent    = MNS_in.Exponent + EB_Adj_MNS_FP16
where
        EB_Adj_MNS_FP16      = -log2(scaling_factor)
```

Hardware supports multiple programmable `EB_Adj_MNS_FP16` values per tensor, to allow for different dynamic ranges for different output channels. The EB Adj value is programmed in the Grid Writeback sequencer.

6.2.2.1.3 Diagonal masking

On the MNS to FP writeback path, “diagonal” masking may be used to mask out an upper or lower triangle of the matrix on writeback, by overriding the data with zeros or **-max** values. This is useful for Attention Q*K^T calculations feeding to a Softmax in “prefill”, where Q (per token) should not attend to K for “future” tokens, so half of the matrix is masked out to -max value prior to Softmax.

This operates by comparing each data row index to the column index (for each data element) as the data streams through. It supports the following masking (override) options.

diagonal_mask_mode	**Determines when/how** values are replaced with diagonal_mask_val in the 128x128 output tile matrix: 0: diagonal masking disabled 1: replace if column <= row (mask lower-left-triangle including diagonal) 2: replace if column < row (mask lower-left -triangle not including diagonal) 3: replace if column == row (mask only the diagonal) 4: replace if column != row (mask everything except the diagonal) 5: replace if column >= row (mask upper-right triangle including diagonal) 6: replace if column > row (mask upper-right triangle not including diagonal)
diagonal_mask_val	If a value is selected for masking, replace it with the following value: 0: replace data with **0** 1: replace with **-max** (fp16)

The diagonal masking feature may be enabled/disabled on a per tensor-row basis by the grid-writeback sequencer.

6.2.2.1.4 Column-wise masking

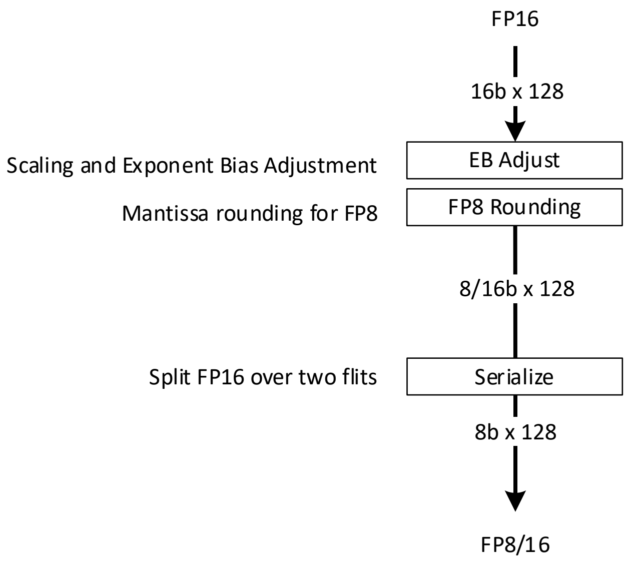
On the MNS to FP writeback path, per-column zero-masking is applied with a configurable 128-bit register. This is useful for cases where the tensor is not a multiple-of-16 width, and we want to cleanly mask-out the invalid columns in the writeback data (both for power savings and to ensure correct computation for any downstream operations).

The mask is configured to force a value of zero, -max or no-masking on a per-column basis.

Four configurable masks are provided, which may be selected on a cycle-by-cycle basis by sequencer programming.

Both diagonal and column masking may be applied. For columns which receive an override value (force to zero or -max) from both the diagonal and column masks, the column mask will take precedence.

6.2.2.2 16b to 8b Converter


FP16
16b x 128
Scaling and Exponent Bias Adjustment
EB Adjust
Mantissa rounding for FP8
FP8 Rounding
8/16b x 128
Split FP16 over two flits
Serialize
8b x 128
FP8/16

Data enters this block from the 3-way switchbox in FP16 format, i.e., 16-bits per column x 128 columns, and the downstream datapath (toward AMEM) is only 8-bits per column. This block either converts the FP16 to FP8, or serializes FP16 data over two cycles to adapt to the 8b path, depending on configuration. Each row of data is tagged with the resulting type, which allows for writing back mixed-precision results where some tensor data-rows are FP8, others FP16.

Conversion to FP8 is based on the following formats:

	(1)	(5)	(10)
FP16:	S	E5	F10
	Sign	Exponent	Mantissa (Fraction)

to

	(1)	(4)	(3)
FP8:	S	E4	F3
	sign	Exponent	Mantissa (Fraction)

6.2.2.2.1 EB Adjuster

For conversion from FP16 to FP8, the Exponent Bias must be adjusted to a configured value, so the resulting exponent can be downsized from 5 bits (FP16) to 4 bits (FP8). The FP16→ FP8 exponent adjustment is:

```
      FP8_out.E4       = FP16_in.E5 + EB_Adj_FP8
where
      EB_Adj_FP8       = EB_FP16 – EB_FP8
```

The EB_Adj_FP8 scaling factor is a programmable parameter (per trip).

EB Adjustment may also be used to **scale** the FP16 data by an arbitrary integer power of two by programming the EB_Adj to increment or decrement the exponent by the scaling factor. This is available even when not converting to FP8. For example, to scale an FP16 value by 2^4, set the EB_adj value to -4. The FP16 → FP16 scaling is:

```
      FP16_out.E5   = FP16_in.E5 + EB_Adj_FP16
where
      EB_Adj_FP16   = -log2(scaling_factor)
```

The EB_Adj_FP16 scaling factor is also a programmable parameter (per trip).

Note: Separate **EB_Adj_FP8** and **EB_Adj_FP16** values may be applied for FP8 and FP16 in mixed-precision trips. Hardware provides support for multiple programmable EB_Adj values per trip

6.2.2.2.2 FP8 Rounding

Converting from FP16 to FP8 requires rounding the mantissa (fraction) from 10 bits to 3 bits. This uses a round-to-even policy:

```
F3 = F10[9:7]                        // start with 3 MSBs from F10 fraction

if (F10[6] && F10[5:0]!=0) ||        // round up if more than half way
   (F10[7] && F10[6] && F10[5:0]==0) // round-to-even if exactly half way
then
    F3 = F3 + 1                // round up
```

If rounding up F3 causes a carry out of the fraction, the exponent is incremented, and if this causes exponent overflow, the value will saturate to the all-1's exponent and mantissa.

6.2.2.2.3 FP16 Serializer

If the data is to be written to AMEM in FP16 format, each data element must be multiplexed (serialized) over 2 cycles since downstream datapaths are 1-byte wide per column. LSByte is sent first.

6.2.2.3 Write Zipper Switch (WZSW)

This is an instance of the Zipper Switch block described in section 6.2.1.2 above.

When writing back a layer with /2 downsampling or stride2 convolutions, only every-other column would have meaningful data. We want to concatenate the meaningful data-columns (which will total half of the grid width) and write these as up to 4 contiguous partitions back to the AMEM. This is done in two parts: first concatenate the 16 meaningful columns out of 32 within each partition-*pair*, to create one full partition's worth from each grid partition-pair. Then concatenate all the meaningful partitions and write as a contiguous block of up to 4 partitions (64 columns) to AMEM.

The “zip” function performs the first part: it can concatenate either all-even or all-odd columns from each partition-pair, to create one partition's worth, i.e., 16 contiguous data columns selected from 32. The even columns are output from the lower-numbered (i.e., even-numbered) partition within each pair, and odd columns are output from the higher-numbered (odd) partition per pair.

Write Zipper can also be used perform space-to-depth or depth-to-space type transformations on data on the Write path.

6.2.2.4 Write Column Switch (WCSW)

This is an instance of the CSW block described in section 6.2.1.3 above

This may be used for shifting columns from the grid alignment to any column position, for storing in AMEM. It may also be used for aligning half-partitions (following the WZSW block) following stride-2 convolution or downsampling.

6.2.2.5 Write Partition Switch (WPSW)

WPSW is an instance of the Partition Switch stacked butterfly, identical to the Read Partition Switch described in 6.2.2.

This supports partition-granular switching functions between the Grid / VPU and AMEM. It has the following main requirements:

- Rotation of the 8 data partitions, on a partition basis. This supports unaligned writes of any 8 contiguous partitions to AMEM.
- Partition-concatenation on the writeback path, used for squeezing every-other partition into a contiguous block of 4 partitions
- Swizzling partitions for FP16 data storage, in effect spreading each two-byte logical column into two physical (1-byte-wide) columns.

6.2.3 AMEM – Grid – VPU Switchbox

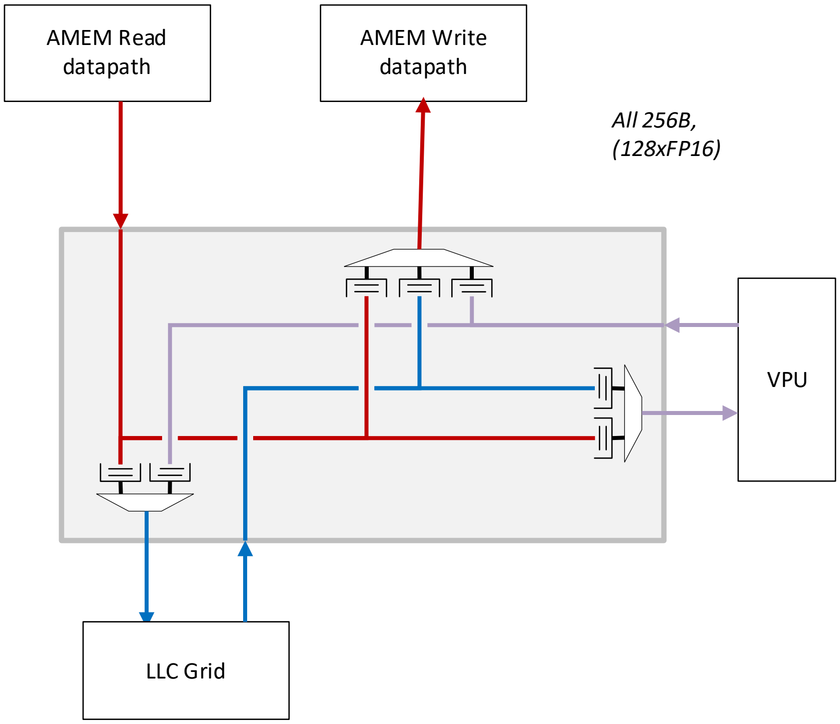


Figure 6-7 AMEM – Grid – VPU Switchbox

This is a buffered crossbar switch connecting the Read & Write datapaths (from/to AMEM), the LLC Grid, and the VPU. It is simply a set of FIFO-buffered muxes and controls which allow routing data among the UIE resources. All inputs and outputs are 128 x FP16 flits.

Data is routed through the switchbox under microcoded sequencer control:

- For each source of data to the switchbox, a sequencer will steer each 256-byte flit to a per-source/destination switchbox FIFO.
- For each destination out of the switchbox, a sequencer will issue FIFO-pop commands for a specific source from the appropriate FIFO.

Note this is **not** an arbitrated switch – the order of data proceeding to each destination is pre-determined by microcode programming since each destination will be executing a specific microprogram to consume data from a specific source. Therefore, it is important to carefully construct the microprograms to avoid deadlock; a source must not push data to the switch unless the program can guarantee the data will be consumed without requiring the source to make any further progress. This is particularly critical because the switch patterns allow for recirculation and therefore cycles can be formed.

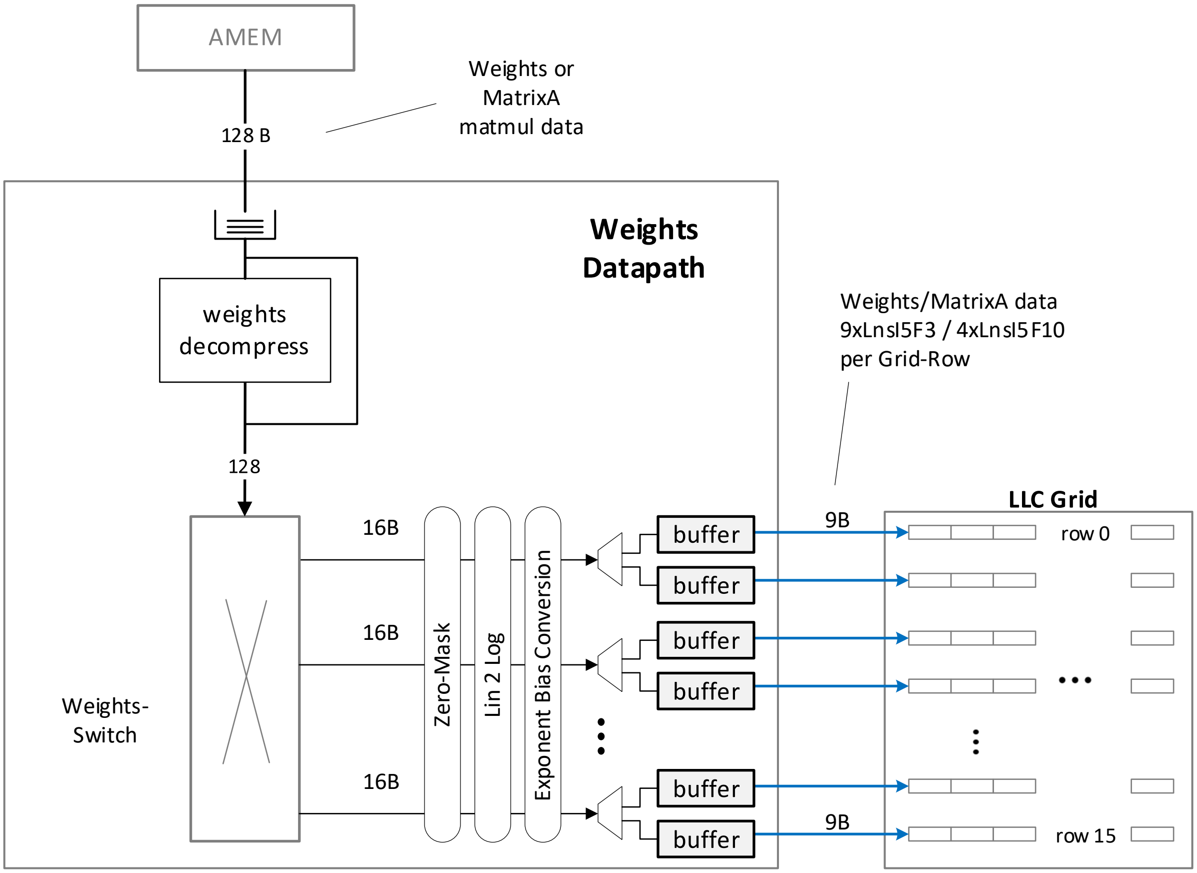
The switchbox connections and routing options are summarized in the following table

From	To	Operation

AMEM (Read datapath)	LLC Grid	Normal data input to the Grid
	VPU	Grid bypass operation (AMEM->VPU) or for VPU preprocessing data before the Grid
	AMEM (Write datapath)	Memcopy or tensor reshaping operations which only require the read/writeback datapath elements.
LLC Grid	AMEM (Write datapath)	Writeback of Grid output data directly to AMEM
	VPU	VPU processing of Grid output, such as “fused” activation functions
VPU	AMEM (Write datapath)	Writeback of VPU output directly to AMEM
	LLC Grid	Recirculation of VPU output back to the LLC Grid, or for VPU preprocessing data before the Grid

Table 6-3: UIE Datapath Switchbox Connections

6.2.4 UIE Weights Datapath



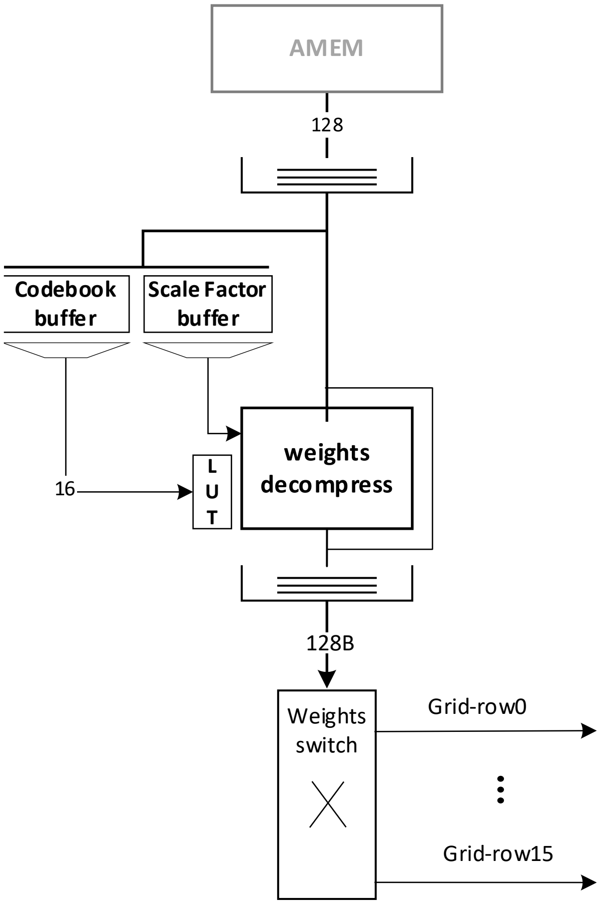
Weights and “horizontal” MatMul data flow from AMEM through the Weights Datapath block to the Grid. This includes the following sub-blocks

- Weights Decompression
- Weights Switch
- Data masking and Log Conversion for Horizontal MatMul data
- Exponent-Bias conversion to match common Grid data-type with 5-bit exponent
- Per-grid-row buffering to supply “horizontal” flowing weights and data to the grid-rows in a fine-grained manner

Weights may be received from AMEM as compressed 4-bit values along with decompression information, or may be received uncompressed in 8-bit (LnsI4F3) or 16-bit (LnsI5F10) format. MatMul-data may be received in FP8 or FP16 format, which must be converted to LNS.

The weights/data are distributed to small buffers on each grid-row which serve as FIFOs to decouple the transport of data from AMEM to the grid, and can store up to 4K bytes per grid-row. From these small buffers, the weights are transferred to the LLCs within each grid-row.

6.2.4.1 Weights Decompressor



Weight compression follows the scheme described in section 13, with 4-bit weight (index) values and associated “codebooks” and scaling factors to reconstruct LnsI4F3 weights.

Hardware to decompress the values (and reconstruct LnsI4F3 weights) is located in the weights-reading path between AMEM and the LLC Grid.

Compressed weights are read from AMEM in “superblocks” as defined in section 14.2. In brief, each superblock consists of a header containing a 16-entry codebook and 128 one-byte scaling factors, followed by 128 blocks of compressed weights. The superblocks must be aligned in AMEM on 128-byte boundaries, and are defined as an integral number of 128-byte words. This structure and alignment are convenient for hardware, so the data transferred from AMEM to the weights-decompression logic does not need to be re-aligned prior to decompression.

When decompression is enabled, this logic will capture the header of each superblock (first one or two 128-byte words) into a codebook buffer and scale-factor buffer. The header contains the codebook and scaling factors for multiple blocks of quantized weights which immediately follow the header. The decompressor can then decode 128 weight values per cycle. The 4-bit weight indices are hydrated into LnsI4F3 (8-bit) weight values, and forwarded to the weights-switch as shown in the figure above.

When decompression is not enabled, the 128-bytes of weights (or activation data for MatMul ops) flow through the block unchanged to the LLC Grid

6.2.4.2 Weights Switch

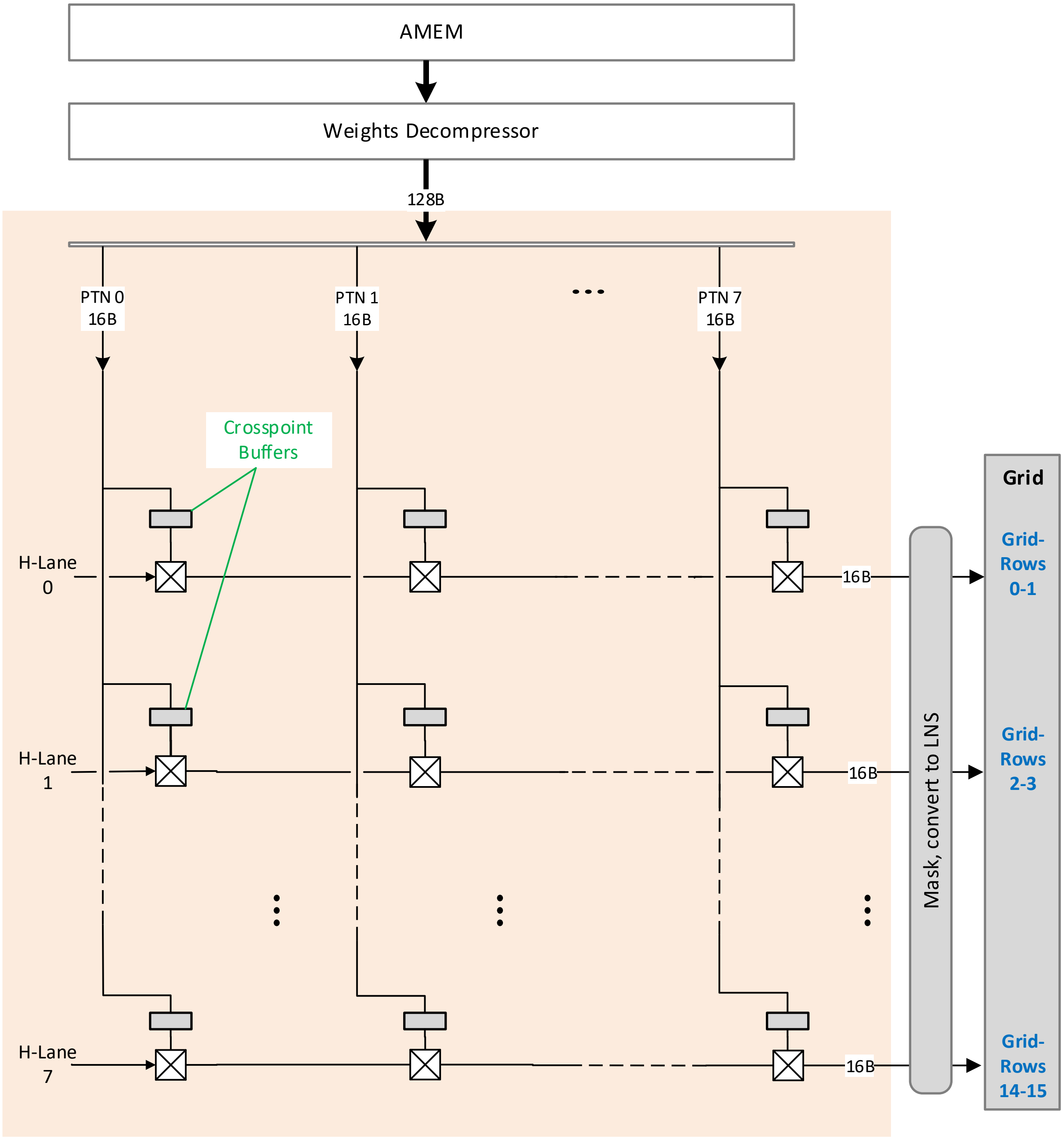


Figure 6-8: Weights Switch

Weights parameters and MatMul “Matrix A” data are transferred from AMEM to the side of the LLC Grid using a dedicated buffered crossbar switch.

Data read from AMEM consists of 8 lanes of 16 bytes each, i.e., 8 contiguous partitions. From the weights-reader port, this data passes through a buffered crossbar which switches 8 “vertical” lanes (from AMEM) to 8 “horizontal” lanes (***H-lanes***) feeding the side of the LLC Grid. This is shown in Figure 6-8. (Note: for simplicity, the switch is illustrated without pipelining stages)

As seen in the figure, each vertical lane corresponds to one partition from AMEM. Each horizontal lane supplies 2 grid-rows in the LLC Grid. The switch moves data from any AMEM partition to any H-lane. On average, each grid-row receives 8 bytes per cycle.

Data may be delivered *transposed* or *row-shifted* to the side of the grid:

- **Transposed**: distributes 8 partitions (128 bytes) from one tensor row onto 8 different H-lanes. This supports MatMul cases where the Matrix A data is ready to apply in this transposed fashion (such as when each physical row in memory corresponds to a logical column of the matrix).

  In this case, data partition 0 is switched onto to H-lane 0, data partition 1 to H-lane 1, etc. This effectively transposes the “Matrix A” data on the fly as discussed in section 6.1.3.1. Note that each partition’s-worth of data now corresponds to 16 different *rows* from the same matrix *column*, so these 16 bytes must be delivered to 16 different virtual grid-rows, which corresponds to 2 physical grid-rows (supporting 8 virtual grid-rows per physical grid-row).

  This mode also supports weight decompression where the compression-blocks comprise one input channel for multiple output channels. The decompressed weights are produced as row vectors, which must be transposed to column vectors to supply one weight to each virtual grid-row (=output channel)

  For FP16 data, each vertical partition (16 bytes) contains 8 two-byte values, so when transposing this data, both of partition 0 and partition 1 are switched to H-lane 0, partition 2 & 3 to H-lane 1, etc. Each partition’s-worth of data now corresponds to 8 different rows (of two-byte values) for one matrix column, which must be delivered to 8 virtual grid-rows occupying a single physical grid-row.

- **Row-shifted**: loads all 8 partitions (128 bytes) from one tensor row into one H-Lane at a time, then shifts the data horizontally over the next 8 cycles. This supports MatMul cases where the Matrix A data is shifted into the grid one column at a time, so each row of incoming data corresponds to one virtual grid-row. This also supports weights transfer in that same format.

  This is accomplished by loading a full AMEM data row (128 bytes) in the crosspoint buffers at each H-lane of the switch, and then unloading 1 buffered partition per cycle on each of the H-lanes for 8 cycles. This loading and unloading can be performed concurrently on all H-lanes, to sustain a throughput of 16B/cycle per H-lane, or 8B/cycle per grid-row, or 128B/cycle in aggregate. Additional buffering in the Weights Datapath further distributes this as one byte per cycle per virtual grid-row.

The weights switch also allows any rotational alignment of the incoming data. The logically “first” partition may be any vertical partition lane, and the weights switch can adapt to the rotation. Consider the transposed-data example above: if the logically first partition were in vertical partition lane 5, then data partition 0 would be switched onto H-lane0, partition 6 into H-line 1, etc. This allows weights and horizontal MatMul data to have any legal partition alignment in AMEM.

6.2.4.3 Zero Masking for Horizontal MatMul Data

For **transposed** data format, 128-bytes of data are read from AMEM per row per cycle, and this would appear as one *column* of “Matrix A” data. Each column of data may consist of blocks of valid rows, with invalid rows sprinkled between (e.g., when processing multiple “short” tensors in a single tile, or a tensor which does not use all of the rows of the grid). The invalid entries would be statically located, appearing as statically invalid (wasted) virtual grid-rows.

The zero-masking feature for the Horizontal flow may be programmed with a unique 16-bit mask per H-lane. This zero-mask can be configured statically per-trip, for the transposed case, to force the invalid virtual grid-row data to zero. Given 8 H-lanes, the zero-mask is a 128-bit vector. The Weights Datapath logic provides a table of four (128-bit mask) configurations to select from.

Note: For transposed FP16, this requires two 16-bit masks per H-Lane, with cycle-by-cycle selection of the appropriate mask, since the per H-lane 16B (8xFP16) data is time-multiplexed over two physical grid-rows. This is described in further detail in the Pyxis UIE Sequencers Functional Specification document.

For the **row-shifted** data format, up to 128 bytes of data are read from AMEM for each row per cycle, and subsequently shifted, one partition at a time, “horizontally” into a particular grid-row. The zero-masking feature is only intended to support a per-trip static configuration, rather than a cycle-by-cycle configurable zero mask. Therefore, this is not sufficient for masking invalid entries (columns) shifted in over multiple cycles on the same H-Lane in the row-shifted mode. Masking of invalid data (columns) sprinkled into row-shifted data must be taken care of by pre-masking (zeroing) those data elements in the prior trip which created the data.

6.2.4.4 Linear-to-Log conversion for Horizontal MatMul Data

Horizontal MatMul data (“Matrix A”) must be converted to LNS before the grid. See Section 6.2.1.5 above for description of Linear-to-log conversion

- FP8 data is converted to **LnsI5F3** (a 9-bit format). This maps the 3-bit mantissa from FP8 to 3-bit log fraction, but maps the 4-bit exponent of FP8 to 5-bit log integer. The 5-bit value is necessary to allow for mixed-precision accumulation where some operands are FP8 and others are FP16, by mapping all data to a common exponent size with common exponent bias for the duration of the dot product accumulation.
- FP16 data is converted to LnsI5F10, but only 4 values per cycle per grid-row are supported with 16-bit data, since this matches the available throughput from AMEM (8 bytes/cycle/grid-row) in the horizontal flow.

6.2.4.5 Exponent Bias Adjustment for Mixed-Precision Support

Weights and horizontal MatMul data support mixed precision (8-bit /16-bit) formats, which must be sent to the grid with a common Exponent Bias. This is necessary to enable mixed precision accumulation, where some input channels’ weights are LnsI4F3 and some are LnsI5F10, or similarly for MatMul where some horizontal input data are FP8, and some are FP16 (and converted to the associated

8/16-bit LNS format in the Lin2Log step above). Using a common EB allows accumulating dot-products from both input precisions in the same accumulator. To support this, the 8-bit data (FP8 or LnsI4F3) are promoted to use a **5**-bit exponent (same as FP16/LnsI5F10), as **LnsI5F3**.

To convert from 4-bit to 5-bit exponent requires adjusting the Exponent Bias. The conversion is similar to that described in section 6.2.1.4.2 where we apply an integer-power-of-two scaling factor to the exponent:

```
       I5                = I4 + EB_Adj_I4I5
where
       EB_Adj_I4I5   = EB_I4F3 – EB_I5F3
```

The `EB_Adj_I4I5` scaling factor is a programmable parameter (per trip). This is applicable for LnsI4F3 weights (in 1x1/linear/3x3 conv layers) **or** for FP8 data in MatMul layers

EB Adjustment may also be used to **scale FP16** data by an arbitrary integer power of two by programming the `EB_Adj` to increment or decrement the exponent by the scaling factor. This is available for FP16 horizontal MatMul data when converting to LnsI5F10, and then scaling the LnsI5F10 value. The I5F10 → I5F10 scaling is:

```
       I5_out            = I5_in + EB_Adj_I5I5
where
       EB_Adj_I5I5   = -log2(scaling_factor)
```

The `EB_Adj_I5I5` scaling factor is also a programmable parameter (per trip).

Note: Separate **`EB_Adj_I4I5`** `and` **`EB_Adj_I5I5`** values may be applied for FP8 and FP16 in mixed-precision trips. Hardware provides support for multiple programmable `EB_Adj` values per trip.

6.2.4.6 Per-Grid-Row “Horizontal” Weights Buffers

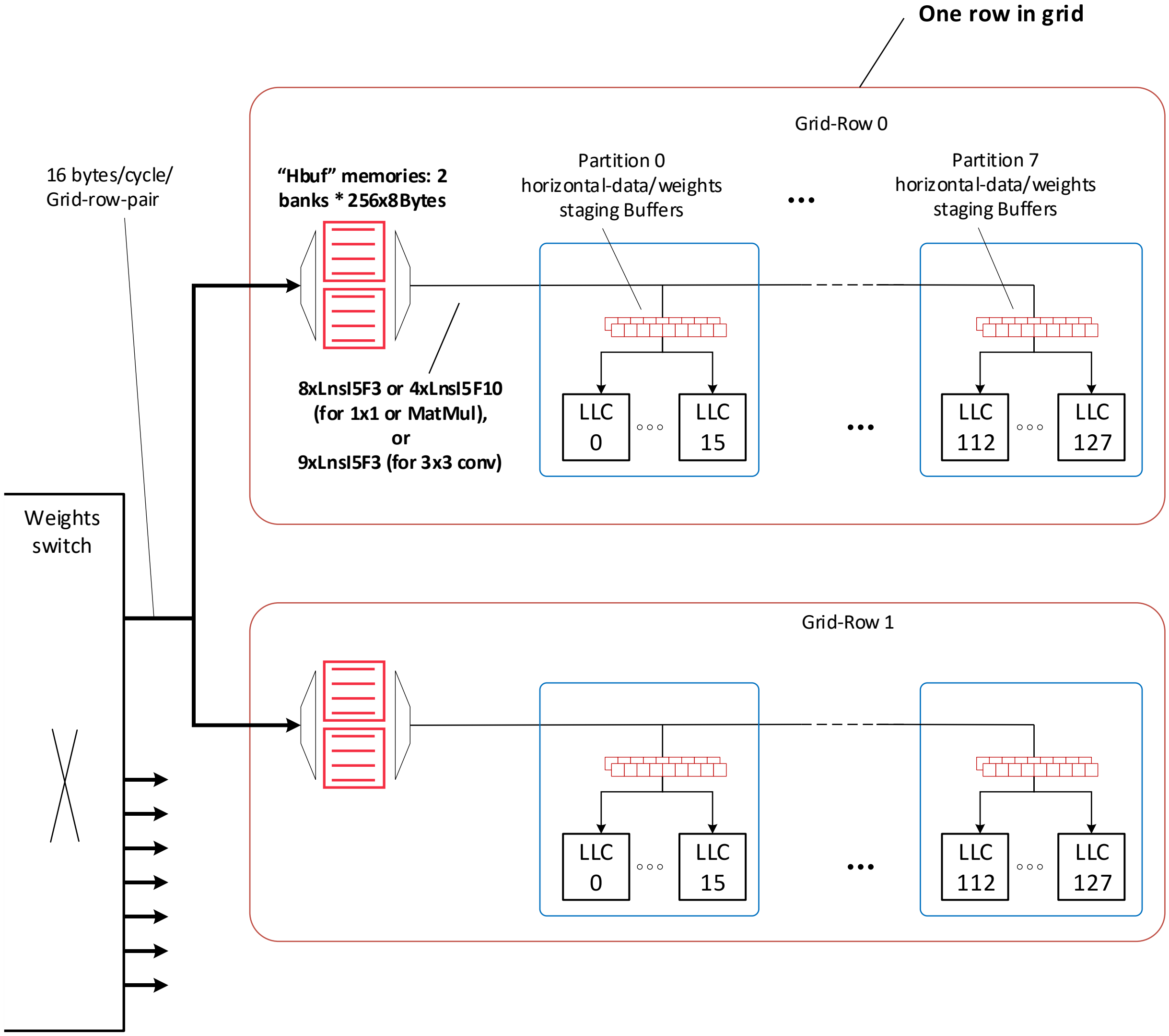


Figure 6-9: Horizontal Weights/Data buffering for the Grid

After the weights switch, zero-masking and linear-to-log conversion, the horizontal weights/data flows into **Horizontal-Buffer (Hbuf) memories**. These are per-grid-row 4K bytes buffers, to decouple the weights/data transfer from AMEM to the LLCs. These memories operates as a block-based FIFOs, which can buffer 512 bytes for each of the 8 virtual-grid-rows.

For “row shifted” MatMul, the data may be written into the buffer in blocks of 128 bytes per virtual-grid-row, but subsequently read out as 8 bytes per virtual-grid-row. For 3x3 convolution, the kernels may be read as 8 bytes from two internal banks, and the hardware automatically aligns this to select 9 bytes.

From the Hbufs, the data is broadcast to one or more (or all) partitions on a per-grid-row basis. In each grid-row partition, there is a **Kernel buffer**, which is a double-buffered register for the current and next-active weights. This provides the 3x3 kernel (9xLnsI5F3), or multiple 1x1 weights (8xLnsI5F3 or 4xLnsI5F10), used by all of the 16 LLCs in that {row, partition}. These are dynamically loaded under microcode control. These buffers are also used for data x data matrix multiplication (“MatMul”), where they buffer 8-bytes from the horizontal-data flow (again, 8xLnsI5F3 or 4xLnsI5F10).

6.3 LLC GRID

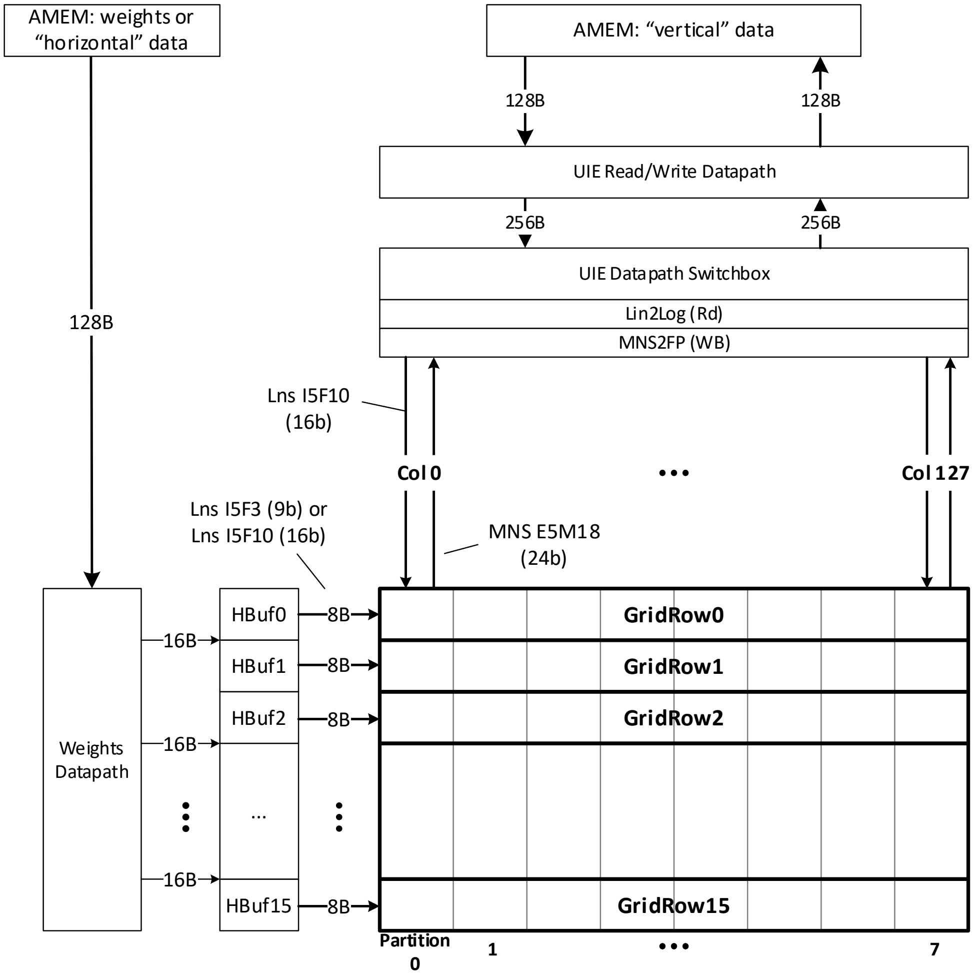


*Figure 6-10: LLC Grid: 16 Rows * 8 Partitions/row * 16 Columns/Partition*

Grid Structure:

- 16 ***grid-rows*** x 128 columns.
- Each column instantiates a ***Leaf Level Convolver*** (***LLC***). Total 2048 LLCs per grid. Each LLC can compute a dot-product in any of the following ways
  - 9 weights x 9 data inputs per cycle for 3x3 convolution with FP8 data or FP16 data.
  - 8 weights (or data) x 8 data inputs per cycle for 1x1 convolution or general MatMul using 8-bit input data and weights.

  - 4 weights (or data) x 4 data inputs per cycle for 1x1 convolution or general MatMul using 16-bit input data and weights

  "Vertical" data inputs are converted from FP8/16 to LnsI5F10 externally to the Grid. For FP16, the external datapath limits execution rate to once per two cycles, but within the grid, vertical FP16 data execution is supported every cycle.

  "Horizontal" data inputs are converted from FP8 to LnsI5F3, or from FP16 to LnsI5F10. For FP16, the external datapath limits execution rate to once per two cycles, and within the grid, execution for FP16 horizontal data operates at an average of once per two cycles.

  Weights are natively in LNS format, either LnsI4F3 (8-bit) or LnsI5F10 (16-bit). Again, the external datapath limits the execution rate for 16-bit weights to half speed.

- LLCs in each grid-row are organized as 8 ***partitions*** with 16 LLCs per partition.

- LLCs in each partition represent 16 columns of the same filter (or output channel) for convolution operations. For 3x3 convolutions, multiple partitions of LLCs can be ganged-together to form wider filters, up to 128 columns. Any power-of-two division of partitions is possible. For 1x1 convolutions, all 128 columns operate on the same filter

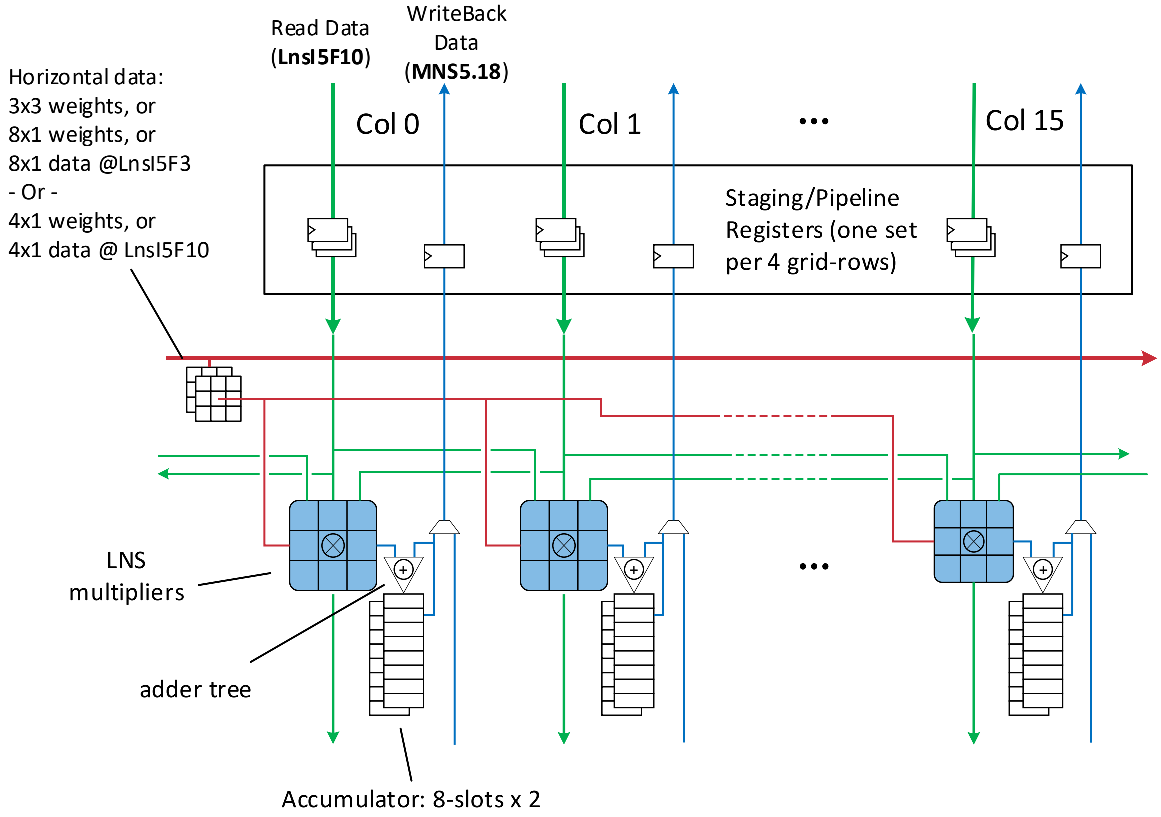


Figure 6-11: One Partition of 16 LLC columns

6.3.1 Vertical Data Flow

Activation Data flows "down" from the top of the grid, 128 columns-worth of LnsI5F10 data per cycle, i.e., the vertical data channel is 128x16-bit values. Logic external to the grid converts incoming data from FP8 or FP16 to a common LnsI5F10 format. Data is buffered in ***staging registers*** logically located at each grid-row[3]. These staging registers collect data over multiple cycles to provide a block of tensor-adjacent values to each LLC, rather than one value at a time..

For 3x3 convolution, the staging registers at *each* column collect data from the prior 3 cycles corresponding to 3 contiguous tensor-rows for the same input channel, for that column *and* the

[3] The staging registers are implemented for a group of grid-rows, and shared among multiple adjacent grid-rows. In Pyxis, there is one row of staging registers for every ***4*** grid-rows. This reduces the total number of staging-register flip-flops by a factor of four, and also reduces the pipeline latency to fill the grid from 16 cycles to 4 cycles.

adjacent LLC columns on each of left and right (higher and lower numbered columns). In this manner, each LLC collects the desired 3x3 data inputs centered at that LLC's row, column location.

For 1x1 convolution (weights x data) with **8-bit** weights and **FP8** data inputs: the staging registers collect data over 8 cycles corresponding to 8 input channels (for that column). Once 8 values have been collected, these are used for dot-product computations against the 8 weights for those 8 input channels, corresponding to one output channel per cycle. This dot-product is computed for 8 different output channels over the next 8 cycles. The staging registers are double-buffered so the next set of 8 input channels' values may be collected while the current set of 8 is used for calculations by the LLCs. In this mode, there is no exchanging of staging register data among adjacent columns.

For 1x1 convolution with **16-bit** weights and **FP16** data inputs, the staging registers collect data over 8 cycles corresponding to **4** input channels. Microcode must insert 4 rows of additional zero data every 8 cycles on the vertical path, specifically 3 rows of zeroes, 4 rows of data, and 1 final row of zeroes. The dot-product of 4 input (data) channels x 4 corresponding weights is then computed **8** times for 8 different output channels over the following 8 cycles.

For 1x1 convolution with **mixed-precision, 8-bit** weights and **FP16** data inputs, the vertical data flow operation is the same as for 8-bit weights and FP8 data.

For 1x1 convolution with **mixed-precision, 16-bit** weights and **FP8** data inputs, the vertical data flow operation is the same as for 16-bit weights and FP16 data.

For MatMul (data x data) with **FP8** data inputs: the staging registers collect data over 8 cycles corresponding to **8 rows** (for that column) from the "vertical" data flow, also referred to as **Matrix B**. Once 8 values have been collected, these are used for dot-product computations with **8 columns** (for one row) *per cycle* from the "horizontal" data flow, also referred to as **Matrix A**. This dot-product is computed for 8 different Matrix A rows over the next 8 cycles. In this manner, the LLCs compute 8 output data rows per grid-row in a time-multiplexed fashion. Hence each (physical) grid-row supports 8 *virtual* grid-rows

For MatMul with **FP16** data inputs on the *vertical* path, the process is the same as for **FP8** inputs.

For MatMul with **FP16** data inputs on the *horizontal* path, the staging registers collect data over 8 cycles corresponding to **4** matrix B rows. Microcode must insert 4 rows of additional zero data every 8 cycles on the vertical path, specifically 3 rows of zeroes, 4 rows of vertical data, and 1 final row of zeroes. Over the next 8 cycles, these 4 matrix B rows are used for dot-product computations with **4 columns** *per cycle* from one row of Matrix A, and this is repeated for 8 rows of Matrix A over the 8 cycles. In this manner, we compute 8 output data rows per grid-row in time-multiplexed fashion; again , each (physical) grid-row appears as 8 virtual grid-rows.

6.3.2 Horizontal Data Flow

Weights, and "matrix A" data for MatMul, flow horizontally across the grid. Each partition (in each grid-row) has a double 9-weight buffer.

For 3x3 convolution, each grid-row may be divided into 1, 2, 4 or 8 filters (i.e., output channels). For "narrow" tensors with Width ≤ 64 columns, two output channels may be computed per grid-row by dividing the LLCs into two subsets of 4 partitions each. Similarly for tensor-width ≤ 32 columns, each grid-row may be divided into 4 output channels x 2 partitions, or for width ≤ 16, divided into 8 output channels x 1 partition each.

Logic at the side of each grid-row can broadcast one 3x3 kernel (9 one-byte LNS values) per cycle, directed at the appropriate subset of partitions. The kernel weights are loaded into double-buffered kernel-buffers at each partition. At the appropriate time, the new kernels are made active concurrently for all partitions in the grid-row, just-in-time to meet the data arriving for the next input channel. This is all under hardware and microcode control.

For 1x1 convolution, each grid-row will compute up to 8 output channels over 8 cycles in a time-multiplexed manner. Each cycle, the grid-row will operate as a different output channel context, and must be provided with a new set of weights for the output channel. As described above, 1x1 convolution uses a grouped-MAC scheme where the staging registers first collect values for the same tensor-row across 8 input channels. These 8 values are then used to compute the dot-product with 8 corresponding weights. Each cycle, the LLC switches to the next output channel, and re-uses the same 8 data values (from the staging registers) with the new output-channel's weights.

To support this operation, logic at the side of each grid-row will prepare 8 weights per cycle (corresponding to 8 input channels for one output channel) and broadcast this to ***every*** partition in that grid-row. These are one-byte LNS values, so this corresponds to 8-bytes per cycle broadcast horizontally to every partition, independently per grid-row. The 3x3 kernel-buffers in each partition are re-purposed as 1x8 weights vector buffers, and the weights are loaded and made active just-in-time to be used by the LLCs.

For Matrix Multiplication, the horizontal flow carries LNS data values from "matrix A" instead of weights, but the function in each grid-row is very similar to the 1x1 case. Logic at the side of each grid-row will broadcast a 1x8 vector of 8 consecutive data-columns for one particular data-row of matrix A. This is used to compute the dot-product with an 8x1 vector from matrix B at each LLC. This cycles through 8 matrix-rows from horizontal "matrix A" since each grid-row appears as 8 virtual grid-rows. Note that the horizontal data here is LNS, and FP8 data values are converted to LnsI5F3 for the horizontal flow in MatMul. This can be contrasted with the vertical flow, where FP8 data is converted to LnsI5F10.

For Matrix Multiplication with FP16 data inputs, the "matrix A" data is converted to LnsI5F10 outside of the grid. Each LnsI5F10 value occupies 2 bytes in the horizontal-flow datapath. Due to AMEM bandwidth limitations, only 8 bytes per cycle are broadcast horizontally per grid-row, comprising a 1x4 vector of LnsI5F10 values for one row of matrix A, each cycle. This is used to compute a dot-product with the 4x1 vector of LnsI5F10 values flowing vertically for matrix B. Each cycle, a new 1x4 vector is broadcast for matrix A, cycling over 8 virtual grid-rows per physical grid-row over the period of 8 cycles.

6.3.3 Leaf Level Convolver (LLC)

6.3.3.1 LLC pipeline: 3x3 Convolution

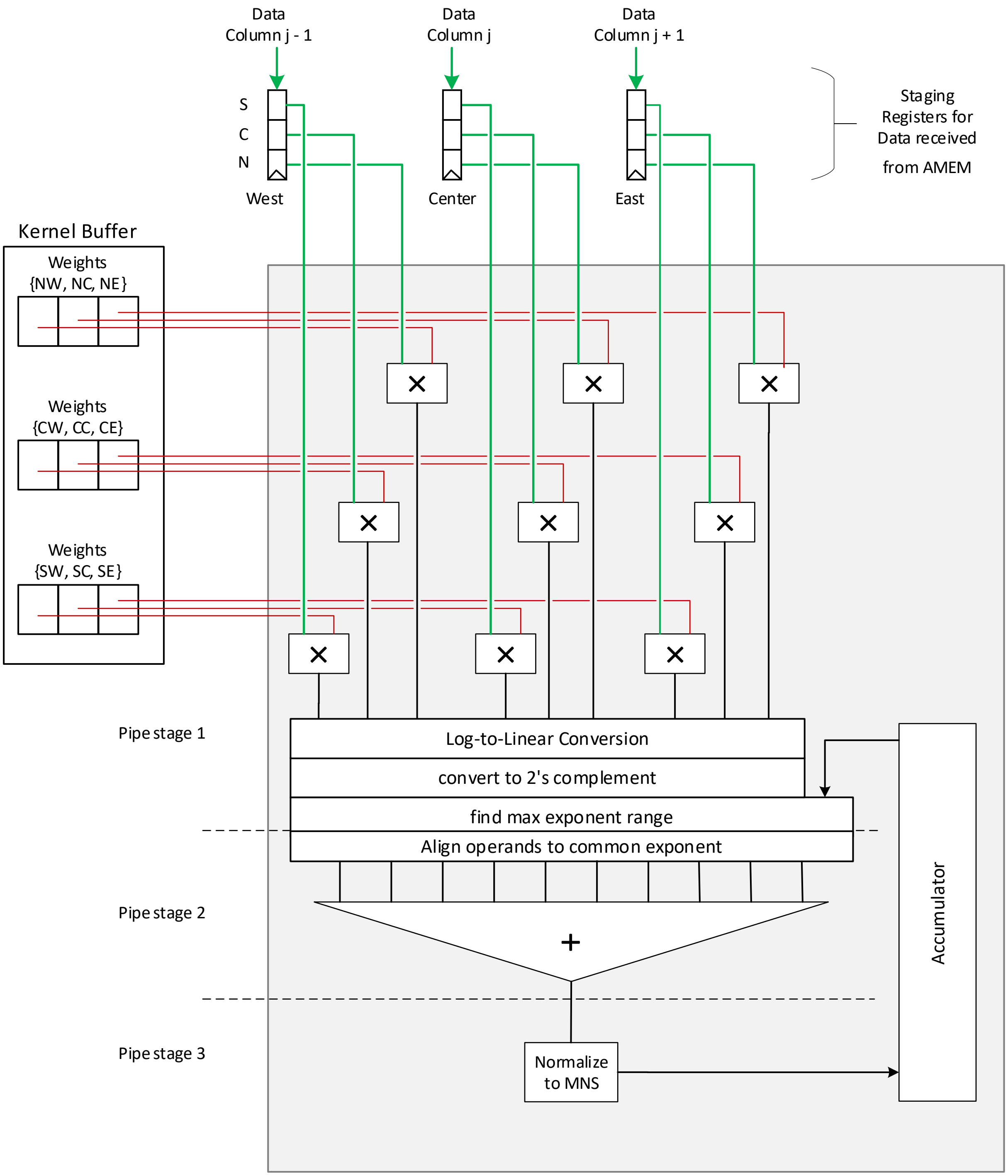


Figure 6-12: LLC Block Diagram: 3x3 Convolution

LLC Summary:

- 9 LNS multipliers, with LnsI5F3 weights x LnsI5F10 data inputs
- 9 Log-to-Linear converters between multipliers & adders
- Converts each product to 2’s complement format
- Pipelined adder tree sums 9 products and the accumulated value
- Normalizes the sum and stores in accumulator in MNS format

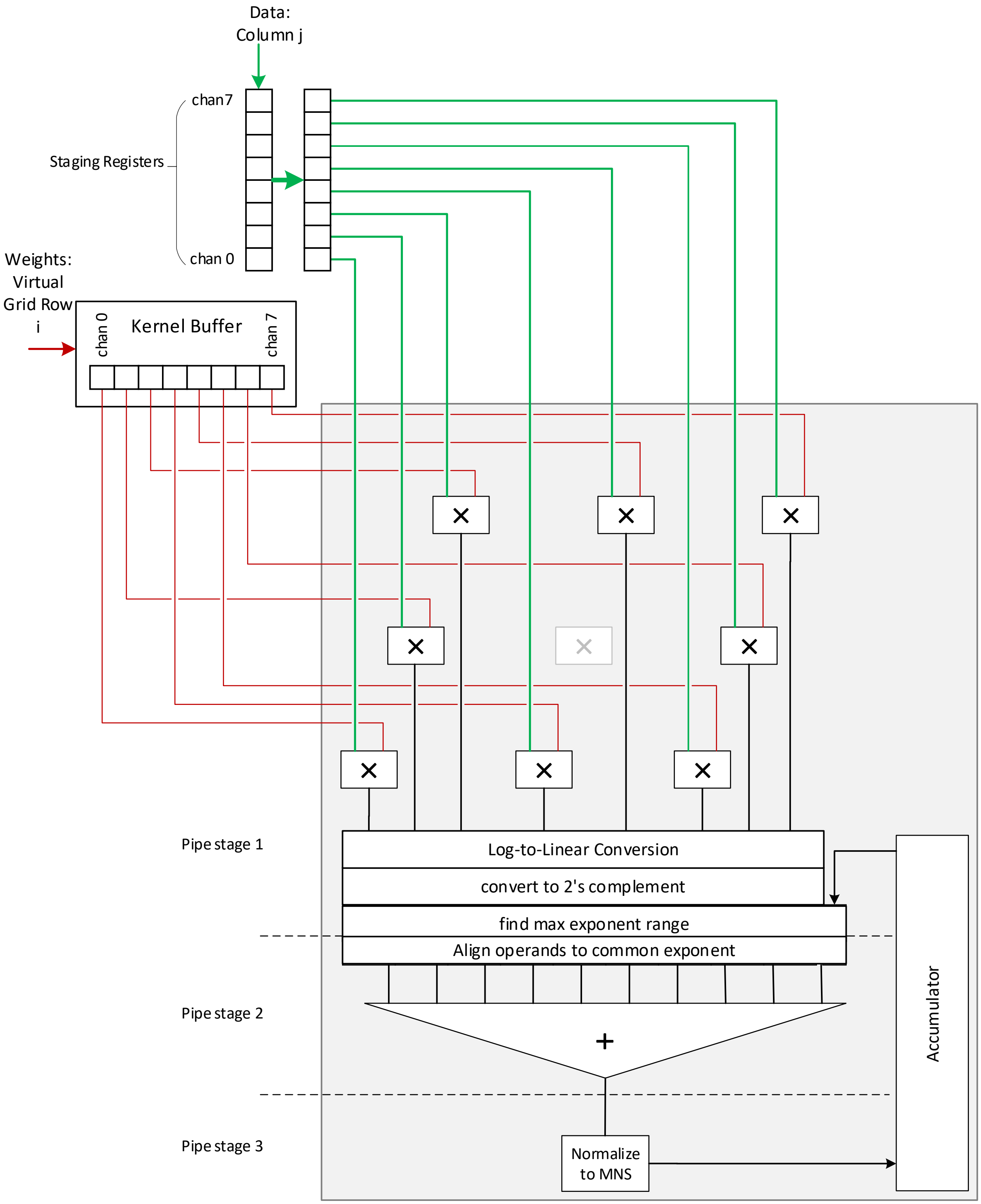
Each 3x3 kernel sums the 9 products of the weights and data of its own value (“C” or center) and the 8 neighbors (NE, N, NW, E, W, SE, S, SW). The data marches down the grid-rows on each clock step. Conceptually, the kernel slides “down” over the tensor data from top to bottom of the tensor, which is equivalent to sliding the data “up” over the kernel. This means that the SE, S, and SW data for clock cycle N will be the E, C, and W data on clock cycle N+1, and will become the NE, N, and NW data on cycle N+2.

The 9 activation values are collected in “staging registers” by shifting-in values over three cycles, and sharing the 3x1 vectors with the east and west neighbors of each LLC. On the fourth cycle, one value is shifted out and a new value is shifted in, effectively sliding the 3x3 window “down” one row, and this is repeated for subsequent rows.

The nine products are computed by a simple addition in LNS domain. To add all the products, as well as the accumulator value together, the 9 products are converted to a floating-point representation in the linear domain using the log-to-linear conversion described in section 4.7 above. The 9 products are also converted to 2’s complement format. Then the 9 products and the current accumulator value are aligned to a common exponent.

The 10 values are then summed in an adder tree. The result is normalized to MNS format, and stored back into the accumulator.

6.3.3.2 1x1 Convolution Support



For 1x1 convolution, the LLCs can perform 8 multiply/add operations per cycle by grouping data across input channels:

- Tensor data received from AMEM is buffered in “staging registers” for each LLC column. These registers collect 8 data elements corresponding to 8 *input* channels for the same spatial location (i.e., same output tensor-row and column). Unlike 3x3 convolution, there is no sharing of data with the east and west neighbors in the 1x1 case.
- Once 8 data elements have been buffered, this data is used for 8 cycles to compute 8 partial dot products for 8 output channels per LLC. Each cycle, the LLC receives a set of 8 weights corresponding to the current 8 input channels for one output channel. The same 8 weights are provided to all LLCs in a grid-row.
- The LLC then computes the (weights * data) dot-product for those 8 input channels and one output channel, and accumulates the result in one accumulator slot. This uses 8 out of the 9 multipliers in the LLC, with one multiplier unused. However, the rest of the computation flow is the same as for 3x3 convolution.
- This continues for 8 cycles; each cycle the LLC receives 8 weights for the next output channel (for the current set of 8 input channels), and subsequently accumulates using the next accumulator slot. Over 8 cycles, a single LLC can compute partial results for 8 output channels.
- During these 8 cycles, the staging registers will collect the *next* set of 8 input channels’ data (for the same spatial location), and the process repeats.
- This continues until all input channels have been seen for a given spatial location, i.e., a tensor-row, at which point accumulation is complete for that tensor-row. Results are now written back from the accumulators to AMEM.
- The input data can then advance to the next spatial location (next tensor-row), to compute dot-products for all input channels for tensor-row.

In summary: each LLC grid-row computes results for 8 output channels for the same tensor-row. Grid-row 0 computes output channels 0-7, grid-row 1 computes output channels 8-15, etc. This is done across the entire grid, one tensor-row at a time.

6.3.3.2.1 Support for 16-bit weights for 1x1 Convolution

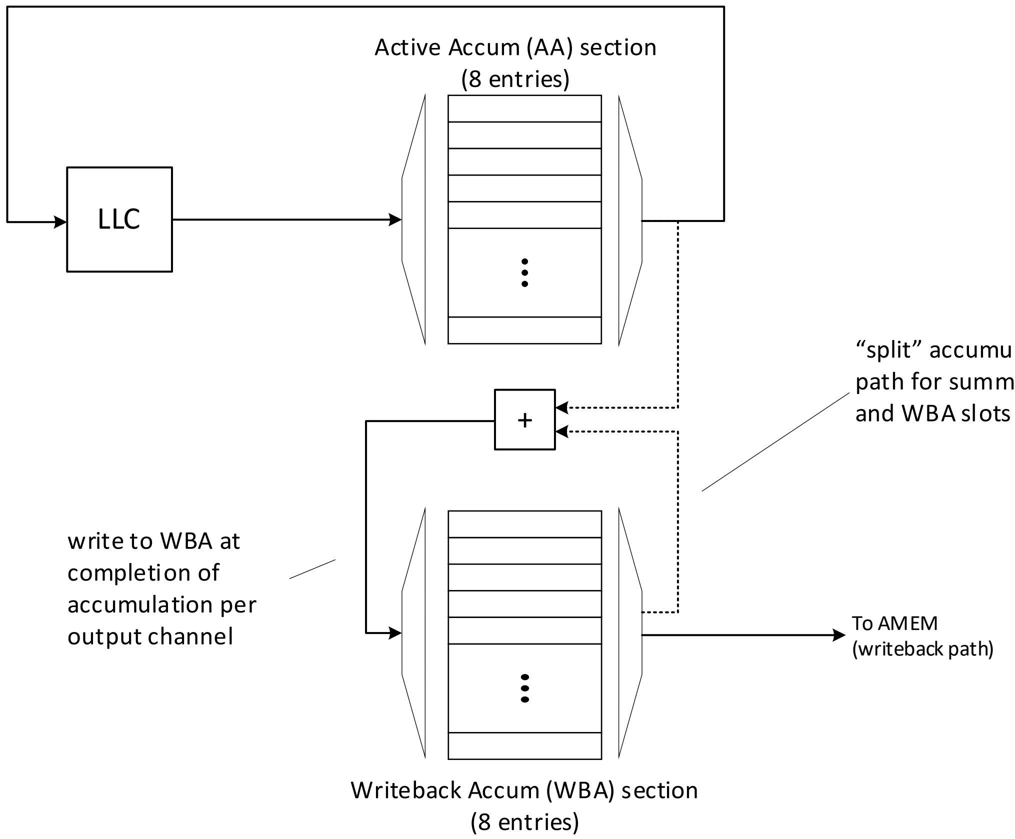
In each LLC, four of the nine multipliers support 16-bit inputs from the horizontal flow (i.e., the weights for 1x1). This matches the datapath bandwidth which allows for 8 bytes/cycle for horizontal data flow. This allows for **four** 16-bit weights per cycle computation

6.3.3.3 Matrix Multiplication Support

In this document, MatMul refers to general data x data matrix multiplication. This is effectively identical to the 1x1 operation described above, but the weights path is overloaded to provide the “horizontal” matrix data flow. So, for matrix multiplication computing **C** = **A x B**, matrix **A** is the horizontal data (instead of weights), **B** is the usual vertical data, and **C** is the result computed in the grid.

The same computation bandwidth metrics apply for 1x1 convolution and general MatMul.

6.3.4 Accumulators


Active Accum (AA) section
(8 entries)
LLC
"split" accumulation path for summing AA and WBA slots
+
write to WBA at completion of accumulation per output channel
To AMEM
(writeback path)
Writeback Accum (WBA) section
(8 entries)

Accumulators summary:

- Store partial sums locally for each LLC
- 2 Read / 1 Write register files
- 8 accumulator *slots* x 2 (double-buffered), per LLC

Accumulators can store sums (or partial sums) for 8 concurrent contexts per LLC. These eight contexts may be 8 output rows of a matrix multiplication, 8 output channels for 1x1 conv, or 8 output tensor rows for 3x3 convolution. There are two banks of accumulators, denoted the *Active Accumulator* (**AA**) and *Writeback Accumulator* (**WBA**). Accumulators store data in **MNS** format {S, E5, Mxx} which is a floating-point format with 2's complement mantissa representation. The number of mantissa bits differs for the AA and WBA; the AA has {S, E5, M13}, and WBA has {S, E5, M18}. So, AA is 19 bits/slot and WBA is 24 bits/slot.

The LLCs compute results to completion for each set of 8 output data-rows – i.e., perform the dot-products and cross-channel sums for all input channels for the associated input data-rows. When computation is complete for a set of 8 data-rows, the results must be written-back to AMEM, and computation continues for the next set of 8 output rows.

The AA and WBA banks provide double-buffering to allow the next set of results to be computed concurrently with the writeback of the prior set. The AA bank is used for computing the cross-channel sums, concurrently on all grid-rows, while the WBA bank is normally written by the LLC only when the cross-channel computation is complete, and unloaded (to be written back to AMEM) sequentially across all the grid-rows. Given 16 grid-rows and 8 accumulator slots, it then requires at least 128 cycles to unload all WBA slots from the full grid.

6.3.4.1 Split Accumulation

Since the accumulators have a limited number of bits of (mantissa) precision, there is a risk of *swamping* when accumulating a large number of small values. This can occur when the accumulated partial sum is sufficiently larger than the incoming values that these smaller values no longer contribute to the accumulation, or the precision of the smaller values is lost. The exposure is most acute for high-channel-count layers deep in a neural network, or some transformer MatMuls where the common matrix dimension is very large.

Pyxis helps mitigate this exposure by splitting the accumulation into two cascaded phases, and providing additional fraction bits in the 2nd phase. In the first phase, values are accumulated in the Active Accumulator (**AA**) for a configurable number of input channels (e.g., 64). The number of first-phase channels is referred to as the “**split chunk”** size. In the second phase, the accumulated AA values are further accumulated into the Writeback Accumulator (**WBA**), and the AA is cleared. The WBA provides additional fraction bits in its (extended) floating point format, to further mitigate swamping for very large-scale accumulations. The AA can then accumulate another set of input channels (e.g., 64) after which it is again added into the to the WBA and cleared.

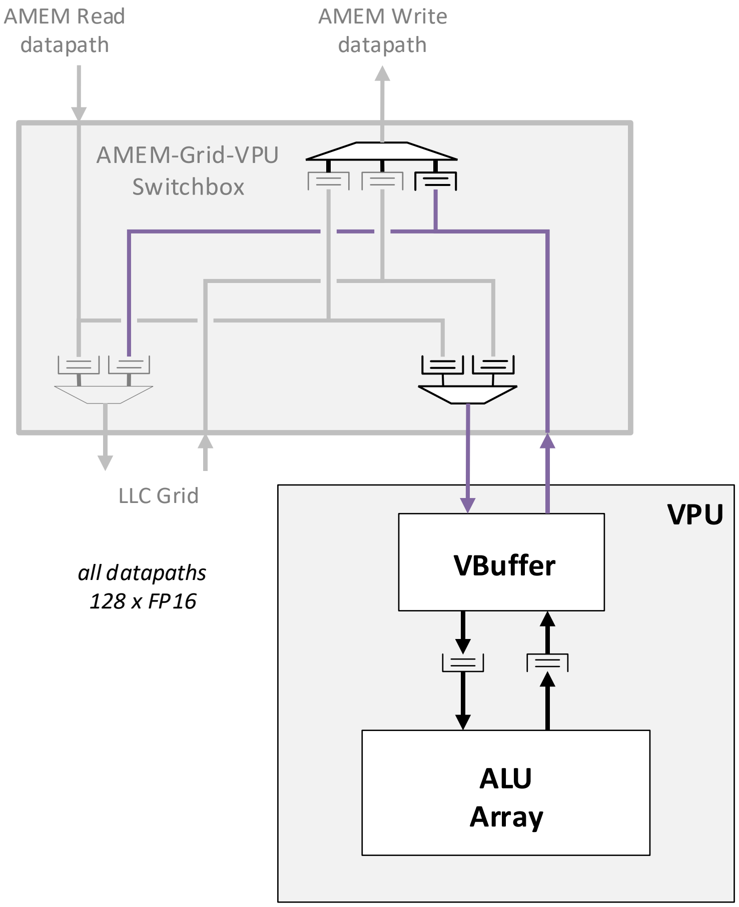
This repeats for each **split chunk**, so that we accumulate a limited number of values in the AA, then accumulate the sum-of-split-chunk partial sums in the WBA. This implements a two-level tree summation. Once all input channels are complete, the WBA is off-loaded to AMEM in the usual way. In a simple analysis, this extends the accumulator precision to an effective width of 18 mantissa fraction bits.

The splits are under microcode control, instructing the LLCs to periodically read *both* the WBA and AA, sum these values, and write the result back to the WBA and (effectively) clear the AA. This can be executed concurrently with normal accumulation of incoming dot-products. (The secret: at the beginning of each split chunk (1st phase), the AA would still hold the values accumulated from the prior split-chunk, and so this time period may be used to read the AA and accumulate it with the WBA (2nd phase) before the AA is overwritten by the initial accumulation of the new split-chunk.

However, this *overloads* the writeback accumulator slots. Normally, the WBA buffers data from the *prior* accumulation for a group of 8 output rows (for 3x3, MatMul) or 8 output channels (for 1x1). The split-accumulation technique also uses the WBA for the *currently active* accumulation group. Hardware will ensure that the prior-group’s WBA entries have been fully unloaded to AMEM before re-using the

WBA for the current-group’s split accumulation, and will **stall the LLCs** if the prior writeback is still pending when a split accumulation step is required.

6.4 Vector Processing Unit



The Vector Processing Unit (VPU) is a programmable SIMD engine with 128 ALUs, natively processing FP16 or Int16 data at one cycle per instruction. It has a rich set of instructions, programmable Look-up Tables, 16 GPRs and several special registers per ALU. There is one VPU in each UIE.

The details of VPU are beyond the scope of this document. The full VPU specification is available at this link

The VPU is attached to the UIE datapath through a Vector Buffer (VBuffer) memory which holds up to 8 128x128 “tiles”. This decouples VPU processing from AMEM read/write and Grid operation.

6.4.1 VPU Operating Modes

The Vector Processing Unit can be configured to operate in one of several modes for each UIE trip:

- As a standalone processor, terminating the AMEM read datapath & sourcing write data for AMEM, effectively replacing the LLC Grid. In this mode, latency and throughput may be variable and are determined by the VPU algorithm
- As an LLC Grid postprocessor, situated in the writeback datapath between the LLC Grid and datapath switching elements. In effect, the Grid and VPU operate in a two-stage pipeline, with tile-sized (128x128) data blocks using the Vbuffer to couple the stages.

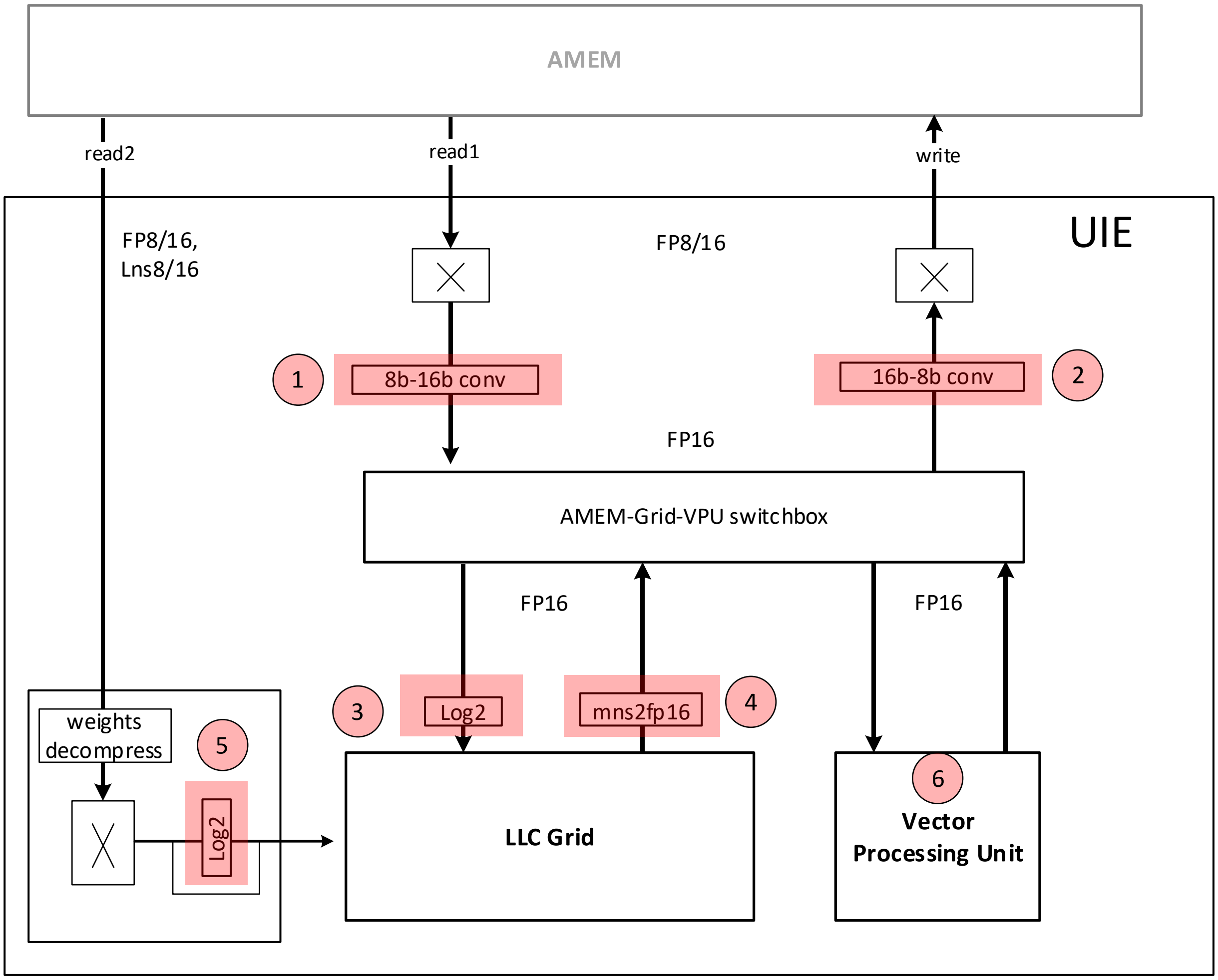
  The primary use is for activation functions using a fixed, single VPU instruction per data element. This can be a ReLU instruction or may use a fixed-latency Lookup Table, which can provide any function of one FP16 value to produce another FP16 value. More complex algorithms may be deployed here as well, depending on throughput requirements per tile.
- As an LLC Grid preprocessor, intercepting data read from AMEM to perform some transform before sending data to the LLC Grid.
- For recirculating data output from the Grid, through VPU and back to the Grid. This provides a “shortcut” to bypass AMEM for writing/reading data for sequential processing. An example for this is support for the “Flash Attention” algorithm which combines matrix multiplication with Softmax in sequential passes through the data.

6.5 Summary of Data Types, EB Adjustments and Type Conversions in UIE Datapath

The following figure & table detail the type conversions implemented as data flows through the UIE. At each conversion point, there are typically two exponent mapping options (for 8-bit or 16-bit data), each with a programmable exponent-bias (EB) adjustment. The appropriate setting for EB adjustment is shown in each case.

Note that Exponent Bias values in Pyxis are typically negative, and EB adjustment can be positive or negative. The EB_adjustment value is **added** to the exponent. Therefore, to *increase* the EB, the EB_adj value would be *negative*. To *decrease* EB, the EB_adj value would be *positive*.

Hardware supports EB_Adjustment values in the range [-32,31] using 6-bit signed integer values. When adjusting the Exponent, in case of exponent-overflow, hardware sets the data to the maximum-magnitude value for the data type. In case of exponent-underflow, hardware sets the value to zero.



Conversion Case		From	To	Exponent mapping, and intended EB_adj programming	Fraction/Mantissa mapping
(1)	AMEM -> Switchbox (8b to 16b conversion)	FP8 E4M3	FP16 E5M10	E5_out = E4_in + EB_adj EB_adj = FP8_in_EB - FP16_out_EB	M10_out = {M3_in, 7'b0}
		OFP8 E4M3		E5_out = E4_in + EB_adj, after normalizing any subnormal input values EB_adj = -7 - FP16_out_EB	M10_out = {M3_in, 7'b0}, after normalizing M3_in in case of subnormal input values
		OFP8 E5F2		E5_out = E4_in + EB_adj, after normalizing any subnormal input values EB_adj = -15 - FP16_out_EB	M10_out = {M2_in, 7'b0} after normalizing M2_in in case of subnormal input values
		FP16 E5M10	FP16 E5M10	E5_out = E5_in + EB_adj EB_adj = FP16_in_EB - FP16_out_EB	Identity
		OFP16 E5M10		E5_out = E5_in + EB_adj, after normalizing any subnormal input values EB_adj = -15 - FP16_out_EB	M10_out = M10_in, after normalizing M10_in in case of subnormal input values
(2)	Switchbox -> AMEM (16b to 8b conversion)	FP16 E5M10	FP8 E4M3	E4_out = E5_in + EB_adj EB_adj = FP16_in_EB - FP8_out_EB	Rounding from M10 to M3
			OFP8 E4M3	E4_out = E5_in + EB_adj, after adjusting to encode output subnormal values EB_adj = FP16_in_EB + 7	Rounding from M10 to M3, and possibly adjusting for subnormal output values
			OFP8 E5M2	E5_out = E5_in + EB_adj, after adjusting exponent to encode output subnormal values EB_adj = FP16_in_EB + 15	Rounding from M10 to M2, and possibly adjusting for subnormal output values
		FP16 E5M10	FP16 E5M10	E5_out = E5_in + EB_adj EB_adj = FP16_in_EB - FP16_out_EB	Identity
			OFP16 E5M10	E5_out = E5_in + EB_adj, after adjusting exponent to encode output subnormal values EB_adj = FP16_in_EB + 15	M10_out = M10_in, after possibly adjusting for subnormal output values
(3)	Switchbox -> Grid (vertical flow, lin->log)	FP16 E5M10	LNS I5Fn	I5_out = E5_in + EB_adj EB_adj = FP16_in_EB- LnsI5_out_EB	Lin2Log mapping with optional truncation to n (<10) fraction bits (padded with zeros) for FP8-in-FP16 container inputs
		FP16 E5M10	LNS I5F10	I5_out = E5_in + EB_adj EB_adj = FP16_in_EB - LnsI5_out_EB	Lin2Log mapping from M10 to F10
(4)	Grid -> Switchbox (MNS->FP16)	MNS E5M18	FP16 E5M10	E5_out = E5_in + EB_adj EB_adj = MNS_in_EB - FP16_out_EB	2's complement conversion and rounding to M10
(5)	AMEM data -> Grid (horiz flow for MatMul, lin->log)	FP8 E4M3	LNS I5F3	I5_out = E4_in + EB_adj EB_adj = FP8_EB - LnsI5F3_out_EB	Lin2Log mapping from M3 to F3. Can be disabled
		FP16 E5M10	LNS I5F10	I5_out = E5_in + EB_adj EB_adj = FP16_in_EB - LnsI5F10_out_EB	Lin2Log mapping from M10 to F10
	AMEM Weights -> Grid	LNS I4F3	LNS I5F3	I5_out = I4_in + EB_adj EB_adj = LnsI4F3_in_EB - LnsI5F3_out_EB	identity
		LNS I5F10	LNS I5F10	identity	identity
(6)	VPU	VPU has extensive programmable capabilities for type conversion and EB adjustment (scaling). See the full VPU specification at this link			

Table 6-4: Summary of Data Conversions

6.6 UIE Microcoded Sequencing Engines

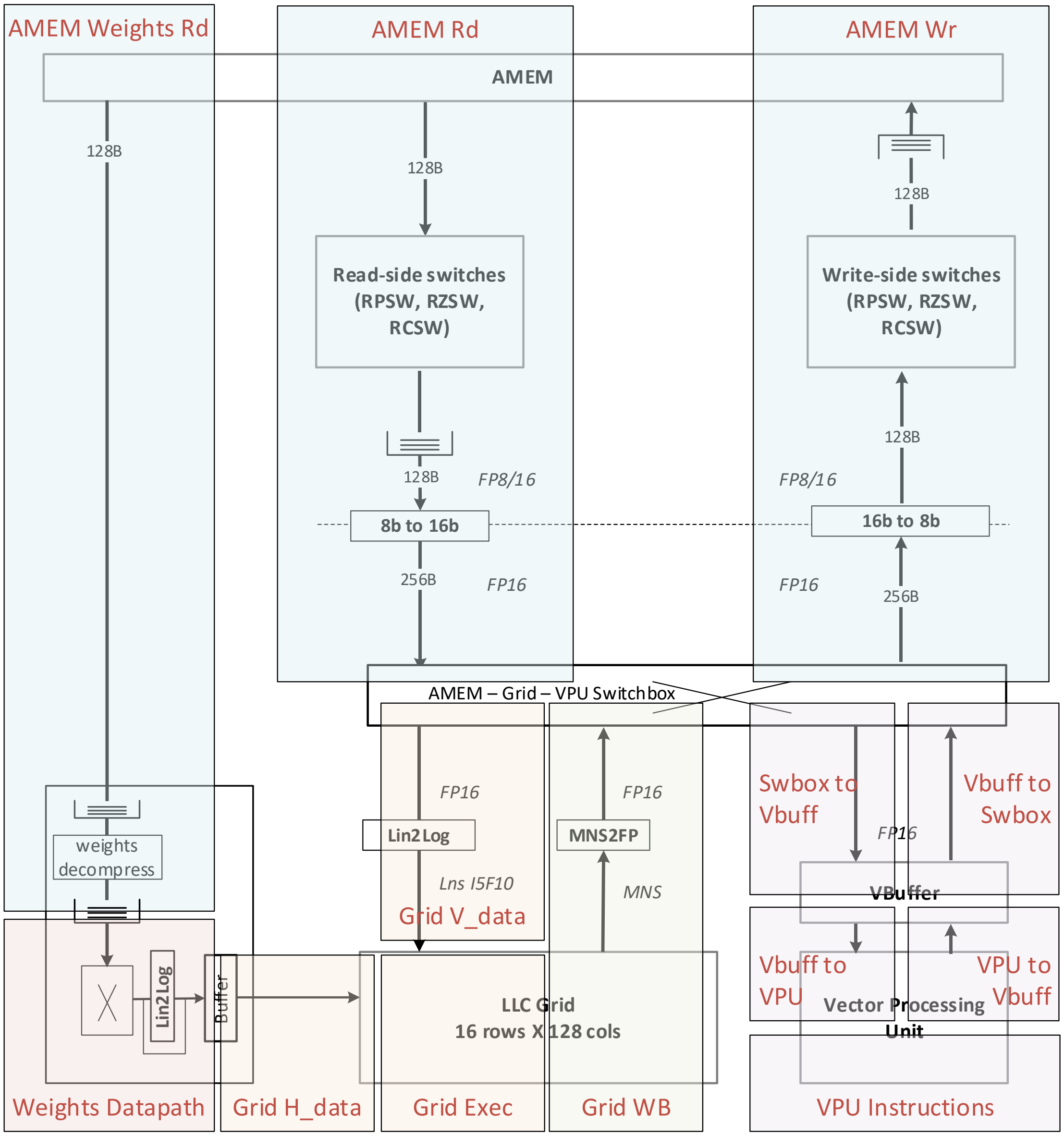


Figure 6-13: UIE Sequencer Domains

The UIEs use microcoded ***Sequencers*** to perform their tasks. Each sequencer controls a portion of the data flow through the UIE. The sequencers and their domains are summarized as follows:

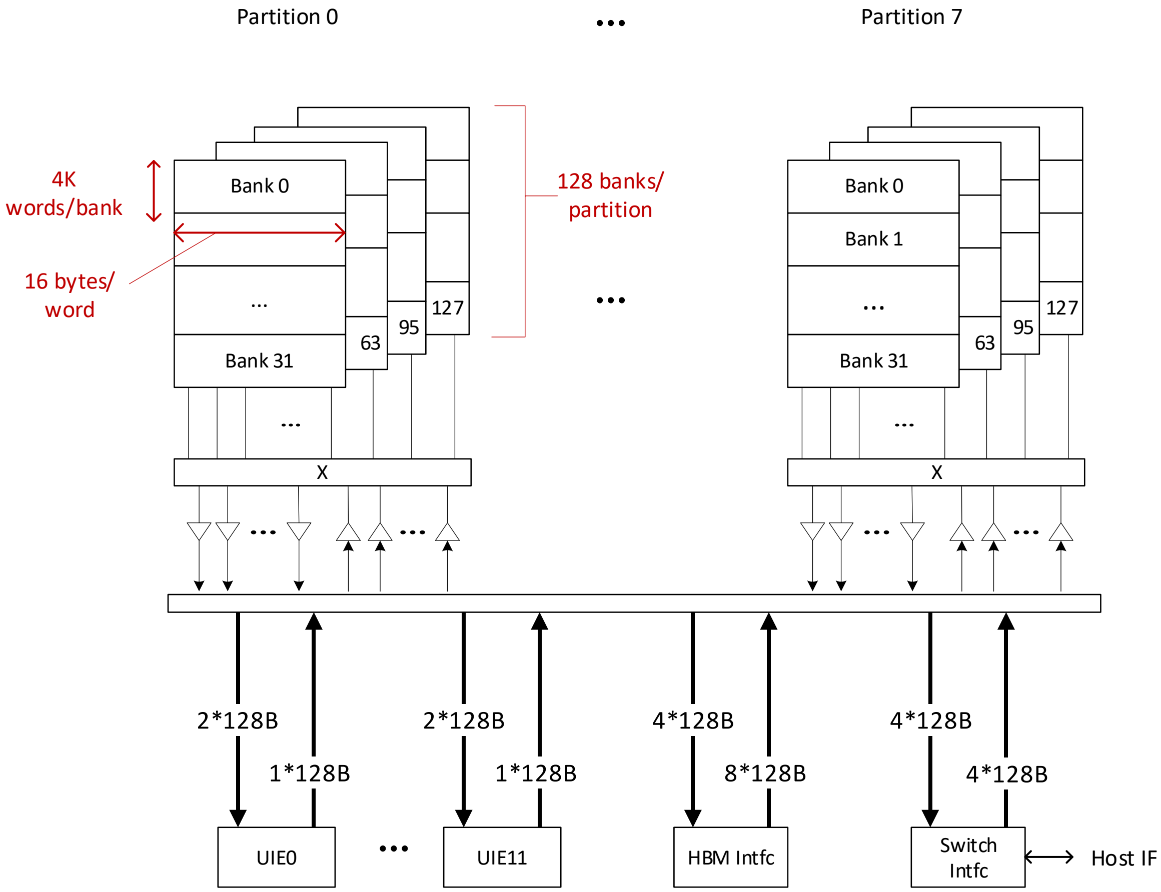
- **AMEM Read**: this sequencer controls reading the primary data from AMEM as well as the flow through the read-side switches, and terminating as the data is written into a Switchbox FIFO
- **AMEM Weights Read**: controls reading the Weights (or 2nd set of data for MatMul) from AMEM, as well as the weights decompressor, terminating at a FIFO which decouples this from the Weights Datapath

- **Weights Datapath**: controls movement of (decompressed) weights or 2nd data for MatMul through the weights-switch, lin2log converter and masking, into the weights buffer memories which will feed the “horizontal” data flow to the Grid.
- **Grid Control**: consists of three component sequencers:
    - **Grid-Horizontal-Data**: coordinates movement of horizontal data from the weights buffer memories to the weights-staging buffers in the LLCs.
    - **Grid-Vertical-Data**: coordinates movement of vertical data from the Switchbox to data-staging buffers in the LLCs.
    - **Grid-Execution**: controls the execution of the LLCs to compute dot products and write data into the accumulators.
- **Grid Writeback**: controls offloading data from the Grid (accumulators), terminating at the Switchbox
- **SwitchBox-Vbuffer Wr**: controls movement of data from the Switchbox to VBuffer
- **SwitchBox-Vbuffer Rd**: controls movement of data from the VBuffer to Switchbox
- **VPU-Vbuffer Wr**: controls movement of data from the VPU to VBuffer
- **VPU-Vbuffer Rd**: controls movement of data from the VBuffer to VPU
- **VPU Instruction:** issues the series of VPU instructions to execute VPU algorithms.
- **AMEM Write**: controls moving data from the Switchbox to AMEM, including the flow through the write-side switches, and provides the write address and controls to AMEM.

Within each section, the Sequencers provide the cycle-by-cycle control points for each functional unit.

Sequencer details are out of scope for this document, and are available in the Pyxis UIE Sequencers Functional Spec document.

7 Accelerator Memory (AMEM)



The Accelerator Memory (AMEM) is a multi-banked, multi-client SRAM structure, which provides high-bandwidth access to the Universal Inference Engines (UIEs) for per-trip activation data, weights, and control microcode. There is one instance of AMEM per UIE Cluster

The total memory array size is **512K words** * **128 bytes/word**, for total **64Mbytes** per AMEM instance.

Organized as a multi-banked structure, with **8 partitions** (width) * **128 banks/partition** (depth)

Each **partition is 16 columns** wide * 1 byte/column, for a total of 128 bytes data-width.

Each **bank is 4K words** deep, for a total of 512K words.

Accesses are in units of 16-byte partitions. An AMEM client can access all 8 partitions (128 Bytes) in one cycle, or a contiguous subset of partitions for narrower access cases.

Each partition is separately addressable. This allows a client to perform partition-“unaligned” reads/writes, to access 8-partition’s worth of data from any partition-granular address. For example, a particular tensor row might be 60 logical partitions (=960 columns) wide, and a client may access any 8 contiguous logical partitions (=128 columns) within that tensor row.

7.1 AMEM Address Structure

AMEM provides a simple “linear” memory model, where memory can be allocated in chunks of any size, with any address alignment, with partition (16-byte) granularity.

AMEM has an access width of 128-byte Words, with each word comprising 8 partitions with 16 bytes per partition. Addresses are constructed as follows:

AMEM address construction:

26	25 … 14	13 … 7	6 … 4	3 … 0
view	word_in_bank [11:0]	bank [6:0]	partition [2:0]	byte in ptn[3:0]

----- AMEM Word Address ----- | ----- byte in word -----

----- AMEM Partition Address ----- | -- byte in ptn --

The “view” bit indicates whether hashed or un-hashed access is used for the bank-index bits. This bit does not define additional memory locations, it is only a virtual address bit. Bank hashing is described in section 7.5 below.

View	Access method
0	Bank hashing enabled
1	Bank hashing disabled

7.1.1 AMEM Endianness

AMEM uses little-endian addressing as seen by processors accessing AMEM:

- Within each 16-byte partition, byte 0 is the LSByte, byte 15 is MSByte. Tensors and matrices are often illustrated the other way, with column 0 on the “left” which is big-endian, but this is just a notational convention. Pyxis stores tensors in memory with column 0 in the LSByte position

- FP16 values are stored in two consecutive bytes: byte0 is LSByte, byte1 is MSByte. This is explicitly little-endian.

7.2 Tensor mapping to AMEM

Pyxis tensors are multi-dimensional arrays, with two most common arrangements

For spatial feature-maps of Convolutional Neural Networks, tensors are described with `(channel, row, column)` dimensions – note this is "channels first" native format. Each spatial slice (rows x columns for one channel) is flattened into a 1D array of size `(rows * roundup_to_16(columns))`, i.e., the bytes of each row are typically stored in contiguous memory locations, and subsequent rows are stored contiguously, but with **partition** (16B) granularity per row. Multiple channels of the tensor may then be tiled in contiguous ***or*** non-contiguous regions to build up the 3D representation for shape (channels, rows, columns).

For LLMs and other tokenized models, tensors may be described with `(batch, sequence_length, features)` dimensions. For hardware efficiency, we normally prefer a "channels first" format, and we can flatten each feature dimension (=channel) into a 1D array of tokens of size (`batches * sequence_len`). This results in a 2D array of shape **(`channels, tokens`)**, i.e.., a matrix. The preferred representation of tokens is therefore column-vectors (per token). In this model, each matrix row represents one channel across multiple tokens. Each matrix row is stored in contiguous bytes in memory, with partition (16B) granularity. Multiple rows may then be placed contiguously or non-contiguously to build up the features (channels) dimension. Hardware also supports transposed representations of any of the above

The detailed mapping depends on the tensor width of the last dimension (in storage columns), and there are three general cases to consider:

- **Wide tensor (W > 128 bytes)**: For this case, each tensor row will occupy 9 or more contiguous partitions and multiple AMEM words. Each AMEM access can return any 8 consecutive partitions within one tensor row. Successive tensor rows can be mapped to contiguous memory locations on partition boundaries – the AMEM-word boundaries are not actually relevant, and there are usually no alignment restrictions among rows and channels.
- **Intermediate width tensor (64 < W <= 128 bytes)**: For this case, each tensor row will occupy 5 or more contiguous partitions. Successive tensor rows can be mapped on sequential partition boundaries, with any starting partition alignment. Each memory access can return at most one row.
- **Narrow tensor (W <= 64 bytes)**: For this case, each tensor row occupies at most 4 consecutive partitions, and sequential rows are mapped into contiguous partitions, with any starting partition alignment. Each row only needs to round up to a multiple of 16 bytes. *Two or more rows may be read or written with a single AMEM access*.

Note: Software always has the option of spacing out the rows or channels with an arbitrarily expanded stride per row and per channel, with strides measured in units of partitions. In some cases, this may provide for more efficient memory access, such as avoiding "hot bank" access patterns, at the cost of some memory wastage.

The mappings are explained in more detail in the following sections.

7.2.1 FP8: Tensor width > 128 columns

Below is an example mapping for a tensor with width 272 columns (17 partitions) and 3 rows per channel, to illustrate tensor packing available in AMEM. Note that rows and channels may be abutted on arbitrary partition boundaries. In this example, the row-stride is 17 partitions, and channel stride is 3*17 = 51 partitions.

For tokenized models, the same mapping rules apply, but these tensors would have only 1 row per channel, with possibly very wide rows (corresponding to the flattened tokens dimension = `batch*sequence_len`)

partition **word**	0	1	2	3	4 .. 6	7
0	**Chan 0**: row0, col 0-15	row 0, col 16-31	row 0, col 32-47	…		row 0, col 112-127
1	col 128-143	…	…	…		col 240-255
2	col 256-271	row 1				
3						
4			row2			
5						
6				**Chan 1**, row 0		
7						
8					…	
…						

Table 7-1: Example AMEM Mapping for W>128, FP8 data type

7.2.2 FP8: Tensor width > 64 and <= 128 columns

Below is an example mapping for a tensor with 80 columns (5 partitions) and 3 rows per channel, again illustrating packing on arbitrary partition boundaries for rows and channels. Note that each successive row is mapped with a stride of 5 partitions, regardless of the AMEM-word-level boundaries.

For tokenized models, the same mapping rules apply, but these tensors would have only 1 row per channel

<table>
<tr><th>partition
word</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td colspan="5">**Chan 0**, row 0</td><td colspan="3">row 1</td></tr>
<tr><td>1</td><td colspan="2"></td><td colspan="5">row 2</td><td>**Chan 1**</td></tr>
<tr><td>2</td><td colspan="4">… **Chan 1**, row 0</td><td colspan="4">row 1</td></tr>
<tr><td>3</td><td></td><td colspan="6">row 2</td><td>…</td></tr>
</table>

Table 7-2: Example AMEM Mapping for 64 < W <= 128, FP8 data type

7.2.3 FP8: Tensor width <= 64 columns

When tensor width <= 64 bytes, multiple rows of the same channel will fit in each AMEM word, which allows some algorithms to read multiple tensor rows concurrently. This example shows a tensor with 48 columns (3 partitions) and 4 rows per channel. Note that AMEM-word-level alignment is not required. The row-stride in this example is 3 partitions, and channel-stride is Num_rows * 3, allowing channels to be abutted in memory on partition boundaries.

For tokenized models, the same mapping rules apply, but these tensors would have only 1 row per channel

<table>
<tr><th>partition
word</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td colspan="3">**Chan 0,** row 0</td><td colspan="3">row 1</td><td colspan="2">row 2</td></tr>
<tr><td>1</td><td>row 2</td><td colspan="3">row 3</td><td colspan="3">**Chan 1**, row 0</td><td>row 1</td></tr>
<tr><td>2</td><td colspan="2"></td><td colspan="3">row 2</td><td colspan="3">row 3</td></tr>
<tr><td>3</td><td colspan="3">**Chan 2,** row 0</td><td colspan="3">row 1</td><td colspan="2">…</td></tr>
<tr><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Table 7-3: Example AMEM Mapping for W <= 64, FP8 data type

7.3 FP16 STORAGE FORMAT

FP16 requires two AMEM bytes for each data element, hence twice the storage footprint vs FP8. Each FP16 value is stored in a 16-bit container occupying two sequential bytes in AMEM, on even-byte boundaries. The bit organization within each 16-bit container is shown below.

<---------- Byte 1 --------->			<---------- Byte 0 --------->
15	14:10	9:8	7:0
S	E	Mantissa.F_2	Mantissa.F_8

FP16 tensors are stored with a **granularity of 16 logical columns**, or 32 bytes (i.e., two partitions) in AMEM.

Note: **FP16 Tensors must be aligned on an even-partition boundary**. **This alignment restriction applies for all tensors with 16-bit data types, including FP16, LnsI5F10 and Int16**

Within a particular partition-pair, we store the two-byte data values for columns 0-7 in the first partition, and the two-byte values for columns 8-15 in the 2nd partition, as follows:

Partition index[0]	**Bytes in each Partition**							
	15:14	13:12	11:10	9:8	7:6	5:4	3:2	1:0
0	col 7	col 6	col 5	col 4	col 3	col 2	col 1	col 0
1	col 15	col 14	col 13	col 12	col 11	col 10	col 9	col 8

When communicating with the UIEs, AMEM will normally receive and transmit each FP16 value in a single column over two transfer cycles. I.e., each FP16 data value is sent to/from UIEs over a single byte-wide column lane, **serialized over two cycles**. This serialization is performed automatically by hardware. Similarly, FP16 data sent from the UIEs to AMEM is serialized over two cycles per column, and AMEM hardware will deserialize and write two sequential byte locations in AMEM as described above.

7.3.1 FP16 Data Swizzle at AMEM

In order to map the two-byte data elements to one-byte lanes (serialized over two cycles), transfers to/from AMEM exhibit a particular partition-level “swizzle”. The partitions are interleaved in the following format, and this must be de-interleaved after reading the (serialized) data and re-interleaved before writing. The de-interleaving and re-interleaving are performed by the **Read Partition Switch** and **Write Partition Switch**, respectively, and these hardware blocks must be programmed to perform the expected swizzle function (See section 6.2.1.1 above).

AMEM Partition Swizzle for Serialized FP16 data on READ								
Physical Partition (exiting AMEM)	0	1	2	3	4	5	6	7
Cols in 1st cycle	0-7	64-71	16-23	80-87	32-39	96-103	48-55	112-119
Cols in 2nd cycle	8-15	72-79	24-31	88-95	40-47	104-111	56-63	120-127
Logical Partition (in Datapath)	0	4	1	5	2	6	3	7

AMEM Partition Swizzle for Serialized FP16 data on WRITE								
Logical Partition (in Datapath)	0	1	2	3	4	5	6	7
Cols in 1st cycle	0-7	16-23	32-39	48-55	64-71	80-87	96-103	112-119
Cols in 2nd cycle	8-15	24-31	40-47	56-63	72-79	88-95	104-111	120-127
Physical Partition (entering AMEM)	0	2	4	6	1	3	5	7

7.3.2 FP16: Tensor width > 64 columns

Below is an example mapping for an FP16 tensor with 80 logical columns, totaling 160 bytes (10 partitions) per row; and 3 rows per channel, to illustrate tensor packing for a wide tensor with FP16 data.

partition word	0	1	2	3	4	5	6	7
0	**Chan 0**: row 0, col 0-7	row 0, col 8-15	row 0, col 16-23	…				row 0, col 56-63
1	col 64-71	col 72-79	Row 1					
2					row2			
3							**Chan 1**, row 0	
4								
…								

*Table 7-4: Example AMEM Mapping for W>64, **FP16** data type*

7.3.3 FP16: Tensor width > 32 and <= 64 columns

Below is an example mapping for an FP16 tensor with 48 logical columns, 96 bytes (6 partitions) per row; and 3 rows per channel. The tensors must be stored with granularity and alignment of 2 partitions.

<table>
<tr><th>partition
word</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td colspan="6"><b>Chan 0</b>, row 0</td><td colspan="2">row 1</td></tr>
<tr><td>1</td><td colspan="4"></td><td colspan="4">row 2</td></tr>
<tr><td>2</td><td colspan="2"></td><td colspan="6">… <b>Chan 1</b>, row 0</td></tr>
<tr><td>3</td><td colspan="6">row 1</td><td colspan="2">…</td></tr>
<tr><td>…</td><td colspan="8"></td></tr>
</table>

*Table 7-5: Example AMEM Mapping for 32 < W <= 64, **FP16** data type*

7.3.4 FP16: Tensor width <= 32 columns

When FP16 tensor width <= 32 columns, multiple rows of the same channel will fit in each AMEM word. Below is an example for tensor width = 32 columns for 64 bytes (4 partitions) per row and 3 rows per channel.

<table>
<tr><th>partition
word</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td colspan="4"><b>Chan 0,</b> row 0</td><td colspan="4">Chan 0, Row 1</td></tr>
<tr><td>1</td><td colspan="4">Chan 0, Row 2</td><td colspan="4"><b>Chan 1</b>, row 0</td></tr>
<tr><td>2</td><td colspan="4">Chan 1, Row 1</td><td colspan="4">Chan 1, Row 2</td></tr>
<tr><td>3</td><td colspan="4"><b>Chan 2</b>, row 0</td><td colspan="4">Chan 2, Row 1</td></tr>
<tr><td>…</td><td colspan="8">…</td></tr>
</table>

*Table 7-6: Example AMEM Mapping for W <= 32, **FP16** data type*

7.4 Read-Modify-Write and In-Memory Add

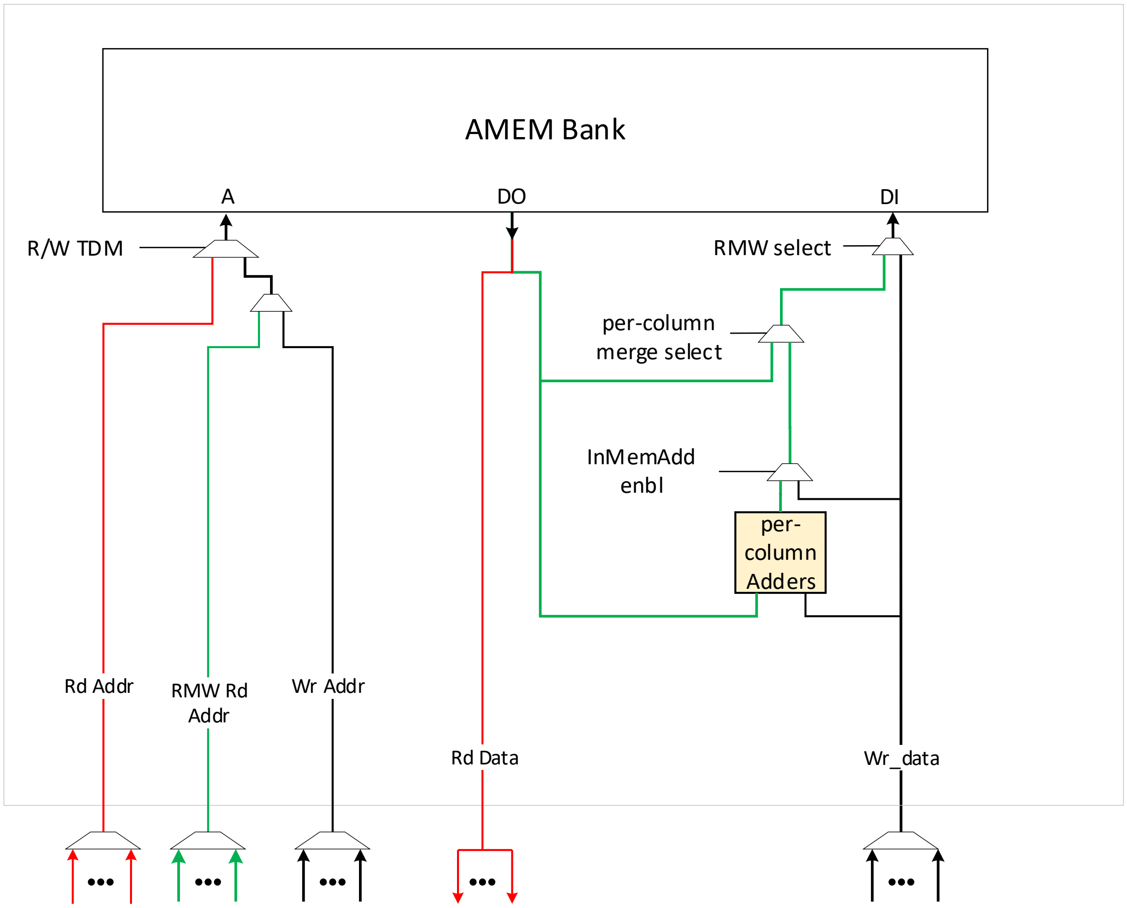


Figure 7-1: One bank of AMEM showing Read-Modify-Write and InMemAdd logic

AMEM supports Read-Modify-Write (**RMW**) primitives, where the data is read from memory, combined with data from the write-client, and written back to the **same or a different address**. These RMW primitives operate within each bank of AMEM.

This is optimized to support ***InMemAdd*** for residual-block architectures, where the output of a layer is added element-wise to a skip-connection tensor from a prior layer. This reads the prior-layer data, adds the incoming ("write") data, and stores the result in the same or a different location in AMEM. This may also be used to accumulate values in a single AMEM location, setting the read address = write address. The adders support FP8 and FP16 data formats.

The RMW primitives also support byte-granular ***merging*** between the read and write data. This supports ***partial writes*** of fewer than 16 contiguous bytes in any particular partition, and also supports partial-width InMemAdds. Hardware generates a merge-selection mask based on the starting and ending columns for each write operation. Hardware will read the entire 16-byte value (for the given partition), and replace bytes with write-data, based on the merge selection mask. The resulting 16-bytes are then written to AMEM. As shown in Figure 7-1, this merging selection is effectively taken after the InMemAdd (if that is enabled).

7.4.1 RMW Address Alignment

The RMW hardware requires that the source and destination tensors must have the same shape in AMEM, and the **same bank alignment**. This ensures that `ReadTensor[row,col,chan]` and `WriteTensor[row,col,chan]` map to the same column in the same partition in the same bank for every element.

To ensure this, software can require that source and destination tensors for the RMW have the same *starting bank* alignment, and same row and channel strides, and same data type (8-bit or 16-bit). This is achieved when the starting address of source and destination tensors are aligned to the same AMEM byte address *modulo 16Kbytes.* Please see section 7.1 above for AMEM address construction, showing the bank index address bits. Also, AMEM Bank Hashing (section 7.5) should be **disabled** for RMW with **different** source and destination addresses. If hashing is enabled, it is nearly impossible for the source and destination tensors to have identical bank-mapping.

In many cases, the source and destination address are the same. This is true for “destructive” InMemAdd, and also true for partial-writes with column merging. In this case, the alignment restrictions are moot.

7.4.2 RMW Atomicity

AMEM guarantees ordering and correct operation for multiple RMWs to the same address **from the same UIE**. For example, a particular UIE may be programmed to accumulate multiple values into AMEM using a series of InMemAdd operations in a single trip.

However, RMW operations are NOT guaranteed to be atomic across **different** UIE., and it is possible for two RMWs to be intertwined, or for a write to sneak in between the read and write portions of the RMW, when two RMW access are issued to the same address from different UIEs. In most cases, this would be an obvious programming mistake since write-ordering is not guaranteed without the use of a barrier.

For partial-writes, there is an additional hazard which may not be obvious to the programmer, in case of two partial-writes to non-overlapping bytes but overlapping partition addresses. These may appear to be writing to disjoint data, but due to the partition-granular access, they will interfere if the RMWs become intertwined. This must be protected with a ***barrier*** to serialize the writes, if they are sent from different UIEs. See section 11.7 for further information on hardware barriers.

Rule: partial-write operations which have address-overlap on a partition-granular basis, when issued from different UIEs, *must* be serialized (such as with a barrier) ensuring that the two partial-writes cannot be intertwined.

7.5 AMEM Bank Hashing

In some cases, the mapping of tensors in memory will result in perfect power-of-two strides when accessing successive rows or channels. This can create “hot bank” problems since the effective number of AMEM banks in use would be limited by the power-of-two strides and power-of-two bank addresses.

To avoid this issue, AMEM provides **bank-index hashing**, in which the bank selected is “hashed” by XORing the 7-bit bank index with 7 bits taken from the upper part of the AMEM address. As a result, sequential accesses with a power-of-two bank stride will sequentially visit *all* of the banks over a long sequence. This is a common and effective technique to break up hot banking

The specific address hashing performed is

```
addr_hashed         = addr_unhashed           // default
addr_hashed[13:7]   = addr_unhashed[13:7] ^ addr_unhashed[20:14]
```

where ‘^’ is bit-wise XOR, and these are byte granular addresses.

Address bits [13:7] select the bank, so this XORs the next 7 MSBits (address[20:14]) into the bank-select.

However, in Pyxis, this interferes with the requirements for InMemAdd when the source and destination have a different address. This is described in section 7.4 above – the source and destination must be in the same AMEM bank. This becomes difficult or impossible to guarantee when bank-hashing is used.

Therefore, AMEM provides two ***views*** of memory, to select either bank-hashing or no bank-hashing. The access mode (or “view”) is selected using an additional upper address bit. All accesses to AMEM, whether from CPUs, DMA or UIEs, may specify either view with each memory access

The hashed/non-hashed view access granularity is 16K bytes, on 16K-byte boundaries. All data accessed within a 16K block must use either hashed or non-hashed access, these cannot be mixed within a block. This means that if any part of a tensor reaches into a 16Kbyte-aligned section of memory, then all tensors sharing that 16KByte region (or any part of that region) must use the same hashed/non-hashed view.

7.6 AMEM Indirect Read / Write (Row-based Scatter / Gather)

AMEM supports indirection on reads or writes via indirection tables stored in memory. This allows reading AMEM words (or “rows”) in random-access order by performing an indirection through these tables. This supports scatter (on write) or gather (on read) operations, when scattering/gathering with granularity of AMEM words. The mechanism may be used for 1-D scatter/gather (such as reordering rows from a matrix) or 2-D (such as for selecting individual spatial locations from a 3-D tensor)

1. The first key requirement is that data will be scattered or gathered from the same relative locations for each channel or feature. Tensors to be gathered-from or scattered-to must be organized in AMEM ***channels-last***, i.e., for 2-D scatter/gather, this is [H, W, Chans], for 1-D, this could be [tokens, features]. The channel (or feature) dimension will occupy the physical columns dimension in memory. This may require that the to-be-scattered/gathered data is first

transposed to channels (or features)-last, and possibly transposed-back after the scatter/gather op.

2. The VPU (or CPU) creates **indirection tables** of index values, written into AMEM.
    - For 2-D gather, the indirection table must be stored in AMEM as a tensor of shape `[2, gather_table_length]`, of type `int16`. The `2` dimension provides two dimensional index values ("Y" and "X") into the data tensor, for each row to be gathered.
    - For 1-D gather, the indirection table may be stored in AMEM in one of three supported formats:
        - a row-vector of shape `[1, gather_table_length]` of type `int16`
        - a row-vector of shape `[1, gather_table_length]` of type `int32`, useful for tensors with > 2^16 rows
        - Alternatively, the (1-D) indirection table may be stored in AMEM as a tensor of shape `[2, gather_table_length]`, of type `int16`, where the `2` dimension provides two 16-bit values to compose a 32-bit index. Channel 0 is the 16 LSBits and channel 1 is 16 MSBits of the 32-bit index. Hardware concatenates the channel 0 & 1 values to form `int32` index values. This is useful for tensors with more than 2^16 rows, and when the VPU is used to prepare the index values, since VPU must write an `int32` as upper & lower `int16` parts into separate channels.
    - The same applies for **scatter**, with an associated scatter indirection table.
3. Programmable engines fetch the indices from AMEM into **indirection FIFOs** which supply the index values to the AMEM read sequencer (for gather) or AMEM write sequencer (for scatter) on demand.

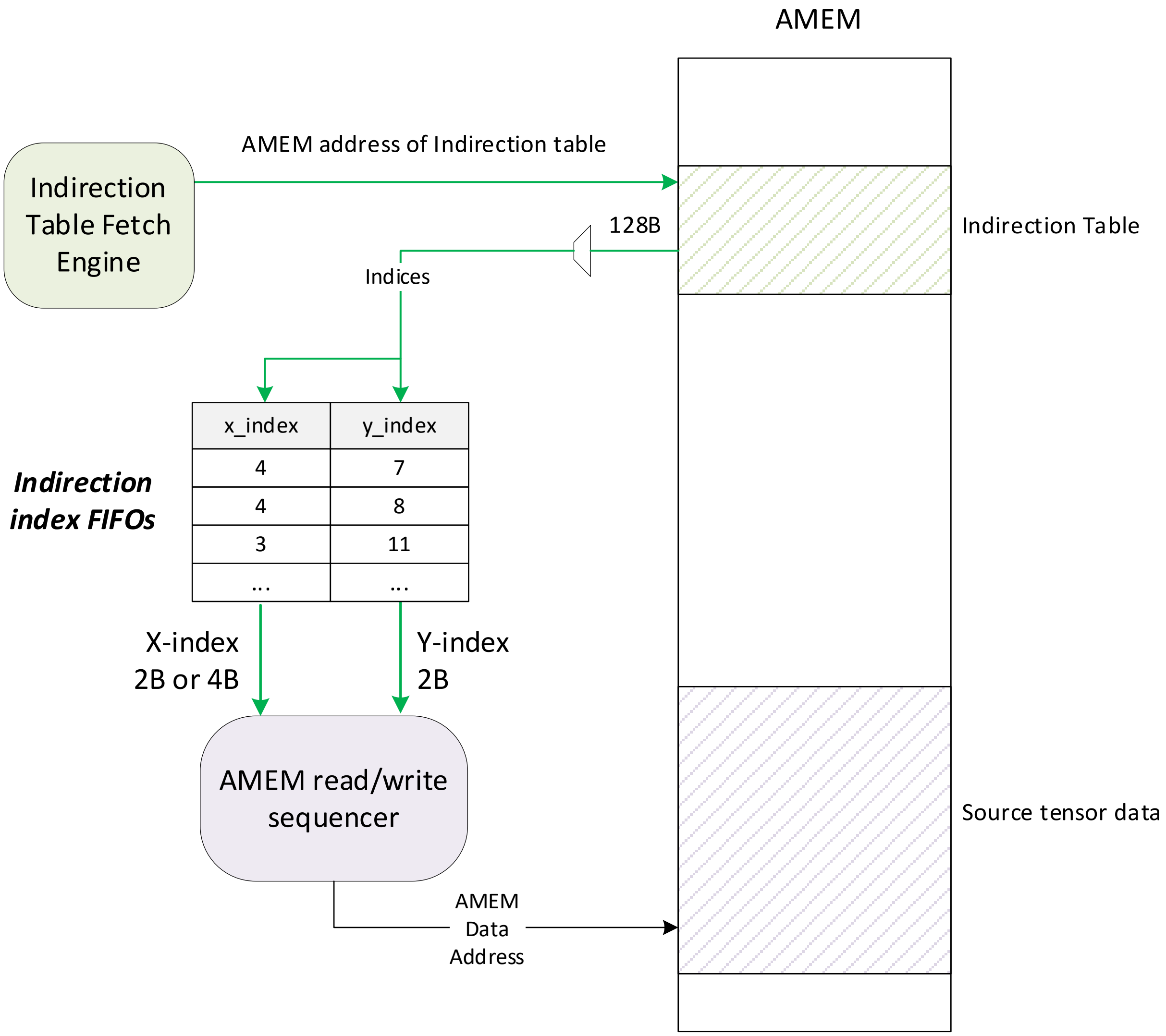
For Gather with indirection on Read from AMEM:

4. For each value to be read, AMEM read sequencer fetches an entry from its indirection FIFO(s), which retrieves the X-index and possibly Y-index of the source data tensor which was organized as [H, W, Chans] or [tokens, features] in step 1 above.
5. The AMEM read sequencer is also programmed with a base address and the per-dimension strides to compute the final AMEM address for the source data. The computation is `amem_addr = base_addr + (x_index * x_stride) + (y_index * y_stride)`. This addresses a row of data (128 columns) from AMEM.
6. The gathered data may be sent to the Grid, to the VPU, or directly written back to AMEM.

For Scatter with indirection on write: the same process applies for the AMEM write sequencer. Data may be sourced from the Grid, from VPU, or directly from the AMEM, and scattered (via indirection) into AMEM.

The "grid sample" application requires a 2D bilinear interpolation, which will read 4 AMEM words corresponding to tensor indices at (x,y), (x+1, y), (x, y+1) and (x+1, y+1). To support this, we must issue four AMEM reads for each indirection-FIFO entry, adjusting each address by `1*x_stride` or `1*y_stride` as appropriate. This can be supported by the AMEM read sequencer programming. One additional aspect: the sequencer must not pop the indirection FIFO(s) until the fourth read. This special behavior is also supported by the AMEM read sequencer.

The hardware for AMEM Read with indirection is shown in Figure 7-2. AMEM Write with indirection is similar, supplying indirection values instead to the AMEM Write sequencer.


AMEM
Indirection Table Fetch Engine
AMEM address of Indirection table
128B
Indices
Indirection Table
Indirection index FIFOs
X-index 2B or 4B
Y-index 2B
AMEM read/write sequencer
AMEM Data Address
Source tensor data

x_index	y_index
4	7
4	8
3	11
...	...

Figure 7-2: AMEM Read with Indirection

7.7 Additional AMEM Datapath Operations

7.7.1 Read_Const

AMEM may be instructed to send a constant values for all pixels, instead of reading the memory. The constant value is supplied by the AMEM Read sequencer.

An application for Read_const with constant value = 0, is for pre and post-padding rows for 3x3 convolution of image data, which typically requires a pad-ring. The sequencer instructs AMEM to emit a row of zeros before and after the tensor data.

Another application for Read_Const is for AMEM to send “1” in every pixel location to multiply by a bias value to accumulate a bias value fused to a convolution. This is needed since the LLC grid must multiply the weights * data, and we set the weight=bias, data=1. In this case the value “1” is an FP8 value encoded with the Exponent Bias (EB) of the data for that layer, i.e., `const = 1.0 x (2^(-EB))`.

AMEM might also be instructed to send `+/-max` (maximum representable magnitude number) for every pixel of certain rows; this is useful for tensor boundaries for min/max-pooling operations.

The constant value is programmable in the AMEM Read sequencers. This supports either 8-bit or 16-bit constant values, consistent with the data type being read from AMEM.

7.7.2 ReLU on Read

AMEM supports a very simple ReLU on the read path. This may be used after layers which implement the element-wise add for residual blocks (common for vision networks); in this case the Activation function (ReLU) is deferred until the *next* read from AMEM.

8 HBM Interface

Pyxis supports four HBM 3E devices, with one HBM device per cluster. Each HBM3 device connects 1024 data pins at up to 9.6Gbps/pin, providing up to 12,288 GBytes/sec bandwidth per device.

The HBM interface logic includes the HBM PHY, HBM controller, and customized “shim” logic to connect the very-wide data buses between the HBM controller, AMEM and the System NoC.

The basic building blocks are shown in the figure below, illustrated for one Pyxis Cluster:

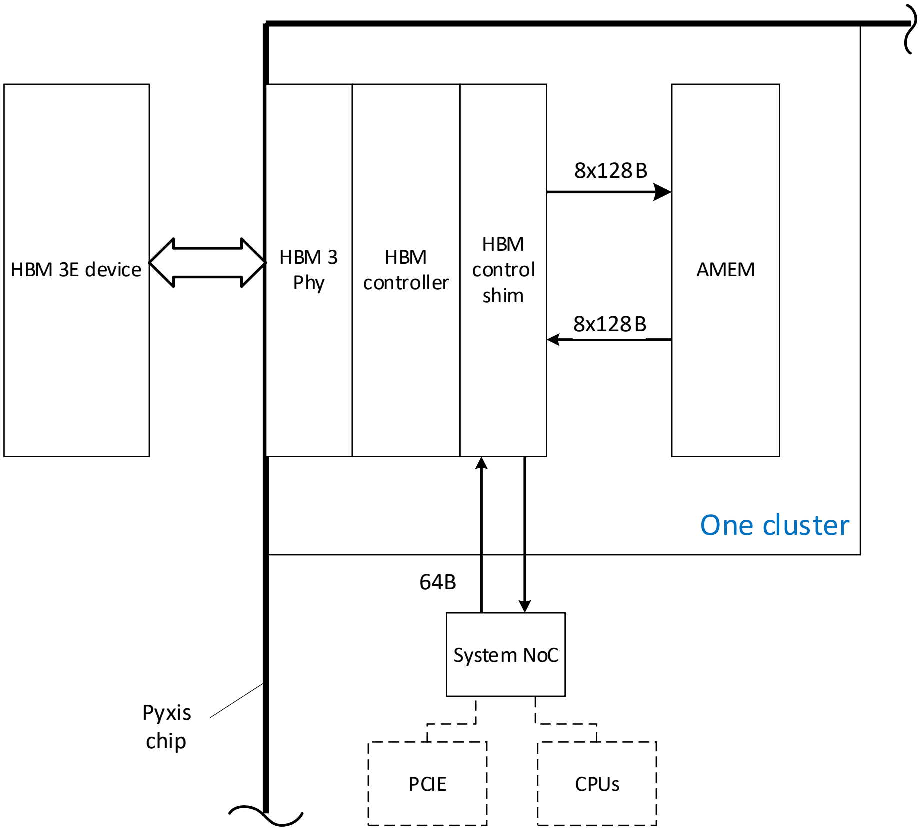


Figure 8-1: HBM Datapath for one Cluster

8.1 HBM Datapaths

The HBM connectivity is now described in simplified form, shown below. (For further details, consult the Pyxis HBM Interface specification.) Each HBM3 device provides 1024 data pins at up to 9.6Gbps/pin, which is "half duplex" (either read or write). This interface is divided into 16 independent channels, each with 1/16th of the memory bandwidth and capacity. The **HBM3 PHY** terminates the high-speed external data interface and connects with a parallel interface to the **HBM Controller**. This is implemented with two 8-channel RAMBUS HBM controller modules. The controllers expose 16 (i.e., per-channel) AXI-4 interfaces to the **HBM control shim** logic. Each AXI interface is 64-bytes wide at 1.33GHz.

The very high HBM3 bandwidth requires a large number of internal wires and routing resources on the Pyxis chip to transport the data. The 16 x 64Byte AXI interfaces totals 8192 data signal wires, plus address and control information, for *each* of the HBM read and write directions. This data path is optimized for direct DMA transfer between HBM and AMEM at the full HBM rate.

To minimize wiring and switching logic between HBM and AMEM, *each pair of HBM channels is associated with exactly one AMEM Partition*. As shown in Figure 8-2, data from two HBM channels (128B) is ganged together and connected to one AMEM partition. This totals 1024 bytes/cycle to and from AMEM.

This connection requires a very specific mapping from HBM to AMEM. Each 128 byte "flit" transferred (between two HBM channels and one AMEM partition) is mapped as 8x16 bytes, with the 16-byte chunks striped over 8 contiguous 128-byte AMEM words. This is necessary because each AMEM partition holds a specific 16-byte portion of every 128-byte word. The mapping is described in more detail in the following sub-sections.

HBM is also accessible by PCIE and the on-chip CPU complex via the Network-on-Chip (NoC). The NoC provides a lower-bandwidth interconnect throughout the chip for configuration and CPU access. This supports auxiliary access to HBM, including

- Direct access to HBM by the PCIE interface or PCIE DMA engines,
- CPUs may access HBM as their main memory
- The specialized DMA engines - Descriptor Prefetch Engine (section 12.5) and Microcode Patch Engine (section 12.6) - may access HBM, e.g., for fetch descriptors or microcode patches.

The HBM control shim provides 64-byte-wide AXI port interfaces to NoC, with one NoC connection for each of the 16 HBM channels. The NoC bandwidth is limited to 64B/cycle (approx 85Gbytes/s) per direction

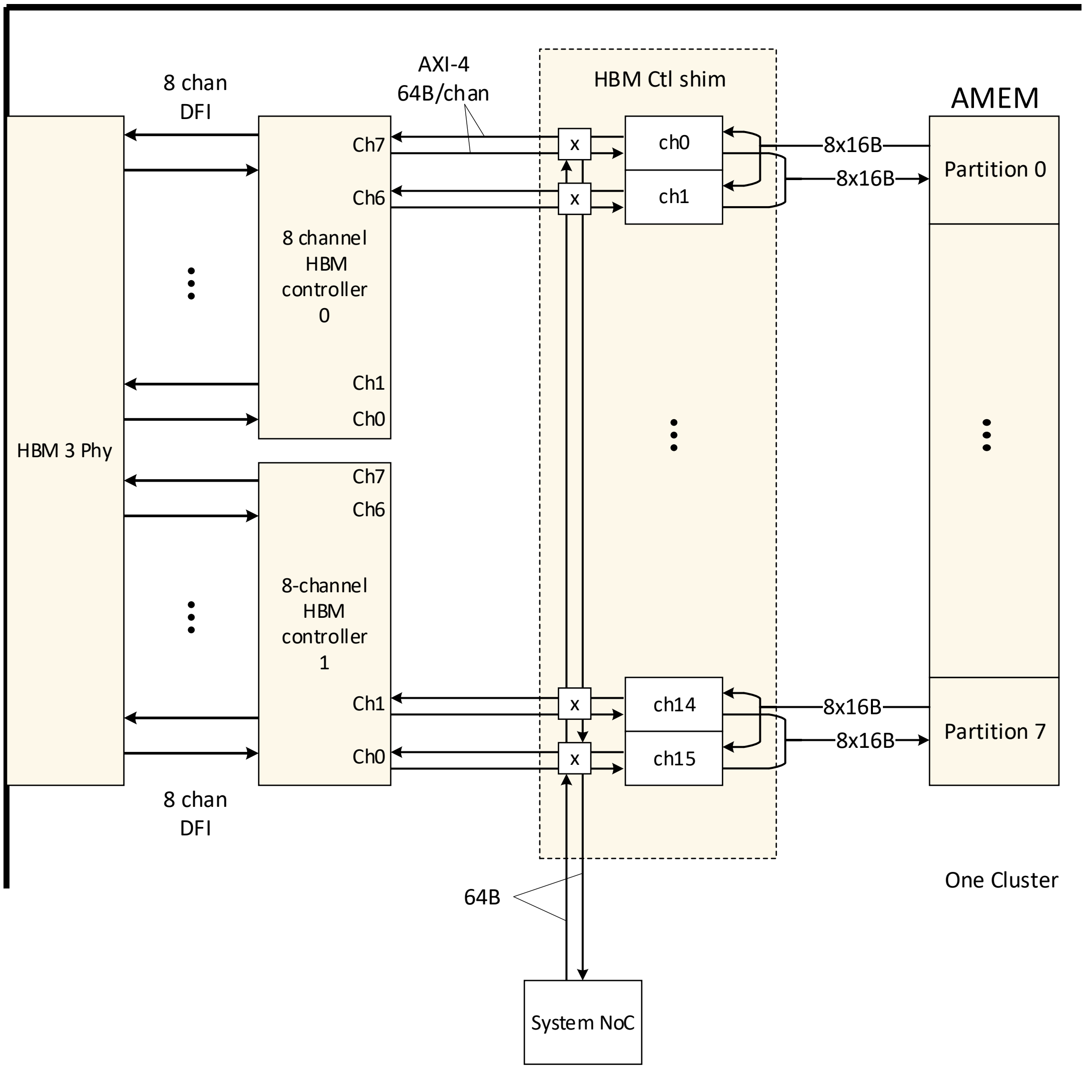


Figure 8-2: HBM Control Datapaths

8.1.1 AMEM to HBM DMA transfers

To satisfy the HBM controller interface bandwidth of 64 bytes/cycle/channel, we must read 8 x 128-byte sequential AMEM words per cycle. From this data, we collect 8 bytes from each of the 8 words for each of 16 channels. I.e., bytes 0-7 from each of the 8 AMEM words are concatenated and sent to HBM channel 15; bytes 8-15 from each of the 8 AMEM words are concatenated and sent to channel 14, etc. The mapping is shown in Figure 8-3.

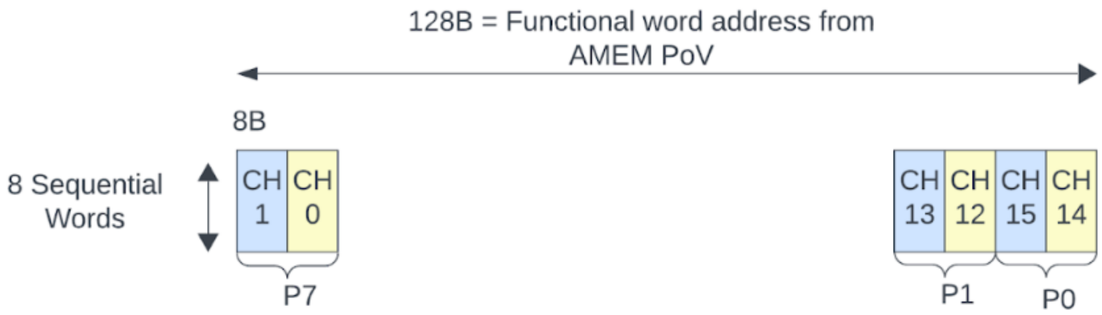
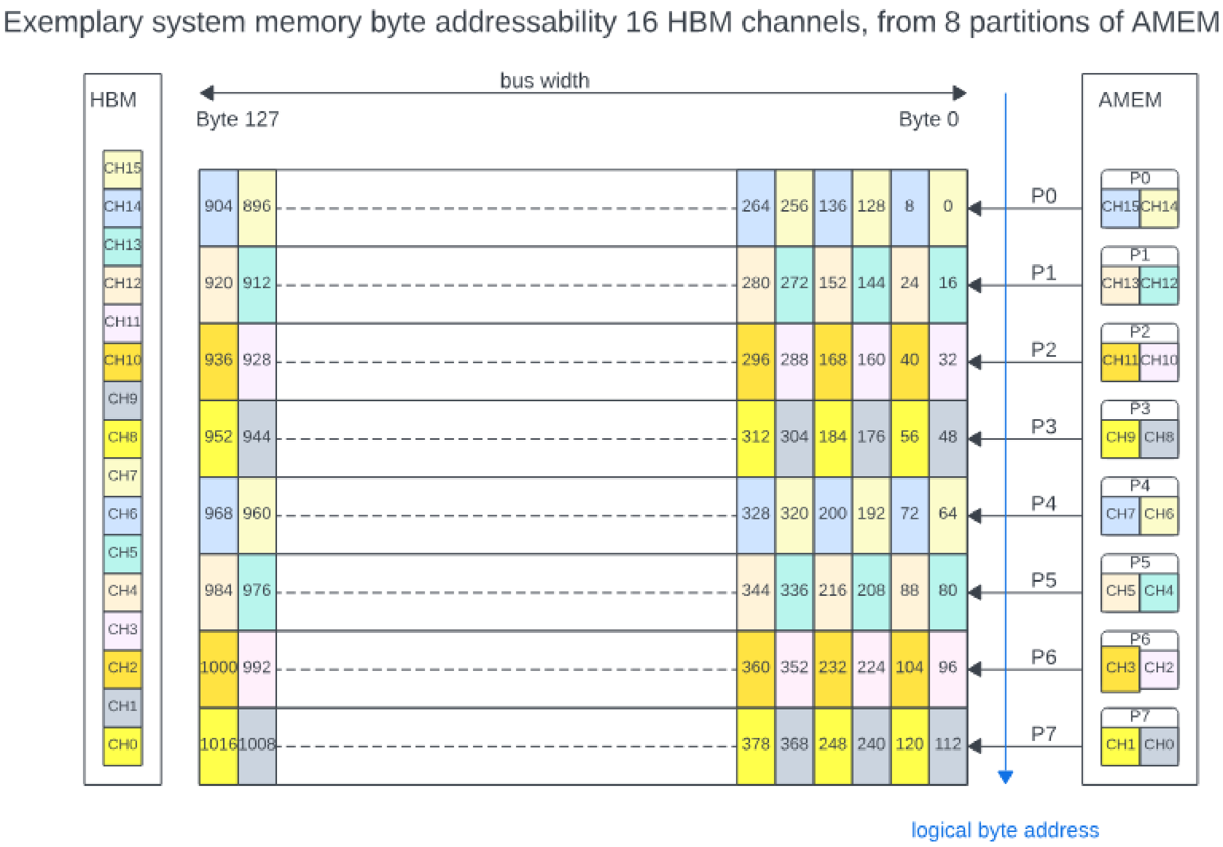


Figure 8-3:AMEM to HBM data transfer mapping



904	896		264	256	136	128	8	0	P0
920	912		280	272	152	144	24	16	P1
936	928		296	288	168	160	40	32	P2
952	944		312	304	184	176	56	48	P3
968	960		328	320	200	192	72	64	P4
984	976		344	336	216	208	88	80	P5
1000	992		360	352	232	224	104	96	P6
1016	1008		378	368	248	240	120	112	P7

Figure 8-4: AMEM to HBM byte address mapping

Note that the partition-to-channel mapping from AMEM to HBM is effectively reversed: HBM channels 0-1 map to AMEM partition 7, and HBM channels 14-15 map to AMEM partition 0. Then even / odd channels are also flipped in the mapping: The even-numbered channels map to bytes [15:8] in each partition, and odd channels map to bytes [7:0].

8.1.2 HBM to AMEM DMA transfers

The HBM controller interface delivers 64 bytes/cycle/channel. Each 64-byte flit is divided into 8 x 8-byte chunks, where each chunk will belong to a different AMEM word. To assemble data for 8 AMEM words per cycle, the first 8-byte chunk from every channel is concatenated to form the first AMEM word, the 2nd 8-byte chunk from each channel forms the 2nd AMEM word, etc. In total, this provides 8x128B words per cycle to be written to AMEM.

Figure 8-5 shows this data swizzling and reassembly for one AMEM partition. The channels are paired-up in the HBM control shim logic, and each channel-pair delivers 128 bytes/cycle to one AMEM partition.

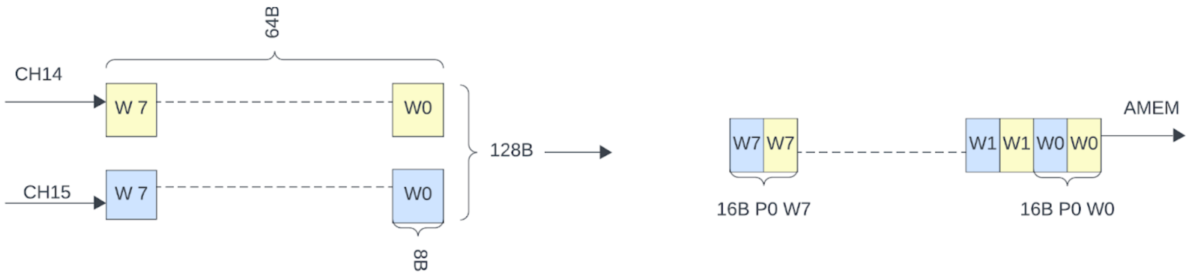


Figure 8-5: HBM to AMEM data transfer mapping

8.2 HBM MEMORY ADDRESSING

HBM supports three address-access modes.

- **Tensor mode** (***AMEM-HBM DMA transfers only***): Each 128-bytes is mapped as 8 bytes to each of the 16 HBM channels, with the same address offset in each channel. (This is equivalent to striping data across HBM channels with 8B stride, but as noted in section 8.1.1 above, the 8-byte-striping order is reversed between AMEM and HBM). This is optimized for DMA transfers between AMEM and HBM, since every 128-byte AMEM word is mapped to all 16 HBM channels, resulting in perfect load balancing across the channels. However, this is very inconvenient for CPU access, since each cache-line would map to *multiple* HBM channels. This access mode is used by AMEM-to-HBM (A2H) and HBM-to-AMEM (H2A) DMA engines.

The ***CPU/PCIE accesses*** may use one of two alternative modes:

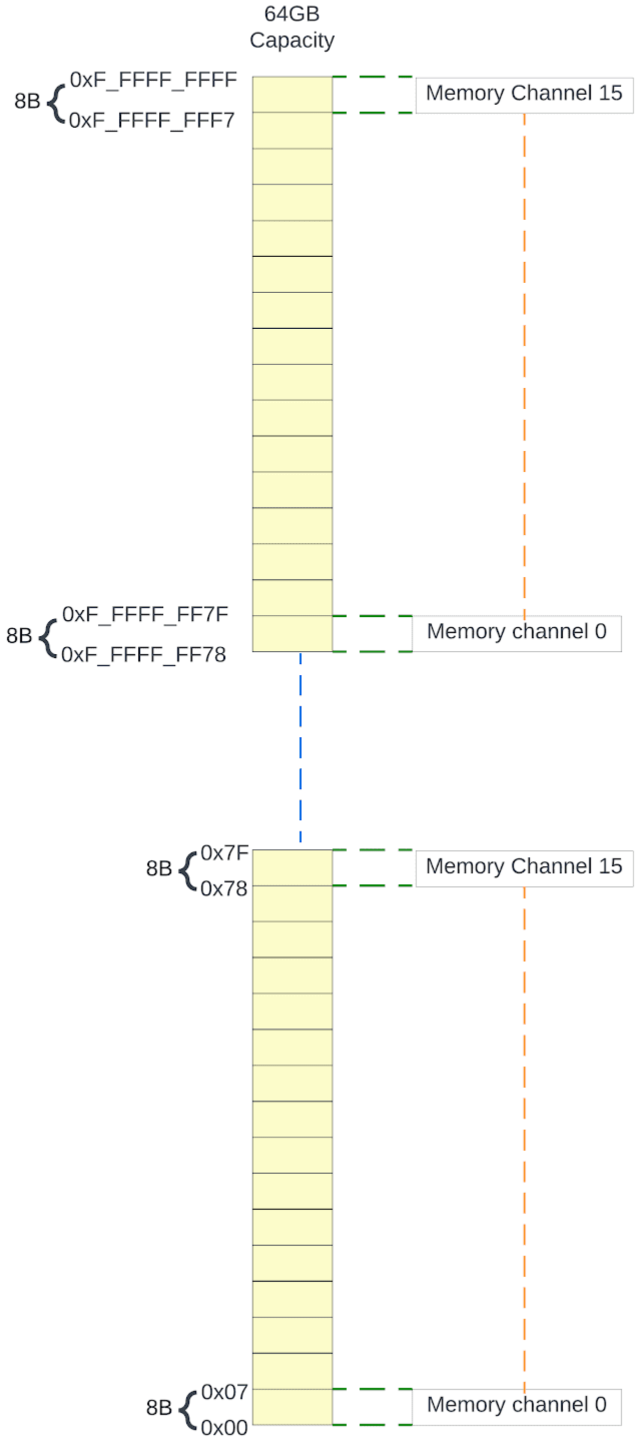
- **1024-byte Striped mode: (preferred)** the address range is striped across the HBM channels with 1024 byte stride. This optimizes CPU-oriented accesses (such as for cache-fills) with spatial locality, organizing multiple sequential cache-lines in the same channel (and page/bank of memory), while still balancing the load across HBM channels.

- **Linear mode**: the address range is contiguous within HBM channels. The first 1/16th of addresses are mapped to HBM channel 0, the 2nd 1/16th to channel 1, etc. This mode may provide high HBM page-hits for processes with very high spatial locality, but may also result in “hot channel” hazards in cases where the databases being accessed are local to a single HBM channel. A hot channel is more likely to interfere with (i.e., degrade) performance of AMEM<->HBM DMA transfers since these transfers must access all HBM channels for every AMEM-word transferred.

The striped vs. linear mode can be set by accessing a CSR in the HBM controller, and defaults to striped mode accesses. See the Pyxis HBM Interface specification for further details.

These mode mappings are illustrated in the following sub-sections.

8.2.1 Tensor Mode Addressing (used by HBM-AMEM DMA transfers)



8.2.2 Striped Mode Addressing

This is one of two optional addressing modes for PCIE/CPU accesses to HBM. This address mapping is summarized in the following figure, which shows thirty two 1K-byte stripes from the full address space. Each 1Kbyte block is mapped to the next successive HBM channel, wrapping after 16 channels.

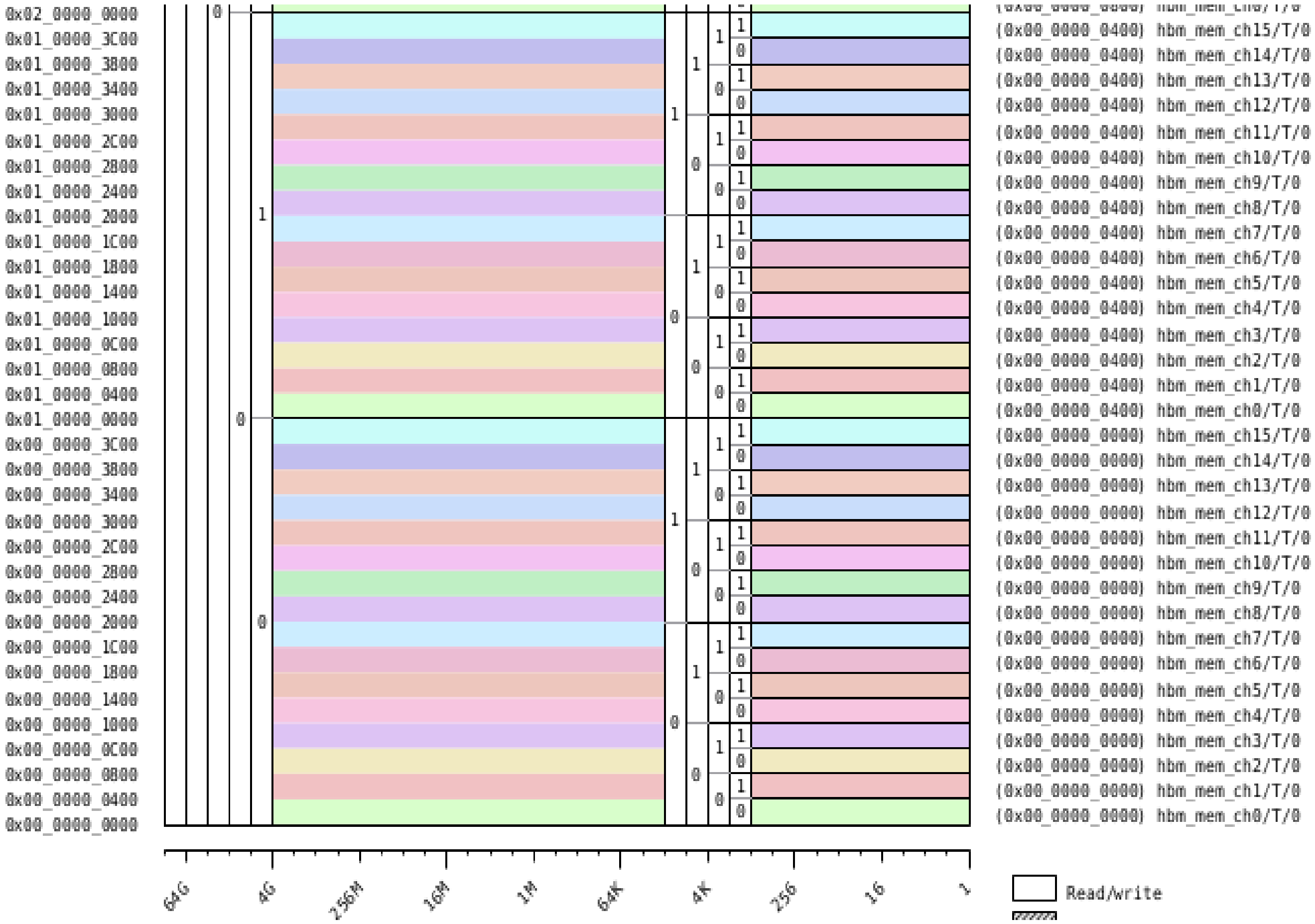

0x02_0000_0000
0x01_0000_3C00
0x01_0000_3800
0x01_0000_3400
0x01_0000_3000
0x01_0000_2C00
0x01_0000_2800
0x01_0000_2400
0x01_0000_2000
0x01_0000_1C00
0x01_0000_1800
0x01_0000_1400
0x01_0000_1000
0x01_0000_0C00
0x01_0000_0800
0x01_0000_0400
0x01_0000_0000
0x00_0000_3C00
0x00_0000_3800
0x00_0000_3400
0x00_0000_3000
0x00_0000_2C00
0x00_0000_2800
0x00_0000_2400
0x00_0000_2000
0x00_0000_1C00
0x00_0000_1800
0x00_0000_1400
0x00_0000_1000
0x00_0000_0C00
0x00_0000_0800
0x00_0000_0400
0x00_0000_0000
(0x00_0000_0400) hbm_mem_ch15/T/0
(0x00_0000_0400) hbm_mem_ch14/T/0
(0x00_0000_0400) hbm_mem_ch13/T/0
(0x00_0000_0400) hbm_mem_ch12/T/0
(0x00_0000_0400) hbm_mem_ch11/T/0
(0x00_0000_0400) hbm_mem_ch10/T/0
(0x00_0000_0400) hbm_mem_ch9/T/0
(0x00_0000_0400) hbm_mem_ch8/T/0
(0x00_0000_0400) hbm_mem_ch7/T/0
(0x00_0000_0400) hbm_mem_ch6/T/0
(0x00_0000_0400) hbm_mem_ch5/T/0
(0x00_0000_0400) hbm_mem_ch4/T/0
(0x00_0000_0400) hbm_mem_ch3/T/0
(0x00_0000_0400) hbm_mem_ch2/T/0
(0x00_0000_0400) hbm_mem_ch1/T/0
(0x00_0000_0400) hbm_mem_ch0/T/0
(0x00_0000_0000) hbm_mem_ch15/T/0
(0x00_0000_0000) hbm_mem_ch14/T/0
(0x00_0000_0000) hbm_mem_ch13/T/0
(0x00_0000_0000) hbm_mem_ch12/T/0
(0x00_0000_0000) hbm_mem_ch11/T/0
(0x00_0000_0000) hbm_mem_ch10/T/0
(0x00_0000_0000) hbm_mem_ch9/T/0
(0x00_0000_0000) hbm_mem_ch8/T/0
(0x00_0000_0000) hbm_mem_ch7/T/0
(0x00_0000_0000) hbm_mem_ch6/T/0
(0x00_0000_0000) hbm_mem_ch5/T/0
(0x00_0000_0000) hbm_mem_ch4/T/0
(0x00_0000_0000) hbm_mem_ch3/T/0
(0x00_0000_0000) hbm_mem_ch2/T/0
(0x00_0000_0000) hbm_mem_ch1/T/0
(0x00_0000_0000) hbm_mem_ch0/T/0
64G
4G
256M
16M
1M
64K
4K
256
16
1
Read/write

Figure 8-6: HBM Striped-Mode Addressing

8.2.3 Linear Mode Addressing

This is the second of two optional addressing modes for PCIE/CPU accesses to HBM. Each channel of HBM is a contiguous one-sixteenth of the full HBM address range.

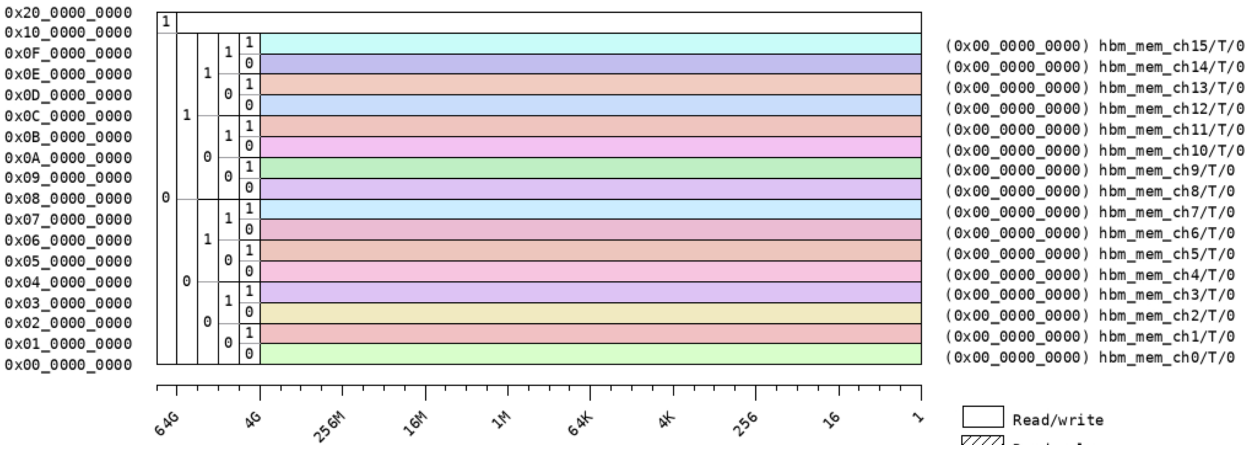

0x20_0000_0000
0x10_0000_0000
0x0F_0000_0000
0x0E_0000_0000
0x0D_0000_0000
0x0C_0000_0000
0x0B_0000_0000
0x0A_0000_0000
0x09_0000_0000
0x08_0000_0000
0x07_0000_0000
0x06_0000_0000
0x05_0000_0000
0x04_0000_0000
0x03_0000_0000
0x02_0000_0000
0x01_0000_0000
0x00_0000_0000
(0x00_0000_0000) hbm_mem_ch15/T/0
(0x00_0000_0000) hbm_mem_ch14/T/0
(0x00_0000_0000) hbm_mem_ch13/T/0
(0x00_0000_0000) hbm_mem_ch12/T/0
(0x00_0000_0000) hbm_mem_ch11/T/0
(0x00_0000_0000) hbm_mem_ch10/T/0
(0x00_0000_0000) hbm_mem_ch9/T/0
(0x00_0000_0000) hbm_mem_ch8/T/0
(0x00_0000_0000) hbm_mem_ch7/T/0
(0x00_0000_0000) hbm_mem_ch6/T/0
(0x00_0000_0000) hbm_mem_ch5/T/0
(0x00_0000_0000) hbm_mem_ch4/T/0
(0x00_0000_0000) hbm_mem_ch3/T/0
(0x00_0000_0000) hbm_mem_ch2/T/0
(0x00_0000_0000) hbm_mem_ch1/T/0
(0x00_0000_0000) hbm_mem_ch0/T/0
64G
4G
256M
16M
1M
64K
4K
256
16
1
Read/write

Figure 8-7: Linear Mode Addressing

8.3 Converting between Tensor-Mode and Striped-Mode HBM Access

Tensors produced or consumed by the UIEs, and stored in HBM, are mapped into HBM using "Tensor Mode". If a CPU or PCIE needs to access this data using "Striped Mode", it must "swizzle" the data to match the expected mapping.

For example, consider a tensor mapping in which the data is unrolled into a linear sequence of bytes in memory. For notational convenience, we can represent the tensor as a sequence of 8-byte blocks numbered from 0 to N, with 0 being least significant block stored in memory.

Tensor byte index	8-byte block index
0x0..0x7	0
0x8..0xF	1
0x10..0x17	2
…	..
N*8 .. N*8 + 7	N

For the tensor mode mapping, is stored in HBM as follows. Note that the channel order for sequential 8-byte blocks (from block index 0 to 15) is {14, 15, 12, 13, 10, 11, 8, 9, 6, 7, 4, 5, 2, 3, 0, 1}.

Tensor-mode (byte) Address	8-byte block indexes				
	Chan 14	Chan 15	…	Chan 0	Chan 1
0x0..0x7F	0x0	0x1	…	0xE	**0xF**
0x80..0xFF	0x10	0x11		0x1E	**0x1F**
0x100..0x17F	0x20	0x21		0x2E	**0x2F**
…	…	…	…	…	…
0x3F80..0x3FFF	0x7F0	0x7F1		0x7FE	**0x7FF**
…	…	…	…	…	…
N*x128..N*128+127	N*16	N*16+1		N*16+14	N*16+15

Table 8-1: Tensor-mode HBM mapping of tensor data

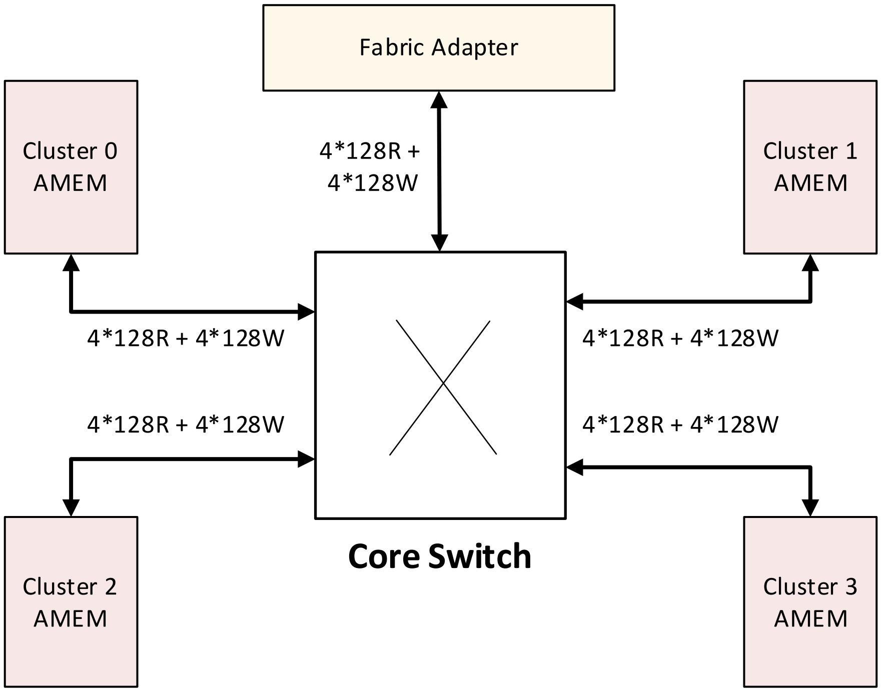
When accessed in Striped-Mode, the data would be organized as follows. Each row in the table corresponds to one channel in striped mode, which is 1024 bytes. Notice the swizzled channel order: for example, the right-most column in Table 8-1 (tensor mode) corresponds to the 2nd row in Table 8-2.

Stripe-mode (byte) Address	Striped-mode HBM channel	8-byte block indexes (128 x 8B blocks per channel)	Group size (bytes)	
0x0..0x3FF	0	0xE, 0x1E, 0x2E, … , 0x7FE	1K	16K
0x400..0x7FF	1	**0xF, 0x1F, 0x2F, … , 0x7FF**	1K	
0x800..0xBFF	2	0xC, 0x1C, 0x2C, … , 0x7FC	1K	
0xC00..0xFFF	3	0xD, 0x1D, 0x2D, … , 0x7FD	1K	
…	…	..	…	
0x3800..0x3BFF	14	0x0, 0x10, 0x20, … , 0x7F0	1K	
0x3C00..0x3FFF	15	0x1, 0x11, 0x21, … , 0x7F1	1K	
0x4000..0x43FF	0	0x80E, 0x81E, 0x82E, … , 0xFFE	1K	16K
0x4400..0x47FF	1	0x80F, 0x81F, 0x82F, … , 0xFFF	1K	
…	…	…	…	
… etc …	…	…	…	…

Table 8-2: Striped-mode HBM mapping of tensor data

9 CORE SWITCH

The Core Switch moves data from any AMEM in any Cluster to AMEM in any other Cluster, and also between the Fabric Adapter and AMEM in any Cluster.



There are four Core Switch "planes" where each plane is a 5x5 buffered crossbar with 128-byte input/output buses. Each data source (4x AMEM clusters and Fabric Rx) has one 128-byte connection to each plane, for total of 4x128B connections from each source to the switch, and 4x128B connections from the switch planes to each data destination (4x AMEM clusters and Fabric Tx).

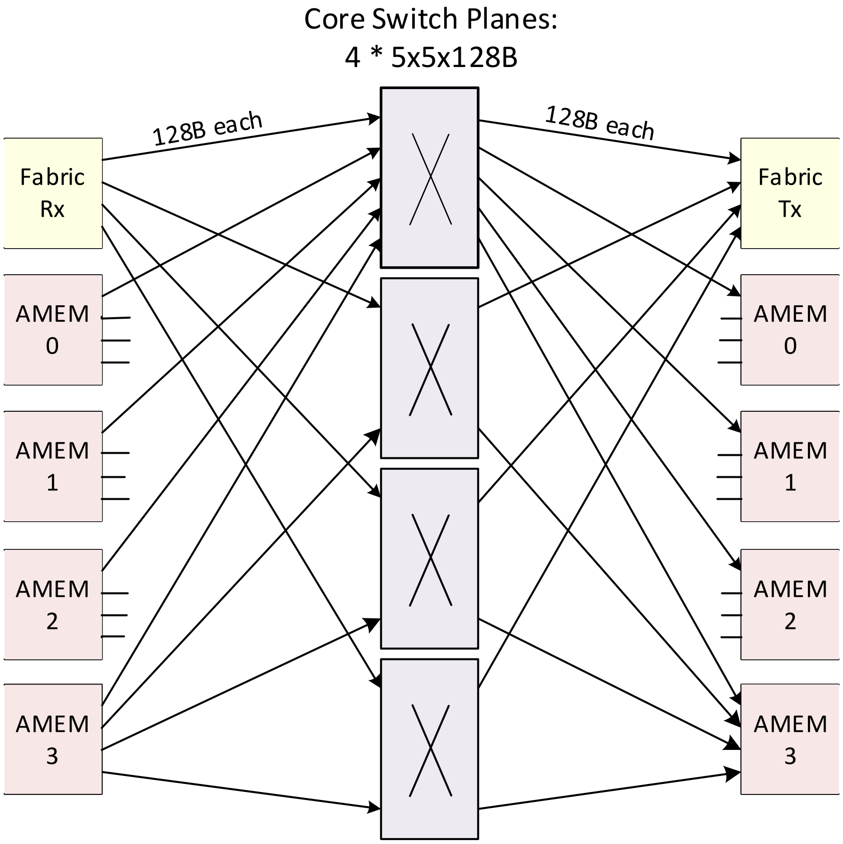


Figure 9-1: Core Switch consists of 4 5x5x128B switch-planes

The source "clients" are the Inter-cluster DMA engines (ICDMA) and Fabric Tx and Rx RDMA engines. For more information on these DMA engines, see sections 12.2, 12.3 and 10.8.8 respectively.

- Each cluster has **three ICDMA engines** which can each transfer 128B/cycle from that (source) cluster to one of the other three (destination) clusters, performing AMEM-to-AMEM DMA transfers.
- Each cluster has **one Fabric Tx RDMA** engine, which can transfer 128B/cycle from that cluster's AMEM to *any* of the four Tx fabric adapter "pipes" to be transmitted across the fabric.
- Each cluster has **one Fabric Rx RDMA** engine, which can transfer 128B/cycle of data received from the fabric to that cluster's AMEM.

Each source client load-balances across the Core-switch planes by "spraying" the requests for each destination across the planes. Each source client maintains a "spray pointer" for each destination, and does a round-robin plane selection separately for each destination.

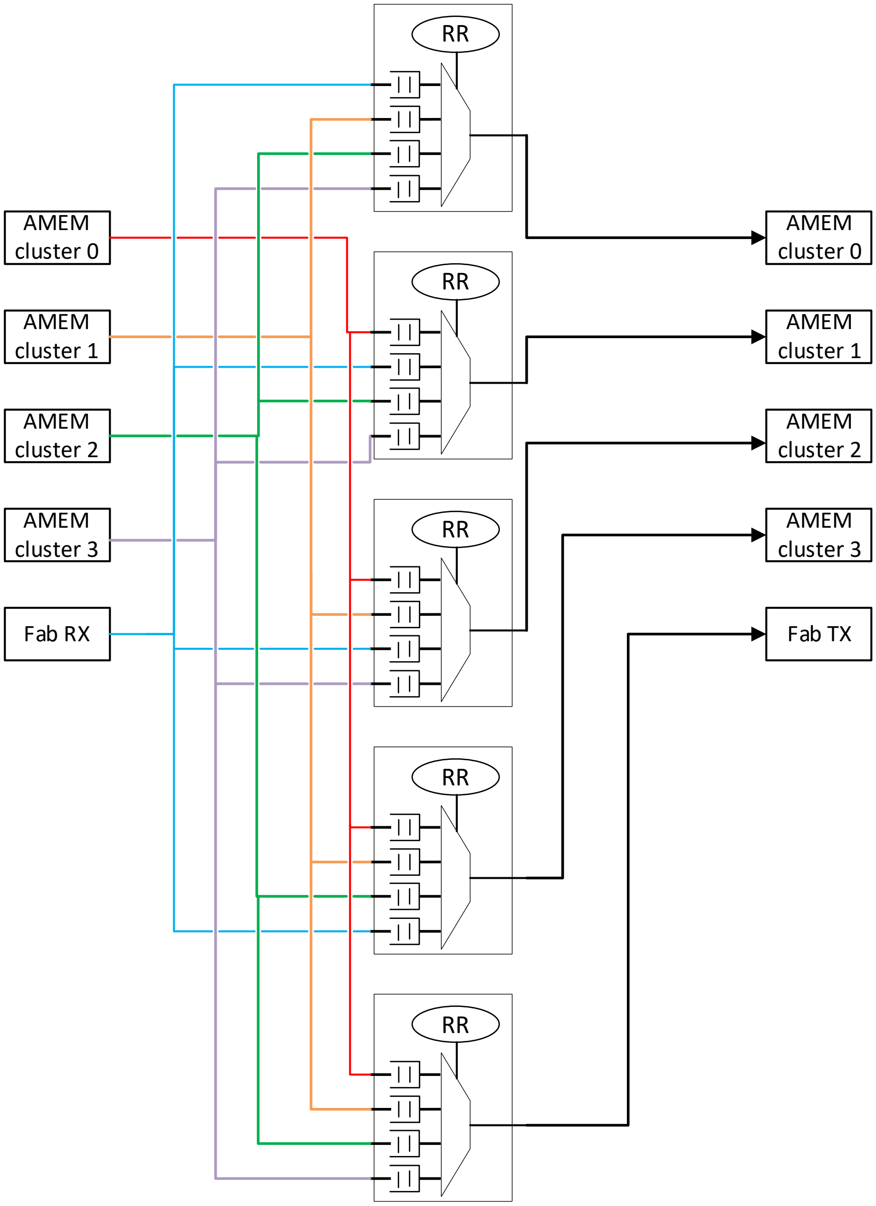


Figure 9-2: One CoreSwitch Plane

Each plane consists of per-source, per-destination FIFO buffers, with a mux for each destination using round-robin arbitration. The per-destination muxes are 4:1 (not 5:1) because there is no loopback path from a source to itself (i.e., no path from AMEM in one cluster back to itself, nor from Fabric Rx to Fabric Tx.). Each client maintains FIFO buffer credits for each destination, for each plane. A client will not issue data to the switch if the desired (plane, destination) buffer has no credits available.

Further details are available in the CoreSwitch (CSX) Specification document .

10 FABRIC ADAPTER

Pyxis provides 36x112Gbps SERDES to a cell-based switch fabric, and the fabric-interface controller to interoperate with the Juniper "BF" Fabric chips. This can carry over 400GB/s of payload traffic (in each direction) to and from the fabric. The transfers are divided into 4 lanes of ~100GB/s each, with each lane typically corresponding to one UIE Cluster.

The Fabric Adapter provides a reliable RDMA-write service, and an unreliable datagram service. The RDMA copies data directly from AMEM on one Pyxis into AMEM on another Pyxis chip, and provides a lossless transport service, including selective retry for the infrequent case of an error in communication. This provides low latency for collective data-sharing applications, such as `all_reduce`, `reduce_scatter`, and `all_gather`.

The Fabric Adapter also supports a datagram service for sending data or messages from a transmit queue (on a source Pyxis chip) to a receive queue (on a destination Pyxis chip). These queues are based in AMEM (in a particular cluster), and are software managed. This is not a lossless "reliable" transport, and allows for dropped datagrams; the CPUs on each end are then responsible for implementing a reliable transport layer in software if desired.

10.1 FABRIC PAGES AND CELLS

The Juniper Fabric manages data transfer in units of "pages", where a page can contain one or more packets, with up to 16K bytes of (fabric) payload per page. This is Virtual-Output-Queue (VOQ) based system where packets are enqueued by the client logic (Pyxis fabric adapter) to a VOQ which points to an Output Queues (OQ) on a target Pyxis chip. Each (source chip) VOQ is mapped to exactly one (target chip) OQ, but VOQs on multiple source Pyxis chips will be mapped to the same OQ. Pyxis will map exactly one packet in each (Juniper) fabric page.

The underlying fabric operation uses a 3-way handshake to transport each page. In the first phase a Request message is sent from the source ("**Tx**", or fabric-ingress) chip to the target ("**Rx**", or fabric-egress) chip, to indicate the desired page size and OQ. The Rx chip keeps track of all Requests from all Tx chips for each OQ, and subsequently returns a Grant message. After receiving the Grant, the Tx chip may transmit a page's worth of data (as indicated by the "size" of the Grant).

This handshaking protocol introduces a start-up latency of one Fabric round-trip-time (RTT), which may be around 2us in a typical case. Multiple Requests and Grants may be outstanding and in-flight between any Tx/Rx chip-pair. Once the data connection has started up (in case of a large transfer), Requests and Grants will continue to be exchanged and data can flow continuously. However, every new communication between chips will experience this start-up latency.

The Juniper Cell Fabric transports data in variable-length *cells*, usually with 176 (payload) bytes per cell. Pyxis must convert internal data bus widths from (typically) 128 bytes to 176 or sometimes multiples of 44 bytes. Similarly, data must be received from the fabric in this format and converted back to Pyxis-friendly data widths. This data-width mismatch implies that transported data may not be directly read or written from AMEM to/from the fabric, but requires buffering and gearbox logic to adapt to the Juniper format.

Normally, Juniper fabric logic will attempt to pack multiple (smaller) packets into each page, to spread the cost of Request/Grant arbitration over a larger amount of data. However, Pyxis will disable this packing, and will ensure that each page contains exactly one packet – hence we can use the terms “packet” and “page” interchangeably for Pyxis. This is fine for Pyxis because for RDMA applications, Pyxis will divide the relatively large blocks of data (to be RDMA’ed) into packets of up to 16K bytes which will each occupy one page.

Juniper Fabric logic supports cell sizes from 96 to 176 bytes, with 16 byte “chunk” granularity. Normally, a page is cellified into maximum-size (176B) cells, but for pages which are not a perfect multiple of 176B, the Juniper Fabric logic will cellify the last *two* cells of a page so that no cell has fewer than 96 bytes (=6 x 16B chunks). The cellfication rules for any page with two or more cells are shown in the table below.

Page len % 11 (in 16B chunks)	**Num Chunks in next-to-last cell**	**Num Chunks in last cell**	**Valid bytes in next-to-last cell**	**Valid bytes in last cell**
			(big-endian byte numbering)	
1	6	6	0:95	0:95
2	7	6	0:111	0:95
3	8	6	0:127	0:95
4	9	6	0:143	0:95
5	10	6	0:159	0:95
6	11	6	0:175	0:95
7	11	7	0:175	0:111
8	11	8	0:175	0:127
9	11	9	0:175	0:143
10	11	10	0:175	0:159
0	11	11	0:175	0:175

10.2 FABRIC VOQ MAPPING

Pyxis supports up to **8K** VOQs for the Cell Fabric. This is a future-proofing number to support 256 Pyxis chips * 8 clusters/chip * 4 OQs/cluster[4] , requiring a 13-bit VOQ number.

The Juniper VOQs use a 16-bit space; therefore, we need to re-map the 13-bit Pyxis VOQ to the 16-bit Juniper space. However, the Juniper fabric uses a specific construction of the VOQ number as follows (for the first generation of Pyxis):

```
    Juniper_VOQ = {VPFE[10:0], VPFE_OQ[4:0]}
```

where

`VPFE` is a “Virtual PFE” which is basically a collection of 32 OQs, and
`VPFE_OQ` is the OQ index within the VPFE

For the VOQ re-mapping, we would want to select fewer OQs per VPFE. For Pyxis, **each VPFE represents one target cluster on one Pyxis chip**, and would require just **4 OQs/VPFE**.

To provide maximum flexibility, we provide a full mapping from Pyxis four-OQ groups to Juniper VPFEs. The mapping allows inter-chip communication to use a fixed set of VOQs for collective operations, **regardless of which physical Pyxis chips are running the model**, by changing the VOQ mapping table on each chip (rather than changing the trip descriptors which specify the communications transactions). In effect, this is virtualizing the Virtual OQ numbering.

The VOQ re-mapping is as follows:

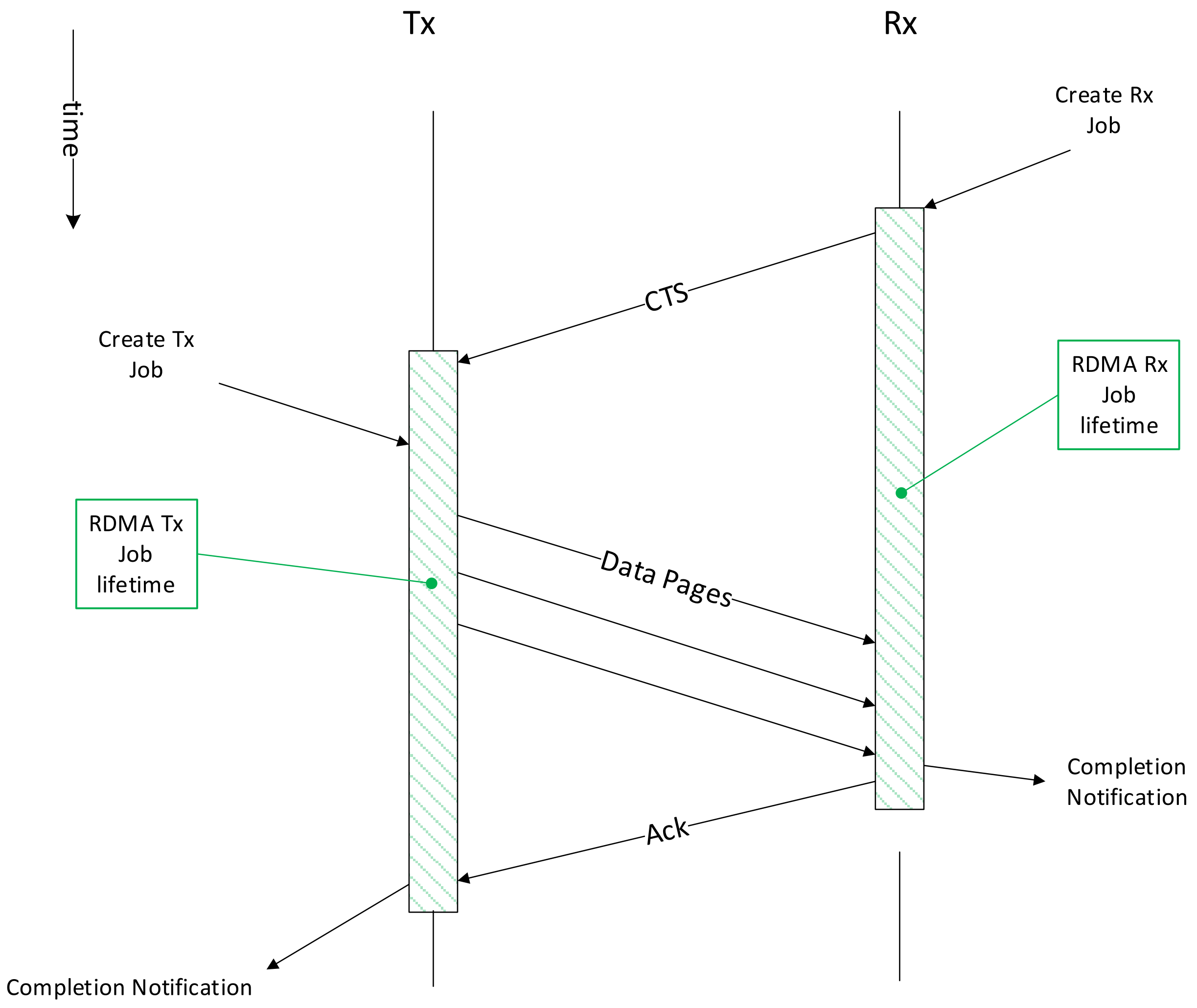
```
    VPFE[10:0]        = Pyxis_VOQ_to_VPFE_map[ Pyxis_VOQ[12:2] ]
    VPFE_OQ[4:0]      = {3’b0, Pyxis_VOQ[1:0]}
```

Where `Pyxis_VOQ_to_VPFE_map` is a 2K x 11-bit lookup table configured by software.

Note: The Juniper VPFE[10:0] also has the **restriction that VPFE[1:0] != 3,** so the legal VPFEs are in the set {0,1 2, 4,5,6, 8,9,10,…,2044, 2045, 2046} . Software must adhere to this restriction when configuring VOQs in Pyxis.

[4] For the initial Pyxis application over Juniper Fabric, the maximums are 144 Pyxis chips * 4 clusters * 4 OQs = **2304 VOQs** needed

10.3 RDMA over Fabric



Pyxis uses a “solicited write” model for RDMA. Before a source chip (Transmitter or “**Tx**”) can RDMA-write data into a destination chip (Receiver, or “**Rx**”), the receiver must allocate memory (in AMEM), assign a receive-job (**Rx Job**) context. The Rx Job includes:

- The destination address and UIE Cluster,
- transfer length in units of 128 bytes, and
- the Tx chip and its source UIE Cluster, and its associated Tx Job ID.

The Tx chip must similarly be programmed with a transmit-job (**Tx Job**) context which includes:

- The source AMEM address and UIE Cluster,
- transfer length in units of 128 bytes, and

- the Rx chip and its destination UIE Cluster, and its associated Rx Job ID.

Once the Rx Job is set up, the Rx chip will send a “Clear to Send” (**CTS**) message to the Tx chip. After receiving CTS message, and after the Tx job has been set up, the Tx chip can commence to transmit the data across the fabric. The CTS message may arrive before or after the Tx job has been initialized; this just requires that the Tx Job ID is not in use by a prior job. (This can be guaranteed by software.)

Each RDMA job may be from 1000’s of bytes up to multiple megabytes. The transfers are divided into large packets of up to 16Kbytes, and packets from multiple RDMAs may be interleaved for transmission across the fabric. This is necessary when communicating with multiple chips in parallel, so the RDMAs can be parallelized.

After all data is transmitted for a job, the receiver will send an **ACK** message, and may also be programmed to issue completion notifications to the Barrier Table to relax hardware barriers (indicating that received data is available, and dependent trips may use the data). At that point, the Rx Job context is retired.

Upon receiving the ACK message, the transmitter may also be programmed to issue completion notifications to its Barrier Table, indicating that the transmitted data buffers may be re-claimed and reused. At that point, the Tx Job context is retired.

10.3.1 Reliable RDMA transport

The underlying hardware for transmitting messages and data is normally very reliable, and uses lossless flow control semantics to eliminate data loss due to congestion. However, no communication mechanism is 100% error free, so a “reliable transport” is built into the RDMA mechanism.

In case of an error, Tx hardware can automatically retry an entire RDMA job, and that is the only granularity of retry for data transfer. There are two main triggers for job retry:

- The receiver may detect an error in the transmitted data sequence comprising the full job. This may be a dropped or missing packet due to fabric transport error, or a timeout waiting for all data to arrive (also possibly due to a dropped last packet). In these cases, the receiver sends a NACK message to the transmitter when the transfer is complete or after the timeout period expires.
- The transmitter may receive a NACK, or timeout waiting for an ACK message after all data for the job has been sent. The timeout can occur in some cases for lost packets (so that the receiver never sees the full transfer), or if an ACK or NACK message is lost.

The Rx chip may also retry sending the CTS message, based on a timeout from when the CTS message is sent to when the first data would be expected to be received.

Retry attempts are limited to just one retry, and in case of an unrecoverable error, the RDMAs job is terminated, and an interrupt signaled to the on-chip CPU.

Reliable RDMA state machines for Tx and Rx are shown in the following figures.

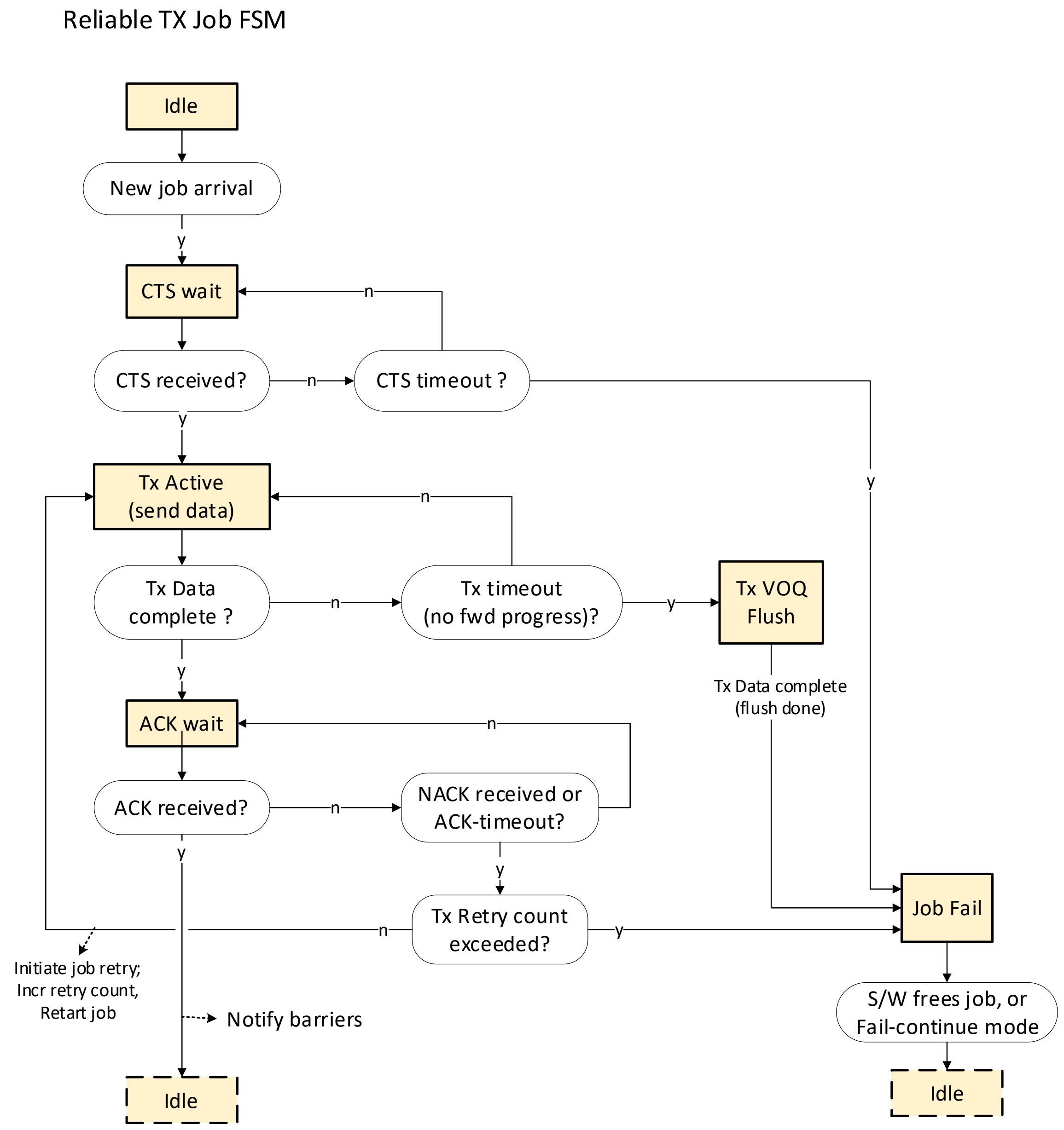


Figure 10-1: Simplified RDMA Reliable Transport State Machine – Tx

Reliable RX Job FSM

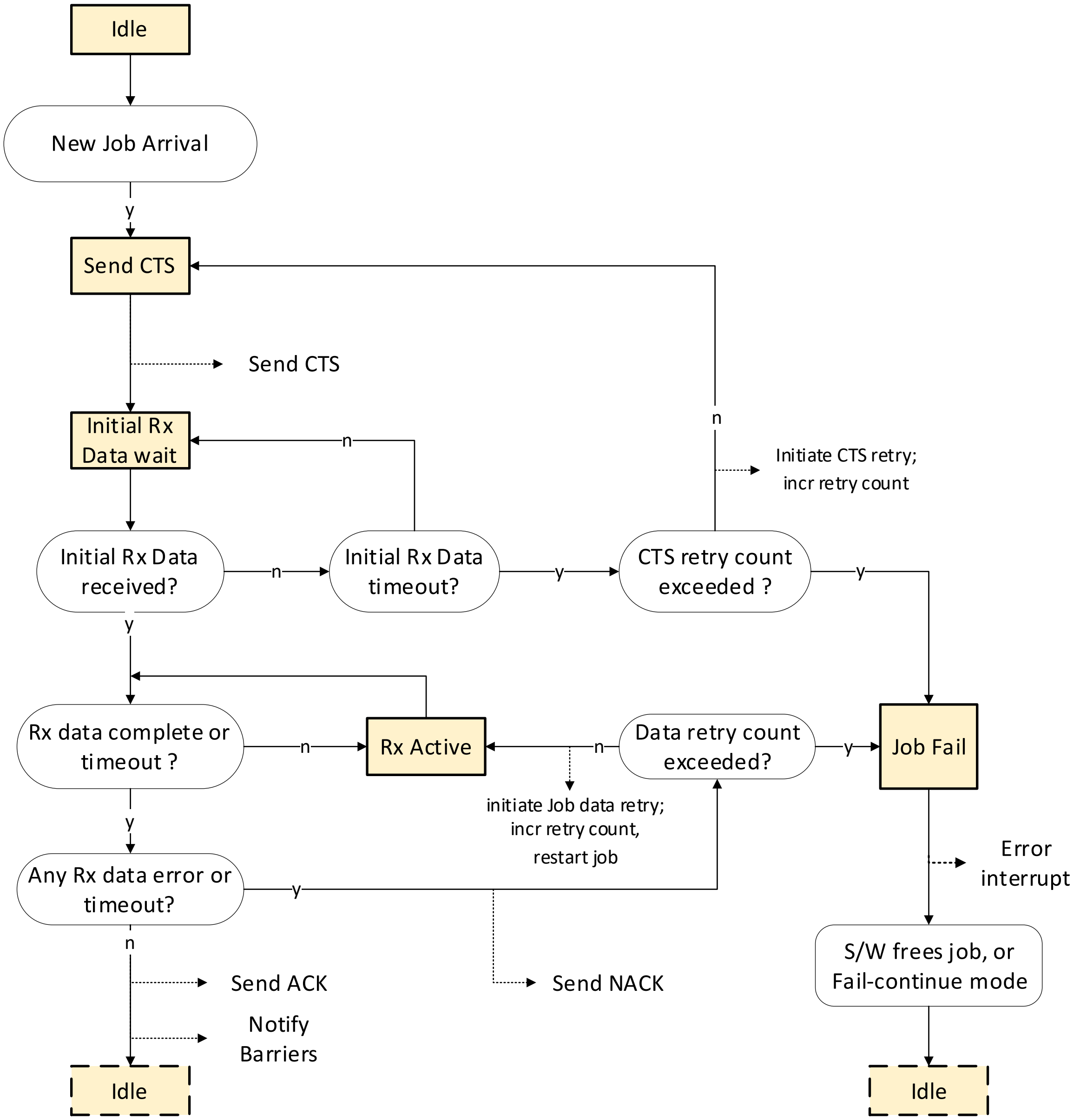

Idle
New Job Arrival
y
Send CTS
Send CTS
Initial Rx Data wait
n
n
Initiate CTS retry; incr retry count
Initial Rx Data received?
n
Initial Rx Data timeout?
y
CTS retry count exceeded ?
y
y
Rx data complete or timeout ?
n
Rx Active
n
Data retry count exceeded?
y
Job Fail
initiate Job data retry; incr retry count, restart job
y
Error interrupt
Any Rx data error or timeout?
y
n
S/W frees job, or Fail-continue mode
Send ACK
Send NACK
Notify Barriers
Idle
Idle

Figure 10-2: Simplified RDMA Reliable Transport State Machine - Rx

10.3.1.1 Clear to Send (CTS) message

CTS is a single packet/single page message automatically generated by Rx RDMA logic when the Rx Job context is established. CTS may also be re-sent after a timeout if no data has been received (suggesting a lost CTS message). CTS messages travel in a different VOQ from RDMA data, and may be given preferential service across the fabric.

CTS messages include the following fields:

Field	Width	Description
Message Type	~2	CTS message type
TJID	8	Tx Job ID, Rx Job ID form an RDMA communication channel ID
RJID	8	
Epoch	1	Incremented each time the TJID/RJID are re-used to de-alias “past” and “future” jobs using the same TJID/RJID. This is managed by hardware
Retry	1	Indicates if this is a retried CTS message.

10.3.1.2 RDMA ACK / NACK messages

ACK/NACK is a single packet/single page message automatically generated by Rx RDMA logic when a job is completed successfully or unsuccessfully. ACK/NACK messages travel in a different VOQ from RDMA data, and may be given preferential service across the fabric.

ACK/NACK messages include the following fields:

Field	Width	Description
Message Type	~2	ACK or NACK message type
TJID	8	Tx Job ID, Rx Job ID form an RDMA communication channel ID
RJID	8	
Epoch	1	Incremented each time the TJID/RJID are re-used to de-alias “past” and “future” jobs using the same TJID/RJID
Retry	1	Indicates if this ACK or NACK applies to a retried job.

10.4 Tx Job ID and Rx Job ID semantics

A communication channel for one RDMA Job is created by linking a Tx Job ID (from a source chip) with an Rx Job ID (on a destination chip), along with a Job-ID “Epoch” value:

- Hardware provides a pool of Tx & Rx Job IDs which may be allocated for communication channels to/from each remote chip. The assignments are managed by software.
- Each UIE Cluster has an independent Job ID space.
- Up to 256 active jobs each of Tx and Rx per UIE cluster, per chip. Multiple Tx Jobs may use the same Rx Job ID (for different Rx chips / clusters); and similarly, multiple Rx Jobs may use the same Tx Job ID (for different Tx chips / clusters).

RDMA jobs are tracked via Tx and Rx **Job Tables** in hardware. The Job Tables maintain the state of the connections between two chips. Each Tx Job / Rx Job pair is a stateful connection which lasts for the lifetime of one RDMA job. Job Table entries are created when hardware processes each RDMA Tx/Rx job descriptor, and retired when the job is complete.

The design intent for software management of Tx/Rx Job IDs:

- On each Tx chip, software should allocate a range of Tx Job IDs (**TJID**s) for each Rx target.
- On each Rx chip, software should allocate a range of Rx Job IDs (**RJIDs**) for each Tx source.
- Each TJID – RJID mappings should be static. A TJID **must not** be assigned to a different RJID or a different Rx chip while there is active traffic flowing between the existing TJID-RJID pair. And similarly, an RJID must not be reassigned to a different TJID or different Tx chip once active)
- The one-bit Job-ID **Epoch** is used to detect messages or data associated with a prior or future invocation of the same TJID/RJID pair. The Epoch bit is initialized to zero (for each TJID and RJID) on the transmitter & receiver respectively, and is incremented automatically by hardware on each new job invocation. Each page and each message sent between chips carries the Tx/Rx Job IDs and Epoch fields; the message Epoch is checked against the Tx Job Table entry’s Epoch to detect future (CTS) messages, and data page Epoch is checked against the Rx Job Table entry’s Epoch to detect prior job (stale or retransmitted) data pages.

The following figure is an example of Tx/Rx Job ID mapping from one Tx chip to two Rx chips. The Job ID mappings are entirely under software control, this illustrates one possible mapping policy

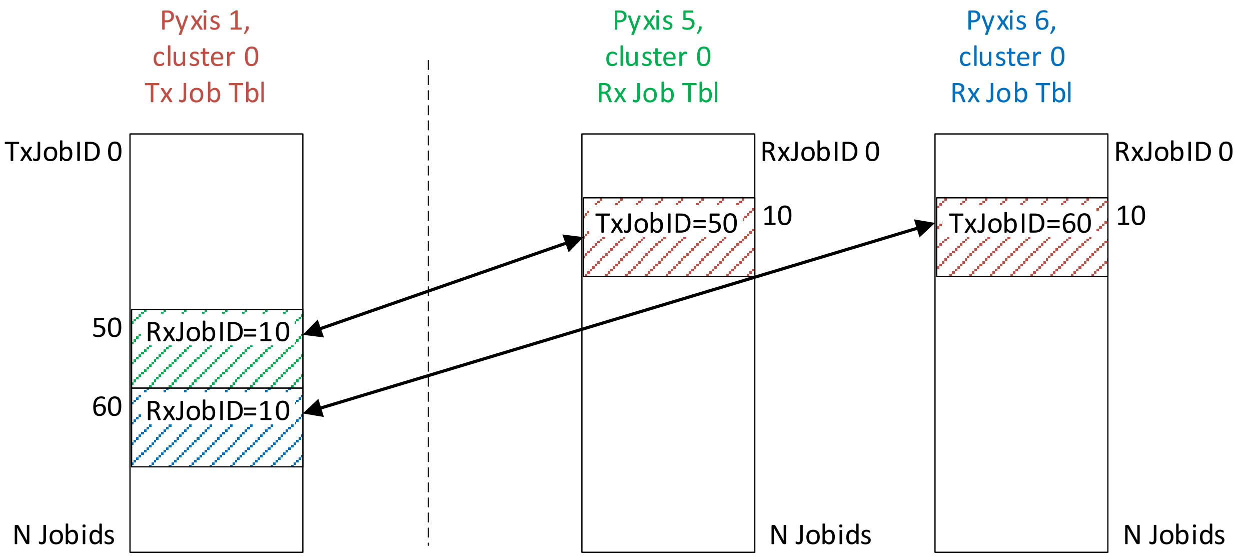
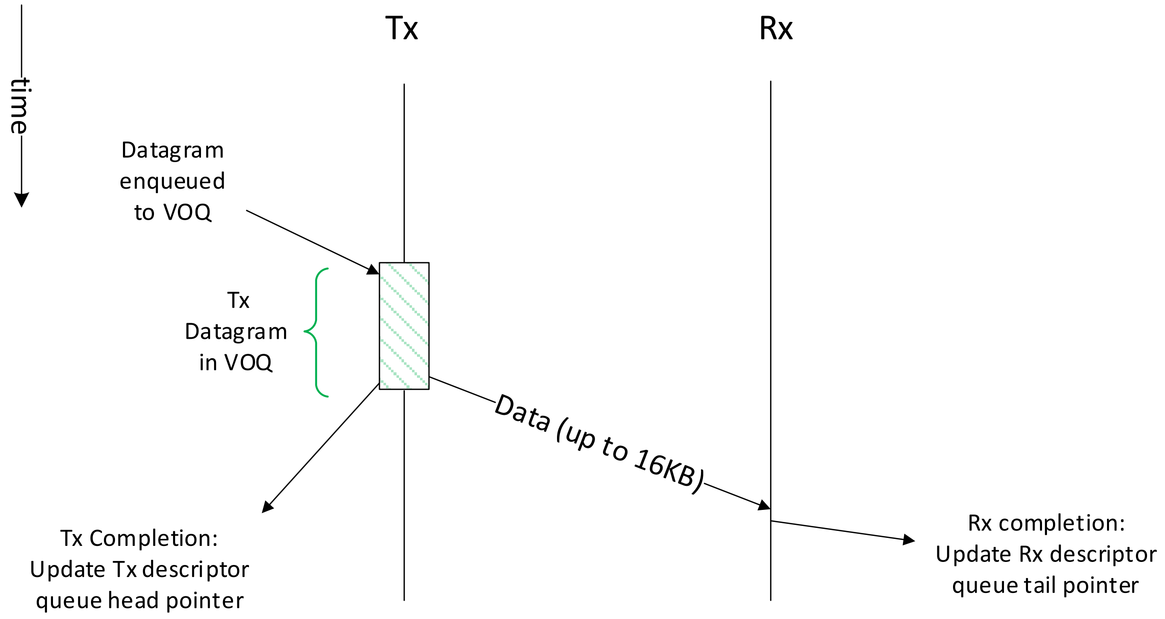


Figure 10-3: Example TJID - RJID mappings for RDMA communication

In cases where Cluster *j* on one Pyxis only communicates with the same Cluster *j* on the other Pyxis chips, there is no ambiguity in re-using the same Job IDs in each cluster. For the example of Figure 10-3, The (TJID=50, RJID=10) pair may be used to link Pyxis 1 Cluster 0 (Tx) to Pyxis 5 Cluster 0(Rx). The **same** (50,10) mapping may also be used for Clusters 1, 2 and 3.

However, when cross-communicating among UIE clusters on different chips, the Job IDs must be allocated as if each remote chip + cluster were a separate target. In the preceding example, the TJID, RJID = (50, 10) might be used to link Pyxis 1, Cluster 0 to Pyxis 5, Cluster 0. But a different mapping, such as (51, 10) would be used to link Pyxis1, Cluster 0 to Pyxis 5, Cluster **1**. The allocation and management of these IDs is entirely under software control.

10.5 Datagram over Fabric


Tx
Rx
time
Datagram enqueued to VOQ
Tx Datagram in VOQ
Data (up to 16KB)
Tx Completion: Update Tx descriptor queue head pointer
Rx completion: Update Rx descriptor queue tail pointer

Pyxis “Datagrams” provide a packet transport service over the cell fabric

- Single packet (page) per datagram
- Data transmitted from a Tx Queue (on source chip) to an Rx Queue (on target chip). The TxQ is written by software, and the RxQ is read by software, so this provides a CPU-to-CPU transmission service.
- There are two TxQs and two RxQs per cluster. This can provide a simple high/low class of service differentiation.
- Datagram servicing Does not use RDMA job tables; no Clear-to-send message, no timers and no job state tracking. This does not provide reliable-transport semantics.
- Received datagram packets will be dropped if the indicated RxQ (ring) has no free descriptors.
- Tx and Rx descriptor processing does not use hardware barriers. Synchronization is provided by the TxQ and RxQ head/tail pointers.

10.6 Fabric Page Header

Each fabric page carries a 16-byte header for communication between Tx and Rx Pyxis chips, inserted and removed by the Pyxis fabric-adaptation Tx and Rx logic. This header appears as payload to the

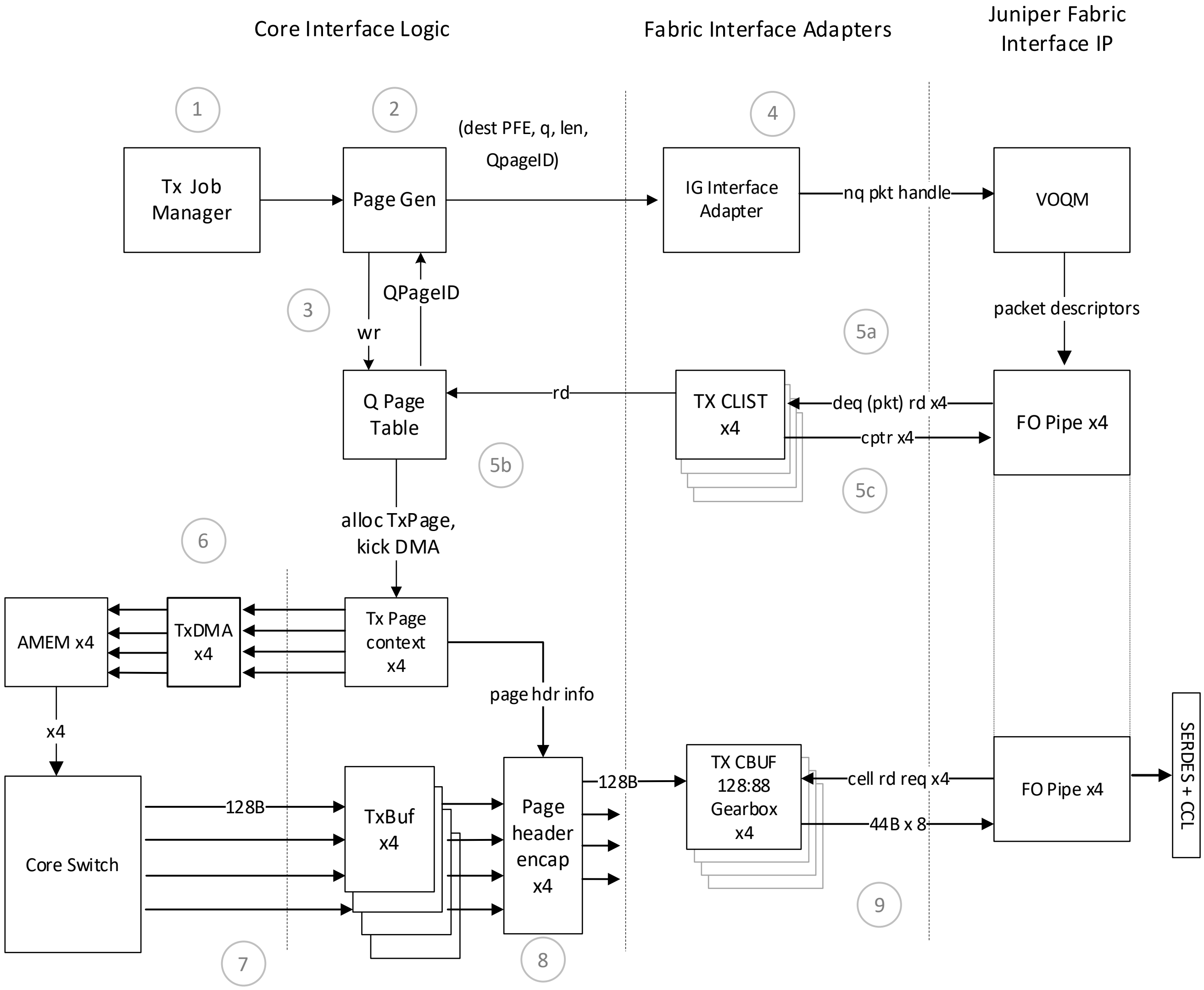
Juniper fabric, and carries information needed by the hardware layer above the fabric (for RDMA or Datagram support).

Hardware generated messages sent over the fabric (CTS, ACK and NACK) consist of only the page header and padding bytes needed to satisfy the minimum packet size of the Juniper fabric IP. For RDMA data and Datagram data pages, the header is prepended to the page data.

The page header includes:

- Page-type field (RDMA Data, Datagram Data, CTS-message, ACK-message, NACK-message)
- Tx Job ID,
- Rx Job ID,
- Job-ID Epoch.
- Retry-bit (used by the reliable-transport retry algorithm)
- Last-page flag (for RDMA).
- RxQ Index (for Datagrams) to identify the receive descriptor queue
- Page size, with byte-level granularity. . The page size does not include the page-header itself. For RDMA pages, the page size is restricted to a perfect multiple of 128 bytes. For Datagrams this can be any size from 1 byte up to 16256. For hardware messages, the page size is zero bytes.
- Page sequence number within the full job for RDMA. This helps Rx hardware to detect lost pages when there is a skip in the received sequence number. Sequence numbers always start from zero for each job.

10.7 Tx Fabric Adapter



The operational flow through the Tx Fabric Adapter is as follows:

[1] Software prepares RDMA Tx descriptors for each RDMA "job", which is a block of data to be transferred, and enqueues these onto per-RDMA Tx engine descriptor queues. The descriptors include the destination chip, cluster, and traffic-class (encoded as a Destination Queue), as well as control information for the RDMA including Tx and Rx RDMA Job IDs, and desired transfer Page size. Each RDMA job may be up to several Megabytes, or as little as 10's of KBytes.

Each cluster has an independent **Tx Job Manager**, with its own descriptor queues, for RDMA data sourced from that cluster's AMEM. The job at the head of each Tx RDMA queue must pass the usual barrier-table wait.

This hardware also supports inter-chip packet transfer via CPU-managed *datagrams* which are single up-to-16KB packets. The datagram service may be used for CPU-to-CPU (queue to queue) communication. Datagrams have their own descriptor queues.

[2] Each RDMA job is then dynamically assigned to a channel in the Tx **Page Generator** engine, which divides the DMA transfers into “pages” of up to 16KB. Each page is treated like a single large packet for the fabric. There are 4 page-generator engines, one per source cluster.

The page-generator engine does round-robin selection among the channels (jobs) on a page-by-page basis. The channels are responsive to congestion-based flow control on their respective VOQs. Congestion monitoring is necessary to regulate the number of pages enqueued to the system for any one VOQ and in aggregate, otherwise the entire job (which may be multiple MB’s) could be enqueued in a short time, using up all internal resources for page tracking.

[3] A “**QPage**” entry is created for each generated page. The QPage table provides metadata for each page outstanding in the Juniper fabric queueing system. The number of QPages is approx. 2 * fabric RTT * fabric bandwidth / page_size, generously up-sized to account for a portion of small pages, as well as for some instantaneous queue congestion. The QPage table is shared by all Clusters’ RDMAs

Each QPage includes the source address of the data in AMEM, page size, page offset within the full RDMA and RDMA ID fields for communicating with the remote Pyxis.

[4] After allocating a QPage, the **Ingress Group** (interface adapter module) issues the enqueue message to the **VOQ Manager** (**VOQM**) in the Juniper IP. The critical information includes the VOQ, packet (page) size and a “packet handle” which is a pointer to the data structure representing the page. In the Juniper IP, the packets are created by stitching together a linked-list of data cells, and the Packet Handle is a pointer to this linked-list structure. Pyxis will instead provide the QPage ID as the packet handle.

The Juniper VOQM normally combines multiple packets into each “page”, and negotiates with the destination chip to transfer data on a page basis (rather than individual small packets). Pyxis will allow exactly one packet in each page.

VOQM maintains the queue of pages, and handles the request/grant handshaking with the destination Pyxis chip across the fabric. After (at least) a fabric RTT, the page would be “granted” and made ready to transfer. At this point, the packet descriptor (handle, size, etc.) is handed to the **Fabric Out** (**FO**) block in Juniper IP. FO will take care of the data transfer.

[5a] The FO block has four “pipes'' to handle the bandwidth load, where each pipe is responsible for ~100GB/s (800Gbps) of fabric traffic. This is strictly for implementation reasons, to scale up to arbitrarily large total bandwidth. Each FO pipe datapath can handle 88 Bytes/cycle, which is one Juniper cell per 2 cycles.

FO load-balances across the 4 pipes on a page-by-page basis. There is no affinity between pipes and the source or destination clusters or queues - any page can end up on any FO pipe. This does have the unfortunate consequence that multiple pipes may be pulling from the same AMEM cluster concurrently.

Each FO pipe then sends **CLIST** Read requests to retrieve the data-cell pointers (from the Juniper design concept). CLIST is the Juniper block which maintains the linked-list-of-cells data structures for each packet. The request from FO provides the packet-handle and number of cell pointers, and expects to receive a stream of cell pointers in return. Pyxis implements an Interface-Adapter module for CLIST to *mimic* these interfaces.

[5b] Tx CLIST reads the QPage context (taken from the packet handle), creates a Tx Page context, and allocates space in the **TxBuffer** for the page. The TxBuf is a data buffer for decoupling the AMEM from the Fabric on a page-by-page basis. The Tx page context includes the source address, RDMA ID, destination address, page offset, etc., as well as pointers to the TxBuf data memory allocated to buffer the page. There is one TxBuf structure for each FO pipe, supporting 1/4th of the total fabric bandwidth on each pipe.

The TxBuf provides enough buffering to cover the time from when FO signals the CLIST Read request for a page, until the associated page data has been retrieved from AMEM and transmitted toward FO. For page-based transactions, this is sized up to two max-sized pages (32KB) per FO pipe.

[5c] Tx CLIST returns a series of *artificial* CellPointers to the FO block. Note that this refers to 176B Juniper Cells. These artificial CellPointers can be constructed with the TxBuf Page ID and cell-index in the page, but they are not pointers to any actual memory location.

[6] A **DMA Read transfer** is initiated for each TxBuf Page context, to move the data from AMEM to the TxBuf. There is one TxDMA engine for each cluster, which can receive page-transfer requests for any FO pipe. This engine generates the AMEM read operations (for the AMEM in the associated cluster) for each AMEM flit in the page, load-balanced over multiple planes of the Core Switch. Note: this TxDMA engine is implemented in the CoreSwitch block, rather than in the Tx Fabric Adapter, because it is more efficient to initiate the data transfers near the source (AMEM) rather than near the destination (Fabric Tx)

[7] When the data returns from the Core Switch, it must be routed from the switch planes to the TxBuf in the targeted pipe. Flits can arrive out of order from the Core Switch planes, so TxBuf maintains a reorder context. Each Flit is transferred through the Core Switch with metadata including Start-of-page, end-of-page and TxBuf address.

[8] A 16-byte page header is created, and prepended to each page data by the **HeaderEncap** module. This header includes information needed by the destination Pyxis chip, most critically the DMA Job IDs which identify the specific transfer job.

[9] FO subsequently sends the stream of Cell Pointers (from 5c) to read data from the CentralBuffer (CBUF) in the Juniper design. Pyxis implements **Tx CBUF** to emulate this behavior, and to translate the 128B AMEM flits into 176B fabric cells.

Tx CBUF receives data from Header Encap in FIFO order. Tx CBUF then performs a 128byte to 88 Byte gearbox to conform to Juniper IP datapath requirements. There are four Tx CBUF instances, supporting 88B/cycle (corresponding to ~100GB/s) each delivered to FO. FO then relays this data to the fabric.

The Tx Fabric Adapter modules are described in the following sub-sections.

10.7.1 Tx Job Manager

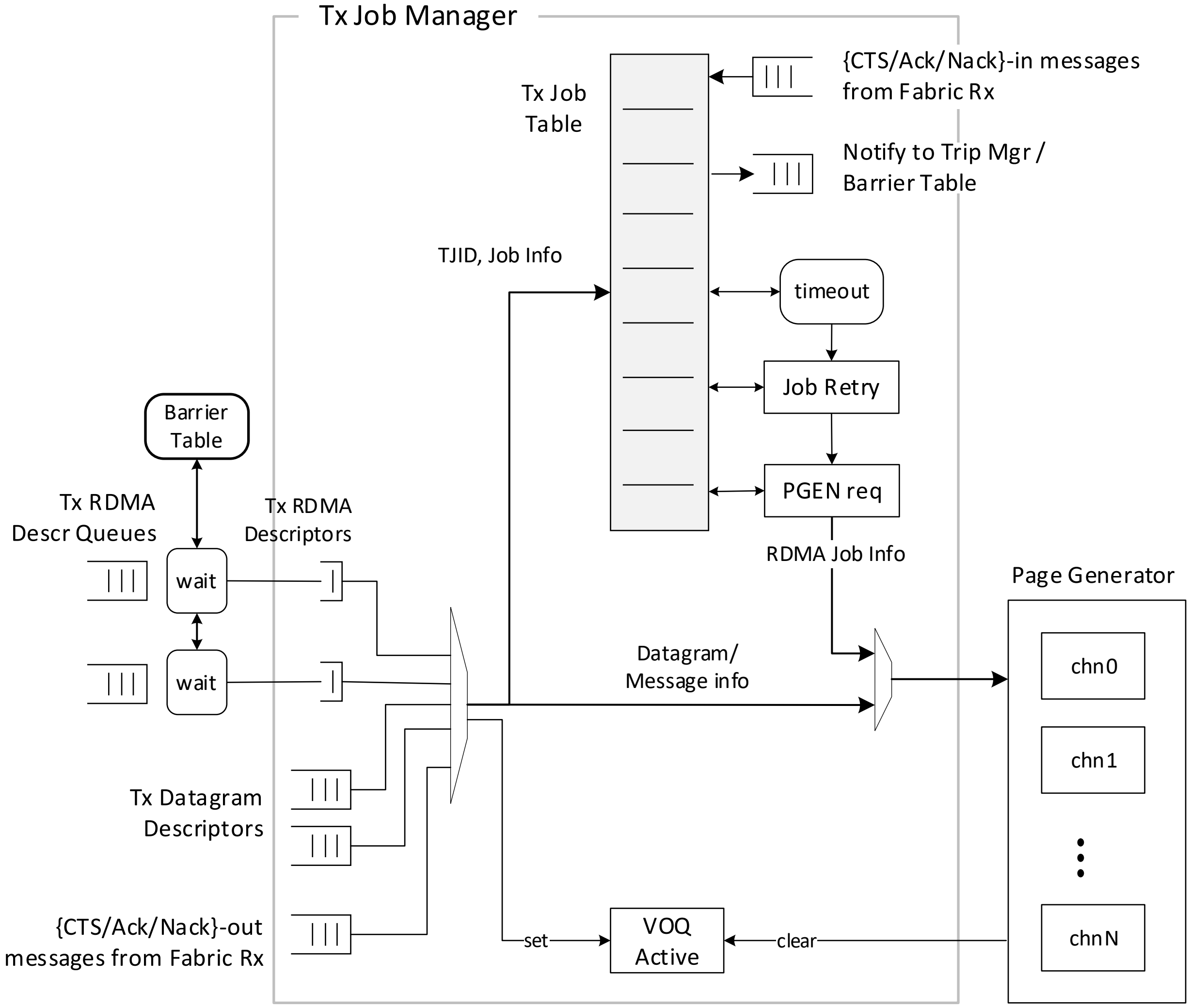


Figure 10-4: Tx Job Manager

There is one Tx Job Manager instance per UIE cluster, for RDMA and Datagram data sourced from that cluster's AMEM.

Tx jobs are enqueued to descriptor queues. There are 2 RDMA and 2 Datagram descriptor queues. The head entries of the RDMA queues may undergo Barrier-waits (in the Trip Manager), then the entries are moved into shallow FIFOs at the entrance to the Tx Job Manager. The barriers can be used to ensure that the RDMA-source data is ready to be sent, and also ensures that the Tx Job Table entry is not currently in use

10.7.1.1 Tx Job Manager operation for RDMA

- The first step in job service is to select a job from one of the descriptor queues. There are two queues for RDMA jobs, and two for Datagrams. Selection uses a simple round-robin arbitration.
- The next step in job service is to test and set a "VOQ Active" lock on the destination VOQ. This lock is set as the job is accepted from descriptor queue, and released once the job is done generating (i.e., enqueueing) all of its pages to its destination VOQ. If an incoming job's VOQ is already active (for a prior job), the incoming job is stalled at the head of the descriptor queue.

  The VOQ-Active lock ensures that sequential RDMA jobs to the same destination are transmitted in order. This is needed for efficiency – a large RDMA transfer may be divided up into multiple smaller RDMA jobs, and the VOQ-Active lock ensures that the later jobs do not compete with earlier jobs for transfer bandwidth. Similarly, sequential Datagrams and outbound messages to the same VOQ are forced to stay in order using this VOQ-Active lock mechanism. This is essential for Datagrams to be transmitted in order. (Ordering of multiple RDMA jobs for the same VOQ is not strictly necessary because the order is implied by the Tx/Rx Job IDs, and multiple outstanding RDMA jobs for the same VOQ would necessarily have unique Tx/Rx Job IDs per job)
- Once it has acquired the VOQ-Active lock, an RDMA job is entered into the **Job Table** using the Tx Job ID as the Job Table index. The Job state is set to a **CTS Wait** state, during which it will wait for Clear-to-Send (CTS) message to arrive from the receiver. In a typical case, the CTS message will already have been received before the Tx Job arrived, and the Job Table entries allow for CTS to come before or after a TX job is established.

  A CTS wait timer is also started for the job. CTS wait timeout should cover sufficient time for the Rx side to have its own timeout waiting for expected data to arrive, so that Rx may re-send CTS if no data has been seen from Tx. So, the Tx CTS timeout is "generous". If the wait timer expires without a received CTS message, this indicates a likely failure of the Rx partner, and the job may be placed in a fail-state (described below).
- Once CTS is received (or in cases where CTS has already been received), the job then allocates a Page Generator (PGen) channel. Multiple jobs may be pending to allocate a PGen channel, so the Job Manager uses a simple round-robin policy to select the next job (from the Job Table) for PGen allocation.
- The Job then enters a **Tx Active** state waiting for all pages to be transmitted. During this phase, the Page Generator will divide the job into pages, enqueue all those pages to the fabric queuing system. Once all pages have been enqueued to the VOQ, the PGen channel is deallocated and the VOQ-Active lock is released. Subsequently, each of the enqueued pages will be dequeued for transmission.

  A Tx-Active wait timer is set during this state, waiting for all pages to be fully dequeued and transmitted to the fabric. If the Tx-Active timer expires, this means that the job's VOQ was not making forward progress, which is a possibility in some fabric failure cases. The VOQ may need to be flushed, and the job placed in the fail-state.

- Once all pages have been transmitted, the job enters an **ACK-Wait** state waiting for an ACK or NACK message from the Rx chip. It also starts another timer to wait for this ACK/NACK handshake.

- In case of successful completion (with ACK received), this triggers the job to advertise its *notify_list* to the barrier table(s). Since multiple jobs may complete in quick succession, there may be multiple jobs which concurrently need to send notifications, so this is managed by a process which selects jobs (from the Job Table) on a round-robin basis.
    - Note: completion notifications are sent from the Tx Job Manager (rather than from the Trip Manager) since jobs may complete out of order.
    - Note: the notify_list will effectively signal that the Job Table entry (and its Tx Job ID) are free to re-use, by relaxing the barrier for a job waiting to re-use the same Tx Job ID. This could create a race condition for re-loading this Job Table entry if it is still busy with sending multiple notifications. This can be resolved with a decoupling FIFO which offloads the entire notify_list from the Job Table entry.

In case of a wait-timeout or received NACK message, the job is placed into a **Retry** state. If retry has already failed (or is disabled), the job is placed into the failed state and flushed as described below.

- The retry causes the job to (re) allocate a channel in the PGen, which causes the entire job to be retransmitted. Retry jobs have higher priority than new jobs from the same descriptor queue. Also, Retry Jobs do not need to acquire the VOQ-active lock. (As noted above, the VOQ active lock is provided for ordering *efficiency* in the case of multiple sequential jobs for the same VOQ. In case of retry, we waive this ordering efficiency requirement and allow any retried jobs to proceed concurrently.

  Once the retry-job has been entered in PGen, the page-generation timer is started again, and the process repeats once. Retry jobs do not need to wait for CTS-received since that has already been satisfied.

10.7.1.2 Job-Failed state

Although hardware provides reliable RDMA mechanisms, certain hardware failures can still result in the inability to complete an RDMA job or a set of jobs. An unrecoverable RDMA (job) failure may manifest itself in one of several ways:

- CTS-wait timeout, indicating that the Rx chip was unable to send a CTS message,

- Tx-wait timeout, indicating that the job’s data could not be successfully transmitted to the Rx chip.

- Multiple NACKs (i.e., a NACK on a retried job transmission).

- ACK wait timeout on a retried job.

In these circumstances, we need to terminate the job, clean up for the failure to complete the data transfer, and return the job to idle state so new RDMA work can subsequently be dispatched. The job is set to a (sticky) **Job Failed** state, an interrupt is set, and the associated Job-ID is recorded. In case multiple RDMA jobs have failed, hardware provides a software-visible vector register of all jobs in the Failed state. Software can read this register to find all the failed jobs, and write the register to clear the sticky failed state and allow the jobs to proceed. Once free, these (failed) RDMA jobs may go directly to the Idle state.

Prior to resetting the failed job(s), software will need to initiate a **Trip Flush** operation to clear out pending work which may have been dependent on the failed job(s). See section 11.8 for more information on this process.

10.7.1.3 Tx Job Manager operation for Datagram service

For Datagrams, the job consists of a single page without reliable-transport. This does not use a Job Table entry. Once the job has acquired the VOQ-Active lock, the Tx Job Manager simply forwards the “job” from the descriptor to the PageGenerator and is (nearly) done. When the page data is subsequently read from AMEM and transmitted to the fabric, the TxQ head-pointer is incremented (by hardware), effectively freeing the buffers used by that descriptor. The TxQ ring buffer mechanism is described in section 11.5.2.1 below.

10.7.1.4 Datagram TxQ completion ordering

Datagram Tx Jobs from the same TxQ may be sent to different VOQs, and can be completed out of (TxQ) order. However, the TxQ mechanism wants to advance the queue head-pointers in FIFO order to allow software to re-use the associated packet buffers. Tx Job Manager provides a reorder mechanism for this. This can keep track of up to 64 outstanding datagram packets (per TxQ) from the time they are entered into the Page Generator (PGen) until the TxBuffer has received the data from AMEM. As each Datagram job is admitted, Tx Job Manager assigns the next available **TxQ Reorder ID** from the pool of 64 (per DG TxQ) and relays this to PGen which records the TxQ Reorder ID (and also the associated TxQ number) in the QPage, which is then enqueued to the VOQ.

When the page is subsequently dequeued for transmission, the TxQ ID & TxQ Reorder ID are retrieved from the QPage database along with the AMEM source address, etc. Once each Tx Datagram packet has been completely fetched from AMEM into the TxBuffer, the TxQ ID / TxQ Reorder ID are sent back to Tx Job Manager, which sets the “done” bit for that reorder context.

When the head entry in the reorder structure is “done” (per TxQ), Tx Job Manager can signal the TxQ to advance the associated (software-visible) head pointer. This effectively signals to software that the TxQ entry, and its associated AMEM buffer resources, are free to re-use.

Note that if the reorder buffer is full, even if only one packet is still outstanding, this will block processing of additional descriptors from that TxQ.

10.7.1.5 Tx Job Manager operation for Outbound Messages

Messages (CTS, ACK or NACK) generated from the Rx Job Manager pass through the Tx Job Manager in order to use the page-transmission facilities. These messages also acquire their respective VOQ-active lock, then allocate a PGen channel for the (single-page) message. Tx Job Manager has no other specific processing for Rx-generated messages.

10.7.2 Tx Job Table

There is one Tx Job Table associated with each (source) UIE cluster. Each Tx Job Table supports up to 256 concurrently active RDMA jobs, and is indexed by Tx Job ID (**TJID**), which is specified with the job descriptors set up by software. The Tx Job Table contents are shown in the table below.

Table 10-1: Tx Job Table Entry Format

Field	Width	Description
RJID	8	Rx Job ID to link to the remote Rx Job Table.
Dest VOQ	13	Target VOQ, identifies target chip, cluster and traffic class
AMEM Base Addr	20	AMEM base address in source cluster. Word (128-byte) aligned.
Total Words	17	RDMA transfer length, in 128-Byte words. Allows up to 8MB job size
Page Size Words	7	Number of 128-Byte words per page transmitted over the fabric. All transmitted pages, except possibly the last page, will use this size. **Note: (PageSizeWords*128 + 16) must be evenly divisible by 176 for the Juniper Fabric**
Notify_list	8*12	Barrier completion-notification list (copied from job descriptor)
The following fields define the overall state of the job		
Epoch	1	Job ID "Epoch" is incremented each time a job is invoked for this TJID, to de-alias "current" and "future" jobs using the same Job Table index.
CTS received	1	Rx Job is valid (CTS has been received).
Retry	1	Indicates the job is being retried (effectively a 1-bit retry count)
State	~4	FSM which tracks the state of the job
Timer timestamp	~10	Timestamp for comparison to a hardware timer, to detect job phase timeout

10.7.3 Tx Page Generator

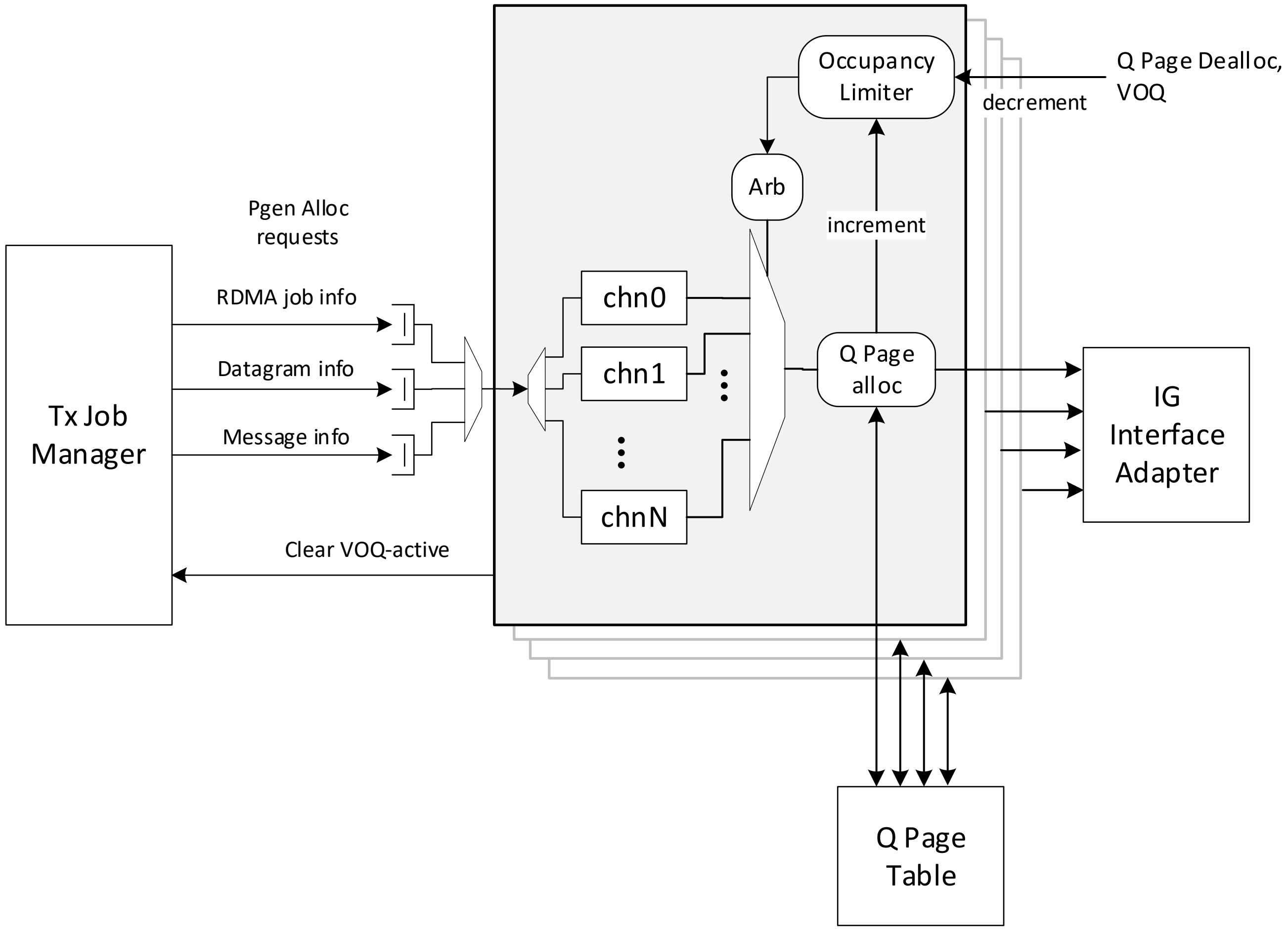


Figure 10-5: Tx Page Generator

- One Page Generator (PGen) instance per UIE cluster, associated with the cluster’s Job Manager
- Channelized, page-at-a-time generator. Channels are assigned dynamically when loaded by the Tx Job Manager. Once a channel is loaded, it generates all of the pages for one job. A PGen channel also generates the (single) page for datagrams and hardware messages.
- PGen provides channel allocation limits for each work type (RDMA jobs, datagram pages, or hardware messages), with a single-bucket, multi-limit model. Critically, PGen can be configured to reserve at least one channel for transmitting messages. This ensures job completion and avoids possible deadlock scenarios.
- PGen generates pages based on the job’s configured page size and total size to be transferred. RDMA job sizes are always in units of 128B “words”. PGen Keeps track of the number of 128B words sent so far (compared to total job size) and identifies the last page and number of words

in the last page, which may be less than the configured page size. For Datagrams, there is a single page per "job", and this is not required to be a perfect multiple of 128B.

For each page to be generated, the channels compete with High/low Priority + Round-robin arbitration across channels (in each cluster) to enter the page-generation pipeline. Priority (high or low) is configured per VOQ, and assigned to each channel based on the VOQ for its job. The arbitration performs a round-robin selection among all high-priority channels (if any are pending) else it performs a round-robin selection among the low-priority channels.
The winning channel generates the next page for its job. This allocates a **QPage** entry, then relays the QPage index as the page-handle to be enqueued, to the Ingress Group (IG) interface adapter.

Note that multiple clusters' Page Generators will be competing for access to the QPage table and IG, which will also implement cross-cluster round-robin arbitration.

- PGen includes an Occupancy Limiter to keep track of outstanding pages per VOQ. This senses and reacts to congestion in fabric-facing VOQs

10.7.3.1 Tx Page Occupancy Limiter

The Tx Occupancy Limiter regulates outstanding pages per VOQ to avoid a congested fabric-facing VOQ from growing without bound. This is the primary congestion control method used for RDMA / Datagram traffic management across the fabric. The outstanding page count is incremented when the QPage is generated and enqueued to the VOQ. The count is decremented when the QPage is dequeued from the VOQ and the QPage is retired. Pyxis does not allow (nor require) extremely deep fabric-facing VOQs, and in case of congestion experienced by one VOQ, one job could consume all of the QPage entries available if there were no occupancy limits.

The occupancy limiter is closely aligned with the Tx Page Generator. When an occupancy limit is reached, this inhibits further page generation for the VOQ, even if the Page Generator is in the middle of a large job for that VOQ.

Occupancy is tracked (and limited) at multiple levels:

1. Each VOQ is configured with one of a small number of occupancy-management ***profiles*** which specify the queue limit. There are **eight** total VOQ occupancy profiles, which effectively allows for eight different VOQ rates. The outstanding page count required for a VOQ is roughly equal to the `fabric_round_trip_time * VOQ_desired_bandwidth / average_page_size`

   Each VOQ also maintains its outstanding QPage count. When the count reaches the profile limit, the Page Generator is inhibited from generating additional pages for that VOQ. This count is maintained separately for each source cluster (i.e., per Page Generator instance)

   VOQ occupancy thresholds use separate (per-profile) "XOFF" and "XON" thresholds with hysteresis A PGen channel (and all channels which are generating pages for that VOQ) are inhibited from generating additional pages when the VOQ occupancy exceeds the XOFF threshold. They are not re-enabled until the occupancy drops below the XON threshold. The XOFF threshold is normally configured to be slightly higher than the XON. The intention is to allow multiple PGen channels which have the same VOQ to have an opportunity to generate another page when the VOQ is near the congestion point. Note that multiple channels for one VOQ occurs primarily due to job retry, where multiple jobs may be actively generating traffic for the same VOQ.

2. Each VOQ is also assigned to **high or low priority** occupancy-tracking ***Groups*** within each source cluster. When the aggregate QPage count for a group exceeds its limit, all VOQs which are members of that group are inhibited from generating additional pages.

   The grouping scheme allows for the per-VOQ limits to oversubscribe the total QPage resources, by limiting the *aggregate* QPage resources used by lower-priority queues to preserve some Qpages for higher priority queues. So, in case of congestion, the higher priority queues will still make forward progress.

   The occupancy-tracking groups are managed independently for each source cluster.

3. Total QPage occupancy count per source cluster is limited to a configurable maximum value. This ensures that each cluster's QPage utilization is limited to a (configurable) fair share of the total Qpages.

10.7.4 Tx Ingress Group

The Tx Ingress Group (Tx IG) is an interface adapter which emulates the behavior of the Juniper BX chip's IngressGroup module, to enqueue generated pages on to the VOQ subsystem.

- One instance total
- Receives QPage index from the PGen blocks, with each page-enqueue request. Multiplexes these and relays this as the "packet handle" sent with the enqueue command to Juniper IP ("VOQM" block).
- Must re-map the Pyxis 13-bit VOQ to 16-bit Juniper VOQ, and conform to the Juniper VOQ numbering scheme, which is constructed as `Juniper_VOQ = {VPFE[10:0], VPFE_OQ[4:0]}`. The specific mapping is defined in "Fabric VOQ mapping", section 10.2.

10.7.5 Tx CLIST

The Tx CLIST block emulates the Juniper BX chip's "Cell List Manager" behavior, to accept dequeue-requests from Juniper IP ("FO" block), and returns a list of (artificial) cell-pointers, corresponding to the number of Juniper-fabric cells expected for the page.

- One instance per FO pipe, total 4.
- Receives "packet-handle" with the Dequeue request from FO, which is the QPage index.
- The dequeue-request triggers deallocation of the QPage, which kicks off the data transfer from AMEM toward the fabric. Four Tx CLIST instances arbitrate for access to the (common) QPage Table

- Dequeue-request also triggers TX CLIST to return artificial Cell Pointers ("cptrs") to FO. One cptr corresponds to (up to) 176 bytes. This is needed to match the Juniper IP protocols. The artificial pointers may be constructed from an incrementing page sequence number and the cell index in the page. This is for debug visibility since the cptr *values* are not actually used.

10.7.6 Tx QPage Table

The QPage table is the database of "Queue Page" objects, i.e., metadata for each page enqueued to a VOQ.

- Free-pool based structure for tracking outstanding pages in the VOQ subsystem.
- Approximately 1024 total pages. For bulk transfer page size of 16KB, and total queue occupancy of ~10us, for 400GB/s this requires ~256 pages, so this is capacity allows for multiple "congested" VOQs and many small pages (e.g., for messaging)
- Entries include all information needed to construct the page, including:
    - AMEM source address & cluster
    - Page size in AMEM words
    - TJID, RJID, Epoch, last-page fields to be added to page header.
    - Datagram RxQ Index to be added to page header
    - TxQ Reorder pointer for Datagrams (to mark the descriptor as complete when the page is eventually transmitted)
    - VOQ number for occupancy tracking
- Allocates entries on demand from Page Generator, and returns the allocated QPage Index.
- Deallocates entries on demand from TX CLIST (when the page is dequeued), and creates an equivalent transmit page context

10.7.7 Tx Page-transfer Contexts and TxDMA Read from AMEM

- One instance per 100GB/s pipe, total 4
- Receives the page information read from the QPage entry, and creates a **Tx Page Context** for each removed QPage entry. The Tx Page contexts are managed as a FIFO of pages (per 100GB/s FO pipe). Each Tx Page Context includes all information needed to construct the page, including:
    - AMEM source address & cluster
    - Page size
    - Starting Txbuffer entry address

  - Additional metadata needed for constructing the page headers (supplied to header-encapsulation module)
  - For Datagrams: the TxQ Reorder pointer (to mark the descriptor as complete when the page is eventually transmitted)
- Tx Page Context creation also allocates a contiguous block of 128B entries in the TxBuffer to receive the entire page from AMEM. If the Tx page contexts are all in use or not enough TxBuffer slots are available for the size of the page, the page transfer is stalled.
- Allocating a TxPage (and associated TxBuf space) initiates a TxDMA Read operation to transfer the requested memory from AMEM to TxBuf. The Tx RDMA Read engine is part of the Core Switch logic. There is one Tx RDMA Read engine for each AMEM cluster (to source the data) and any FO Pipe can send a DMA Page transfer request to any cluster's Tx RDMA engine. Each page will be resident in only one cluster's AMEM.
- The selected Tx RDMA Read engine generates AMEM read requests which are sprayed over the four Core Switch planes, and steers the data to the requesting FO pipe and TxBuffer slot for each flit. Note that data may be returned out of order from the Core Switch.
- As each flit is reordered and popped from the TxBuf, this is reported to the Tx Page context. When the last flit is popped, the Tx Page context may be retired. If this is the last (or only) page in a job, this is also reported to the Job Table or Datagram TxQ (via the Tx Job Manager) to update state

10.7.8 Tx Buffer

- One instance per 100GB/s pipe.
- TxBuffer receives and reorders data (flits) from AMEM (via the CoreSwitch), and provides buffering to decouple this from the Tx CBUF adapter module and FO data interface.
- TxBuffer capacity covers approximately 2x the AMEM read RTT + one maximum-sized page. Storage is allocated & deallocated in contiguous blocks, so TxBuf operates as a circular buffer.
- Maintains received-valid bits for each word in the buffer, and keeps a global reorder pointer, since data may be received from CoreSwitch out of order (due to the multiple-planes design of Core Switch). TxBuf presents in-order received data to the Tx CBUF.
- When Tx CBUF accepts data, the TxBuf entry (128B) is popped and becomes available to re-allocate.
- Each TxBuf instance can read and write one 128-byte word per cycle. Since the CoreSwitch may present up to 4 words/cycle, there is another buffered switch between CoreSwitch and TxBuf to reduce this to the (sustainable) one 128-byte word/cycle/TxBuf.

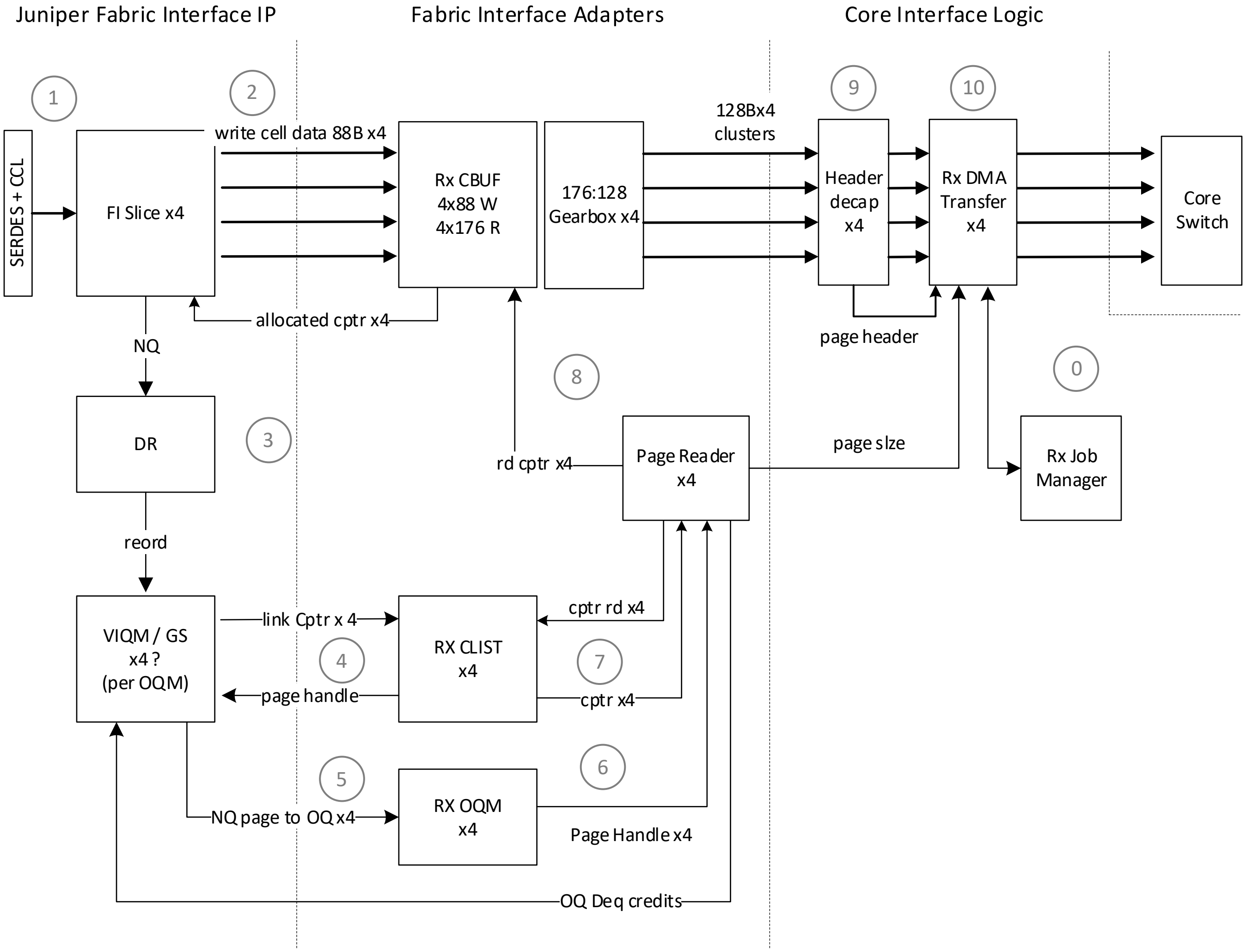
10.7.9 Tx Page Header Encapsulation

- One instance per 100GB/s pipe
- Performs page-header encapsulation. Page header is 16 bytes, prepended to each page.
- Receives metadata to build the page header from the Tx Page context, in page-FIFO order.
- Constructs the page header, including CRC32 as an integrity check for the eventual receiver. See section 10.6 for page header contents.
- Receives 128B-flits from the TX Buffer (in the same order as the headers), and inserts the constructed page header at the start of each page, and forwards the data to the CBUF. This requires a simple residue buffer to keep the excess 16B from each flit. At End of Page, this can emit one final flit with remaining bytes.
- **For hardware-generated RDMA handshaking messages,** there is no data from AMEM, only the header. For this case the packet/page size will be just 16 Bytes at this point. The hardware can pad this up to 64B before relaying this to TxCBUF, and indicate the size as 64B .

10.7.10 Tx CBUF

- One instance per 100GB/s pipe
- Models the Juniper "Cell Buffer" behavior to accept cell-pointer read requests, and returns data to Juniper IP. One instance per 100GB/s "FO Pipe" in the Juniper IP.
- Receives buffered data from Header Encap in FIFO order. The page order is determined from the FO-to-CLIST Dequeue request order.
- Performs 128byte to 88 Byte gearbox to conform to Juniper IP datapath requirements.

10.8 Rx Fabric Adapter



The operational flow through the Rx Fabric Adapter is as follows:

[0] Before any RDMAs can proceed, using a “solicited” transfer model, the RDMA-Rx contexts must be set up. AMEM buffers are allocated, and an RX Job descriptor is enqueued to a descriptor queue. Each Job may represent as little as one flit (128B), or bulk transfer of 10’s of KBytes up to MBytes. The Job descriptor provides the destination address, expected length, Tx and Rx Job IDs used to identify the connection, as well as barrier controls (wait_barrier_id and notify_list).

Once the barriers are cleared, an Rx Job Table entry is created for the indicated Rx Job-ID and destination cluster. This also triggers hardware to send a ClearToSend message to the source chip to enable the RDMA transfer.

[1] Subsequently, the source chip will begin transmitting cellified data. Cells may show up at any SERDES and may arrive out of order. The Juniper datapath divides the SERDES into blocks of 9, supporting about 100GBytes/s fabric throughput per block. Each block is attached to an instance of the Juniper Fabric Input (FI) block. Pyxis requires 4 instances of FI.

[2] FI receives data cells and immediately writes these to the Rx CBUF module. This is a “fabric termination” buffer with substantial size, at least equivalent to the Fabric Round-trip-time * bandwidth with generous excess capacity.

Rx CBUF must accept 88B per cycle from each of 4 FI blocks continually. Rx CBUF uses a freepool-based, multi-bank structure, which can allocate 4 x 176 Bytes at least once per 2 cycles, and can write 4 x 88B/cycle. CBUF returns the allocated CBUF pointers to the FI blocks

[3] Fabric cells may arrive out of order, so each FI takes the received CBUF-cell pointers, combined with ordering information, and hands this to the (Juniper) Data Reorder (DR) block. DR keeps a reorder context for each source/destination chip-pair, and hands the properly-ordered pointers to the Juniper VIQ Manager (VIQM) block.

[4] VIQM then assembles the received pages. There is one VIQM “slice” per Pyxis Cluster, responsible for 1/4th of the fabric bandwidth. The received Output Queue (OQ) number indicates which VIQM slice, and therefore which cluster for each received page.

Each VIQM slice signals the Rx CLIST block to stitch cells into a linked-list per page. The cells of each page may have been received on any FI Slice - note that FI slices have affinity with fabric links, but VIQM slices have affinity with OQs (=Clusters), and Rx CLIST blocks are per-cluster as well. Each VIQM slice can signal one CLIST pointer per cycle, along with Start-of-page, end-of-page, and reassembly context (= source chip). Rx CLIST is expected to keep one open reassembly context per source Pyxis chip, and returns a Page Handle (which identifies the stitched-together list) to VIQM when the end-of-page cell is received.

[5] Once a full page has been stitched, VIQM will enqueue the page to the appropriate Output Queue (OQ). In our application, we require four OQs per cluster, to support two classes of service for each of RDMA and Datagram services. The Rx OQ Manager (OQM) receives the enqueue command including OQ number, page size, and the packet handle from VIQM. There is one Rx OQM instance per cluster.

[6] Rx OQM maintains queues which keep track of received pages with a simple priority scheduler to dequeue the head page from one of the 4 queues per cluster, and presents the dequeued page information to per-cluster Page Reader threads. The OQs must support at least the fabric-RTT’s-worth of pages per cluster, generously sized up to allow for multiple nonempty queues per cluster and a combination of small and large pages.

[7] Rx Page Reader signals the Rx CLIST block to unstitch the pages from the linked list of cells. CLIST returns a stream of individual CBUF cell pointers to the Page Reader

[8] Page Reader then issues CBUF reads to fetch the cell data, and this must support four active Page-Reader threads competing for access to the multi-bank Rx CBUF block. Data read from CBUF is gear-boxed from 176B cells to 128B flits, for each of the page-reader threads.

CBUF cells are freed on read, with the cell pointer returned to the free-pool. As each CBUF cell is freed, the Page Reader signals this to the VIQM (and associate Grant Scheduler). This is how the GS detects whether it may issue more grants, to avoid overrunning the OQs.

[9] After the gearbox, the Header Decap block extracts the page header which was inserted on the Tx side. This header indicates the RDMA Job ID which is provided to the Rx RDMA Write engine.

[10] RxDMA Write engine sends the Job ID to the Rx Job Manager, to index the RDMA Job Tables to validate the received page. Rx Job Manager keeps track of the state of the RDMA job, including the total number of received bytes, and returns the destination AMEM (base) address for the page to RxDMA.

RxDMA Write then sends each flit for the page to the Core Switch, providing the appropriate AMEM address with each flit. Core Switch logic sprays the flit write-requests from each RxDMA engine over the four core-switch planes, and routes the data to the target AMEM (cluster).

RxDMA also signals to the Rx Job Manager when the last page of a job has been fully written to AMEM. This signals Rx Job Manager that the RDMA job is complete, and Rx Job Manager may then relax hardware barriers to enable other engines to use the received data.

The Rx Fabric Adapter modules are described in the following sub-sections

10.8.1 Rx Job Manager

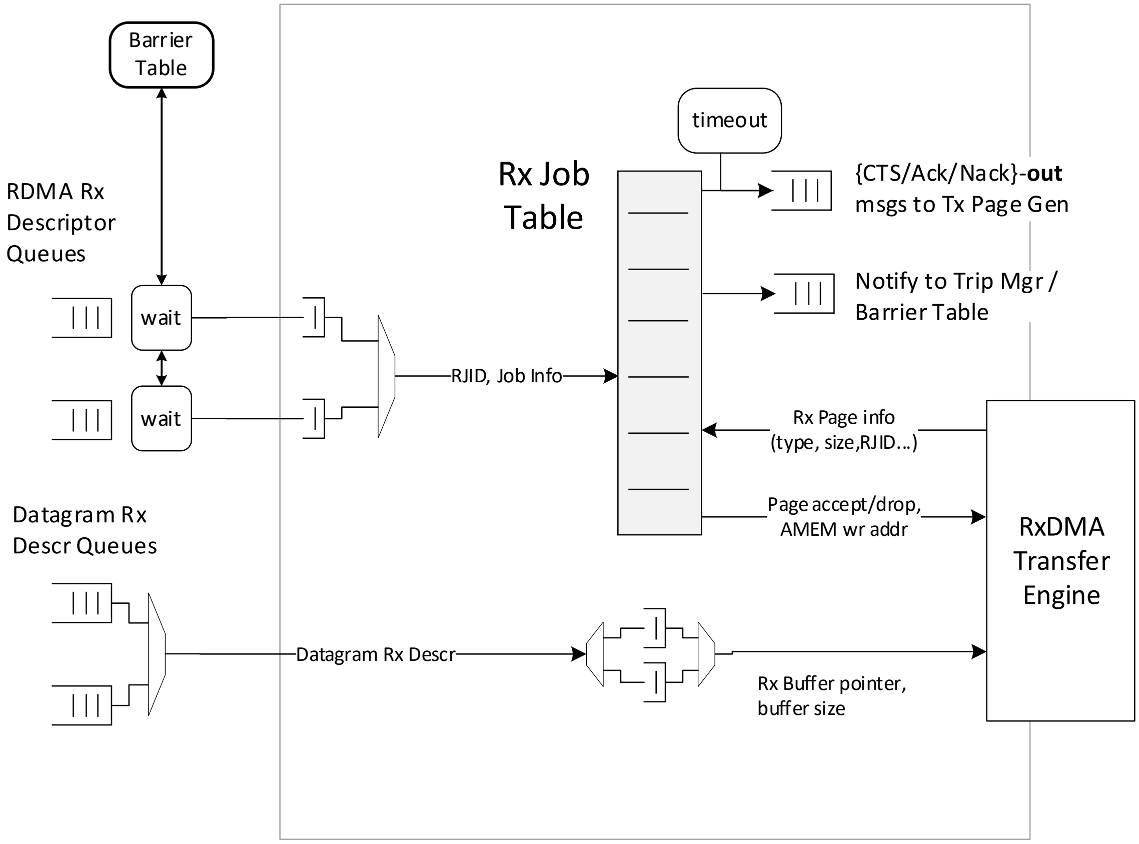

Barrier Table
RDMA Rx Descriptor Queues
wait
wait
Datagram Rx Descr Queues
Rx Job Table
timeout
{CTS/Ack/Nack}-**out** msgs to Tx Page Gen
Notify to Trip Mgr / Barrier Table
RJID, Job Info
Rx Page info (type, size,RJID…)
Page accept/drop, AMEM wr addr
RxDMA Transfer Engine
Datagram Rx Descr
Rx Buffer pointer, buffer size

Figure 10-6: Rx Job Manager

There is one Rx Job Manager instance per UIE cluster, for RDMA and Datagram data received to that cluster's AMEM.

10.8.1.1 Rx Job Manager operation for RDMA

- Rx job descriptors are enqueued to descriptor queues. There are 2 RDMA Rx descriptor queues (per cluster). The head entries of the queues may undergo barrier-waits (in the Trip Manager), then the entries are moved into shallow FIFOs at the entrance to the Rx Job Manager. The barriers can be used to ensure that the RDMA-destination memory locations are ready to receive data, and also ensures that the Rx Job Table entry is not currently in use.
- The first step in job service is to write the Rx Job Table entry (for RDMA, but not for Datagram). This will place the job in a **CTS Send** state, and trigger the Job to send "Clear To Send" (**CTS**) message to the remote (Tx) Pyxis chip. The message is actually sent by the local Fabric-Tx logic, so there is a FIFO which links the Rx CTS-out message from Fabric_Rx to Fabric_Tx.

  Job Manager sends the CTS-out message based on a state in the Job Table, with a process which selects the next job for CTS-out service with a round-robin selection.
- Once the job is active (and CTS message has been sent), the job enters an **RX Data Wait** state waiting for data to arrive from the Tx side. A wait timer is also set. If no data is received within a timeout period, this may indicate a lost CTS message, so the Job Manager would re-send the CTS message once, with a retry flag marking on the CTS message. If the wait timer expires a second time, this indicates a likely failure of the Tx partner, and the job may be placed in a fail-state (described below)
- Once (any) data is received, the job is moved to **Rx Data Active** state, and a job-completion timer is started to monitor for complete receipt of the expected data for the job.
- As data is received, page-by-page, the Rx fabric adapter modules (specifically: the RxDMA transfer engine) will query the Rx Job Manager / Job Table to validate the received page and determine the target AMEM address for the received data. The Job Table provides the expected Tx Job ID and Job-ID Epoch corresponding to each Rx Job ID, AMEM base address and expected total transfer size, as well as the state of the job including the next expected page-sequence number. From this information, the received page is validated and the next AMEM address is calculated.
- When all the pages have been successfully received, the job is complete, which triggers sending an ACK-out message to the Tx partner. The message is actually sent by the local Fabric_Tx logic, so there is a FIFO which links the Ack-out message from Fabric_Rx to Fabric_Tx.
- Job completion also triggers the Job Manager to advertise its *notify_list* to the barrier table(s). In case multiple jobs have concurrent barrier-notifications pending, a process selects the next job to perform the notifications in a round-robin manner
  - Note: completion notifications are sent from the Rx Job Manager (rather than from the Trip Manager) since jobs may complete out of order.

   - Note: the notify_list will effectively signal that the Job Table entry (and its Rx Job ID) are free to re-use, by relaxing the barrier for a job waiting to re-use the same Rx Job ID. This could create a race condition for re-loading this Job Table entry if it is still busy with sending multiple notifications. This can be resolved with a decoupling FIFO which offloads the entire notify_list from the Job Table entry.

- If an error occurs while receiving the data, such as a lost page, or a page marked with error by the Juniper Fabric IP blocks, this triggers sending a NACK message to the Tx partner at the end of the job. This will cause the Tx chip to retry the **entire** job. The job state (Current page sequence number, AMEM write-data address offset) is also reset to start the entire job over, and an "is retry" state bit is set for the job. If a job in the retry state again has an error, we will place the job in a failed state. There is at most one retry allowed per job.
- If the job-completion timer expires, this may also indicate a failure of the Tx partner since the Tx chip would have retried even if the ACK or NACK message was lost. The job is then placed in a failed state. Note that job-completion timer is generous to allow for the Tx side's retry timer to expire and retry to be attempted before the Rx job-completion timer would expire.

10.8.1.2 Job-Failed state

Although hardware provides reliable RDMA mechanisms, certain hardware failures can still result in the inability to complete an RDMA job or a set of jobs. An unrecoverable RDMA (job) failure may manifest itself in one of several ways:

- 2nd Rx Data wait timeout when a retried CTS message has already been sent. This may indicate that the CTS messages cannot be delivered, or the Tx chip is unable to send any data.
- 2nd job data transfer error (such as a page sequence error), when the first job transfer failed, such as with a page sequence error or job completion timeout. This may indicate a persistent problem with data transfer from the Tx chip.
- 2nd job-completion-wait timeout when the first job transfer failed (as above). This may also indicate a persistent problem with data transfer from the Tx chip.

In these circumstances, we need to terminate the job, clean up for the failure to complete the data transfer, and return the job to idle state so new RDMA work can subsequently be dispatched. The job is set to a (sticky) **Job Failed** state, an interrupt is set, and the associated Job-ID is recorded. In case multiple RDMA jobs have failed, hardware provides a software-visible vector register of all jobs in the Failed state. Software can read this register to find all the failed jobs, and write the register to clear the sticky failed state and allow the jobs to proceed. Once free, these (failed) RDMA jobs may go directly to the Idle state.

Prior to resetting the failed job(s), software will need to initiate a **Trip Flush** operation to clear out pending work which may have been dependent on the failed job(s). See section 11.8 for more information on this process.

The RDMA protocol allows for retry in case of a lost or errored page, which is detected by a page sequence number mismatch in the received page-stream (for a given job), or by a timeout. However, in some cases the retry will also fail. If this occurs, Rx Job Manager concludes that the RDMA job will not be able to successfully complete, and the job is moved to the "Job Failed" state. This may be triggered by a 2nd timeout of the initial-data-receive timer (indicating lost CTS) or 2nd timeout of the job-completion timer, or a retried job with a receive-error.

When a job is set to Job Failed state, and interrupt is set and the associated Job-ID is recorded. In case multiple RDMA jobs have failed, hardware provides a software-visible vector register of all jobs in the Failed state. Software can read this register to find all the failed jobs, and write the register to clear the sticky failed state and allow the jobs to proceed. Once free, these (failed) RDMA jobs may go directly to the Idle state.

Software will need to initiate a Trip Flush operation to clear out pending work which may have been dependent on this (failed) job. See section 11.8 for more information on this process.

10.8.1.3 Rx Job Manager Operation for Datagram service

For Rx Datagrams, each "job" consists of a single page without reliable-transport. This does not use the hardware barriers mechanism, and does not use a Job Table entry. The Job Manager simply waits for the next Rx Datagram page to arrive, and retrieves the receive AMEM buffer address from the head of the RxQ indicated with the received page (in the page header).

There are at least two RxQs for received datagrams, to provide two classes of service. High-priority RxQs may also use smaller AMEM buffer sizes, while low priority RxQ may use larger buffers for bulk data transfer.

When the data has been successfully written to AMEM, the RxQ head pointer is incremented (by hardware) effectively making the queue entry (page) visible to the CPU. The RxQ ring buffer mechanism is described in section 11.5.2.2 below.

If the RxQ is empty when a page is received, (specifically, if there are no prefetched RxQ descriptors available), the received page is **dropped**. This condition occurs when there are no AMEM buffers available for that RxQ. The page may also be dropped if the page size is larger than the configured buffer size for the RxQ. Hardware maintains counts of accepted and dropped packets/pages per RxQ.

10.8.2 Rx Job Table

There is one Rx Job Table associated with each UIE cluster. Each Rx Job Table supports up to 256 concurrently active RDMA jobs, and is indexed by Rx Job ID (**RJID**), which is specified with the job descriptors set up by software. The Rx Job Table contents are shown in the following table.

Table 10-2: Rx Job Table Entry Format

Field	Width	Description
TJID	8	Tx Job ID to link to the remote Tx Job Table
AMEM Base Addr	20	AMEM base address in target cluster. Word (128-byte) aligned.
TotalWords	17	RDMA transfer length, in 128-Byte words. Allows up to 8MB job size
Msg VOQ	13	VOQ pointing to the Tx chip, for sending CTS and ACK/NACK messages
Notify_list	8*12	Barrier completion-notification list (copied from job descriptor)
The following fields define the overall state of the job		
Epoch	1	Job ID "Epoch" is incremented each time a job is invoked for this RJID, to de-alias "current" and "past" jobs using the same Job Table index.
State	~4	FSM which track the overall state of the job
WordsReceived	17	State which tracks the number of 128-B words received so far.
PageSeqNum	~10	Next expected page sequence number
Retry	1	Indicates the job phase is being retried (effectively a 1-bit retry count)
Timer timestamp	~10	Timestamp for comparison to a hardware timer, to detect job phase timeout

10.8.3 Rx CBUF

- One instance shared by all FI slices and all Rx clusters.
- Receives data from each of four Juniper Fabric-In (FI) blocks. Each FI can send 88 bytes/cycle which must be stored in a freepool-based Cell Buffer structure. Each CBUF "cell" is up to 176 bytes (two 88B flits) but not all cells will be full. RX CBUF returns the allocated cell pointers (cptrs) at up to one cptr per cycle per FI block. The cells will be stitched together later after reordering.
- The cells are subsequently read out once the full page has been assembled and ready to be relayed to AMEM. RxCBUF maintains four "gearbox" contexts (one per Cluster) to reshape the data from 176B Juniper Cells to 128B Pyxis databus flits.
- The last and next-to-last cells in each page may contain fewer than 176B, with granularity of 16B "chunks" since the Juniper fabric has variable cell sizes with a minimum of 96B (6 chunks) and max of 176B (11 chunks). The Juniper fabric IP will automatically allocate chunks to the last two cells of a page, if necessary, to prevent any cell from having fewer than 6 chunks. Therefore, if the total page chunk count modulo-11 < 6, this redistribution will happen so that the last cell has exactly 6 chunks and next-to-last cell can have fewer than 11. This creates a "hole" in the stored data which RxCBUF must compress-out when reading. The Rx PageReader block will determine the valid number of chunks in each cell and indicate this to RxCBUF

- Total Rx CBUF size = 24K cells, which supports 4.3x10^6 bytes, or about 10.8usec of bandwidth-delay buffering at 400GB/s.

10.8.4 Rx CLIST

- One instance per Rx Cluster
- Maintains linked-lists of CBUF cells for each received page.
- After reordering cells, the cell-pointers are sent from VIQM to CLIST to be stitched per page. There may be one partially-stitched page from each source Pyxis chip (or "PFE" in Juniper fabric terminology), so total of up to 144 page contexts need to be maintained. VIQM will indicate start-of-page and end-of-page with each cell pointer (and the context number). CLIST will keep track of the current page's linked list, and return a "page handle" to VIQM after receiving the End of Page cell pointer. For a single-cell page, the handle is the Rx CBUF cell pointer. For larger pages, this is a pointer to a linked-list structure, which may be indexed by head cell pointer in the page.
- Subsequently, the Page Reader will send the page handle, and the number of 16-byte chunks in the page, to CLIST to unstitch the cells and return the individual cell pointers to Page Reader. CLIST will determine the number of 176-byte cells corresponding to the chunk-count for the page. CLIST also determines the valid number of 16-byte chunks for each cell – this is 11 chunks (176B) for all cells except possibly the last two cells of each page. The last two cells may divide the remaining chunks (in the page) so that no cell has fewer than 6 chunks. The valid number of chunks is returned to Page Reader with each cell pointer.
- Supports at least 0.5 cells/cycle for linking and 0.5 cells/cycle for unlinking, but this is bare minimum with no speedup, so the design will support burst linking (write) rate of ~1 cell/cycle and read rate of ~2cell/3cycles

10.8.5 Rx OQM

- One instance per Rx Cluster
- Maintains Output Queues (OQs) per cluster for received, assembled pages which are waiting to be moved into AMEM. A total of 4 OQs/cluster allows for two types of traffic (RDMA/Datagram) and two classes of service each.
- Receives page handles from VIQM with OQ-number and page length, and enqueues to simple FIFO queues.
- Simple priority + round-robin scheduler to dequeue the head page from one of the four queues per cluster, and presents the dequeued page information to the Rx Page Reader.
- The OQs must support enough pages for all the data in the Rx CBUF. Given ~1MB of CBUF allocated per cluster and 16Kbyte pages, the bare minimum would be 64 queued pages per cluster. To support smaller pages, this is easily upsized to 128-256 queued pages/cluster.

10.8.6 Rx Page Reader

This module assembles data for each page into a contiguous byte-stream to be transferred to AMEM. The incoming data is a (scattered) linked list of Rx CBUF cells which must be gathered.

- One instance per Rx Cluster
- Receives page-handle from Rx OQM, along with the page size in 16-Byte chunks. For a single-cell page, the handle is the Rx CBUF cellpointer. For larger pages, this is a pointer to a linked-list structure maintained by Rx CLIST.
- For each page, Page Reader issues page-read requests to CLIST, including the page size in 16-byte chunks. CLIST unstitches and returns the full list of cell pointers, including the valid number of chunks in each cell.
- Page Reader then issues a series of RxCBUF cell-read requests for the duration of the page. The valid number of chunks is sent to RxCBUF with each cell pointer. Page Reader can issue up to one 176-Byte cell-read requests per cycle, to take advantage of speedup in the Rx CBUF gearbox to supply 128-Byte flits toward AMEM.
- Page Reader also sends the page-size (in chunks) the Rx RDMA Write engine.

10.8.7 Rx Page-Header Decapsulation

- One instance per Rx Cluster
- Receives a series of 128B flits from the RxCBUF (after the 176:128 gearbox), with SOP/EOP markings to identify the start of each page.
- Extracts the 16-byte page header and re-packs the remaining data back into 128-byte flits. The header includes a CRC which is checked for integrity. Header fields are sent as metadata to the Rx RDMA transfer engine

10.8.8 Rx DMA Transfer Engine

- One instance per Rx Cluster
- Receives the page length from Page Reader, and (subsequently) the header information from Header Decap, and the series of 128B flits (post decap). The Page Type field, extracted from the page header, indicates whether the page is RDMA or Datagram, or a hardware-generated message type
- The header provides the information needed to route the received page to AMEM. This includes the page type (message, RDMA or Datagram)

  - for RDMA: header provides Rx Job ID and Tx Job ID (RJID/TJID) which are relayed to the Rx Job Manager to access the Rx Job Table.
  - For Datagram: the header includes the RxQID, which is used to retrieve an Rx Datagram (buffer) descriptor.
  - For incoming messages, the header includes the entire message.

For RDMA Data pages, processing includes the following:

- RxDMA signals the Rx Job Manager to perform integrity checks between the received page and Rx Job state.
  - If the RJID/TJID/Epoch don’t match for an active job, this may be a stale/old job and the page is dropped.
  - If the page sequencer number (from the page header) does not match the next expected value, this indicates a lost (dropped or skipped) page during the transmission, and the Rx Job state is updated to indicate the error. This can trigger a NACK message once the last page has been received.
  - Similarly, if the total byte count received for a job, after For the last page (in the job), if the page size (in 128B words) + previously received count is less than the total expected job size (in 128B words), this may indicate a lost (dropped or skipped) page during the transmission, and the Rx Job state is updated to indicate the error. This can trigger a NACK message.
  - For any page, if the page size + previously received count exceeds the total expected job size, this is a strange error indicating corruption; the page is dropped, and the Rx Job state is updated to indicate error and send a NACK
  - If the page header has a CRC error, this may indicate a corrupted page, and the RJID/TJID and other fields cannot be used. The page is dropped.
- If the integrity checks pass, Rx Job Manager computes the page starting AMEM address from the Job Table’s AMEM base address and number of words received so far; the sum of these is the starting write address for this page in AMEM…which is returned to RxDMA. Rx Job Manager updates the Rx Job Table entry for the received word count for the page, and signals Rx RDMA to transfer the page to AMEM. Otherwise, Rx RDMA is signaled to drop the page.
- Rx RDMA Transfer Engine then sends each flit to the Core switch interface to be routed to AMEM, and provides the AMEM address per flit, incrementing the address by one AMEM word with each flit. However, if Rx Job Manager signaled to drop the page, then RxDMA simply discards all of the page flits received from the header decap module
- For received hardware-generated RDMA messages conveying CTS, ACK or NACK, the entire message information is contained in the header, and no data needs to be forwarded. The page type (message or data transfer) is encoded in the page header. For these cases, the message contents are relayed directly to the Tx Job Manager (on the Tx side of the fabric adapter). Any

additional data beyond the header is considered garbage, and will be dropped by RxDMA engine.

- Outstanding writes per page are tracked using a Write-Completion Tracking ID (see section 11.7.2, DMA Write Completion Tracking). Each Rx RDMA page transfer allocates a **Tracking ID** from a set of 16 IDs. AMEM uses the Tracking ID to indicate when the writes have been fully committed to memory. If no Tracking IDs are available, Rx RDMA is stalled.
- When the last page of an RDMA job is successfully transferred to AMEM (and all pending writes are complete as indicated by the write-completion tracker) the Rx Job state is updated and will subsequently trigger Rx Job Manager to issue the completion notifications to the hardware Barrier Table(s).

For Datagram pages, processing includes the following:

- The RxQ is determined by a field in the page header (extracted by Header Decap module).
- RxDMA can do an integrity check between the received page-length (as indicated by PageReader and the length indicated in the page header). If these do not match (with appropriate chunk level granularity) the page will be dropped.
- RxDMA checks the selected RxQ's ***descriptor queue*** for the appropriate AMEM buffer pointer. RxDMA provides a shallow FIFO of prefetched descriptors per RxQ, and the Descriptor Prefetch Engine (DPE) will normally try to keep these FIFOs topped-up with available descriptors. If the FIFO is empty, the page will be dropped. If the received page is larger than the buffer size indicated in the RxQ descriptor (selected from the FIFO), the page is dropped. Otherwise, the page is transferred to AMEM as with RDMA.
- Rx RDMA Transfer Engine sends each flit to the Core switch interface to be routed to AMEM, and provides the AMEM address per flit, incrementing the address by one AMEM word with each flit.
- Outstanding writes per page are tracked using a Write-Completion Tracking ID as described above for RDMA processing.
- The received page length is also relayed to the DPE, which is responsible for managing the descriptor queues. The DPE will write the received length into the descriptor in memory, to inform the receiving CPU of the actual data length received.
- Once the transfer to AMEM is complete, RxDMA signals the DPE to increment the RxQ hardware pointer. (DPE also makes sure it has updated the page length in the descriptor). This makes the received page visible to software.

For incoming hardware generated messages, processing includes the following:

- The message type, and any associated fields (Tx JobID, Rx JobID, Epoch) are fully in the page header, so are provided from header decap module.

- These received messages are actually intended to be used by the Tx Job Manager (on the Tx-side of the fabric adapter logic) so RxDMA forwards the message metadata to Tx.
- The received page must be exactly one flit, and would have a minimum size of 64 bytes, as indicated from Page Reader to RxDMA. Header-decap would send those remaining bytes as one flit in the data path to RxDMA. These are just pad bytes, so RxDMA will discard them.

10.9 Fabric Adapter Statistics and Debug Support

TBD

11 Pyxis Programming Model

11.1 Operational Overview

The UIEs and specialized DMA engines are programmed to execute a sequence of ***trips***, where each trip roughly corresponds to a neural-network layer or collective operation. Trips can be chained together such that the completion of one trip automatically triggers the following trip. Hardware ***barriers*** can be used to control dependencies between trips and/or pass control to or from a supervising CPU. All of these workloads are specified through **Trip Descriptors** which are prepared by (on-chip) software and enqueued to the various engines.

Each trip includes the following steps, with specific emphasis on the UIEs.

1. UIEs are controlled by microcoded ***sequencers,*** and the microcode may be dynamically loaded for each trip. This microcode can be predetermined and stored in a database in AMEM or external HBM DRAM. For each trip, the appropriate microprograms must be transferred from the database to small microcode memories in each UIE. This is performed by the **Microcode Patch Engine (MPE)**, which is a specialized DMA engine.

   The Microcode Patch Engine receives a descriptor consisting of a pointer to the microcode data structure for the trip, and the size of that data. The data structure is organized as a list of (µcode_address, length, µcode_data) tuples. The MPE simply reads each entry in the list, and writes the µcode_data portion to the appropriate sequencer microcode memory and memory index, as indicated by the µcode_address.

   UIE microcode is very compact, and the sequencers' microcode memories have space for multiple trips'-worth of microprograms. This allows microcode loading to be performed ahead of when the trip is to be executed.

2. For some trips, it is also necessary to first DMA-transfer weights or data from the HBM DRAM into AMEM before the UIE can read the weights or data from AMEM. The DMA engine operates from a descriptor which provides the starting source address in HBM, the starting destination address in AMEM, and the size of the transfer.

3. Once these setups are done, the UIE trip can be launched. The UIE will fetch weights & data from AMEM, execute the task (such as a convolution or matrix multiplication) and write the results back to AMEM. At the completion of each trip, the UIEs can immediately start the next trip once any dependencies are satisfied (e.g., the next microcode has been loaded). For efficiency, the various execution steps can be chained via a hardware-based dependency tracking mechanism, and multiple trips can be queued up by the controlling software.

11.2 Trip Programming and Trip Chaining

Software sets up each trip by queueing ***trip descriptors*** to the UIEs and DMA engines, e.g., the Microcode Patch Engine, HBM-to-AMEM DMA controller, and UIE sequencers. Multiple trip descriptors can be set up and enqueued for efficiency.

The (per-engine) trip descriptors include starting microcode program counters (***StartPCs***), any engine-specific trip parameters, and dependency barriers described below.

Each subsequent trip may be dependent on the results from a prior trip, and also require its own microcode and weights loading. The phases may be overlapped as illustrated in the example flow diagram below, where the 3rd trip is dependent on results from the 1st trip, and 4th is dependent on the 2nd. The arrows show the dependencies linking each of the phases within and between trips.

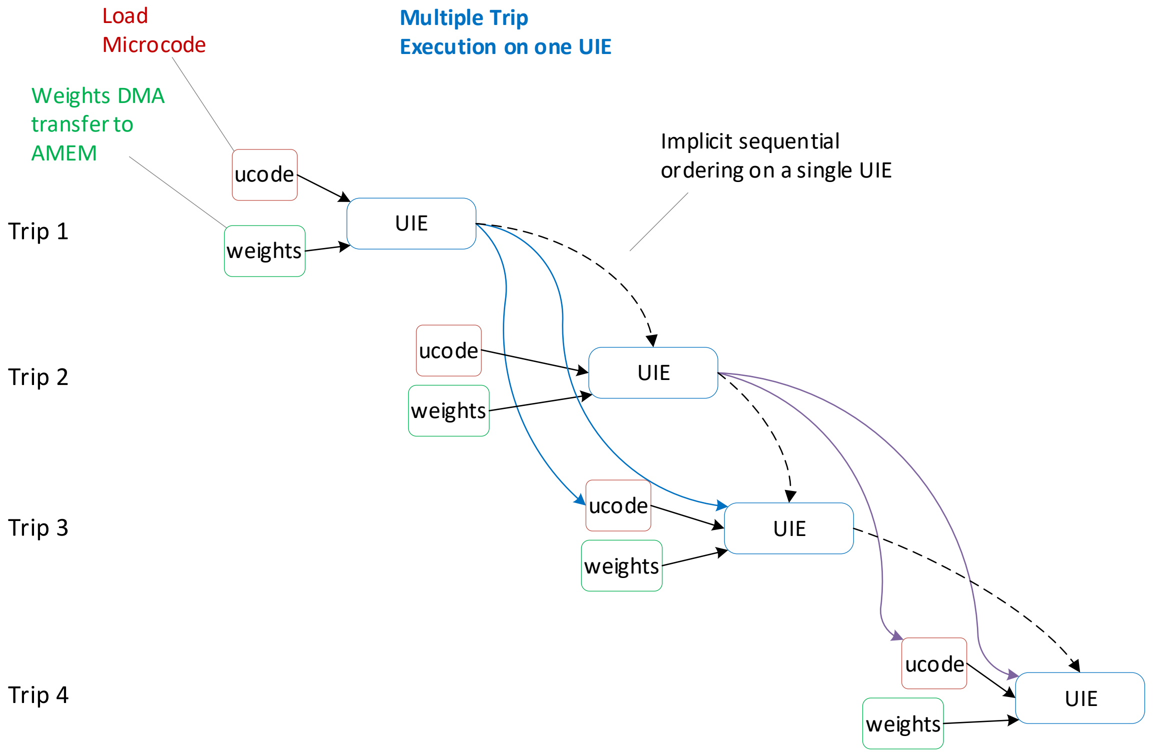

Load Microcode
Multiple Trip Execution on one UIE
Weights DMA transfer to AMEM
Implicit sequential ordering on a single UIE
Trip 1
Trip 2
Trip 3
Trip 4
ucode
weights
UIE
ucode
weights
UIE
ucode
weights
UIE
ucode
weights
UIE

Figure 11-1: Example Trip Dependency Chaining

The key objective is to keep the UIEs busy, and overlap the trip phases as much as necessary to that end. Pyxis supports efficient trip chaining by allowing each phase of multiple trips to be enqueued in hardware, and providing a hardware barrier mechanism for launching the various phases as the dependencies are resolved.

The hardware also allows these chains to be forked or joined for multiple UIEs (or other tasks such as DMA transfers) to create complex execution graphs.

11.3 Trip Manager

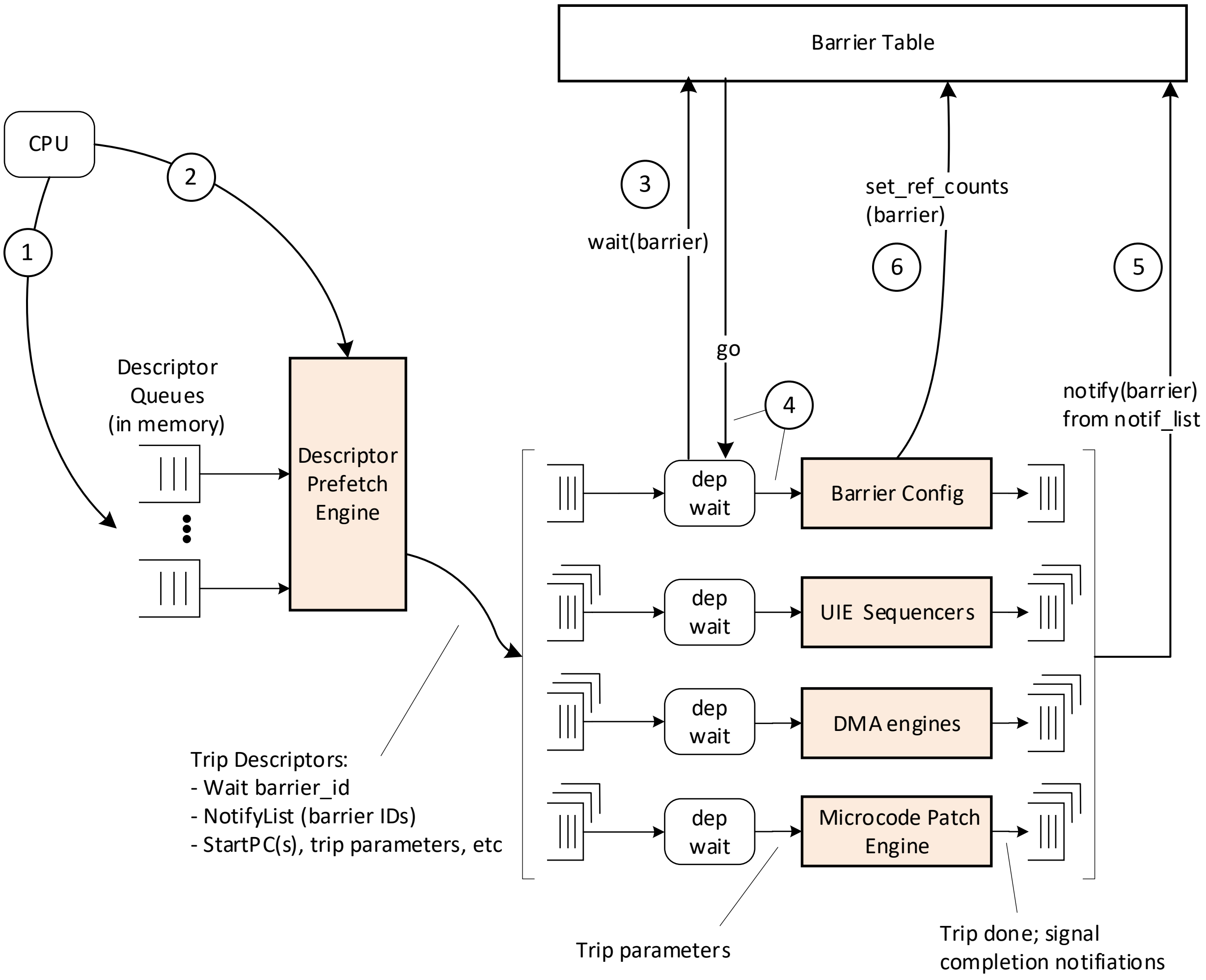


Figure 11-2: Trip Control Flow

Each of the major phases of a typical trip (microcode loading, weights transfer, and UIE execution) is a separate task programmed by software, with a dedicated memory-mapped hardware Descriptor Queue for the associated engine (Microcode Patch Engine, UIE sequencers, etc.). The overall trip control flow is summarized in Figure 11-2 .

1. Software sets up **Trip Descriptors** in memory, in per-engine **Descriptor Queues**. Each Descriptor includes fields for waiting on a hardware barrier, signaling completion to one or more barriers, and engine-specific trip parameters.

   The hardware barriers support fields are:

   - `wait_barrier_id`: aggregates all dependencies for this trip, and

- `notify_list`: specifies the barriers to be signaled ("notified") when the trip is done.

The engine-specific trip parameters include:

- For the Microcode Patch Engine, the trip parameters specify the location (global memory address) and size of the microcode blocks to be loaded.
- For DMA transfers, the descriptor includes the source and destination starting addresses and transfer size information
- For the UIEs, the trip descriptor includes StartPCs and trip parameters for all of the sequencers in the UIE.
- Each hardware barrier object is also configured dynamically via a descriptor, using this same mechanism. A barrier descriptor specifies the following fields
    - `Barrier_id`: index into a hardware table of barrier objects
    - `notify_count`: the number of tasks (or threads) which must be completed for this barrier, i.e., the number of producer dependencies for this barrier.
    - `wait_count`: the number of consumer engines which are dependent on this barrier.

2. The descriptor queues are arranged as ring-of-lists buffers in memory, served by a **Descriptor Prefetch Engine**. (See sections 11.5 and 12.5.) Each queue has memory-mapped targets to enqueue a list of descriptors to the ring. Software enqueues a list of descriptors by writing an (address, length) tuple to the enqueue target for that queue, which adds the list to the ring-of-lists structure for the queue. This triggers the Descriptor Prefetch Engine to prefetch descriptors from memory into shallow hardware-based (FIFO) queues.
3. For each engine, before the trip can be launched from the head of the shallow hardware-based descriptor queue, Trip Manager issues a `wait(barrier_id)` request to the Barrier Table, and waits until the barrier dependencies are resolved.
4. Subsequently, when the barrier dependencies have been resolved, the Barrier Table will signal "go" back to the Trip Manager, at which point the trip parameters may be sent to the engine for execution. The engine then executes the indicated tasks (UIE trip, DMA transfer, etc.)
5. When the task is complete, the engine signals the completion back to the Trip Manager, which then issues the `notify(barrier_id)` to the Barrier Table for each barrier_id in the descriptor's `notify_list`. This decrements the notify_count for each dependent barrier of this trip.
6. The barriers are set up by a small engine which is itself subject to barrier-waits. To set up each barrier, hardware issues a `set_ref_counts(barrier_id, notify_count, wait_count)` operation to the **Barrier Table**. As described in section 11.5.2.1, barriers may safely be set up *before or after* they are queried by the other engines with `wait()` and `notify()` operators.

11.4 Trip Descriptor Formats

Descriptors include a common header, followed by engine-specific fields. Descriptors are fixed-size per engine type. The specified fields are listed in the following sections. Details are provided in the Pyxis Register Programming Guide

Section	Size (bits)	Description
Common descriptor header	128	Barriers and Trip-ID information. See section 11.4.1
Engine-specific trip parameters descriptor	Variable	Per-engine descriptor fields. Fixed size per engine type.
Padding (as needed)	Variable	Pad to multiple of 64 bits (8 bytes)

Note: All descriptors are rounded-up to a multiple of **8 bytes** when stored "at rest" in memory, with padding, if needed, on most-significant bits.

11.4.1 Common Descriptor Header

Each descriptor includes the following common fields. This header occupies 16 bytes in memory:

Field	Size (bits)	Description
Wait_barrier_id	10	(local) Barrier_id to wait on for this trip. Barrier_id=0 is "no wait"
Notify_list	8*12	List of (global) barrier_ids to be "notified" when trip is complete. Barrier_id `0` indicates a no-op. Each element in the list is the concatenation of `{cluster_id[1:0], local_barrier_id{9:0]}`
Trip_id	20	Software-configured Trip ID For debug and tracing
Descriptor Format	2	Set to 2'b00

Note that 12-bit "global" barrier IDs consist of a 2-bit Cluster index and 10-bit "local" barrier ID. This can address 1024 barrier objects in each of 4 clusters.

11.4.2 UIE Descriptors

This defines a UIE "trip", with starting microcode program-counters for each of the sequencers in a UIE, and per-trip function parameters.

The cluster-ID and UIE-ID are implied by the queue for the descriptor, which maps to a specific engine and a specific cluster.

The descriptor format shown below includes all fields. See the Pyxis register programming guide for exact field names, bit positions, and organization.

Field	Size (bits)	Description
Common header	128	
Per-sequencer trip valid mask, **Per-ITFE valid mask**	13 2	Per-sequencer bits indicating whether the sequencer has valid work for this trip. If a sequencer's bit is set to '1', that sequencer will be active for this trip, and the associated parameters (such as StartPC) must be valid. If set to '0', the sequencer will not be active for this trip. Also includes 2 mask bits for the Indirection Table Fetch Engines (ITFEs) for AMEM-read and AMEM-write. Note that if an ITFE valid_mask bit is set, the associated sequencer (AMEM read or AMEM write) valid_mask bit must also be set. However, if indirection is not being used for the trip, the IFTE valid bit is not set, even if the associated AMEM read/write sequencer valid_mask bit is set. It is legal for no sequencers to be active (all mask bits zero) to create a "no-op" trip. This can be used to effectively extend the Notify_list by adding a "no-op" trip descriptor after a normal trip descriptor in the same UIE's descriptor queue.
StartPC[4:0] * 13	5 * 13	Microcode start-PC for each of the Sequencers in the UIE.
AMEM Read base address[25:4]	22	AMEM partition-granular starting address for the primary read port ("vertical" data), within the engine's local cluster AMEM.
AMEM Weights base address[25:4]	22	AMEM partition-granular starting address for the weights-read ("horizontal" data) port, within the engine's local cluster AMEM.
AMEM Write base address[25:4]	22	AMEM partition-granular starting address for the Write port, within the engine's local cluster AMEM.
AMEM Read bank hash dsbl	1	Use hashed (0) vs unhashed (1) bank addressing for reads ("vertical" data)
AMEM Write bank hash dsbl	1	Use hashed (0) vs unhashed (1) bank addressing for writes
AMEM Weights bank hash dsbl	1	Use hashed (0) vs unhashed (1) bank addressing for weights ("horizontal" data) read
AMEM Read Indirect Vld	1	Enables the AMEM Read Indirection Table Fetch engine operation for this trip
AMEM Read Indirect Cmd index	2	Selects one of the 4 programmed command words for the AMEM Read Indirection Table Fetch engine (if valid is set)
AMEM Write Indirect Vld	1	Enables the AMEM Write Indirection Table Fetch engine operation for this trip
AMEM Write Indirect Cmd index	2	Selects one of the 4 programmed command words for the Write Indirection Table Fetch engine (if valid is set)
Trip Power	7	Estimated Power for the trip, in normalized power units. This is used to control the dynamic power ramp-up rate by the Active Power Manager (see Section 13, "Active Power Management")

11.4.3 HBM to AMEM DMA Descriptors

This defines a DMA transfer from HBM to AMEM within one cluster.

Field	Size (bits)	Description
Common header	128	
AMEM start address[26:7]	20	AMEM target base address. Word (128-byte) aligned. Includes bank-hash-disable as the top bit.
HBM base address[35:10]	26	HBM source base address. 1024-byte aligned.
Length	16	DMA transfer length in 1K-byte blocks. N-1 encoded.
Trip Power	7	Estimated Power for the DMA job, in normalized power units. (See Section 13, "Active Power Management")

11.4.4 AMEM to HBM DMA Descriptors

This defines a DMA transfer from AMEM to HBM within one cluster. .

Field	Size (bits)	Description
Common header	128	
AMEM start address[26:7]	20	AMEM source base address. Word (128-byte) aligned. Includes bank-hash-disable as the top bit.
HBM base address[35:10]	26	HBM target base address. 1024-byte aligned.
Length	16	DMA transfer length in 1K-byte blocks. N-1 encoded
Trip Power	7	Estimated Power for the DMA job, in normalized power units. (See Section 13, "Active Power Management")

11.4.5 Inter-Cluster DMA Descriptors

This defines a DMA transfer from one cluster's AMEM to another cluster's AMEM. The specific source and destination clusters are implied by the queue for the descriptor, which maps to a specific cluster-pair. Note that the ICDMA engine (and descriptor queue) for each transfer is located in the source AMEM cluster.

Field	Size (bits)	Description
Common header	128	
AMEM Source start address[26:7]	20	AMEM base address in source cluster. Word (128-byte) aligned. Includes bank-hash-disable as the top bit.
AMEM Dest start address[26:7]	20	AMEM base address in destination cluster. Word (128-byte) aligned. Includes bank-hash-disable as the top bit.
Length	16	DMA transfer length in 128-byte words. N-1 encoded
Core Switch Plane Mask	4	May be used to constrain which of four planes in the core-switch may be used for this transfer. A set-bit indicates a plane may be used. Typical value is 0xF. The value 0x0 is illegal – at least one plane mask bit must be set.
Trip Power	7	Estimated Power for the DMA job, in normalized power units. (See Section 13, "Active Power Management")

11.4.6 RDMA Tx Descriptors

This defines an RDMA transfer from this Pyxis to a remote Pyxis chip. There are separate RDMA Tx descriptor queues for each (source) UIE cluster. The source AMEM address for the data is relative to the local cluster. For RDMA Tx operation, see section 10.7.1

Field	Size (bits)	Description
Common header	128	
TJID	8	Tx Job ID
RJID	8	Rx Job ID to link to the remote Rx Job Table.
Dest VOQ	13	Target VOQ, which identifies target chip, cluster and traffic class
AMEM Address	20	AMEM base address in source cluster. Word (128-byte) aligned. Includes bank-hash-disable as the top bit.
NumWords	17	RDMA transfer length, in 128-Byte words. Allows up to 8MB job size with 2^16 words; legal range of this field is **[1, 2^16]**
PageSizeWords	7	Number of 128-Byte words per page transmitted over the fabric. All transmitted pages, except possibly the last page, will use this size. Legal range of this field is **[32, 127]**. The lower bound of 32 words/page ensures that the number of pages generated (for a large job) cannot exceed certain internal hardware limits. A recommended value, for maximum efficiency is **125 words/page**

11.4.7 RDMA Rx Descriptors

This defines an RDMA transfer to this Pyxis from a remote Pyxis chip. There are separate RDMA Rx descriptor queues for each (source) UIE cluster. The Target AMEM address for the data is relative to the local cluster. For RDMA Rx operation, see section 10.8.1

Field	Width	Description
Common header	128	
TJID	8	Tx Job ID to link to the remote Tx Job Table
RJID	8	Rx Job ID
Msg VOQ	13	VOQ pointing to the Tx chip for the associated Tx Job, for sending CTS and ACK/NACK messages
AMEM Address	20	AMEM base address in target cluster. Word (128-byte) aligned. Includes bank-hash-disable as the top bit.
NumWords	17	RDMA transfer length, in 128-Byte words. Allows up to 8MB job size with 2^16 words; legal range of this field is **[1,2^16]**

11.4.8 Datagram Tx Descriptors

This defines a single-packet transfer from this Pyxis to a remote Pyxis chip, using the “Datagram over Fabric” service semantics. See section 10.5 “Datagram over Fabric” and 11.5.2.1 “Datagram Tx

Descriptor Queues”. These descriptors do *not* include a barrier-information header.   All descriptors are padded up to a multiple of 64 bits (8 bytes).

Field	Size (bits)	Description
Dest VOQ	13	Target VOQ, which identifies target chip, cluster and traffic class
AMEM Address	20	AMEM base address in source cluster. Word (128-byte) aligned. Includes bank-hash-disable as the top bit.
NumBytes	14	RDMA transfer length, in bytes. Legal range of this field is **[1, 16256].** Note that the maximum value (16256) equals 127*128 bytes.
RXQ Index	1	Target Receive-Queue, which identifies the receive descriptor queue on the Rx side.

11.4.9 Datagram Rx Descriptors

This descriptor defines the receive buffer information, for a single-packet transfer to this Pyxis from a remote Pyxis chip, using the “Datagram over Fabric” service. See section 10.5 “Datagram over Fabric” and 11.5.2.2 “Datagram Rx Descriptor Queues”. These descriptors do *not* include a barrier-information header. All descriptors are padded up to a multiple of 64 bits (8 bytes).

Field	Size (bits)	Description
AMEM Address	20	AMEM base address in target cluster. Word (128-byte) aligned. Includes bank-hash-disable as the top bit.
BufferSizeBytes / PacketSizeBytes	14	This field is shared by software and hardware. This field is initially set up by the software as **BufferSizeBytes** to specify the AMEM buffer size to receive the data. The maximum legal value is 16,256 bytes. When a datagram packet is received, hardware will write the received packet length as **PacketSizeBytes** into this field. This value will always be less than or equal to BufferSizeBytes, and has granularity of one byte.

11.4.10 Microcode Patch Engine Descriptors

The Microcode Patch Engine is described in more detail in section 12.6 below. It takes a pointer to a “blob” of patch descriptors, or a more traditional DMA descriptor. The two descriptor formats are overloaded using a “union” with a mode indicator as shown below

Field	Size (bits)	Description
Common header	128	
Descriptor parameters	96	Union which overloads two different descriptor types Note: descriptor parameters are rounded up to 128b (16B) for at-rest storage since all descriptors are stored in multiples of 8 bytes.

11.4.10.1 MPE Descriptor Parameters for Patch Mode

Field	Size (bits)	Description
Patch blob address	32	Pointer to the patch blob in global address space. This address will be remapped from 32b to the full NoC-visible address space
Patch blob length	16	Length of the blob, in units of 8 bytes. Patches must always be a multiple of 8 bytes . N-1 encoded.
Num Patches	16	Number of patches in the blob. N-1 encoded.
Reserved	16	
Mode	16	Set to **0x0000** for Patch mode

11.4.10.2 MPE Descriptor Parameters for DMA Mode

Field	Size (bits)	Description
Source Address	32	Base address of data to be read. This address will be remapped from 32b to the full NoC-visible address space
Target Address	32	Base address to be written. This address will be remapped from 32b to the full NoC-visible address space
Length	16	Transfer length, in units of 8 bytes. N-1 encoded
Mode	16	Set to **0x0001** for DMA mode

The 32-bit addresses in MPE descriptors are remapped to address the full NoC-visible address space. The remapping is described in section 12.7.

11.4.11 Barrier Configuration Engine Descriptors

There is one barrier-configuration engine descriptor queue per Trip Manager (i.e., per cluster). The barrier_ids in the descriptor are local to the particular barrier table.

Field	Size (bits)	Description
Common header	128	
Barrier_id	10	The barrier to be initialized
Notify_count	5	Supports up to 31 producers for this barrier.
Wait_count	5	Supports up to 31 consumers for this barrier.

11.5 Descriptor Queues

The following tables summarize the descriptor queues for all of the specialized engines on Pyxis

<table>
<tr><th>Descriptor Queue</th><th>Quantity</th><th colspan="4">Description</th><th>Queue ID (per-cluster)</th></tr>
<tr><td><b>UIE</b></td><td>12 per cluster</td><td colspan="4">Each queue corresponds to one UIE</td><td>0x0 – 0xB</td></tr>
<tr><td><b>RDMA Tx</b></td><td>2 per cluster</td><td colspan="4">RDMA Tx Descriptor Queues (per cluster)</td><td>0xC – 0xD</td></tr>
<tr><td><b>RDMA Rx</b></td><td>2 per cluster</td><td colspan="4">RDMA Rx Descriptor Queues (per cluster)</td><td>0xE – 0xF</td></tr>
<tr><td><b>H2A</b></td><td>1 per cluster</td><td colspan="4">HBM-to-AMEM transfer DMA</td><td>0x10</td></tr>
<tr><td><b>A2H</b></td><td>1 per cluster</td><td colspan="4">AMEM-to-HBM transfer DMA</td><td>0x11</td></tr>
<tr><td rowspan="14"><b>ICDMA</b></td><td rowspan="14">3 per cluster</td><td colspan="4">Inter-cluster DMA, from AMEM to AMEM</td><td rowspan="14">0x12 – 0x14</td></tr>
<tr><td>(Source) Cluster</td><td>Descr Queue in cluster</td><td>Queue ID</td><td>Source to Dest clusters</td></tr>
<tr><td rowspan="3">0</td><td>0</td><td>0x12</td><td>0 ⇨ 1</td></tr>
<tr><td>1</td><td>0x13</td><td>0 ⇨ 2</td></tr>
<tr><td>2</td><td>0x14</td><td>0 ⇨ 3</td></tr>
<tr><td rowspan="3">1</td><td>0</td><td>0x12</td><td>1 ⇨ 0</td></tr>
<tr><td>1</td><td>0x13</td><td>1 ⇨ 2</td></tr>
<tr><td>2</td><td>0x14</td><td>1 ⇨ 3</td></tr>
<tr><td rowspan="3">2</td><td>0</td><td>0x12</td><td>2 ⇨ 0</td></tr>
<tr><td>1</td><td>0x13</td><td>2 ⇨ 1</td></tr>
<tr><td>2</td><td>0x14</td><td>2 ⇨ 3</td></tr>
<tr><td rowspan="3">3</td><td>0</td><td>0x12</td><td>3 ⇨ 0</td></tr>
<tr><td>1</td><td>0x13</td><td>3 ⇨ 1</td></tr>
<tr><td>2</td><td>0x14</td><td>3 ⇨ 2</td></tr>
<tr><td><b>Microcode Patch Engine</b></td><td>13 per cluster</td><td colspan="4">Each queue corresponds to one MPE channel</td><td>0x15 – 0x21</td></tr>
<tr><td><b>Barrier Config Engine</b></td><td>1 per cluster</td><td colspan="4"></td><td>0x22</td></tr>
<tr><td colspan="7">The following descriptor queues implement <b>datagram</b> (packet) ring buffers, rather than providing trip-descriptors for the specialized engines. These descriptors do not include a trip-descriptor header. The datagram descriptor rings are described in section 11.5.2 below.</td></tr>
<tr><td><b>Datagram Tx</b></td><td>2 per cluster</td><td colspan="4">Transmit Datagram buffer-pointers & metadata</td><td>0x23 – 0x24</td></tr>
<tr><td><b>Datagram Rx</b></td><td>2 per cluster</td><td colspan="4">Receive Datagram buffer-pointers & metadata</td><td>0x25 – 0x26</td></tr>
</table>

Table 11-1: Pyxis Descriptor Queues

11.5.1 Trip Descriptor Queues

Each of the per-engine trip-descriptor queues is organized as a **ring of lists of descriptors**. For each engine, software constructs a list of descriptors (of arbitrary length) in memory, and when the descriptors are to be executed by hardware, software enqueues this list onto a ring of these lists. Software may then construct the next list of descriptors, and links that into the ring, and so on. Hardware will fetch (and execute) all descriptors in the first list, then look for the next list in the ring and execute all descriptors on that list, etc.

This two-dimensional structure is shown below. Each queue has a Head/Tail pointer register into a 4-element ring. The ring is also implemented in registers. Each element in the ring is an (Address, Length) register, pointing to a list of descriptors in memory.

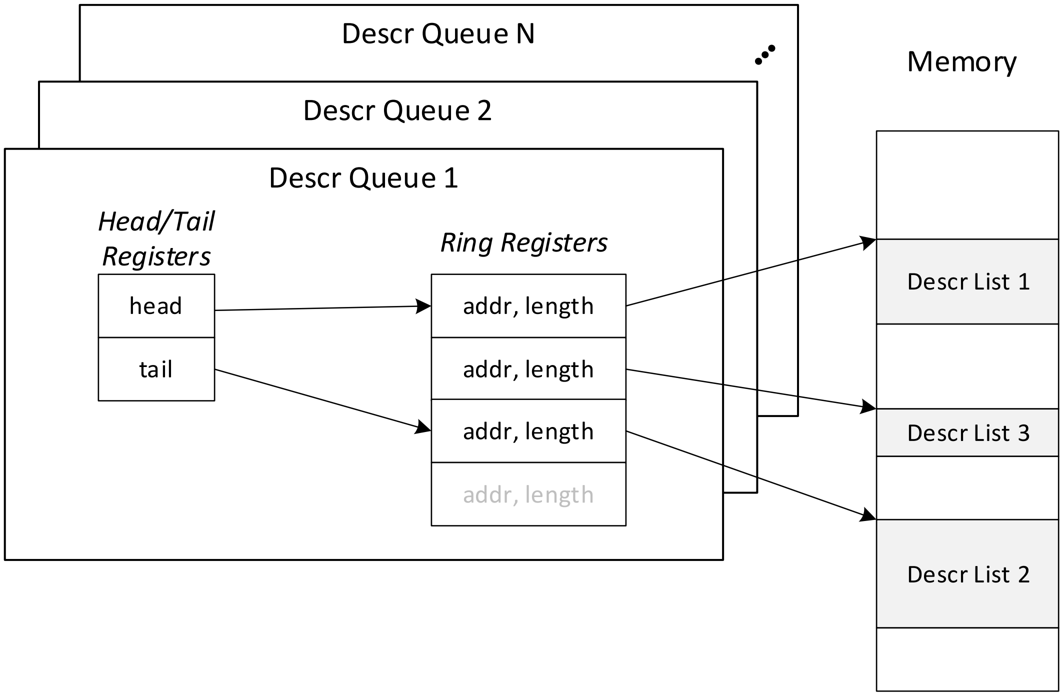


Figure 11-3:Trip Descriptor Queue Structure

The (per-queue) Head and Tail pointers are each 3 bits, which may be thought of as the concatenation of **1 wrap bit** and **2 index bits**. The wrap bit is used to distinguish the full vs empty states.

```
struct q_ring_pointer_t {
    uint q_idx : 2;  // bits [1:0]
    uint wrap  : 1:  // bit [2]
}

struct descr_q_hd_tl_t {
    q_ring_pointer_t head;
    q_ring_pointer_t tail;
}
```

The tail pointer is increment by software (through a special CSR access, described below), and the head pointer is incremented by hardware.

Operation of the trip descriptor queue is as follows:

- Initially the Head and Tail pointer registers are equal, and this indicates the queue is empty.

- To add a list of descriptors, software first prepares the list by storing the descriptors into contiguous memory locations.
- Software then enqueues the (Address, Length) of the list to a ring entry. This is implemented by writing `(QueueID, Address, Length)` to a special CSR target which loads the `(Address, Length)` into the next available Ring Register entry for that `QueueID`. This also increments that queue's ring Tail pointer as a side-effect.
- Hardware will make this entry visible to the Descriptor Prefetch engine at that point, which will start fetching descriptors from the list entry at the head of the ring. When the list has been fully fetched (though not necessarily executed yet), hardware will increment the Head pointer register and look for the next (Address, Length) entry in the ring.
- The queue is full when `head ^ tail == 0x4`, i.e., when the `head.q_idx == tail.q_idx` and `head.wrap != tail.wrap`. The queue is empty when `head==tail`.

Software may monitor and manage the queue in the following manner:

- Hardware **exports the current state of the head pointers** for all queues via status registers (CSRs). For efficiency, the head pointers for all queues are packed into a minimum number of 64-bit CSRs
- Software maintains a **shadow copy of the tail pointers**., which it (internally) increments each time it issues and enqueue to hardware. Note that the shadow copies includes the wrap bits.
- Software polls the head pointer CSR(s), and compares the head pointer to the shadow copy of the tail pointer for each queue. Software can then determine the number of outstanding elements (i.e., descriptor-lists) in each queue as the **difference between the pointers**.

11.5.1.1 Trip Descriptor List format

All trip descriptors are sized in units of 8 bytes, and must use 8-byte alignment in memory. The address of each list-of-descriptors is 8-byte aligned. The length of each list-of-descriptors is in units of 8 bytes.

To construct a descriptor list in memory, the trip descriptors are padded to a multiple of 8 bytes (per descriptor) and stored in contiguous memory locations, with 8-byte alignment. To enqueue this list, the address and length of the list are written to a special CSR which performs the enqueue operation – pushing the (address, length) onto the next entry in the ring for the . The format of data written for the enqueue operation is shown below. There is one CSR for the trip-descriptor queues in each cluster.

Field	Size (bits)	Description
Queue ID	6	Indicates the specific queue to be enqueued
Descriptor List address	32	Address of the beginning of the descriptor list. Must be 8-byte aligned (3 lsb's must be zero)
Descriptor List Length	16	Length of the list, in units of 8-byte blocks. N-1 encoded

Table 11-2: Descriptor List Enqueue CSR Target format

Descriptor Lists may be stored in HBM or AMEM. As described in section 12.7, the 32-bit Descriptor List address is remapped to be able to reach anywhere in the Pyxis address space.

See the Pyxis register programming guide for addressing information.

11.5.2 Datagram Descriptor Queues

For datagrams, the Tx and Rx descriptor queues are managed by software using descriptor rings in memory

11.5.2.1 Datagram Tx Descriptor Queues

TxQ descriptors are organized in a ring buffer with the **tail-pointer written by software** and **head-pointer written by hardware**. Each descriptor includes a buffer pointer for the transmit data, payload size, and the VOQ to identify the target chip, cluster and RxQ at the target chip/cluster. These are set up by software. The TxQ descriptor ring also has a **prefetch pointer** between head & tail, for the Descriptor Prefetch Engine to move descriptors into the hardware FIFOs for immediate use.

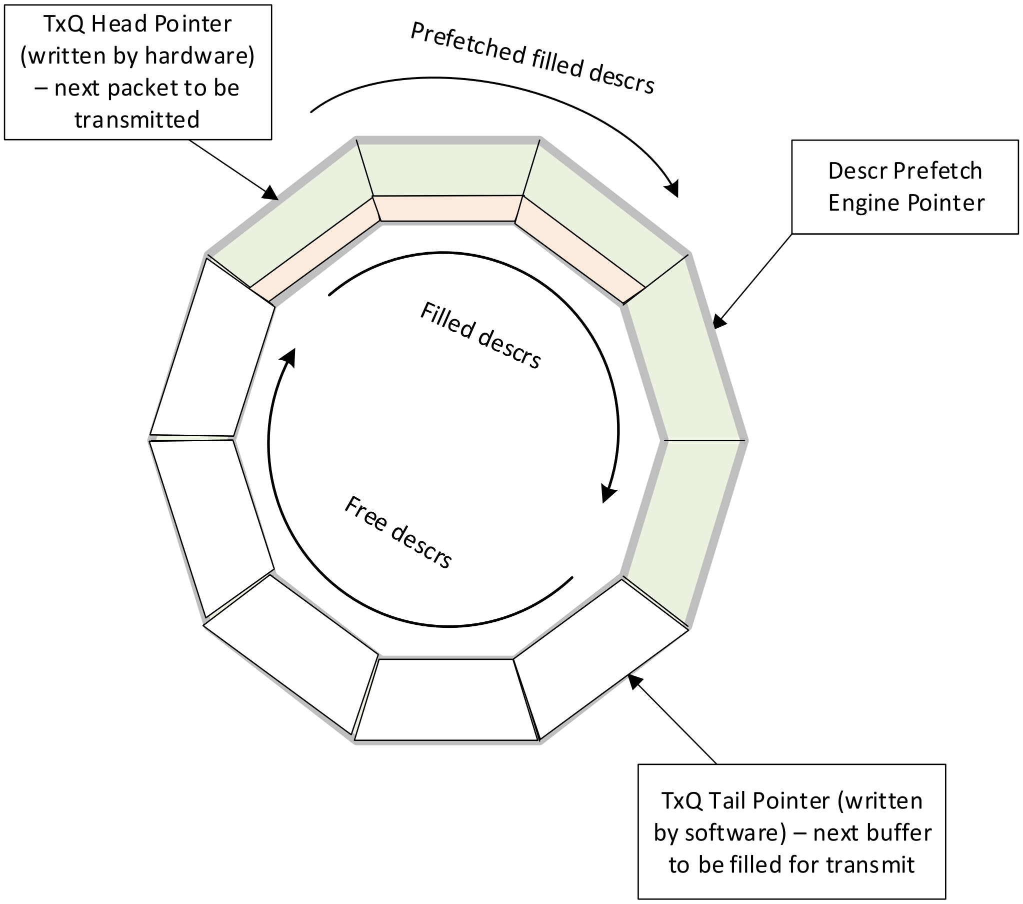

TxQ Head Pointer (written by hardware) – next packet to be transmitted
Prefetched filled descrs
Descr Prefetch Engine Pointer
Filled descrs
Free descrs
TxQ Tail Pointer (written by software) – next buffer to be filled for transmit

Figure 11-4: TxQ Datagram Descriptor Rings

Initially, software sets the TxQ Head pointer = Tail pointer since the queue is empty. As software fills the buffers, it can advance the Tail pointer to make the descriptors visible to hardware. Software advances the tail pointer to the address of the entry just past the last valid descriptor.

This triggers the Descriptor Prefetch Engine to move some number of descriptors from SRAM-based descriptor rings into the (shallow) hardware FIFOs feeding to the Tx Fabric Adapter.

As hardware subsequently transmits the packets, it advances the Head pointer and may signal an interrupt to indicate that the buffer has been freed.   Hardware advances the Head pointer to one entry past the last freed descriptor.

Software can maintain a shadow-copy of the Head pointer, and determines which descriptors' buffers have been freed by comparing the shadow pointer with the hardware Head pointer (after which software updates it shadow pointer to match the hardware pointer)

In this scheme, the TxQ is full when the Head pointer = Tail pointer+1 (modulo the ring size).   A ring of N descriptors is full when it holds N-1 entries.

11.5.2.2 Datagram Rx Descriptor Queues

RxQ descriptors are organized in a ring buffer with **tail-pointer written by software** and **head-pointer written by hardware**. Each descriptor includes a buffer pointer for the received data (written by software), and the received datagram size (written by hardware). The RxQ descriptor ring also has a **prefetch pointer** between head & tail, for the Descriptor Prefetch Engine to move descriptors into the hardware FIFOs for immediate use.

The RxQ descriptor ring is logically divided into **free descriptors**, and **filled-descriptors** which hold the received packets waiting to be processed by software. The free descriptors section is further divided by the intermediate Prefetch pointer.

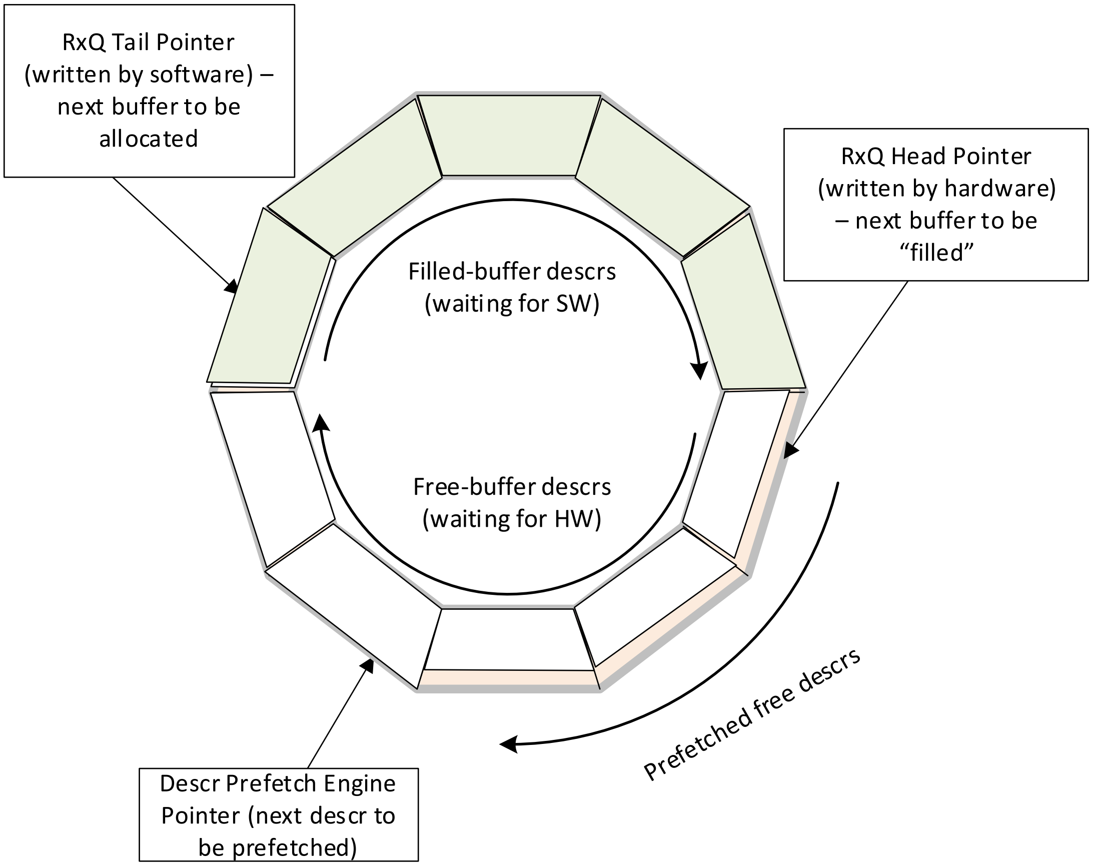

RxQ Tail Pointer (written by software) – next buffer to be allocated
RxQ Head Pointer (written by hardware) – next buffer to be "filled"
Filled-buffer descrs (waiting for SW)
Free-buffer descrs (waiting for HW)
Prefetched free descrs
Descr Prefetch Engine Pointer (next descr to be prefetched)

Figure 11-5: RxQ Datagram Descriptor Rings

Initially, software sets Head pointer = Tail pointer = Prefetch pointer to indicate the RxQ is empty. Software then prepares descriptors and writes them into locations starting from the tail pointer, and then advances the tail pointer to make the descriptors visible to hardware. Software advances the tail pointer to the address of the entry just past the last valid descriptor.

This triggers the Descriptor Prefetch Engine to prefetch some number of descriptors which are sent to the Fabric Rx logic (which effectively caches these descriptors), and advances the Prefetch pointer.

As each (Datagram) packet is received, it is written to the buffer location for the next (prefetched) descriptor, and hardware will update the descriptor (in memory) to convey the length of the packet and the packet-available status. Hardware then advances the RxQ Head pointer which tells software that the descriptor has been updated (effectively removed from the free-descriptors and added to the filled-descriptors section). This signals an interrupt to indicate that the buffer has been filled to software. Hardware advances the Head pointer to one entry past the last filled descriptor.

When software receives the interrupt, it reads the hardware Head pointer to see how many descriptors have been “filled” since the last time it serviced the RxQ. Software can keep a shadow copy of the hardware head pointer. Software processes the packets between its (shadow) head pointer and the hardware's, and then updates the shadow pointer to match the hardware pointer.

After processing the received packet descriptors, software can subsequently prepare those descriptors with fresh buffers. Once that’s done, software updates (advances) the Tail pointer to resupply the Free descriptors, by setting the tail pointer one entry past the last free descriptor.

In this scheme, the RxQ is full when the Head pointer = Tail pointer+1 (modulo the ring size). A ring of N descriptors is full when it holds N-1 entries.

11.6 HARDWARE-MANAGED BARRIERS

In the following description, the term “thread” indicates a trip or task running on an engine such as UIE, DMA engine or Microcode Patch engine.

Hardware barriers provide synchronization by enforcing the dependencies between **producer threads** and **consumer threads**. The following rules define dependency graphs using these hardware barriers.

- Each producer thread may *notify* **zero or more** barriers.
- Each consumer thread may *wait* on **at most one** barrier.
- Each barrier may have **one or more** producer threads (notif_count >= 1)
- Each barrier may have **one or more** consumer threads (wait_count >= 1)

Some examples are shown in Figure 11-6, where “T” nodes are threads and “B” nodes are barriers

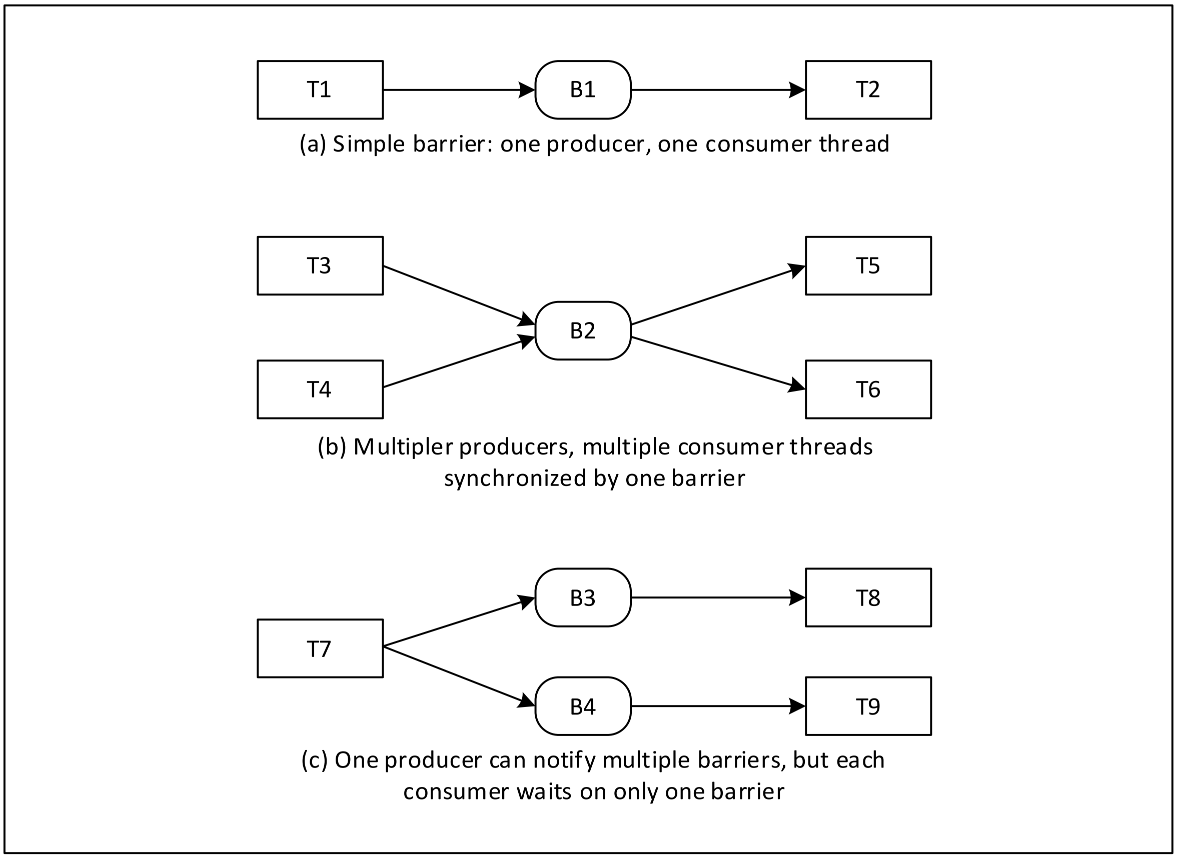

T1
B1
T2
(a) Simple barrier: one producer, one consumer thread
T3
T4
B2
T5
T6
(b) Multipler producers, multiple consumer threads synchronized by one barrier
T7
B3
B4
T8
T9
(c) One producer can notify multiple barriers, but each consumer waits on only one barrier

Figure 11-6: Barrier Synchronization Examples. Ti = thread i, Bj = Barrier j

11.6.1 Barrier Objects

Hardware maintains the barrier objects to encapsulate thread dependencies. Each barrier object contains:

- `notify_count` (signed integer) which tracks the number of producer thread dependencies for the barrier. It is incremented to set the initial count, and decremented by each producer when it completes its task. While nonzero, any consumer thread (i.e., trip) will be blocked.
- `wait_count` (signed integer) which tracks the number of consumer threads which are still waiting on this barrier. This is incremented to set the initial count, and decremented by each consumer when the thread (trip) is launched. When this reaches zero, the barrier may be freed
- `ref_counts_valid` flag which indicates whether the initial notify_count and wait_count have been set up. This is used to disambiguate the state when all counts are zero, which would occur when the barrier is new and the counts not yet set up, and also when the counts have reached zero when the barrier is complete

Using C struct-like nomenclature, the barrier definition is

```
struct bt_barrier_t {
   int notify_count       : 6;  // 6-bit signed value allows up to 31 notifiers
   int wait_count         : 6;  // 6-bit signed value allows up to 31 waiters
   uint ref_counts_valid  : 1;
};
```

11.6.2 Barrier Operation

The following pseudocode describes the barrier operation. There are three main primitives:

- `set_ref_counts()` sets up the `notify_count`, `wait_count` and a `ref_counts_valid` flag, by incrementing the count variables by specified amounts, namely the total number of expected “references” by producers (`notify_count`) or consumers (`wait_count`) . This is normally invoked by a barrier-configuration hardware thread, as described in section 11.3
- `wait()` is sent by a consumer-thread to wait on a barrier until dependencies are cleared.
- `notify()` is sent by a producer-thread to decrement dependency count for a barrier.

Note: This design allows the barrier to be set up independently of when the producers and consumers notify/wait on the barrier. The atomic increment/decrement scheme eliminates races between barrier configuration and barrier usage (the order doesn’t matter) as long as a barrier has been freed before it is re-used.

If a barrier has a negative `notify_count`, this indicates that one or more `notify()`’s has been processed before the `set_ref_counts()` operation.

Similarly, if a barrier has a negative `wait_count`, this indicates that one or more `wait()`’s has been processed before the `ref_counts_valid()` operation.

```
Barrier class:
    self.notify_count       = 0         # signed integer
    self.wait_count         = 0         # signed integer
    self.ref_counts_valid   = False

    def set_ref_counts(self, notify_count, wait_count):
        # sets the expected number of references (notify and wait)
        # by incrementing the variables by specified amounts
        self.ref_counts_valid = True
        self.notify_count     += notify_count
        self.wait_count       += wait_count

        # incrementing ref counts may complete the barrier
        # if notifs already done!
        if self.notify_count == 0:
            release_waiting_consumers(self)
        if self.notify_count == 0 and self.wait_count == 0:
             # implies all notifications and all waits have already posted
             self.free(self)

    def notify(self):
        self.notify_count -= 1
        if self.notify_count == 0 and self.ref_counts_valid:
            release_waiting_consumers(self)
            if self.wait_count == 0      # all consumers are done too
                self.free(self)

    def wait(self, consumer):
        self.wait_count -= 1
        if self.notify_count == 0 and self.ref_counts_valid:
            consumer.go()
            if self.wait_count == 0:
                self.free(self)
        else:
            consumer.set_waiting(self)

    def release_waiting_consumers(self):
        # scan through all of the engines to release any threads
        # which were waiting on this barrier
        for consumer in self.all_consumers:
            if consumer.is_waiting():
                consumer.go()

    def free(self):
        self.ref_counts_valid = False
```

11.6.3 Barrier Tables

Pyxis implements **four** Barrier Table instances: **one Barrier Table per cluster**, supporting the UIEs, HBM-AMEM DMAs in that cluster, as well as Inter-cluster DMAs and Tx RDMAs sourced from that cluster, and Rx RDMAs received at that cluster. Each cluster's Barrier Table supports **1024** barrier objects**.**
Barrier_id 0 is not used and reserved as a no-op indicator.

```
bt_barrier_t   cluster_barrier_table[1024];
```

Each Barrier Table implements the `set_ref_counts()`, `notify()`, and `wait()` primitives atomically.

11.6.4 Barrier_ID Cluster Mapping

Barrier_ids in the trip descriptor `notify_list` include a 2-bit cluster ID as a prefix:

barrier_id[11:10]	**barrier_id[9:0]**
Cluster index	barrier index in cluster's barrier table

Table 11-3: Barrier_ID format for notify() Operation

The cluster ID is an absolute (target) cluster number for barrier notifications, regardless of which is the source cluster number.

For barrier `wait(barrier_id)` and `set_ref_counts(barrier_id)`, the operation is always local to the cluster, so the 2-bit cluster ID prefix is not required or included

11.6.5 Barrier Interrupts

Barriers may be configured to signal an interrupt to the on-chip CPUs, when a barriers' `notify_count` is decremented to zero.

Hardware provides a **barrier_interrupt_config** table, programmable by software, with 8 entries. Each entry specifies a barrier ID and a valid bit. When the specified barrier's notify_count is decremented to zero, this triggers the entry's associated interrupt.

Note that each cluster has its own barrier table and associated set of 8 interrupts to the CPU complex.

```
struct barrier_interrupt_config_t {
   uint   interrupt_valid  : 1;
   uint   barrier_id       : 10;
};

barrier_interrupt_config_t cluster_barrier_interrupt_config_table[8] ;
```

Note that when a barrier is selected to signal an interrupt, it may also be configured with a `wait_count` which includes one reference for the CPU as a consumer. When software has seen and responded to this barrier, it may then decrement the `wait_count` for that barrier via a memory-mapped register access to the Barrier Table.

11.6.6 Software Access to Barrier Table

Hardware provides a special memory-mapped access to the Barrier Tables for performing atomic updates to the barrier state. This is the `bt_mem_decrement` array target (in each cluster). Software may write to this array to atomically decrement either the notify_count or wait_count of a particular barrier. Using C struct-like nomenclature, this can be defined as

```
struct bt_barrier_decr_cnt_t {
   uint   wait_decr    : 1;   // Decrement wait count by 1 if set
   uint   notif_decr   : 1;   // Decrement notification count by 1 if set
   uint   trip_id      : 20;  // Trip ID for tracing purposes
};
```

And the target is

```
bt_barrier_decr_cnt_t bt_mem_decrement[1024]
```

`bt_mem_decrement` is a **write-only** memory-mapped array

- Writing to `bt_mem_decrement[barrier_id]` with `notif_decr==1` will cause that barrier's `notify_count` to be decremented, and triggers any subsequent hardware operations which occur as a side effect

  This operation may be used to trigger dependent trips waiting for software.

- Writing `bt_mem_decrement[barrier_id]` with `wait_decr==1` will cause that barrier's `wait_count` to be decremented by 1, and triggers any subsequent hardware operations which occur as a side effect

  This operation may be used to free a barrier after responding to an interrupt.

NOTE: **only one** of `wait_decr` or `notif_decr` fields may be nonzero for each write to `bt_mem_decrement` array. Software must not decrement both counts in the same access.

Hardware also provides memory-mapped access to the Barrier Table (in each cluster) to allow software to **directly read or write** any barrier. Note that *writing* barriers is **not** a normal operation since barrier state is maintained by hardware using atomic increment/decrement operations, so direct software writes to barriers must be done under controlled circumstances. One such use case is for cleaning up barriers after a flush when in a known quiescent state. Writing a barrier does *not* trigger hardware operations (such as enabling trips or relaxing barriers) as a side-effect using this view into the barrier table:

```
bt_barrier_t   bt_mem[1024];
```

11.6.7 Barrier Timeouts

For debug purposes, barriers may be configured with a timeout. This uses a coarse two-pass marking system. Periodically, a hardware process will walk through the entire Barrier Table, and inspect and "mark" all valid barriers. When a barrier is completed (all reference counts returned to zero), the mark is cleared by hardware. If the timeout process sees the mark still set on a subsequent pass through the table, this can infer a timeout, and an interrupt is set and the barrier_id is saved in a software-readable register for the first timeout detected. It is up to software how to handle the timed-out barrier.

The barrier timeout process may be configured to walk the barrier table once per N cycles, where N should be set as a large enough to ensure that all barriers should have been cleared within that time.

11.7 Data Ordering and Consistency Rules

- AMEM writes from UIEs and DMA engines are posted, with indefinite completion delays. Subsequent reads are subject to read-after-write hazards if the write has not been retired to memory, and write-completion tracking mechanisms (described below) are introduced to protect against this hazard. When a second trip is dependent on data written to AMEM by a first trip, a barrier must be used between the two trips.
- AMEM writes from different clients (UIEs and DMA engines) can complete in any order. If writes are non-overlapping (at a 16-byte granularity – see next bullet) there is no interference or hazard. If writes from two clients are overlapping, then a barrier must be used between their trips to enforce the desired order.
- AMEM read-modify-writes are NOT atomic. AMEM hardware cannot ensure that two read-modify-writes to the same address cannot be intertwined, and cannot prevent a write from being inserted between the read and the write portions of RMW. This is an important consideration for ***partial writes***, in which a UIE is programmed to write a subset of the 16 bytes in an AMEM partition. (See section 7.4)

  Internally, AMEM converts Writes into RMW partial-writes on a partition-by-partition basis. Only those partitions which write fewer than 16 bytes will need RMW. For these partitions, hardware first reads the 16-bytes from AMEM, replaces some of the bytes as required, and then writes back the full 16 bytes.

  Because the RMW is not atomic, AMEM hardware cannot ensure that two read-modify-writes to the same address cannot be intertwined. Therefore, we introduce the following restriction:

  If two partial-writes overlap on any 16-byte boundary, they must be separated by a barrier, even if there is no byte-level overlap

  This restriction does not apply if two writes (or partial writes) are accessing the same AMEM word (128 bytes) but disjoint partitions within the word.
- HBM write completion and hazard avoidance is provided natively by the HBM controller IP. Once writes have been delivered to the HBM controller, subsequent reads issued to the HBM controller will return the most recently “written” data.

11.7.1 UIE Write Completion Tracking

At the end of each trip, once the UIEs write results into AMEM, they will typically "notify" a barrier which signals a waiting trip (or DMA transfer) that the data is available in AMEM and may be read and used by the dependent trip. However, since writes are *posted* to AMEM, we provide the following mechanism to ensure write-completion to avoid a race where data is read before the write completes.

- Each UIE includes a 3-valued **Tracking ID** as metadata with every write sent to AMEM. The Tracking ID increments with each trip, so the first trip uses Tracking ID=0, 2nd trip uses ID=1, etc.

This is managed by hardware. The Tracking ID counts 0-1-2-0-1-2, etc., and is incremented with each trip.

- Each UIE maintains a **PendingWriteCount** per Tracking ID. This count is incremented with each write posted by the UIE to AMEM, and is decremented as each write is committed to AMEM such that any subsequent Read would see the latest data. These decrements, along with associated Tracking ID, are signaled by AMEM.
- Each UIE also maintains a trip completion **scoreboard** for each trip, which waits for all Sequencers to post "done" *and* for the trip's PendingWriteCount to be zero. At this point the UIE signals “done” to the Trip Manager, which can notify the barrier(s) safely.

As each UIE trip is launched, it must allocate a completion-tracking scoreboard entry, and multiple active scoreboard entries cannot have the same Tracking ID to avoid aliasing. This requires 3 Tracking IDs (and 3 scoreboard entries) per UIE because:

- A UIE will often have two active trips in various stages at the same time. For example, the AMEM Read Sequencer may have started on trip N+1 while the Write Sequencer is still busy with trip N; and
- the write-completion tracking may still be working on trip N-1.

The scoreboard can continue to monitor the PendingWriteCount after all the sequencers are done and have moved on to the next trip, so they are not stalled waiting for the write completion.

11.7.2 DMA Write Completion Tracking

The AMEM write-completion scheme described above is also used for DMAs writing to AMEM.

For **HBM to AMEM DMA**, and **Inter-Cluster DMA** transfers: hardware will include a Tracking ID with each AMEM write (per 128B flit). This requires only a small number of Tracking IDs per DMA channel, to cover a small number of outstanding writes from (otherwise-complete) DMA jobs at any time.

- Each DMA engine implements a **PendingWriteCount** register per Tracking ID. This can be loaded by the engine at the start of a trip based on the total number of flits to be written to AMEM.
- Each flit-write to AMEM includes the Tracking ID.
- AMEM will signal write-commit back to the engines as a per-Tracking ID vector. The vector bits decrement the associated PendingWriteCount
- When PendingWriteCount is decremented to zero, the DMA engine may signal “done” to the Trip manager. This must be sent in trip order (i.e., in the order the trip descriptors were received from Trip Manager).
- The Trip Manager will notify completion (to the DMA job’s dependent barriers) once Done has been received.

For Fabric **Rx RDMA** writes to AMEM: since there may be many concurrently active Rx RDMA Jobs, it is too expensive to maintain individual PendingWriteCount trackers for every Rx Job individually. However, each received page is written as a series of contiguous requests - i.e., each received page is transferred to AMEM in its entirety before the next page is transferred (per cluster). So, each page can appear as a mini-DMA for completion tracking.

- The RDMA Rx Jobs share a set of Tracking IDs (and associated PendingWriteCounts), per cluster RxDMA block. When an assembled page is received from the fabric, it must allocate a Tracking ID before the data may be written to AMEM. If no trackers are available, the Rx RDMA will stall.

  Due to the occasional small page, this requires more than two or three Tracking IDs: in a worst-reasonable case, the Rx Fabric Adapter logic might receive one small page from many communicating Tx chips in succession. Given a targeted minimum processing time of 8 cycles per small page, then with a ~100 cycle write completion latency, a set of **16** Tracking IDs is sufficient to avoid bottlenecks due to exhausting the Tracking IDs.
- The Rx Fabric Adapter hardware will maintain the PendingWriteCounts for each Tracking ID locally in the **RxDMA** block (see section 10.8.8) for the receiving cluster. As each page's PendingWriteCount is decremented to zero, RxDMA reports this to the **Rx Job Manager** (section 10.8.1) for that cluster.
- Note there may be many outstanding Rx RDMA jobs and they may complete in any order. A full Rx RDMA job is complete when all of its pages have been received and marked as fully written to AMEM. When the last page has been fully committed, Rx Job Manager then notifies the Rx RDMA job's dependent barriers directly.

11.7.3 CPU and PCIE Write Tracking for AMEM

AMEM provides an AXI-4 (slave) interface for CPU and PCIE initiated writes to AMEM. AMEM keeps track of the outstanding writes per burst transaction, and the AXI-4 interface logic will not signal the write completion "B" response until all writes for a given burst have been committed.

11.8 TRIP FLUSH

Communications errors on the fabric may occur, and Pyxis hardware has multiple layers of retry to recover from these errors. However, in some cases, recovery may not be possible, e.g., due to failing (or imminently-failing) hardware. An example would be a failing fabric link.

In these cases, existing AI inference jobs cannot be completed, and some higher level recovery may be required, such as disabling a faulty fabric link and restarting the inference tasks. When such a failure occurs, multiple Pyxis chips may stall waiting on barriers which will never be relaxed due to the failure of a fabric RDMA job. There may be many trips already queued up, with many barriers set up to manage the sub-graph dependent on this (failed) RDMA.

To avoid having to reset the affected hardware, Pyxis provides a mechanism to flush the trip descriptor queues. This allows the descriptors and barrier dependencies to unwind, allowing the UIEs and DMA engines to return to a safe, quiescent state:

- When the Fabric Tx or RX RDMA Job Managers encounter an error which is not repairable with retry (or after a failed retry), they enter a **Job Failed** state for the affected RDMA job(s). (See section 10.3). This sets an interrupt to software running on Pyxis CPUs. The Job Failed state is sticky until released by software. In their respective Failed states, Tx will no longer be sending data for that job ID, and Rx will drop any packets received from the fabric for that job ID.
- Software then sets a global **Trip Flush mode** register in the Trip Manager (in each cluster), via a CSR access. This is described below.
- Software can then release the RDMA job (or multiple jobs) from the failed state. In case multiple RDMA jobs have failed, hardware provides a software-visible vector register of all jobs in the Failed state. Software can read this register to find all the failed jobs, and write a related register to clear the sticky failed state and allow the jobs to proceed. Once free, these (failed) RDMA jobs may go directly to the Idle state. On the Rx side, any subsequently arriving fabric data will be discarded.

  Software may also set a global CSR which inhibits the sticky Failed state, so additional failing RDMAs can go directly to the Idle state. This avoids repeatedly interrupting the CPUs to handle additional failing jobs stemming from a single fabric hardware problem.

The Trip Flush mode affects any trip which has not yet passed its barrier-wait in the Trip Manager. In this mode, all trips are effectively canceled:

- The Trip manager will discard all entries in the descriptor queues, neither waiting on barriers nor notifying successor barriers for these trips. Any trips which were waiting on barriers are also discarded - the barrier wait is canceled and the trip is canceled without signaling the engines to process anything.
- This will cause the descriptor queues to flush. The descriptor Prefetch engine will continue to fetch the descriptors from memory-based descriptor queues into the Trip Manager (where the trips will be discarded) until the descriptor queues are empty.

- Any trip which was already past the barrier-wait at the time of setting Trip Flush will be allowed to complete normally. It is implicit that such trips were **not dependent on the failing RDMA**(s), so it is safe for those trips to proceed normally. **The key ordering step here is for software to set the Trip Flush mode before releasing the failed RDMA job(s)**.

Software must monitor for descriptor queues to be empty and for the Trip Manager to have zero outstanding trips. The fabric-adapter's RDMA Job Tables must also have zero active RDMA jobs (all jobs in Idle state). The Trip manager and fabric Tx/Rx Job Managers make this status visible to software via CSRs. Note that this operation is required in each cluster.

Once the trip-flush is complete, the descriptor queues will be empty, all engines (UIEs, DMAs and RDMAs) will be idle and in quiescent states. The Barrier Table is also cleared and re-initialized by hardware as part of the trip-flush process.

It is likely that multiple communicating chips will experience this same failure state - once we flush pending jobs, the RDMAs will all fail. Software may need to flush all of the communicating chips, and subsequently reinitialize the RDMA Job Tables to re-synchronize among the communicating chips (for example, to establish the same Job-ID epoch on Rx and Tx sides for each job-ID).

This provides a clean point where the queues and engines are all properly flushed and idle. This should allow the chip to be reconfigured as needed to work around any hardware failure, and a new workload can be deployed.

12 Specialized DMA Engines

12.1 HBM – AMEM DMA Transfer

Pyxis provides two DMA engines for transferring high-bandwidth data between HBM and AMEM, one engine for HBM->AMEM and one for AMEM->HBM transfer. There is one instance of each engine per cluster. Each engine has a single channel, optimized for bulk transfer of tensor data. The nominal bandwidth of each transfer engine is up to 1.36 TBytes/sec, which is more than sufficient to saturate the HBM device bandwidth.

HBM DMA jobs are defined using descriptors, following the usual trip-descriptor semantics with barriers for dependency controls:

- HBM->AMEM DMA ("H2A") descriptors include the source base address in HBM, the destination base address in the cluster's AMEM, and the transfer length.
- AMEM->HBM DMA ("A2H") descriptors include the source base address in AMEM, destination base address in HBM, and transfer length.

All HBM DMA transfers are in units of **1024** bytes.

The base address for HBM is aligned on **1024**-byte boundaries, and the base address for AMEM is aligned on **128**-byte boundaries.

Each 1024-byte unit of data transfer corresponds to 64 bytes on each of 16 HBM channels, and also corresponds to 8 sequential 128-byte AMEM words. This uses the "tensor-mode" address mapping for HBM, where data is effectively striped across the HBM channels on an 8-bytes-per-channel basis (See section 8.2 above).

- For AMEM-to-HBM transfers, hardware collects 8 bytes from each of 8 sequential AMEM words, per HBM-channel, to form the 64-byte burst written to that channel in AMEM.
- For HBM-to-AMEM transfers, hardware splits each 64-byte burst (per HBM channel) into 8 different AMEM words, and collates the per-word 8-byte pieces from each of the 16 channels to assemble each AMEM word.

For best efficiency, H2A and A2H DMA jobs should transfer at least 100K bytes of data (per trip descriptor). Smaller transfers, down to 1K bytes, are possible but will operate at much lower efficiency. Also, both of the H2A and A2H DMA transfers may be concurrently active (for a given cluster) but this creates more than 2x the bandwidth demand sustainable by the HBM device. In that case, the H2A and A2H will share access to HBM according to the read/write sharing bias configured for the HBM controller block.

12.2 Inter-Cluster DMA Transfer

Inter-Cluster DMA (ICMDA) engines provide AMEM-to-AMEM DMA transfer. There are 3 ICDMA engines per cluster, rigidly assigned to transfer data from AMEM in the local cluster to the AMEM in each of the other 3 clusters. Each engine has a single channel for bulk DMA transfer from the local AMEM to one other cluster's AMEM. There is no ICDMA engine for moving data within a cluster

ICDMA jobs are defined using descriptors, following the usual trip-descriptor semantics with barriers for dependency controls. The descriptors include source base address (in the local AMEM) and destination base address in the target cluster's AMEM, and the transfer length. All DMA transfers are in units of 128-byte AMEM words, with addresses aligned on 128-byte boundaries.

The three ICDMA engines plus Tx RDMA (to-Fabric) share the 4 AMEM core-switch-client Read interfaces corresponding to 4 CoreSwitch planes. At the destination AMEM, the four switch planes carry data from the three source clusters' ICDMA data and the Rx RDMA (from-Fabric). The four planes each access exactly one of the 4 AMEM core-switch-client Write interfaces.

ICDMA engines are assigned as shown in the following table

ICDMA Engine Cluster	ICDMA engine instances in cluster	Target AMEM clusters
0	0,1,2	1,2,3
1	0,1,2	0,2,3
2	0,1,2	0,1,3
3	0,1,2	0,1,2

Table 12-1: ICDMA engine to target cluster mapping

12.3 Tx RDMA Read (from AMEM to Fabric)

- One engine per cluster, per fabric-output (FO) pipe (See section 10.7). Any FO pipe can request data to be read and transferred from any of the clusters. Each page-transfer request includes the transfer size (in AMEM words), starting AMEM word-aligned address, and starting TX Buffer index for the receiving FO Pipe.
- The Tx RDMA engine generates the AMEM read requests, at up to one AMEM word (128B) per cycle per FO Pipe. In case multiple FO pipes are requesting data from the *same* cluster's AMEM, this allows for supplying data to all of the FO pipes concurrently from a single cluster. Note: this may be constrained by total bandwidth demand on the AMEM read ports which are shared by ICDMA and Tx RDMA, and in the worst case, Tx RDMA will only receive it's fair share of AMEM bandwidth (i.e., a total of one AMEM word per cycle) to be shared among the FO pipes.

12.4 Rx RDMA Write (from fabric to AMEM)

This is a sub-module in the RX Fabric Adapter. See section 10.8.8, "Rx DMA Transfer Engine".

12.5 Descriptor Prefetch Engine

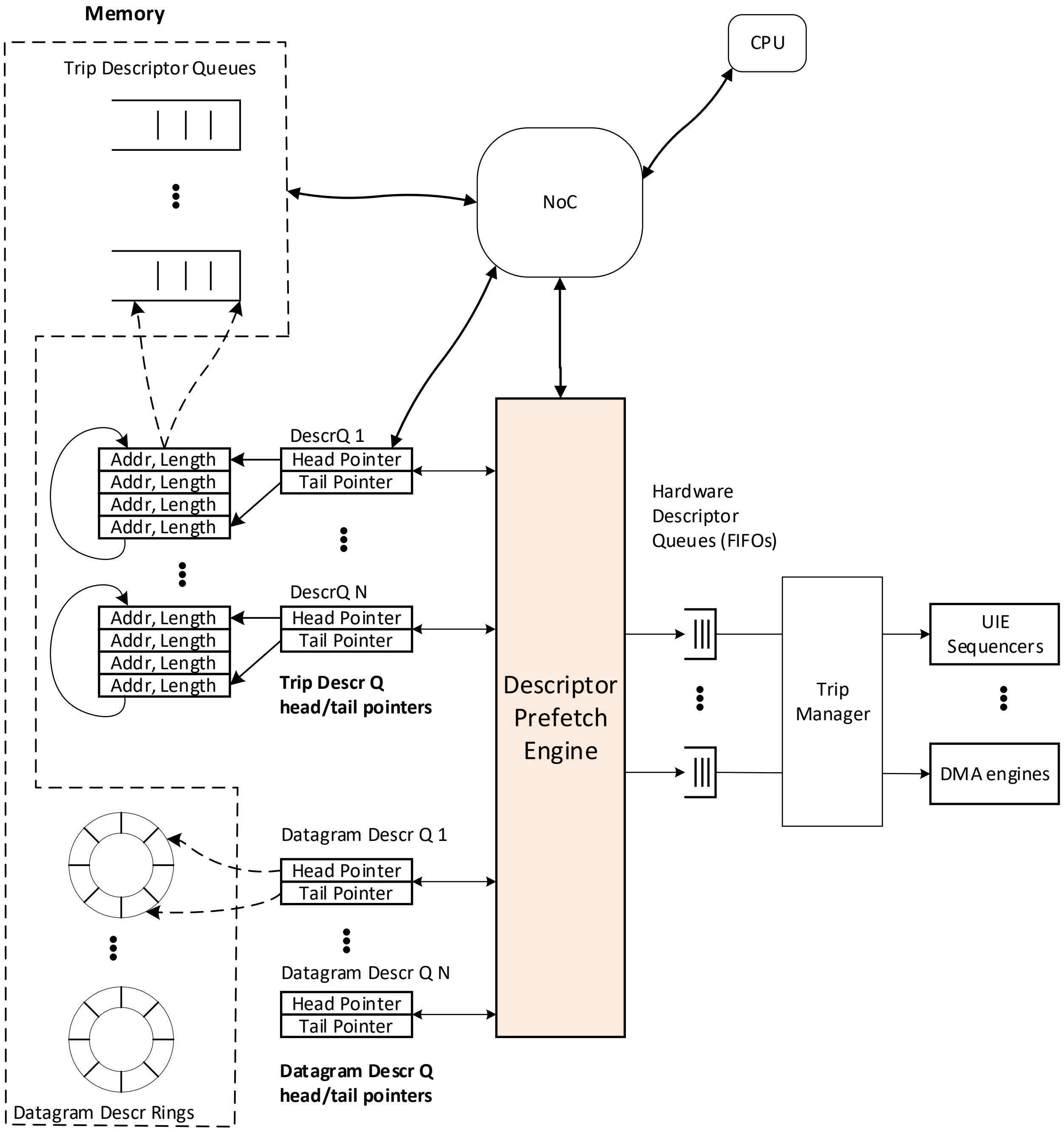


Figure 12-1: Descriptor Prefetch Engine

Each engine (UIEs, DMAs, etc.) has a shallow trip-descriptor prefetch-queue in hardware, backed by the deeper queue in memory described in section 11.5 above. Software prepares descriptors for each queue as lists of descriptors in memory, and enqueues each descriptor list to a given queue by writing the (queue id, list-address, list-length) tuple of the list to an “enqueue” CSR target.

Hardware monitors the availability of entries in the queues, and the **Descriptor Prefetch Engine** (DPE) prefetches descriptors from memory to keep the (shallow) hardware prefetch queues nonempty.

The DPE continually scans the head & tail pointers to look for non-empty queues. When a valid descriptor list is found in a queue’s ring, the DPE reads the head pointer to get the address and length of the list in memory. It then proceeds to prefetch descriptors from this list into its prefetch queues. This transfer is performed by the Network-On-chip (NoC) which relays the request to the appropriate memory, and returns the data from the memory to DPE.

When DPE has consumed all descriptors in a list, it advances the queue’s ring Head pointer and signals a Descriptor Queue Dequeue Event (as an interrupt) to the CPU.

For processing the lists for each queue, DPE can be configured with “hungry” and “starving” levels for its prefetch queues, with two priority levels for issuing the prefetches to replenish the prefetch queues. When a prefetch queue occupancy falls below the “hungry” level, and the in-memory portion of the queue is not empty, the DPE will fetch a (configurable) burst amount of descriptor data from the in-memory queue. If the prefetch queue occupancy falls below the “starving” level (and the in-memory queue is not empty), DPE will promote the prefetch to high priority (giving the starving queues higher priority than hungry queues to issue the prefetches read requests to the NoC).

All prefetches are in units of 8-bytes, and all trip descriptors are also sized in units of 8 bytes, and must use 8-byte alignment in memory. The “hungry” and “starving” levels are global configurations for all queues.

The DPE also prefetches descriptors for the **Datagram Tx** and **Rx queues**. These queues are organized in more-traditional descriptor rings in memory, which are managed jointly by hardware and software. Operation of the Datagram descriptor queues is described in section 11.5.2.1 and 11.5.2.2.

12.6 MICROCODE PATCH ENGINE

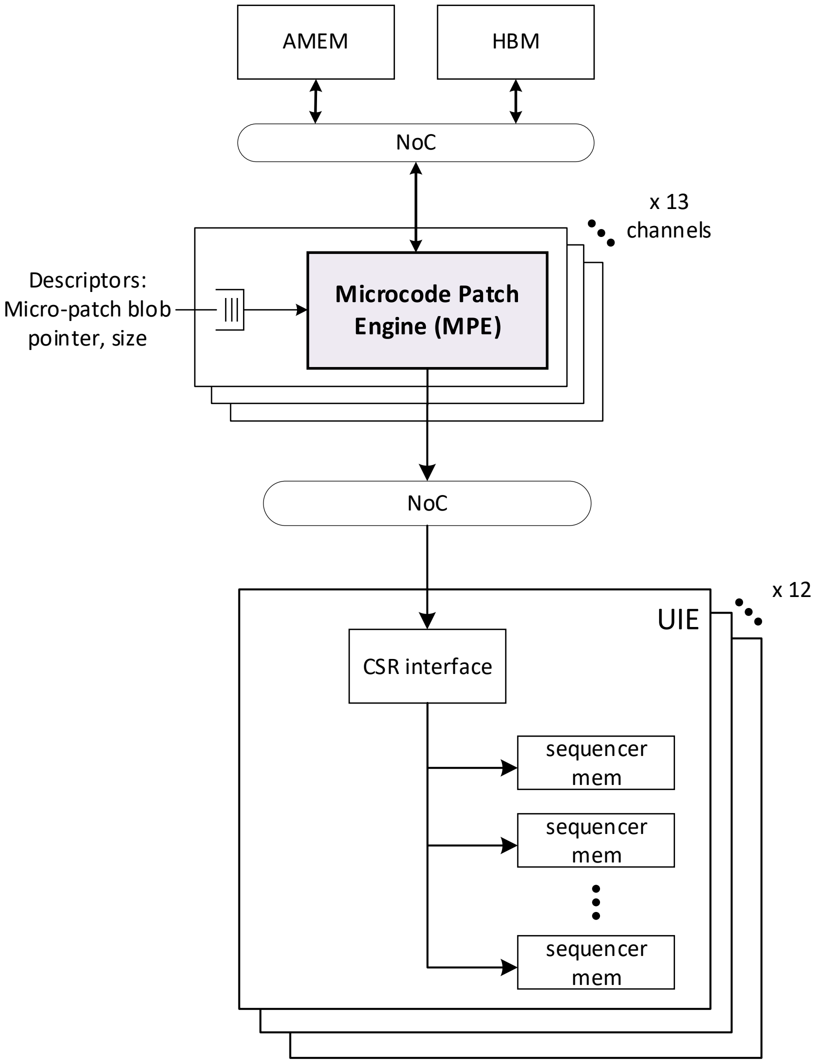


Figure 12-2: Microcode Patch Engine

The Microcode Patch Engine (MPE) is a specialized DMA transfer engine for applying many small patches to UIE microcode and configuration tables, or performing DMA transfers from one memory to another. There are 13 MPE instances per cluster with one thread per MPE.

The MPE has two modes of operation: “Patch” mode and “DMA” mode, as specified in the trip-descriptor.

The MPE tasks are invoked by writing a descriptor to a memory-mapped hardware queue. The descriptor includes the Patch Blob or DMA transfer descriptor, and the notify_list and wait_barrier_id as described in section 11.4 above.

In normal operation, each `MPE` will wait on a barrier which prevents the MPE from apply its patches too early (e.g., to avoid overwriting microcode which is still in use). After applying the patches, the MPE will then notify barriers for engines which may be waiting for the patch to be done.

12.6.1 MPE Patch Mode operation

For **Patch Mode**, the MPE can be programmed to read a patch ***blob*** from anywhere in its cluster's memory, using the NoC. This blob is a list of patches, to write registers or memory locations within the chip. Each patch includes the target address, register size and repeat, and the data to be written (in-line). The patches may be applied to any memory-mapped address, so this may patch microcode (in Sequencer memories) or configuration tables in the UIEs, or Control/Status registers (CSRs), or even the trip descriptors in memory.

The following describes how the patch descriptors must be prepared for use by MPE

12.6.1.1 MPE Patch-mode Descriptor Format

The (trip) descriptor provides the starting address of the blob in the (cluster-local) NoC address space, the number of patches contained in the blob and the length of the blob. As the blob data is received, it is interpreted, and the patch data is written via NoC accesses to the targeted CSRs or any memory-mapped target.

```
struct mpe_patch_blob_descr_t {
   uint32_t  blob_address;    // pointer to blob in memory. 8B aligned, byte address
   uint16_t  blob_length;     // length of the blob, in units of 8 bytes, N-1 encoded
   uint16_t  num_patches;     // number of micro_patch_t's in the blob, N-1 encoded
   uint16_t  reserved;        //
   uint16_t  mode;            // Set to 0x0000 for Patch mode
   uint32_t  pad;             // pad up to multiple of 64bits
};
```

The blob_address is remapped from 32 bits to the full NoC address space, but must be cluster-local. The remapping is described in section 12.7. This is byte-granular address, but it must be 8-byte aligned.

12.6.1.2 Micro_patch Format

```
struct micro_patch_t {
   uint32_t  target_address;  // starting address to write. 8B aligned, byte address
   uint16_t  reg_size;        // length of each register element in bytes. N-1 encoded
   uint16_t  reg_repeat;      // repeat (count) of each register element. N-1 encoded
   uint64_t  data[];          // variable length series of data words
};
```

Figure 12-3: Micro Patch format

Each patch (micro_patch_t) has a header providing {target_address, reg_size, reg_repeat}.

- The **target_address** is remapped from 32 bits to the full NoC address space, but must be cluster-local. The remapping is described in section 12.7.

- The **reg_size** indicates the size of the specific register (or register-like) entity to be written.

  Note: Hardware imposes a maximum reg_size of 32 bytes for CSRs.

- The **reg_repeat** value allows one patch to update multiple sequential registers or memory-mapped structures such as Sequencer Microcode memories. For example, to patch (rewrite) N microcode instructions in a particular sequencer, the patch can specify the memory-mapped address of the first microinstruction, the microinstruction word size (in bytes), and the number (N) of microinstructions to be written, followed by the data for the N microinstruction words.

After the patch header, the source data is included in-line, as an array of 64-bit (8-byte) elements. The source data must be stored ("at rest" in memory) according to the following rule:

```
Patch data[] size = reg_repeat * (reg_size rounded up to multiple of 8 bytes)
```

Software must pad the MSBytes of each register value to a multiple of 8 bytes, if necessary.

12.6.1.3 Patch Blob Format

```
microcode_patch_t blob[]; // list of microcode patches to apply
```

The **blob** is constructed by concatenating micro_patches in memory. All patches are a multiple of 8-bytes in length, and the entire blob must be **aligned on an 8-byte boundary**.

12.6.2 MPE DMA Mode operation

The MPE may be instructed to perform a simple block transfer, with a descriptor which specifies the source and target addresses and the transfer length. This may be used for block transfers from AMEM or HBM, to AMEM, HBM or other memory-mapped structures on the chip, with dependencies managed by the hardware barrier mechanism.

12.6.2.1 MPE DMA-mode Descriptor Format

```
struct mpe_dma_descr_t {
   uint32_t  source_address;  // starting address to read. 8B aligned, byte address
   uint32_t  target_address;  // starting address to write. 8B aligned, byte address
   uint16_t  length;          // transfer length in 8-byte words. N-1 encoded.
   uint16_t  mode;            // Set to 0x0001 for DMA mode
   uint32_t  pad;             // pad up to multiple of 8 bytes
};
```

The source_address and target_address are remapped from 32 bits to the full NoC address space. The remapping is described in section 12.7. Addresses are byte-granular, but must be 8-byte aligned.

12.6.2.2 MPE DMA-mode Address Alignment and Length Restrictions

The DMA-mode source_address and target_address must be 8-byte aligned, i.e., the 3 LSBits of each address must be zeros.

12.6.3 MPE Instances

There are 13 MPE instances (“channels”) per cluster. This nominally provides one MPE per UIE, plus one extra MPE (per cluster) for general patching tasks. The MPE channels in each cluster share the NoC access port, but have independent descriptor queues, and therefore operate independently.

This organization is chosen for the following reasons:

- MPEs need to wait on barriers which prevent overwriting a UIE’s microcode before the UIE is done using that microcode, and later, notify a barrier for the UIE which allows it to use the newly patched-in microcode. By having an independent MPE for each UIE, and a separate descriptor queue for each MPE, this keeps each UIE<->MPE pairs’ dependencies in a separate thread. If one UIE is executing a different, possibly slower workload, this will not block other UIEs.
- The process of applying patches may be serialized to a given UIE due to the limited throughput available for updating microcode in a given UIE. By assigning different UIEs’ patching tasks to different MPEs, the patches can be applied concurrently to multiple UIEs.

The MPEs are not hard-mapped to UIEs, and software may choose to assign any patching workload to any MPE within the cluster.

The 13th MPE in each cluster may also be used as a generic patching thread. This may be used to apply patches to memory outside of the UIEs or to perform generic DMA block transfers.

12.7 Address Remapping for Microcode Patch Engine and Descriptor Prefetch Engine

Some specialized engines on Pyxis have limited view of the Pyxis address space, restricted to 32 bits, for accessing descriptors, patch blobs and other control information.   This applies to the Descriptor Prefetch Engine (DPE) and Microcode Patch Engine (MPE).   See sections 12.5 and 12.6 for the descriptor address formats of these engines.

In order to access control information anywhere in the Pyxis memory map, including all of HBM memory, the 32-bit address can be re-mapped to provide four 30-bit (1Gbyte) windows into the full NoC-addressable address space.

The top-two bits of the address are used to select a base register and mask (from a set of four) The selected base and mask are then applied to the remaining 30 bits of the provided address, with the mask used to select which bits are taken from the selected base address and which are passed through as-is. The procedure is described with the following verilog pseudocode:

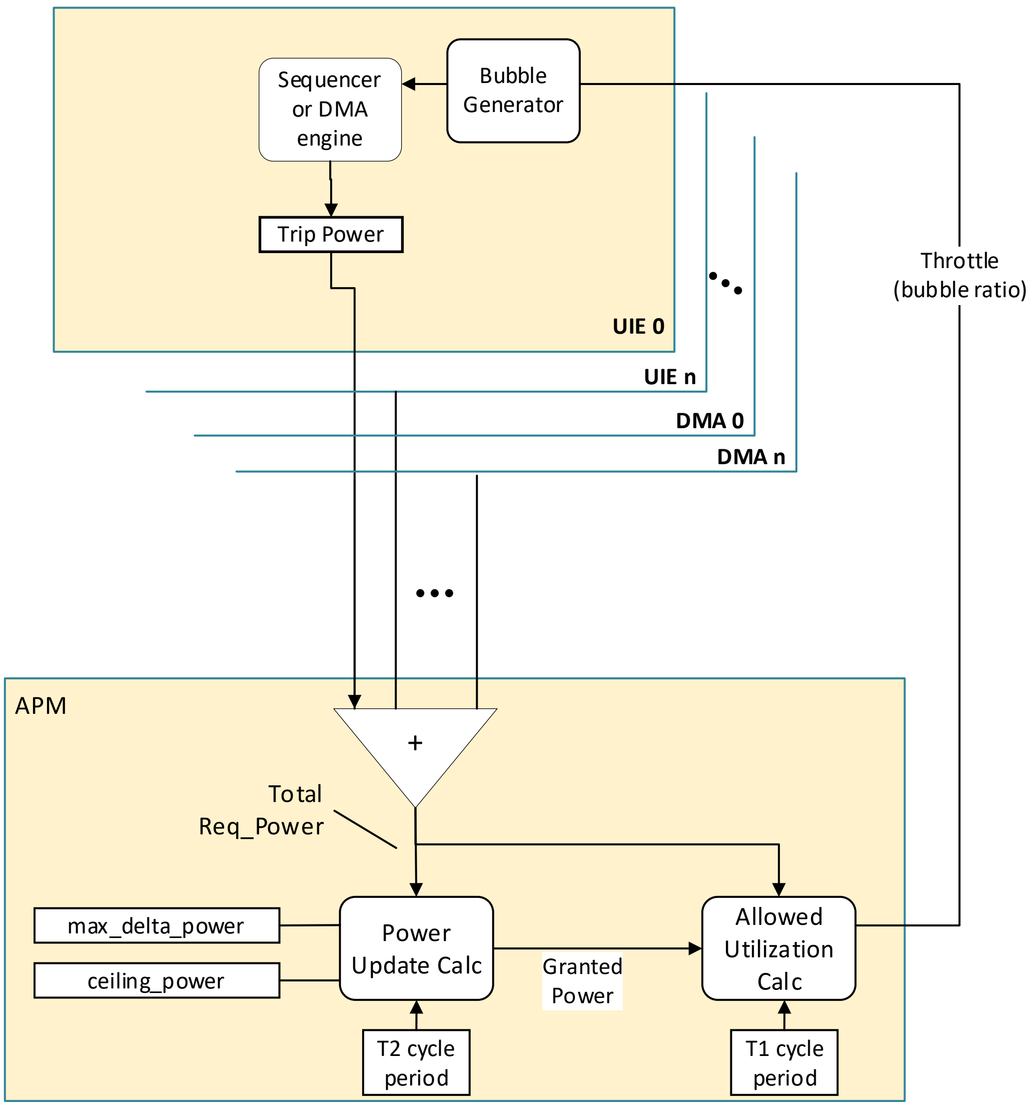
```
parameter noc_addr_w = 39;   // Full NoC address width

function void eng2noc_addr_remap (
    input  logic [31:0] eng_addr,
    input  logic [noc_addr_w-1:0] noc_addr_base[0:3], // 4 windows
    input  logic [noc_addr_w-1:0] noc_addr_mask[0:3], // 4 windows
    output logic [noc_addr_w-1:0] noc_addr
);

    // select the base & mask from the set of four
    addr_base[noc_addr_w-1:0] = noc_addr_base[ eng_addr[31:30] ];
    addr_mask[noc_addr_w-1:0] = noc_addr_mask[ eng_addr[31:30] ];

    // apply base & mask
    noc_addr =  (eng_addr[29:0] & addr_mask) | (addr base & ~addr_mask);

endfunction
```

In an expected usage model, this provides one window for all of AMEM, one for CSRs, and two up-to-1GB windows into HBM. The window mappings will be statically configured by software.

There are independently configurable mapping tables for each of DPE and MPE, for each of the 4 clusters. Each mapping table has four entries as shown above.

Note: The 12 LSBits of `addr_mask` must be set to `0xFFF`. I.e., address bits [11:0] may not be remapped.

13 Active Power Management



The UIEs create very large dynamic power demand when operating at full rate. This can create very large load-current demand spikes on the power supplies with high slew-rate (large di/dt), particularly if the UIEs ramp up from idle to full operation in a very few clock cycles. The same applies for the DMA engines and HBM.

The **Active Power Manager** (**APM**) module smooths and limits the power ramp up rate. It operates by dynamically limiting the execution rate of selected sequencers in the UIEs, and ICDMA and HBM DMA engines, by inserting stalls (bubbles) into the sequencer / engine pipelines. Stalling the LLC Grid, VPU or AMEM data movement for a fraction of cycles linearly reduces the dynamic power demand.

The algorithm damps the change in load by continually adjusting the allowed engine pipeline utilization based on the recent load and requested load. Engine utilization rate is throttled by regularly inserting bubbles in the computed fraction of cycles.

Active Power Management uses a ***proxy*** for power which is configured for every trip. This proxy power represents the average power consumption for the trip operating at full rate. This value may be determined empirically based on lab measurements for different trip types.

The APM operation includes the following steps:

- The configured trip power is reported by every UIE and DMA engine to the central APM.
- APM sums theses values to determine a total RequestedPower at any given time
- APM also maintains a state of GrantedPower which is regulated to represent the characteristics of the power supply
- If RequestedPower > GrantedPower, APM computes the AllowedUtilization for all engines as GrantedPower/RequestedPower. This is then sent to all of the UIEs, and ICDMA and HBM DMA engines.
- The engines insert "bubbles" in their execution pipelines to reduce their execution rate, as needed, to meet the AllowedUtilization.
- APM gradually increases GrantedPower over time, to conform to the power supplies' capabilities for Delta Load Current / Time. Once the power has ramped up, Granted Power will exceed RequestedPower, and AllowedUtilization will be 100%. As long as the UIEs and DMAs maintain a fairly consistent power demand, within the power supply's short-term dynamic range capabilities, no reduction in execution will be required. However, if the engines are all idle or reducing their aggregate requested power over a period of time, APM will throttle down the GrantedPower and may restrict the engines again when they throttle back up.

APM is configured with a MaxDeltaPower and PowerUpdatePeriod, as well as MaxGrantedPower and MinGrantedPower settings. Each trip descriptor (for UIEs, ICDMA, A2H and H2A DMA engines) includes a TripPower field. All power units are normalized and fairly low-precision, since this is using power estimates only.

This is a very brief description. More details are available in the Active Power Manager specification.

14 WEIGHTS COMPRESSION

For LLMs with huge models, even a modest compression of weight parameters will yield cost savings in reducing both the HBM memory capacity and bandwidth required. Pyxis aims for up to 2x compression, converting 8-bit weights to **4-bit weight *index* values**, coupled with a “codebook” to convert the indices back to 8-bit weight values, and scaling factors.

14.1 BLOCK-WISE SCALING FACTORS WITH SHARED CODEBOOK

This scheme uses block-wise compression with 4, 8, 16, or 32 weights per block. Each block of weights has an associated LnsI4F3 *scaling factor*, and multiple blocks share one *codebook* of 16 LnsI4F3 base weight values.

The codebook has 16 LnsI4F3 entries, which would normally be programmed to take values between -1 and 1, with a fixed EB = **-15**. This maps the I4F3 dynamic range to **2^-15..2^0** , to represent the widest set of values between -1 and 1.

The scaling factors are one-byte LnsI4F3 values with user-defined EB. The decompressed weights will have the same user-defined EB as the scaling factors. Typically, all scaling factors for a tensor use the same EB, since all the compressed weights would resolve to the same EB per tensor.

The decompressed value for each weight is

```
weight = codebook[weight_index] * block_scaling_factor
```

Since these values are all LNS, the multiplication is performed with addition in hardware. The computation in the log domain is

```
weight_lns = codebook[weight_index] + block_scaling_factor_lns – 15
```

The resulting `weight_lns` value is in LnsI4F3 format, with the user-defined EB.

14.2 SUPER-BLOCK FORMATS

For hardware efficiency, the weights are organized into “superblocks” in memory. Each superblock has a header with **128 LnsI4F3 scaling factors** followed by **128 blocks of weight-indices**.

The first superblock must also be preceded with a 128-byte word consisting of the **16-byte codebook plus padding**. Subsequent superblocks may omit the codebook word from the header.

Note: Superblocks must be aligned in AMEM memory on 128-byte boundaries.

14.2.1 Block-size 32

Byte 127 … 15 0

Padding (112 bytes)							Codebook
Scaling factors 127:0							
Block 7 32 x 4b	Block 6 32 x 4b	Block 5 32 x 4b	Block 4 32 x 4b	Block 3 32 x 4b	Block 2 32 x 4b	Block 1 32 x 4b	Block 0 32 x 4b
…							
Block 127	Block 126	Block 125	Block 124	Block 123	Block 122	Block 121	Block 120

1st superblock only

127 … 47 32 31 16 15 0

Table 14-1: Superblock for shared codebook + per-block scaling factors: block size = 32

14.2.2 Block-size 16

Byte 127 … 15 0

Padding (112 bytes)														Codebook	
Scaling factors 127:0															
Blk 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Blk 0
…															
Blk 127															Blk 112

1st superblock only

127 … 23 16 15 8 7 0

Table 14-2: Superblock for shared codebook + per-block scaling factors: block size = 16

14.2.3 Block-size 8

Byte 127			…	15		0	
Padding (112 bytes)					Codebook		1st superblock only
Scaling factors 127:0							
Blk 31	30	29	…	2	1	Blk 0	
63	62	61	…	34	33	32	
95	94	93	…	66	65	64	
127	126	125	…	98	97	96	
127			…	11 8	7 4	3 0	

Table 14-3: Superblock for shared codebook + per-block scaling factors: block size = 8

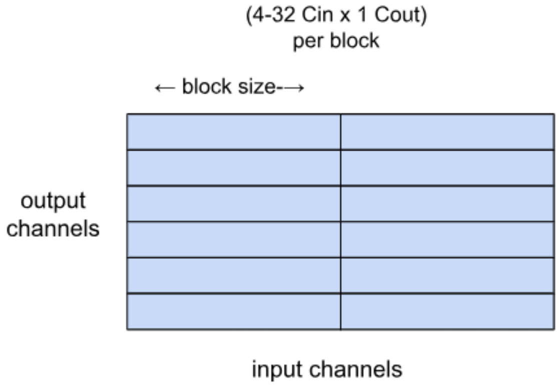
14.2.4 Block-size 4

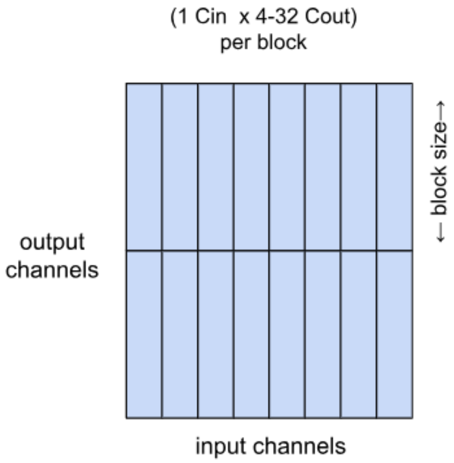
Byte 127			…	15		0	
Padding (112 bytes)					Codebook		1st superblock only
Scaling factors 127:0							
Blk 63	62	61	…	2	1	Blk 0	
127	126	125	…	66	65	64	
127			…	7 4	3 2	1 0	

Table 14-4: Superblock for shared codebook + per-block scaling factors: block size = 4

14.3 Mapping Weight-Compression Blocks to Channels

Weights are notionally 2D arrays, with shape `(output_channels, input_channels)`. Weight-blocks may be organized to span multiple input channels for one output channel, or for a single input channel spanning multiple output channels. Either organization is supported in hardware. This is illustrated below with the hardware-centric view with output channels in rows and input channels in columns. The blue rectangles represent the set of weights in a single compression block


(4-32 Cin x 1 Cout)
per block
← block size-→
output
channels
input channels


(1 Cin x 4-32 Cout)
per block
← block size→
output
channels
input channels

15 Weights Database in AMEM

All of the weight value needed to execute a neural network are stored in a database in HBM, and DMA-transferred to AMEM prior to use by the UIE. This section describes example formats of this database.

15.1 Weights Database: 3x3 Kernels

The 3x3 Kernels database holds the set of (3x3) kernels to be used for a given trip. To prepare the kernels for the database, software will **conceptually concatenate the kernels for each grid-row into a packed byte array**, in the order they will be used by hardware.

Each 3x3 kernel consists of nine 8-bit LnsI4F3 weight values, so each kernel requires 9 bytes. The 9 weights of the kernel are organized as follows, where “NW” is “north-west” weight position in the kernel, and “SE” is “south-east”, etc.:

71:64	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
SE	SC	SW	CE	CC	CW	NE	NC	NW

Table 15-1: 9-byte kernel format

The specific organization of kernels depends on how many filters (output channels) are mapped to each grid-row. In the simple case of a single output channel per grid-row, the kernel order is identical to the input channel order. For multiple output channels per grid row, the list of kernels is interleaved by output channel. Specifically, for N output channels per grid-row, and M total input channels (per trip), the kernel order is as follows, where `Ki,j` is the 9-byte kernel for Cin `i` and Cout `j`:

$\{K_{0,0}, K_{0,1}, K_{0,2}, \ldots K_{0,N-1},$
$K_{1,0}, K_{1,1}, K_{1,2}, \ldots K_{1,N-1,}$
…
$K_{M-1,0}, K_{M-1,1}, K_{M-1,2}, \ldots K_{M-1,N-1\}}$

This list of kernels may be viewed as a packed array of bytes, stored in little-endian format with $K_{0,0}$ in the LSB position. This requires a total of `M*N*9` bytes per grid-row.

Each grid-row’s array is then chopped into 8-byte *chunks*. If that is not a perfect multiple of 8, the data is MSB-padded up to the next multiple of 8 bytes (for each grid-row array).

The series of 8-byte chunks for a single grid row is illustrated below for the tiny example of 8 kernels/grid-row with M=4 and N = 2. This occupies 9 chunks exactly.

8Byte Chunk	bit 63	bit 0
0	kernel 0,0 [63:0]	
1	kernel 0,1 [55:0]	[71:64]
2	kernel 1,0 [47:0]	k 0,1 [71:56]
3	Kernel 1,1 [39:0]	kernel 1,0 [71:48]
4	Kernel 2,0 [31:0]	Kernel 1,1 [72:40]
5	Kernel 2,1 [23:0]	Kernel 2,0 [72:32]
6	Kernel 3,0 [15:0]	Kernel 2,1 [72:24]
7	K 3,1 [7:0]	Kernel 3,0 [72:16]
8	Kernel 3,1 [72:8]	

The 8-byte chunks from all of the grid-rows are then organized by concatenating the first chunk from each grid-row, then the 2nd chunk from each grid-row, etc., for grid rows to be used for the trip. In most cases, this will be 16 grid-rows, but it is possible to operate fewer than 16 grid-rows for smaller tensors.

If there are an odd-number of grid-rows used, the data is MSB padded with one pad-chunk (8 bytes of zeros) to make an even number of grid-rows. This is the format stored in the Weights Database in memory. This is illustrated below for a toy example with 5 grid rows used, plus padding to make (effectively) 6 grid rows.

3x3 Kernels Chunk (8B per chunk)	Grid Row	Weights Database bytes
0	0	0-7
	1	8-15
	2	16-23
	3	24-31
	4	32-39
	Pad with zeros	40-47
1	0	48-55
	1	56-63
	2	64-71

	3	72-79
	4	80-87
	Pad with zeros	88-95
2	0	96-103
	1	104-110
	…	…
…	…	…

Table 15-2: Example 3x3 Kernel Weights Database format

15.2 Weights Database: 1x1 / Linear, 8-bit weights

This database format supports 1x1 convolution (linear layers) with 8-bit weights using LnsI4F3 values.

To prepare this database, we conceptually create **per-output-channel** packed byte arrays consisting of the weights for all input channels for that output channel, in input-channel order. Each weight is one byte for LNSI4F3. If the number of *input* channels is not a multiple of 8, each array is zero-padded up to a multiple of 8 bytes.

The 1x1 weights (I4F3) database may then be constructed by concatenating the first 8-bytes from each output channel's array (i.e., for the first 8 input channels for each output channel), **for 128 output channels** in the output-channel-order illustrated in the table below. In this arrangement, the number of output channels must be padded up to a multiple of 128. (Other arrangements are possible for small output channel counts to avoid padding, but this is the most efficient arrangement for regularly-sized weights arrays)

Then the second 8-bytes from each output channel (i.e.., for the 2nd group of 8 input channels per output channel) are collected in the same swizzled order, and this pattern repeats for all groups of 8 input channels.

If there are more than 128 output channels, this entire pattern is repeated for each set of 128 output channels. However, each UIE grid computes only 128 output channels at a time.

Input channels	Output Channels	Weights Database bytes
0-7	0,8,16,24,…,120	0-127
	1,9,17,25,…,121	128-255
	2,10,18,26,…,122	256-383

	…	…
	7,15,23,31,…,127	896-1023
8-15	0,8,16,24,…,120	1024-1151
	1,9,17,25,…,121	1152-1279
	2,10,18,26,…,122	1280-1407
	…	…
	7,15,23,31,…,127	1920-2047
16-23	(same as above)	2048 - 3071
etc.	…	…

Table 15-3: Example 1x1, 8-bit Weights Database format

15.3 Weights Database: 1x1 / Linear, 16-bit weights

This database format supports 1x1 convolution (linear layers) with 16-bit weights using LnsI5F10 values.

To prepare this database, we conceptually create per-output-channel packed byte arrays consisting of the weights for all input channels for that output channel, in input-channel order. Each weight is two bytes (LnsI5F10). If the number of *input* channels is not a multiple of 4, each array is zero-padded up to a multiple of 8 bytes.

The 1x1 weights (I5F10) database is then constructed by concatenating the first 8-bytes from each output channel's array (i.e., for the first 4 input channels for each output channel), in the output-channel-order illustrated in the table below. In this arrangement, the number of output channels must be padded up to a multiple of 128. (Other arrangements are possible for small output channel counts to avoid padding, but this is the most efficient arrangement for regularly-sized weights arrays)

Then the second 8-bytes from each output channel (i.e.., for the 2nd group of 4 input channels per output channel) are collected in the same swizzled order, and this pattern repeats for all groups of 4 input channels.

If there are more than 128 output channels, this entire pattern is repeated for each set of 128 output channels.

Input channels	Output Channels	Weights Database bytes
0-3	0,8,16,24,…,120	0-127
	1,9,17,25,…,121	128-255
	2,10,18,26,…,122	256-383
	…	…
	7,15,23,31,…,127	896-1023
4-7	0,8,16,24,…,120	1024-1151
	1,9,17,25,…,121	1152-1279
	2,10,18,26,…,122	1280-1407
	…	…
	7,15,23,31,…,127	1920-2047
8-11	(same as above)	2048 - 3071
etc.	…	…

Table 15-4: Example 1x1, 16-bit Weights Database format

15.4 Weights Database: 1x1 / Linear, 4-bit (compressed) weights

This section describes recommended organizations of the compressed-weights “superblocks” for deployment on Pyxis hardware. Weights compression is described in section 14 above. The basic compression-block formats support 4, 8, 16 or 32 weights/block. These may correspond to multiple input channels for a single output channel (per block), or multiple output channels for a single input channel. Based on the block size and input/output channel grouping, the guidelines here allow for the hardware to uncompress the weights and distribute them to the LLC Grid in an efficient manner.

15.4.1 Superblock organizations: Multiple input channels for one output channel

15.4.1.1 Compression Block-size 32

32-entry blocks may be organized as shown in the following table. Each superblock would comprise weights for 32 input channels for each of the 128 output channels mapped to one Grid. The specified swizzled channel order is efficient for hardware.

Superblock #	Block within superblock	Input channels	Output Channels
0	0-3	0-31	0,16,32,48
	4-7		64,80,96,112
	8-11		1,17,33,49
	12-15		65,81,97,113
	…		…
	120-123		15,31,47,63
	124-127		79,95,111,127
1	(same as above)	32-63	(same as above)
2	(same as above)	64-95	(same as above)
	etc.	etc.	…

Table 15-5: Example 1x1 compressed weights organization: block size 32 (multiple C_in per C_out)

15.4.1.2 Compression Block-size 16

16-entry blocks may be organized as shown in Table 15-6. Each superblock would comprise weights for 16 input channels for each of the 128 output channels mapped to one Grid. The specified swizzled channel order is efficient for hardware.

Superblock #	Block within superblock	Input channels	Output Channels
0	0-7	0-15	0,16,32,…,112
	8-15		1,17,33,…,113
	16-23		2,18,34,…,114
	…		…
	120-127		15,31,47,…,127
1	(same as above)	16-31	(same as above)
2	(same as above)	32-47	(same as above)
	etc.	etc.	…

Table 15-6: Example 1x1 compressed weights organization: block size 16 (multiple C_in per C_out)

15.4.1.3 Compression Block-size 8

8-entry blocks should be organized as shown in Table 15-7. Each superblock would comprise weights for 8 input channels for each of the 128 output channels mapped to one Grid. The specified swizzled channel order is efficient for hardware.

Superblock #	Block within superblock	Input channels	Output Channels
0	0-15	0-7	0,8,16,…,120
	16-31		1,9,17,…,121
	32-47		2,10,18,…,122
	…		…
	112-127		7,15,23,…,127
1	(same as above)	8-15	(same as above)
2	(same as above)	16-23	(same as above)
	etc.	etc.	…

Table 15-7: Example 1x1 compressed weights organization: block size 8 (multiple C_in per C_out)

15.4.1.4 Compression Block-size 4

4-entry blocks should be paired up to have 2 adjacent blocks (8 input channels) per output channel, as shown in the table below. For example, blocks 0&1 would together comprise weights for 8 input channels for output channel 0. Blocks 2&3 would comprise 8 input channels for output channel 1, etc. However, since each superblock has 128 blocks, each superblock only supports 64 output channels in this format, which is only half of the output channels for one Grid. So, two sequential superblocks are needed to provide the weights for 8 input channels x 128 output channels for the entire grid.

Superblock #	Blocks within superblock	Input channels (per block-pair)	Output Channels
0	0+1, 2+3,…,30+31	0-3 + 4-7	0,8,16,…,120
	32+33, 34+35,…,62+63		1,9,17,…,121
	…		…
	96+97,98+99,…,126+127		3,11,19,…,123
1	0+1, 2+3,…,30+31		4,12,20,…,124
	32+33, 34+35,…,62+63		5,13,21,…,125
	…		…
	96+97,98+99,…,126+127		7,15,23,…,127
2	0+1, 2+3,…,30+31	8-11 + 12-15	0,8,16,…,120
	…		…
	…		…
	96+97,98+99,…,126+127		3,11,19,…,123
3	0+1, 2+3,…,30+31		4,12,20,…,124
	…		…
	…		…
	96+97,98+99,…,126+127		7,15,23,…,127
4	(same as above)	16-19 + 20-23	(same as above)
5			
	etc.	etc.	…

Table 15-8: Example 1x1 compressed weights organization: block size 4 (multiple C_in per C_out)

15.4.2 Superblock organizations: Multiple output channels for one input channel

When the weights compression blocks are organized as multiple output channels per input channel, the decompressed weights need to be transposed by the hardware on horizontal path to the grid.

15.4.2.1 Compression Block-size 32

Each set of four 32-entry blocks comprise weights for 128 output channels for a single input channel.

Superblock #	Block within superblock	output channels	Input Channels
0	0-3	0-127	0
	4-7		1
	…		…
	124-127		31
1	0-3	0-127	32
	4-7		33
	…		…
	124-127		63
2	(same as above)	0-127	64-95
3	etc.	etc.	etc.

Table 15-9: Example 1x1 compressed weights organization: block size 32 (multiple C_out per C_in)

15.4.2.2 Compression Block-size 16

Each set of eight 16-entry blocks comprise weights for 128 output channels for a single input channel.

Superblock #	Block within superblock	output channels	Input Channels
0	0-7	0-127	0
	8-15		1
	…		…
	120-127		15
1	0-7	0-127	16
	8-15		17
	…		…
	120-127		31
2	(same as above)	0-127	32-47
3	etc.	etc.	etc.

Table 15-10: Example 1x1 compressed weights organization: block size 16 (multiple C_out per C_in)

15.4.2.3 Compression Block-size 8

Each set of sixteen 8-entry blocks comprise weights for 128 output channels for a single input channel.

Superblock #	Block within superblock	output channels	Input Channels
0	0-15	0-127	0
	16-31		1
	32-47		2
	…		…
	112-127		7
1	0-7	0-127	8
	8-15		9
	16-23		10
	…		…
	120-127		15
2	(same as above)	0-127	16-23
3	etc.	etc.	etc.

Table 15-11: Example 1x1 compressed weights organization: block size 8 (multiple C_out per C_in)

15.4.2.4 Compression Block-size 4

Each set of 32 four-entry blocks comprise weights for 128 output channels for a single input channel.

Superblock #	Block within superblock	output channels	Input Channels
0	0-31	0-127	0
	32-63		1
	64-95		2
	96-127		3
1	0-31	0-127	4
	32-63		5
	64-95		6
	96-127		7
2	(same as above)	0-127	8-11
3	etc.	etc.	etc.

Table 15-12: Example 1x1 compressed weights organization: block size 4 (multiple C_out per C_in)

15.5 Support of Additive Bias ("Beta")

Batch-normalization (in inference) specifies per-output-channel constants to shift and scale the results of a 1x1/Linear layer or a 3x3 convolution. The additive bias, referred to as "Beta", is added to the output of a 3x3 or 1x1 convolution. The scaling amount, referred to as "Gamma", is multiplied with the output of adding Beta, so notionally Z = Gamma(WX + Beta), where Gamma & Beta are per-output-channel constants.   It is possible to "fold" the Gamma into the weights & Beta value, so we only need to compute WX + Beta.

For Pyxis, this requires adding a Beta value at the end of a 1x1 or 3x3 convolution, into the accumulators in the grid. This appears as one extra input channel per output channel, and we can program the weights with this value. Then the UIE is programmed to effectively multiply this by '1' in the grid to effectively add the Beta value.

For 1x1 convolution, the Beta value can take advantage of mixed-precision computation, and program the Beta as an LnsI5F10 (16-bit) value.   For 3x3 convolution, mixed precision is not supported, and the 3x3 kernel weights must be LnsI4F3 (8-bit).   However, it is possible to configure a 3x3 kernel with 9 LnsI4F3 values which, when added together, can approximate the precision of a single LnsI5F10 value. This provides a software technique to support an effectively-higher-precision Beta value with the available 3x3 LnsI4F3 kernel weights.

16 System On Chip (SOC) Subsystems

16.1 PCIE Subsystem

TBD

16.2 Network On Chip (NoC)

See Pyxis NoC Specification

17 PYXIS SPECIAL PROGRAMMING TOPICS

17.1 MATRIX TRANSPOSITION

The native tensor organization for Pyxis is (channels, height, width), and many applications may require transposing or permuting the dimensions.

Any dimensions may be permuted on-the-fly, except the last (width) dimension, by instructing microcode to read data in the permuted dimension order. Therefore, these sorts of permutations do not need a special trip or operation, and can be embedded as part of other operations.

For transposing the *width* dimension, however, we need a traditional matrix transpose operation, to swap the width (columns) with height (rows). This can be done using the MatMul operation, using the "Matrix A Transpose" option (section 6.1.3.1), which delivers Matrix A data in transposed form to participate in MatMul. This can be matrix multiplied by the Identity matrix to compute A^T in the grid. Quite simply:

```
        A^T * I = A^T
```

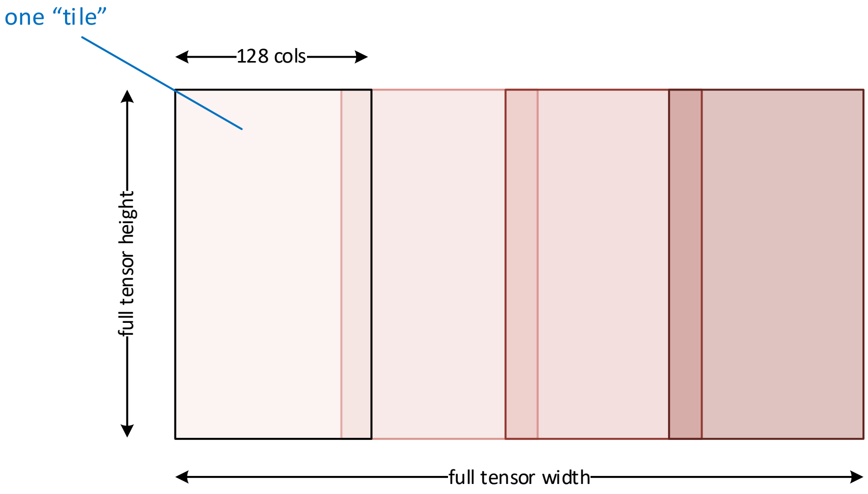
This can transpose the height/width dimensions in 128x128 tiles, with execution time approximately the same as a mem-copy of the same size. The Identity matrix, I_{128}, may be stored in AMEM as a pre-loaded constant.

Note: this requires that data passing through the grid must be unchanged by the matrix multiply operation. The linear-to-log and log-to-linear transforms must be configured to use simple, Scorpio-style mappings without mapping-correction, so that linear-to-log-to-linear is an ***identity***.

17.2 CONVOLUTION TILING FOR WIDE TENSORS

The UIE Grid can natively execute 1x1 and 3x3 convolutions for tensors with up to 128 columns, and an unlimited number of input rows.

When an input tensor is wider than 128 columns, this requires the UIE to process the tensor in multiple tiles of at most 128 columns each. However, for 3x3 (or larger-kernel) convolutions, this requires sharing of pixels across the edges of each tile, e.g., to compute a 3x3 convolution for output column 127 requires inputs from input columns (126,127,128). I.e., there must be some overlapping of the tiles.



17.2.1 Conv2D 3x3 Stride 1 – width-wise tiling

Since the AMEM is organized with 16-column *partition* granularity, we can overlap by a full partition (on each edge), in effect stepping column-wise by 7 partitions (=112 columns) per tile.

- The first tile can compute all but the last column correctly, so it must mask-out this last column when writing the data to AMEM.
- Each subsequent tile will advance by 7 partitions, and overlap by a full partition from the prior tile. This allows computing the previously-missing last column from the prior tile, but effectively recomputes the other 15 columns in the overlapped partition, so these must be masked-out when writing the data to AMEM. The last column of these tiles must also be masked off since (like the first tile) it will not produce correct results for the last column of the tile.
- The final tile must also overlap by one, but it does not need to mask off the last valid column.

The tiling progression for a tensor of width 512 columns (32 partitions) is shown below:

Tile	0	1	2	3	4
Input tensor partitions	0-7	7-14	14-21	21-28	28-31
Output column masking (dsbl)	127	112-126, 239	224-238, 351	336-350, 463	448-462
Output columns to write	0-126	127,128-238	239, 240-350	351,252-462	463,464-511
Output column masking (dsbl) relative to LLC Grid location	Ptn 7, col 15	Ptn 0, col 0-14 Ptn 7, col 15	Ptn 0, col 0-14 Ptn 7, col 15	Ptn 0, col 0-14 Ptn 7, col 15	Ptn 0, col 0-14

Table 17-1: Width-wise Tile Progression – Stride 1 Convolution

18 DEBUG AND OBSERVABILITY FACILITIES

TBD

- Barrier-Table event tracing and filtering
  - Events:
    - Initialize(barrier_id, notif_count, wait_count, initializer trip_id, timestamp)
    - Notify(barrier_id, notif_count, wait_count, notifier trip_id, timestamp)
    - Wait((barrier_id, notif_count, wait_count, waiter trip_id, timestamp)
  - Filter controls:
    - barrier_id value and mask.
    - Engine_id value and mask,
    - Trip_id value and mask.
    - Trigger events?
    - Capture rolling N prior cycles or stop when full.
- Fabric Tx and Fabric Rx Job table observability and event counters
- Fabric event tracing on a per-page basis
  - For Rx this would capture received pages,
  - For Tx this would capture transmitted pages at the time of VOQ Enqueue.
  - Information captured would include fields from the page header: (rjid, tjid, epoch, seq_num, page size, retry bit), with a timestamp.
  - Uses a fixed-size trace buffer (dedicated SRAMs within Fabric Tx or Rx)
- Per-UIE event counters
- Miscellaneous counters:

18.1 NOTES ON BARRIER TRACING FACILITIES FROM GRAPH-TO-CHIP GROUP DISCUSSION, 2024/09/26:

- We can afford 20-bit trip ID per engine, prepended with an enumeration of the engine-ID (there are ~35 barrier-signaling engines per cluster currently).
  - The trip ID is configured by s/w in each descriptor.
  - hardware will add the engine-ID to the 20b trip ID when tracing.

- Tracing can be triggered to start/stop when a trip does its "wait()" query to the barrier table.
- Tracing is filtered with three trace-control registers.
    - Start - triggers tracing to begin, including this event
    - Stop - triggers tracing to stop, including the stop-event in the trace
    - Trace (filter) control - limits which events are traced during the period between start & stop. (This one seems to me to be nice to have to reduce the volume/clutter of tracing info)
- Each tracing control register includes configurable mask and value registers,
    - Tracing control is based on the following tuple: {engine_id, trip_id, barrier_id}
    - The event (start/stop/trace) is valid if the (tuple & mask) == (value & mask)
    - Tracing is for any barrier table event, i.e., barrier_init(), notify(), or wait()
- The fields traced include (engine_id, trip_id, barrier_id, barrier state values, event_id, timestamp)
    - Event_id is one of (setup, notify, wait)
- Timestamps can have single-cycle granularity, and 32-bit precision.
    - when timestamp rolls over, the tracing hardware will (if tracing is active at that point) insert a "time rollover" event to serve as a marker in the trace, so time can be reconstructed later
- Tracing is written to a trace-buffer in hardware, which must be offloaded to a larger memory buffer. Details of this (including long term buffer rollover management) are TBD.

19 Clock Domains

See Pyxis clock specification

20 VOLTAGE DOMAINS

Voltage Domain Name	Nom Voltage	Associated Ground	Voltage	Description	Package Notes	References
VDD	0.75	VSS	0	Core side		On chip core voltage rail
VDD_GRID	0.75	VSS	0	LLC Grid		On chip compute voltage rail
PCIE_AVDD	0.8	AGND	0	PCIe SerDes		
PCIE_AVDDH	0.9	AGND	0	PCIe SerDes		
PCIEPLL_VDD1P8	1.8	PCIEPLL_VSS	0	PCIe-Core-PLL		
PCIE_PLL_AVDD	0.8	VSS	0	PCIe SerDes PLL - Int		
FAB_AVDD	0.8	AGND	0	112G SerDes		
FAB_AVDDH	0.9	AGND	0	112G SerDes		
FABPLL_VDD1P8	1.8	FABPLL_VSS	0	Fabric Core SerDes PLL		
FABSER_PLL_AVDD	0.8	VSS	0	112G SerDes PLL - Int.		
VDDIO_L	0.4	VSS	0	HBM	LVSTL	
VDDIO	1.1	VSS	0	HBM		
HBMPLL_VDD1P8	1.8	HBMPLL_VSS	0	HBM Core PLL		
HBM_PLL_AVDD	0.8	VSS	0	HBM PLL - Int		
CORE_PLL1P8	1.8	COREPLL_VSS	0	Core PLL		
CORE_PLL_AVDD	0.8	VSS	0	Core PLL		
PADS_VDDO	1.2/1.5/1.8	PADS0_VSSO	0	PADS0		
AVDD_THERM_1P8	1.8	AVSS_THERM	0	Thermal sensor		
VDDQ_EFUSE	1.8	VSS	0	eFUSE		
VDDQ_EFUSE_1	1.8	VSS	0	eFUSE		
EDM_VDD	0.75	VSS	0	Die Edge Monitor (BRCM only)		